DOI: https://doi.org/10.24297/ijct.v20i.8533

Model-based Parallelization for Simulink Models on Multicore CPUs and GPUs

Zhaoqian Zhong[1], Masato Edahiro[2]

[1]Ph.D. candidate, Graduate School of Information Science, Nagoya University

[2]Professor, Graduate School of Informatics, Nagoya University

[1]zhaoqian@ertl.jp

Abstract

In this paper we propose a model-based approach to parallelize Simulink models of image processing algorithms on homogeneous multicore CPUs and NVIDIA GPUs at the block level and generate CUDA C codes for parallel execution on the target hardware. In the proposed approach, the Simulink models are converted to directed acyclic graphs (DAGs) based on their block diagrams, wherein the nodes represent tasks of grouped blocks or subsystems in the model and the edges represent the communication behaviors between blocks. Next, a path analysis is conducted on the DAGs to extract all execution paths and calculate their respective lengths, which comprises the execution times of tasks and the communication times of edges on the path. Then, an integer linear programming (ILP) formulation is used to minimize the length of the critical path of the DAG, which represents the execution time of the Simulink model. The ILP formulation also balances workloads on each CPU core for optimized hardware utilization. We parallelized image processing models on a platform of two homogeneous CPU cores and two GPUs with our approach and observed a speedup performance between 8.78x and 15.71x.

Keywords: homogeneous multicore CPU, NVIDIA GPU, parallelization, model-based development, MATLAB Simulink

1. Introduction

Model-based development (MBD) with platforms such as MATLAB Simulink [1] has been widely used in modeling and simulating complex systems. In many cases of MBD, a Simulink model (or a part of the model) is used to process a large amount of data, such as in image processing [2] and scientific data calculation [3]. It is possible to accelerate the execution of such model-based applications by executing the data parallelism blocks on graphical processing units (GPUs) rather than on central processing unit (CPU) cores for improved performance [2]. For implementing such Simulink models on a platform consisting of both CPUs and GPUs, it is critical to extract the blocks of data parallelism for execution on GPUs and parallelizing Simulink blocks for proper workload balance on CPU cores.

In this paper we propose a model based approach to parallelize the Simulink models of image processing on homogeneous multicore CPUs and NVIDIA GPUs [4]. The target architecture is a platform of homogeneous CPU cores and identical GPUs, where the number of CPU cores and GPUs is equal, thereby enabling multiple CUDA kernels to be executed concurrently. An available integer linear programming (ILP) formulation is proposed to assign blocks, which are used for image processing, to GPUs and other blocks to CPU cores to (a) minimize the execution time of the whole model on the target platform, and to (b) distribute the workloads of execution on CPU cores in a balanced manner. The main contribution of our work is that the proposed approach implements Simulink models that are used to process image data on a platform of homogeneous CPU cores and GPUs. We obtained a speedup performance between 8.78x and 15.71x from the evaluation experiments, which demonstrates the effectiveness of the proposed approach.

Section 2 gives a brief background on related work and previous studies. Section 3 is an overview of our proposed approach. Section 4 describes the ILP formulation we use to assign blocks and subsystems of the model to the target platform, and Section 5 describes the generation of parallel codes for execution on the target platform. In Section 6 we implement image processing algorithms with Simulink and parallelize the models with the proposed approach.

2. Related Work

MATLAB Simulink is a graphical programming environment for modeling, simulating and analysing dynamical systems [1]. Developers can describe their systems in a Simulink model, which is a diagram of function blocks and relations between blocks [5], and implement systems on the hardware based on the model. Currently, a large amount of research is being done on implementing control models in MBD with Simulink. Commonly, Simulink blocks are partitioned and parallelized to processing elements of the target platform using heuristic algorithms [6] or mathematical methods [7, 8, 9], and execution codes are generated based the sequential codes generated by Simulink Embedded Coder [10]. In previous studies, we used integer linear programming (ILP)-based approaches to parallelize Simulink models on homogeneous multicore processors [11] and single-ISA heterogeneous multicore processors [12]. The objective of the ILP formulations is to reduce the execution time of the models on the target processors by minimizing the inter-core communication cost. We also provided parallel code generation for execution on processors and observed reasonable speedup performance in implementing real-scenario models. On the other hand, even though Simulink is widely used for control design, it is possible to implement data parallelism algorithms, such as in image processing and scientific data calculation, in Simulink [2, 3, 13]. In such cases of MBD, a large amount of input data is stored to the workspace of MATLAB, and a Simulink model or a part of the model with the combinations of basic Simulink blocks and MATLAB function blocks can be used to process the data. When implementing such models to hardware, we can use GPUs to execute the data parallelism parts for improved performance. For example, a Simulink model for valve body failure mode detection is implemented using CUDA to be executed on GPU for parallel computation in [3]. The model is first converted to sequential codes with Simulink Coder and then the codes of detecting failure mode are converted to a CUDA kernel to be executed on GPU. If the data processing parts of the model are not continuous and distributed on different paths in the block diagram, we can convert them to multiple CUDA kernels and use multi-core processor to execute the model for parallel execution. Moverover, if there are more than one GPU, the CUDA kernels can be executed concurrently on different GPUs for better parallelism. To achieve this goal, we focus on execution platforms where GPUs and CPU cores are in equal numbers, so that when multiple GPU kernels are launched simultaneously, they can be executed concurrently to parallelize the model execution for improved efficiency.

However, even though such platform of CPUs and GPUs represents a heterogeneous architecture of multiple instruction sets [14], the data communication overhead between the CPU and the GPU is much heavier than inter-core communication overhead between CPU cores. Therefore, the sum of the GPU-CPU overhead and the execution time of a kernel executed on GPUs may be larger than the execution time of this kernel executed on a CPU core. Hence, minimizing communication cost alone [11, 12] cannot solve the parallelization problem with GPUs. Moreover, although impressive speedups have been reported for many cases of GPU execution [15, 16], the effective speedup improvement of models implemented on the GPU needs to be calculated considering the whole execution on both CPUs and GPUs [17, 18]. Therefore, we use the speedup between the execution of the parallel code execution on both GPUs and CPUs and the sequential code execution on CPU alone as the metric to evaluate the parallelization approach.

3. Overview of the Proposed Approach

Fig. 1 provides an overview of the proposed approach for model-based parallelization of image processing models on homogeneous multicore CPU cores and NVIDIA GPUs. It is used to find the parallelization solution at the block level for input Simulink models and to generate CUDA C code for execution on the target architecture. The proposed approach targets single-rate Simulink models in which image processing algorithms are implemented by basic blocks or MATLAB function blocks in MATLAB 2017a MBD environment. Since the ILP formulation used in the proposed approach is based on the communicating sequential process (CSP), the diagram of the input model must be able to be converted to directed acyclic graphs (DAGs) without feedback edges, and the DAG must have at least two paths from the start vertice to the termination vertice.

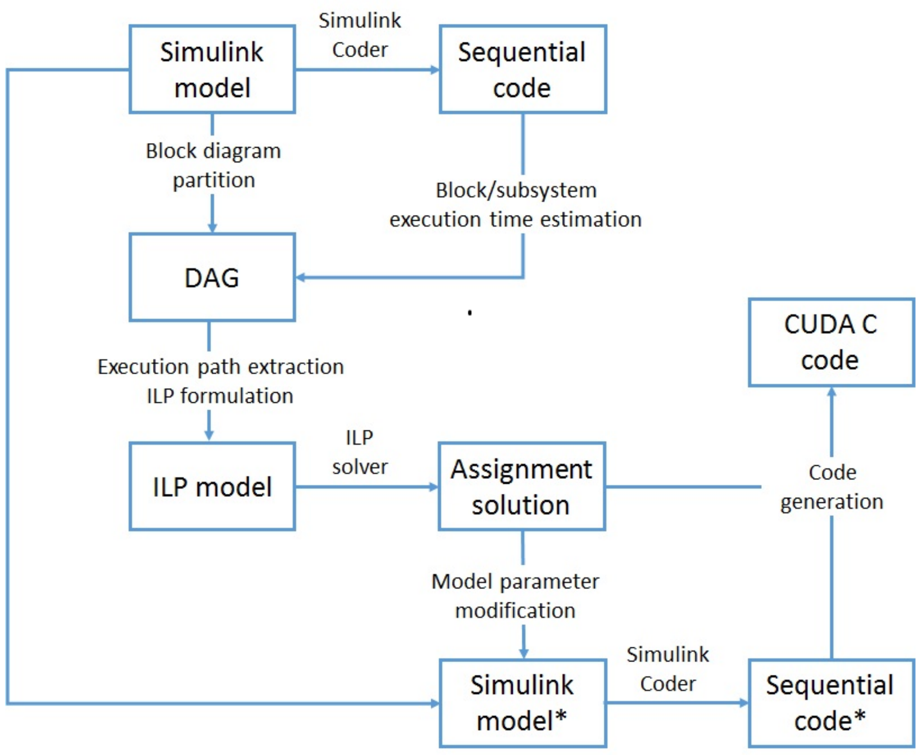


Fig. 1 An overview of proposed approach

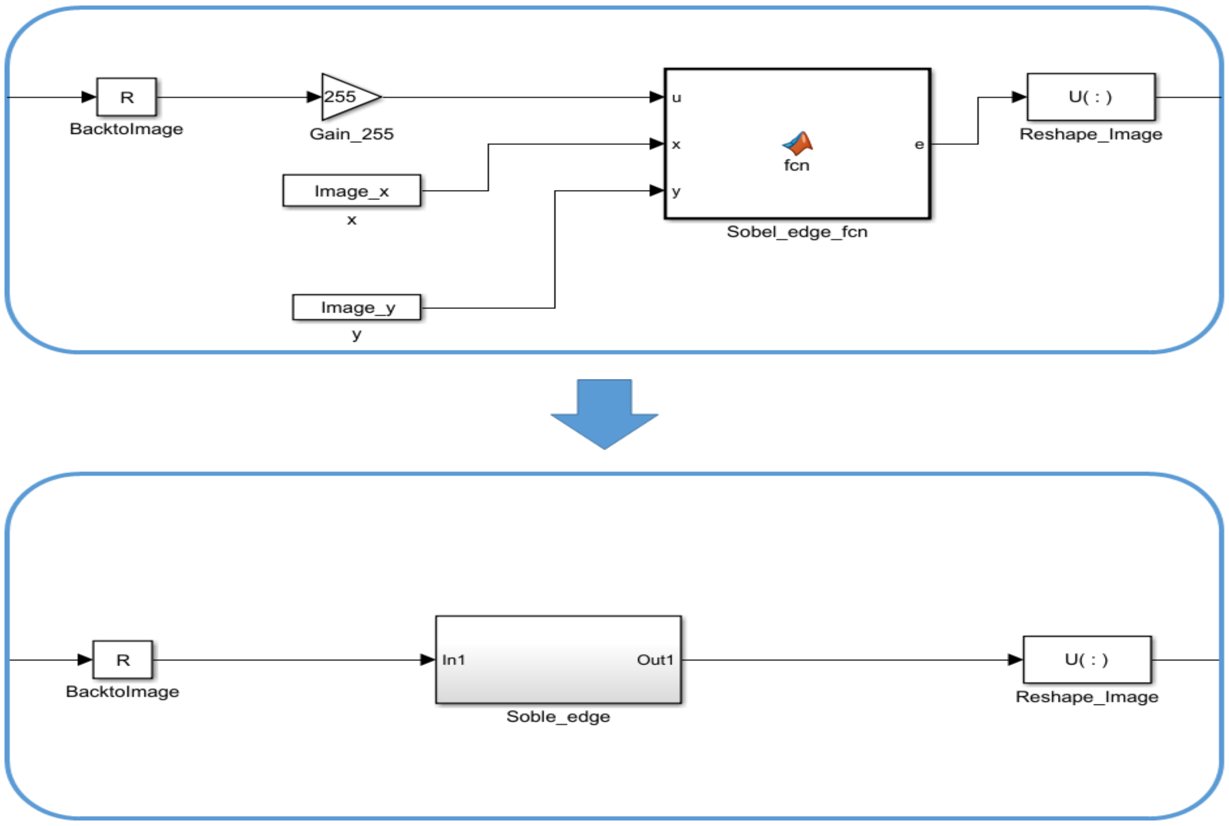
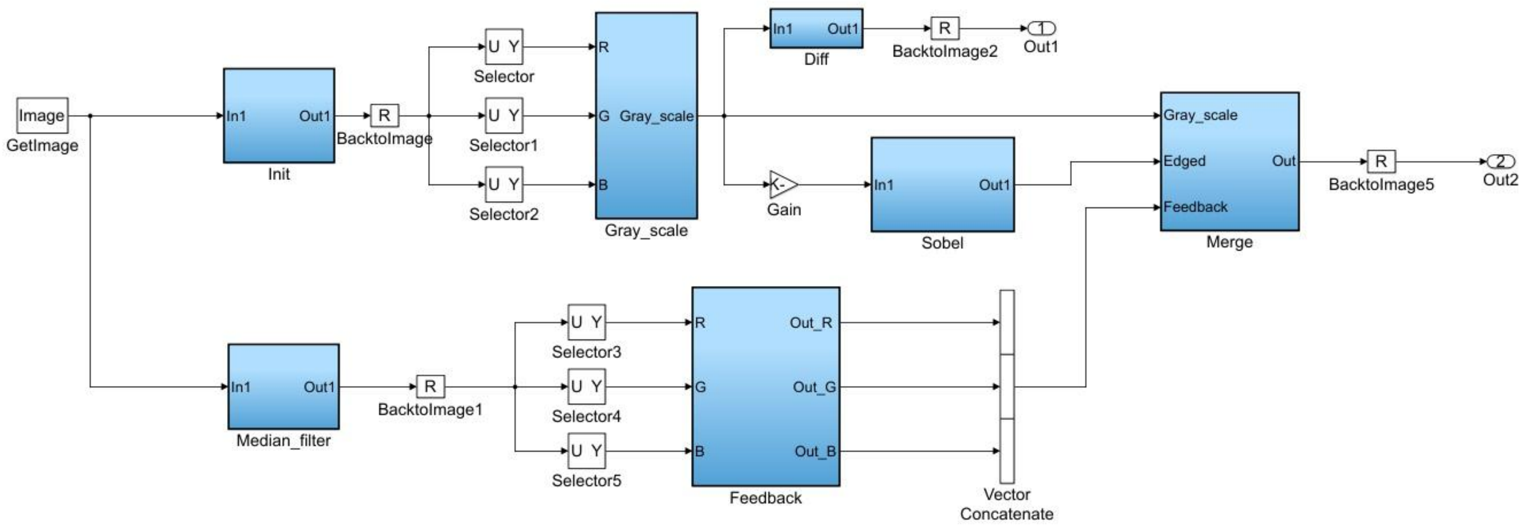


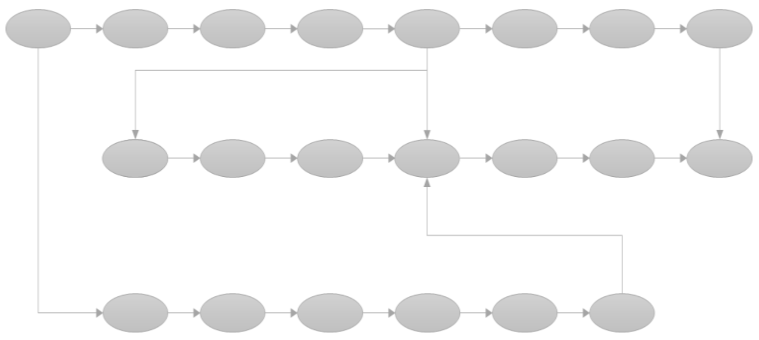
Fig. 2 Example of grouping continuous blocks of image processing into an atomic subsystem

For applications that focus on floating point computation on large amounts of data, GPUs can accelerate them by performing computations in parallel, which are much faster than the traditional CPUs [15, 16, 18]. To take this advantage in MBD, we firstly specify the data type of blocks/subsystems, which are used to process the image data, to 'single' in MATLAB Simulink environment, and cluster the continuous blocks and subsystems of image processing into atomic subsystems, with the clustering method similar to [11, 12]. The clustered blocks and subsystems should have the same input data size and data type, or shoud be constant blocks connected to a block of image processing. Special blocks such as reshape blocks and selector blocks are not clustered to atomic subsystems. The function packaging option of these atomic subsystems are set to 'inline' [19]. Therefore, using Simulink Embedded Coder the code of the blocks and subsystem inside these atomic subsystems can be merged and converted to an independent forloop in the generated sequential code files, and we can generate CUDA kernels based on the forloops of the atomic subsystems.

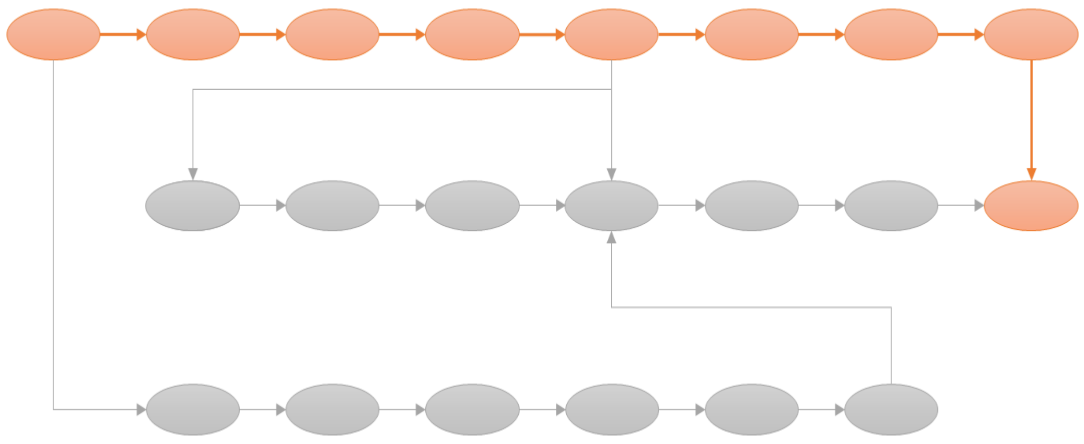
Moreover, the ILP formulation we use in the proposed approach is based on the graph extracted from the input model. By clustering blocks and subsystems we can greatly reduce the number of nodes and edges in the graph, and therefore reduce the numbers of variables and constants in the formulation, thereby reducing the solver time of the ILP formulation. In Fig. 2, the blocks Gain_255 and MATLAB function block Sobel_edge_fcn are used to process pixel data of the input image in a Simulink model. They are clustered to an atomic subsystem Sobel_edge with constant blocks x and y, and their data type are specified to single for being converted to a CUDA kernel in code generation.



(a) Simulink model of image processing algorithm



(b)DAG of the Simulink model



(c)An execution path on the DAG of the Simulink model

Fig. 3 Example of a Simulink model for image processing and its DAG

Secondly, we convert the model to a directed acyclic graph (DAG) based on the block diagram of the model and add dummy nodes and edges to the DAG based on the communicating sequential process (CSP). The dummy nodes ensure that execution of this graph starts at only node and terminates in one node, and the nodes that may be assigned to GPUs are launched and terminated on the same CPU core. Based on [5], we define the DAG as $G = (V, E)$, where V is the set of nodes to which blocks or subsystems are clustered, and $E \subseteq V \times V$ is set of the connections between nodes, which are extracted from the signal lines between blocks and subsystems. Fig. 3 (a) shows Simulink model of image processing algorithm and Fig. 3 (b) represents the directed acyclic graph (DAG) converted from the model in Fig. 3 (a). To numericalize the execution of the DAG we have to estimate the execution times of these nodes, communication overhead between CPU cores and communication overhead between a CPU core and a GPU on the target platform. As estimation of execution time is difficult on a heterogeneous architecture with GPUs [20], we generate the sequential code of the input model with Simulink Embedded Coder, and execute it on the target hardware. The execution times on CPU cores can be obtained from the execution of sequential codes. To estimate the execution time on GPUs, we converted the codes of atomic subsystems, which consist of blocks and subsystems of image processing, to CUDA kernels using the method in [3], and record the execution times of these kernels as the GPU execution times of these nodes.

Next, we assign the DAG to the processing elements of the target platform with an ILP formulation. We extract all execution paths of the DAG using depth-first search. An execution path consists of the sequence of connections all directed in the same direction, and a sequence of nodes from the start node to the termination node of the DAG and joined by these connections. Each execution path represents an independent dataflow execution instance of the model. Fig. 3 (c) shows one of the execution paths extracted from the DAG in Fig. 3 (b) by depth-first search. We use the sum of the execution times of nodes and communication times of connections to denote the execution time of an execution path, which is dynamically calculated in the ILP formulation. The longest execution path is the critical path of the DAG, and the execution time of the critical path represents the execution time of the model on the target platform. Therefore, the objective function of the ILP formulation is to minimize the execution time of the critical path. Finally, we modify the model based on the result of the ILP formulation and convert it model to parallel code for execution on the target platform.

4. ILP Formulation

4.1 Target Platform

The target architecture of the proposed approach is a platform of homogeneous CPU cores and NVIDIA GPUs, where the number of CPU cores and GPUs is equal and GPUs are identical. In the proposed approach, we do not allow direct data transfer between GPUs. The number of CPU cores/GPUs is represented by c, and for each CPU core $core_g$, where $g \in [0, c-1]$, all CUDA kernels launched by $core_g$ should be executed on only one GPU GPU_g. We group CPU core $core_g$ and GPU GPU_g as a PE PE_g in our approach. The overhead of copying the input image data from the CPU core to the GPU and transfer the modified data from the the GPU to the CPU core is denoted by $overhead\_gpu\_copy$. Since we use a homogeneous multicore processor and identical GPUs to execute the Simulink models in the proposed approach, we assume that the overhead $overhead\_gpu\_copy$ is identical in each PE PE_g in our approach.

4.2 DAG definition

The DAG of the input model is denoted by $G = (V, E)$, where V is the set of nodes, and $E \subseteq V \times V$ is set of the connections between nodes. A node in V represents an independent task in the execution of DAG, to which blocks, or subsystems are clustered. Connections in E are visualized as edges in the DAG. They represent the data dependence between these tasks, which are extracted from the signal lines between blocks and subsystems in the block diagram of the model.

The number of nodes in V is denoted by m and the set of m nodes is defined as $V = \{t_i | i \in [0, m-1]\}$. Each node is defined as $t_i = (cpu\_util_i,\ gpu\_util_i)$, where cpu_util_i is the estimated execution time of the node t_i on CPU cores, and gpu_util_i is the estimated execution time of the node t_i on GPUs. For nodes that are not clustered for GPU execution, their gpu_util_i are set to MAX_VAL, which is a very large constant value, to avoid being assigned to GPUs. We use $V\_gpu = \{t\_gpu_i | i \in [0, m\_gpu - 1]\}$ to represent the set of nodes that are clustered for GPU execution and m_gpu is the number of such nodes, where $m\_gpu < m$. Each node in V_gpu is denoted by a three-tuple: $t\_gpu_i = (t\_gpu\_id_i, t\_gpu\_from_i, t\_gpu\_to_i)$, where $t\_gpu\_id_i$ represents the ID of the responding node of t_gpu_i in V. $t\_gpu\_from_i$ represents the id of node preceding to t_gpu_i in V, and $t\_gpu\_to_i$ represents the ID of node succeeding to t_gpu_i in V.

The number of connections is denoted by n. We define the set of n connections as $E = \{e_j | j \in [0, n-1]\}$. Each connection is defined as a three-tuple: $e_j = (edge\_s_j, edge\_t_j, edge\_time_j)$, where $edge\_s_j$ and $edge\_t_j$ represent the id of start node $t_{edge\_s_j} \in V$ and termination node $t_{edge\_t_j} \in V$ joined by the connection e_j, and $edge\_time_j$ represents the inter-core overhead of e_j if $t_{edge\_s_j}$ and $t_{edge\_t_j}$ are not assigned to the same core.

Next, we use depth-first search on the DAG to extract all execution paths. The number of execution paths is denoted by r, and the set of r execution paths is denoted by $P = \{p_k | k \in [0, r-1]\}$. Each execution path is defined as $p_k = (V'_k,\ E'_k)$, where $V'_k \subseteq V$ represent the nodes on execution path p_k, and $E'_k \subseteq E$ represent the connection of nodes on execution path p_k.

4.3 Variables

We use the following variables in our ILP formulation to denote which core or GPU a node $t_i \in V$ is assigned to:

- $x\_core_{i,g}$: equals to 1 if node t_i is assigned to CPU core $core_g$; otherwise equals to 0.
- $x\_gpu_{i,g}$: equals to 1 if node t_i is assigned to GPU GPU_g; otherwise equals to 0.

We use the following variables to denote whether the start node and termination node of a connection e_j are assigned to the same CPU core or whether one of them is assigned to GPU:

- y_gpu_j: equals to 1 if either end node, $edge\_s_j$ or $edge\_t_j$, is assigned to a GPU; otherwise equals to 0. For $\forall j \in E$, y_gpu_j is calculated as:

$$\forall j \in E: y\_gpu_j = \sum_{g\in[0,c-1]} (x\_gpu_{edge\_s_j,g} + x\_gpu_{edge\_t_j,g}) \tag{1}$$

- y_cpu_j : equals to 0 if both end nodes, $edge\_s_j$ and $edge\_t_j$, are assigned to the same CPU core; otherwise equals to 1. For $\forall j \in E$, y_cpu_j is calculated as:

$$\begin{aligned}\forall j \in E: y\_cpu_j = &\sum_{g\in[0,c-1]} abs(x\_core_{edge\_s_j,g} - x\_core_{edge\_t_j,g}) \\ &- \sum_{g\in[0,c-1]} abs(x\_gpu_{edge\_s_j,g} - x\_gpu_{edge\_t_j,g})\end{aligned} \tag{2}$$

For each PE_g, we use $workload\_pe_g$ to denote the total workload that is executed on CPU core $core_g$ and GPU_g. Each $workload\_pe_g$ consists of the execution times of nodes which are assigned to $core_g$, the inter-core overhead of e_j whose $edge\_t_j$ is assigned to CPU core $core_g$, and the execution times of nodes on GPU which are assigned to GPU GPU_g. In the proposed approach, we assume that the latency of transferring data from CPU core $core_g$ to other CPU cores can be ignored in $workload\_pe_g$; on the other hand, data transfer from other cores to CPU core $core_g$ cannot be parallelized with processing on CPU core $core_g$, so that the inter-core overhead of transferring data from other cores to CPU core $core_g$ should be added to $workload\_pe_g$. The following variable is used to mark whether a connection e_j is used for transferring data from other cores to CPU core $core_g$:

- $y\_end\_core_{j,g}$: equals to 1 if $edge\_t_j$ is assigned to a CPU core $core_g$; otherwise equals to 0. For $\forall j \in E$, $\forall g \in [0, c-1]$, $y\_end\_core_{j,g}$ is calculated as:

$$\forall j \in E, \forall g \in [0, c-1]: y\_end\_core_{j,g} \le (x\_core_{edge\_t_j,g} + y\_cpu_j)/2 \tag{3}$$

$$\forall j \in E, \forall g \in [0, c-1]: y\_end\_core_{j,g} \ge x\_core_{edge\_t_j,g} + y\_cpu_j - 1 \tag{4}$$

In addition, if any CUDA kernel is launched on the CPU core $core_g$, data transfer overhead $overhead\_gpu\_copy$ should be added to $workload\_pe_g$. We use the following variable to denote whether any CUDA kernel is launched on the CPU core $core_g$:

- z_gpu_g: equals to 1 if at least one node is assigned to GPU_g; otherwise equals to 0. For $\forall g \in [0, c-1]$, z_gpu_g is calculated as:

$$\forall g \in [0, c-1]: z\_gpu_g \le \sum_{i\in V} x\_gpu_{i,g} \tag{5}$$

$$\forall g \in [0, c-1], \forall i \in V: z\_gpu_g \ge x\_gpu_{i,g} \tag{6}$$

For $\forall g \in [0, c-1]$, $workload\_pe_g$ is calculated as:

$$\begin{aligned}\forall g \in [0, c-1]: &workload\_pe_g \\ &= z\_gpu_g * overhead\_gpu\_copy + \sum_{i\in V}(cpu\_util_i * x\_core_{i,g}) \\ &+ \sum_{i\in V}(gpu\_util_i * x\_gpu_{i,g}) + \sum_{j\in E}(edge\_time_j * y\_end\_core_{j,g})\end{aligned} \tag{7}$$

We use $workload\_total$ to denote the total workload on all PEs.

$$workload\_total = \sum_{g\in[0,c-1]} workload\_pe_g \tag{8}$$

4.4 Objective Function

For each execution path p_k, the sum of the estimated execution times of nodes and communication times of connections on the path is used to represent the length of each path. We use p_length_k to present its execution time, which is calculated as:

$$\forall k \in P: p\_length_k = \sum_{i \in V'_k} (cpu\_util_i * (\sum_{g \in [0,c-1]} x\_core_{i,g}) + gpu\_util_i * (\sum_{g \in [0,c-1]} x\_gpu_{i,g})) + \sum_{j \in E'_k} (edge\_time_j * y\_cpu_j) \tag{9}$$

The longest of these execution paths is the critical path of the DAG, the length of which represents the total execution time of the model on the target platform. We use p_length_max to denote the execution time of the longest execution path, where $\forall k \in P$: $p\_length\_max \geq p\_length_k$. Therefore, the objective function of the ILP formulation, which aims to minimize the length of the critical path to reduce the execution time of the model, is:

$$\boldsymbol{minimize\ p\_length\_max} \tag{10}$$

4.5 Constraints

The constraints for the core assignment problem are defined as:

- Each node shall be assigned to only one CPU core or GPU:

$$\forall i \in V: \sum_{g \in [0,c-1]} x\_core_{i,g} + \sum_{g \in [0,c-1]} x\_gpu_{i,g} = 1 \tag{11}$$

- If t_gpu_i is assigned to a GPU, the nodes preceding and succeeding to t_gpu_i shall be assigned the same CPU core:

$$\forall i \in V\_gpu, g \in [0, c-1]: x\_core_{t\_gpu\_from_i,g} + x\_core_{t\_gpu\_to_i,g} \geq 2x\_gpu_{t\_gpu\_id_i,g} \tag{12}$$

- To utilize all cores and GPUs given in the proposed, we set a lower limit to $workload\_pe_g$:

$$\forall g \in [0, c-1]: workload\_pe_g \geq 1 \tag{13}$$

- To assign nodes to each PE in a balance manner and avoid the proposed formulation from 'no solution' result, we set a upper limit to $workload\_pe_g$ and a user-given $threshold$, which we set to 1.5 in our experiments:

$$\forall g \in [0, c-1]: workload\_pe_g \leq workload\_total/c * threshold \tag{14}$$

5. Execution Code Generation

The proposed LIP formulation is implemented by Optimization Programming Language [21]. Firstly, based on the assignment solution, y_cpu_j determines whether the two nodes connected by an edge are placed on two different CPU cores, and we can locate the corresponding signal lines in the model block diagram. To reduce the size of data that is transferred between different CPU cores, we should firstly set the storage class parameter of these signal lines to 'ExportedGlobal'. 'ExportedGlobal' exports the data transferred by a signal line as a global variable [19], with which inter-core communication transfers only the necessary data instead of the whole input data. Next, we generate the sequential code of the modified model using Simulink Embedded Coder. We implement the host code of the model using POSIX Threads. We use the code generation method in [11] to generate POSIX threads for homogenous CPU cores. For each group of CPU core $core_g$ and GPU_g, we create one pthread and specify an independent core on the processor to execute it using pthread affinity. At the beginning of each pthread of CPU core $core_g$, we firstly specify a GPU GPU_g to execute all CUDA kernels launched in the pthread, and copy the image data in parameters of auto storage [19] in the sequential code to GPU_g. Due to the code generation of Simulink Embedded Coder, the data in parameters of auto storage remains unchanged during the execution of the sequential code, therefore, we only need to copy this data to GPUs once at the beginning of the pthread. We then copy the code of step function in the sequential code to each pthread. Based on the assignment solution, $x\_core_{i,g}$ and $x\_gpu_{i,g}$ determine to which GPU or CPU core a node is assigned, and we expand the node assignment solution to the assignment of Simulink blocks and subsystems. Then, for the pthread of each group of CPU core $core_g$ and GPU_g, we delete the codes of blocks or subsystems that are not assigned to CPU core $core_g$ and GPU_g. At the end of each pthread of CPU core $core_g$ and GPU_g, the computation results in external outputs, which is generated from root outports fed by signals with auto storage in sequential code [19], are copied to CPU core $core_g$ from GPU_g. User should add necessary functions to merge computation results from each core and generate the overall output based on their model design. As the number of GPUs is equal to the number of CPU cores in the target platform, concurrently launched CUDA kernels do not have to wait for available GPU resources during execution. For each atomic subsystem whose $x\_gpu_{i,g} = 1$, its forloop code is converted to a CUDA kernel using the method in [2]. After each CUDA kernels is launched, synchronization in the host code is necessary to ensure that the execution order of the generated code is identical to the Simulink blocks. If a signal line between two blocks is cut between two different CPU cores, we need to transfer the data of this signal line between the two cores [11, 12]. Since the storage class parameters of such signal lines are set to 'ExportedGlobal', we can build this communication behavior by passing the values of global variables generated from these signal lines. Besides, if this signal line is used to transfer the processed image data from $core_i$ to $core_j$, we should (a) copy the image data in block signals of auto storage from GPU_i to $core_i$, (b) transfer the image data in block signals of auto storage to $core_j$, and (c) copy the image data in block signals of auto storage from $core_j$ to GPU_j. Note that such inter-core communication of image data is rather heavy, and can be avoided by the proposed ILP formulation.

6. Evaluation Experiments

6.1 Experimental setup

To evaluate our approach we implemented image processing algorithms in Simulink models and parallelized thems on a platform of homogeneous multicore CPU processors and GPUs. The implementation of models and code generation were done in MATLAB 2017a MBD environment. The target platform for the evaluation experiments consisted of two cores of a 4.00 GHz Intel i7-6700k CPU and two TITAN X (Pascal) GPUs. The ILP formulations of parallelizing these models were implemented to executable model files in Optimization Programming Language, and we used IBM ILOG CPLEX Optimization Studio 12.7.0.0 [21] to solve the formulations. The acceptable upper time limit of CPLEX execution was set to 5 hours.

In the implementation of the image processing algorithms in Simulink, the image data were read and stored to workspace in the initfcn of the models. The size of input image was 4872*2824, which was close to the buffer size limit in MATLLAB Simulink 2017a environment. The models in our experiment were:

- Sobel_edge: this model performs grayscale reconstruction and edge detection using Sobel operator [16] on the input image, and merges the edged image with the original image.
- Color histogram(CH): this model extracts color data of each pixel and equalizes them to a close range based on the histograms of each color space [16].
- Adaptive histogram equalization (AHE): this model partitions the input image to 8*8 sub-images and redistribute the lightness of the input image based on their histograms. We used foreach subsystem to implement the repeated processing.

6.2 Speedup Performance

We parallelized the models of image processing with the proposed approach and executed the generated CUDA C codes on the target platform. We recorded the average times for executing these codes. Fig. 4 shows the speedup performance of these image processing models implemented on the target platform, where speedup metric is the ratio between the average execution time of sequential C code executed on only one CPU core, and the average execution time of CUDA C code executed on the platform of two CPU cores and two GPUs.

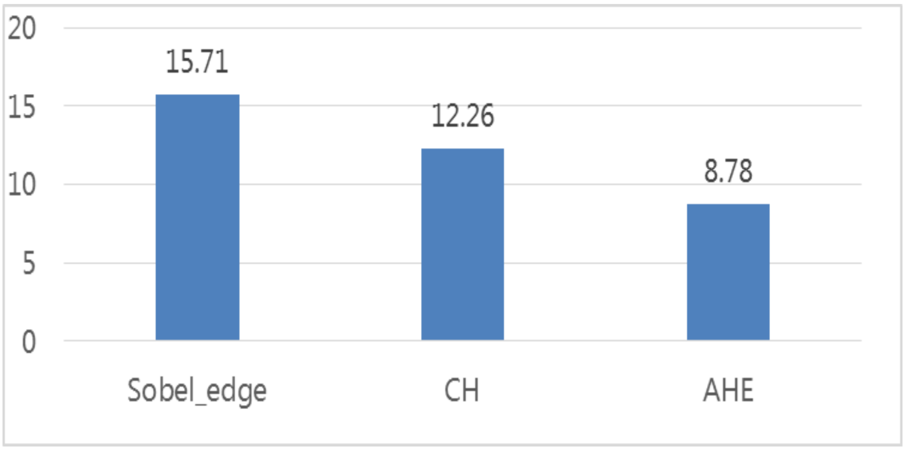


Fig. 4 Speedup performance of parallel codes generated by the proposed approach

The results show that the execution with GPU achieved a speedup between 8.78x and 15.71x as compared to sequential C implementation on only one CPU core. Using GPU to accelerate the execution of image processing blocks it is possible to reduce the execution time of the whole model while parallelizing other blocks on CPU cores for workload balance. The experiment results demonstrate the speedup of Sobel_edge is comparatively higher than in other models. This is because Sobel_edge copies the modified image data from GPUs only once in each pthread. Conversely, the speedup of CH and AHE is limited because all execution paths on the DAG of CH and AHE model have the same length. Especially, as AHE uses foreach subsystems to repeat the histogram equalization on each partitioned image, the histogram of each partitioned image is transferred from GPUs to CPU cores after each execution path is executed. Even though the generated histogram data, which is a 1*256 matrix in the AHE model, is small in size, the frequent transfer significantly slows the execution of AHE model.

To gain more insights on the efficiency of the proposed approach we implemented these models with other methods and executed them on the target platform. We present the speedup of these implementations in Table 1, where Target platform denotes how many CPU cores and GPUs are used to execute the implementation. For 2 CPU cores + 2 GPUs the model was parallelized with the proposed approach. For 1 CPU core + 1 GPU we used the sequential C code of the model as the host code and simply converted the atomic subsystems of image processing to CUDA kernels using only one GPU to execute all of them. For 2 CPU cores we used the method in [11] to generate parallel code for execution on homogeneous multicore processors.

Table 1. Speedup of implementation with the proposed approach compared to other methods.

Model	Target platform	Average speedup
Sobel_edge	2 CPU cores + 2 GPUs	15.71
	1 CPU core + 1 GPU	13.14
	2 CPU cores	1.22
CH	2 CPU cores + 2 GPUs	12.26
	1 CPU core + 1 GPU	9.36
	2 CPU cores	1.33
AHE	2 CPU cores + 2 GPUs	8.78
	1 CPU core + 1 GPU	5.77
	2 CPU cores	1.53

As shown in Table 1 the execution of CUDA C code with the proposed approach has over 1x speedup compared to other methods. As the proposed approach is based on MBD methods in [11, 12], it stipulates that GPU can only be used to accelerate the image processing on large amounts of data and the rest of the model are performed on CPU cores. This leads to a lower speedup of the proposed approach for MBD than the typical GPU implementions in [15, 16].

7. Conclusion

In this paper, we addressed a model-based parallelization approach based on ILP to parallelize Simulink models of image processing algorithms to homogeneous multicore CPUs and NVIDIA GPUs. We use ILP-based optimization method to assign blocks and subsystems to GPUs and CPU cores for the minimized execution time and generate CUDA C code of the models for execution on the target platforms. We implemented image processing algorithms with Simulink, and parallelized the models with the proposed approach. On execution of the generated codes, we experimentally demonstrated that the proposed approach achieves a reasonable speedup over implementations with existing methods.

As future work, we predetermine to reduce the manual work in the proposed approach, and visualize the execution of the model on the target platform. This can help model designers to understand how their model is executed in parallel on the target platform. In addition, we plan to introduce general principles of optimizing CUDA application [15] to the proposed approach. Such optimizations accelerate the application execution by optimizing the kernel codes, and depend on the input model separately. Moreover, we intend to expand the proposed approach to solve the parallelization problem for different CPU cores sharing GPU resources, where the number of GPUs is less than the number of used CPU cores. In this situation, CUDA kernels launched by different CPU cores wait for available GPU to be executed, so that the execution delay of pending kernels needs to be considered in the parallelization.

8. Conflicts of Interest

The authors declare no conflicts of interest associated with this manuscript.

Funding Statement

This work is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

References

1. MathWorks, Inc. "Simulation and Model-Based Design." https://jp.mathworks.com/products/simulink.html, 2015.

2. Saini, Mandeep Singh, et al. "Comparative analysis of digital image watermarking techniques in frequency domain using matlab simulink." International Journal of Engineering Research and Applications (IJERA) 2.4 (2012): 2248-9622.

3. Suzuki, Tomonori, et al. "GPGPU-based high performance parallel computation method for valve body failure mode." SAE International Journal of Passenger Cars-Mechanical Systems 9.2016-01-1353 (2016): 301-309.

4. Nvidia, C. U. D. A. "Compute unified device architecture programming guide." 2007.

5. Alur, Rajeev, et al. "Symbolic analysis for improving simulation coverage of Simulink/Stateflow models." Proceedings of the 8th ACM international conference on Embedded software. ACM, 2008.

6. Peranandam, Prakash, et al. "An integrated test generation tool for enhanced coverage of Simulink/Stateflow models." Proceedings of the Conference on Design, Automation and Test in Europe. EDA Consortium, 2012.

7. Höttger, Robert, Lukas Krawczyk, and Burkhard Igel. "Model-based automotive partitioning and mapping for embedded multicore systems." International Conference on Parallel, Distributed Systems and Software Engineering. Vol. 2. No. 1. 2015.

8. Yi, Ying, et al. "An ILP formulation for task mapping and scheduling on multi-core architectures." Proceedings of the conference on design, automation and test in Europe. European Design and Automation Association, 2009.

9. Tuncali, Cumhur Erkan, Georgios Fainekos, and Yann-Hang Lee. "Automatic Parallelization of Simulink Models for Multi-core Architectures." High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on. IEEE, 2015.

10. MathWorks, Inc. "MATLAB Coder Generate C and C++ code from MATLAB code." https://jp.mathworks.com/products/matlab-coder.html, The MathWorks, Inc, 2012.

11. Zhong, Zhaoqian, and Masato Edahiro. "Model-based Parallelizer for Embedded Control Systems on Multicore Processors." IPSJ Journal, 2018, 59.2: 735-747 (in Japanese).

12. Zhong, Zhaoqian, and Masato Edahiro. "Model-Based Parallelizer for Embedded Control Systems on Single-ISA Heterogeneous Multicore." INTERNATIONAL JOURNAL OF COMPUTERS & TECHNOLOGY 19: 7470-7484.

13. MathWorks, Inc. "Implement Data Parallelism in Simulink." https://ww2.mathworks.cn/help/simulink/ug/implement-data-parallelism-in-simulink.html, 2015.

14. Mittal, Sparsh, and Jeffrey S. Vetter. "A survey of CPU-GPU heterogeneous computing techniques." ACM Computing Surveys (CSUR) 47.4 (2015): 69.

15. Ryoo, Shane, et al. "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA." Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. ACM, 2008.

16. Yang, Zhiyi, Yating, Zhu, and Yong, Pu. "Parallel image processing based on CUDA." 2008 International Conference on Computer Science and Software Engineering. IEEE, 2008.

17. Gregg, Chris, and Kim Hazelwood. "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer." (IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software. IEEE, 2011.

18. Mokhtari, Reza, and Michael Stumm. "BigKernel--High Performance CPU-GPU Communication Pipelining for Big Data-Style Applications." 2014 IEEE 28th international parallel and distributed processing symposium. IEEE, 2014.

19. MathWorks, Inc. "Embedded Coder User's Guide." https://www.mathworks.com/help/pdf_doc/ecoder/, The MathWorks, Inc, 2019.

20. Betts, Adam, and Alastair Donaldson. "Estimating the WCET of GPU-accelerated applications using hybrid analysis." 2013 25th Euromicro Conference on Real-Time Systems. IEEE, 2013.

21. CPLEX, IBM ILOG. "12.7, User's Manual for CPLEX." CPLEX division, 2016.