Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

COPPER: a combinatorial optimization problem solver with processing-in-memory architecture*

Qiankun WANG[1], Xingchen LI[2,3], Bingzhe WU[4], Ke YANG[3], Wei HU[5], Guangyu SUN†‡[3,6,7], Yuchao YANG[3]

[1]*School of Software & Microelectronics, Peking University, Beijing 100871, China*
[2]*School of Computer Science, Peking University, Beijing 100871, China*
[3]*School of Integrated Circuits, Peking University, Beijing 100871, China*
[4]*Tencent AI Lab, Shenzhen 518057, China*
[5]*College of Physics and Information Engineering, Fuzhou University, Fuzhou 350116, China*
[6]*Beijing Advanced Innovation Center for Integrated Circuits, Beijing 100871, China*
[7]*Beijing Academy of Artificial Intelligence, Beijing 100080, China*
†E-mail: gsun@pku.edu.cn

Received Oct. 14, 2022; Revision accepted Mar. 2, 2023; Crosschecked Apr. 7, 2023

Abstract: The combinatorial optimization problem (COP), which aims to find the optimal solution in discrete space, is fundamental in various fields. Unfortunately, many COPs are NP-complete, and require much more time to solve as the problem scale increases. Troubled by this, researchers may prefer fast methods even if they are not exact, so approximation algorithms, heuristic algorithms, and machine learning have been proposed. Some works proposed chaotic simulated annealing (CSA) based on the Hopfield neural network and did a good job. However, CSA is not something that current general-purpose processors can handle easily, and there is no special hardware for it. To efficiently perform CSA, we propose a software and hardware co-design. In software, we quantize the weight and output using appropriate bit widths, and then modify the calculations that are not suitable for hardware implementation. In hardware, we design a specialized processing-in-memory hardware architecture named COPPER based on the memristor. COPPER is capable of efficiently running the modified quantized CSA algorithm and supporting the pipeline further acceleration. The results show that COPPER can perform CSA remarkably well in both speed and energy.

Key words: Combinatorial optimization; Chaotic simulated annealing; Processing-in-memory
https://doi.org/10.1631/FITEE.2200463 **CLC number:** TP389.1

1 Introduction

Combinatorial optimization problems, abbreviated COPs, are prevalent but intractable in our daily lives. A classic COP is the traveling salesman problem (TSP), whose purpose is finding the shortest path through all cities and then back to the point of departure. Another COP we often meet is the work planning problem, which looks for an order of handling tasks to minimize the total timeout. Unfortunately, many COPs are non-deterministic polynomial complete (NPC). Consequently, we cannot solve them quickly when the problem size increases to a large scale. The prior theory proved that NPC problems can be transformed into each other in polynomial time (Karp, 1972). In other words, we can

‡ Corresponding author
* Project supported by the National Natural Science Foundation of China (Nos. 61832020, 62032001, 92064006, and 62274036), the Beijing Academy of Artificial Intelligence (BAAI) of China, and the 111 Project of China (No. B18001)
ORCID: Qiankun WANG, https://orcid.org/0000-0002-3723-3444; Guangyu SUN, https://orcid.org/0000-0002-7315-6589

simply pay attention to only one specific problem.

Many methods have been proposed to solve COPs. Though exact algorithms, such as enumeration and branch and bound, can obtain the optimal solution, they are usually extremely time-consuming. Approximate algorithms, such as the nearest neighbor algorithm for TSP, can obtain less acceptable solutions in polynomial time. However, they are normally designed for particular problems. Heuristic algorithms, which always focus on the thinking mindset to solve problems, are able to obtain decent but usually not optimal solutions in a specified time. However, none of the above methods can balance cost and performance well, especially when the scale of the problem is extensive. Machine learning has been proven to perform extremely well in many fields, so some works suggested applying it to COP, such as reinforcement learning (Mirhoseini et al., 2020), Hopfield neural networks (HNNs) (Hopfield and Tank, 1985), and pointer networks (Vinyals et al., 2015).

HNN is a typical machine-learning-based method. HNN was proposed in the 1980s (Hopfield, 1982). It is a type of fully connected recurrent neural network. Every neuron receives outputs from all other neurons and sends its output to all other neurons. An important property of HNN is Hamiltonian, or energy, which accounts for the stability of HNN. Starting with the initialized weights and neuron states, the energy of HNN keeps decreasing with computation until it reaches a minimum.

Hopfield and Tank (1985) proved that TSP could be solved using HNN. As shown in Fig. 1, when solving a COP with HNN, we need to carefully map the energy of a network to the solution space of the COP. The mapping should ensure that the lower the energy, the better the corresponding solution to the problem. Therefore, we could obtain a sub-optimal/optimal solution when the HNN energy reaches a local/global minimum.

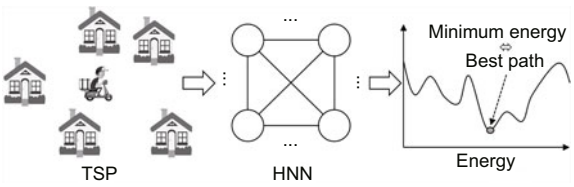

Fig. 1 Solving the traveling salesman problem (TSP) with a Hopfield neural network (HNN)

However, because the HNN energy decreases only during computation, it is easy to get stuck at a locally optimal solution. Thus, avoiding the local optimal becomes the most challenging issue of the optimization problem. Therefore, Chen LN and Aihara (1995) proposed chaotic simulated annealing (CSA), making neurons in HNN receive not only outputs of other neurons but also their own outputs. The self-feedback will decay with time, so HNN should be chaotic at first and tend to stabilize gradually, which expands the exploration space and increases the probability of finding the global optimal solution.

Although the CSA method achieves great performance, it also suffers from the "memory wall" and low parallelism when it is processed on current processors. The processing-in-memory (PIM) architecture based on the memristor crossbar is a potential key to the dilemma. The memristor is an electronic device whose resistance can be adjusted, and it can be easily integrated into a crossbar to compute matrix-vector multiplication efficiently. Many works have applied the memristor PIM architecture to accelerate neural network computation (Chi et al., 2016; Shafiee et al., 2016; Song et al., 2017). Yang et al. (2020) suggested that the memristor PIM had the potential to implement a COP solver using CSA. However, the hardware architecture and many details of implementation, such as quantization and the design of the peripheral circuit, were not taken into consideration. Zhu et al. (2019) studied the quantization technique and PIM architecture design well. Zhu et al. (2019) focused on the multi-layer convolution neural network, while HNN is a fully connected cyclic neural network, so reanalysis is needed. Also, the previous PIM accelerators cannot directly perform CSA efficiently. Co-design of the algorithm and hardware is necessary.

In this paper, we propose an efficient PIM architecture, called COPPER, as the COP solver. COPPER is designed for CSA based on memristor crossbars. Thus, it can run a raw HNN naturally. Users just need to input the parameters of CSA, and COPPER will give an appropriate solution. Our main contributions are as follows:

1. We explore the influence that quantization brings to CSA, and determine the appropriate bit width.

2. We adapt CSA to hardware without reducing

its performance.

3. We design an efficient COP solver with CSA and PIM, and propose a pipeline method.

Experimental results show that the speed and energy efficiency of COPPER are significantly better than those of CPU and GPU.

2 Preliminaries

In this section, we first provide the formulation of CSA and evaluate its performance. Then we introduce the memristor and PIM based on it.

2.1 Chaotic simulated annealing (CAS)

CSA is based on HNN, which is inspired by the Ising model (Cipra, 1987). To explain the phenomenon of ferromagnetic phase transition in physics, Ernst Ising proposed the Ising model. In the Ising model, each neuron can perceive only the neurons adjacent to it. The Ising model succeeded in both physics and computer science. It was discovered that the Ising model could be used to solve the COP, as long as the problem is modeled suitably. Lucas (2014) showed the Ising formulations for many NP problems, most of which are COPs.

Different from the Ising model, HNN is a fully connected cyclic neural network, which means that each neuron can perceive all other neurons. The discrete HNN (DHNN) was proposed first, and the continuous HNN (CHNN) was proposed years after. The former can be used as content-addressed memory, and the latter can be used as a COP solver. Although CHNN is a little different from the Ising model, the formulations for COPs used by Lucas (2014) can also be applied to CHNN. Because each neuron can directly perceive more neurons, CHNN can achieve a better solution in fewer iterations.

CHNN is also troubled by local minima, so Chen LN and Aihara (1995) suggested adding self-feedback to CHNN, which is CAS. Here we use the symbols of Chen LN and Aihara (1995) to describe CSA. As shown in Fig. 2, every neuron i has state y_i and output x_i, and is influenced by the bias I_i. There is a connection weight w_{ij} between any two neurons n_i and n_j, and $w_{ij} = w_{ji}$. The formulation of CSA is

$$x_i(t) = \frac{1}{1 + \mathrm{e}^{-y_i(t)/\epsilon}}, \tag{1}$$

$$\begin{aligned} y_i(t+1) =& ky_i(t) + \alpha \left(\sum_{j=1, j\neq i}^{n} w_{ij} x_j(t) + I_i \right) \\ & - z_i(t)(x_i(t) - I_0), \end{aligned} \tag{2}$$

$$z_i(t+1) = (1-\beta) z_i(t), \tag{3}$$

where ϵ, k, α, β, and I_0 are constants, and z_i is the self-feedback coefficient of neuron i. The original CHNN has no self-feedback; that is, $z_i = 0$. At every tick, every neuron updates its own state and output according to Eqs. (1)–(3). The order of neuron operation is not essential. All neurons can run asynchronously or synchronously in any order.

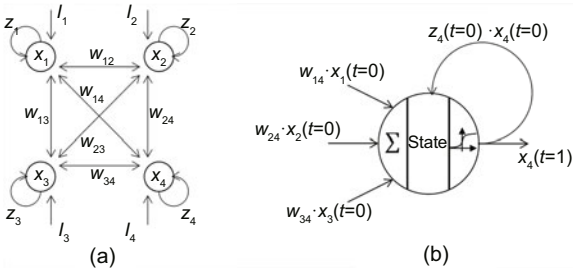

Fig. 2 Chaotic simulated annealing (CSA) (a) and its neuron (b)

CSA has the same energy representation as the CHNN, which is shown in Eq. (4). It can be proved that without self-feedback (like the original HNN), the network energy will gradually decrease with time. When the self-feedback coefficients are not zero, the energy change direction of the network is unpredictable, namely chaotic. After the self-feedback coefficients are all zero, the energy decreases until reaching a local minimum or the global minimum. That is the reason why it is called CSA.

$$E = -\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} x_i x_j - \frac{1}{2} \sum_{i=1}^{n} I_i x_i. \tag{4}$$

2.2 CSA performance

Here we test the CSA performance on TSP. To solve TSP, we need to model the problem. Assuming that there are n cities in all, an $n \times n$ binary matrix $\boldsymbol{X}$ represents a path we choose. In $\boldsymbol{X}$, $x_{ij} = 1$ means that the salesman will visit city i at the j^{th} position. Therefore, the constraints consist of two parts:

1. Hard constraint: any position corresponds to one and only one city; any city corresponds to one and only one position.

2. Soft constraint: the shorter the path length, the better the path.

Now we give the whole constraint, that is,

$$E = \frac{W_1}{2}\left\{\sum_{i=1}^{n}\left(\sum_{j=1}^{n}x_{ij}-1\right)^2+\sum_{j=1}^{n}\left(\sum_{i=1}^{n}x_{ij}-1\right)^2\right\} + \frac{W_2}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}(x_{kj+1}+x_{kj-1})x_{ij}d_{ik}, \tag{5}$$

where d_{ik} represents the distance between city i and city k. The first part of the right-hand side of Eq. (5) is the hardware constraint, which should be zeros when $\boldsymbol{X}$ represents a feasible solution. The second part is the software constraint, which should be smaller when the solution is better. By adjusting W_1 and W_2, we can indicate whether we prefer to obtain a feasible solution or a better solution.

By matching the coefficients of every x in Eqs. (4) and (5), we can compute the values of w_{ij} and I_i needed by CSA, and then perform calculations according to Eqs. (1)–(3). CSA will return a solution after reaching stability. The initial state can be set randomly.

We evaluate the performance of CSA on different numbers of cities. Positions of cities are generated randomly, and coordinates are in 0–1. Table 1 shows the constant parameters we set to test the performance of CSA. For problems of different scales, we use different constant parameters. Constant parameters used in the problem with 10 cities are from Yang et al. (2020), and we fine-tune constant parameters for the others. As the number of cities increases, we tend to obtain feasible solutions (by decreasing W_2) and increase the amount of annealing (by decreasing β and increasing z_0) to explore a larger solution space.

Table 1 Constant parameters used in chaotic simulated annealing when solving the traveling salesman problem

City number	W_1	W_2	k	α	β	ϵ	z_0	I_0
10	1.0	1.0	1.0	0.015	0.0050	1/256	0.08	0.65
20	1.0	1.0	1.0	0.015	0.0015	1/256	0.08	0.65
30	1.0	1.0	1.0	0.015	0.0005	1/512	0.08	0.75
40	1.0	0.5	0.9	0.015	0.0004	1/512	0.10	0.55
50	1.0	0.5	0.9	0.015	0.0003	1/512	0.10	0.50

Table 2 shows the results. The best paths are obtained using Concorde, which is typical software to solve TSP. For each scale, we apply CSA 100 times and the simulated annealing (SA) algorithm 100 times for comparison. For fairness, we set different annealing rates for SA so that the amount of annealing approximately equals the number of iterations of CSAs. We show the best and average path lengths found by CSA and SA.

According to Table 2, CSA needs multiple iterations to reach the stable state, and it does not always obtain a feasible solution. As for the quality of the solution, CSA is better than SA. In fact, the solution quality is still not as good as that in Chen LN and Aihara (1995), because we do not spend much time in adapting the constant parameters. However, we can see that CSA can obtain good solutions, although the scale of the problem increases.

2.3 Memristor-based processing-in-memory architecture

Because CSA is fully connected, the amount of computation required will increase rapidly, which is not something that current general-purpose processors can handle easily. In fact, many neural networks are troubled by such extensive computation. Too many calculations are needed, although most of the calculations are similar. Therefore, PIM is proposed which intends to perform some simple operations in memory. PIM is created with a high degree

Table 2 Performance of chaotic simulated annealing (CSA) and simulated annealing (SA)

City number	Real best	Iteration number	P_{feasible} (%)	Best		Average	
				CSA	SA	CSA	SA
10	1.768	906	96	**1.768**	**1.768**	1.776	**1.773**
20	4.172	1187	91	**4.172**	**4.172**	**4.316**	4.647
30	4.293	2962	96	**4.565**	4.575	**4.838**	4.980
40	5.451	4855	99	**5.502**	5.726	**5.705**	6.364
50	5.939	6903	99	**6.086**	6.444	**6.379**	7.180

The column "iteration number" is the average number of iterations needed by CSA to reach stability, and P_{feasible} means the percentage of obtaining a feasible solution using CSA. The bold values represent the better solutions from CSA and SA

of parallelism and does not require much time for fetching data.

A memristor is very suitable for implementing PIM. The memristor crossbar can be used to efficiently perform matrix-vector multiplication in memory. We know that $I = U/R$ according to Ohm's law. Now we can control the voltage U and the resistance R; that is, we can calculate $I = U\frac{1}{R}$. Because the currents can be accumulated just by putting them into the same wire, we can calculate $\sum_{i=1}^{n} U_i \frac{1}{R_i}$. We can regard U as the input and $\frac{1}{R}$ as the weight, and accumulate the current to obtain the result of matrix-vector multiplication, as shown in Fig. 3. Of course, Fig. 3 is simplified, and some details, such as conversion between digital and analog, are not shown. Using a memristor crossbar, we need not fetch weight frequently and can perform a lot of calculations at once, which is highly efficient.

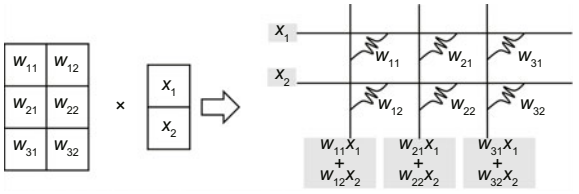

Fig. 3 Memristor crossbar for multiplication

Yang et al. (2020) attempted to implement CSA using a memristor crossbar. In this work, the weights and self-feedback coefficients were written into the memristor cells, and other operations were simulated. The results showed that the memristor crossbar can perform CSA correctly. In addition, this work explored the convergence when self-feedback coefficients decayed with linear, exponential, and long-term depression, separately. However, the programming of the memristor cell is time-consuming (Chen JR et al., 2020), so the time cost will increase rapidly if the memristor cell is adjusted frequently. The aim of Yang et al. (2020) was to verify the feasibility, so many practical problems, such as hardware architecture, scale, and algorithm quantization, were not taken into consideration. In the following sections, we will address these problems.

3 Algorithm

In this section, we first quantize CSA and find the proper bit width. Then we propose schemes to adapt CSA to hardware implementation. Finally, we evaluate the performance of the quantization and schemes on the max-cut problem.

3.1 Quantization

Quantization is necessary before implementing CSA on special hardware. Because matrix-vector multiplication is deployed on memristor crossbars, quantization of the weight and output should be taken into consideration. We analyze the influence of quantization by solving a 30-city TSP 100 times. The positions and constant parameters are the same as in Section 2.2. We pre-calculate float32 weights with float32 position coordinates and then quantize the weights into different bit widths for subsequent computing. Here we use symmetric uniform quantization, which is easy to implement.

Fig. 4 shows the influence of quantization of the weight and output. The solid line represents the ratio between the path lengths of float CSA and quantized CSA. The dotted line represents the rate of obtaining a feasible solution. An interesting note is that quantized CSA sometimes obtains better solutions (the length ratio of float path to quantized path >1), which may be owing to the randomness. We can see that CSA is more sensitive to the quantization of the weight than to that of output. A weight of low bit width may incur a high failure ratio and solutions with bad performance.

We think that the weight determines the trend of the network energy change. After the weight is quantized with too few bits, the network cannot obtain the right direction in every iteration, so it will

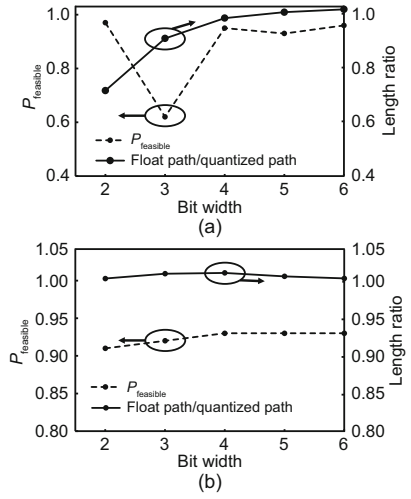

Fig. 4 Quantization influence: (a) weight quantization; (b) output quantization

fail at last. In addition, fewer bits erase the gap between a longer and a shorter distance, so the network cannot perceive the better path. According to Fig. 4, we suggest using 4 bits for the weight. The finer the output granularity, the more the states that each neuron can have, and the larger the space that the entire network can explore. Also, CHNN and CSA have richer dynamic characteristics, which increase their adaptability. Therefore, we suggest using 8 bits for the output. The following subsections are based on 4-bit weight and 8-bit output.

3.2 Adaptation

We will propose some schemes in this subsection to adapt CSA to hardware.

The implementation of the sigmoid function is very expensive. Fortunately, fewer bits are enough for the output according to Section 3.1, so we could use a lookup table (LUT) to reduce the cost.

Because the output of the sigmoid is between 0 and 1 and we use 8 bits to represent the output, the input of sigmoid, assumed to be a, needs only to be between -5.54 and $+5.54$. According to Eq. (1), we have $a = y/\epsilon$. We can specify ϵ as $-p$ power of 2, which does not influence the performance of CSA, so the division could be transformed into a shift. Note that we do not change y but just use it to calculate a. The result a will be truncated into 8 bits and be used as the sigmoid LUT. Therefore, Eq. (1) could be replaced by

$$a_i = \text{Truncate}(y_i(t)<<p),\ x_i(t) = \text{LUT}(a_i). \quad (6)$$

In addition, z decays exponentially in Eq. (3), which is not hardware-friendly. We can replace exponential decay with linear decay. That is, Eq. (3) could be replaced by

$$z_i(t+1) = z_i(t) - \delta. \quad (7)$$

We apply the adaptation to the 30-city TSP. As shown in Table 3, linear decay has little effect on the algorithm.

Table 3 Influence of the adaptation

Scheme	P_{feasible} (%)	Iteration number	Length
Original	96	2962	4.838
Linear decay	93	3081	4.875
Output stability	92	2093	4.806

P_{feasible}: the percentage of obtaining a feasible solution

Eq. (2) could also be transformed into

$$\begin{aligned} y_i(t+1) =& ky_i(t) + \sum_{j=1,j\neq i}^{n} w'_{ij}x_j(t) + I'_i \\ & - z_i(t)(x_i(t) - I_0), \end{aligned} \quad (8)$$

where $w'_{ij} = \alpha w_{ij}$ and $I'_i = \alpha I_i$.

Although it seems typical, we do not fuse the self-feedback and weight, because modifying the weight, represented by the conductance of a memristor, takes a lot of time.

Another problem we need to consider is discontinuing computation in time after the network becomes stable. The stability of the network means that every value does not change again, including the energy of the network and the states and outputs of neurons. According to Eq. (4) or (5), the calculation of energy is especially complex, so we prefer using output as the criterion. To reduce the overhead of comparison between the last and the current outputs, we suggest just contrasting their highest bit. This is not absurd, because the final outputs always converge to 0 or 1. We believe that if the outputs do not change 10 times in a row, the network becomes stable. According to Table 3, the results of comparison between rounding values can represent the raw results very closely. In addition, it is shown that using output rounding values can evidently reduce the number of iterations, because the redundant iterations just change the output from >0.5 to 1.0 in most cases.

3.3 Performance on the max-cut problem

CSA can solve many COPs directly as long as the energy function is suitably constructed. Also, many COPs can be transformed into each other, so CSA can also solve COPs indirectly by transforming them into those that can be solved.

To evaluate the feasibility of CSA and the above schemes on other COPs, we carry out experiments on the max-cut problem, which is also an NPC problem. The max-cut problem is to divide the node set of a graph into two complementary subsets so that the number of edges between the two subsets is maximum. Assuming that the graph includes n nodes, the following formulation can be used to evaluate a segmentation:

$$\text{score} = \sum_{i<j}^{n} a_{ij}\frac{1-(2x_i-1)(2x_j-1)}{2}, \quad (9)$$

where i and j represent the nodes. $a_{ij}=1$ if node i connects with node j; $a_{ij}=0$, otherwise. $x_i \in \{0,1\}$ reflects the subset node to which i belongs.

According to Eq. (9), the energy function of the max-cut problem is as follows:

$$E=\sum_{i<j}^{n} a_{ij}(2x_ix_j-x_i-x_j). \tag{10}$$

By building a Hopfield network of n nodes, and comparing Eqs. (4) and (10), we can see that $w_{ij}=-4a_{ij}$ and $I_i=\sum_j^n a_{ij}$.

Fig. 5 is a graph whose edges are randomly generated. Fig. 5 also shows one of the optimal segmentations, and the number of edges that are cut through is 12. We evaluate CSA with different schemes 100 times, and the results are shown in Table 4. The constant parameters in the experiments are consistent with those in the previous TSP problems of 10 cities. Weights and outputs are quantized with 4 and 8 bits, respectively. Obviously, when using various schemes, CSA can still solve the max-cut problem well.

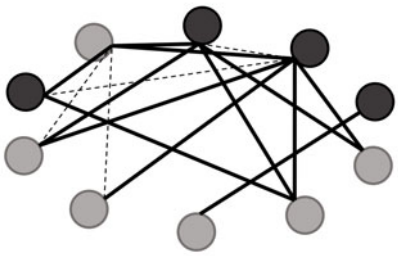

Fig. 5 Max-cut problem

Table 4 Chaotic simulated annealing performance on max-cut

Scheme	Iteration number	Best	Best ratio (%)	Average
Original	232	12	100	12.00
Quantization	338	12	100	12.00
Linear decay	252	12	100	12.00
Output stability	167	12	99	11.99
All*	130	12	99	11.97

*Applying all schemes

4 Architecture design

4.1 Overview

COPPER is composed mainly of a subprocessor-element (subPE) array and peripheral circuits, as shown in Fig. 6. Each subPE can calculate matrix-vector multiplication independently. The peripheral circuit, responsible for the parts except for matrix-vector multiplication, is implemented using traditional CMOS circuits. With the increase of the scale of CSA, the scale of subPE increases in the square while the peripheral circuit increases linearly. That is, each column of subPE is equipped with one peripheral circuit. COPPER covers only the operation of the network, so the calculation of weights is the concern of the host. In consideration of the sign of parameters, the weights may need equivalent linear transformation to make them all non-negative numbers. This was discussed in Yang et al. (2020).

After parameters are written, COPPER starts running until the network stabilizes or reaches the upper limit of the cycle. During operation, data will be transmitted between subPEs. Specifically, a subPE communicates from left to right and from top to bottom, which can save the bandwidth between the subPE and buffer. The pipeline between the subPE and peripheral circuit can save time, and the delay of the peripheral circuit is covered by the subPE.

4.2 SubPE

Each subPE has 16 128×128 memristor crossbars, digital-to-analog converters (DACs), analog-to-digital converters (ADCs), sample-and-hold (S+H) devices, registers, and shift-and-add (S+A) units.

Due to IR drop, thermal noise, random telegraph noise, and so on, memristor crossbars may not always work ideally, which will result in computing errors. It is not mature to write a 4-bit weight into just one cell, which may lead to more errors. However, with the help of the compensation circuit and other means, memristors composed of single-bit cells can run stably. In Hung et al. (2022), the system-level inference accuracy of a ResNet-20 model did not decrease at all. Therefore, we split one 4-bit weight into four cells. Similarly, we divide one input into 8 bits and send one bit into the crossbar at a time, because using an 8-bit DAC is risky. In this way, one bit of the input produces 4 bits, which should be put in appropriate positions, separately. One input produces eight groups of outputs, which should be shifted and accumulated. In fact, this is the same as the column multiplication of binary multiplication. In a word, we spend more time on inputs and more

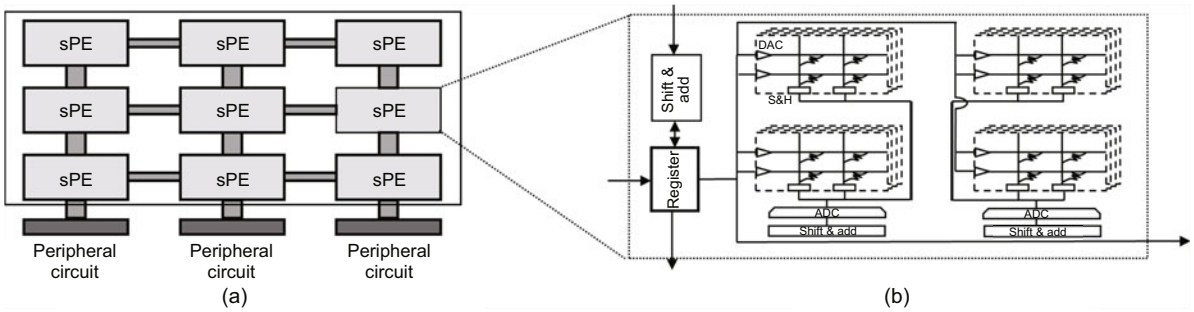

Fig. 6 COPPER overview (a) and subPE (b)

hardware on weights, but achieve perfect stability.

Because each row of subPEs shares the same input, we make each subPE receive the input from left and immediately transmit the input to the subPE on right. The leftmost subPE obtains data from the data line. Because the results of each column of subPEs will be accumulated together, we make each subPE receive the result from top, accumulate it in its own result, and send the new result to the subPE below. The results received by the uppermost subPE are all zeros. Therefore, we can use different numbers of subPEs to adapt to problems with different scales.

4.3 Peripheral circuit

The peripheral circuit starts after subPEs provide the result. In detail, the peripheral circuit will complete the remaining computation and judge whether the network reaches stability. Fig. 7 shows the main data path of the peripheral circuit. As mentioned in Section 3.2, we make self-feedback z decay linearly and replace the sigmoid function with LUT. The final results are x_{t+1}, y_{t+1}, and z_{t+1}. We use the highest bit of x_t and x_{t+1} for comparison to confirm that the network is stable.

4.4 Pipeline

Because each iteration of CSA depends on the results of the previous iteration, it seems that there is no way to pipeline. However, considering the possible failure of CSA and the fact that the optimal solution cannot be obtained every time, we may need to run CSA several times to obtain an excellent solution for a particular problem. Therefore, we can run two CSAs for one problem at a time, without the CSAs interfering with each other, which is a pipeline not for different iterations in the same network but for different networks.

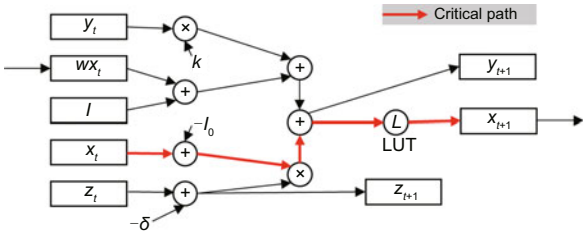

Fig. 7 Peripheral circuit data path (References to color refer to the online version of this figure)

Specifically, for the same problem, the weight and bias are exactly the same, but the initial state is different, so we do not need to rewrite the weight and bias. We can write the first batch of the initial state $y_1(t=0)$, calculate $x_1(t=0)$, and put $x_1(t=0)$ into subPEs. After obtaining $wx_1(t=0)$, we input the second batch of the initial state $y_2(t=0)$, calculate $x_2(t=0)$, and put $x_2(t=0)$ into subPEs. At the same time, we use $wx_1(t=0)$ to calculate a new $y_1(t=1)$, so we can realize the pipeline. By simply incurring some costs to store two sets of constant parameters, we can obtain twice solutions in the same amount of time.

5 Evaluation

5.1 Experiment setup

Because there is no work designing a special hardware architecture for CSA, we compare COPPER with the general CPU and GPU processors. For TSPs with different numbers of cities, we run CSA on CPU and GPU, and estimate the energy cost and speed. The CPU is Intel® Core™ i7-6700K CPU @ 4.00 GHz with 32 GB RAM, GPU 1 is NVIDIA GeForce GTX 1080 Ti, and GPU 2 is NVIDIA A100

80 GB. We use the Python packages pyRAPL and pynvml to estimate the cost of CPU and GPU, respectively. As for COPPER, we use 10×10 subPEs, which can solve a 50-city TSP, in evaluation. The parameters in Table 5 are from Shafiee et al. (2016).

Obviously, as the scale of the problem increases, the workload of peripheral circuits increases linearly, while the subPE sustains square growth. As shown in Table 5, each column of subPEs could obtain no more than 256 products in 1.2×2^{-21} s. It is enough for the peripheral circuit to perform 256 times in 1.2×2^{-21} s. That is, the frequency should be 600 MHz. The critical path (shown as the red line in Fig. 7) contains two additions, one multiplication and an LUT, so the peripheral circuit can easily reach 600 MHz. Therefore, matrix-vector multiplication is the main bottleneck, and COPPER does not require high-speed peripheral circuits, so we focus only on the comparison of matrix-vector multiplication.

Table 5 Device parameters

Device	Number	Power	Area (mm^2)	Note
ADC	16	32 mW	0.019 20	1.2 GS/s
DAC	2048	8 mW	0.000 34	1 bit
S+H	2048	20 μW	0.000 08	
Crossbar	16	4.8 mW	0.000 40	
S+A	2048	0.4 mW	0.000 48	
Register		2.94 mW	0.005 74	516 KB
Total	100	4.82 W	5.248	

GS/s: giga samples per second

Table 6 Comparison of COPPER with CPU and GPU

City number	Speedup			Efficiency ratio		
	CPU	CPU 1	GPU 2*	CPU	CPU 1	GPU 2*
10	7.23	15.3	19.3	4.65	15.5	22.4
20	19.0	18.7	13.3	18.9	21.1	17.2
30	42.3	19.4	14.0	52.3	52.4	23.3
40	531	49.9	15.6	226.7	142.3	33.6
50	1510	128	21.1	581.4	370.4	57.2

* Here, CSA on GPU 2 is slower when the scale is 10, compared with other scales. Because we call GPU using PyTorch instead of the CUDA codes directly, perhaps PyTorch calls different kernels

5.2 Speed

We run CSA 10 000 times and compute the average time cost on CPU and GPU. According to Table 6, as the scale of the problem increases, the acceleration of COPPER becomes obvious.

The practical speedup should not be as significant, because many details, such as communication, buffer, and synchronization, are not taken into consideration. However, the obvious speedup is predictable due to the significant reduction in communication, and the bigger the scale, the more obvious the speedup.

5.3 Energy efficiency

We run CSA 10 000 times and compute the average energy cost on CPU and GPU. For CPU, the Python package pyRAPL can directly estimate the energy cost of the selected codes. For GPU, the Python package pynvml can monitor the real-time power, and we calculate the energy by multiplying the power and time used.

As shown in Table 6, COPPER is much more efficient than CPU and GPU. With the increase of the problem scale, the improvement of efficiency is more and more obvious. We think that there are two main reasons. First, we assume that all subPEs are running, even if the problem scale is not enough to use all crossbars, so the bigger the problem, the better it is for COPPER. Second, with the increase of the problem scale, general-purpose processors spend much more energy on data handling and cyclic calculation, while COPPER holds the calculation in memory, which significantly reduces the energy cost.

5.4 Comparison with SA

Although COPPER can implement CSA efficiently, it is necessary to compare COPPER with other algorithms on general-purpose processors. Here, we test the time and energy cost of SA algorithm on CPU. Because the SA algorithm is not computation-intensive, GPUs are not in the test. The SA setup is the same as in Section 2.2, and its time and energy are still from the Python package pyRAPL. The time of CSA on COPPER equals the average iteration number divided by the frequency of COPPER, and the energy equals the value of the time times the power. Note that the energy of the peripheral circuit is ignored because this is negligible (Li et al., 2022).

Table 7 shows the results. Compared with SA on the CPU, COPPER still exhibits clear speedup but fails in energy. The reason is that CSA contains a lot of computation (matrix-vector multiplication), while SA does not. COPPER reduces the time by parallel

computing, while the energy needed is still so much. However, according to Tables 2 and 7, COPPER has advantages in speed and quality of the solution, so it is very competitive.

Table 7 Speed and efficiency of CSA compared to those of SA

City number	Speedup	Efficiency ratio
10	74.6	0.02
20	103	0.04
30	120	0.12
40	139	0.23
50	166	0.39

6 Related works

The Ising model was proposed for the phenomenon of ferromagnetic phase transition in physics. It regards the object as a d-dimensional lattice, and each lattice site represents a small magnet. Adjacent magnets interact with each other, which could be considered as coefficients/weights. The whole system has a Hamiltonian/energy, which is related to weights, magnet properties, and external magnetic fields.

After it was discovered that the Ising model could be used to solve COPs, a series of hardware was designed, such as quantum annealing (Johnson et al., 2011; Shin et al., 2014) and CMOS annealing (Yamaoka et al., 2016; Takemoto et al., 2019, 2021; Yamamoto et al., 2020). They all strived to simulate more and more lattice sites to solve problems on a larger scale, and Takemoto et al. (2021) even implemented 1.44×10^5 lattice sites. To escape from local minima, these designs introduced various annealing mechanisms. For example, Yamaoka et al. (2016) and Takemoto et al. (2019) made the state of spin, corresponding to the small magnet in the Ising model, inverse randomly, while Takemoto et al. (2021) used probabilistic inversion. In addition, they equipped tens of thousands of spins to handle large-scale problems.

7 Conclusions and discussion

To solve combinatorial optimization problems, we improve the chaotic simulated annealing algorithm and design COPPER. COPPER uses advanced storage technology to improve the parallelism of operation and obtains great performance in power consumption and speed. COPPER is scalable and can solve large-scale COPs. In addition, COPPER can not only run chaotic simulated annealing, but also support the original continuous Hopfield neural network for other applications.

However, we also see that the Ising model, CHNN, and CSA all require a high degree of parallelism, while the current hardware does not support reuse, so the hardware scale must increase with the increase of the problem scale. Physically, the hardware scale cannot increase without limits. So far, the biggest ReRAM nonvolatile PIM macro has been 8 Mb (Hung et al., 2022). Using one such macro, we can solve the TSP with 37 cities at most when quantizing the weights into 4 bits. Furthermore, using more macros, TSP with more cities can be solved. Meanwhile, except for the development of hardware, supporting temporal reuse of hardware or implementing CSA using quantum may be a promising direction.

Contributors

Qiankun WANG led the research and was mainly responsible for implementing the algorithm, designing the hardware, and drafting the paper. Xingchen LI provided the design ideas and some data for the hardware part. Bingzhe WU sorted out the algorithm and pointed out the possibility of combining software and hardware. Ke YANG and Yuchao YANG laid the foundation for this research and provided some parameters for the algorithm. Wei HU provided the stability analysis of ReRAM and the latest research progress of ReRAM PIM macro. Guangyu SUN made many suggestions on the research, and revised and finalized the paper.

Compliance with ethics guidelines

Qiankun WANG, Xingchen LI, Bingzhe WU, Ke YANG, Wei HU, Guangyu SUN, and Yuchao YANG declare that they have no conflict of interest.

Data availability

The data that support the findings of this study are available from the corresponding author upon reasonable request.

References

Chen JR, Wu HQ, Gao B, et al., 2020. A parallel multibit programing scheme with high precision for RRAM-based neuromorphic systems. *IEEE Trans Electron Dev*, 67(5):2213-2217. https://doi.org/10.1109/TED.2020.2979606

Chen LN, Aihara K, 1995. Chaotic simulated annealing by a neural network model with transient chaos. *Neur Netw*, 8(6):915-930. https://doi.org/10.1016/0893-6080(95)00033-V

Chi P, Li SC, Xu C, et al., 2016. PRIME: a novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. *ACM SIGARCH Comput Archit News*, 44(3):27-39. https://doi.org/10.1145/3007787.3001140

Cipra BA, 1987. An introduction to the Ising model. *Am Math Mon*, 94(10):937-959. https://doi.org/10.1080/00029890.1987.12000742

Hopfield JJ, 1982. Neural networks and physical systems with emergent collective computational abilities. *Proc Natl Acad Sci USA*, 79(8):2554-2558. https://doi.org/10.1073/pnas.79.8.2554

Hopfield JJ, Tank DW, 1985. "Neural" computation of decisions in optimization problems. *Biol Cybern*, 52(3):141-152. https://doi.org/10.1007/BF00339943

Hung JM, Huang YH, Huang SP, et al., 2022. An 8-Mb DC-current-free binary-to-8b precision ReRAM nonvolatile computing-in-memory macro using time-space-readout with 1286.4-21.6TOPS/W for edge-AI devices. IEEE Int Solid-State Circuits Conf, p.1-3. https://doi.org/10.1109/ISSCC42614.2022.9731715

Johnson MW, Amin MHS, Gildert S, et al., 2011. Quantum annealing with manufactured spins. *Nature*, 473(7346):194-198. https://doi.org/10.1038/nature10012

Karp RM, 1972. Reducibility among combinatorial problems. In: Miller RE, Thatcher JW, Bohlinger JD (Eds.), Complexity of Computer Computations. Springer, New York, USA, p.85-103. https://doi.org/10.1007/978-1-4684-2001-2_9

Li XC, Yuan ZH, Sun GY, et al., 2022. Tailor: removing redundant operations in memristive analog neural network accelerators. Proc 59th ACM/IEEE Design Automation Conf, p.1009-1014. https://doi.org/10.1145/3489517.3530500

Lucas A, 2014. Ising formulations of many NP problems. *Front Phys*, 2:5. https://doi.org/10.3389/fphy.2014.00005

Mirhoseini A, Goldie A, Yazgan M, et al., 2020. Chip placement with deep reinforcement learning. https://arxiv.org/abs/2004.10746

Shafiee A, Nag A, Muralimanohar N, et al., 2016. ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Comput Archit News*, 44(3):14-26. https://doi.org/10.1145/3007787.3001139

Shin SW, Smith G, Smolin JA, et al., 2014. How "quantum" is the D-wave machine? https://arxiv.org/abs/1401.7087

Song LH, Qian XH, Li H, et al., 2017. PipeLayer: a pipelined ReRAM-based accelerator for deep learning. IEEE Int Symp on High Performance Computer Architecture, p.541-552. https://doi.org/10.1109/HPCA.2017.55

Takemoto T, Hayashi M, Yoshimura C, et al., 2019. 2.6 A 2×30k-spin multichip scalable annealing processor based on a processing-in-memory approach for solving large-scale combinatorial optimization problems. IEEE Int Solid-State Circuits Conf, p.52-54. https://doi.org/10.1109/ISSCC.2019.8662517

Takemoto T, Yamamoto K, Yoshimura C, et al., 2021. 4.6 A 144Kb annealing system composed of 9×16Kb annealing processor chips with scalable chip-to-chip connections for large-scale combinatorial optimization problems. IEEE Int Solid-State Circuits Conf, p.64-66. https://doi.org/10.1109/ISSCC42613.2021.9365748

Vinyals O, Fortunato M, Jaitly N, 2015. Pointer networks. Proc 28th Int Conf on Neural Information Processing Systems, p.2692-2700.

Yamamoto K, Ando K, Mertig N, et al., 2020. 7.3 STATICA: a 512-spin 0.25M-weight full-digital annealing processor with a near-memory all-spin-updates-at-once architecture for combinatorial optimization with complete spin-spin interactions. IEEE Int Solid-State Circuits Conf, p.138-140. https://doi.org/10.1109/ISSCC19947.2020.9062965

Yamaoka M, Yoshimura C, Hayashi M, et al., 2016. A 20k-spin Ising chip to solve combinatorial optimization problems with CMOS annealing. *IEEE J Sol-State Circ*, 51(1):303-309. https://doi.org/10.1109/JSSC.2015.2498601

Yang K, Duan QX, Wang YH, et al., 2020. Transiently chaotic simulated annealing based on intrinsic nonlinearity of memristors for efficient solution of optimization problems. *Sci Adv*, 6(33):eaba9901. https://doi.org/10.1126/sciadv.aba9901

Zhu ZH, Sun HB, Lin YJ, et al., 2019. A configurable multi-precision CNN computing framework based on single bit RRAM. Proc 56th Annual Design Automation Conf, Article 56. https://doi.org/10.1145/3316781.3317739