Design Automation for Embedded Systems (2021) 25:325–351
https://doi.org/10.1007/s10617-021-09255-9

A hardware/software partitioning method based on graph convolution network

Xin Zheng[1,2] · **Shouzhi Liang**[1] · **Xiaoming Xiong**[1]

Received: 24 January 2021 / Accepted: 12 October 2021 / Published online: 30 October 2021

Abstract
Hardware/software (HW/SW) partitioning is the crucial step in HW/SW co-design, which can significantly reduce the time-to-market and improves the performance of an embedded system. Due to that the majority of previous works have large exploration time and generate often low-quality solutions for large scale systems, we propose a fast HW/SW partitioning approach based on graph convolution network (GCN) to address this problem. To the best of our knowledge, it is a new partitioning method based on GCN which is a gradient-based optimization approach. It can aggressively speed up the partitioning process. To quantify the quality of solutions, the scheduling is integrated into the partitioning process. The experiment results show that not only does our proposed method outperform existing metaheuristics approaches in terms of the efficiency (e.g., 18× faster than Kernighan–Lin algorithm for the task graphs with 1000 nodes), but it also improves the quality of HW/SW partitioning (e.g., more than 10% acceleration ratio (AR) improvement for the 1000 nodes graphs).

Keywords HW/SW partitioning · Graph convolution network · Scheduling

1 Introduction

HW/SW co-design is the cooperative development of the hardware and software components to achieve the best performance of an embedded system. Since the 1980s, HW/SW Co-design has become a new methodology to design complex embedded systems, which includes system specification and modeling, HW/SW partitioning, system verification and implementation [17,45]. One of the key topics to be investigated in HW/SW co-design is HW/SW partitioning [17], which is the problem of dividing a system into software parts and hardware parts that satisfy the design constraint. The software part will execute on processors to provide flexibility and reduce the cost of the system. The hardware part runs as parallel circuits on ASIC or

✉ Xiaoming Xiong
xmxiong@gdut.edu.cn

Xin Zheng
xinzheng9209@gmail.com

[1] School of Automation, Guangdong University of Technology, Guangzhou, China

[2] School of Computer and Information Science, Linköping University, Linköping, Sweden

FPGA to enhance the performance of the system. The aim of HW/SW partitioning is to find a design implementation that optimizes the objectives (hardware area, execution time, power, and so on) of the system under design constraints.

Generally, the HW/SW partitioning problem can be formalized as a bi-partitioning problem to find the global optimal solution [14]. Since it is an NP-hard problem and cannot find the optimum result in polynomial time, various heuristic methods are proposed to solve this problem [3]. There are mainly two categories for partitioning approaches: exact solutions and heuristic solutions. The family of exact solutions includes dynamic programming (DP) [20,34,43,48], integer linear programming (ILP) [3,38,50,55], and branch and bound [7,21,36]. An exact algorithm performs an exhaustive search in the solution space and finds the optimal result. Since the search strategy of the exact algorithm is similar to the exhaustive method, the search time of the exact method will increase approximately exponentially with the increase of the number of nodes ($O(2^N)$). Thus, these exact solutions are successful only for an HW/SW partitioning problem of a system with a small number of tasks [17]. When the scale of the problem gets larger, exact solutions tend to be inefficient. Trindade and Cordeiro [46] shows that the limit of ILP is that the number of task nodes is less than or equal to 400. To this end, many heuristic methods are proposed to solve large-scale problems, which can be simplified into two categories: specific heuristics and general heuristics. For specific heuristics, there are two classical and complementary domain-specific heuristic approaches at the early stage of HW/SW co-design [11,13]. The former is a software-oriented approach based on time-constraint, and the latter is a hardware-oriented approach based on performance-constraint. After that, the specific heuristics approaches, including Global Criticality/Local Phase (GCLP), Mapping, Implementation Bin, and Schedule (MIBS) [23], Hardware-Oriented Partitioning (HOP), and Consistency and Hardware Oriented Partitioning (CHOP) [24], are proposed for multi-function partitioning problems. For the general-purpose heuristics (metaheuristics), such as genetic algorithm (GA) [42,44,54], simulated annealing (SA) [5,10,16,33], and tabu search (TS) [10,19], which can achieve better partitioning solutions for complex systems compared with the specific heuristics approaches. Moreover, the efficiency of general-purpose heuristics for large-scale systems will also decrease [17]. There are also other well-known metaheuristics methods for partitioning, such as clustering [1], expert systems [33], neural networks [12], and the Kernighan–Lin algorithm [35]. The family of these metaheuristic algorithms can also achieve a better solution for some specific context.

However, with the increasing complexity of embedded systems, the scale of system design becomes larger, and the number of partitioning alternatives will extremely increase. Although the problem of HW/SW partitioning has been studied for many years, for large-scale systems, the quality of HW/SW partitioning directly affects the time and cost of product launch. The increased complexity can also be attributed to the communication-intensive applications, which will increase the consuming time of partitioning. On the other hand, the evaluation criteria have not been uniformed, and there is still a huge gap between the existing methods and the practical applications. In these cases, the traditional partitioning methods cannot achieve high efficiency while ensuring the quality of partitioning. Therefore, the efficiency of partitioning for large-scale systems is still an important issue to be solved.

Motivated by the above problem, this paper concentrates on fast generating a partition solution for large-scale systems. As the HW/SW partitioning problem can be formulated as the node classification problem. GCN can rapidly extract features from graph data and be successfully applied to various fields, we can naturally cast the partitioning problem into a node classification problem and leverage the learning-based approaches associated with GCN. For the problem with graph-structure tasks, GCN has been widely used in node classification in recent years [52]. This observation motivates that it is worth exploring how

to design a fast partitioning model. It is a new idea to apply GCN to partitioning problems, which is different from the traditional heuristic algorithm. GCN is a gradient-based approach, which can effectively process graph data and aggregate the features of neighbor nodes to generate new node representations. Since the operations of GCN are matrix calculations, it can converge rapidly to realize the node classification efficiently. To this end, firstly, we formulate the partitioning problem as an optimization problem to minimize the execution time of all tasks under a hardware area constraint. Secondly, a gradient-based approach—GCN is utilized to solve this optimization problem. We also integrate the scheduling into the partitioning process, which is used to quantify the quality of solutions and also improve the task execution efficiency.

The main contributions of this work can be summarized as follows:

- Based on GCN, a fast HW/SW partitioning method—Graph Convolution Partitioning and Scheduling (GCPS) for large-scale systems is proposed.
- We demonstrate a strong empirical performance of the proposed GCPS. These experiments show that our approach can rapidly achieve high-quality solutions for large-scale partitioning problems.

The rest of the paper is organized as follows. Section 2 describes the related work of HW/SW partitioning. Section 3 describes the problem. Section 4 proposes a new partitioning method and implementation based on GCN. Section 5 presents the experiment and analysis, and a real-life example is proposed to verify our approach. Section 6 concludes the paper.

2 Related work

2.1 HW/SW partitioning

A substantial amount of researches has been investigated on HW/SW partitioning. For small scale problems in HW/SW partitioning, some exact algorithms are proposed to obtain an exact solution via the exhaustive search in the feasible space [3,7,20,21,34,36,38,43,48,50,55]. In [38], an algorithm that is able to solve HW/SW partitioning problem using integer programming is presented. The better metrics lead to nearly optimal results, and fewer iteration steps can decrease calculation time. Wu and Srikanthan [48] proposes an algorithm that utilizes dynamic programming principles and considers communication delays. It is shown that the time complexity of the latest algorithm has been reduced without an increase in space complexity. In [21], a hybrid branch and bound strategy based on global best-first (GBF) scheme and local best-first (LBF) scheme is proposed to produce fewer expanded nodes and reduce the search space. On the other hand, with the increasing scale and the complexity of designing embedded systems, it is very difficult to find an exact solution for large-scale problem in a limited time. To this end, heuristic algorithms or combination of exact and heuristic algorithms are proposed to obtain a solution with high quality in a reasonable time [3,5,10,16,19,33,42–44,50,54]. In [49], the HW/SW partitioning problem is simplified to a variation of the knapsack problem that is approximately solved by searching 1D solution space to reduce time complexity. The SA approach, based on a greedy strategy, is put forward in [22]. In this approach, ten initial solutions with high quality are generated for SA by the greedy strategy, which can have a better initial position in the search space to converge rapidly. In the work of [27], a hybrid algorithm of GA and TS is studied, and the hardware orientation is proposed to reduce the number of iterations and generate a better initial solution which will be used for TS to search the approximate global optimal result. The above

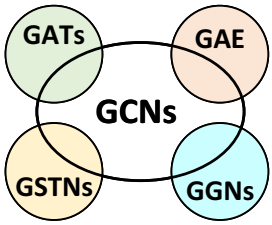


Fig. 1 The relationship among graph neural networks

approaches mainly combine two methods to overcome the defects of a single approach and are better solved than the general algorithms.

However, as these methods are based on the probability try out and have large exploration space for large-scale problems, it is difficult for them to consider both efficiency and solution quality. Many approaches are also designed for some specific problems; they need to be redesigned if the application context is changed. Whereas we proposed a gradient-based optimization approach, which is different from these heuristic methods, and it can generate high-quality solutions very fast for large-scale problems.

2.2 GCN

Motivated by the convolution neural network (CNN) in deep learning (DL) [30], several studies recently tried to apply convolutional operations on graph-structure data. GCN was first introduced in [25]. It is a scalable approach that is based on an efficient variant of convolutional neural networks and directly operates on graphs. GCN has a strong capability of feature extraction and aggregation to find out the optimal solution in a more aggressive manner. For the classification methods of graph neural networks, many algorithms are based on GCN, and further improved and expanded to form new graph neural networks (GNNs) [52]. They include Graph Attention Networks (GATs) [47], Graph Auto-encoder (GAE) [26], Graph Generative Networks (GGNs) [31], Graph Spatial-temporal Networks (GSTNs) [32], etc. Figure 1 summarizes the relationship between these GNNs. The key difference between GATs and GCNs is that GATs adopt attention mechanisms, which assign larger weights to the more important nodes [47]. GAE is an unsupervised learning framework designed to learn low-dimensional node vectors through an encoder, and then reconstruct graph data through a decoder. For attributed graphs, the graph autoencoder model tends to use GCN as the building block of the encoder [26]. GGNs aim to generate reasonable structures from graph data. GGNs either use GCN as a building block or adopt a different architecture [31]. The key idea of GSTNs is to consider spatial dependency and temporal dependency at the same time. Recently, many approaches apply GCNs to capture the dependency together with some RNN or CNN to model the temporal dependency [32].

Many other approaches by improving GCN have also been proposed in recent years [4,8,18]. Since more layers added to GCN will make the training phase time-consuming. The output features may be over smoothed and nodes from different clusters may become indistinguishable. To solve this problem, a FastGCN is proposed, which improves the original GCN model with efficient small batch training [4]. In [8], an adaptive hierarchical sampling method is proposed to accelerate the training process of the GCN model. The work in [18] develops the algorithm to approximate the GCN model based on the control variable and puts forward an efficient random algorithm based on sampling to training.

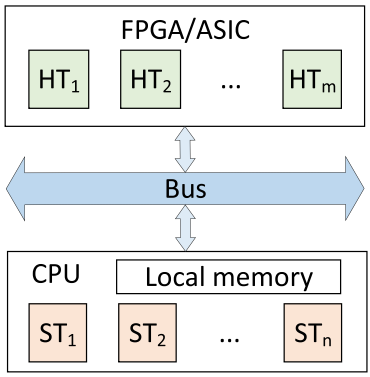


Fig. 2 Target architecture of system

The above studies indicate that the graph convolution of the GCN model is a special form of Laplacian smoothing, which aggregates the features of a node and its neighbors [28]. The smoothing operation makes the features of nodes in the same cluster similar, which is a benefit for classification. This is the key reason why GCN works so well for node classification problems, including the HW/SW partitioning problem. To our knowledge, the partitioning problem can be transformed into the node classification problem. GCN is a simple training method that can effectively process large-scale graph data. Inspired by the above researches, we use the two-layer GCN model as the basis and introduce feature similarity to construct an improved GCN model, and improve the optimization process to achieve efficient HW/SW partitioning. This is a new idea and has high feasibility.

In this paper, the GCN model and an efficient gradient-based optimization method are available to make the model converges rapidly. Experimental results show that GCPS outperforms the state-of-the-art methods.

3 Problem description

The target architecture in our work consists of a single software (processor) and several hardware (ASIC/FPGA) components, as shown in Fig. 2. ST_1, ST_2, …, ST_n represent the software tasks sequentially executed by the single CPU; HT_1, HT_2, …, HT_m represent the dedicated hardware tasks implemented by ASIC/FPGA. The hardware tasks that have no data dependencies can be executed in parallel. All data is transferred over a shared bus. The purpose of HW/SW partitioning is to find an optimal partitioning scheme that satisfies the design constraints.

The problem can be formalized as follows. The applications of an embedded system can be generally modeled as a direct acyclic task graph (DAG) $G(\boldsymbol{T}, \boldsymbol{E})$. $\boldsymbol{T} = \{T_1, T_2, \ldots, T_N\}$ denotes the set of tasks (nodes), and N is the number of nodes. Each node has three attributes: execution time in software t_s, execution time in hardware t_h, and area cost s in hardware. $\boldsymbol{E}$ denotes the set of edges between two tasks, which describes the dependencies between the tasks of the system. For example, an edge $(T_i, T_j) \in \boldsymbol{E}$ represents the dependency between T_i and T_j, indicating that T_i has to be executed before T_j. c_{ij} is the communication time between T_i and T_j. Assume that the communication time is ignored between two adjacent nodes that have the same context (software or hardware). Figure 3 illustrates an example.

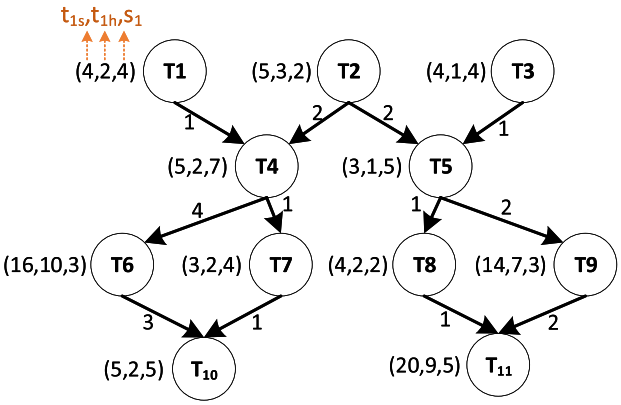


Fig. 3 A task graph example

There are 11 tasks, and the i-th of them has the attributes t_{is}, t_{ih}, and s_i. The edge weight represents the communication time between the two connected tasks.

The objective function of the partitioning problem can be formulated as follows:

$$\begin{aligned} &\min \ T_{SL} \\ &s.t. \sum_{i=1}^{n} s_i y_i \leq C, \ y_i \in \{0, 1\}, i = 1, 2, \ldots, n \end{aligned} \tag{1}$$

As can be seen from Eq. (1), our optimization objective is minimizing the schedule length T_{SL}. T_{SL} can be calculated by the list scheduling algorithm with static priorities (LSSP) [41]. $Y = \{y_1, y_2, \ldots, y_N\}$ represents the partitioning results. If $y_i = 1$ or 0, it shows that the i-th task is assigned to hardware or software. $\sum_{i=1}^{n} s_i y_i$ is the total hardware area of whole labeled tasks. The hardware area of software tasks is 0. C denotes the hardware area constraint. Therefore, the optimization problem can be summarized as minimizing the schedule length under the hardware area constraint.

The inputs to the problem solved in this paper are hardware area constraint, termination condition parameter, and a task graph with given nodes, edges, and attributes. The outputs are partitioning result, total hardware area, and schedule length. The objective of this work is to minimize the schedule length under the given area constraint. To efficiently obtain a high-quality partitioning solution, the GCPS is proposed to solve the partitioning problem for a large-scale system.

4 Proposed approach, GCPS

Our partitioning method is mainly performed in three steps as described in the following subsections.

4.1 Preprocessing of task graph

To divide the system into hardware and software components, the first step is to construct a system task graph based on the system specification. We firstly construct a directed acyclic

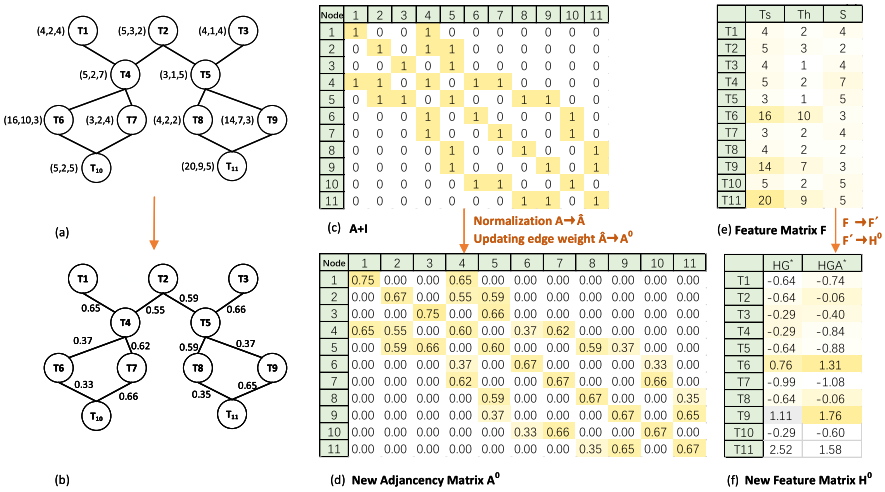


Fig. 4 Construction and preprocessing of task graph

task graph, where the direction information of edges is not used in the convolution process, while in the scheduling process, the direction information should be considered. Figure 4a is a diagram of the undirected graph that can also be represented by an adjacency matrix A. However, the constructed adjacency matrix cannot be directly applied to the GCN model. There are two reasons. One is that for each node, we can aggregate the feature vectors of its neighbors through GCN, but it does not include the information of the node itself. So we introduce the self-connection that is an identity matrix as denoted by I added to the adjacency matrix. Let $\tilde{A} = A + I$. Figure 4c is the matrix form of the task graph after adding the self-connection. If the element $\tilde{a}_{ij}(i \neq j)$ in $\tilde{A}$ is equal to 1, it indicates there is an edge between node i and j. The other reason is that GCN is sensitive to the scale of input data. Feature values of the nodes with multiple edges may be significantly larger than that of the nodes with fewer edges. If A is non-normalized, the feature values will be on a different scale after convolution. Therefore, motivated by [25], a normalization trick is used to prevent numerical instabilities and ensures the convergence of the GCN model. The degree matrix $\tilde{D} = diag(d_1, d_2, \ldots, d_N)$ of $\tilde{A}$ is introduced. $\tilde{D}$ is a diagonal matrix, and each diagonal element $\tilde{d}_i$ is the number of edges connected to the current node ($\tilde{d}_i = \sum_j \tilde{A}_{ij}, i, j = 1, 2, \ldots, N$). We apply the following normalization trick to the adjacency matrix and generate a new transformed adjacency matrix $\hat{A}$:

$$\hat{A} = \tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2} \tag{2}$$

Another important input that should be constructed for GCN is the feature matrix. The attributes of nodes can be combined into a feature matrix $F \in N \times M$ as shown in Fig. 4e, where each column represents an attribute; M is the number of attributes. Considering the features that should be a benefit for partitioning, the hardware gain (HG) and hardware gain per unit area (HGP) are taken into account to construct a new feature matrix H^0, where

$$\begin{aligned} HG &= t_s - t_h \\ HGP &= \frac{t_s - t_h}{s} \end{aligned} \tag{3}$$

The larger HG and HGP values indicate that the task has a higher probability of being assigned in hardware. Firstly, HG and HGP are used as new attributes to transform the F into $F' = [HG, HGP] \in N \times 2$. And then, normalization operation is also utilized to ensure the input features basically in the same magnitude [53]. Let $H^0 = [HG^*, HGP^*]$. The preprocessing of F can be represented as $H^0 = (F' - \bar{M})/std$, where $\bar{M} = F'_{ij} / \sum_j F'_{ij}$ is the mean by the column of F', and $std = \sqrt{\sum_j (F'_{ij} - \bar{M}_j)^2 / N}$ is the variance by the column of F'. After normalization, a new feature matrix H^0 can be generated, and the matrix representation of H^0 can be seen from Fig. 4f.

As mentioned before, the edge weights in the adjacency matrix A are 0 or 1. The correlations between different nodes are actually different, and the communication time cannot intuitively reflect the similarity between different nodes. Therefore, we use cosine similarity instead of the constant (0 or 1) to evaluate the relationship between the two tasks. The cosine similarity of the feature matrix H^0 is calculated to update the edge weight, which measures the cosine of the angle between two vectors of the feature matrix:

$$\cos\theta_{ij} = \frac{f_i \cdot f_j}{\|f_i\| \left\|f_j\right\|} = \frac{\sum_{i=1}^{N} f_i \times \cdot f_j}{\sqrt{\sum_{i=1}^{N} (f_i)^2} \times \sqrt{\sum_{j=1}^{N} (f_j)^2}} \tag{4}$$

We define $H^0 = [f_1, f_2, \ldots, f_N]^T$. f_i represents the feature vector of the i-th node. The similarity can be calculated by Eq. (4). The value of $\cos\theta_{ij}$ is mapped into the range of [0, 1], which shows that the T_i and T_j have a high probability of being allocated to the same category (software or hardware) if there is a large similarity between them. Otherwise, these two tasks will be assigned to different categories. Finally, we update the $\hat{A}$ through Eq. (5) to generate a new adjacency matrix $A^0 = \{a^0_{ij}\}, i, j = 1, 2, \ldots, N$.

$$\begin{cases} \hat{a}_{ij} = \cos\theta_{ij}, (\hat{a}_{ij} \neq 0 \; and \; i \neq j) \\ A^0 = \hat{A} \end{cases} \tag{5}$$

The updated edge weights show the similarity between the connected nodes, which can be seen in Fig. 4b. Figure 4d is the matrix form of A^0. The values marked with yellow color except for the diagonal elements shows the new edge weights.

4.2 HW/SW partitioning based on GCN

HW/SW partitioning problem is transformed into a node classification problem, which can be efficiently processed by GCN. The process of node classification by using GCN can be simply interpreted as matrix transformation. The model of GCN consists of three types of layers that are input layer, hidden layers (graph convolution layer), and output layer. In each hidden layer, node representations are updated in three stages: feature propagation, linear transformation, and nonlinear activation. Based on the preprocessing of Sect. 4.1, the inputs to GCN include the new adjacency matrix A^0, feature matrix H^0, and the parameter k in termination criterion, which means the algorithm will terminate if the best result achieved so far is maintained after k successive iterations. The output of GCN is the probabilities of hardware implementation, which can be transformed into hardware/software labels. The model of GCN can be formulated as follows.

$$\begin{cases} H^{l+1} = f(H^0, A^0) = ReLU(A^l H^l W^l) \quad l = 0, 1 \\ X = Softmax(F_C(H^2)) \end{cases} \tag{6}$$

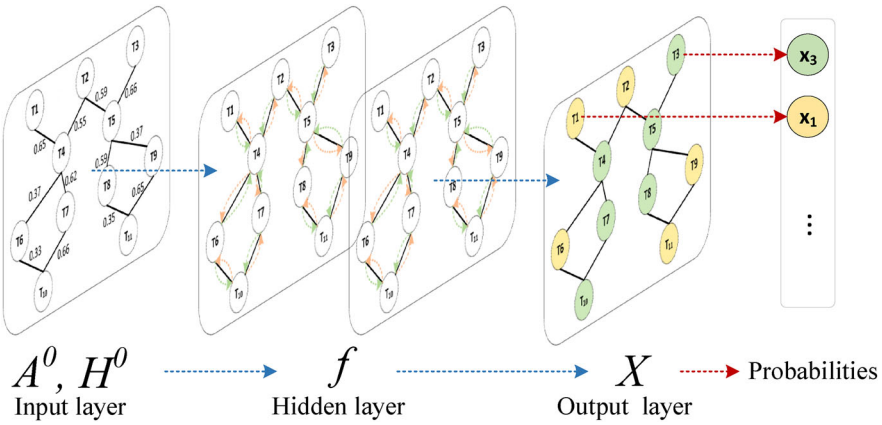
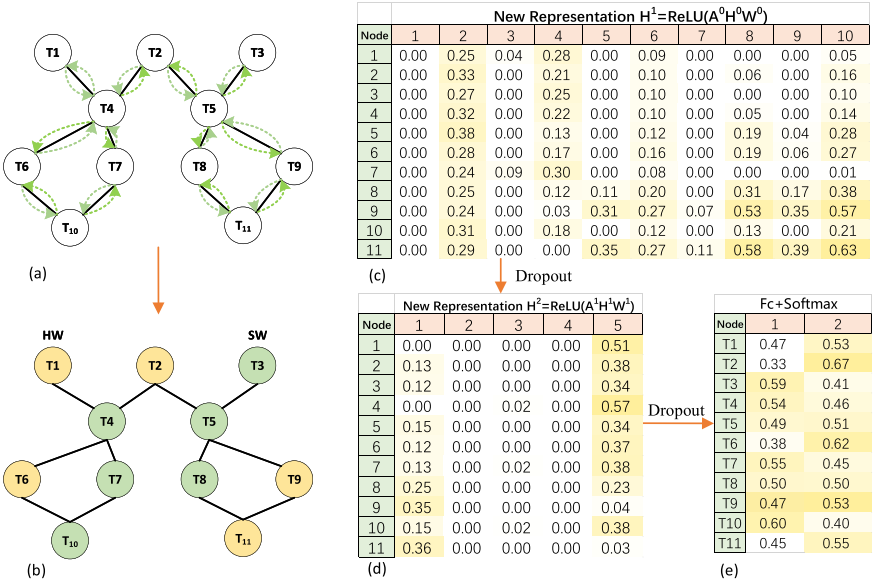


Fig. 5 The 2-hidden layer GCN model

where W^l is the weight matrix for the l-th layer of GCN, and the initial value of W^l is randomly generated. H^l is the feature representation of nodes on the l-th GCN layer. For the propagation rule, after introducing the weight matrix to aggregate the features of nodes, the rectified linear unit (ReLU) activation function is adopted to convert features into data on another dimension. For many types of neural networks, the activation function of a node defines the output of that node given an input or set of inputs. It is introduced to increase the nonlinearity of the neural network model and approximate any nonlinear function. If there is no activation function, the output of the model will be a linear combination of the inputs no matter how many layers the neural network has. Thus, the ReLU is easier to be applied to a nonlinear model and often achieves better performance [37]. F_C is the linear layer to map and reshape the output into a lower dimension. There is a normalization method $Softmax$ that is often used for the output layer of the neural network. It can map each non-normalized output to a probability over predicted classes. Finally, the output of GCN model is $X = [x_1, x_2, \ldots, x_N]^T$ (hardware implementation probabilities).

To the best of our knowledge, the previous studies showed that the best classification results are obtained with a 2 or 3 layers model [25]. As the number of hidden layers increases, training will become difficult and time-consuming. Furthermore, the performance of GCN will not improve as the number of layers increases. This effect is known as over-smoothing. It will be another issue affecting performance as the quantity of parameters increases. Therefore, the 2-hidden layer GCN model is constructed in our approach, as shown in Fig. 5.

The detailed graph convolution process can be illustrated in Fig. 6. As can be seen from Fig. 6a, the green arrows pointing to the current node represent that the features of neighboring nodes are aggregated to the current node. A new node representation H^1 can be generated after the first convolution, and Fig. 6c shows the matrix form of H^1. Due to the introduction of similarity and the updating mechanism of the adjacency matrix, the nodes with large similarity can aggregate more feature information of neighboring nodes, otherwise less information will be aggregated. Moreover, for the second-layer convolution operation, the feature information of 2-level neighboring nodes can be aggregated to the current node in the same way. The aggregation result is shown in Fig. 6d. In addition, after each convolution layer, $Dropout$ is utilized to randomly deactivate some hidden units to avoid over-smoothing in the training process. The hidden units are in the hidden layer. It has no direct relationship with the size of the adjacency matrix. A hidden unit corresponds to the output of a single filter at a single



New Representation H^1=ReLU($A^0H^0W^0$)

Node	1	2	3	4	5	6	7	8	9	10
1	0.00	0.25	0.04	0.28	0.00	0.09	0.00	0.00	0.00	0.05
2	0.00	0.33	0.00	0.21	0.00	0.10	0.00	0.06	0.00	0.16
3	0.00	0.27	0.00	0.25	0.00	0.10	0.00	0.00	0.00	0.10
4	0.00	0.32	0.00	0.22	0.00	0.10	0.00	0.05	0.00	0.14
5	0.00	0.38	0.00	0.13	0.00	0.12	0.00	0.19	0.04	0.28
6	0.00	0.28	0.00	0.17	0.00	0.16	0.00	0.19	0.06	0.27
7	0.00	0.24	0.09	0.30	0.00	0.08	0.00	0.00	0.00	0.01
8	0.00	0.25	0.00	0.12	0.11	0.20	0.00	0.31	0.17	0.38
9	0.00	0.24	0.00	0.03	0.31	0.27	0.07	0.53	0.35	0.57
10	0.00	0.31	0.00	0.18	0.00	0.12	0.00	0.13	0.00	0.21
11	0.00	0.29	0.00	0.00	0.35	0.27	0.11	0.58	0.39	0.63

New Representation H^2=ReLU($A^1H^1W^1$)

Node	1	2	3	4	5
1	0.00	0.00	0.00	0.00	0.51
2	0.13	0.00	0.00	0.00	0.38
3	0.12	0.00	0.00	0.00	0.34
4	0.00	0.00	0.02	0.00	0.57
5	0.15	0.00	0.00	0.00	0.34
6	0.12	0.00	0.00	0.00	0.37
7	0.13	0.00	0.02	0.00	0.38
8	0.25	0.00	0.00	0.00	0.23
9	0.35	0.00	0.00	0.00	0.04
10	0.15	0.00	0.02	0.00	0.38
11	0.36	0.00	0.00	0.00	0.03

Fc+Softmax

Node	1	2
T1	0.47	0.53
T2	0.33	0.67
T3	0.59	0.41
T4	0.54	0.46
T5	0.49	0.51
T6	0.38	0.62
T7	0.55	0.45
T8	0.50	0.50
T9	0.47	0.53
T10	0.60	0.40
T11	0.45	0.55

Fig. 6 The process of graph convolution

specific x/y offset in the input volume. The number of hidden units is a hyperparameter. If the number of hidden units is too small, the model will be inaccurate. If there are too many hidden units, overfitting will occur. Therefore, we need to choose the appropriate hidden unit. In this work, the number of hidden units for the first hidden layer and the second hidden layer are set to be 10 and 5 ($h_1 = 10$, $h_2 = 5$), respectively. These indicate the dimension of the new representation matrices H^1 and H^2. And then, hardware implementation probabilities can be generated through the operation of linear transformation (F_C) and normalization ($Softmax$). The output result of GCN can be illustrated in Fig. 6e, and the second column shows the probabilities of hardware implementation.

To transform the probability values into 0 or 1 (software:0, hardware:1). It is necessary to set a cut-line. Since the label of each node is related to the probability of hardware implementation but also limited by area constraints. In this case, the area constraint is 18. Therefore, we use the greedy selection strategy to generate the initial hardware/software labels. It first sorts the hardware implementation probabilities of all tasks in descending order. Then the nodes with the highest to lowest hardware implementation probability are sequentially labeled as hardware. Finally, it does not stop until the area constraint is not satisfied, and tasks that exceed the area constraint and other remaining tasks are labeled as software. Eventually, we can obtain the initial labels Y. Figure 6b shows that the nodes colored in yellow and green are assigned to hardware and software, respectively.

GCN is a graph neural network approach based on gradient descent algorithm, which is for finding a local minimum of an objective function. The optimization of the GCN model can be regarded as a convex optimization problem, so the objective function should be continuous and differentiable. For our objective function which takes the scheduling into account, the partitioning problem will become a non-convex optimization problem, and the objective function will be non-differentiable. Therefore, we introduce the auxiliary objective function T_{cost} to avoid this issue so that the optimization approach based on the gradient descent can

be applied.

$$T_{cost} = \sum_{i=1}^{N} (t_{ih} x_i + t_{is}(1 - x_i)) \tag{7}$$

In Eq. (7), the T_{cost} is a function that approximates the sum of execution time of all partitioned tasks. x_i is the probability of hardware implementation for the i-th task. The optimization problem is transformed to minimize T_{cost} in terms of GCN. We use the gradient descent method to update weight parameters of GCN model and minimize T_{cost}. In addition, scheduling is also an important part in GCPS. The outcome of scheduling is used as feedback to the training process. The details of scheduling are described in the next subsection.

4.3 Scheduling

Our proposed method performs a scheduling operation after a training iteration, and the result of the scheduling can be fed back to the partitioning process, which is an indirect combination of the scheduling and the partitioning to make the result more precise. Thus, the final objective function T_{SL} can be also optimized. As the static scheduling is more suitable for systems with known and fixed task graphs, for our target architecture, a list scheduling with the static priorities (LSSP) algorithm is adopted [51]. The main steps of LSSP list as follows:

- Calculate the static priority of each task.
- Allocate the task to the ready list based on the priority.
- Remove the finished tasks and update the ready list.
- Output the schedule length.

The calculation of priority ($Pri(\cdot)$) is based on the topological structure of the task graph. The order of calculating the priority is from the bottom to the top, and the direction of task graph is also taken into account. T_b and T_e respectively represent the beginning task and the exit task. The priority of each task is the length of the longest path from the current task T_i to the exit task T_e. Besides, the counting of path length includes both execution and communication times. For instance, the red dotted line in Fig. 7 shows the longest path, and the schedule length of this example is 32.

LSSP is summarized in Algorithm 1. After the task graph is constructed and the calculation of the priorities is completed, it is necessary to create r_{list}, f_{list}, and u_{list} (Step 3). r_{list} is used to store the ready tasks whose connected previous tasks all have been allocated. The part of Steps 4 to 8 represent the process of allocating the tasks to r_{list}. In r_{list}, tasks will be sorted in descending order of static priority, and the task with highest static priority is firstly taken out for allocation. f_{list} is utilized to store tasks that have been allocated. u_{list} is temporarily created and used to store tasks that have not been allocated. From step 10 to step 18, the start/finish time of a task (t_{start}/t_{finish}) and the start/finish time of an edge (tc_{start}/tc_{finish}) are set. For a task T_i or an edge c_{ij}, it is allocated based on the t_{finish} of the previous tasks (P(T_i)) and whether the task is implemented in hardware or software. The hardware task can be executed in parallel, while the software tasks need to consider whether the processor is idle or not. For steps 19 to 22, after completing the task allocation and data transmission, the task will be removed from the r_{list} and added to the f_{list}. The state of the tasks in u_{list} should be updated to check whether there are new ready tasks that can be added to the r_{list}. Then the process of task allocation and transmission is repeated until all tasks are allocated.

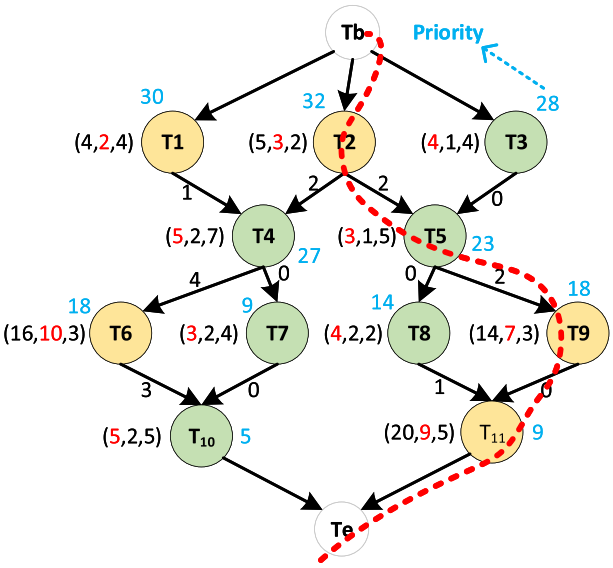


Fig. 7 An example of priority calculation

Table 1 The scheduling process of the task graph

Task	$Pri(\cdot)$	t_{exe}	t_{start}	t_{finish}	*Bus time*
T1(HW)	30	2	0	2	T1 → T4:2–3
T2(HW)	32	3	0	3	T2 → T4:3–5, T2 → T5:5–7
T3(SW)	28	4	0	4	**T3 → T5:7–7**
T4(SW)	27	5	5	10	T4 → T6:10–14, **T4 → T7:16–16**
T5(SW)	23	3	10	13	T5 → T9:14–16, **T5 → T8:16–16**
T6(HW)	18	10	14	24	T6 → T10:24–27
T7(SW)	9	3	20	23	**T7 → T10:23–23**
T8(SW)	14	4	16	20	T8 → T11:20–21
T9(HW)	18	7	16	23	**T9 → T11:23–23**
T10(SW)	5	5	27	32	–
T11(HW)	9	9	23	32	–

The scheduling process of the example given in Fig. 7 is illustrated in Table 1. The first column of Table 1 represents the name of labeled tasks. HW or SW shows the task is implemented in hardware or software. The second column is the priority of each task. t_{exe} is the execution time of a task. If the task is implemented in hardware, t_{exe} will be the value of hardware execution time, otherwise the software execution time. The t_{start} and the t_{finish} respectively means the start time and finish time of a task. In *Bus time* column, "$T1 \rightarrow T4$:2–3" indicates that the data currently transmitted on the bus is from task 1 to task 4, and the time on the bus is 2 to 3, i.e. the start and the finish time of the edge $c_{14}(tc_{start} = 2, tc_{finish} = 3)$. The bold text indicates that the communication time between two tasks assigned to the same context (software or hardware) is assumed to be zero. Eventually, we can obtain the result of scheduling, as shown in Fig. 8.

Algorithm 1 LSSP

Input: Y.
Output: T_{SL}
Construct the task graph for scheduling.
Calculate the static priority $Pri(\cdot)$ of each task.
Create r_{list}, f_{list}, u_{list}.
for i=1:N **do**
  if T_i is ready **then**
    Add T_i to the r_{list}.
  end if
end for
while u_{list} != NULL **do**
  for $T_j \in r_{list}$ **do**
    for c_{ij} in $\{c_{ij} | T_i \in P(T_j)\}$ **do**
      if $c_{ij} \neq 0$ **then**
        Set tc_{start} and tc_{finish} for c_{ij}.
      end if
    end for
    Set t_{start} and t_{finish} for T_j.
    Add T_j to f_{list}.
  end for
  for $T_k \in f_{list}$ **do**
    Update the state of task in u_{list}.
    Update the r_{list}.
  end for
  if u_{list} == NULL **then**
    break.
  end if
end while
Return T_{SL}

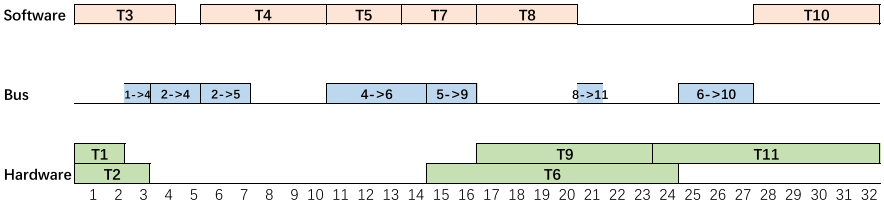


Fig. 8 The result of scheduling

The target hardware architecture for the example consists of a single processor and several ASIC components. All data is transferred over a shared bus. The processor and the hardware components have their own local memory. The data transfer between software tasks or hardware tasks without going through the bus. The Bus axis in Fig. 8 shows the transfer process between the software task and the hardware task. "$1 \rightarrow 4$" means that the data is transferred from task 1 to task 4, and the length of the blue box indicates the transfer time is 1. The result shows that the overall execution time of the system can be effectively reduced after scheduling.

In our proposed method, some tricks are also proposed to accelerate the model training and enhance accuracy:

– In order to get a better initial solution, the pre-training strategy is used. We define t is the iterations of pre-training, which can be tuned according to the different applications.

Meanwhile, to further speed up the training process without influence on the precision, we skip s iterations of training so that the number of scheduling can be dramatically reduced.

- To reduce the time of scheduling and find the cut line of hardware and software, a quick search method based on a greedy algorithm is proposed. Let $skip\_num = C/S_{mean}$, where S_{mean} is the average hardware area of whole tasks, which represents that $skip\_num$ tasks are firstly assigned to hardware based on the probability under the condition of area constraint. Moreover, if the total area does not meet the area constraint, the number of hardware tasks will be dynamically adjusted.
- The activation function of ReLU can make it easy to train the GCN model, while it might lose some original information. Inspired by the work of [15], the feature of HGP (defined as F_1) is aggregated into the output to preserve the original information and further enhance the performance of the model. The operation can be reformulated as $X' = (1-\sigma)X + \sigma F_1$, where σ is the proportion of the aggregated information.

In this work, k is a dynamic variable related to the number of nodes N. We have done extensive experiments to find the best values for these parameters. Eventually, $k = N/\alpha$ if $n \leq 200$, otherwise $k = N/(2\alpha)$, where $\alpha = 5$. The proportion parameter σ is set to be 0.3.

The flow chart of GCPS is summarized in Fig. 9. GCPS can be summarized into three steps:

- The first step is to construct a task graph and preprocess it to meet the input constraints of the model. The inputs of the first step are adjacent matrix A, feature matrix F, hardware area constraint C, and termination condition parameter k. The output of the first step is the normalized adjacent matrix A^0 and feature matrix H^0.
- In the second step, the partitioning model is constructed. The pre-training and the training are performed. As can be seen from Fig. 9, the 'training' box performs the same function as the pre-training process. The process of pretraining is shown in Algorithm 2. The input of the second step is the normalized adjacent matrix A^0 and feature matrix H^0. The output is the probability of hardware implementation X. The operations in the training process are the same as the pre-training.
- In the third step, when we achieve the probability X, after the above three optimization tricks, we can get the initial partitioning result. Then through LSSP scheduling, we can obtain the scheduling time, and the results are fed back to the model until convergence. The process of GCPS can be also summarized in Algorithm 3. The input is the initial partitioning result, and the output is the labels Y, schedule length T_{SL}, and total hardware area $Area$.

In Fig. 9, we define '1 epoch' that is a term used in machine learning and indicates one complete pass through the training data. The maximum number of the epoch is set to be 800. The minimum schedule length min_T_{SL} will remain, and its corresponding epoch value opt_epoch is recorded. When the best result obtained so far is maintained after k successive epochs, the algorithm will terminate.

5 Experiment and analysis

In this section, we evaluate the effectiveness and quality of the proposed GCPS. Since there are no datasets for HW/SW partitioning of embedded system in the real world that can be directly used, we adopt a task graph generation tool named TGFF (Windows Version 3.1) to generate DAGs randomly [9]. TGFF is a highly configurable pseudo-random task graph generator

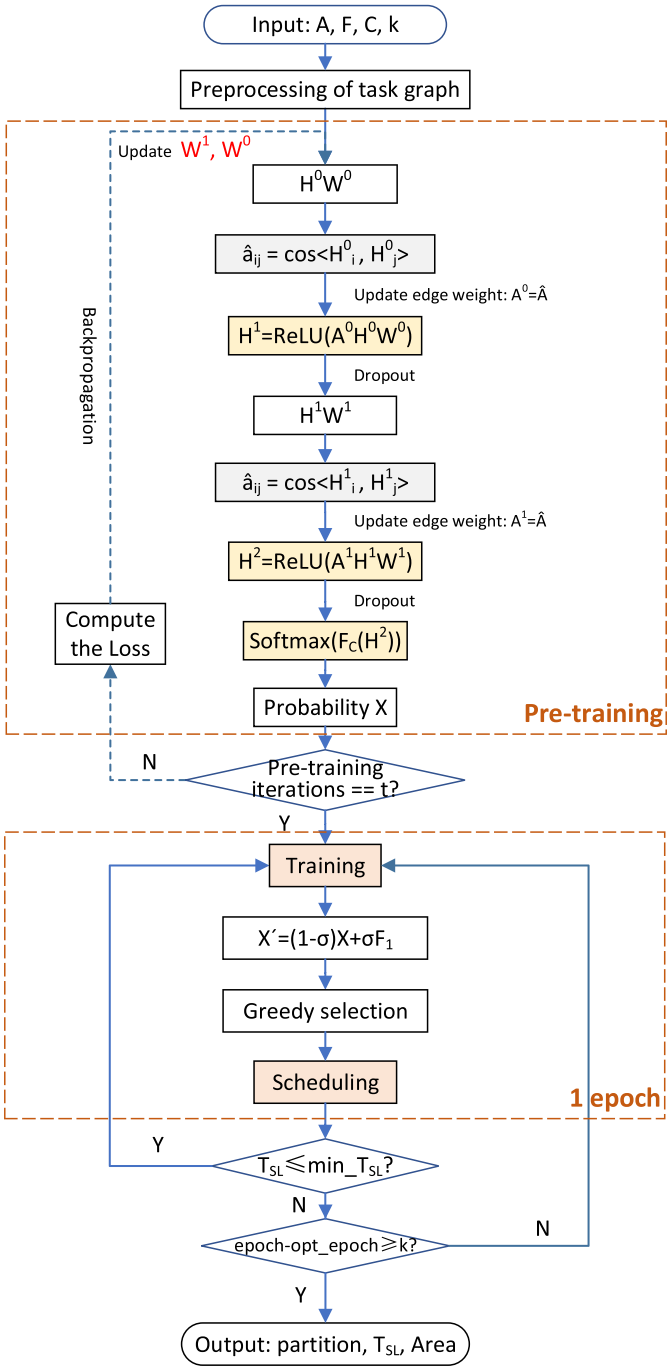


Fig. 9 The flow chart of GCPS

Algorithm 2 Pre-training

Input: A^0, H^0
Output: X
1: **for** i = 1 : t **do**
2: **for** l=1:2 **do**
3: Calculate $H^{l-1}W^{l-1}$ and update A^{l-1} to A^l.
4: Calculate $H^l = ReLU(A^{l-1}H^{l-1}W^{l-1})$ and dropout.
5: **end for**
6: Calculate $X = Softmax(F_c(H^2))$.
7: **end for**
8: Return X

Algorithm 3 GCPS

Input: A^0, H^0, C, k
Output: Y, T_{SL}, $Area$
1: $X = Pretraining(A, F, C)$.
2: **while** $epoch - opt\_epoch < k$ **do**
3: **if** $T_{SL} \leq min\_T_{SL}$ **then**
4: $X = Training(A, X, C)$. (Training ⟵ Pretraining)
5: $X' = (1 - \sigma)X + \sigma F_1$
6: Greedy selection and generate labels Y.
7: $T_{SL} = LSSP(Y)$
8: **end if**
9: **end while**
10: Calculate $Area$.
11: Return Y, T_{SL}, $Area$.

to facilitate standardized random benchmarks for partitioning and scheduling research. We implemented all algorithms in Python and ran on CPU I7-8700k with 3.2G Hz frequency and 16G memory.

5.1 Comparators and metrics

In the experiments, we compare the proposed GCPS algorithm with five metaheuristic methods. The first is the traditional GA algorithm [40]. The second one is a new algorithm base on GA named HGA [29] which introduced the hardware orientation and three dynamic probabilities. The third one is the traditional SA method [6]. The fourth one is the hybrid partitioning method named GSA which put forward an initial solution generated by the hardware gain per unit [22]. And the last one is the KL algorithm which can quickly generate solution for the partitioning problem [35]. For a fair comparison, the five partitioning metaheuristics are respectively combined with the LSSP scheduling algorithm to evaluate the performance. We use the acceleration ratio (AR) and the improvement as the metrics for comparison. AR is formulated as follows:

$$AR = \frac{T_{max}}{T_{SL}} \tag{8}$$

where T_{max} is the sum of execution times of all tasks implemented in software, and T_{SL} is the schedule length. Let A and B denote two different methods, the AR improvement of A over B is defined as follows:

$$Improvement = \left(\frac{A}{B} - 1\right) \times 100\% \tag{9}$$

Table 2 Parameters defined in DAGs

Parameters of DAG	Values
Number of nodes	20, 50, 100, 200, 500, 1000
Area percentage (AP)	1/5, 1/3, 2/5, 1/2, 3/5, 2/3, 4/5
Range of software execution time	$2 * 10^3$–$1.2 * 10^4$ (ns)
Range of hardware execution time	$2 * 10^2$–$1.2 * 10^3$ (ns)
Range of hardware area	100–400 (unit)
Range of communication time (CT1)	2–100 (ns)
Range of communication time (CT2)	100–600 (ns)
Range of communication time (CT3)	800–2000 (ns)

Table 3 Hyperparameter settings

Methods	Hyperparameters
GA	$\alpha = 0.75, \beta = 0.25, P_c = 0.9, P_m = 0.005$
HGA	$\alpha = 0.75, \beta = 0.25, p_{m_1} = p_{c_1} = 0.15, p_{m_2} = p_{c_2} = 0.75$
SA	$T = 1000, L = 300, M = 500, r = 0.999$
GSA	$T = 1000, L = 300, M = 500, r = 0.999$
KL	$N = 1.1$
GCPS	$\alpha = 5, l_r = 0.001, h_1 = 10, h_2 = 5, s = 5, \sigma = 0.3$

5.2 Experiment settings

To ensure the diversity of datasets, six different DAGs with 20, 50, 100, 200, 500, and 1000 nodes are respectively generated by TGFF. The settings of attributes on tasks in each DAG are described in Table 2. As seen from Table 2, we have DAGs with six different number of nodes. Area percentage (AP) defines the area constraint, which accounts for p percentage of the total hardware area of all tasks. The hardware area represents the resource occupancy of the silicon area for a task when it is assigned to hardware. The communication time is varied for different applications. For example, there are three kinds of communication cost (CT1, CT2, and CT3) in our experiments to explore the impact of different communication time on the results, and each of them in its interval follows a uniform random distribution. CT1 is the communication cost for the tasks with little communication, and CT3 is that for the communication-intensive tasks. CT2 is set between CT1 and CT3.

The five existing metaheuristics respectively use the same experimental parameters as given in [6,22,29,35,40]. The parameter settings are summarized in Table 3. For GCPS, the lr is the learning rate of GCPS model, and h is the number of hidden units. The hyperparameters of GCPS in this work can be tuned to support the different scale of systems.

In our experiments, we applied GCPS and five existing metaheuristics to three HW/SW partitioning cases as follows.

5.2.1 Case 1

To demonstrate the effectiveness of GCPS with the different number of nodes, six kinds of task graphs (datasets) with the characteristics of AP = 1/3 and CT2 are used. For each

Table 4 The result of acceleration ratio with different nodes under the communication time of CT2

Settings	GA	HGA	SA	GSA	KL	GCPS
20N	1.441	1.580	1.444	1.612	1.459	**1.635**
50N	1.591	1.623	1.525	1.641	1.569	**1.723**
100N	1.590	1.565	1.523	1.646	1.568	**1.756**
200N	1.575	1.573	1.541	1.635	1.580	**1.768**
500N	1.552	1.546	1.533	1.641	1.601	**1.796**
1000N	1.514	1.563	1.548	1.668	1.575	**1.835**

dataset, there are ten graphs with the same number of nodes and different structure, which are randomly generated by TGFF.

5.2.2 Case 2

To evaluate the performance of GCPS on the different communication intensities, we use the dataset with 1000 nodes and choose the graphs with the characteristics of CT1, CT2, and CT3. For each kind of communication time, there are also ten different graphs, and we test on the condition of AP = 1/3.

5.2.3 Case 3

To further demonstrate the quality of GCPS under different hardware area constraints, the task graphs with the property of 1000 nodes and CT2, AP = 1/3, 2/3, and 4/5 are utilized. For each kind of AP, there are also ten different graphs.

5.3 Result and analysis

For case 1, there are ten different task graphs for each kind of settings, and the final result of AR is achieved by averaging the results of ten graphs. Table 4 shows the AR with the settings of CT2 and AP = 1/3. The bold value in each row in Table 4 indicates that the AR result of this method is the best under the current settings. In terms of the acceleration ratio of these six methods, our proposed GCPS outperform other approaches, especially for the large-scale graphs. Figure 10 presents the acceleration ratio improvement of GCPS over the best results of other five approaches, i.e. (GCPS/optimal(GA, HGA, SA, GSA, KL) $-$ 1) $\times$ 100%. It can be concluded that the improvements of acceleration ratio increase with the number of nodes. For large-scale graphs, e.g., the improvement of acceleration ratio is over 10% for the graph with 1000 nodes. The reason is that the solution space will expand as the number of nodes increases, and GCPS benefits from the unique optimization mechanism, which is different from other metaheuristics. It can rapidly extract and search for better results.

For case 2, the task graphs with 1000 nodes and different communication times (CT1, CT2, and CT3) are chosen to study the impact of different communication times on performance. Table 5 and Fig. 11 shows that GCPS outperforms other methods, and there is not a big difference of acceleration ratio improvement among three different communication time scenarios. It demonstrates that our approach can not only apply to the systems with little communication, but also be suitable for communication-intensive systems.

For case 3, Table 6 depicts the acceleration ratio improvement of GCPS over other metaheuristics for the graph with different area percentages (AP $=$ 1/3, AP $=$ 2/3, and

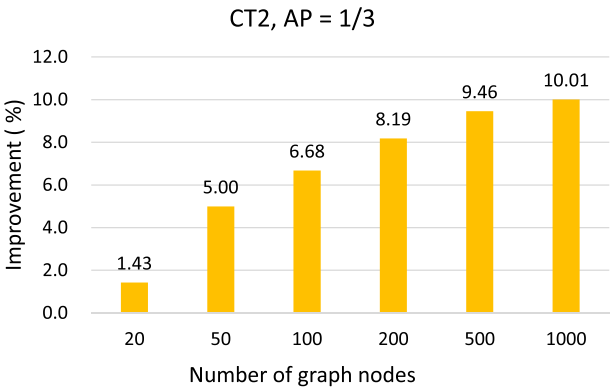


Fig. 10 The experiment results with different number of nodes

Table 5 The acceleration ratio improvement (%) of GCPS over the other five metaheuristics under different communication costs

Settings	GA	HGA	SA	GSA	KL
CT1	20.938	17.297	18.282	10.902	18.053
CT2	20.935	17.455	18.135	10.066	16.709
CT3	20.935	17.530	17.983	10.000	16.192

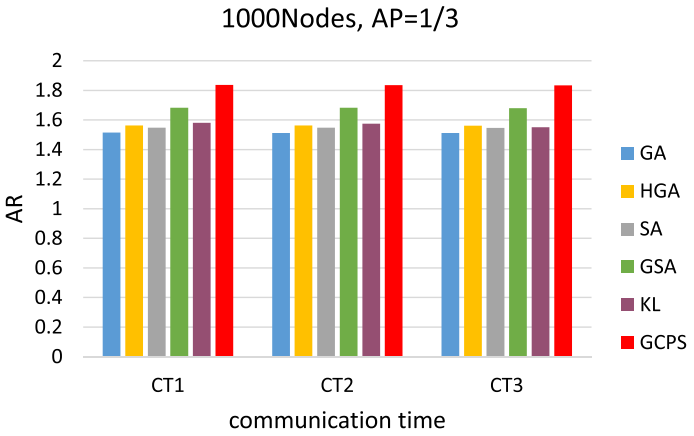


Fig. 11 The experiment results with different communication costs

AP $= 4/5$), and we can see that our proposed method has an improvement over 20% for large scale system when the area percentage is 4/5. There are two reasons. The first is that we focus on the HW/SW partitioning problem for large-scale systems. If the AP is larger, the search space will be larger, and the optimal solution will be more difficult. Thus, the other methods cannot obtain better solutions efficiently under certain constraints. Second, because other methods' initial and termination conditions are set differently, and they are not aimed at such large-scale task graphs, their search capabilities need to be improved. It demonstrates that GCPS has a better quality for large systems, especially the system with loose hardware area constraints. Moreover, although the improvement of acceleration ratio

Table 6 The acceleration ratio improvement (%) of GCPS over the other five metaheuristics under different area percentage values

Settings	GA	HGA	SA	GSA	KL
AP = 1/3	20.935	17.455	18.135	10.066	16.709
AP = 2/3	36.213	31.258	32.065	21.549	20.541
AP = 4/5	47.221	46.802	46.892	30.600	24.419

Table 7 The result of acceleration ratio with the same timestamp under the communication time of CT2

Timestamp(s)	GA	HGA	SA	GSA	KL	GCPS
0.005	1.4804	NA	1.4804	1.6779	1.1031	1.7835
0.25	1.4804	1.5469	1.4821	1.6784	1.1145	1.7839
0.5	1.4804	1.5469	1.4838	1.6789	1.1459	1.7843
0.75	1.4804	1.5469	1.4875	1.6798	1.1773	1.7857
1	1.4804	1.5469	1.4912	1.6807	1.2087	1.7871
1.25	1.4804	1.5469	1.4949	1.6826	1.2401	1.7885
1.5	1.4804	1.5469	1.4986	1.6857	1.2715	1.7899
1.75	1.4804	1.5469	1.5023	1.6912	1.3029	1.7913
2	1.4804	1.5469	1.5060	1.6967	1.3343	1.7927
3	1.4804	1.5469	1.5097	1.7022	1.3657	1.8038
4	1.4804	1.5469	1.5134	1.7077	1.3971	1.8140
8	1.4804	1.5469	1.5171	1.7132	1.4285	1.8342
30	1.4804	1.5469	1.5208	1.7187	1.4599	1.8343
60	1.4804	1.5469	1.5245	1.7242	1.4913	1.8343
80	1.4804	1.5469	1.5282	1.7245	1.5227	1.8343
100	1.5208	1.5469	1.5319	1.7248	1.5541	1.8343
200	1.5260	1.5469	1.5356	1.7258	1.5855	1.8343
300	1.5260	1.5489	1.5393	1.7274	1.6169	1.8343
500	1.5268	1.5546	1.5458	1.7301	1.6219	1.8343

is not so big when AP is 1/3, it still shows that our approach can achieve a better solution for the strict area constraint.

For large-scale systems (1000 Nodes), in order to compare the efficiency of different partitioning methods to search for solutions, we set different timestamps under the communication time of CT2 and sample the AR results as shown in Table 7, where the first timestamp is set as the time required to get the first solution. For these experiments, the termination condition is not applied for the sake of fairness. As seen from Table 7, we can have the conclusion that GCPS can quickly get a better solution via graph convolution. For GA, since only the initial solution (individual) satisfies the area constraint and all other individuals are randomly generated, it could take a long time to search for a solution that satisfies the constraint and is better than the initial solution. In the case of HGA, since the hardware orientation of each individual needs to be calculated and the differences between individuals have to satisfy the Hamming distance which is greater than 4, it is time-consuming for the process of generating the initial population. On the other hand, both HGA and GSA generate initial solutions in a specific way, but which are not necessarily better for large-scale systems; that is, the initial solutions of HGA and GSA may not be as good as the random initial solutions.

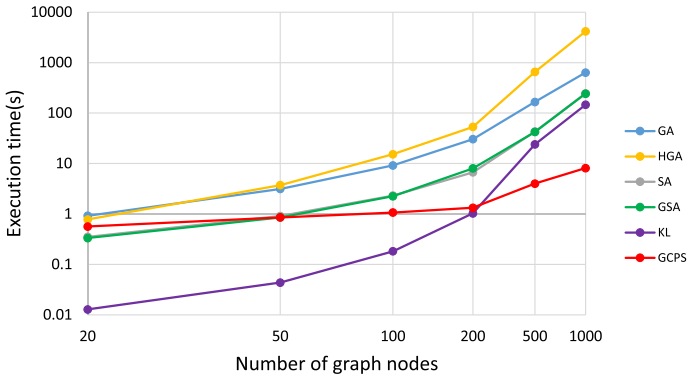


Fig. 12 Execution time with GA, HGA, SA, GSA, KL, GCPS

Finally, we compare the partitioning times of different approaches for the different number of nodes. We define the partitioning time as the time from the beginning of the approach to the convergence of the approach. Figure 12 illustrates that the partitioning times with KL are shorter than those with the other five methods for the graphs with up to 200 nodes. When the number of nodes is less than 200, the efficiency of the KL to find a solution that meets the termination condition is higher than that of GCPS. The reason is that KL algorithm is one of the most efficient metaheuristic algorithms for solving combinatorial optimization problems. It is fast as a greedy algorithm, but it can escape from some local optima. It aims to partition a graph into two parts of equal size with a minimal number of cutting edges. In [35], the KL starts from an arbitrary partition, and make single node moves, rather than node swaps. It can improve the efficiency especially for the graph with a smaller number of nodes. The GSA generates a better initial solution by calculating the area speedup ratio when the number of nodes is 20, making it possible to find the optimal solution with a small number of iterations, which will also take less time than our method. When the number of nodes more than 200, GCPS outperforms the other algorithms. Since the search space become larger with the increasing number of nodes, the metaheuristic methods take more time to search for the optimal solution, and GCPS can learn a better node representation faster even with a large number of nodes through convolution operations (matrix operations). As seen from Fig. 12, GCPS is on average six times faster than KL for 500 nodes graphs, and 18 times faster for graphs with 1000 nodes.

Therefore, the above results and analysis indicate that our proposed approach GCPS has high efficiency and outperforms other metaheuristic methods.

5.4 A real-life example

The conclusions presented above are based on the experiments with random graphs, as discussed in Sect. 5.3. To validate the proposed approach, we experimented on real-life models: the digital signature system based on the ECDSA algorithm [39]. The process of generating the signature and verify the signature can be seen from Fig. 13. Besides, the hash algorithm in ECDSA we used is the SHA-256 algorithm, which is more difficult to crack and has higher security than SHA-1 [2].

The system architecture adopted in this example includes the hardware layer, the embedded software layer, and the top layer. The hardware layer implements hardware tasks, the

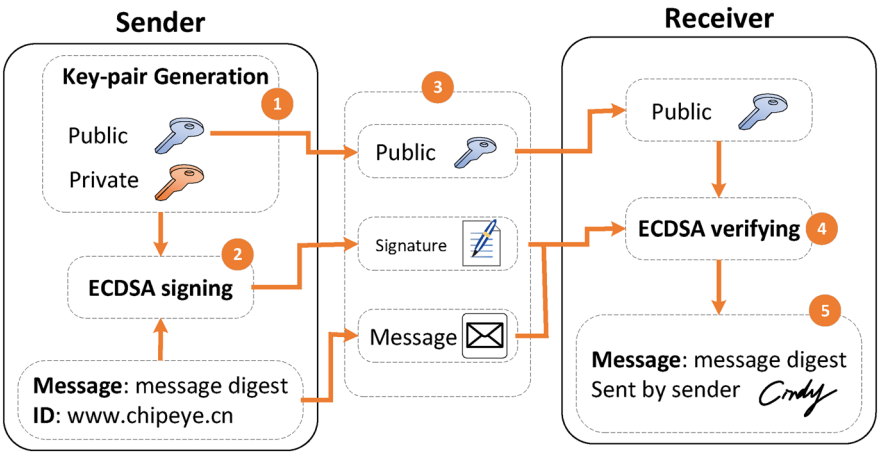


Fig. 13 The digital signature process

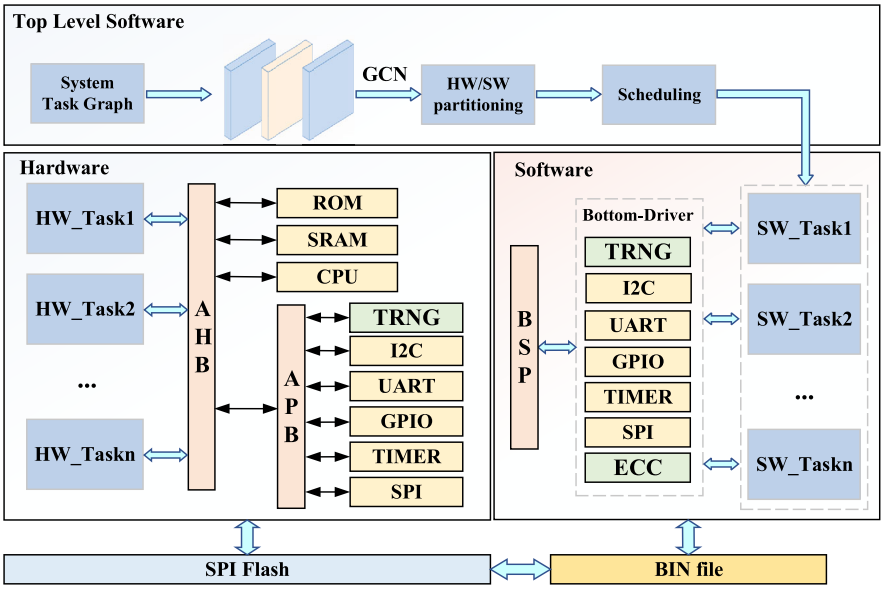


Fig. 14 The system structure diagram

embedded software layer executes software tasks, and the top layer performs the HW/SW partitioning and scheduling for these tasks. The block diagram of the system structure is shown in Fig. 14.

This design can be realized into an SoC (System on Chip). The advanced high-performance bus (AHB) and an advanced peripheral bus (APB) interface protocols are used for data transmission, and the embedded CPU, read-only memory (ROM), static random access memory (SRAM), true random number generator (TRNG), General interface (I2C, SPI, UART, GPIO), Timer are integrated into this SoC to support the ECDSA application.

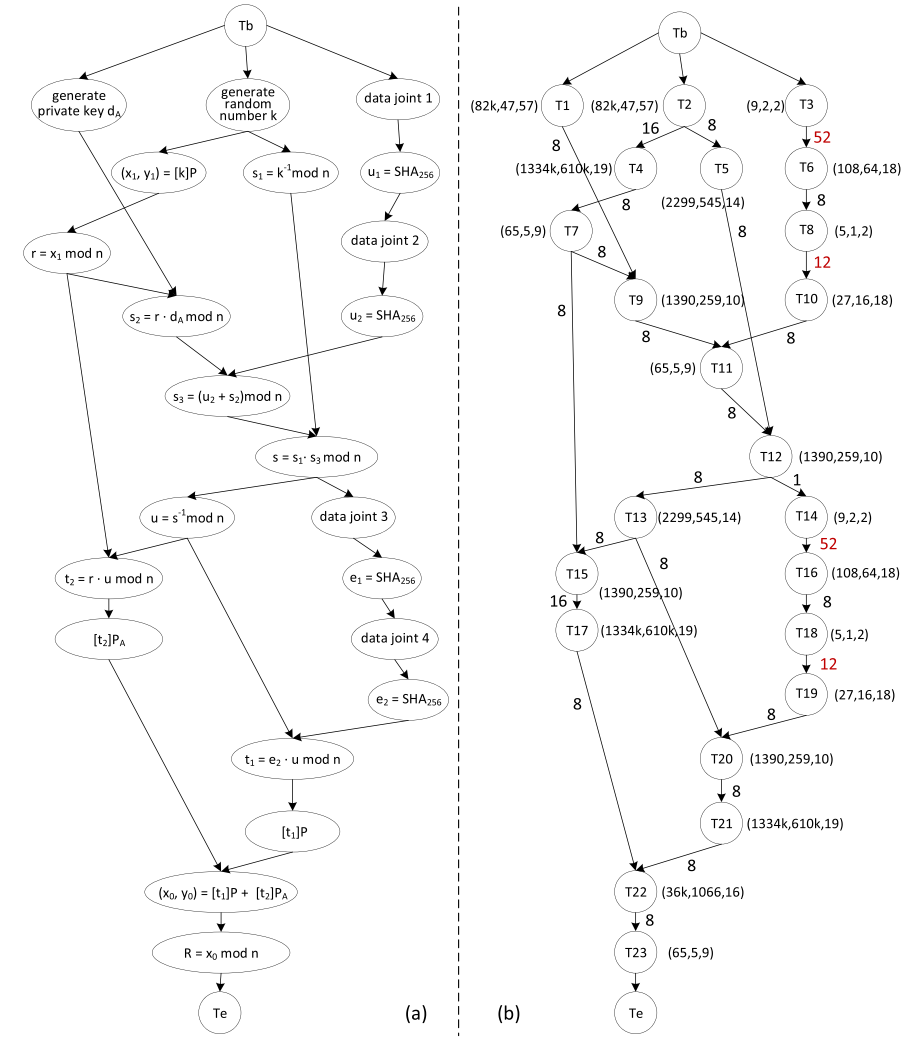


Fig. 15 The task graph construction of digital signature system

The pure software and hardware design of this system are implemented in C and Verilog, respectively. Based on the procedure of the ECDSA algorithm, the coarse-grained task graph can be generated as shown in Fig. 15a. The digital signature system can be divided into 23 tasks, and the features of these tasks are illustrated in Fig. 15b.

After constructing the task graph of the ECDSA digital signature system, partitioning and scheduling are performed using GCPS and the other five compared metaheuristics, with the cost function T_{SL} and two area constraints of AP = 1/3 and AP = 1/2. Through experimental comparison with the other five methods, the performance results are shown in Table 8. Initial strategy represents the strategy used to generate the initial solution. Initial AR is the initial acceleration ratio calculated from the initial solution. Best AR is the optimal acceleration ratio achieved during the search process. Search time means that the model can find the optimal solution at the current epoch. Improvement GCPS versus indicates that the AR improvement

Table 8 The comparison of GCPS and other five comparators of digital signature system

Model	AP	Initial strategy	Initial AR	Best AR	Search time	Improvement GCPS versus
GA	1/3	Random	1.958	3.219	17	0.217
	1/2		1.283	3.431	9	0.321
HGA	1/3	Zcline	1.989	3.223	1	0.093
	1/2		3.136	3.433	6	0.262
SA	1/3	Random	1.21	3.224	10	0.062
	1/2		1.239	3.442	46	0.000
GSA	1/3	GA	3.213	3.226	2	0.000
	1/2		3.213	3.439	3	0.087
KL	1/3	Random	3.129	3.216	5	0.311
	1/2		3.223	3.439	9	0.087
GCPS	1/3	Random	3.216	3.226	6	0.000
	1/2		3.224	3.442	10	0.000

of GCPS compared with the current model. For GA, SA, and KL, the initial solution strategy adopted by these three methods is random, while HGA uses the hardware inclination (Zcline) to generate a better initial solution. The initial solution of GSA is generated by GA. It can be seen from Table 8 that the graph convolution operation can obtain a better initial solution in the initial stage so that the model can converge faster. The search time means that the model can find the optimal solution at the current epoch. In contrast, the GA, SA, and KL approaches converge faster in this case, and the time to find a better solution is relatively short, but it is easy to fall into a locally optimal solution. Besides, it can be seen that when the area percentage is 1/2, the SA can search for the same solution as the GCPS, but its search time is long. When the area percentage is 1/3, the GSA can quickly search for the same solution as our approach, but this does not indicate that the GCPS is inefficient. In this real-life example, since the task graph of the system is unique and the number of tasks is small, the search solution space will be small, and it is easier to quickly find the optimal solution, resulting in a not high improvement ratio. In conclusion, compared with other methods, the GCPS can achieve a better trade-off between HW/SW partitioning efficiency and quality, which verifies the feasibility of the GCPS.

6 Conclusion

We have presented a HW/SW partitioning method based on graph convolution network (GCPS) to find out a partitioning solutions which has minimized schedule length under a given hardware area constraint. The proposed partitioning algorithm is based on GCN and LSSP to improve partitioning efficiency and generates better solutions for large-scale systems. The experiment results show that the AR improvement of GCPS is more than 10% in large-scale systems compared with previous works, and our approach is 18× faster than KL for large-scale systems. In the future, GCPS extended to the system of multi-processors and reconfigurable devices will be researched.

Acknowledgements The authors would like to thank my supervisors, Prof. Zebo Peng and Prof. Petru Eles, for their good ideas, valuable guidance, and patient support throughout this research. This work is sup-

ported in part by the Science and Technology Planning Project of Guangdong Province of China under Grant 2019B010140002.

References

1. Abdelzaher TF, Shin KG (2000) Period-based load partitioning and assignment for large real-time applications. IEEE Trans Comput 49(1):81–87
2. Ahmed A, Hasan T, Abdullatif FA, Mustafa S, Rahim MSM (2019) A digital signature system based on real time face recognition. In: 2019 IEEE 9th International conference on system engineering and technology (ICSET). IEEE, pp 298–302
3. Arató P, Juhász S, Mann ZÁ, Orbán A, Papp D (2003, September) Hardware-software partitioning in embedded system design. In: IEEE International Symposium on Intelligent Signal Processing, 2003. IEEE, pp 197–202
4. Atwood J, Towsley D (2016) Diffusion-convolutional neural networks. In: Advances in neural information processing systems, pp 1993–2001
5. Banerjee S, Dutt N (2004, September) Efficient search space exploration for HW-SW partitioning. In: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, pp 122–127
6. Banerjee S, Dutt N (2004) Very fast simulated annealing for HW–SW partitioning. In: Technical Report, CECS-TR-04-17. Citeseer
7. Chatha KS, Vemuri R (2002) Hardware–software partitioning and pipelined scheduling of transformative applications. IEEE Trans Very Large Scale Integr (VLSI) Syst 10(3):193–208
8. Chen J, Zhu J, Song L (2017) Stochastic training of graph convolutional networks with variance reduction. arXiv preprint arXiv:1710.10568
9. Dick RP, Rhodes DL, Wolf W (1998) TGFF: task graphs for free. In: Proceedings of the sixth international workshop on hardware/software codesign (CODES/CASHE'98). IEEE, pp 97–101
10. Eles P, Peng Z, Kuchcinski K, Doboli A (1997) System level hardware/software partitioning based on simulated annealing and tabu search. Des Autom Embed Syst 2(1):5–32
11. Ernst R, Henkel J, Benner T (1993) Hardware–software cosynthesis for microcontrollers. IEEE Des Test Comput 10(4):64–75
12. Guo B, Wang D, Shen Y, Liu Z (2006) Hardware–software partitioning of real-time operating systems using Hopfield neural networks. Neurocomputing 69(16–18):2379–2384
13. Gupta RK, De Micheli G (1993) Hardware–software cosynthesis for digital systems. IEEE Des Test Comput 10(3):29–41
14. Han H, Liu W, Wu J, Jiang G (2013) Efficient algorithm for hardware/software partitioning and scheduling on MPSoC. JCP 8(1):61–68
15. He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: 2016 IEEE Conference on computer vision and pattern recognition, CVPR 2016, Las Vegas, NV, USA, June 27–30. IEEE Computer Society, pp 770–778
16. Henkel J, Ernst R (2001) An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques. IEEE Trans Very Large Scale Integr (VLSI) Syst 9(2):273–289
17. Hou N, Yan X, He F (2019) A survey on partitioning models, solution algorithms and algorithm parallelization for hardware/software co-design. Des Autom Embed Syst 23(1–2):57–77
18. Huang W, Zhang T, Rong Y, Huang J (2018) Adaptive sampling towards fast graph representation learning. arXiv preprint arXiv:1809.05343
19. Jemai M, Dimassi S, Ouni B, Mtibaa A (2017) A metaheuristic based on the tabu search for hardware-software partitioning. Turk J Electr Eng Comput Sci 25(2):901–912
20. Jiang G, Wu J, Lam SK, Srikanthan T, Sun J (2015) Algorithmic aspects of graph reduction for hardware/software partitioning. J Supercomput 71(6):2251–2274
21. Jigang W, Chang B, Srikanthan T (2009, June) A hybrid branch-and-bound strategy for hardware/software partitioning. In: 2009 Eighth IEEE/ACIS International Conference on Computer and Information Science. IEEE, pp 641–644
22. Jing Y, Kuang J, Du J, Hu B (2013, May) Application of improved simulated annealing optimization algorithms in hardware/software partitioning of the reconfigurable system-on-chip. In: International Conference on Parallel Computing in Fluid Dynamics. Springer, Berlin, Heidelberg, pp 532–540
23. Kalavade A, Lee EA (1997) The extended partitioning problem: hardware/software mapping, scheduling, and implementation-bin selection. Des Autom Embed Syst 2(2):125–163

24. Kalavade A, Subrahmanyam P (1998) Hardware/software partitioning for multifunction systems. IEEE Trans Comput Aided Des Integr Circuits Syst 17(9):819–837
25. Kipf TN, Welling M (2016) Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907
26. Kipf TN, Welling M (2016) Variational graph auto-encoders. arXiv preprint arXiv:1611.07308
27. Li G, Feng J, Wang C, Wang J (2014) Hardware/software partitioning algorithm based on the combination of genetic algorithm and tabu search. Eng Rev Međunarodni časopis namijenjen publiciranju originalnih istraživanja s aspekta analize konstrukcija, materijala i novih tehnologija u području strojarstva, brodogradnje, temeljnih tehničkih znanosti, elektrotehnike, računarstva i građevinarstva 34(2):151–160
28. Li Q, Han Z, Wu XM (2018) Deeper insights into graph convolutional networks for semi-supervised learning. arXiv preprint arXiv:1801.07606
29. Li SG, Feng FJ, Hu HJ, Wang C, Qi D (2014) Hardware/software partitioning algorithm based on genetic algorithm. JCP 9(6):1309–1315
30. Li Y, Hao Z, Lei H (2016) Survey of convolutional neural network. J Comput Appl 36(9):2508–2515
31. Li Y, Vinyals O, Dyer C, Pascanu R, Battaglia P (2018) Learning deep generative models of graphs. arXiv preprint arXiv:1803.03324
32. Li Y, Yu R, Shahabi C, Liu Y (2017) Diffusion convolutional recurrent neural network: data-driven traffic forecasting. arXiv preprint arXiv:1707.01926
33. López-Vallejo M, López JC (2003) On the hardware–software partitioning problem: system modeling and partitioning techniques. ACM Trans Des Autom Electron Syst (TODAES) 8(3):269–297
34. Madsen J, Grode J, Knudsen PV, Petersen ME, Haxthausen A (1997) LYCOS: the Lyngby co-synthesis system. Des Autom Embed Syst 2(2):195–235
35. Mann Z, Orbán A, Farkas V (2007) Evaluating the Kernighan–Lin heuristic for hardware/software partitioning. Int J Appl Math Comput Sci 17(2):249–267
36. Mann ZÁ, Orbán A, Arató P (2007) Finding optimal hardware/software partitions. Formal Methods Syst Des 31(3):241–263
37. Nair V, Hinton GE (2010, January) Rectified linear units improve restricted boltzmann machines. In: International Conference on Machine Learning, pp 807–814
38. Niemann R, Marwedel P (1997) An algorithm for hardware/software partitioning using mixed integer linear programming. Des Autom Embed Syst 2(2):165–193
39. Pornin T (2013) Deterministic usage of the digital signature algorithm (DSA) and elliptic curve digital signature algorithm (ECDSA). Internet Eng Task Force RFC 6979:1–79
40. Purnaprajna M, Reformat M, Pedrycz W (2007) Genetic algorithms for hardware–software partitioning and optimal resource allocation. J Syst Archit 53(7):339–354
41. Radulescu A, Van Gemund AJ (2002) Low-cost task scheduling for distributed-memory machines. IEEE Trans Parallel Distrib Syst 13(6):648–658
42. Saha D, Mitra RS, Basu A (1997, January) Hardware software partitioning using genetic algorithm. In: Proceedings Tenth International Conference on VLSI Design. IEEE, pp 155–160
43. Shi W, Wu J, Lam SK, Srikanthan T (2016) Algorithms for bi-objective multiple-choice hardware/software partitioning. Comput Electr Eng 50:127–142
44. Srinivasan V, Radhakrishnan S, Vemuri R (1998, February) Hardware software partitioning with integrated hardware design space exploration. In: Proceedings Design, Automation and Test in Europe. IEEE, pp 28–35
45. Teich J (2012) Hardware/software codesign: the past, the present, and predicting the future. Proc IEEE 100(Special Centennial Issue):1411–1430
46. Trindade AB, Cordeiro LC (2016) Applying SMT-based verification to hardware/software partitioning in embedded systems. Des Autom Embed Syst 20(1):1–19
47. Veličković P, Cucurull G, Casanova A, Romero A, Lio P, Bengio Y (2017) Graph attention networks. arXiv preprint arXiv:1710.10903
48. Wu J, Srikanthan T (2006) Low-complex dynamic programming algorithm for hardware/software partitioning. Inf Process Lett 98(2):41–46
49. Wu J, Srikanthan T, Chen G (2009) Algorithmic aspects of hardware/software partitioning: 1D search algorithms. IEEE Trans Comput 59(4):532–544
50. Wu J, Sun Q, Srikanthan T (2012) Algorithmic aspects for multiple-choice hardware/software partitioning. Comput Oper Res 39(12):3281–3292
51. Wu MY, Gajski DD (1990) Hypertool: a programming aid for message-passing systems. IEEE Trans Parallel Distrib Syst 1(3):330–343
52. Wu Z, Pan S, Chen F, Long G, Zhang C, Philip SY (2020) A comprehensive survey on graph neural networks. IEEE Trans Neural Netw Learn Syst 32:4–24

53. Zill D, Wright WS, Cullen MR (2011) Advanced engineering mathematics. Jones & Bartlett Learning, Burlington
54. Zou Y, Zhuang Z, Chen H (2004, June) HW-SW partitioning based on genetic algorithm. In: Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753), vol 1. IEEE, pp 628–633
55. Zuo W, Pouchet LN, Ayupov A, Kim T, Lin CW, Shiraishi S, Chen D (2017, June) Accurate high-level modeling and automated hardware/software co-design for effective SoC design space exploration. In: Proceedings of the 54th Annual Design Automation Conference 2017, pp 1–6