Physical Synthesis for Advanced Neural Network Processors

(Invited Paper)

Zhuolun He
Chinese University of Hong Kong

Peiyu Liao
Chinese University of Hong Kong

Siting Liu
Chinese University of Hong Kong

Yuzhe Ma
Chinese University of Hong Kong

Yibo Lin
Peking University

Bei Yu
Chinese University of Hong Kong

Abstract

The remarkable breakthroughs in deep learning have led to a dramatic thirst for computational resources to tackle interesting real-world problems. Various neural network processors have been proposed for the purpose, yet, far fewer discussions have been made on the physical synthesis for such specialized processors, especially in advanced technology nodes. In this paper, we review several physical synthesis techniques for advanced neural network processors. We especially argue that datapath design is an essential methodology in the above procedures due to the organized computational graph of neural networks. As a case study, we investigate a wafer-scale deep learning accelerator placement problem in detail.

1 Introduction

Deep learning has emerged as one of the most important workloads due to its extraordinary performance gains in a number of disciplines. Despite of that, how to efficiently execute deep neural network (DNN) models remains a crucial concern since the beginning of deep learning resurgence. Exacerbating the problem is the progressively sophisticated network architectures, as well as the irregularity due to network pruning and compression, which unarguably impedes the deployment of modern DNNs onto devices. The above challenge intrinsically highlights the demand for customized physical synthesis methodologies, since the quality of physical synthesis directly determine the performance of neural network processors.

As neural network processors getting increasingly hierarchical and structural, dataflow optimization becomes essential to boost the system capabilities. Dataflow optimization schedules operation by data availability, which exposes opportunity for parallelism and data reuse. To illustrate some achievements in dataflow optimization, Zhang *et al.* [1] quantitatively analyze the computing throughput and required memory bandwidth using various conventional optimization techniques, such as loop tiling and transformation, and then apply roofline model to balance the resource; Alwani *et al.* [2] fuse the processing of adjacent convolutional layers with the data on-chip and use a pyramid-shaped multi-layer sliding window to minimize off-chip transfer; Ma *et al.* [3] present an in-depth analysis of convolution loop acceleration strategy by numerically characterizing the loop optimization techniques and then use multiple optimization algorithms to optimize the loop operation and the dataflow. Zhang *et al.* [4] propose a fine-grained layer-based pipeline architecture and a column-based cache scheme for higher throughput, lower pipeline latency, and smaller on-chip memory consumption; Wei *et al.* [5] provide an analytical model for performance and resource utilization and develop an automatic design space exploration framework to generate a convolutional neural network (CNN) implementation using systolic arrays. Sun *et al.* [6] improve the power performance by combining layer fusion and dataflow optimization techniques.

In addition to optimizing the dataflow itself, dataflow regularity also give rise to new methodologies in physical synthesis, the critical stage in modern very-large-scale integrated (VLSI) circuit design flow. In VLSI design, datapaths are characterised by high degree of bit-wise parallelism, which are placed with high regularity and compactness to achieve high performance [7]. In this sense, datapath-driven placement approaches have been attracting researchers' attention.

The rest of our paper is organized as follows. Section 2 provides preliminaries about neural network processors and the physical design flow. Section 3 conducts a comprehensive review of datapath-driven placement methodologies. Section 4 investigates a wafer-scale deep learning accelerator placement problem. Section 5 discusses some advanced technologies for neural network processors, followed by conclusion in Section 6.

2 Preliminaries

2.1 Neural Network Processor

In deep neural networks, executing inference such as convolution performs a very large amount of multiply-accumulate (MAC) operations, since a single convolution comprises of iterating over every channel and every pixel for each given input, typically with billions or even trillions of iterations. Besides, the model itself must be executed once for each new input.

While central processing units (CPUs) are effective at processing highly serialized instruction streams, machine learning workloads tend to be highly parallelizable, which is a good fit for graphics processing units (GPUs). Moreover, neural processing units (NPUs) benefit from vastly simpler logic because their workloads tend to exhibit high regularity in the computational patterns of deep neural networks. For the above reasons, many customized dedicated neural processors have been developed.

An NPU is a well-partitioned circuit that includes all the control and arithmetic logic components necessary to execute machine learning algorithms. NPUs are designed to accelerate the performance of common machine learning tasks such as image classification, machine translation, object detection, and various other predictive

ASPDAC '21, January 18–21, 2021, Tokyo, Japan

ACM ISBN 978-1-4503-7999-1/21/01…$15.00
https://doi.org/10.1145/3394885.3431625

models. NPUs might be parts of a large SoC, a plurality of NPUs may be instantiated on a single chip, or they may be part of a dedicated neural-network accelerator.

2.2 Physical Design Flow

Physical design is based on a netlist synthesized from an RTL design to a gate-level description. Generally, the physical design flow is divided into several steps: floorplanning, partitioning, placement, clock-tree synthesis, routing, physical verification, and layout post-processing with mask data generation. Floorplanning, placement, and routing are the most essential steps in physical design.

Floorplanning determines geometric relations between modules to optimize some objectives such as area, wirelength, and some desired performance. A bad floorplan leads to wastage of die area and routing congestion. As for the circuit performance, lower area is usually desired, as it indicates shorter interconnect distances, fewer routing resources used, faster end-to-end signal paths, and even faster and more consistent place and route time. However, routing may be more difficult with fewer assigned routing resources. In general, floorplanning benefits from hierarchy information like datapaths.

Placement is another crucial stage in physical design. A poor placement not only affects the chip performance but also makes it non-manufacturable with an excessive wirelength beyond available routing resources. Therefore, placement always processes with several objectives to ensure that a circuit meets its performance demands. Routing builds on placement and it assigns wires to properly connect the placed components under all design rules for the integrated circuits. Together, the placement and routing steps of integrated circuits design are known as place and route (PnR).

3 Datapath Driven Placement: A Survey

We present in this section some useful techniques in datapath driven placement.

3.1 Placement with Datapath Constraints

The idea of datapath driven placement can date back to no later than the work [8] published in the year of 1990, which considers automatic generation of bit-sliced datapaths in high performance DSP circuits. The datapath consists of multi-bit operators called functional building blocks (FBBs), such as adders or registers. The proposed linear placement tool generates a linear ordering of the FBBs to minimize the layout area. In their work, the ordering solution space is represented as an acyclic directed graph, so that the orderings can be searched with the A* algorithm. The algorithm achieves good performance and runs much faster than metaheuristics (e.g. simulated annealing), and the authors emphasized that the algorithm is flexible in adapting various cost functions.

Later on, datapath driven standard cell placement was proposed [9]. In this work, strongly connected subcircuits (i.e., cones) are extracted by a breadth-first algorithm augmented with heuristic rules. These cones are treated as soft macro cells, and are placed by a macro-cell placement algorithm to reduce the intercone wiring length. Macros are converted back to cells by a mapping subsystem to preserve the topological relationship between them.

It is argued that if the datapath is generated separately and simply merged with netlists of other parts, the placement tool has little control of the exact location where a cell might be placed [10]. In this way the regularity information will be lost. Given that, the

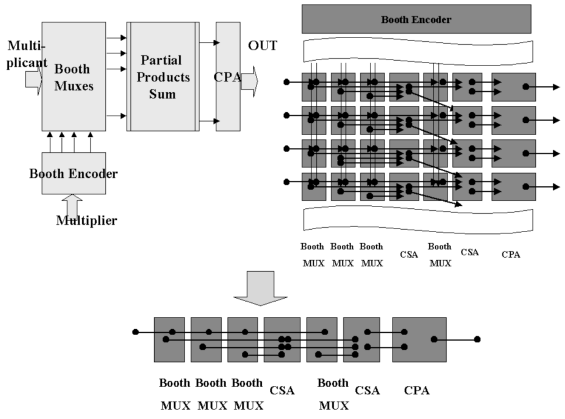
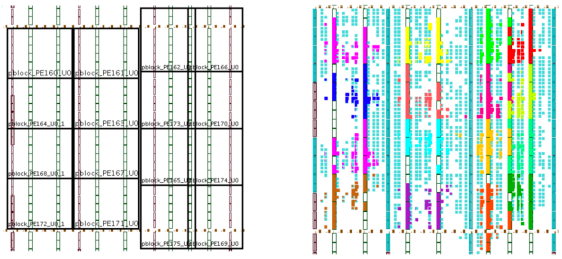


Figure 1: Abstract physical model is a bit-sliced abstraction of a datapath circuit. The figure illustrates the APM of a booth multiplier [10].

authors proposed an abstract physical model (APM), a bit-sliced abstraction of a datapath circuit. Figure 1 demonstrates the APM of a booth multiplier (adopted from [10]). The APM is compiled from HDL, and the blocks in APM are placed abutted to each other. Since the linear placement problem is NP-hard, they proposed a two step heuristic that first determines an initial ordering by a quadratic optimization procedure, and then a sliding window optimization is performed to solve wirelength and congestion violations.

Datapath has been considered in detailed placement [11], where a modified O-tree [12] based placer is able to place components on reflection lines while obeying design rules. It is also considered in physical design inside SOC [13, 14], and for parallel multiplier design [15]. A lot more works for datapath driven general ASIC design have been presented. In [16], datapath clusters are placed with constraints that 1) the relative locations of the clusters should follow the dataflow order and 2) the relative orientations of the clusters should follow the bit order, and the same bit order should be preserved throughout the dataflow. The authors name it '1.5 dimensional placement' and proposed to solve it by linear placement heuristics similar to [10]. A sigmoid-function-based density model is proposed [17] for separate optimization in horizontal and vertical directions. Blocks of each functional stage align vertically due to the regularity in datapaths, which reduces variables in the optimization problem. In [7], datapath macros are placed with other random blocks by an analytical placement algorithm, while the relative location of bit-slices can be adjusted inside the datapath macros to optimize total wirelength. Results have shown that these techniques produce better results in wirelength (HPWL, StWL) and/or routability.

Systolic arrays are a popular choice to support neural network computations due to their regular topology and simple interconnections. Despite the layout-friendly structures, it was reported that current FPGA CAD tools are unable to synthesize systolic arrays in high quality [18]. One of the key reasons might be that DSPs are distributed into columns over the whole FPGA chip. Meanwhile, there are around 15× more intra-PE nets than inter-PE nets, optimizing the length of which result in a distorted layout. Given the above, the authors propose to perform floorplanning and set placement



(a) (b)

Figure 2: PE placement with floorplan constraints. Floorplanning re-organizes PEs more regularly [18].

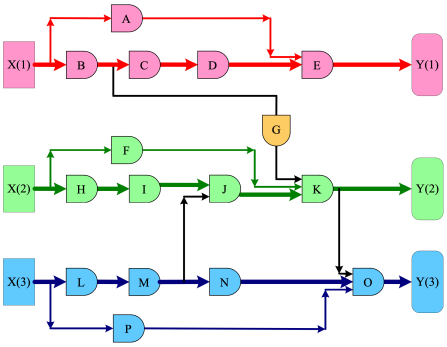


Figure 3: The problem of datapath main frame identification can be transformed to a network flow problem [24].

constraints to restrict fixed locations for PEs. To enable the topology-aware floorplanning, the region in which the PEs are placed should provide sufficient hardware resources, and should be as close as possible to the used I/O banks. Then the PEs are mapped to the available DSP columns by enumeration. Figure 2 shows the placement result of PEs with floorplan constraints (adopted from [18]). The authors reported 1.29× frequency improvement and 1.5TOP/S in deploying a VGG model onto the Xilinx KCU1500 platform.

Benchmarks are released [19, 20] to evaluate the performance of the placers.

3.2 Methods for Regularity Extraction

Along with the methodologies in datapath driven placement, various approaches for regularity extraction have been proposed. We review some of the representative approaches in this section.

Intuitively, as described in [21], consider cells associated with the same bit-*slice* are lined up horizontally, and the cells of the same type occurring at similar places are stacked alongside forming *stages*. The circuit is thus fitted onto a matrix of rectangular buckets that yields maximum density cell placement. In addition to the above *geometric regularity*, the *interconnect regularity* indicates that almost all nets are contained either within one slice or within one stage. In [21], a local regularity metric is defined by interpreting the distribution of the number of pins, and a regularity extraction algorithm is proposed by expanding search-waves through the network, stage by stage according to the regularity metric.

A signature-based regularity extraction algorithm is proposed later [22]. The *signature* of a random instance is dictated by its master cell and its connectivity to datapath instances. Then a connectivity cost function is defined based on some objective, such as the vertical distance between two pins. The random instances are sorted based on the signatures and are partitioned into blocks with the same signature. Finally, the regular functions are generated taking the connectivity cost into consideration. The authors also propose a relaxed function-based signature.

Covering a circuit by templates is another line of research. Besides assuming a library of provided templates, Chowdhary *et al.* [23] present an approach to automatically generate all possible templates for the input circuit. Despite the inherent difficulty in template generation (which is similar to enumerating isomorphic subgraphs), they propose to extract only maximum degree of regularity (assumption 1 in their paper), and to assign incoming edges a unique index (assumption 2 in their paper). With the two assumptions, the number of possible tree templates is reduced to within V^2 from 2^V. The template generation algorithm is then extended to generate multi-output function. The authors also propose two heuristics to effectively cover the graph by templates, including *largest-fit-first* that selects the template with the maximum area, and *most-frequent-fit-first* that selects the template with the maximum number of subgraphs.

The regularity extraction problem can be converted to a network-flow problem. In [24], *datapath main frame* (DMF) is defined as a set of n disjoint paths from input to output such that the number of datapath gates on these paths is maximized. To identify DMF, an optimal network-flow based algorithm is presented. Basically, capacities U and costs C are assigned to the network graph, where capacity is 1 for all the nodes and edges, and the cost for nodes is a constant negative number, and 0 for the edges. The min-cost max-flow algorithm will be applied to the flow network, which guarantees the optimality with polynomial runtime. Figure 3 demonstrates the data main frame identification problem (adopted from [24]).

Properties of bit-slices include 1) small area variance (similar), 2) large area mean (long), and 3) minimized overlaps [25]. In the same paper [25], the authors propose a two-stage method that first optimize 1) and 2) with a balanced bipartite edge-cover algorithm, and minimize 3) with simulated annealing. Since every bit-slice path is a bit line connecting I/O vectors, a datapath is modeled as a bipartite graph, where the vertices correspond to either larger I/O vector or narrow I/O vector, and the edges correspond to the bit lines. Therefore, the problem of extracting feasible set of bit-slice paths is transformed to the problem of covering vertices in the bipartite graph.

It is worth mentioning that these techniques have also been utilized in logic synthesis [26–28]. However, it was also pointed out [29] that extraction of regularity from synthesized netlists is difficult and requires counterproductive simplifications to the synthesis process.

3.3 Machine Learning Techniques

Recently, machine learning techniques are utilized for datapath extraction [30]. Specifically, graph-based (e.g., automorphism) and physical features (e.g., cell area) are analyzed and extracted from the netlist. These features are fed to support vector machine (SVM) and neural networks (NNs) to classify and evaluate datapath patterns. Both SVM and NNs are to maximize the evaluation accuracies of datapath and non-datapath patterns, which are defined as the rate of correctly detected datapath/non-datapath patterns over the total

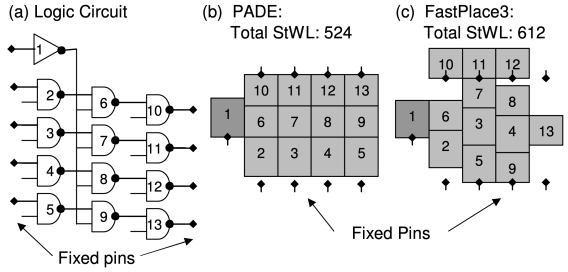


Figure 4: PADE effectively handles datapath in placement. Adopted from [30].

number of corresponding structures. The proposed placer, PADE, has demonstrated reasonable improvement in wirelength. Figure 4 shows placement result by PADE that effectively handles the datapath (adopted from [30]). The authors also proposed a unified placement flow [31, 32] that handles both random logic and datapath standard cells on top of a force-directed placer. Several techniques for structural-aware placement, such as skewed weighting for net alignment and bit-stack aligned cell swapping, are proposed and discussed in the paper.

4 Datapath Driven Floorplan: A Case Study

In this section, we focus on a wafer-scale deep learning accelerator placement problem introduced in ISPD2020 contest [33] as a case study. We argue that the problem is more like a typical floorplanning problem, in which there are tens or hundreds of macro blocks, and the block shapes are flexible (we actually need to determine the shapes in the solution). Therefore, the contest has put forward a floorplanning problem for neural network optimization on a wafer-scale computing engine. Considering that neural networks are a stack of single layers where each performs a single function, naturally an AI compiler decomposes such neural network computation into a stack of single units called *kernels*. To solve this floorplanning problem, we are required to assign each compute kernel a two-dimensional position on the wafer without overlap and utilize computing resources as much as possible.

4.1 Kernel Description

Formally, a *kernel* is a parametric program that performs specific tensor operations. For instance, a convolution kernel performs several kinds of convolution operations. To better describe how a kernel is customized on the wafer, its arguments are classified into two main groups. *Formal arguments* specify the exact shapes of tensor operations to be performed. They are uniquely determined by the network architecture, so we consider them to be fixed in our optimization. *Execution arguments* describe how the operation is parallelized across tiles. We are required to find the optimal combination of execution arguments to obtain a maximum utilization of computation resources.

Still, we take a convolution kernel as an example. A normal convolution kernel contains 8 formal arguments, represented by a tuple (H, W, R, S, C, K, T, U). In detail, (H, W) specify the height and width of input image respectively; (C, K) represent the number of channels of input and output image; (R, S) describe the kernel size in two dimensions; and (T, U), similarly, are strides in two dimensions, respectively. They are fixed arguments or intrinsic attributes of a convolution kernel. On the other hand, four execution arguments (h', w', c', k') are free to be specified to maximize resource utilization.

4.2 Problem Formulation

Since we are provided with different types of kernels, each type of kernels have a function to evaluate performance. The performance of a convolution kernel is evaluated by a 4-tuple (h, w, t, m) called *performance cuboid*, where h, w, t, m represent height, width, time and memory this kernel requires, respectively.

The ISPD2020 contest benchmarks provide two main categories of kernels, convolution kernels and block kernels. The performance evaluator of a convolution kernel K (denoted type `conv` hereafter) with formal arguments $\mu_K = (H, W, R, S, C, K, T, U)$ is defined as a function convperf that maps execution arguments $x_K = (h', w', c', k')$ to a resource cuboid $r_K = (h, w, t, m)$, where

$$\begin{aligned} h &= h'w'(c'+1), \qquad w = 3k', \\ t &= \left\lceil \frac{H}{h'} \right\rceil \cdot \left\lceil \frac{W}{w'} \right\rceil \cdot \left\lceil \frac{C}{c'} \right\rceil \cdot \left\lceil \frac{K}{k'} \right\rceil \cdot \frac{RS}{T^2}, \\ m &= RS\frac{C}{c'}\frac{K}{k'} + \frac{W+S-1}{w'}\frac{H+R-1}{h'}\frac{K}{k'}. \end{aligned} \tag{1}$$

Notation $\lceil \cdot \rceil$ represents math `ceil` function. Each of block kernel consists of several `conv` kernels. For example, a block kernel K, with formal arguments $\mu_K = (H, W, F)$ and execution arguments $x_K = (h', w', c'_1, \cdots, c'_n, k'_1, \cdots, k'_n)$, comprises of n `conv` kernels $K_i (i = 1, \cdots, n)$. Specifically, a convolution kernel K_i contains $x_{K_i} = (h', w', c'_i, k'_i)$ as its execution arguments. The formal argument tuple μ_{K_i} of `conv` kernel K_i is determined by $\mu_K = (H, W, F)$ and specific attributes of the current block type in detail. The ISPD2020 contest benchmarks provide us with two types of block kernels, `dblock` and `cblock`.

- `dblock`. A `dblock` kernel consists of 3 different `conv` kernels K_1, K_2, K_3. Corresponding formal arguments are
$$\begin{aligned} \mu_{K_1} &= (H, W, 1, 1, \quad F, F/4, 1, 1), \\ \mu_{K_2} &= (H, W, 3, 3, F/4, F/4, 1, 1), \\ \mu_{K_3} &= (H, W, 1, 1, F/4, \quad F, 1, 1). \end{aligned}$$
- `cblock`. A `cblock` kernel consists of 4 different `conv` kernels K_1, K_2, K_3, K_4. Similarly the corresponding formal arguments are
$$\begin{aligned} \mu_{K_1} &= (\quad H, \quad W, 1, 1, F/2, F/4, 1, 1), \\ \mu_{K_2} &= (\quad H, \quad W, 3, 3, F/4, F/4, 2, 2), \\ \mu_{K_3} &= (H/2, W/2, 1, 1, F/4, \quad F, 1, 1), \\ \mu_{K_4} &= (\quad H, \quad W, 1, 1, F/2, \quad F, 2, 2). \end{aligned}$$

Formal arguments of each component `conv` kernel are uniquely determined by $\mu_K = (H, W, F)$. Similar to the performance evaluator of `conv` kernels, the performance of a block kernel K consisting of n different `conv` kernels $K_i (i = 1, \cdots, n)$ can be evaluated by

$$\text{blockperf}(\mu_K; x_K) = r_K = (h, w, t, m), \tag{2}$$

where components of 4-tuple r_K is formulated as

$$\begin{aligned} h &= \max_{1 \le i \le n} h_i, \quad w = \sum_{i=1}^{n} w_i, \\ t &= \max_{1 \le i \le n} t_i, \quad m = \max_{1 \le i \le n} m_i; \end{aligned} \tag{3}$$

$$(h_i, w_i, t_i, m_i) := \text{convperf}(\mu_{K_i}; x_{K_i}). \tag{4}$$

To formulate the optimization problem, objective and constraints must be reasonably specified. Suppose that we are provided with kernel library $\mathcal{K} = \{K_1, K_2, \cdots, K_n\}$, where K_i is a kernel of type `conv`, `dblock` or `cblock`. A floorplan solution assigns location vectors $\boldsymbol{x}, \boldsymbol{y}$ to these kernels. Then our final objective function contains three parts:

- **Maximum execution time** of any placed kernel. Since any kernel K in the kernel library $\mathcal{K}$ will be evaluated and return a resource cuboid r_K containing execution time t_K, this term is normally formulated as $\max_{1 \le i \le n} t_{K_i}$.
- **Total L_1 wirelength**. It is determined by a floorplan $\boldsymbol{x}, \boldsymbol{y}$ of kernels $\mathcal{K}$. We formulate total wirelength as such a function $W(\boldsymbol{x}, \boldsymbol{y})$ of location vectors.
- **Protocol differences** P between connected kernels.

Suppose that there is a directed edge from kernel K_1 to kernel K_2, whose execution arguments are represented by

$$x_{K_1} = (h'^{(1)}, w'^{(1)}, c_1'^{(1)}, \cdots, c_m'^{(1)}, k_1'^{(1)}, \cdots, k_m'^{(1)}),$$
$$x_{K_2} = (h'^{(2)}, w'^{(2)}, c_1'^{(2)}, \cdots, c_n'^{(2)}, k_1'^{(2)}, \cdots, k_n'^{(2)}).$$

Note that when m or n is exactly 1, the kernel type is `conv`. Then the protocol number of this edge is defined as

$$P = 3 - \delta_{h'^{(1)}, h'^{(2)}} - \delta_{w'^{(1)}, w'^{(2)}} - \delta_{p^{(1)}, p^{(2)}}, \tag{5}$$

where $p_1 = \min\{c_m'^{(1)}, k_m'^{(1)}\}$, $p_2 = \min\{c_1'^{(2)}, k_1'^{(2)}\}$ are split numbers in two dimensions, and δ_{ij} is *Kronecker delta* which takes value 1 if the two subscripts are equal and 0 otherwise. The protocol number is uniquely determined by the kernel graph G.

The objective we are about to optimize is the weighted sum of the three parts mentioned above.

$$J = \max_{1 \le i \le n} t_{K_i} + \lambda_1 W(\boldsymbol{x}, \boldsymbol{y}) + \lambda_2 P(G), \tag{6}$$

where λ_1 and λ_2 are hyper-parameters provided by benchmarks. J should be optimized under specific constraints, 1) all kernels must fit within the fabric area; and 2) kernels must not overlap. The constraints are straight-forward for industrial practitioners to follow. They are discrete enough to make our optimization problem difficult to solve.

4.3 Algorithm

Considering that a neural network usually is a stack of layers, we can extract a clear datapath to describe how data is processed during a forward pass. That inspires us that it is possible to arrange the floorplan according to the datapath, since we have great flexibility to control the shape of each module.

4.3.1 *One-Row Floorplan*

Many cases in ISPD2020 contest benchmarks are a chain-like connection of kernels. Intuitively, if the total number of kernels is not very large, we can always place them horizontally one by one following the dataflow order, shown in Figure 5(a).

Take a look at the performance function of a `conv` kernel. If we ignore the math `ceil` function, it can be easily derived that execution time t is inversely proportional to kernel area, approximately. However, three unroll factors h', w', k' are capable of affecting the value of height h, while w is only affected by c'. Therefore, it is more reasonable to pinch each module to a thin and tall one, so that the height can shoulder more unrolling burden.

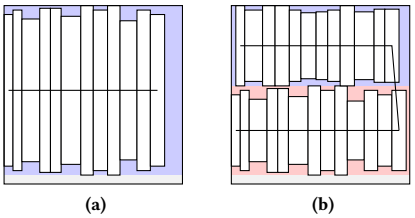


Figure 5: (a) One-row floorplan; (b) Multi-row Mamba floorplan.

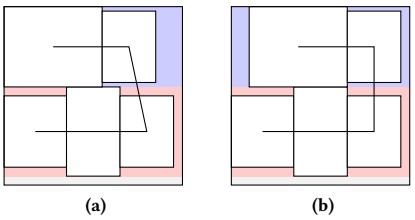


Figure 6: (a) Mamba floorplan; (b) Horizontally compacted Mamba floorplan.

This one-row strategy should work pretty fine especially when the weight of wirelength λ_1 is significantly considerable, since the total Manhattan distance between connected kernels could be tremendously optimized to an observable value.

4.3.2 *Mamba Floorplan*

Unfortunately, the one-row floorplan described in the subsection 4.3.1 does not always work. In fact, in most of large cases it will definitely fail, considering that the width allowed for floorplan is at most 633 while the minimum width of a block is at least 3. Then it is natural to extend our solution to multi-row floorplan. Specifically, to reduce the total wirelength, it should be better to connect the rows head-to-head and tail-to-tail, and thus we call such floorplan strategy *Mamba* floorplan, illustrated in Figure 5(b).

4.3.3 *Floorplan Compacting*

From Figure 5(b) we observe that our mamba floorplan strategy can be further compressed. Once the total width of kernels placed in a row is not strictly equal to the maximum wirelength, we can always slide them horizontally preserving the kernel order. Figure 6 illustrates our compacting strategy. In Figure 6(a), the blue and red colored regions are the first and second row respectively, and the black lines connecting five kernels indicate the datapath.

We observe that neither the two kernels in the first row nor the three kernels in the second row are able to fill the width of corresponding row without any gap. Therefore we can horizontally move kernels preserving order to further reduce the total wirelength, shown in Figure 6(b). The abstracted wire connecting the two rightmost kernels are strictly vertical so that the total wirelength is guaranteed to be less than that in Figure 6(a).

In general cases where we apply multi-row mamba floorplan, the total number of rows might be larger. Therefore our row-shift

will be much more complicated. Assume that we have n rows in total. For the i-th row, we have three variables a_i, b_i, r_i determining its characteristics, where a_i is the *center* horizontal coordinates of the leftmost kernel, b_i is the *center* horizontal coordinates of the rightmost kernel, and r_i is the total width of all consecutive kernels placed in this row. The goal of compacting is to find $x_1, \cdots, x_n$ where x_i represents the distance between the left boundary of region and the leftmost kernel of the i-th row. This problem can be formulated as a linear programming.

$$\begin{aligned} \min \quad & \sum_{i=1}^{n-1} |x_i - x_{i+1} + t_i| \\ \text{s.t.} \quad & x_i + r_i \le w, \quad i = 1, \cdots, n, \\ & x_i \ge 0, \quad i = 1, \cdots, n, \\ & t_{2i} = a_{2i} - a_{2i+1}, \quad i = 1, \cdots, \left\lfloor \frac{n-1}{2} \right\rfloor, \\ & t_{2i-1} = b_{2i-1} - b_{2i}, \quad i = 1, \cdots, \left\lfloor \frac{n}{2} \right\rfloor, \end{aligned} \tag{7}$$

where w represents the total width of the floorplan region. Any solver aiming at solving linear programming can be applied to Problem (7). Empirically the total number of rows is always a small number (less than 6) to make the solving efficient.

4.3.4 Execution Arguments Selection

We have already determined the floorplan strategy, so the only thing left is to decide the execution arguments of the kernels. Generally, execution arguments of kernels are not independent to each other, however, we tend to consider them separately to reduce the computational complexity.

For a `conv` kernel, we simply perform a grid search in all possible combinations of h', w', c', k'. Note that lots of solutions can be trivially pruned. For example, $c' = \lceil C/2 \rceil$ and $c' = C - 1$ yield the same unroll strategy on C because $\left\lceil \frac{C}{\lceil C/2 \rceil} \right\rceil = \left\lceil \frac{C}{C-1} \right\rceil$, while the latter one consumes far more processing tiles when C is large. Specifically, we have the following results.

Lemma 1. *For a positive integer N, there are $\left\lfloor \sqrt{N} \right\rfloor$ different integer pairs $(x, \lceil \frac{N}{x} \rceil)$ such that $1 \le x \le \lceil \frac{N}{x} \rceil$ and $\lceil \frac{N}{x} \rceil \ge 1$.*

Theorem 1. *For any positive integer N, there are at most $2\left\lfloor \sqrt{N} \right\rfloor$ different integer pairs $(x, \lceil \frac{N}{x} \rceil)$ such that $x \ge 1$ and $\lceil \frac{N}{x} \rceil \ge 1$.*

Proof. From Lemma 1 we know that $(1, \lceil N \rceil)$, $(2, \lceil N/2 \rceil)$, $\cdots$, $\left(\left\lfloor \sqrt{N} \right\rfloor, \left\lceil N / \left\lfloor \sqrt{N} \right\rfloor \right\rceil\right)$ are legal pairs. Take arbitrary positive $x \le N$, and let $y = \lceil N/x \rceil$. It is obvious that $x = \lceil N/y \rceil$ is also true. Therefore, if (x, y) is a legal pair such that $x > y$ according to theorem description, then (y, x) is also legal and thus $y \in \{1, \cdots, \left\lfloor \sqrt{N} \right\rfloor\}$. If N is a square number, $(\sqrt{N}, \sqrt{N})$ is legal, then the total number is $2\left\lfloor \sqrt{N} \right\rfloor - 1$, otherwise it is $2\left\lfloor \sqrt{N} \right\rfloor$. □

Theorem 1 indicates that, we only need to perform a grid search in $\mathcal{O}(\sqrt{HWCK})$ time complexity, providing that if $\lceil H/h'_1 \rceil = \lceil H/h'_2 \rceil$ we always prefer the smaller one $\min\{h'_1, h'_2\}$ to make the `conv` kernel compact.

For a `dblock` or `cblock` kernel, we balance the heights of its internal `conv` kernels first, otherwise the resulted empty space due to the height difference will be wasted. Take a `dblock` kernel as an example. It has three `conv` kernels $K_i (i = 1, 2, 3)$ with execution arguments $x_{K_i} = (h', w', c'_i, k'_i)$. To balance the height of kernels we have $c'_1 = c'_2 = c'_3$. Similarly, to balance the runtime we have $k'_1 : k'_2 : k'_3 = 4 : 9 : 4$, based on the following results.

Theorem 2. *For a `dblock` or `cblock` kernel K, its execution arguments $(h', w', c'_1, \cdots, c'_n, k'_1, \cdots, k'_n)$ is no better than the modified one $(h', w', c', \cdots, c', k'_1, \cdots, k'_n)$ where $c = \max_i\{c'_i\}$ with respect to height, width, time and memory.*

Proof. The proof is straight-forward, since runtime t and memory m is non-increasing with respect to c', and width w is irrelevant to c'. Height is the maximum of three `conv` kernels, from $\max_{i \in \{1, \cdots, n\}}\{h'w'(c'_i + 1)\} = h'w'(c' + 1)$, we know that the height to this `dblock` remains unchanged. Therefore the performance of execution arguments $(h', w', c', \cdots, c', k'_1, \cdots, k'_n)$ is no worse than the original one. It applies to the `dblock` kernel when $n = 3$, and `cblock` kernel when $n = 4$. □

The result related to k' is not easy to derive because of the `ceil` functions. From the argument selecting strategy for `conv` kernel we tend to select those numbers that *roughly* divide the corresponding formal arguments as execution arguments, *e.g.* h' such that H/h' is close to $\lceil H/h' \rceil$. Therefore, it should be reasonable to approximate $\lceil H/h' \rceil$ by H/h'. We define the approximated runtime $\tilde{t}$ of a `conv` kernel K with formal arguments $\mu_K = (H, W, R, S, C, K, T, U)$ and execution arguments $x_K = (h', w', c', k')$ as follows,

$$\tilde{t} := \frac{HWCKRS}{h'w'c'k'T^2}. \tag{8}$$

For `block` kernels, $\tilde{t}$ is defined as the maximum of that of its `conv` kernels. We have the following result.

Theorem 3. *Given a `dblock` kernel K with execution arguments $x_K = (h', w', c', c', c', k'_1, k'_2, k'_3)$, let $k' = \min\{k'_1, k'_3\}$. Then x_K is no better than $(h', w', c', c', c', \lceil \frac{4}{9}k'_2 \rceil, k'_2, \lceil \frac{4}{9}k'_2 \rceil)$ if $\frac{4}{9}k'_2 \le k'$, otherwise no better than $(h', w', c', c', c', k', \lceil \frac{9}{4}k' \rceil, k')$ with respect to height, width and approximated time defined in Equation (8).*

Proof. The height is irrelevant to k' so it remains unchanged. Consider the approximated time.

$$\tilde{t}_1 = \frac{HWF^2}{4h'w'c'k'_1}, \quad \tilde{t}_2 = \frac{9HWF^2}{16h'w'c'k'_2}, \quad \tilde{t}_3 = \frac{HWF^2}{4h'w'c'k'_3}.$$

Suppose that $\frac{4}{9}k'_2 \le k'$, then $\tilde{t}_2 \ge \max\{\tilde{t}_1, \tilde{t}_3\}$ and thus $\tilde{t}$ is not determined by k'_1, k'_3. We simply decrease k'_1 and k'_3 such that they are still no less than $\frac{4}{9}k'_2$, then $\tilde{t}$ of this kernel must remain unchanged and additionally total width is reduced.

If $\frac{4}{9}k'_2 > k'$, then $\tilde{t} = \max\{\tilde{t}_1, \tilde{t}_3\}$ and thus $\tilde{t}$ is determined by k'_1, k'_3. We decrease k'_1 and k'_3 to k', and let k'_2 be the minimum number such that $\frac{4}{9}k'_2 \ge k'$ (in other words $k'_2 = \lceil \frac{9}{4} \rceil k'$), it is clearly that $\tilde{t}$ remains unchanged but $w = 4(k'_1 + k'_2 + k'_3)$ is reduced. Hence we completed the proof. □

Theorem 3 indicates that the optimal settings of `dblock` kernel execution arguments should have $k'_1 : k'_2 : k'_3 \approx 4 : 9 : 4$. Similarly, in `cblock` kernels, execution arguments $k'_1 : k'_2 : k'_3 : k'_4 \approx 8 : 9 : 4 : 8$

Table 1: Benchmark statistics and experimental results.

Benchmark Statistics				TBS SA Algorithm [34]				Our Strategy			
Case	#Kernels	λ_1	λ_2	Max Time	WL	Protocol	Cost	Max Time	WL	Protocol	Cost
A	17	1	0	37044	3611.5	11	40655.5	35280	2047	13	**37327**
B	34	1	0	70560	6657	20	77217	65856	4905	17	**70761**
C	102	1	0	76608	15696	69	92034	65772	4278	281	**70050**
D	54	1	0	38304	9327.5	44	47631.5	34944	3071.5	89	**38015.5**
E	17	10	100	36288	2080.5	7	57793	39690	590	16	**47190**
F	34	10	100	76608	3237	15	110478	70560	1475	14	**86710**
G	102	10	100	91728	7784	29	172468	69888	2508	141	**109068**
H	54	10	100	47040	4450	21	93640	43008	893	115	**63438**
I	27	4	0	56448	3790	16	71608	52920	612	13	**55368**
J	81	4	0	63504	8009.5	52	95542	57792	1117.5	286	**62262**
K	18	4	0	576	236	3	**1520**	504	400	14	2104
L	54	4	0	1280	910.5	60	4922	504	774	114	**3600**
M	25	4	0	2359296	9359	24	2396732	2336256	5100	67	**2356656**
N	28	4	0	2268	707.5	0	5098	1599	448.5	9	**3393**
O	27	40	400	63504	1202	6	113984	52920	612	13	**82600**
P	81	40	400	115101	4015	24	285301	66528	2273	102	**198248**
Q	18	40	400	1152	178	1	**8672**	504	400	14	22104
R	54	40	400	1372	1443	30	**71092**	504	774	114	77064
S	25	40	400	2495376	3551	25	2647416	2396160	1899.5	65	**2498140**
T	28	40	400	5720	555.5	0	27940	2015	367.5	9	**20315**
Avg							1.17×				**1.00**×

are reasonably close to optimum. In our argument selecting strategy, such prior knowledge significantly reduces the search space.

4.4 Experimental Results

We implemented our `cupid` kernel floorplan engine in `C++` programming language on a 64-bit Linux machine with a 3.4GHz Intel Xeon CPU and 32GB RAM. The results are evaluated on the ISPD20 contest benchmark suites [33].

The benchmark statistics and our experimental results are listed in Table 1. Weights λ_1 and λ_2 represent the weight of wirelength and protocol cost in the objective function, respectively. The `cost` columns describe the value of the objective in Equation (6). Our methodology performs much better than the baseline floorplan algorithm [34] based on twin binary sequence (TBS) [35] and simulated annealing, although the latter one could handle more general cases.

5 Discussion: Advanced Technologies

Advanced technologies have shown great potential to address scaling challenges. Due to the page limit, we mention a few of them and refer readers to [36] for a more comprehensive survey.

Processing-in-memory (PIM) provides massive parallelism with high energy efficiency [37], offering new solutions to address challenges in modern computer systems. Recent work have demonstrated that neural network computation can be implemented in various emerging non-volatile memories (NVM), such as RRAM [38, 39], STT-MRAM [40, 41], PCM [42], and memristor [43]. In-memory analog simulation is another promising approach, for instance [44] based on memristor crossbar and [45] based on FTJ. Without the need for moving data between memory and processor, these accelerators substantially improve the performance and efficiency of neural network execution. However, lots of systems rely on an external control device that may reduce the benefit of PIM.

The nanophotonic circuit is an alternative neuromorphic computing system due to its ultra-high bandwidth, speed and ultra-low energy consumption. In nanophotonic, signals are encoded in the amplitude of optical pulses propagating in the photonic integrated circuit, which implements a multi-layer optical neural network (ONNs) [46] . Recent advances include exploring nonlinear functions with ONNs [47] and reducing area overhead of ONNs [48].

Different 3D technologies are available to offer a wide spectrum of integration schemes. A neural network accelerator Tetris [49] implemented with through-silicon-via (TSV) 3D memory achieves 4.1× and 1.5× performance and energy improvements. Specifically, the memory substrate of Tetris is hybrid memory cube (HMC) [50], one of the well-known realizations of 3D memory (another is high bandwidth memory (HBM) [51]). ThruChip Interface (TCI), an alternative to TSV, is a high-performance wireless vertical interconnect technology used to transmit signals between multiple stacked dies. QUEST [52] is a DNN inference engine stacked with multiple SRAMs using TCI, which enables large memory capacitance. It is also studied whether Monolithic 3D (M3D) can benefit deep learning hardware design [53, 54], where a Gaussian Mixture Model for acoustic modeling is mapped to both 2D and M3D designs for comparison. Experiments conclude that M3D is a good fit for low-power DNN hardware implementation, especially for architecture with complex combinational logic.

Emerging beyond-CMOS devices give rise to new solutions for low-power designs. It is shown that HyperFET, a MOSFET replacement, greatly lowers the power consumption of spiking neural networks compared to the CMOS-based counterpart [55]. Similar attempts include Cellular Neural Networks on TFET [56]. [57] reviews emerging materials for next-generation computing technologies.

6 Conclusion

In this paper, we discussed physical synthesis for advanced neural network processors. Due to the regularity of neural network processors, we reviewed existing literature on datapath driven placement that takes circuit topology into physical design consideration. We also scrutinized a wafer-scale deep learning accelerator placement problem, a case study of specific physical synthesis for advanced neural network processors. Experimental results show that datapath driven floorplan greatly outperforms standard methods such as simulated annealing. Advanced technologies for neural network processor design are additionally discussed.

References

[1] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. FPGA*, 2015.

[2] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proc. MICRO*, 2016.

[3] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *Proc. FPGA*, 2017.

[4] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs," in *Proc. ICCAD*, 2018.

[5] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Proc. DAC*, 2017.

[6] Q. Sun, T. Chen, J. Miao, and B. Yu, "Power-driven DNN dataflow optimization on FPGA," in *Proc. ICCAD*, 2019.

[7] Y. Wang and H. Shin, "Effective regularity extraction and placement techniques for datapath-intensive circuits," *IET Circuits, Devices & Systems*, 2017.

[8] H. Cai, S. Note, P. Six, and H. De Man, "A data path layout assembler for high performance DSP circuits," in *Proc. DAC*, 1990.

[9] Y.-W. Tsay and Y.-L. Lin, "A row-based cell placement method that utilizes circuit structural properties," *IEEE TCAD*, 1995.

[10] T. T. Ye and G. De Micheli, "Data path placement with regularity," in *Proc. ICCAD*, 2000.

[11] T. Serdar and C. Sechen, "Automatic datapath tile placement and routing," in *Proc. DATE*, 2001.

[12] P.-N. Guo, C.-K. Cheng, and T. Yoshimura, "An o-tree representation of non-slicing floorplan and its applications," in *Proc. DAC*, 1999.

[13] J. Tong, H. Xianlong, C. Yici, X. Jingyu, Y. Changqi, Z. Yiqian, Z. Qiang, and W. Weimin, "Challenges to data-path physical design inside SOC," *CHINESE JOURNAL OF SEMICONDUCTORS-CHINESE EDITION-*, 2002.

[14] T. Jing, X.-L. Hong, Y.-C. Cai, J.-Y. Xu, C.-Q. Yang, Y.-Q. Zhang, Q. Zhou, and W. Wu, "Data-path layout design inside SOC," in *Proc. ICCCAS*, 2002.

[15] S. Bae, H.-O. Kim, J. Choi, and J. Park, "Coarse-grained structural placement for a synthesized parallel multiplier," in *Proc. ISPD*, 2015.

[16] T. T. Ye, S. Chaudhuri, F. Huang, H. Savoj, and G. De Micheli, "Physical synthesis for ASIC datapath circuits," in *Proc. ISCAS*, vol. 3, 2002.

[17] S. Chou, M.-K. Hsu, and Y.-W. Chang, "Structure-aware placement for datapath-intensive circuit designs," in *Proc. DAC*, 2012.

[18] J. Zhang, W. Zhang, G. Luo, X. Wei, Y. Liang, and J. Cong, "Frequency improvement of systolic array-based CNNs on FPGAs," in *Proc. ISCAS*, 2019.

[19] S. Ono and P. H. Madden, "On structure and suboptimality in placement," in *Proc. ASPDAC*, vol. 1, 2005.

[20] S. I. Ward, D. A. Papa, Z. Li, C. N. Sze, C. J. Alpert, and E. Swartzlander, "Quantifying academic placer performance on custom designs," in *Proc. ISPD*, 2011.

[21] R. X. Nijssen and J. A. Jess, "Two-dimensional datapath regularity extraction," in *Proc. IWLS*, 1996.

[22] S. R. Arikati and R. Varadarajan, "A signature based approach to regularity extraction," in *Proc. ICCAD*, vol. 97, 1997.

[23] A. Chowdhary, S. Kale, P. K. Saripella, N. K. Sehgal, and R. K. Gupta, "Extraction of functional regularity in datapath circuits," *IEEE TCAD*, 1999.

[24] H. Xiang, M. Cho, H. Ren, M. Ziegler, and R. Puri, "Network flow based datapath bit slicing," in *Proc. ISPD*, 2013.

[25] C.-C. Huang, B.-Q. Lin, H.-Y. Lee, Y.-W. Chang, K.-S. Wu, and J.-Z. Yang, "Graph-based logic bit slicing for datapath-aware placement," in *Proc. DAC*, 2017.

[26] T. Kutzschebauch and L. Stok, "Regularity driven logic synthesis," in *Proc. ICCAD*, 2000.

[27] T. Kutzschebauch, "Efficient logic optimization using regularity extraction," in *Proc. ICCD*, 2000.

[28] A. P. Rosiello, F. Ferrandi, D. Pandini, and D. Sciuto, "A hash-based approach for functional regularity extraction during logic synthesis," in *Proc. ISVLSI*, 2007.

[29] P. Ienne and A. Grießing, "Practical experiences with standard-cell based datapath design tools. do we really need regular layouts?" in *Proc. DAC*, 1998.

[30] S. Ward, D. Ding, and D. Z. Pan, "PADE: A high-performance placer with automatic datapath extraction and evaluation through high-dimensional data learning," in *Proc. DAC*, 2012.

[31] S. I. Ward, M.-C. Kim, N. Viswanathan, Z. Li, C. Alpert, E. E. Swartzlander Jr, and D. Z. Pan, "Keep it straight: teaching placement how to better handle designs with datapaths," in *Proc. ISPD*, 2012.

[32] S. I. Ward, M.-C. Kim, N. Viswanathan, Z. Li, C. J. Alpert, E. E. Swartzlander, and D. Z. Pan, "Structure-aware placement techniques for designs with datapaths," *IEEE TCAD*, 2013.

[33] "ISPD 2020 contest: Wafer-scale deep learning accelerator placement," https://www.cerebras.net/ispd-2020-contest/.

[34] B. Jiang, J. Chen, J. Liu, L. Liu, F. Wang, X. Zhang, and E. F. Young, "CU. POKer: Placing DNNs on wafer-scale Al accelerator with optimal kernel sizing," in *Proc. ICCAD*, 2020.

[35] E. F. Young, C. C. Chu, and Z. C. Shen, "Twin binary sequences: a nonredundant representation for general nonslicing floorplan," *IEEE TCAD*, 2003.

[36] D. Ielmini and S. Ambrogio, "Emerging neuromorphic devices," *Nanotechnology*, 2019.

[37] F. Wang, G. Luo, G. Sun, J. Zhang, P. Huang, and J. Kang, "Parallel stateful logic in RRAM: Theoretical analysis and arithmetic design," in *Proc. ASAP*, vol. 2160, 2019.

[38] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," *ACM SIGARCH Computer Architecture News*, 2016.

[39] X. Sun, S. Yin, X. Peng, R. Liu, J.-s. Seo, and S. Yu, "XNOR-RRAM: A scalable and parallel resistive synaptic architecture for binary neural networks," in *Proc. DATE*, 2018.

[40] H. Yan, H. R. Cherian, E. C. Ahn, and L. Duan, "Celia: A device and architecture co-design framework for STT-MRAM-based deep learning acceleration," in *Proc. SC*, 2018.

[41] Y. Pan, P. Ouyang, Y. Zhao, W. Kang, S. Yin, Y. Zhang, W. Zhao, and S. Wei, "A multilevel cell STT-MRAM-based computing in-memory accelerator for binary convolutional neural network," *IEEE Transactions on Magnetics*, 2018.

[42] B. Kim, S. H. Lee, H. Kim, D.-T. Nguyen, M.-S. Le, I. J. Chang, D. Kwon, J. H. Yoo, J. W. Choi, and H.-J. Lee, "PCM: precision-controlled memory system for energy efficient deep neural network training," in *Proc. DATE*, 2020.

[43] P. Yao, H. Wu, B. Gao, J. Tang, Q. Zhang, W. Zhang, J. J. Yang, and H. Qian, "Fully hardware-implemented memristor convolutional neural network," *Nature*, 2020.

[44] C. Li, D. Belkin, Y. Li, P. Yan, M. Hu, N. Ge, H. Jiang, E. Montgomery, P. Lin, Z. Wang *et al.*, "Efficient and self-adaptive in-situ learning in multilayer memristor neural networks," *Nature communications*, 2018.

[45] J. Li, C. Ge, J. Du, C. Wang, G. Yang, and K. Jin, "Reproducible ultrathin ferroelectric domain switching for high-performance neuromorphic computing," *Advanced Materials*, 2020.

[46] Y. Shen, N. C. Harris, S. Skirlo, M. Prabhu, T. Baehr-Jones, M. Hochberg, X. Sun, S. Zhao, H. Larochelle, D. Englund *et al.*, "Deep learning with coherent nanophotonic circuits," *Nature Photonics*, 2017.

[47] Y. Zuo, B. Li, Y. Zhao, Y. Jiang, Y.-C. Chen, P. Chen, G.-B. Jo, J. Liu, and S. Du, "All-optical neural network with nonlinear activation functions," *Optica*, 2019.

[48] J. Gu, Z. Zhao, C. Feng, M. Liu, R. T. Chen, and D. Z. Pan, "Towards area-efficient optical neural networks: An FFT-based architecture," in *Proc. ASPDAC*, 2020.

[49] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proc. ASPLOS*, 2017.

[50] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *Proc. VLSIT*, 2012.

[51] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin *et al.*, "25.2 a 1.2 v 8gb 8-channel 128gb/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV," in *Proc. ISSCC*, 2014.

[52] K. Ueyoshi, K. Ando, K. Hirose, S. Takamaeda-Yamazaki, J. Kadomoto, T. Miyata, M. Hamada, T. Kuroda, and M. Motomura, "QUEST: A 7.49 TOPS multi-purpose log-quantized DNN inference engine stacked on 96MB 3D SRAM using inductive-coupling technology in 40nm CMOS," in *Proc. ISSCC*, 2018.

[53] K. Chang, D. Kadetotad, Y. Cao, J.-s. Seo, and S. K. Lim, "Monolithic 3D IC designs for low-power deep neural networks targeting speech recognition," in *Proc. ISLPED*, 2017.

[54] K. Chang, D. Kadetotad, Y. Cao, J.-S. Seo, and S. K. Lim, "Power, performance, and area benefit of monolithic 3D ICs for on-chip deep neural networks targeting speech recognition," *ACM JETC*, 2018.

[55] W.-Y. Tsai, X. Li, M. Jerry, B. Xie, N. Shukla, H. Liu, N. Chandramoorthy, M. Cotter, A. Raychowdhury, D. M. Chiarulli *et al.*, "Enabling new computation paradigms with HyperFET-an emerging device," *IEEE TMSCS*, 2016.

[56] I. Palit, X. S. Hu, J. Nahas, and M. Niemier, "TFET-based cellular neural network architectures," in *Proc. ISLPED*, 2013.

[57] C. Liu, H. Chen, S. Wang, Q. Liu, Y.-G. Jiang, D. W. Zhang, M. Liu, and P. Zhou, "Two-dimensional materials for next-generation computing technologies," *Nature Nanotechnology*, 2020.