CU.POKer: Placing DNNs on WSE With Optimal Kernel Sizing and Efficient Protocol Optimization

Bentian Jiang, Jingsong Chen, Jinwei Liu, Lixin Liu, Fangzhou Wang, Xiaopeng Zhang, and Evangeline F. Y. Young, *Senior Member, IEEE*

***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 toward accelerating computation-intensive DL 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 DL accelerator, CS-1 wafer-scale engine (WSE). In this article, we proposed CU.POKer, a high-performance engine fully customized for WSE's deep neural network 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 (SOTA) quality on the real industrial benchmarks. Experimental results on ISPD 2020 contest evaluation suites demonstrated the superiority of our proposed framework over not only the SOTA placer but also the conventional heuristics used in general floorplanning.**

***Index Terms*—AI chip compilation, deep learning (DL) accelerator, neural network workload placement, wafer-scale engine (WSE).**

I. Introduction

APPLICATIONS of deep learning (DL) have risen in an eye-watering fashion. From the invincible go "player" AlphaZero [1], to the state-of-the-art (SOTA) NLP model GPT-3 [2], and the well-known U-Net [3], ResNet [4] 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 300000× and will continue to increase at a rate far beyond Moore's Law [5]. In response to such growing demands, both industry and academia have been actively exploring dedicated computer architectures specialized for DL. AI hardware, including but not limited to GPGPU, FPGA, and various dedicated ASICs, also shared a rapid evolution in the past decade.

Manuscript received January 3, 2021; revised April 30, 2021; accepted June 21, 2021. Date of publication July 12, 2021; date of current version May 20, 2022. This work was supported in part by the Research Grants Council of the Hong Kong SAR under Project CUHK14209320. The preliminary version has been presented at IEEE/ACM International Conference on Computer-Aided Design (ICCAD) in 2020. This article was recommended by Associate Editor I. H. R. Jiang. *(Bentian Jiang and Jingsong Chen contributed equally to this work.) (Corresponding author: Bentian Jiang.)*

The authors are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong (e-mail: btjiang@cse.cuhk.edu.hk).

Digital Object Identifier 10.1109/TCAD.2021.3096458

The generalized training procedure of the DNN 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 pretrained 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 (224 × 224) consumes 100 millions training steps while each step takes around 24 billion floating-point operations (FLOPs) [6]. 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 [5]. Targeting toward the acceleration of training models with millions or billions of parameters, industry recently released CS-1 Wafer-Scale Engine (WSE), the largest commercial DL chip with a novel customized architecture. Architecturally, WSE consists of over 400 000 programmable execution cores, while each of them possesses 48-KB 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 DL 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 [6].

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 [6]. The corresponding compilation stage of WSE is depicted in Fig. 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. The CAD tool is then used to place and route the constituent

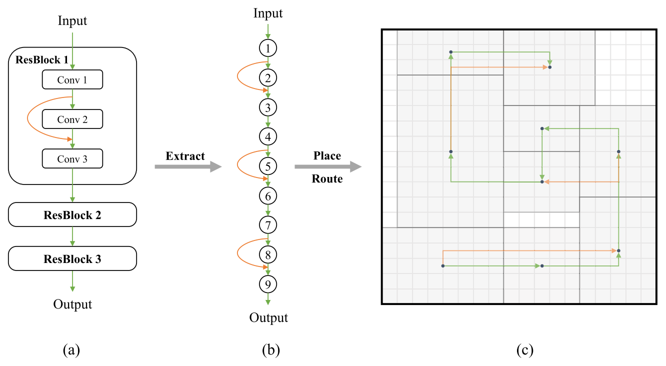


Fig. 1. Overview of WSE compilation flow, the proposed framework focuses on the placement stage of compilation. (a) Network Architecture. (b) Kernel Graph. (c) Execution Plan.

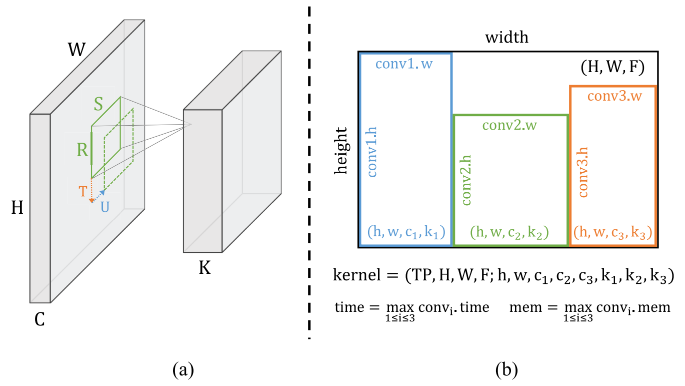


Fig. 2. Visualizations of the (a) arguments of a basic convolution and (b) performance of a kernel with three basic conv units.

kernels of a neural network on the computational fabric with certain objectives and constraints.

Our answer to the aforementioned challenge is to develop the 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 [7] for the first time casts the conventional nonlinear standard cell placement problem as a DNN training process without quality loss, while a recent work [8] 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 [9] 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 article, 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.

1) 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.
2) 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.
3) A universal scheme regardless of kernel types and protocol functions is proposed to optimize the adapter cost.
4) An interval query-based adapter cost optimization technique is developed. Different types of kernels and kernel graphs are handled accordingly.
5) We further compared the proposed flow with several conventional heuristics used in general floorplanning. The algorithmic details of the conventional heuristics are described, and experimental results show that CU.POKer outperforms not only all the contestants but also the conventional heuristics used in general floorplanning.

The remainder of this article is organized as follows. Section II lists some preliminaries. Section III discusses the details of the framework and algorithms. Section IV describes the algorithmic details of two conventional floorplanning heuristic baselines. Section V presents experimental results, followed by a conclusion in Section VI.

II. Preliminaries

During the WSE compilation, the layers of the input neural networks are decomposed into a set of kernels and each performs an individual DL task (e.g., to compute a “5 $\times$ 5 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.

A. 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 [6]. 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 [6]. It contains eight formal arguments (H, W, R, S, C, K, T, and U) and four execution arguments (h, w, c, and k). As illustrated in Fig. 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 strides of the convolution operation. The rest four execution arguments (h, w, c, and k) are the variables to be determined.

B. 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 [6].

1) Kernel Performance Function: Each kernel is provided with a unique performance function mapping the kernel arguments to a 4-tuple *performance cuboid* (height, width, time, and 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:

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

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 convs, its performance cuboid is given by

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

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, and U_i (for $i = 1, \ldots, n$) according to a look-up table, and $(h, w, c_1, \ldots, c_n, k_1, \ldots, k_n)$ are the execution variables to be optimized by the placer. Fig. 2(b) visualizes the physical shape of a kernel that contains three conv cores.

2) Kernel Protocol Function: 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 (two 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

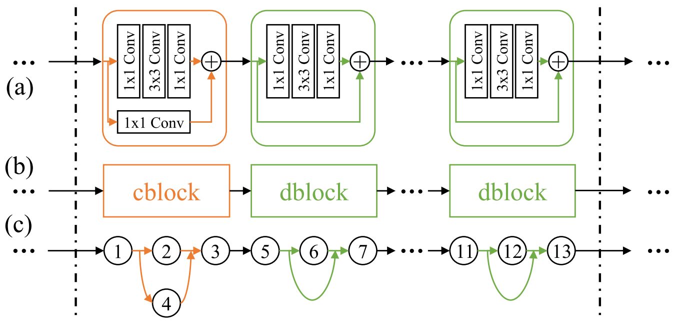


Fig. 3. Snapshot of a ResNet's kernel graph: (a) ResNet structure; (b) its coarse-grained kernel graph; and (c) its fine-grained kernel graph.

$$\text{ker}_1 = \{TP_1, H_1, W_1, F_1; h_1, w_1, c_{1,1}, \ldots, c_{1,n}, k_{1,1}, \ldots, k_{1,n}\}$$

be a predecessor of

$$\text{ker}_2 = \{TP_2, H_2, W_2, F_2; h_2, w_2, c_{2,1}, \ldots, c_{2,m}, k_{2,1}, \ldots, k_{2,m}\}$$

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[1]

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

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

C. Kernel Graph Granularity

For WSE placement, an input neural networks will be converted into a kernel-graph using a set of predetermined kernels in the kernel library. In the ISPD-2020 contest benchmarks, the predetermined kernels includes fine-grained block with only one convolution kernel (conv), and other coarse-grained blocks with multiple convolution kernels (e.g., *cblock* with four *convs*, and dblock with three convs). As a result, some specific network architectures can be implemented as either a fine-grained or a coarse-grained kernel graph.

Take the ResNet structure as an example, as illustrated in Fig. 3(a), ResNet is famous for its residual blocks which utilizing skip connections to avoid the vanishing gradients. In the ISPD20 benchmarks, the kernel graph of a ResNet can be implemented in the following.

1) Coarse-grained manner with predefined *cblock* and *dblock*. The kernel graph is a *directed single-path* graph [Fig. 3(b)], where each node in the kernel graph corresponds to a residual block in the ResNet [Fig. 3(a)].
2) Fine-grained manner with only convolution kernel. The kernel graph is a directed acyclic graph with multiple single-path and *fork structures* [Fig. 3(c)], where each

[1]Equation (3) is different from the protocol cost function in [6], since the contest organizers revised the function during the contest. In this article, we keep consistence with the final version used for the contest evaluation.

node in the kernel graph corresponds to a convolution layer in the ResNet [Fig. 3(a)].

The fine-grained kernel graph allows more flexible optimization during placement but at a cost of higher complexity comparing to the coarse-grained one.

D. Problem Formulation

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 fulfill the following constraints [6].

1) All kernels must fit within the fabric area (633 × 633 tiles).
2) No kernels may overlap.
3) 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.

1) The maximum execution time (in the performance cuboid) among all placed kernels.
2) The total L1 distance (wirelength) of all connected kernels.
3) 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.

III. 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 II-B1, 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 Fig. 4, 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 III-A. We will then detail our kernel candidate generation in Section III-B, data-path-aware placement in Section III-C, and post refinements for adapter cost in Sections III-D and III-E.

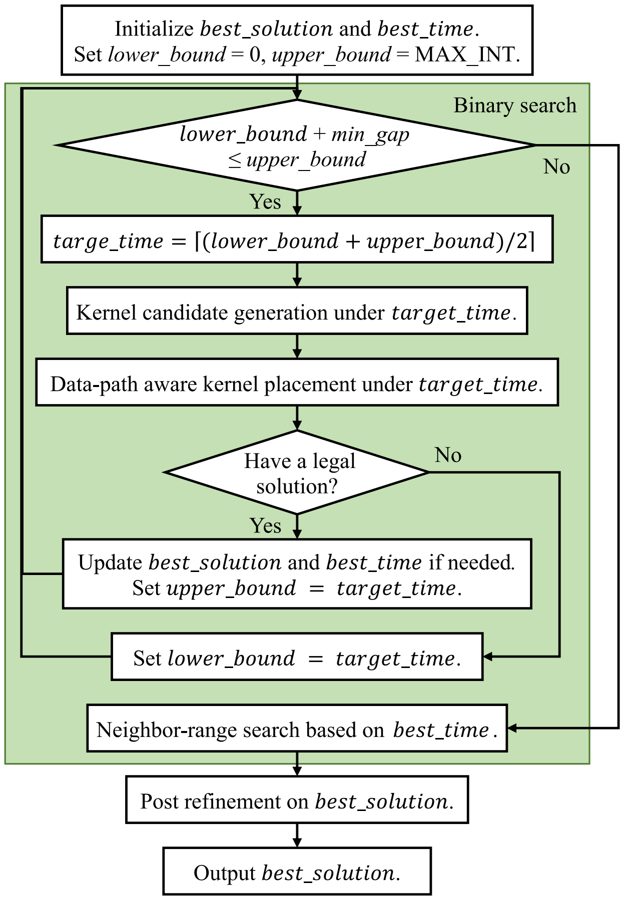

Initialize $best\_solution$ and $best\_time$.
Set $lower\_bound = 0$, $upper\_bound =$ MAX_INT.
Binary search
$lower\_bound + min\_gap \leq upper\_bound$
No
Yes
$targe\_time = \lceil (lower\_bound + upper\_bound)/2 \rceil$
Kernel candidate generation under $target\_time$.
Data-path aware kernel placement under $target\_time$.
Have a legal solution?
No
Yes
Update $best\_solution$ and $best\_time$ if needed.
Set $upper\_bound = target\_time$.
Set $lower\_bound = target\_time$.
Neighbor-range search based on $best\_time$.
Post refinement on $best\_solution$.
Output $best\_solution$.

Fig. 4. Overall flow.

A. Overall Flow

As finding the optimal placement 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 Fig. 4, 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 for checking whether a certain targeted time is feasible is not optimal (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 that 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.

B. Kernel Candidate 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, let us assume the kernels considered in this work contain x convolution cores. The performance (height, width, time, and memory) of each convolution core can be obtained using (1), while the kernel performance can be computed by (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).

Here, we refer to a kernel (height $= h_1$ and 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, at the same time having 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 four candidates as shown in Fig. 5 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 suboptimal 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 $c_1 = c_2 = \cdots = c_x = c$ will not sacrifice optimality by Theorem 1.

Theorem 1: For any argument $\{h, w, c_1, \ldots, c_x, k_1, \ldots, k_x\}$, there exist a $c = \max(c_1, \ldots, c_x)$ such that

$$\text{ker}_1 = \text{blockperf}(TP, H, W, F; h, w, c, \ldots, c, k_1, \ldots, k_x)$$

is no worse than

$$\text{ker}_2 = \text{blockperf}(TP, H, W, F; h, w, c_1, \ldots, c_x, k_1, \ldots, k_x)$$

with regard to height, width, time, and memory.

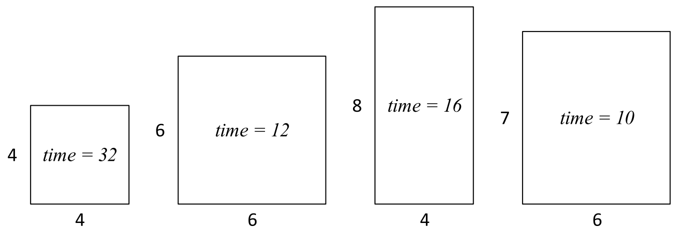


Fig. 5. Optimal and suboptimal kernel shapes.

Proof: Let $c = \max(c_1, \ldots, c_x)$, and all the four attributes will be either improved or unchanged. First, the height will be kept the same because,

$$\begin{aligned}\text{ker}_1.\text{height} &= h * w * (c+1)\\ &= \max_{1\leq j\leq x} h * w * (c_j + 1)\\ &= \text{ker}_2.\text{height}\end{aligned}$$

Second, the width will also be unchanged because its computation is irrelevant to $c_1, \ldots, c_x$. Finally, both the time and memory of the jth convolution core are inversely correlated to c_j according to (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 (2). ■

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

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

Note that all capital letters in above equations are constants. We will solve the optimization problem for each η by factorizing η to obtain the possible combinations of h, w, and c. For each possible combination of h, w, and c, we will compute the minimum $k_1, \ldots, k_x$ that satisfy both the target_time and the memory_limit constraints. More specifically we will factorize η into three integral terms, i.e., $\eta = \alpha_1 * \beta_1 * \gamma_1, \ldots, \eta = \alpha_n * \beta_n * \gamma_n$ (where $\gamma_i \geq 1$, for $i = 1, \ldots, n$). For each factorization, we will put $h = \alpha_i$, $w = \beta_i$, and $c = (\gamma_i - 1)$, and

Algorithm 1 Optimal Kernel Sizing Algorithm

```
1: function KERNELSIZING($TP$, $H$, $W$, $F$, $target\_time$)
2:     $res \leftarrow \{\}$;
3:     Decide $x$ according to $TP$;
4:     for $\eta = 1, ..., max\_height$ do
5:         Factorize $\eta$ to get $\eta = \alpha_1 * \beta_1 * \gamma_1, ..., \eta = \alpha_n * \beta_n * \gamma_n$;
6:         $best\_arg \leftarrow None$;
7:         $best\_width \leftarrow MAX\_NUM$;
8:         for $i = 1, ..., n$ do
9:             $h \leftarrow \alpha_1$, $w \leftarrow \beta_1$, $c \leftarrow (\gamma_1 - 1)$;
10:            for $j = 1, ..., x$ do
11:                $k_j \leftarrow$ compute with Equation (5);
12:            end for
13:            $ker_i \leftarrow blockperf(TP, H, W, F; h, w, c, k_1, ..., k_x)$;
14:            if $ker_i.width < best\_width$ then
15:                $best\_width \leftarrow ker_i.width$;
16:                $best\_arg \leftarrow \{h, w, c, k_1, ..., k_x\}$;
17:            end if
18:        end for
19:        if $best\_arg$ is not $None$ then
20:            Append $best\_arg$ to $res$;
21:        end if
22:    end for
23:    FinalPruning($res$);
24:    return $res$;
25: end function
```

find the values of $k_1, \ldots, k_x$ that lead to the minimum width. Since width is computed as three times the sum of $k_1, \ldots, k_x$, we can compute the minimum value of each k_j separately by

For j $= 1, \ldots, x$:

$$k_j^t = \text{ceil}\left(\text{ceil}\left(\frac{H_j}{h}\right)\text{ceil}\left(\frac{W_j}{w}\right)\text{ceil}\left(\frac{C_j}{c}\right)\frac{R_j S_j K_j}{T_j^2 * \text{target\_time}}\right)$$
$$k_j^m = \text{ceil}\left(\frac{C_j K_j R_j S_j}{c * \text{memory\_limit}} + \frac{(W_j + S_j - 1)(H_j + R_j - 1)K_j}{wh * \text{memory\_limit}}\right)$$
$$k_j = \max\left(k_j^t, k_j^m\right) \tag{5}$$

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 $\{h, w, c, k_1, \ldots, k_x\}$ for each η. The combination that achieves the minimum width for each η is kept.

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. Fig. 6 gives an example solution of our optimal kernel sizing algorithm, where the grey rectangles are the shapes obtained after solving the optimization problems in (4) for all the values of heights, and the red rectangles are the ones remained after the final pruning.

C. 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.

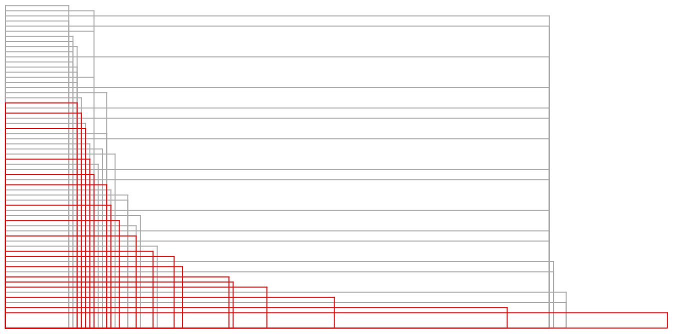
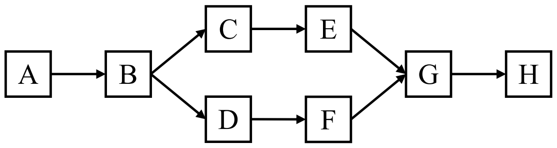


Fig. 6. Example solution of kernel sizing. For this specific case, we obtain 60 shapes after solving (4), but only 18 of them (the red rectangles) are optimal and output.



$$\begin{tikzcd} & C \arrow[r] & E \arrow[dr] & & \\ A \arrow[r] & B \arrow[ur] \arrow[dr] & & G \arrow[r] & H \\ & D \arrow[r] & F \arrow[ur] & & \end{tikzcd}$$

Fig. 7. 8-kernel connectivity graph.

The former step has generated all the kernel candidates with optimal shapes under the targeted time constraint. The task of the kernel placement is to select which candidate to use for each kernel and to determine the kernel locations. 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 Fig. 7 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., Fig. 8). 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 Fig. 4, the first and third parameters should be 1 and 0, respectively, which means the placement starts placing with the first kernel and from the bottom of the chip. Then the function `placement` will be called recursively to place multiple rows, one on top of the other (line 28 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 in H_k, for the kernel to be placed, only the candidates with height not exceeding h will be considered. In order to minimize wirelength, the candidate with the minimum width will be selected

Algorithm 2 Data-Path-Aware Kernel Placement

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

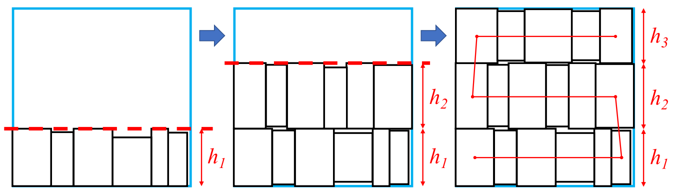


Fig. 8. Most placed kernels can not achieve the max floor height and thereby generate wasted deadspace.

(line 10). After placing one row, if all the kernels have been placed (line 24), 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 28).

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 $t_{\text{row}}+1$. 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.

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.

D. Adapter Cost Optimization for Arguments (h, w)

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 III-C).

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 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 in (3).

1) Wasted Deadspace: As illustrated in Fig. 8, a typical solution generated by our placer tries to arrange the kernels in a row-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 ith floor of the layout, for each kernel $\text{ker}_{i,j}$, $j \in \{1, \ldots, n\}$, we have

$$\text{ker}_{i,j}.\text{height} \le \text{floor}_i.\text{height} = \max_{1\le j\le n} \text{ker}_{i,j}.\text{height}.$$

If $\text{ker}_{i,j}.\text{height} < \text{floor}_i.\text{height}$, there exists wasted deadspace around $\text{ker}_{i,j}$ with $\Delta\text{height}_{i,j} = (\text{floor}_i.\text{height} - \text{ker}_{i,j}.\text{height})$ and $\text{width}_{i,j} = \text{ker}_{i,j}.\text{width}$. 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).

2) Unifying Arguments (h, w): W.l.o.g., lets assume $\text{ker}_{i,j}$, the jth kernel on the ith floor, contains m convolution cores (conv), and it is connected with $\text{ker}_{i,j-1}$ and $\text{ker}_{i,j+1}$, respectively. Its kernel height is given by

$$\text{ker}_{i,j}.\text{height} = h * w * (c_{\max} + 1) = \max_{1 \le j \le m} h * w * (c_j + 1).$$

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

$$c_{\max} = \text{floor}_i.\text{height}/(h * w) - 1. \tag{6}$$

Given $c_{\max}$ as in (6), a new assignment for $\text{ker}_{i,j}$'s arguments $(c_1, \ldots, c_m)$ can be

$$c_1 = \cdots = c_m = c_{\max} = \text{floor}_i.\text{height}/(h * w) - 1. \tag{7}$$

Note that after extending the height of $\text{ker}_{i,j}$ to $\text{floor}_i.\text{height}$ and subject to the requirement of having a given $(h,\ w)$ pair, the assignment of $(c_1, \ldots, c_m)$ according to (7) is optimal, by following a similar argument as in the proof of Theorem 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 $c_1 = \cdots = c_m = c_{\max} = \text{floor.height}/(h_{\text{ref}} * w_{\text{ref}}) - 1$;
3) check legality of the new kernel.

Recall that the input and output protocol cuboids (Section II-B2) of a kernel consist of three elements $(h,\ w,\ c_x)$, where the first two elements $(h,\ w)$ are identical for both cuboids, but c_x is selected from $(c_1, \ldots, c_m)$ 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 $(h,\ w)$. Our optimization on arguments $(h,\ w)$ 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.

E. Adapter Cost Optimization for Argument c

The previous step attempts to unify arguments $(h,\ w)$ with (7) in a row-based manner. In the following section, we will discuss how to further mitigate the adapter cost by modifying argument c in (3) using interval intersection queries, while not changing any major characteristic of the current solution (e.g., arguments h and w, max time, memory, wirelength, and placement layout). W.l.o.g., lets assume the input placement layout contains n kernels, each kernel contains m convolution cores (conv, and $m \ge 1$) with a execution argument (variable) set $\{h, w, c_1, \ldots, c_m, k_1, \ldots, k_m\}$.

1) Valid Intervals for Modifying Arguments c: According to the adapter cost function in (7), each kernel's first and last c arguments, i.e., $(c_1,\ c_m)$, are regarded as the input and output protocols, respectively. Our objective is to modify $(c_1,\ c_m)$ of each kernel to mitigate protocol differences in the kernel graph subject to not changing any major characteristic of the current solution. Therefore, an additional round of profiling is required to determine the valid changeable scopes for the arguments c_1 and c_m of every kernel.

Definition 1 (Valid Interval of Argument c): Given a placement solution, the valid interval of an argument c refers to an interval $l = [c_{\min},\ c_{\max}]$, where assigning the value of c within the interval l will not violate the max time, memory, and shape constraints of the solution.

Similar to optimization on argument $(h,\ w)$ (Section III-D), optimization on argument c also fully leverages the wasted dead space in each row of the layout. Therefore, the shape constraints mentioned in the above definition ensure that the height of any modified kernel does not exceed the corresponding row height, while the kernel width keeps unchanged.

For any kernel, a valid interval l_i of its argument c_i, $i \in \{1, m\}$, should satisfy

$$\begin{aligned}
\text{height} &= h * w * (c_i + 1) \\
&\le \text{row\_height} \\
\text{time} &= \text{ceil}\left(\frac{H_i}{h}\right)\text{ceil}\left(\frac{W_i}{w}\right)\text{ceil}\left(\frac{C_i}{c}\right)\text{ceil}\left(\frac{K_i}{k_i}\right)\frac{R_i S_i}{T_i^2} \\
&\le \text{maximum\_time} \\
\text{mem} &= \frac{C_i K_i R_i S_i}{c k_i} + \frac{(W_i + S_i - 1)(H_i + R_i - 1)K_i}{w h k_i} \\
&\le \text{memory\_limit}.
\end{aligned} \tag{8}$$

Solving (8), we can determine the interval l_i by

$$\begin{aligned}
c_i^h &= \text{floor}\left(\frac{\text{row\_height}}{wh} - 1\right) \\
c_i^t &= \text{ceil}\left(\text{ceil}\left(\frac{H_i}{h}\right)\text{ceil}\left(\frac{W_i}{w}\right)\text{ceil}\left(\frac{K_i}{k_i}\right)\frac{R_i S_i C_i}{T_i^2 * \text{maximum\_time}}\right) \\
c_i^m &= \text{ceil}\left(\frac{C_i K_i R_i S_i}{k_i * \left(\text{memory\_limit} - \frac{(W_i+S_i-1)(H_i+R_i-1)K_i}{whk_i}\right)}\right) \\
c_{i,\min} &= \max\left(c_i^t, c_i^m\right),\ c_{i,\max} = c_i^h,\ l_i = \left[c_{i,\min},\ c_{i,\max}\right].
\end{aligned}$$

Eventually, for a given kernel, an interval set L is given by $L = \{l_{\text{in}},\ l_{\text{out}}\} = \{l_1,\ l_m\}$, which corresponds to the valid intervals of its input and output protocols.

2) Interval Intersection Query on Single-Path Kernel Graph: As described in Section II-C, a kernel graph can be implemented in either coarse-grained or fine-grained manner, where coarse-grained kernel refers to predetermined blocks with multiple convolution cores, i.e., $m > 1$, $c_1 \ne c_m$; and fine-grained kernel refers to single convolution core, i.e., $m = 1$ and $c_1 = c_m$. In the contest benchmarks, a single-path kernel graph can be either fine-grained or coarse-grained (as

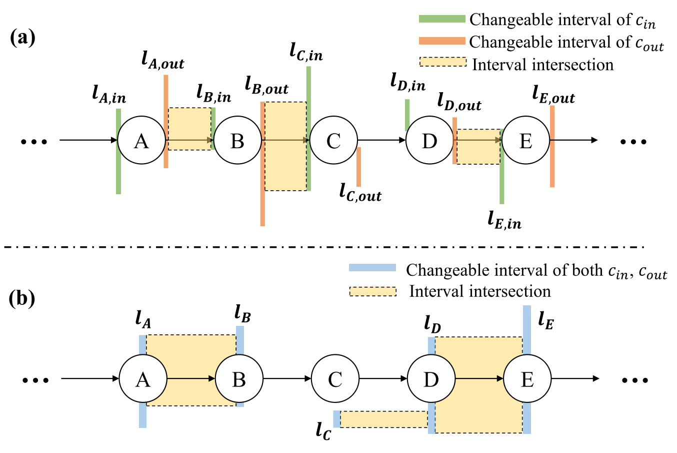


Fig. 9. Example of interval intersection query on: (a) coarse-grained single-path kernel graph ($m > 1,\ c_1 \neq c_m$, for each kernel) and (b) fine-grained single-path kernel graph ($m = 1,\ c_1 = c_m$, for each kernel).

shown in Fig. 9), while more complicated kernel graph with forks can only be a fine-grained kernel graph.

Once the profiling is done, the adapter cost optimization on argument c can be transformed into the interval intersection queries on the kernel graph edges. Lets first take the single-path kernel graph for illustration. Given the valid intervals for all kernels, the optimization for a coarse-grained kernel graph [see Fig. 9(a)] is equivalent to maximizing the following objective (cumulative unified c protocols)

$$\mathrm{cost}(v) = \mathrm{cost}(u) + \max_{c_{u,\mathrm{out}} \in l_{u,\mathrm{out}},\ c_{v,\mathrm{in}} \in l_{u,\mathrm{in}}} \mathrm{edge\_cost}(u, v) \quad (9)$$

where $(u,\ v)$ are two connected kernels in the single-path kernel graph, and the corresponding edge cost is given by

$$\mathrm{edge\_cost}(u, v) = \begin{cases} 1, & \text{if } l_{u,\mathrm{out}} \bigcap l_{v,in} \neq \emptyset \\ 0, & \text{if } l_{u,\mathrm{out}} \bigcap l_{v,in} = \emptyset. \end{cases} \quad (10)$$

Apparently, if there exists an nonempty intersection $l_{\mathrm{uv}} = l_{u,\mathrm{out}} \bigcap l_{v,\mathrm{in}}$ on edge $(u,\ v)$, we can unify the arguments $c_{u,\mathrm{out}}$ and $c_{v,\mathrm{in}}$ by assigning them the maximum value of l_{uv}, thereby reducing the adapter cost.

Note that, when dealing with fine-grained single-path kernel graph, the final intersection of a kernel is coupled with both its input and output edges [e.g., the intersection of l_C, l_D, and l_E for kernel D in Fig. 9(b)], as there is only one convolution core per kernel ($m = 1$). If the input and output intersections of a fine-grained kernel are conflicted, the input intersection will be kept. Note that such assignment method is optimal for both coarse-grained and fine-grained single-path kernel graphs (within the feasible modification scopes) due to its single-path property. It takes only O(n) time to finish the optimization on a single-path kernel graph with n kernels, where O(1) for interval calculation per kernel, and O(1) for intersection query per edge.

3) Handle Fine-Grained Kernel Graph With Basic Forks: Apart from the single-path kernel graph, some networks in the ISPD20 contest are implemented as fine-grained kernel graph with different fork structures (e.g., ResNet-50, ResNet-75, UNet, etc.). We additionally consider the basic fork structures (i.e., forks without nested relation) in our adapter cost optimization process to cover most of the benchmarks provided. Given an input fine-grained kernel graph with only basic forks, the optimization on argument c consists of four steps.

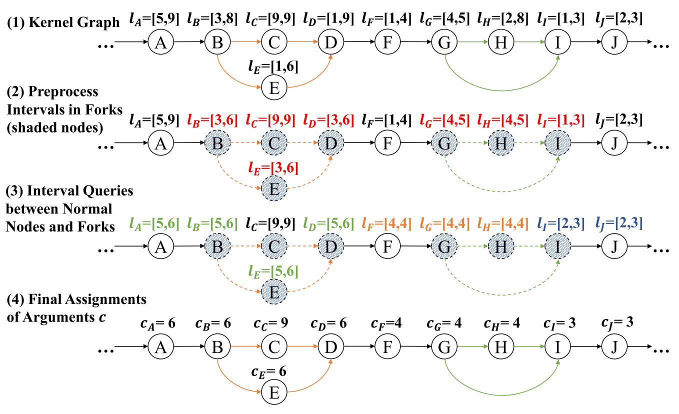

(1) Kernel Graph
l_A=[5,9] l_B=[3,8] l_C=[9,9] l_D=[1,9] l_F=[1,4] l_G=[4,5] l_H=[2,8] l_I=[1,3] l_J=[2,3]
l_E=[1,6]
(2) Preprocess Intervals in Forks (shaded nodes)
l_A=[5,9] l_B=[3,6] l_C=[9,9] l_D=[3,6] l_F=[1,4] l_G=[4,5] l_H=[4,5] l_I=[1,3] l_J=[2,3]
l_E=[3,6]
(3) Interval Queries between Normal Nodes and Forks
l_A=[5,6] l_B=[5,6] l_C=[9,9] l_D=[5,6] l_F=[4,4] l_G=[4,4] l_H=[4,4] l_I=[2,3] l_J=[2,3]
l_E=[5,6]
(4) Final Assignments of Arguments c
c_A= 6 c_B= 6 c_C= 9 c_D= 6 c_F=4 c_G= 4 c_H= 4 c_I= 3 c_J= 3
c_E= 6
A B C D E F G H I J

Fig. 10. Flow of handling fine-grained kernel graph with basic forks.

TABLE I
BENCHMARK STATISTICS OF ISPD'20 SUITE [9] AND PERFORMANCE COMPARISONS W/ AND W/O ADAPTER OPTIMIZATION

	Benchmark Stat.				W/O Adapter Opt.	Opt. $(h,\ w)$		Opt. $(h,\ w,\ c)$	
Case	#Krns	$\mathrm{W_{MT}}$	$\mathrm{W_{WL}}$	$\mathrm{W_{AC}}$	$\mathrm{Cost_{Apt.}}$	$\mathrm{Cost_{Apt.}}$	Ratio	$\mathrm{Cost_{Apt.}}$	Ratio
A	17	1	1	0	15	15	1	15	1
B	34	1	1	0	18	18	1	18	1
C	102	1	1	0	234	185	0.79	**150**	**0.64**
D	54	1	1	0	139	123	0.88	**118**	**0.85**
E	17	1	10	100	11	11	1	11	1
F	34	1	10	100	13	**12**	**0.92**	**12**	**0.92**
G	102	1	10	100	221	98	0.44	**59**	**0.27**
H	54	1	10	100	77	49	0.64	**28**	**0.36**
I	27	1	4	0	13	13	1	13	1
J	81	1	4	0	193	**69**	**0.36**	**69**	**0.36**
K	18	1	4	0	9	**3**	**0.33**	**3**	**0.33**
L	54	1	4	0	140	18	0.13	**13**	**0.09**
M	25	1	4	0	41	41	1	41	1
N	28	1	4	0	10	10	1	**8**	**0.80**
O	27	1	40	400	13	13	1	13	1
P	81	1	40	400	154	85	0.55	**63**	**0.41**
Q	18	1	40	400	6	6	1	**5**	**0.83**
R	54	1	40	400	68	**20**	**0.29**	**20**	**0.29**
S	25	1	400	400	60	**48**	**0.80**	**48**	**0.80**
T	28	1	40	400	4	4	1	**1**	**0.25**
Avg.	-	-	-	-	71.95	42.05	0.76	**35.4**	**0.66**

$\mathrm{W_{MT}}$: weight of max time; $\mathrm{W_{WL}}$: weight of wirelength; $\mathrm{W_{MT}}$: weight of adapter cost.

1) Profile valid intervals l for all kernels.
2) Extract and preprocess the forks (clusters of shaded nodes in Fig. 10). For each fork, find and update the best interval of every shaded node inside, so as to maximize the total number of nonempty interval intersections on the fork edges (the dotted edges in Fig. 10).
3) Perform interval intersection queries between any connected normal node and fork, update the new intervals of the normal node and fork nodes accordingly.
4) Determine the final assignment of c for each node.

Fig. 10 (1)–(4) illustrate the detail procedures for above adapter cost optimization.

We evaluated the effectiveness of our adapter optimization techniques on ISPD 2020 contest suites [9]. As shown in Table I, comparing to the original input placement solution, we achieve 24% adapter cost improvement on average after optimizing arguments $(h,\ w)$, and obtain additional 10% improvement on average after optimizing argument c, that is, in total, 34% on average after optimizing arguments $(h,\ w,\ c)$. Such adapter cost optimization takes only negligible runtime and does not degrade the maximum execution time and wirelength of the solution.

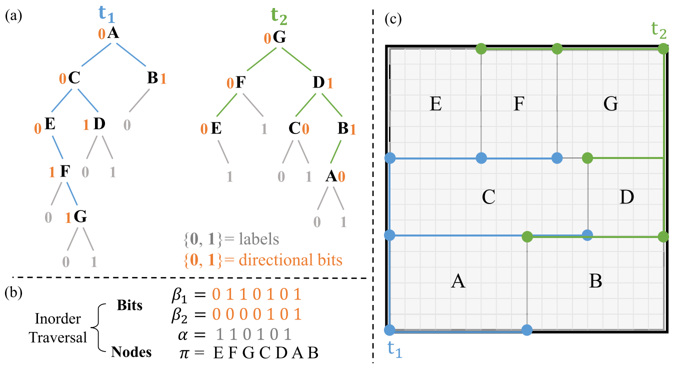


Fig. 11. Example of TBS: (a) A pair of twin binary trees (TBT = $\{t_1, t_2\}$); (b) its twin binary sequences representation (TBS = $\{\pi, \alpha, \beta_1, \beta_2\}$); and (c) its corresponding floorplan.

Algorithm 3 Divide-And-Conquer Heuristic

```
1:  function DNCPlace($beg$, $end$)
2:      if $beg = end$ then
3:          return all possible shapes of kernel $k_{\mathrm{beg}}$
4:      end if
5:      Find $mid$ such that the estimated sum of areas of kernels $k_{\mathrm{beg}}, \dots, k_{mid}$ and kernels $k_{mid}, \dots, k_{\mathrm{end}}$ are as close as possible.
6:      $rectSet_1 \leftarrow$ DNCPlace($beg$, $mid$)
7:      $rectSet_2 \leftarrow$ DNCPlace($mid$, $end$)
8:      $newRects \leftarrow$ all rectangular bounding boxes can be acquired by combining one shape from $rectSet_1$ and one shape from $rectSet_2$
9:      Prune $newRects$ to remove redundant or inferior shapes
10:     return $newRects$
11: end function
```

IV. Conventional Floorplanning Baselines

Intuitively, the WSE placement task is reminiscent of the traditional floorplanning problem in physical design. Therefore, it is worth to investigate how good those commonly used floorplanning heuristics are in WSE placement, which offers an important baseline for CU.POKer. Motivated by this, two placers are further developed based on the most commonly used floorplanning heuristics: 1) simulated annealing and 2) divide-and-conquer.

A. Simulated Annealing-Based Floorplanning

Finding optimal floorplan is NP-hard, many mainstream floorplaners employ methods of perturbations with random searches and meta-heuristics [10], [11]. Obviously, the searching efficiency depends very much on the efficiency of the floorplan representation, as it determines the size of the search space and the complexity of realization from the representation to the floorplan. In our implementation, we use twin binary sequences (TBS) [12] representation to conduct SA searching. TBS representation is an efficient nonredundant representation for general floorplan. It converts a pair of twin binary trees $\{t_1, t_2\}$ [Fig. 11(a)], which is originally proposed for the mosaic flooplan, into a tuple of sequences $\{\pi, \alpha, \beta_1, \beta_2\}$ [Fig. 11(b)], and further extends this representation for general flooplan by inserting irreducible dummy blocks into the TBS [12].

As shown in Fig. 11(a), a pair of twin binary trees TBT = $\{t_1, t_2\}$ can uniquely determine a (mosaic) floorplan, where t_1 specifies the topology by visiting bottom-left corners of all the blocks, and t_2 specifies the topology by visiting upper-right corners of all the blocks. Leaves of $\{t_1, t_2\}$ are labeled by either bit "0" or bit "1," where the p$^{\text{th}}$ label "0" ("1") in t_1 corresponds to the pth vertical (horizontal) cutline in the floorplan, and t_2's labels are complementary to t_1's labels. TBT takes only linear time for TBT-to-floorplan transformation, the algorithmic details can be found in [13].

However, maintaining TBT structures during perturbations is inefficient/complicated for real implementation. Observing the facts that a pair of TBT $\{t_1, t_2\}$ always share same inorder traversal and complementary labels, high-storage-efficient and manipulation-friendly implementation is feasible. Here, we encode TBT to TBS, which consists of a sequence tuple $\{\pi, \alpha, \beta_1, \beta_2\}$. In TBS, π is an inorder traversal of the two trees and α is the label of t_1; β_1 and β_2 are two sets of auxiliary directional bits for t_1 and t_2, which help to identify the one-to-one mapping between TBS and TBT (only labels and inorder traversal are not sufficient to identify a unique pair of $\{t_1, t_2\}$).

Same as TBT, TBS takes only linear time (lower bound for packing) for TBS-to-floorplan transformation, and achieves p-admissible solution space for searching. More importantly, TBS supports general flooplan searching by inserting the exact number of irreducible dummy blocks (refer to [12] for details), where in our implementation, at most 100 dummy blocks are allowed. There are three types of perturbations (with equal probabilities) in our SA placer.

1) *Sizing:* Pick up a new candidate for a kernel in π.
2) *Swapping:* Exchange two kernels in π.
3) *Rotation:* Rotate t_1 or t_2 to change the packing topology (operating on corresponding $\{\pi, \alpha, \beta_1, \beta_2\}$).

where the candidates used for the sizing action are generated by similar algorithm described in Section III-B, but without specifying target time constraint.

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 unsatisfactory quality.

B. Divide-and-Conquer-Based Floorplanning

We have also tried to develop a kernel placement heuristic adopting a divide-and-conquer strategy. To perform the heuristic, we will divide the kernels into two groups recursively, and place each of the groups separately before merging them back to get the placement for both groups.

More specifically, we will first obtain a topological sort of the kernels, $k_1, k_2, \dots, k_n$, so as to have a rough idea of which kernels should be placed closer to each other. Next, we will call function DNCPlace$(1, n)$ in Algorithm 3 to place the n kernels. DNCPlace(beg, end) will try to place kernels $k_{\mathrm{beg}}, \dots, k_{\mathrm{end}}$ inside a compact rectangular bounding box, and return all possible rectangular bounding boxes after placement. If $beg = end$, the function will simply return the possible shapes of the kernel; if $beg \neq end$, it will try to evenly divide the kernels into two groups according to the estimated area. Each group will be placed into rectangular bounding boxes by recursive

TABLE II
COMPARISON ON ISPD20 SUITE [9] WITH SOTA PLACER AND OTHER CONTESTANTS. SYMBOL "*" INDICATES BEING NORMALIZED BY OUR CORRESPONDING SCORE. AVG$_h^*$ IS THE AVERAGE NORMALIZED SCORE OF THE HIDDEN CASES I~T. AVG$_{rt}$ IS THE AVERAGE RUNTIME (SECONDS) OF ALL THE CASES

	1st Place Team [14] (SOTA)				2nd Place Team				3rd Place Team				CU.POKer			
Case	MT	WL	AC	Score*	MT	WL	AC	Score*	MT	WL	AC	Score*	MT	WL	AC	Score
A	34496	1314	15	1.00	35280	1186	15	1.02	35280	2047	13	1.04	34496	1314	15	**35810**
B	63504	2639.5	18	1.00	63504	3660	16	1.02	65856	4905	17	1.07	63504	2639.5	18	**66143.5**
C	64512	2408	185	1.00	64512	2471.5	217	1.00	65772	4278	281	1.05	64512	2407.5	150	**66919.5**
D	33712	2722	123	1.00	33712	2078.5	134	**0.98**	34944	3071.5	89	1.04	33712	2726	118	36438
E	39312	562	11	1.00	39312	563	12	1.00	39690	590	16	1.03	39312	562	11	**46032**
F	65170	1489.5	12	1.00	66010	1489.5	18	1.02	70560	1475	14	1.07	65170	1488.5	12	**81255**
G	64512	2508.5	98	1.04	64512	2494.5	149	1.10	69888	2508	141	1.14	64512	2506	59	**95472**
H	39520	1104.5	49	1.04	39312	1033.5	60	1.04	43008	893	115	1.19	39520	1101.5	28	**53335**
I	52136	617.5	13	1.00	49392	1288	13	**1.00**	52920	612	13	1.01	52136	617.5	13	54606
J	50274	2472.5	69	1.00	50176	1793	164	**0.95**	57792	1117.5	286	1.03	50274	2472.5	69	60164
K	432	240	3	1.00	252	423	7	1.40	504	400	14	1.51	432	240	3	**1392**
L	864	279	18	1.00	252	789	142	1.72	504	774	114	1.82	864	279	13	**1980**
M	2211840	3610.5	41	1.00	2260992	3899	41	1.02	2336256	5100	67	1.06	2211840	3610.5	41	**2226282**
N	1482	480.5	10	1.00	1651	437.5	8	1.00	1599	448.5	9	**1.00**	1482	479.5	8	3400
O	52136	617.5	13	1.00	54720	614	19	1.06	52920	612	13	1.01	52136	617.5	13	**82036**
P	57792	1101	85	1.07	60270	1096.5	78	1.06	66528	2273	102	1.56	57792	1109.5	63	**127372**
Q	882	166	6	1.04	252	423	7	2.10	504	400	14	2.32	882	166	5	**9522**
R	8064	259	20	1.00	252	789	11	1.37	504	774	114	2.92	8064	259	20	**26424**
S	2396300	1897.5	48	1.00	2396160	1349	47	**0.93**	2396160	1899.5	65	1.00	2396300	1897.5	48	3174500
T	4102	208.5	4	1.09	1651	437.5	8	1.74	2015	367.5	9	1.58	4102	208.5	1	**12842**
Avg$_h^*$				1.02				1.28				1.49				**1.00**
Avg*				1.01				1.18				1.32				**1.00**
Avg$_{rt}$				131.6				32.1				**7.3**				54.7
Avg$_{rt}^*$				2.41				0.59				**0.13**				1.00

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

calls. Finally, the function will enumerate the solutions of each group, and enumerate all possible ways of combining them to get the possible rectangular bounding boxes that can fit both groups. Some prunings are done to remove redundant or inferior boxes.

The divide-and-conquer heuristic is relatively fast, but it does not fully capture the inner structure of this WSE placement task, in which data path alignment is critical.

V. EXPERIMENTAL RESULTS

We implemented CU.POKer in C++ programming language and evaluated it on the ISPD'20 contest benchmark suite [9]. All evaluations are executed on a 64-bit Linux machine with two 2.2-GHz Intel Xeon CPUs (ten cores) and 256-GB RAM. The benchmarks statistics are listed in Table I, where column "# Krns" lists the number of kernels to be placed, columns "W_{MT}," "W_{WL}," and "W_{AC}" denote the weights of the maximum execution time, wirelength and adapter cost. In all the experiments, the score is compute as

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

A. Comparisons With SOTA Placer and Other Contestants

In this section, we compare our proposed framework (CU.POKer) with the SOTA WSE placer [14] and other contestants in the ISPD'20 contest to verify its effectiveness. Quantitative results of top-3 teams are listed in Table II. Comparing with the SOTA placer and the other two teams, on average, our placement engine outperforms them by 1%, 18%, and 32%, respectively. More importantly, on the hidden cases

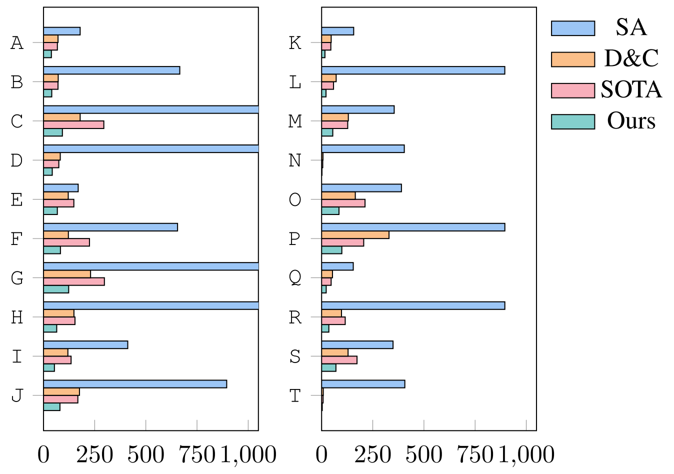


Fig. 12. Runtime (seconds) comparisons on ISPD'20 suite with SA placer, D&C placer and the SOTA [14] placer.

I~T, our placement engine can make an even greater improvement over them by 2%, 28%, and 49%. For the overall runtime, comparing with the other methods, our placer consumes a reasonable average runtime while achieving significantly better quality of results. As shown in Fig. 12, our placer also achieves much better runtime performance comparing to the SOTA placer [14].

B. Comparisons With Conventional Heuristics

We further compare our flow with two conventional floorplanning heuristics to demonstrate the importance of customization. Note that our TBS-based simulated annealing placer (TBS-SA) and divide-and-conquer placer (D&C) are

TABLE III
COMPARISON ON ISPD20 SUITE [9] WITH CONVENTIONAL FLOORPLAN HEURISTICS. SYMBOL "*" INDICATES BEING NORMALIZED BY OUR CORRESPONDING SCORE. AVG_h^* IS THE AVERAGE NORMALIZED SCORES OF THE HIDDEN CASES I∼T. AVG_{rt} IS THE AVERAGE RUNTIME (SECONDS) OF ALL THE CASES

	TBS-Simulated Annealing Placement				Divide&Conquer Placement				CU.POKer			
Case	MT	WL	AC	Score*	MT	WL	AC	Score*	MT	WL	AC	Score
A	37044	3611.5	11	1.14	35280	1565	12	1.03	34496	1314	15	**35810**
B	70560	6657	20	1.17	64512	3450	22	1.03	63504	2639.5	18	**66143.5**
C	76608	15696	69	1.38	63504	6308	163	1.04	64512	2407.5	150	**66919.5**
D	38304	9327.5	44	1.31	34048	6100.5	100	1.10	33712	2726	118	**36438**
E	36288	2080.5	7	1.26	35280	1565	12	1.13	39312	562	11	**46032**
F	76608	3237	15	1.36	65016	2650.5	18	1.15	65170	1488.5	12	**81255**
G	91728	7784	29	1.81	63504	6308	163	1.50	64512	2506	59	**95472**
H	47040	4450	21	1.76	36400	2654	108	1.38	39520	1101.5	28	**53335**
I	56448	3790	16	1.31	49392	1741.5	17	1.03	52136	617.5	13	**54606**
J	63504	8009.5	52	1.59	49392	4294	210	1.11	50274	2472.5	69	**60164**
K	576	236	3	1.09	828	267	10	1.36	432	240	3	**1392**
L	1280	910.5	60	2.49	1764	785	113	2.48	864	279	13	**1980**
M	2359296	9359	24	1.08	2276350	4313	58	1.03	2211840	3610.5	41	**2226282**
N	2268	707.5	0	1.50	1911	904	13	1.63	1482	479.5	8	**3400**
O	63504	1202	6	1.39	57624	649	12	1.08	52136	617.5	13	**82036**
P	115101	4015	24	2.24	63504	2519.5	134	1.71	57792	1109.5	63	**127372**
Q	1152	178	1	**0.91**	2898	171.5	8	1.36	882	166	5	9522
R	1372	1443	30	2.69	14112	480.5	53	2.06	8064	259	20	**26424**
S	2495376	3551	25	1.24	2276350	4313	58	1.27	2396300	1897.5	48	**3174500**
T	5720	555.5	0	2.18	6080	521.5	13	2.50	4102	208.5	1	**12842**
Avg_h^*				1.64				1.55				**1.00**
Avg^*				1.54				1.40				**1.00**
Avg_{rt}				747.2				117.8				**54.7**
Avg_{rt}^*				13.66				2.15				**1.00**

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

TABLE IV
RUNTIME (SECONDS) COMPARISONS BETWEEN ONE THREAD AND EIGHT THREADS

Case	1-thread	8-threads	Ratio(%)	Case	1-thread	8-threads	Ratio(%)
A	73.68	38.46	52.21	K	49.16	16.29	33.14
B	82.29	40.28	48.94	L	62.04	21.52	34.68
C	231.77	92.28	39.81	M	134.54	54.55	40.55
D	82.04	42.77	52.13	N	6.65	1.87	28.09
E	162.13	67.54	41.66	O	237.78	84.93	35.72
F	237.84	82.76	34.80	P	205.89	98.83	48.00
G	293.83	122.93	41.84	Q	54.16	22.46	41.46
H	140.14	64.49	46.02	R	116.39	35.38	30.40
I	136.32	53.30	39.10	S	180.47	70.03	38.81
J	185.09	80.29	43.38	T	7.67	3.15	41.02

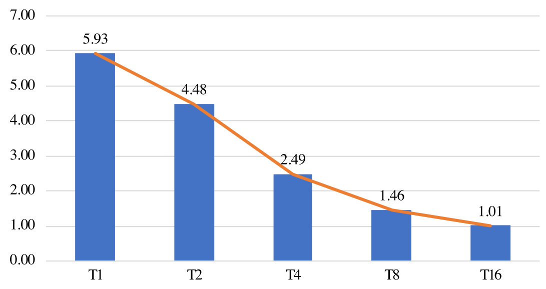


Fig. 13. Ratios between the runtime of neighbor-range search and that of the remaining part with different numbers of threads (e.g., T1 = 1 thread).

carefully tuned for this WSE placement problem. These two placers outperform the best SA placer (B* tree-based SA, fourth-place team) among all ISPD'20 contestants [9] by over 11% and 32% on the public benchmarks (case A∼H), which verified the effectiveness of our implementations.

As depicted in Table III, our algorithm outperforms the simulated annealing approach and the divide-and-conquer approach by 54% and 40%, respectively. Moreover, Fig. 12 shows that our algorithm is much more efficient than both TBS-SA and D&C approaches. In conclusion, our customized algorithm demonstrates dominating superiority in comparison with these two general floorplanning heuristics.

C. Multithreading

In the overall flow discussed in Section III-A, neighbor-range search is the most time-consuming step which conducts a fined-grained search on the timeline to further improve the solution obtained after binary search. On average, this step contributes 83.79% of the whole-process runtime. It should be executed in parallel for effective runtime reduction. To this end, all the target execution times to be tried during the neighbor-range search will be evenly distributed to each thread. Table IV shows that the applied 8-thread parallelization can significantly reduce the whole-process runtime by 59.41% on average, which corresponds to 2.53× runtime speedup.

In addition, the neighbor-range search is quite suitable to be executed in parallel due to the fact that no intercommunication is needed between different threads. Good scalability on the number of threads is shown by Fig. 13. From 1 thread to 16 threads, the average ratio over all the cases between the runtime of neighbor-range search and that of the remaining part is stably decreased from 5.93 to 1.01.

VI. Conclusion

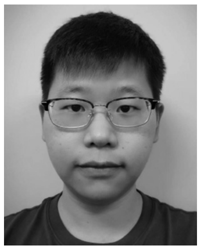
In this work, we proposed CU.POKer, a high-performance engine fully customized for the on-WSE DNN placement problem, with efficient implementations. In the CU.POKer, we designed two collaborative schemes for optimizing the maximum execution time and wirelength: a optimal placeable kernel candidate searching scheme, as well as a data-flow-aware placement scheme. A series of efficient adapter cost optimization techniques and a parallelism scheme are also developed to further improve the solution quality and flow turn around time. Besides, we also systematically investigated the performances of conventional floorplanning heuristics on the WSE placement problem, which further verified the superiority of our proposed flow. Experimental results show that our CU.POKer has achieved the SOTA quality on the real industrial benchmarks. Future works would include the explorations on simultaneously partitioning, placement, and routing on the WSE with various computational intensive tasks such as mapping a 3-D finite element model for solving the physic modeling.

Acknowledgment

The authors would like to thank Dr. Gengjie Chen for providing helpful discussions.

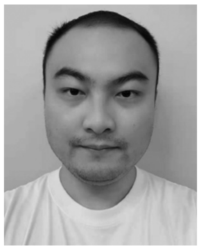
References

[1] D. Silver *et al.*, “Mastering chess and shogi by self-play with a general reinforcement learning algorithm,” 2017. [Online]. Available: http://arxiv.org/abs/1712.01815.
[2] T. B. Brown *et al.*, “Language models are few-shot learners,” 2020. [Online]. Available: arXiv:2005.14165.
[3] O. Ronneberger, P. Fischer, and T. Brox, “U-Net: Convolutional networks for biomedical image segmentation,” in *Proc. Int. Conf. Med. Image Comput. Comput. Assist. Intervention*, 2015, pp. 234–241.
[4] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Las Vegas, NV, USA, 2016, pp. 770–778.
[5] D. Amodei and D. Hernandez. (May 2018). *AI and Compute*. Accessed: May 20, 2020. [Online]. Available: https://openai.com/blog/ai-and-compute/
[6] M. James, M. Tom, P. Groeneveld, and V. Kibardin, “ISPD 2020 physical mapping of neural networks on a wafer-scale deep learning accelerator,” in *Proc. Int. Symp. Phys. Design*, 2020, pp. 145–149.
[7] 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 *Proc. 56th Annu. Design Autom. Conf.*, Las Vegas, NV, USA, 2019, pp. 1–6.
[8] A. Mirhoseini *et al.*, “Chip placement with deep reinforcement learning,” 2020. [Online]. Available: arXiv:2004.10746.
[9] (2020). *ISPD 2020 Contest: Wafer-Scale Deep Learning Accelerator Placement*. [Online]. Available: https://www.cerebras.net/ispd-2020-contest/
[10] R. B. Singh, A. S. Baghel, and A. Agarwal, “A review on VLSI floorplanning optimization using metaheuristic algorithms,” in *Proc. Int. Conf. Elect. Electron. Optim. Techn. (ICEEOT)*, Chennai, India, 2016, pp. 4198–4202.
[11] H.-M. Chen, M. D. F. Wong, H. Zhou, F.-Y. Young, H. H. Yang, and N. Sherwani, “Integrated floorplanning and interconnect planning,” in *Layout Optimization in VLSI Design*. Boston, MA, USA: Springer, 2001, pp. 1–18.
[12] E. F. Y. Young, C. C. N. Chu, and Z. C. Shen, “Twin binary sequences: A nonredundant representation for general nonslicing floorplan,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 22, no. 4, pp. 457–469, Apr. 2003.
[13] B. Yao, H. Chen, C.-K. Cheng, and R. Graham, “Floorplan representations: Complexity and connections,” *ACM Trans. Design Autom. Electron. Syst.*, vol. 8, no. 1, pp. 55–80, 2003.
[14] B. Jiang *et al.*, “CU.POKer: Placing DNNs on wafer-scale AI accelerator with optimal kernel sizing,” in *Proc. 39th Int. Conf. Comput.-Aided Design*, San Diego, CA, USA, 2020, pp. 1–9.



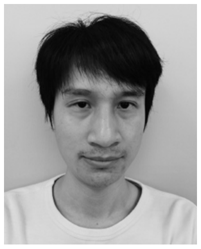
Bentian Jiang received the B.Eng. degree in electronics and information engineering from Sichuan University, Chengdu, China, in 2017. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong.

His research interests include VLSI design for manufacturability and physical design.



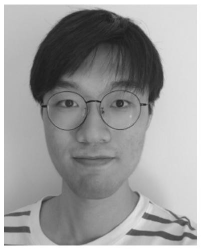
Jingsong Chen received the B.Eng. degree in computer science and technology from Zhejiang University, Hangzhou, China, in 2017. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong.

His current research interests include physical design of VLSI circuits and related ML-based problems.



Jinwei Liu received the B.Sc. degree in computer science and technology from Sichuan University, Chengdu, China, in 2018. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong.

His current research interests include electronic design automation, combinatorial optimization, and machine learning.



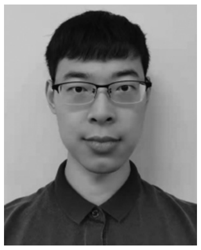
Lixin Liu received the B.Eng. degree in electronic science and technology from the South China University of Technology, Guangzhou, China, in 2019. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong.

His research interests include deep learning and its applications on physical design.

Fangzhou Wang received the B.Sc. degree in computer science from City University of Hong Kong, Hong Kong, in 2019. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong.

His current research interests include VLSI routing, combinatorial optimization, and boolean satisfiability.

Mr.Wang has won three championships in renowned EDA contests, including 2020 and 2019 CAD Contests at ICCAD, and ISPD 2020 Contest.



Xiaopeng Zhang received the B.Eng. degree in software engineering from Shandong University, Jinan, China, in 2015, and the M.Sc. degree in computer science and technology from Beihang University, Beijing, China, in 2018. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong.

His current research interests include physical design and design for manufacturability, and machine learning with applications in VLSI CAD.

Mr. Zhang received four ISPD/ICCAD contest awards.

Evangeline F. Y. Young (Senior Member, IEEE) received the B.Sc. degree in computer science from The Chinese University of Hong Kong (CUHK), Hong Kong, and the Ph.D. degree from The University of Texas at Austin, Austin, TX, USA, in 1999.

She is currently a Professor with the Department of Computer Science and Engineering, CUHK. Her research interests include EDA, optimization, algorithms and AI. Her research focuses on floorplanning, placement, routing, DFM and EDA on physical design in general.

Dr. Young's research group has won best paper awards from ICCAD 2017, ISPD 2017, SLIP 2017, and FCCM 2018, and several championships and prizes in renown EDA contests, including the 2018-20, 2015-16, 2012-13 CAD Contests at ICCAD, DAC 2012, and ISPD 2015-20 and 2010-11. She has served on the organization committees of ICCAD, ISPD, ARC and FPT and on the program committees of conferences, including DAC, ICCAD, ISPD, ASP-DAC, SLIP, DATE, and GLSVLSI. She also served on the editorial boards of IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, *ACM Transactions on Design Automation of Electronic Systems*, and *Integration, the VLSI Journal*.