CU.POKer: Placing DNNs on Wafer-Scale AI Accelerator with Optimal Kernel Sizing

Bentian Jiang*
Chinese University of Hong Kong
btjiang@cse.cuhk.edu.hk

Jingsong Chen*
Chinese University of Hong Kong
jschen@cse.cuhk.edu.hk

Jinwei Liu
Chinese University of Hong Kong
jwliu@cse.cuhk.edu.hk

Lixin Liu
Chinese University of Hong Kong
lxliu@cse.cuhk.edu.hk

Fangzhou Wang
Chinese University of Hong Kong
fzwang@cse.cuhk.edu.hk

Xiaopeng Zhang
Chinese University of Hong Kong
xpzhang@cse.cuhk.edu.hk

Evangeline F.Y. Young
Chinese University of Hong Kong
fyyoung@cse.cuhk.edu.hk

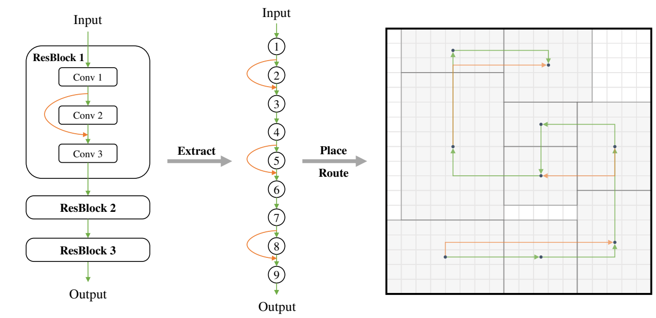
ABSTRACT

The tremendous growth in deep learning (DL) applications has created an exponential demand for computing power, which leads to the rise of AI-specific hardware. Targeted towards accelerating computation-intensive deep learning applications, AI hardware, including but not limited to GPGPU, TPU, ASICs, etc., have been adopted ubiquitously. As a result, domain-specific CAD tools play more and more important roles and have been deeply involved in both the design and compilation stages of modern AI hardware. Recently, ISPD 2020 contest introduced a special challenge targeting at the physical mapping of neural network workloads onto the largest commercial deep learning accelerator, CS-1 Wafer-Scale Engine (WSE). In this paper, we proposed CU.POKer, a high-performance engine fully-customized for WSE's DNN workload placement challenge. A provably optimal placeable kernel candidate searching scheme and a data-flow-aware placement tool are developed accordingly to ensure the state-of-the-art quality on the real industrial benchmarks. Experimental results on ISPD 2020 contest evaluation suites [1] demonstrated the superiority of our proposed framework over other contestants.

1 INTRODUCTION

The applications of deep learning (DL) have risen in an eye-watering fashion. From the invincible go "player" AlphaZero [2], to the state-of-the-art (SOTA) NLP model GPT-3 [3], and the well-known U-Net [4], ResNet [5] families for image processing, there is a deep neural network (DNN) at the core of things unsurprisingly. The tremendous growth in DL applications has created an exponential demand for computing power, and the amount of computing resources needed for training is very likely correlated to how powerful our best models are. It has been reported that from 2012 to 2018, the computing requirements in the largest AI training runs increased by over 300,000x and will continue to increase at a rate far beyond the Moore's Law [6]. In response to such growing demands, both industry and academia have been actively exploring dedicated computer architectures specialized for deep learning. AI hardware, including but not limited to GPGPU, TPU and various dedicated ASICs, also shared a rapid evolution in the past decade.

The generalized training procedure of the deep neural network consists of iterative forward propagation and backward propagation over a training dataset, while the inference procedure usually refers to one-round of forward propagation of the pre-trained network with respect to a given input. However, the training process is drastically computationally intensive and requires high memory and communication bandwidth. Take the ResNet-50 as an example, a well-trained ResNet-50 on a small images set (224x224) consumes 100 millions training steps while each step takes around 24 billion floating point operations (FLOPs) [7]. It is not hard to imagine that even with advanced algorithmic innovations and sizable GPU/TPU farm, training those powerful commercial models usually takes hours to days [6]. Targeting towards the acceleration of training models with millions or billions of parameters, industry recently released CS-1 Wafer-Scale Engine (WSE), the largest commercial



(a) Network Architecture (b) Kernel Graph (c) Execution Plan

Figure 1: Overview of WSE compilation flow, the proposed framework focuses on the placement stage of compilation.

* These authors contributed equally to this work.

ICCAD '20, November 2–5, 2020, Virtual Event, USA

ACM ISBN 978-1-6654-2324-3/20/11…$15.00
https://doi.org/10.1145/3400302.3415688

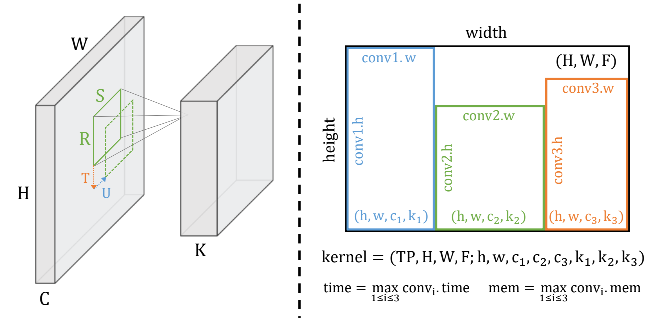
DL chip with a novel customized architecture. Architecturally, WSE consists of over 400,000 programmable execution cores, while each of them possesses 48KB SRAM (18 GB in total), and a router for interconnection with other cores (100 Pb/s communication bandwidth). Powered by its wafer-scale die size (56× larger than today's largest GPU), WSE can provide cluster-scale resources and model parallelism on a single chip, such that a typical deep learning model with tens of billions of arithmetic operations can be rapidly evaluated on a compiled WSE. Consequently, industry claimed WSE to be the largest and most powerful DL processor ever built [7].

Considering the fact that the key feature of WSE lies in its sufficiently large capability to run every layer of a neural network simultaneously, a critical challenge naturally arises: how to maximize the on-chip resource utilization to achieve a substantial increase in FLOPS/Watt with respect to different DNN architectures? For a given DNN, the compiled model of WSE must assign each arithmetic operation to an execution core as well as to specify the communication path to deliver the result of one operation to all dependent operations [7]. The corresponding compilation stage of WSE is depicted in Figure 1, the input neural network models (expressed in common ML frameworks) will be converted to a graph representation using a set of predetermined kernels provided in a kernel library. CAD tool is then used to place & route the constituent kernels of a neural network on the computational fabric with certain objectives and constraints.

Our answer to the aforementioned challenge is to develop CAD tool specialized for the WSE compilation. Interestingly, most previous learning-based researches in the placement area mainly focus on how to integrate DNNs and AI hardware into conventional placement flows so as to achieve breakthrough performance boosting. E.g., DREAMPlace [8] for the first time casts the conventional nonlinear standard cell placement problem as a DNN training process without quality loss, while a recent work [9] from Google is able to achieve better marco placement quality based on reinforcement learning. It is worth noting that the majority of research interests focus on specializing AI for placement rather than specializing placement for AI. For efficiency consideration, we believe that it is imperative to derive fully customized CAD flows for those dedicated AI chips. Recently, ISPD 2020 contest [1] introduced a special challenge targeting at placing neural network workloads onto the CS-1 WSE. Intuitively, this task is reminiscent of the traditional floorplanning problem in physical design. However, with further explorations, we find that common floorplanning heuristics are not able to handle this new challenge well, and a novel framework is required to address the intrinsic properties of this problem.

In this paper, we focus on placing DNNs on the AI accelerator so as to achieve its maximum resource utilization (i.e., placement for AI). Our key contributions are summarized as follows.

- An extremely fast and provably optimal kernel sizing algorithm is proposed, which helps to select the kernel candidates with optimal shapes while eliminating most unnecessary enumeration overheads.
- Our proposed data-path-aware placement algorithm is fully customized for this on-WSE DNN placement problem. Multiple objectives are simultaneously considered during the optimization process.



(a) Arguments of *conv* (b) Performance of a kernel with 3 *convs*

Figure 2: Visualizations of the (a) arguments of a basic convolution (b) performance of a kernel with 3 basic conv units.

- A universal scheme regardless of kernel types and protocol functions is proposed to optimize the adapter costs.
- We further compared the proposed flow with several conventional heuristics used in general floorplanning/placement. Experimental results show that CU.POKer outperforms not only the other contestants but also the conventional heuristics used in general floorplanning/placement.

The rest of the paper is organized as follows. Section 2 lists some preliminaries. Section 3 discusses the details of the framework and algorithms. Section 4 presents experimental results, followed by a conclusion in Section 5.

2 PRELIMINARIES

During the WSE compilation, the layers of the input neural networks are decomposed into a set of **kernels** and each performs an individual deep learning task (e.g., to compute a '5x5 convolution' or to implement a '256 to 16 fully connected' layer), while the data dependency information between layers are embedded into a connected kernel graph. The primary goal of this DNN placement problem is to select the configurations of the constituent kernels of a neural network and to determine their locations on the computational fabric, in such way to optimize the ultimate performance.

2.1 Kernel Definition

Formally, a kernel is a parametric program that performs specific tensor operations. It consists of a set of **formal arguments** to specify how the tensor operations are performed on the execution cores, and a set of **execution arguments** to describe how the operation is parallelized across the execution cores [7]. For a given kernel, its formal arguments are uniquely determined by the input neural network specification and should remain unchanged during the entire compilation process, while its execution arguments are configurable, whose values are variables to be optimized by the placer. Take the basic convolution kernel (*conv*) as an example, *conv* provides the basic convolution operation corresponding to the forward propagation, backward propagation and weight update of a neural network [7]. It contains 8 formal arguments (H, W, R, S, C, K, T, U) and 4 execution arguments (h, w, c, k). As illustrated in Figure 2 (a), for the formal arguments, (H, W) specify the input image size in 2-dimensions; (R, S) correspond to the size of the convolution core in 2-dimensions; (C, K) refer to the number of input and output features and (T, U) indicate the horizontal and vertical striding of the convolution operation. The rest 4 execution arguments (h, w, c, k) are the variables to be determined.

2.2 Kernel Evaluation

For each kernel, the formal arguments and the execution arguments can jointly determine the shape, area, execution time, memory, and I/O protocols used by this kernel.

2.2.1 Kernel Performance Function [7]. Each kernel is provided with a unique performance function mapping the kernel arguments to a 4-tuple **performance cuboid** (height, width, time, memory), which describes the on-chip computing resources needed to execute this kernel. The performance functions of all placeable kernels are specified in a given kernel library. For instance, the performance cuboid of the basic convolution kernel (*conv*) can be defined as follows,

$$\begin{aligned}
&\text{convperf}(\overbrace{H,\ W,\ R,\ S,\ C,\ K,\ T,\ U}^{\text{Formal arguments}};\ \overbrace{h,\ w,\ c,\ k}^{\text{Execution arguments}}) = \{\\
&\quad height = h \times w \times (c+1)\\
&\quad width = 3 \times k\\
&\quad time = \text{ceil}(\frac{H}{h}) \times \text{ceil}(\frac{W}{w}) \times \text{ceil}(\frac{C}{c}) \times \text{ceil}(\frac{K}{k}) \times \frac{RS}{T^2}\\
&\quad memory = \frac{C}{c} \times \frac{K}{k} \times RS + \frac{W+S-1}{w} \times \frac{H+R-1}{h} \times \frac{K}{k}\ \ \}.
\end{aligned} \tag{1}$$

In the benchmarks provided, all other kernels are built upon this basic convolution kernel (*conv*). In other words, every kernel is consisted of several *convs* with different formal arguments. For a certain type of kernel that contains n *conv*, its performance cuboid is given by,

$$\begin{aligned}
&\text{blockperf}(TP, H, W, F; h, w, c_1, ..., c_n, k_1, ..., k_n) = \{\\
&\quad conv_i = \text{convperf}(H_i, W_i, R_i, S_i, C_i, K_i, T_i, U_i; h, w, c_i, k_i),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \forall\ i \in \{1, ..., n\}\\
&\quad height = \max_{1\le i\le n} conv_i.height, \quad width = \sum_{i=1}^{n} conv_i.width\\
&\quad time = \max_{1\le i\le n} conv_i.time, \quad mem = \max_{1\le i\le n} conv_i.mem\ \ \},
\end{aligned} \tag{2}$$

where TP (type of the kernel), H, W and F are given. TP will determine n, the number of convolution cores. TP, H, W and F will jointly determine the values of $H_i, W_i, R_i, S_i, C_i, K_i, T_i, U_i$ (for $i = 1, ..., n$) according to a look-up table, and $(h, w, c_1, ..., c_n, k_1, ..., k_n)$ are the execution variables to be optimized by the placer. Figure 2 (b) visualizes the physical shape of a kernel that contains 3 *conv* cores.

2.2.2 Kernel Protocol Function [7]. Each kernel is also provided with a protocol function mapping kernel arguments to a 3-tuple **input/output** protocol cuboid, that specifies how many wires are **consumed/produced** by each communication port. For each connected kernel pair (2 kernels) in the kernel graph, there is an adapter cost that reflects the mismatch of their communicating wires. The adapter cost is computed as the summation of the number of tuple elements that are not equal. For instance, let $ker_1 = \{TP_1, H_1, W_1, F_1; h_1, w_1, c_{1,1}, ..., c_{1,n}, k_{1,1}, ..., k_{1,n}\}$ be a **predecessor** of $ker_2 = \{TP_2, H_2, W_2, F_2; h_2, w_2, c_{2,1}, ..., c_{2,m}, k_{2,1}, ..., k_{2,m}\}$, their protocol cuboids are $(h_1, w_1, c_{1,n})$ for the **output** port of ker_1 and $(h_2, w_2, c_{2,1})$ for the **input** port of ker_2. Then, the adapter cost between this pair is given by,

$$\text{cost}_{\text{adapter}} = \mathbb{1}(h_1 != h_2) + \mathbb{1}(w_1 != w_2) + \mathbb{1}(c_{1,n} != c_{2,1}), \tag{3}$$

where $\mathbb{1} \in \{0,\ 1\}$ is the indicator function.

2.3 Problem Formulation [7]

Given a kernel library and an input kernel graph, the problem is to determine the execution parameters and the locations for all kernels. The solutions must fullfill the following constraints:

- All kernels must fit within the fabric area (633 x 633 tiles).
- No kernels may overlap.
- No kernel's memory exceeds the tile's memory limit.

The placement run time must not exceed a specified time limit, and the quality of a feasible solution is given by a weighted summation of the following objectives that need to be minimized:

- The maximum execution time (in the performance cuboid) among all placed kernels.
- The total L1 distance (wirelength) of all connected kernels.
- The total adapter cost of all connected kernels.

A compiled model on WSE is executed in a pipeline fashion, so the kernel instance with the slowest throughout (corresponds to the kernel with the maximum execution time) will determine the overall performance of the pipeline. L1 distance provides a simplified proxy for the routing overhead. Adapter cost reflects the effort needed to unify the I/O protocols among kernels in the real-world solutions.

3 ALGORITHMS

The objective of our placement engine is to minimize a weighted summation of the maximum execution time, the total wirelength, and the total adapter cost. According to the kernel performance function in Section 2.2.1, a shorter kernel execution time will lead to a larger kernel physical size, which will hinder not only the minimization of the total wirelength but also the packing of all the kernels. Thus, the maximum execution time and the total wirelength should be considered simultaneously. As shown in Figure 3, our placement engine adopts a *binary search* and a *neighbor-range search* to find a good maximum execution time. For each targeted maximum execution time (*target_time*) T, kernel candidates with optimal shapes and execution times not exceeding T will be generated, which will be used to perform data-path-aware kernel placement with the objective of minimizing the total wirelength while no kernel's execution time shall exceed T. Since one kernel candidate corresponds to a suite of execution parameters, during kernel placement, a complete solution can be generated by selecting which candidate to use and deciding the location for each kernel. After the neighbor-range search, a post refinement will be performed to reduce the total adapter cost of the current best solution.

We will describe our binary search and neighbor-range search which constitute the main framework of the overall flow in Section 3.1. We will then detail our kernel candidate generation in Section 3.2, data-path-aware placement in Section 3.3, and post refinement for adapter cost in Section 3.4.

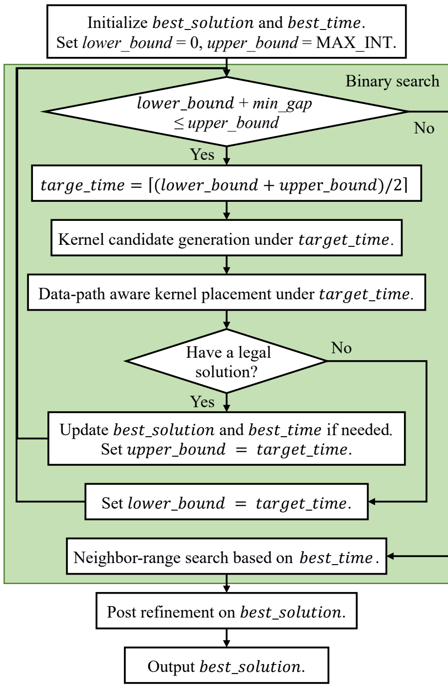


Figure 3: Overall Flow

3.1 Overall Flow

As this placement problem is NP-hard, our methodology finds a good solution by a two-step search consisting of a binary search and a neighbor-range search. As shown in the green box of Figure 3, a typical binary search will first be performed on the maximum execution time. This binary search will rapidly locate a small and feasible maximum execution time while the total wirelength will usually increase when the targeted time T decreases. However, we will always maintain the best current solution in terms of the overall objective value.

After the binary search, the maximum execution time will be well optimized, but since there are other metrics in the objective function and the placement method to check whether a certain targeted time is feasible is not perfect (an NP-hard problem), more searching in a neighborhood range will optimize the solution further. Therefore, after the binary search, we will take the best maximum execution time T_b found as the starting point for a fine-grained neighbor-range search to further improve the solution. In our neighbor-range search, the neighbor ranges above and below T_b on the timeline will be scanned in a predefined step size (in our setting, the total searching scope is 40% of T_b, and the step size is 0.2% of T_b). Similar to the binary search, for each targeted time, we will check whether a feasible placement can be obtained by our data-path aware placer which will minimize wirelength as much as possible. In this way, after the neighbor-range search, we can obtain a good solution which is well optimized in terms of both the maximum execution time and the total wirelength. The current best solution will then be passed to the post refinement step for adapter cost optimization.

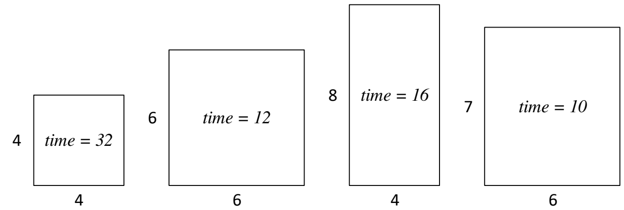


Figure 4: Optimal and sub-optimal kernel shapes

3.2 Kernel Candidates Generation

Generating and pruning the kernel candidates is one of the major challenges for wafer placement, given that the total number of argument options can be astronomically large. Meanwhile, our proposed optimal kernel sizing algorithm can find all the kernel candidates with optimal shapes and satisfying the *target_time* constraint in a very short time. To begin with, the kernels considered in this work have a variable number (x) of convolution cores. The performance (*height*, *width*, *time* and *memory*) of each convolution core can be obtained using Equation (1), while the kernel performance can be computed by Equation (2).

The task of kernel candidate generation is formulated as finding the set of execution variables that produce the kernels with optimal shapes while satisfying the *target_time* constraint ($time \leq target\_time$). We refer to a kernel ($height = h_1, width = w_1$) having optimal shape if and only if there does not exist another kernel ($height = h_2, width = w_2$) satisfying the same *target_time* constraint, such that it has a better shape with or without rotation, i.e., $max(h_2, w_2) \leq max(h_1, w_1)$ and $min(h_2, w_2) \leq min(h_1, w_1)$. For example, suppose that there are totally 4 candidates as shown in Figure 4 and $target\_time = 16$. The first kernel will be immediately ruled out because of exceeding *target_time*. The second kernel has a better shape compared to the fourth one, and both the second and the third kernels are regarded as optimal. Note that though the fourth kernel has a better execution time compared to the second kernel, it is still unwanted, since the overall maximum execution time is already bounded by *target_time*, and reducing the execution time of one single kernel will not lead to a better overall solution. The reason of keeping the optimal shapes only is that we can always find an optimal replacement for a sub-optimal shape, such that the wirelength is reduced or remains the same. Besides, all kernels considered should be legal, i.e., their heights and widths should not exceed the fabric size, and their memories must be lower than the given threshold *memory_limit*.

Next we will show that enforcing $c_1 = c_2 = ... = c_x = c$ will not sacrifice optimality by Theorem 3.1.

THEOREM 3.1. *For any argument* $\{h, w, c_1, ..., c_x, k_1, ..., k_x\}$, *there exist a* $c = \max(c_1, ..., c_x)$ *such that*

$$ker_1 = blockperf(TP, H, W, F; h, w, c, ..., c, k_1, ..., k_x),$$

is no worse than

$$ker_2 = blockperf(TP, H, W, F; h, w, c_1, ..., c_x, k_1, ..., k_x)$$

with regard to height, width, time and memory.

PROOF. Let $c = \max(c_1, ..., c_x)$, and all the four attributes will be either improved or unchanged. Firstly, the *height* will be kept the same because,

$$\begin{aligned} ker_1.height &= h \times w \times (c+1) \\ &= \max_{1 \le j \le x} h \times w \times (c_j + 1) \\ &= ker_2.height \end{aligned}$$

Secondly, the *width* will also be unchanged because its computation is irrelevant to $c_1, ..., c_x$. Lastly, both the *time* and *memory* of the j^{th} convolution core are inversely correlated to c_j according to Equation (1), which means that increasing c_j to c will reduce both of them. Thus, the *time* and *memory* of the kernel itself will also be reduced or kept unchanged according to Equation (2). □

Besides, there exists at most one optimal shape for each *height*, which is the one with the smallest *width*. If multiple legal candidates have the same *height* and *width*, only the one first found is kept. Therefore, finding the optimal-shaped kernel candidate for $height = \eta$ can be formulated as the optimization problem in Equation (4). We will solve the optimization problem for $\eta = 1, ..., max\_height$ (where *max_height* is an upper bound for *height*) separately to find the optimal shapes for each *height*. Note that some of them can have no solution, as all the arguments should be positive integers.

$$\begin{aligned} &\underset{h,w,c,k_1,...,k_x}{\text{Minimize:}} \ width \\ &\quad \text{s.t.: } height = h \times w \times (c+1) = \eta \\ &\quad\quad width = \sum_{j=1}^{x} 3 \times k_j \\ &\quad\quad time = \max_{1 \le j \le x} \text{ceil}(\frac{H_j}{h})\text{ceil}(\frac{W_j}{w})\text{ceil}(\frac{C_j}{c})\text{ceil}(\frac{K_j}{k_j})\frac{R_j S_j}{T_j^2} \\ &\quad\quad\quad \le target\_time \\ &\quad\quad mem = \max_{1 \le j \le x} \frac{C_j K_j R_j S_j}{c k_j} + \frac{(W_j + S_j - 1)(H_j + R_j - 1)K_j}{w h k_j} \\ &\quad\quad\quad \le memory\_limit. \end{aligned} \tag{4}$$

Note that all variables in capital letters are constants. We will solve the optimization problem for each η by factorizing η to obtain the possible values of h, w and c. For each possible values of h, w and c, we will compute the minimum $k_1, ..., k_x$ that satisfy both the *target_time* and the *memory_limit* constraint. More specifically we will factorize η into three integral terms, i.e., $\eta = \alpha_1 \times \beta_1 \times \gamma_1$, ..., $\eta = \alpha_n \times \beta_n \times \gamma_n$ (where $\gamma_i \ge 1$, for $i = 1, ..., n$). For each factorization, we will put $h = \alpha_i$, $w = \beta_i$ and $c = (\gamma_i - 1)$, and find the values of $k_1, ..., k_x$ that lead to the minimum *width*. Since *width* is computed as three times the sum of $k_1 ...., k_n$, we can compute the minimum value of each k_j separately by,

For j = 1, ..., x :

$$\begin{aligned} k_j^t &= \text{ceil}(\text{ceil}(\frac{H_j}{h})\text{ceil}(\frac{W_j}{w})\text{ceil}(\frac{C_j}{c})\frac{R_j S_j K_j}{T_j^2 \times target\_time}) \\ k_j^m &= \text{ceil}(\frac{C_j K_j R_j S_j}{c \times memory\_limit} + \frac{(W_j + S_j - 1)(H_j + R_j - 1)K_j}{wh \times memory\_limit}) \\ k_j &= \max(k_j^t, k_j^m), \end{aligned}$$

where k_j^t is the minimum value of k_j that satisfies the *time* constraint, and k_j^m is the minimum value of k_j that satisfies the *memory* constraint. The minimum k_j that satisfies both constraints is simply the larger one of the two. This is the way we compute all possible values of $\{h, w, c, k_1, ..., k_x\}$ for each η. The combination that achieves the minimum *width* for each η is kept.

Algorithm 1 Optimal Kernel Sizing Algorithm

function KernelSizing($TP, H, W, F, target\_time$)
  $res \leftarrow \{\}$
  Decide x according to TP
  for $\eta = 1, ..., max\_height$ **do**
    Factorize η to get $\eta = \alpha_1 \times \beta_1 \times \gamma_1, ..., \eta = \alpha_n \times \beta_n \times \gamma_n$
    $best\_arg \leftarrow None$
    $best\_width \leftarrow MAX\_NUM$
    for $i = 1, ..., n$ **do**
      $h \leftarrow \alpha_1$, $w \leftarrow \beta_1$, $c \leftarrow (\gamma_1 - 1)$
      for $j = 1, ..., x$ **do**
        $k_j \leftarrow$ minimum k_j such that $conv_j.time <$ $target\_time$ and $conv_j.mem < memory\_limit$
      $ker_i \leftarrow blockperf(TP, H, W, F; h, w, c, k_1, ..., k_x)$
      if $ker_i.width < best\_width$ **then**
        $best\_width \leftarrow ker_i.width$
        $best\_arg \leftarrow \{h, w, c, k_1, ..., k_x\}$
    if $best\_arg$ is not $None$ **then**
      Append $best\_arg$ to res
  FinalPruning(res)
  return Return res

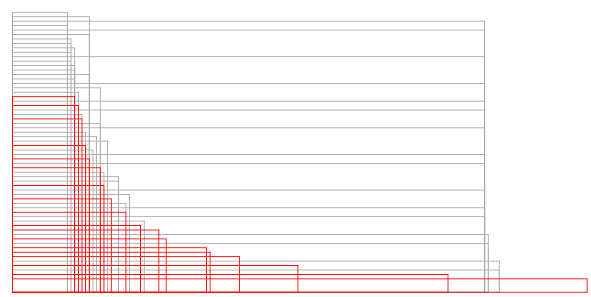


Figure 5: An example solution of kernel sizing. For this specific case, we obtain 60 shapes after solving Equation (4), but only 18 of them (the red rectangles) are optimal and output.

Algorithm 1 gives the full algorithm. After finding the argument that produces the minimum *width* for each *height*, we will perform a final pruning step to keep the kernel candidates with optimal shapes only. Figure 5 gives an example solution of our optimal kernel sizing algorithm, where the grey rectangles are the shapes obtained after solving the optimization problems in Equation (4) for all the values of *heights*, and the red rectangles are the ones remained after the final pruning.

3.3 Data-path-aware Kernel Placement

In this section, we will detail our data-path-aware kernel placement method which can efficiently generate a solution with the maximum execution time not exceeding a given *target_time*. The former step has generated all the kernel candidates with optimal shapes under the targeted time constraint.

The task of kernel placement is to select which candidate to use and to decide the location for each kernel. The objective is to minimize the total wirelength under the given $target\_time$. Note that as rotated candidates have been generated, there is no need to consider rotation during this kernel placement. The method of our kernel placement aims at placing compactly the kernels one by one according to their topological order in the connectivity graph. The topological order is generated by depth-first search on the connectivity graph. Take the 8-kernel connectivity graph in Figure 6 as an example, the topological order is $ABCEDFGH$. By placing kernels in topological order, connected kernels will be placed close to each other, which facilitates the minimization of the total wirelength.

In Algorithm 2, the kernels will be placed row-wise, starting from the bottom of the chip (e.g., Figure 7). The function `placement` is implemented in a recursive manner where each recursive step will place one row of kernels. The three input parameters of `placement` indicate the index (in the topological order) of the kernel to start placing with ($next\_index$), the targeted maximum execution time ($target\_time$), and the height (from the bottom of the chip) at which to start placing the kernels ($floor\_height$). Thus, when invoking `placement` as in Figure 3, the first and third parameters should be 1 and 0 respectively which means that the placement starts placing with the first kernel and from the bottom of the chip. Note that the function `placement` will be called recursively to place multiple rows, one on top of the other (line 23 of Algorithm 2).

To fully explore the solution space, different heights of all the kernel candidates (stored in H_k) will be traversed in ascending order. When placing a row with a height h from H_k, for the kernel to be placed, only the candidates with height not exceeding h can be used. In order to minimize wirelength, the candidate with the minimum width will be selected (line 9). After placing one row, if all the kernels have been placed (line 20), a legal solution is obtained and the current best solution will be updated accordingly. Otherwise, there are some more kernels to be placed and a new row will be created to accommodate them (line 24).

In this algorithm, since all different heights of a candidate set will be explored and only the candidates with the minimum width will be selected, our placer can guarantee a well-optimized total wirelength under the targeted maximum execution time. However, such recursion can be computational expensive and some pruning techniques are needed to speed up the search. First, a threshold t_{row} is used to control the maximum number of rows in `placement`, which is initialized to one. In other words, when performing `placement` with a threshold t_{row}, the kernels will not be placed in more than t_{row} rows. If and only if no legal solution can be found for a threshold t_{row}, `placement` will be invoked again with the threshold set to $t_{row} + 1$. By using this technique, `placement` will not explore all different row numbers to find the optimal solution, which will be extremely computational expensive and not affordable. Meanwhile, this strategy is very reasonable since less number of rows will usually lead to a smaller total wirelength.

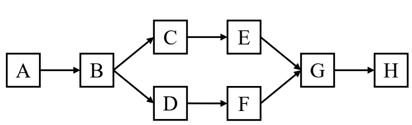


Figure 6: A 8-kernel connectivity graph.

Algorithm 2 Data-path-aware Kernel Placement

1: **function** `Placement`($next\_index, target\_time, floor\_height$)
2: $\quad H_k \leftarrow$ a sorted height set of all the kernel candidates
3: $\quad$ **for** each height h in H_k **do**
4: $\quad\quad$ **if** $h + floor\_height > chip\_height$ **then**
5: $\quad\quad\quad$ **break**
6: $\quad\quad w_{idle} \leftarrow chip\_width$
7: $\quad\quad max\_height \leftarrow 0$
8: $\quad\quad$ **for** $i = next\_index, ..., num\_kernel$ **do**
9: $\quad\quad\quad w_i \leftarrow$ minimum width of the i^{th} kernel's candidates meeting the requirements of $target\_time$ and h
10: $\quad\quad\quad h_i \leftarrow$ the corresponding height of w_i
11: $\quad\quad\quad$ **if** $w_i > w_{idle}$ **then**
12: $\quad\quad\quad\quad i \leftarrow i - 1$
13: $\quad\quad\quad\quad$ **break**
14: $\quad\quad\quad$ **else**
15: $\quad\quad\quad\quad w_{idle} \leftarrow w_{idle} - w_i$
16: $\quad\quad\quad\quad max\_height \leftarrow \max(max\_height, h_i)$
17: $\quad\quad$ **if** $i < next\_index$ **then**
18: $\quad\quad\quad$ **continue**
19: $\quad\quad$ Place the kernels of indices from $next\_index$ to i in a row on $floor\_height$
20: $\quad\quad$ **if** $i \equiv num\_kernel$ **then**
21: $\quad\quad\quad$ Update the best solution if needed
22: $\quad\quad$ **else**
23: $\quad\quad\quad floor\_height \leftarrow floor\_height + max\_height$
24: $\quad\quad\quad$ `Placement`($i, target\_time, floor\_height$)

In addition, two pruning steps are deployed in the `placement` function. First, after placing one kernel, our placer will check if the remaining empty space on the fabric is less than the smallest total area of the kernels yet to be placed. If so, the current placement iteration will be stopped. Second, recall that when placing a row, the heights in H_k will be tried in ascending order. However, some heights in H_k are redundant. For example, if there are two adjacent height values h_1 and h_2 in H_k where $h_1 < h_2$. After placing a row of height h_1 with a set Q of kernels, h_2 will be tried. If it happens that all the kernels in Q have no candidate with height between h_1 and h_2, we do not need to try h_2 since the same placement will be resulted for that row. We will identify and skip such redundant height values when traversing H_k to avoid unnecessary iterations.

3.4 Adapter Cost Optimization

Unlike the intrinsic tradeoff between maximum execution time and wirelength (faster execution time indicates larger kernel physical size), there is no explicit correlation between the adapter cost and the above two objectives. The adapter cost of a placed kernel can only be determined after the statuses of all its connected kernels were known. Any revision on the arguments of a kernel may affect the adapter costs of both its input and output ports. Thus, the optimization of the adapter cost requires a global view of the entire kernel graph and one may need to revise multiple relevant kernels simultaneously to bring improvement to the cost, which is **not compatible** with the sequential placement flow (Section 3.3).

Considering the irrelevance among the adapter cost and the other two objectives (wirelength and execution time), an alternative strategy is to spilt them into two stages of optimization, where the

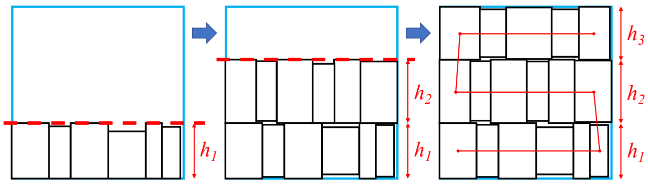


Figure 7: Most placed kernels can not achieve the max floor height and thereby generate wasted deadspace.

placer only focuses on finding the best tradeoff between wirelength and maximum execution time; based on which an additional process of post refinement is applied to minimize the adapter cost.

3.4.1 Wasted Deadspace. As illustrated in Figure 7, a typical solution generated by our data-path-aware placer tries to arrange the kernels in a row-based (floor-based) manner. The height of each floor is uniquely determined by the maximum height among all the kernels placed on this floor. It is worth noting that not every kernel placed on this floor will have its height equal to the floor height since they have different performance functions and formal arguments. Suppose there are n kernels on the i^{th} floor of the layout, for each kernel $ker_{i,j}$, $j \in \{1, ..., n\}$, we have

$$ker_{i,j}.height \leq floor_i.height = \max_{1 \leq j \leq n} ker_{i,j}.height.$$

If $ker_{i,j}.height < floor_i.height$, there exists wasted deadspace around $ker_{i,j}$ with $\Delta height_{i,j} = (floor_i.height - ker_{i,j}.height)$ and $width_{i,j} = ker_{i,j}.width$. Since the kernels on each floor are aligned horizontally, the deadspace can be made use of to adjust the executions arguments of the corresponding kernel, i.e., to increase the kernel height to the floor height and to optimize the adapter cost without sacrificing wirelength (kernel width remains unchanged).

3.4.2 Unifying (h, w) Pair. W.l.o.g., lets assume $ker_{i,j}$, the j^{th} kernel on the i^{th} floor, contains m convolution cores (*conv*), and it is connected with $ker_{i,j-1}$ and $ker_{i,j+1}$ respectively. The kernel height is given by,

$$ker_{i,j}.height = h \times w \times (c_{max} + 1) = \max_{1 \leq j \leq m} h \times w \times (c_j + 1).$$

Observing that $floor_i.height = (ker_{i,j}.height + \Delta height_{i,j})$. If we **fix** the height of $ker_{i,j}$ to exactly the floor height, a new c_{max} can be obtained by,

$$c_{max} = floor_i.height/(h * w) - 1. \tag{5}$$

Given c_{max} as in Equation (5), a new assignment for $ker_{i,j}$'s arguments $(c_1, ..., c_m)$ can be,

$$c_1 = ... = c_m = c_{max} = floor_i.height/(h * w) - 1. \tag{6}$$

Note that after extending the height of $ker_{i,j}$ to $floor_i.height$ and subject to the requirement of having a given (h, w) pair, the assignment of $(c_1, ..., c_m)$ according to Equation (6) is the optimal, by following a similar argument as in the proof of Theorem 3.1. Above observation demonstrates the feasibility of unifying the (h, w) pairs of all kernels on the same floor to a reference pair $(h_{\text{ref}}, w_{\text{ref}})$ by: (1) fixing their heights to the exact floor height; (2) adjusting $c_1 = ... = c_m = c_{max} = floor.height/(h_{\text{ref}} * w_{\text{ref}}) - 1$.

Recall that the input and output protocol cuboids (Section 2.2.2) of a kernel consist of 3 elements (h, w, c_x), where the first two elements (h, w) are identical for both cuboids, but c_x is selected from $(c_1, ..., c_m)$ by the kernel protocol function, and m is the number of *conv* cores in this kernel. Therefore, unifying the (h, w) pair is a universal scheme to optimize adapter cost **regardless of** the kernel types and protocol functions, as the first two elements of any protocol cuboid are always (h, w). It is found that unifying the (h, w) pair can help to reduce the adapter cost more directly than trying to unify the single element c_x.

Our adapter cost optimization is conducted in a greedy manner. The reference pair $(h_{\text{ref}}, w_{\text{ref}})$ of each floor is selected from the existing (h, w) pairs on that floor. For a certain floor, all its possible reference pairs will be evaluated and the one leading to the best adapter cost will be committed. Assuming that there are n placed kernels in total, the worst case complexity of this optimization is bounded by $O(n^2)$. However, in practice, the optimization process is still extremely efficient (with negligible run time) since there are only thousand of kernels at most to be placed [1]. We evaluated the effectiveness of our adapter optimization technique on ISPD 2020 contest suites [1]. As shown in Table 1, the adapter cost can be reduced up to 87% for a single case, and the average improvement for all cases is 24% without compromising the maximum execution time and the wirelength.

Table 1: Benchmark Statistics of ISPD'20 Suite [1] and Performances Comparisons w/ and w/o Adapter Optimization.

	Benchmark Statistics [1]				W/O Adapter Opt.		W/ Adapter Opt.	
Case	# Krns	$\text{Weight}_{\text{MT}}$	$\text{Weight}_{\text{WL}}$	$\text{Weight}_{\text{AC}}$	AC†	Ratio	AC†	Ratio
A	17	1	1	0	15	1.00	15	1.00
B	34	1	1	0	18	1.00	18	1.00
C	102	1	1	0	234	1.00	**185**	**0.79**
D	54	1	1	0	139	1.00	**123**	**0.88**
E	17	1	10	100	11	1.00	11	1.00
F	34	1	10	100	13	1.00	**12**	**0.92**
G	102	1	10	100	221	1.00	**98**	**0.44**
H	54	1	10	100	77	1.00	**49**	**0.64**
I	27	1	4	0	13	1.00	13	1.00
J	81	1	4	0	193	1.00	**69**	**0.36**
K	18	1	4	0	9	1.00	**3**	**0.33**
L	54	1	4	0	140	1.00	**18**	**0.13**
M	25	1	4	0	41	1.00	41	1.00
N	28	1	4	0	10	1.00	10	1.00
O	27	1	40	400	13	1.00	13	1.00
P	81	1	40	400	154	1.00	**85**	**0.55**
Q	18	1	40	400	6	1.00	6	1.00
R	54	1	40	400	68	1.00	**20**	**0.29**
S	25	1	400	400	60	1.00	**48**	**0.80**
T	28	1	40	400	4	1.00	4	1.00
Avg.	-	-	-	-	71.95	1.00	**42.05**	**0.76**

† AC *denotes the adapter costs.*

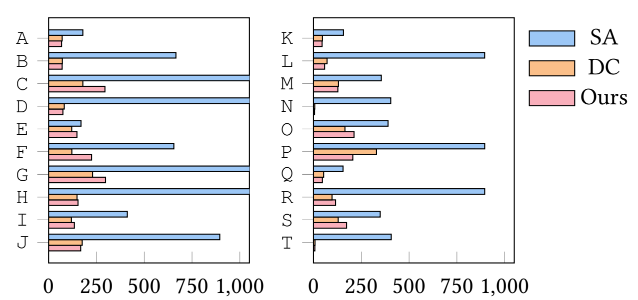


Figure 8: Runtime (seconds) comparisons on ISPD'20 suite with SA placement and DC placement.

Table 2: Comparison on ISPD20 Suite [1] with other contestants and two conventional heuristic approaches. Symbol '*' indicates being normalized by our corresponding score. Avg_h^* is the average normalized scores of the hidden cases I~T.

	2nd Place Team				3rd Place Team				SA Placement				DC Placement				Ours (1st Place Team)			
Case	MT	WL	AC	Score*	MT	WL	AC	Score*	MT	WL	AC	Score*	MT	WL	AC	Score*	MT	WL	AC	Score
A	35280	1186	15	1.02	35280	2047	13	1.04	37044	3611.5	11	1.14	35280	1565	12	1.03	34496	1314	15	**35810**
B	63504	3660	16	1.02	65856	4905	17	1.07	70560	6657	20	1.17	64512	3450	22	1.03	63504	2639.5	18	**66143.5**
C	64512	2471.5	217	1	65772	4278	281	1.05	76608	15696	69	1.38	63504	6308	163	1.04	64512	2408	185	**66920**
D	33712	2078.5	134	**0.98**	34944	3071.5	89	1.04	38304	9327.5	44	1.31	34048	6100.5	100	1.1	33712	2722	123	36434
E	39312	563	12	1	39690	590	16	1.03	36288	2080.5	7	1.26	35280	1565	12	1.13	39312	562	11	**46032**
F	66010	1489.5	18	1.02	70560	1475	14	1.07	76608	3237	15	1.36	65016	2650.5	18	1.15	65170	1489.5	12	**81265**
G	64512	2494.5	149	1.05	69888	2508	141	1.1	91728	7784	29	1.74	63504	6308	163	1.44	64512	2508.5	98	**99397**
H	39312	1033.5	60	1	43008	893	115	1.14	47040	4450	21	1.69	36400	2654	108	1.33	39520	1104.5	49	**55465**
I	49392	1288	13	1	52920	612	13	1.01	56448	3790	16	1.31	49392	1741.5	17	1.03	52136	617.5	13	**54606**
J	50176	1793	164	**0.95**	57792	1117.5	286	1.03	63504	8009.5	52	1.59	49392	4294	210	1.11	50274	2472.5	69	60164
K	252	423	7	1.4	504	400	14	1.51	576	236	3	1.09	828	267	10	1.36	432	240	3	**1392**
L	252	789	142	1.72	504	774	114	1.82	1280	910.5	60	2.49	1764	785	113	2.48	864	279	18	**1980**
M	2260992	3899	41	1.02	2336256	5100	67	1.06	2359296	9359	24	1.08	2276350	4313	58	1.03	2211840	3610.5	41	**2226282**
N	1651	437.5	8	1	1599	448.5	9	1	2268	707.5	0	1.5	1911	904	13	1.62	1482	480.5	10	**3404**
O	54720	614	19	1.06	52920	612	13	1.01	63504	1202	6	1.39	57624	649	12	1.08	52136	617.5	13	**82036**
P	60270	1096.5	78	1	66528	2273	102	1.46	115101	4015	24	2.1	63504	2519.5	134	1.6	57792	1101	85	**135832**
Q	252	423	7	2.01	504	400	14	2.23	1152	178	1	**0.87**	2898	171.5	8	1.31	882	166	6	9922
R	252	789	11	1.37	504	774	114	2.92	1372	1443	30	2.69	14112	480.5	53	2.06	8064	259	20	**26424**
S	2396160	1349	47	**0.93**	2396160	1899.5	65	1	2495376	3551	25	1.24	2276350	4313	58	1.27	2396300	1897.5	48	3174500
T	1651	437.5	8	1.59	2015	367.5	9	1.45	5720	555.5	0	1.99	6080	521.5	13	2.29	4102	208.5	4	**14042**
Avg_h^*				1.25				1.46				1.61				1.58				**1.00**
Avg^*				1.16				1.30				1.52				1.37				**1.00**

MT: maximum execution time; WL: wirelength; AC: adapter cost; Score: the weighted sum of MT, WL, and AC.

4 EXPERIMENTAL RESULTS

We implemented CU.POKer in `C++` programming language and evaluated it on the ISPD'20 contest benchmark suite [1]. Following the specification of the contest organizers, all evaluations are executed in single-thread on a 64-bit Linux machine with a 3.4GHz Intel Xeon CPU and 32GB RAM. The benchmarks statistics are listed in Table 1, where column "# Krns" lists the number of kernels to be placed, columns "$\text{Weight}_{\text{MT}}$", "$\text{Weight}_{\text{WL}}$", and "$\text{Weight}_{\text{AC}}$" denote the weights of the maximum execution time, wirelength and adapter cost. In Table 2, the score is compute as,

$$\text{Score} = \text{Weight}_{\text{MT}} * \text{MT} + \text{Weight}_{\text{WL}} * \text{WL} + \text{Weight}_{\text{AC}} * \text{AC}.$$

4.1 Comparisons with Other Contestants

Our proposed framework (CU.POKer) won the championship at ISPD'20 contest [1]. Quantitative results of top-3 teams are listed in Table 2. For fairness, all results presented in Table 2 are consistent with their/our `Final` submitted versions. In other words, the contest hidden cases I~T are totally blind for all the presented results. Comparing with these two teams, on average, our placement engine outperforms them by 16% and 30%, respectively. More importantly, on the hidden cases I~T, our placement engine can make an even greater improvement over them by 25% and 46%.

4.2 Comparisons with Conventional Heuristics

We further compare our flow with two conventional floorplanning heuristics to demonstrate the importance of customization.

4.2.1 Simulated Annealing. Simulated annealing (SA), is still the most commonly used heuristic in general floorplanning [10]. We implemented a SA-based placer (SA placement in Table 2) with the twin binary sequences representation (TBS) [11]. Compact packing is used to realize a layout from a given TBS. The annealing actions (with equal probabilities) are (1) pick up a new kernel candidate; (2) swap two kernels; (3) rotate the sequences to change the packing topology. However, it is found that SA-based placer is too general for this specific task in which the connections are mostly aligned data paths with some forks and each kernel block can have many choices of candidate shapes. In the experimental result, the SA-placer turns out yielding the worst quality.

4.2.2 Divide and Conquer. We have also engineered a divide-and-conquer heuristic (DC placement in Table 2) to compare with our method. For this heuristic, we will divide the kernel graph and the placement region into two recursively, such that the two kernel sub-graphs have similar total area, and the interconnections between them are as few as possible. At each level of division, we will place the two kernel sub-graphs recursively, and then merge them back to get the placement at this level. Note that, at the lowest level, there is only one kernel to be placed. This DC method is very fast but the quality is still far from the best. It is again because this DC method did not capture the specific features of this WSE placement task in which data path alignment is essential.

As depicted in Table 2, our algorithm outperforms the SA approach and the DC approach by 52% and 37%, respectively. Moreover, Figure 8 shows that our algorithm shares a comparable running time with respect to the DC approach, and is much more efficient than the SA approach. In conclusion, our customized algorithm demonstrates dominating superiority in comparison with these two general floorplanning heuristics.

5 CONCLUSION

In this work, we presented CU.POKer, a high-performance engine fully customized for the on-WSE DNN placement problem. Experimental results have verified its effectiveness. Future work would include dedicated parallelism for the proposed algorithms.

REFERENCES

[1] "ISPD 2020 Contest: Wafer-Scale Deep Learning Accelerator Placement," https://www.cerebras.net/ispd-2020-contest/.

[2] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," 2017.

[3] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[4] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Proc. MICCAI*, 2015, pp. 234–241.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[6] D. Amodei and D. Hernandez, "AI and Compute," https://openai.com/blog/ai-and-compute/, May 16, 2018, accessed on 20-May-2020.

[7] M. James, M. Tom, P. Groeneveld, and V. Kibardin, "Ispd 2020 physical mapping of neural networks on a wafer-scale deep learning accelerator," in *Proceedings of the 2020 International Symposium on Physical Design*, ser. ISPD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 145–149.

[8] Y. Lin, S. Dhar, W. Li, H. Ren, B. Khailany, and D. Z. Pan, "Dreamplace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[9] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, S. Bae *et al.*, "Chip placement with deep reinforcement learning," *arXiv preprint arXiv:2004.10746*, 2020.

[10] H.-M. Chen, M. D. Wong, H. Zhou, F.-Y. Young, H. H. Yang, and N. Sherwani, "Integrated floorplanning and interconnect planning," in *Layout optimization in VLSI design*. Springer, 2001, pp. 1–18.

[11] E. F. Young, C. C. Chu, and Z. C. Shen, "Twin binary sequences: a nonredundant representation for general nonslicing floorplan," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 4, pp. 457–469, 2003.