Placement for Wafer-Scale Deep Learning Accelerator

Benzheng Li
Xidian University
bzli@stu.xidian.edu.cn

Qi Du
Xidian University
qdu@stu.xidian.edu.cn

Dingcheng Liu
Xidian University
dcliu1997@gmail.com

Jingchong Zhang
Xidian University
jczhang_1@stu.xidian.edu.cn

Gengjie Chen
Giga Design Automation
gjchen@giga-da.com

Hailong You
Xidian University
hlyou@mail.xidian.edu.cn

ABSTRACT

To meet the growing demand from deep learning applications for computing resources, accelerators by ASIC are necessary. A wafer-scale engine (WSE) is recently proposed [1], which is able to simultaneously accelerate multiple layers from a neural network (NN). However, without a high-quality placement that properly maps NN layers onto the WSE, the acceleration efficiency cannot be achieved. Here, the WSE placement resembles the traditional ASIC floorplan problem of placing blocks onto a chip region, but they are fundamentally different. Since the slowest layer determines the compute time of the whole NN on WSE, a layer with a heavier workload needs more computing resources. Besides, locations of layers and protocol adapter cost of internal IO connections will influence inter-layer communication overhead. In this paper, we propose GigaPlacer to handle this new challenge. A binary-search-based framework is developed to obtain a minimum compute time of the NN. Two dynamic-programming-based algorithms with different optimizing strategies are integrated to produce legal placement. The distance and adapter cost between connected layers will be further minimized by some refinements. Compared with the first place of the ISPD2020 Contest, GigaPlacer reduces the contest metric by up to 6.89% and on average 2.09%, while runs 7.23× faster.

ACM Reference Format:
Benzheng Li, Qi Du, Dingcheng Liu, Jingchong Zhang, Gengjie Chen, and Hailong You. 2021. Placement for Wafer-Scale Deep Learning Accelerator. In *26th Asia and South Pacific Design Automation Conference (ASPDAC '21), January 18–21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3394885.3431563

1 INTRODUCTION

With a blossoming of deep learning applications in the 2010s [9], accelerating neural networks (NNs) is becoming a prominent issue in order to satisfy the rapidly increasing computational requirements. Specialized accelerator by ASIC is a promising technique for this problem. For instance, an accelerator called DianNao [3] is 117.87 times faster than a 128-bit 2GHz SIMD processor, with a reduction of total energy by 21.08×. Tensor processing unit (TPU) proposed by Google [6] is on average about 15-30× faster over its contemporary CPU or GPU, improving TOPS/Watt by about 30-80×. Recently, a wafer-scale engine (WSE) is proposed by Cerebras [1]. Consisting of 400k programmable cores and distributed on-chip memory, the WSE can parallelly accelerate computations of multiple layers from a NN. It also offers high bandwidth for communication between cores by local connectivity with a 2D mesh topology. However, to fully utilize its capability, the placement of the layers implemented on WSE needs thorough optimization.

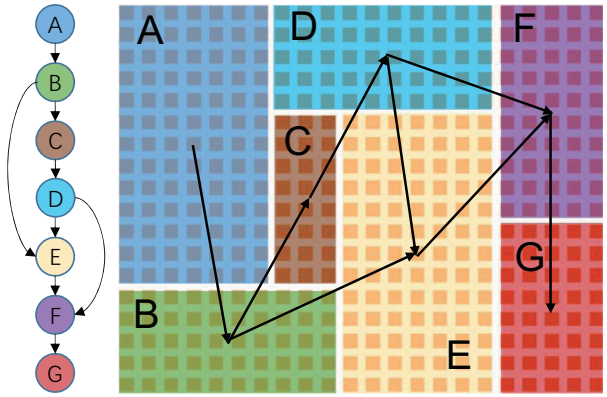


Figure 1: Example placement of a 7-layer neural network onto a slice of WSE with 22×18 cores.

In the WSE placement, each layer is mapped to be a rectangular array of cores, called a kernel, as shown in Figure 1. The influence on the acceleration efficiency by the placement includes allocation of cores, arrangement of the kernels' locations and protocol adapter cost of connections. Since the kernel with the lowest throughput determines the overall performance, the compute time should be kept uniform across all the kernels. Thus, the number of cores allocated to a kernel needs to be proportional to the workload. Additionally, the cores transfer data via the on-chip mesh network. So the intra-kernel communication overhead is affected by the distance between the cores allocated to the kernel, which can be characterized as the shape of the kernel. Similarly, the inter-kernel communication overhead is influenced by the distance between cores in different kernels. Besides, the protocol adapter cost, which is related to the parameters of the connected kernels, also affects the inter-kernel communication.

The WSE placement problem resembles the traditional floorplan problem, which is to place a set of blocks (modules or macros) onto the 2D chip region. Simulated annealing (SA) is a classic algorithm to search a good topology of blocks in a floorplan. The topologies can be divided into two classes. Slicing structure is a rectangle dissection generated by recursively slicing a rectangular region, while non-slicing structure allows rectangles piled up irregularly. A representation of slicing structures, called normalized Polish expression (nPe) [10], provides operations for SA to perturb the floorplan. And the search of non-slicing floorplans is enabled by representations like O-tree [4] and B*-tree [2]. Recently, an approach based on reinforcement learning (RL) is

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

ACM ISBN 978-1-4503-7999-1/21/01…$15.00
https://doi.org/10.1145/3394885.3431563

proposed [7] to place the macros one at a time. States in this RL placement problem are all the possible partial placement of a netlist, and the available actions of a macro are the feasible locations. Actions are taken by a policy network trained to generalize to unseen blocks.

However, WSE placement is fundamentally different from the floorplan problem. Both the areas and the shapes of kernels remain to be determined in the WSE placement problem. Instead, blocks in the floorplan problem are all with fixed area, and mostly with fixed shapes. Besides, the number of adjustable parameters in WSE placement problem is much greater, which significantly enlarges the solution space. As a result, it's difficult for SA-based and learning-based methods to generate nearly optimal placements for WSE. On the other hand, the connectivity of a NN is dramatically simpler than that of a floorplan netlist. In SA-based and learning-based methods, great efforts are put into handling the complex connectivity of the blocks. Therefore, placing layers in topological orders of the NN can substantially reduce runtime with little sacrifice of quality. In our early exploration, we tried implementing a SA algorithm based on nPe, and found its performance poor. Even in the SA algorithm's best case, its score was still 13% worse than GigaPlacer, but took a 2.6 times longer runtime.

To optimize the compute time of NNs' acceleration, we proposed an effective and efficient placer, called GigaPlacer, to handle the WSE placement problem. Our contribution is summarized as follows:

- We proposed two dynamic-programming-based (DP-based) algorithms to place the layers. A binary-search-based framework is adopted to guarantee the uniformity of compute time of all kernels.
- We proposed a row height minimization algorithm to determine the optimal execution parameters for each kernel. We exploit an effective enumeration technique with pruning, guided by binary search. In this way, the wafer area will be fully utilized.
- We proposed two techniques to further refine the results. Inter-kernel distance is reduced by swapping some kernels, and the protocol adapter cost is minimized by slightly adjusting parameters of connected kernels.
- In comparison with the first place of the ISPD2020 Contest, we achieved a reduction on the contest metric by up to 6.89% and on average 2.09%, with 7.23× runtime speed up.

2 PRELIMINARIES

This section introduces the basic structure of WSE at first. The parameters and performance functions of kernels are presented later, and the problem formulation is introduced in the end.

2.1 Wafer-Scale Engine

In WSE, there are around 400k tiles arranged in a 633×633 grid. Each tile has a programmable execution core, 48KB SRAM, and a router for interconnection with other tiles.

In the WSE placement problem, layers in NN are mapped to a set of rectangular kernels, where each kernel occupies multiple tiles and performs an individual NN computational task by running the execution cores of tiles in parallel [5].

2.2 Kernel Parameters

The performance of kernels is determined by various parameters including formal parameters and execution parameters.

H, W, R, S, C, K, T are the formal parameters that are defined by NN specification and cannot be modified by compilation procedure. As depicted in Figure 2, H and W represent the sizes of input images; R and S represent the sizes of the receptive field; C and K are the numbers of input and output features; T represents the step size of the operation.

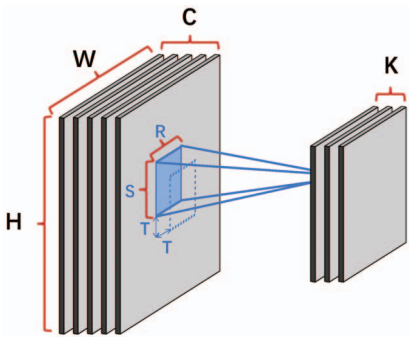


Figure 2: Formal parameters of NN kernels.

```
convperf(H,W,R,S,C,K,T;h,w,c,k)={
  height  =h*w*(c+1)
  width   =3*k
  time    =ceil(H/h)*ceil(W/w)*ceil(C/c)*ceil(K/k)*R*S/T/T
  memory  =C/c*K/k*R*S+(W+S-1)/w*(H+R-1)/h*K/k
}
dblockperf(H,W,F;h,w,c1,c2,c3,k1,k2,k3)={
  conv1   =convperf(H,W,1,1,F,   F/4,1,h,w,c1,k1)
  conv2   =convperf(H,W,1,1,F/4,F/4,1,h,w,c2,k2)
  conv3   =convperf(H,W,1,1,F,   F/4,1,h,w,c3,k3)
  height  =max(conv1.height,conv2.height,conv3.height)
  width   =conv1.width+conv2.width+conv3.width
  time    =max(conv1.time,conv2.time,conv3.time)
  memory  =max(conv1.memory,conv2.memory,conv3.memory)
}
cblockperf(H,W,F;h,w,c1,c2,c3,c4,k1,k2,k3,k4)={
  conv1 =convperf(H,   W,   1,1,F/2,F/4,1,h,w,c1,k1)
  conv2 =convperf(H,   W,   3,3,F/4,F/4,2,h,w,c2,k2)
  conv3 =convperf(H/2,W/2,1,1,F/4,F,   1,h,w,c3,k3)
  conv4 =convperf(H,   W,   1,1,F/2,F,   2,h,w,c4,k4)
  height=max(conv1.height,conv2.height,conv3.height,conv4.height)
  width =conv1.width+conv2.width+conv3.width+conv4.width
  time  =max(conv1.time,conv2.time,conv3.time,conv4.time)
  memory=max(conv1.memory,conv2.memory,conv3.memory,conv4.memory)
}
```

Figure 3: Performance functions of two types of kernels, *dblock* and *cblock*, given by the kernel library.

h, w, c, k are the execution parameters that determine the amount and arrangement of tiles in kernels and describe how parallel operations can be performed on the tiles. They separate the various dimensions of tensor operations, H, W, C, K. Unlike formal parameters, they are allowed to be modified.

2.3 Kernel Performance Functions

For each kernel, its size, compute time and memory are calculated by parameters. These metrics represent the compute resources required to execute a kernel. In the given kernel library, the performance functions of various types of kernels are specified. Among them, the performance functions of a basic convolution kernel, *conv*, are as follows.

$$height = hw(c+1) \tag{1a}$$

$$width = 3k \tag{1b}$$

$$time = \lceil H/h \rceil \lceil W/w \rceil \lceil C/c \rceil \lceil K/k \rceil RS/T^2 \tag{1c}$$

$$memory = \lfloor \frac{CKRS}{ck} + \frac{(W+S-1)(H+R-1)K}{whk} \rfloor \tag{1d}$$

Note that the *memory* refers to the memory required for each tile.

In addition to *conv* kernel, there are other types of kernels which contain multiple *conv* kernels. The performance functions of two types of kernels, *dblock* and *cblock*, are shown in Figure 3. Take the *dblock*

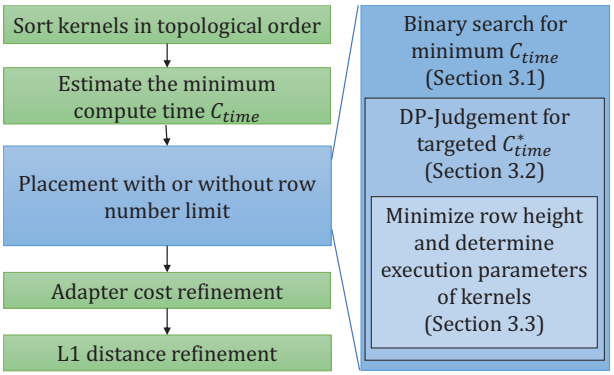


Figure 4: GigaPlacer framework.

kernel for example, it contains three *conv* kernels, where the calculation methods of metrics of each *conv* kernel are same as in Equation (1). And in the *conv* kernels, the formal parameters R and S are determined by F, a given formal parameter of the *dblock* kernel. Because the *convs* are placed horizontally, the width of the *dblock* kernel is the sum of widths of *conv* kernels, while other metrics pick the maximum one.

In the WSE placement problem, the kernel connections are specified by a kernel graph. The communication overhead between two connected kernels is related to their distance but also the protocol adapter. The adapter cost represents the mismatching of the I/O protocols of connected kernels. For each connection, the adapter cost is the number of parameter (h, w, c) mismatches between the two connected kernels.

2.4 Problem Formulation

In general, the WSE placement problem can be formally defined as follows. A kernel library specifying the performance functions of kernels and a kernel graph with connection relationships are given. The following constraints should be satisfied:

- All kernels must be placed in the tile grid with dimensions of 633×633;
- A kernel must be a rectangle and contain an integer number of tiles;
- No overlap between kernels is allowed;
- The unit memory of each tile in kernels should not exceed the memory limit, 48KB.

And the optimization objective is to minimize a weighted sum of

- C_{dist}, the total L1 distance between communicating kernels,
- C_{time}, the maximum compute time of all kernels,
- $C_{adapter}$, the sum of the adapter cost of each connections.

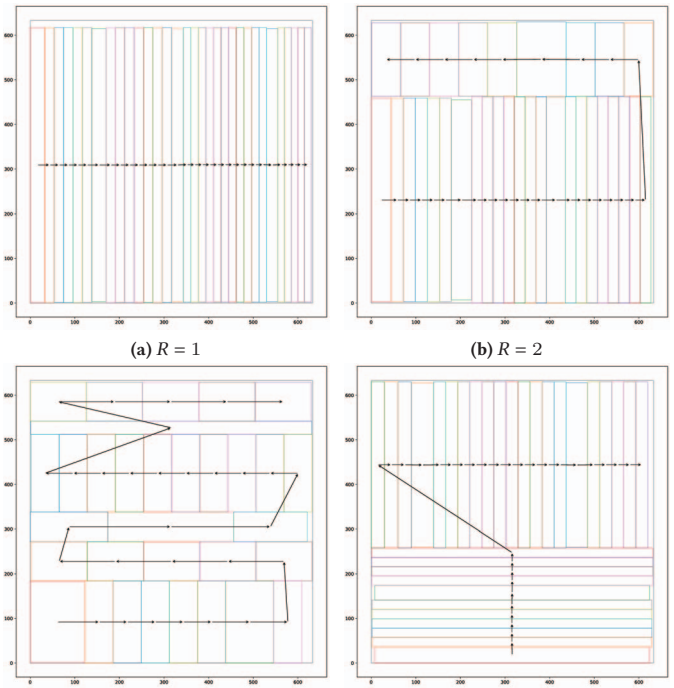
That is, the total cost C_{total} is:

$$C_{total} = w_{dist}C_{dist} + w_{time}C_{time} + w_{adapter}C_{adapter}, \tag{2}$$

where w_{dist}, w_{time} and $w_{adapter}$ are the weights for the corresponding costs.

3 ALGORITHMS

In this section, we will describe the placement generation algorithms. Figure 4 describes the overall flow for GigaPlacer. The main framework is described in Section 3.1. To get a balanced result between compute time C_{time} and L1 distance C_{dist}, approaches based on binary search but with different judgment by DP are proposed in Section 3.2. Section 3.3 introduces the algorithm for placing kernels in a row, which is used in DP. We also perform a refinement to decrease C_{dist} and adapter cost $C_{adapter}$ and it will be mentioned in Section 3.4.



(a) $R = 1$ **(b)** $R = 2$ **(c)** $R = 6$ **(d)** $R = 12$

Figure 5: Example placements by GigaPlacer. Kernels are placed row by row. They are generated for a same NN design while row number R is different.

3.1 Binary-Search-Based Framework

In this section, we propose a framework which generates a placement with minimum C_{time} while kernels are legally placed in restricted area. GigaPlacer places the kernels row by row as in Figure 5. In this way, if the kernels are placed in a certain order, the C_{time} is monotonic with the total height. So binary search can help us find the minimum C_{time}. In each iteration of binary search, we define a guessed value of C_{time} as C^*_{time}. The C^*_{time} is equal to the middle number of the search range. If all kernels can be placed legally while their compute time is smaller than C^*_{time}, the search range of C_{time} will be shrunk to the smaller half, otherwise it will be shrunk to the larger half.

Since a smaller initial search range helps to reduce the runtime of binary search, an estimation of C_{time} is needed. The minimum C_{time} can be achieved when the compute time is uniform across all kernels. To keep the compute time roughly equal, the kernels' areas should generally subject to a set of ratios that is calculated from the formulas in Figure 3. To be more specific, the area weights of kernels (for *conv*, *cblock* and *dblock* respectively) are set to:

$$w_{conv} = HWCKRS/T^2 \tag{3a}$$

$$w_{cblock} = 29/64 \times HWF^2 \tag{3b}$$

$$w_{dblock} = 17/16 \times HWF^2 \tag{3c}$$

For instance, based on Equation (1), the weight w_{conv} of a *conv* kernel is approximated by

$$\begin{aligned} w_{conv} &= width \times height = hw(c+1) \times 3k \\ &\approx 3 \times hwck \approx \frac{3}{time} \times \frac{HWCKRS}{T^2} \end{aligned} \tag{4}$$

Here, $3/time$ is treated as a constant and removed. The weights of *cblock* and *dblock* kernels are obtained similarly.

Algorithm 1 DP-Judgment Without Row Number Limit

Input: The set of kernels $Ker = \{Ker_1, Ker_2, \ldots, Ker_n\}$; Compute time limit C^*_{time}; Height limit of WSE H_{WSE};

for $i = 1 \ldots n$ **do**
  for $k = 1 \ldots i$ **do**
    $NewHeight \leftarrow f_{i-k}$ +GetHeight($\{Ker_{i-k+1}, \ldots, Ker_i\}, C^*_{time}$)
    if $NewHeight < f_i$ **then**
      $f_i \leftarrow NewHeight$
Placement with compute time limit C^*_{time} is achievable if and only if $f_n \leqslant H_{WSE}$

However, even with given areas, the C_{time} still varies with the parameters of kernels. So we pick the kernel with largest area, calculate it's all compute times in different combination of parameters under the area constraint, and select the minimum one to be the estimated C_{time}. The initial search interval is set from one-tenth of the estimated C_{time}, to ten times of it, which is proved practically large enough in our experiments.

Next, For the targeted C^*_{time}, we will exploit DP algorithm to generate a placement with minimum height to judge if it is legal. During DP progress, all execution parameters of each kernel will be determined.

3.2 DP-Judgment

We design two different DP based judgment algorithms. The first one will get a placement without limiting number of rows and it tends to get better C_{time}. The second one sets a row number limit because many rows always lead to large C_{dist}. Although this limit will lead to a slight deterioration of C_{time}, it will bring much better C_{dist}.

3.2.1 DP-Judgment Without Row Number Limit. Algorithm 1 describes the judgment function for a targeted C^*_{time} and the row number will be determined automatically. f_i means the minimum height used in the case of the first i kernels were placed in topological order. If there are k kernels in the last row, f_i can be calculated by:

$$f_i = f_{i-k} + \text{GetHeight}(\{Ker_{i-k+1}, \ldots, Ker_i\}, C^*_{time}) \tag{5}$$

Here, Ker_i is the i-th kernel. The GetHeight is a function which places a set of kernels in a row and returns the minimum height of the row, which will be introduced in Section 3.3. We get the minimum f_i after enumerating all k. If more than one k get the minimum f_i, we choose the one whose C_{dist} and $C_{adapter}$ is smaller. In the end, we get f_n as the final height while all the kernels are placed. We can accept the C^*_{time} if f_n is smaller than height limit H_{WSE}.

3.2.2 DP-Judgment With Row Number Limit. In order to find some other result with possibly better C_{dist}, we come up with another DP-Judgment described in Algorithm 2. The main difference is that the row number R is fixed in the second algorithm. $f_{i,j}$ means the minimum height used in the case that the first i kernels were placed in topological order and j rows are used in the meantime. If there are k kernels in the last row, $f_{i,j}$ can be calculated by:

$$f_{i,j} = f_{i-k,j-1} + \text{GetHeight}(\{Ker_{i-k+1}, \ldots, Ker_i\}, C^*_{time}) \tag{6}$$

Using this DP equation, we can get the minimum C^*_{time} while kernels are placed in R rows. In general, as the number of rows decreases, C_{dist} will also decrease, but C_{time} will increase. So we can try different R to find a balance between C_{dist} and C_{time}. Our experimental result demonstrates that when the placement of Algorithm 2 is better than Algorithm 1, R always be one or two.

Algorithm 2 DP-Judgment With Row Number Limit

Input: The set of kernels $Ker = \{Ker_1, \ldots, Ker_n\}$; Compute time limit C^*_{time}; Row number R; Height limit of WSE H_{WSE};

for $i = 1 \ldots n$ **do**
  $f_{i,1} \leftarrow$ GetHeight($\{Ker_1, Ker_2, \ldots, Ker_i\}, T_{limit}$)
for $j = 2 \ldots R$ **do**
  for $i = 1 \ldots n$ **do**
    for $k = 1 \ldots i$ **do**
      $NewHeight \leftarrow f_{i-k,j-1}$ +GetHeight($\{Ker_{i-k+1}, \ldots, Ker_i\}, C^*_{time}$)
      if $NewHeight < f_{i,j}$ **then**
        $f_{i,j} \leftarrow NewHeight$
Placement with compute time limit C^*_{time} is achievable if and only if $f_{R,n} \leqslant H_{WSE}$

When the value of R is one, all the kernels are placed as strips as Figure 5(a) shows, and we always get the best C_{dist}. However, this placement always get a high C_{time}. As a result we try to decrease some kernel's height. We place some kernels as horizontal strip, next place other kernels in one row as Figure 5(d) shows. The C_{time} and C_{dist} of this strategy are between the result of DP with $R = 1$ and $R = 2$.

When the weight of C_{dist} is huge, the $R = 1$ appears to get the best result. In order to get better C_{dist}, compute time limit is enlarged. After getting minimum C_{time}, we multiply it by α and run DP-Judgment to get the result. The α is assigned 80 times from 1.02 to 1.02^{80} in our algorithm.

In short, several strategies with row number limit are tried and the best result will be selected. In order to get the minimum total cost C_{total}, Algorithm 1 and Algorithm 2 will both be run and the best placement will be selected.

3.3 Row Height Minimization

In this section, the algorithm for function GetHeight is introduced. Another binary search is used to find the minimum row height H_{limit} that all the kernels can be placed in a row in the case of compute time and width limit are fixed. We define a guessed value of H_{limit} as H^*_{limit}. The H^*_{limit} is equal to the middle number of the binary search's range. The height of every kernel is set to be H^*_{limit} and then we place kernels in a row from left to right one by one. We can accept H^*_{limit} if the total width is small than the width limit of WSE after all the kernels are placed, and then shrink the range of row height to the smaller half, otherwise to the larger half.

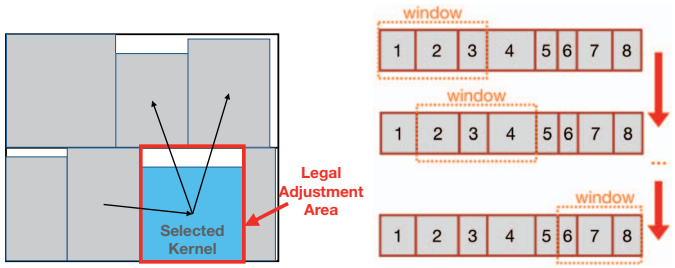
For a given height, compute time and memory limit, we need a function to figure out a suit of parameter for a kernel to get the minimum width. Our algorithm enumerates the values of h and w using two levels of loops with effective pruning and then figures out the corresponding c and k. After that, we pick a suit of parameter which brings minimum width. We consider two situations, rotating kernels by 90 degree or not.

If we do not rotate the kernel, in order to get a smaller compute time, the height of every convolution has better to be H^*_{limit}. Based on Equation (1a), the parameter c of every convolution is:

$$c = \lfloor H^*_{limit} / (h \times w - 1) \rfloor \tag{7}$$

In order to satisfy the memory M_{limit}, according to Equation (1d), the parameter k of each convolution must satisfy:

$$k \geqslant \frac{K \times \frac{C}{c} \times R \times S + \frac{W+S-1}{w} \times \frac{H+R-1}{h}}{M_{limit}} \tag{8}$$



(a) Adapter cost refinement. (b) L1 distance refinement.

Figure 6: Two refinement algorithms.

In order to satisfy the compute time limit C^*_{time}, according to Equation (1c), the parameter k of each convolution must satisfy:

$$k \geqslant \lceil \frac{K}{\lfloor C^*_{time} \div (\lceil \frac{H}{h} \rceil \times \lceil \frac{W}{w} \rceil \times \lceil \frac{C}{c} \rceil \times R \times S \div T^2) \rfloor} \rceil \tag{9}$$

So we can get the minimum k which satisfies Equations (8) and (9).

In addition, we can rotate a kernel by 90 degree. In this case, the total width of all convolution is H^*_{limit}. In order to get similar height of each convolution, the width of them is allocated according to the proportion of:

$$\lceil \frac{H}{h} \rceil \times \lceil \frac{W}{w} \rceil \times R \times S \times C \times K \div T^2 \tag{10}$$

According to Equation (1b), parameter k of each convolution is calculated after allocating width. In order to satisfy the memory limit, according to Equation (1d), the parameter c of each convolution must satisfy:

$$c \geqslant \frac{C \times \frac{K}{k} \times R \times S}{M_{limit} - \frac{W+S-1}{w} \times \frac{H+R-1}{h} \times \frac{K}{k}} \tag{11}$$

In order to satisfy the compute time limit, according to Equation (1c), the parameter c of each convolution must satisfy:

$$c \geqslant \lceil \frac{C}{\lfloor C^*_{time} \div (\lceil \frac{H}{h} \rceil \times \lceil \frac{W}{w} \rceil \times \lceil \frac{K}{k} \rceil \times R \times S \div T^2) \rfloor} \rceil \tag{12}$$

So we can get the minimum c which satisfies Equations (11) and (12). Until now, all the parameters of these kernels and H_{limit} are defined.

We can prune out many h and w values which are not able to bring best result. Taking h as an example, if some different h generate the same result in $\lceil H/h \rceil$, only the smallest one is necessary. This pruning brings about 3× speedup for enumeration.

3.4 Refinement

After placement generation, we perform two refinement algorithms to reduce adapter cost $C_{adapter}$ and L1 distance C_{dist}. Where the adapter cost refinement algorithm adjusts the execution parameters, the L1 distance refinement algorithm does not modify any parameters, but only changes the position of kernels or reorder them.

3.4.1 Adapter Cost Refinement. In adapter cost refinement algorithm, the parameters of the kernels will be redetermined to decrease $C_{adapter}$ while the compute time C_{time} is unchanged. A kernel will be picked up and its h and w parameters will be changed to be the same to kernels which are connected to it. Then other parameters will be recalculated to keep compute time and width unchanged. This progress changes the selected kernel's height. The legal adjustment area is defined as an area whose width is the kernel's width and height is the row height. As Figure 6(a) shows, the blue kernel is selected and the highlight area is legal adjustment area. If the kernel is still in the legal adjustment area , this change is acceptable. The h and w parameter will be redetermined to be the pair which brings the minimum $C_{adapter}$. All kernels will be tried like this. The process will run several times until there is no improvement.

3.4.2 L1 distance Refinement. The L1 distance refinement algorithm picks each row and move kernels in the row to search better C_{dist}. We adapt a window-based local reordering technique similar to FastPlace-DP [8]. A window covering several continuously kernels will move from the beginning to the ending of the row as Figure 6(b) shows. The all possible orders of kernels in the window will be enumerated and the result will be saved if C_{dist} decreases. The size of the window is set to 6 to balance the run time overhead and quality improvement. Besides, the position of some continuously kernels will be reversed as another simple strategy whose window size is much bigger than local reordering. This step will be iterated for many times until there is no improvement.

4 EXPERIMENTAL RESULTS

We implement GigaPlacer in C++ and compile it with g++ 5.4.0. Benchmarks [5] are from ISPD 2020 Contest with number of kernels ranges from 16 to 100 as Table 1 shows. All experiments were conducted on a linux machine with a 1.6GHz intel i5-10210U processor and 8-GB memory.

GigaPlacer generates solutions by embedding several strategies in the DP-based algorithm. The experiment shows that the placement generation strategies we design are all effective, and all these strategies appear in the final result. Figure 7 shows the final results of GigaPlacer on test cases E, I, M, and F.

Although L1 distance is mainly determined by the DP-based placement, we perform L1 distance refinement by swapping some kernels locally to further reduce the C_{dist}. Figure 8 shows the effect of L1 distance refinement on case D, where C_{dist} is reduced by 12.5%. In general, it reduces C_{dist} by an average of 3% on ISPD2020 benchmarks. Besides, we perform adapter cost refinement to reduce the $C_{adapter}$. This refinement decreases $C_{adapter}$ by 25.1% on average.

The detailed scores of GigaPlacer are shown in Table 1. Each case have its w_{time}, w_{dist} and $w_{adapter}$, which represent the weight of compute time C_{time} , L1 distance C_{dist} and adapter cost $C_{adapter}$, respectively. In all these test cases, w_{time} is equal to 1, w_{dist} is from 1 to 400, and $w_{adapter}$ ranges from 0 to 400. The final score is evaluated by C_{total}, where C_{total} is the weighted sum of C_{time}, C_{dist} and $C_{adapter}$. It can be observed that results of GigaPlacer have a good trade off in these aspects. For example, Case C and Case G are only different from w_{dist} and $w_{adapter}$, Case G have larger w_{dist} = 10 and $w_{adapter}$ = 100, so GigaPlacer will consider C_{dist} and $C_{adapter}$ more. Take $C_{adapter}$ as an example. The $C_{adapter}$ of case C ($w_{adapter}$ = 0) is 249, but the $C_{adapter}$ of case G ($w_{adapter}$ = 100) is reduced to 17.

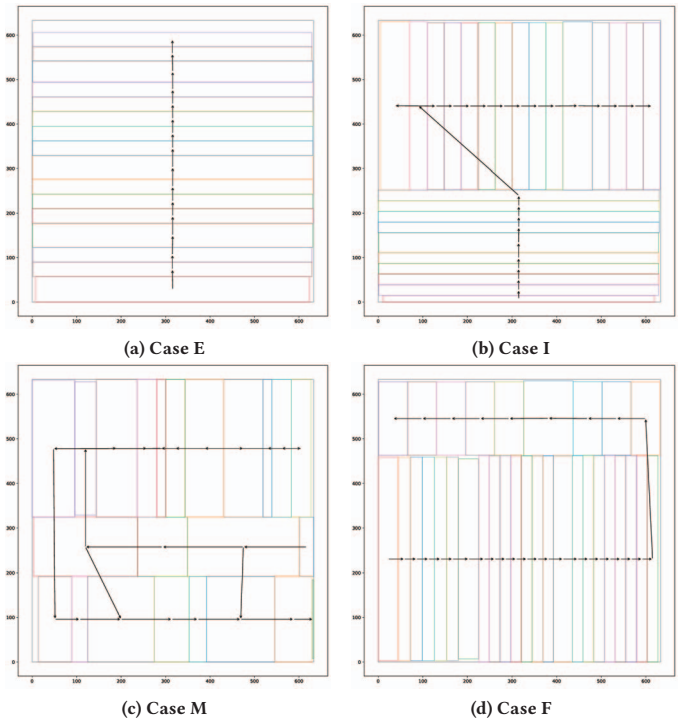
Table 1 also shows the comparison with the first place of ISPD 2020 Contest. Compared with them, GigaPlacer performed better in 16 out of 20 cases, with a maximum improvement of 6.89% and an average of 2.09% in C_{total}. In addition, GigaPlacer achieves 7.23× runtime speed up.

5 CONCLUSION

In this paper, we present GigaPlacer to place NN layers on WSE. A minimum compute time is achieved by a binary-search-based framework. Two DP-based algorithms focus on different objectives are integrated to place the kernels. To further reduce inter-kernel communication overhead, we adjust the parameters of connected kernels and try to

Table 1: Result Comparison on ISPD2020 Benchmarks

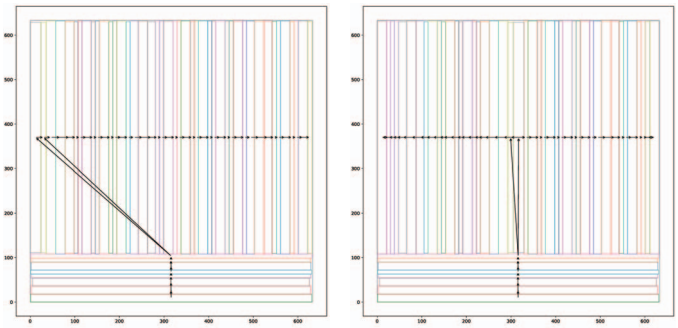
Case	Kernel Num	w_{time}	w_{dist}	$w_{adapter}$	C_{time}		C_{dist}		$C_{adapter}$		C_{total}		Runtime(s)	
					ISPD'20	Ours	ISPD'20	Ours	ISPD'20	Ours	ISPD'20	Ours	ISPD'20	Ours
A	16	1	1	0	34496	34496	1314	1189.5	15	16	35810	**35685.5**	35	**3**
B	32	1	1	0	63504	64512	2639.5	2004	18	10	**66143.5**	66516	37	**15**
C	100	1	1	0	64512	64768	2408	1959.5	185	249	66920	**66727.5**	210	**59**
D	52	1	1	0	33712	33712	2722	2084.5	123	137	36434	**35796.5**	37	**22**
E	16	1	10	100	39312	39312	562	569.5	11	8	46032	**45807**	76	**5**
F	32	1	10	100	65170	65170	1489.5	1488.5	12	14	**81265**	81455	116	**15**
G	100	1	10	100	64512	71680	2508.5	1945.5	98	17	99397	**92835**	188	**56**
H	52	1	10	100	39520	43008	1104.5	887	49	12	55465	**53078**	76	**20**
I	25	1	4	0	52136	49392	617.5	1284.5	13	10	54606	**54530**	69	**10**
J	79	1	4	0	50274	49392	2472.5	1988	69	134	60164	**57344**	90	**49**
K	16	1	4	0	432	504	240	211.5	3	7	1392	**1350**	28	**1**
L	52	1	4	0	864	864	279	283	18	178	**1980**	1996	38	**16**
M	23	1	4	0	2211840	2211840	3610.5	3365.5	41	44	2226282	**2225302**	68	**13**
N	26	1	4	0	1482	1640	480.5	437.5	10	9	3404	**3390**	3	**1**
O	25	1	40	400	52136	52136	617.5	617.5	13	13	82036	**82036**	121	**10**
P	79	1	40	400	57792	57792	1101	1625	85	11	135832	**127192**	122	**38**
Q	16	1	40	400	882	896	166	153	6	7	9922	**9816**	27	**2**
R	52	1	40	400	8064	1008	259	324	20	27	26424	**24768**	66	**10**
S	23	1	400	400	2396300	2396160	1897.5	1352	48	47	3174500	**2955760**	95	**13**
T	26	1	40	400	4102	4875	208.5	176.5	4	5	14042	**13935**	5	**1**
Avg Ratio					1.32	**1**	1.08	**1**	1.65	**1**	1.02	**1**	7.23	**1**



(a) Case E (b) Case I

(c) Case M (d) Case F

Figure 7: Results of GigaPlacer on some ISPD2020 cases.



(a) Without refinement, C_{dist} = 2374.5 (b) With refinement, C_{dist} = 2078.5

Figure 8: Results on case D with/without L1 distance refinement

reorder some kernels. Compared with the first place of the ISPD2020 Contest, GigaPlacer is not only better but also faster. Our future work will be improving the judgment function to automatically optimize the inter-kernel distance, instead of setting limitation of row numbers manually. We also intend to explore more topologies to place the kernels for better performance.

REFERENCES

[1] Cerebras. 2019. Wafer-Scale Deep Learning. In IEEE Hot Chips Symposium. 1–31.

[2] Yun-Chih Chang, Yao-Wen Chang, Guang-Ming Wu, and Shu-Wei Wu. 2000. B*-trees: A new representation for non-slicing floorplans. In ACM/IEEE Design Automation Conference (DAC). 458–463.

[3] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 269–284.

[4] Pei-Ning Guo, Chung-Kuan Cheng, and Takeshi Yoshimura. 1999. An O-tree representation of non-slicing floorplan and its applications. In ACM/IEEE Design Automation Conference (DAC). 268–273.

[5] Michael James, Marvin Tom, Patrick Groeneveld, and Vladimir Kibardin. 2020. ISPD 2020 Physical Mapping of Neural Networks on a Wafer-Scale Deep Learning Accelerator. In ACM International Symposium on Physical Design (ISPD). 145–149.

[6] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In IEEE/ACM International Symposium on Computer Architecture (ISCA). 1–12.

[7] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Sungmin Bae, et al. 2020. Chip Placement with Deep Reinforcement Learning. arXiv:2004.10746 (2020).

[8] Min Pan, Natarajan Viswanathan, and Chris Chu. 2005. An efficient and effective detailed placement algorithm. In IEEE/ACM International Conference on Computer-Aided Design (ICCAD). 48–55.

[9] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. Proc. IEEE 105, 12 (2017), 2295–2329.

[10] Martin D.F. Wong and CL Liu. 1986. A new algorithm for floorplan design. In ACM/IEEE Design Automation Conference (DAC). 101–107.