International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 5, Number 1, pages 159–179, January 2015

Development of an Algorithm for Extracting Parallelism and Pipeline Structure from Stream-based Processing flow with Spanning Tree

Shinichi Yamagiwa, Guyue Wang and Koichi Wada
Department of Computer Science / Faculty of Engineering, Information and Systems,
University of Tsukuba
Tsukuba, Ibaraki, JAPAN

Received: August 1, 2014
Revised: November 3, 2014
Accepted: December 3, 2014
Communicated by Susumu Matsumae

Abstract

It is a fashion to use the manycore accelerators to promote the computing power in a computing platform. Especially GPU is one of the main series of the high performance computing, which is also employed by top supercomputers in the world. Programming methods on such accelerators includes development of control programs which accelerators executes to schedule the invocation of the accelerator's kernel program. The kernel program needs to be written based on the stream computing paradigm. Connecting I/Os of the kernel programs, we can develop a large application. When we consider the processing flow as a directed graph, we can implement a GUI-based programming tool for the accelerators. It visualizes a pipeline-based processing flow. However, it is very hard to find a starting point of a complex processing flow. Moreover, although the processing pipeline include the potential parallelism, it is hard for the programmer to exploit it intuitively. This paper proposes an algorithm applying the spanning tree that mechanically exploits the parallelism and determines an execution order. To verify the algorithm, this paper performs evaluation with realistic applications. The algorithm exploits effectively the parallelism and construct the optimal pipeline processing flow.

Keywords: High Performance Computing, Stream Computing, GPUs, Caravela, Spanning Tree Algorithm

1 Introduction

Corresponding to a multicore/manycore approach, the supercomputing technologies are going to a new era that computer scientists focus on exploiting the thread level parallelism from applications [12]. Especially the manycore technology currently relying on the drastic growth of the graphics processing unit (GPU) such as the NVIDIA K20 integrates about 2500 processing units into an LSI, and also achieves Tera-flops class performance. Therefore, as we can see such accelerators in fastest supercomputers in the world, it is a mandatory choice of such accelerators for the computing node to achieve high performance [17].

Many runtime environments for the accelerators are proposed such as CUDA [28] and OpenCL [25]. These programming interfaces provide a programming language of the accelerator's *kernel program*, and runtime functions for controlling the execution of the kernel programs and the I/O data input/output by the program. While CUDA is dedicated for the NVIDIA GPUs, OpenCL provides a common interface to heterogeneous accelerators such as multicore GPUs, CPUs [14] and FPGA [1]. A programmer must select one of the runtime environments to develop an application. We have pointed out inconveniences of the code migration among different types of accelerators, and thus have proposed a unified interface called Caravela platform

that executes *flow-model*. It packs the kernel program and the I/O information into a flow-model [39]. A programmer on Caravela platform is able to develop a kernel program without any care for the lower runtimes of the accelerator, and just focuses on developing the flow-model by packing the kernel program, and specifying the I/O relations of the arguments of the kernel program and the target runtime type. Following the processing steps and using the *Caravela library* functions on the host CPU side, Caravela automatically chooses the available lower runtime in the system and invokes the kernel program embedded in a flow-model.

Although Caravela provides a unified interface for any accelerator, it still has the difficulty for a programmer who unavoidably needs the double programming on CPU and the accelerator. To eliminate the difficulty, we have proposed a commandline-based programming tool for the accelerator applying the Caravela framework called *CarSh* [41]. CarSh provides a conventional shell-like interface that accepts an *executable* XML file. The file includes an execution model of a flow-model and also accepts a *batch* XML file with the multiple executables described in the order of the target application. It is organized with buffer creation commands and the repeat execution command, etc. Programmers is finally released from the double programming situation and just becomes possible to focus on developing the flow-models and the executables or the batches of CarSh.

Using the CarSh framework, we have implemented a GUI as an Eclipse plugin to develop applications targeted to accelerators. The GUI provides the interface for defining flow-models and the connections via the I/O data, and then implements a processing pipeline. From the pipeline flow definition with multiple flow-models and the I/O connections, the CarSh executable can be mechanically generated. However we have found that it is not easy to generate the CarSh batch file automatically when it includes an execution order that is not deterministic. For example it includes feedback I/Os. If the flow is simple, a programmer can imagine some execution order and the concurrency among flow-models immediately. However, when the processing flow becomes large or complex, it is hard to determine the best order to execute the processing flow in a pipeline manner. Therefore it is indispensable to invent an algorithm to decide the processing orders and to find the parallelism mechanically from the processing pipeline.

This paper proposes a novel algorithm named Parallelization Extraction Algorithm with Spanning Tree (PEA-ST) to exploit the processing order and the parallelism systematically from a static execution flow organized by multiple flow-models. We will show the theory of the PEA-ST that generates the CarSh batch.

This paper begin to describe the detailed backgrounds and the definitions in the next section. Section 3 describe design and implementation of PEA-ST. Section 4 shows the performance evaluation applying the PEA-ST to a realistic application. Section 5 will show the related works and compares ours work with the advanced research results. Finally section 6 concludes the paper and will show the future work.

2 Research Backgrounds

2.1 Stream computing on manycore architecture

Multicore/manycore architectures are promoted because of the saturation of the Moore's law related to the growth of transistor count per silicon platform. The manycore architecture has the origin from the graphics processing demands that needs a fast computation to achieve a high frequency framing on a dynamic graphics especially in entertainment market [26]. It requires the concurrent processing for multiple pixels computed by the hundreds of processing units. It also exploits the fine-grained parallelism from application program potentially [30]. Thus, GPU has become one of indispensable tools to implement a high performance computing platform.

In the manycore architecture, each processing unit identifies its target computing element regarding a processor index. For example, assuming a vector summation is $r_c = r_a + r_b$, a programmer needs to consider that the calculation is separated into each element of vectors like $r_c[id] = r_a[id] + r_b[id]$, where the id is the index of the vector element. Each calculation for $r_c[id]$ is assigned to a computing unit, then the summations of elements in the vector are performed in parallel. Optimistically, the vector summation needs only the processing time to calculate the "+" operation when the number of computing units is larger than the length of r_c. Thus, a programmer needs certainly needs to consider the indexing of computing elements and also independent computation for each computing element assigned to a unit. Because of the processing style based on the indexed processors, it is called *stream computing*.

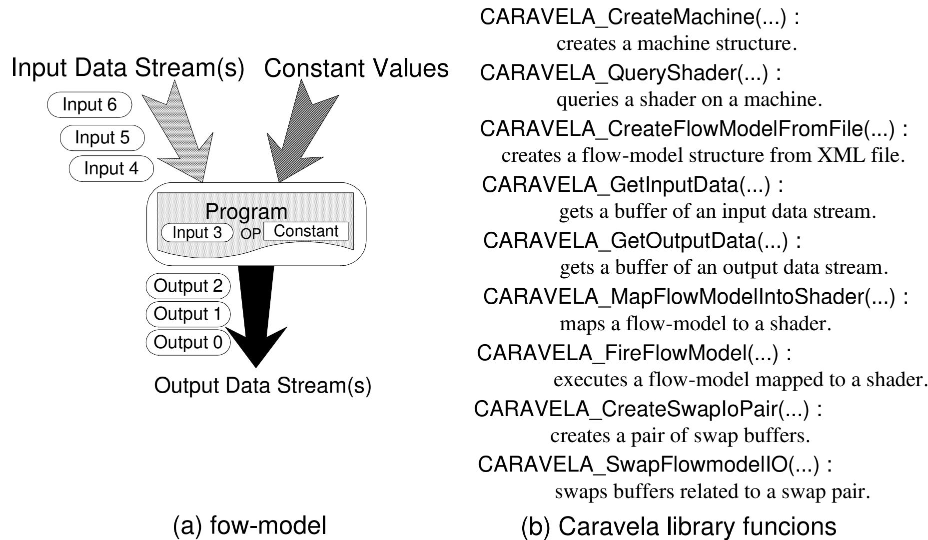

Figure 1: Caravela platform (a) flow-model that is executed on an accelerator using (b) Caravela library from the host CPU.

An accelerator in the manycore architecture works as a co-processor of a host CPU connected via the peripheral bus. Therefore, for programming on the accelerator, the host CPU is indispensable and controls the configuration and the behavior of the accelerator. Thus, the programmer must write both computing programs on the accelerator side and the controlling program on the host CPU side unavoidably. To reduce the difficulty of the double programming situation, there exist programming languages and the runtimes. For example, the recent standard ones are NVIDIA's CUDA [28] and OpenCL [25].

2.2 Caravela Platform

To standardize the programming interfaces of the stream computing, *Caravela platform* has been proposed and developed by the author of this paper [39]. It works like a wrapper interface of the stream computing runtimes. Currently, Caravela supports both CUDA and OpenCL for the lower level runtime.

The Caravela platform uses a concept of *flow-model* for programming a given task. Applications are written on this platform by using the *Caravela library*, which maps flow-models into the available accelerators. As shown in Figure 2(a), the flow-model is composed of 1-dimensional input/output data streams, constant input parameters and a program which processes the input data streams and generates the output data streams. The methods to execute a given task in a flow-model can be encapsulated into an XML file listed in Figure 1. Properties of the I/O data streams and the kernel program can be specified by tags in an XML file because those are stored in a text format. Actually the kernel function is written in the language specified by a runtime type tag in the XML,and is executed by the accelerator via the specified lower level runtime.

The Caravela library has the resource hierarchy definition stratified by *Machine* that is a host machine of a peripheral bus adapter, the *Adapter* is a peripheral bus adapter that includes one or multiple accelerators and finally the *Shader* is an accelerator. Figure 1(b) shows the basic Caravela functions for executing a given flow-model. Using those functions, a programmer can easily implement an application using the flow-model framework, just by mapping flow-models into shaders.

Figure 2 shows an example of flow-model that invokes vector summation over OpenCL runtime with 1024 compute units. Although a programming language used in writing a kernel program is different from the lower level runtime such as CUDA, the execution style in the accelerator is standardized by the flow-model execution framework unified by the Caravela library.

Although Caravela absorbs difference of the lower runtimes depending on the target accelerator, a programmer who develops an application using an accelerator unavoidably needs the double programming on the host CPU side and the accelerator side. Moreover the programming styles on both sides are different

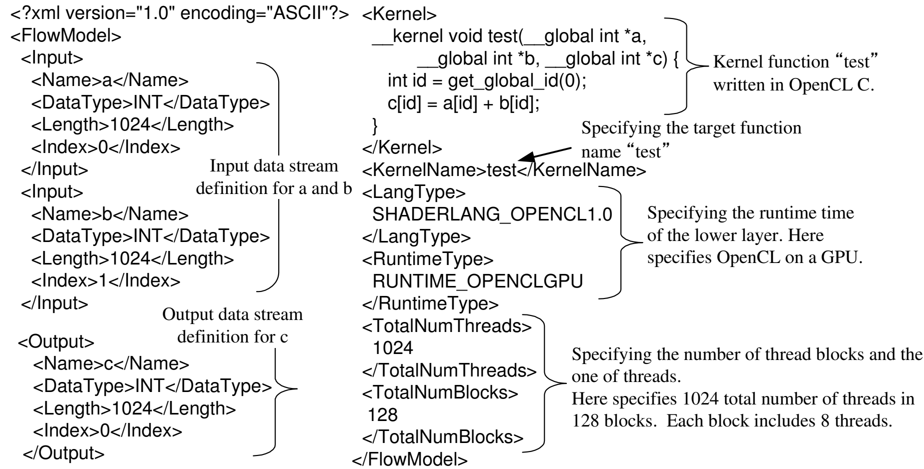

Figure 2: A flow-model example that defines a vector summation targeted to OpenCL on GPU.

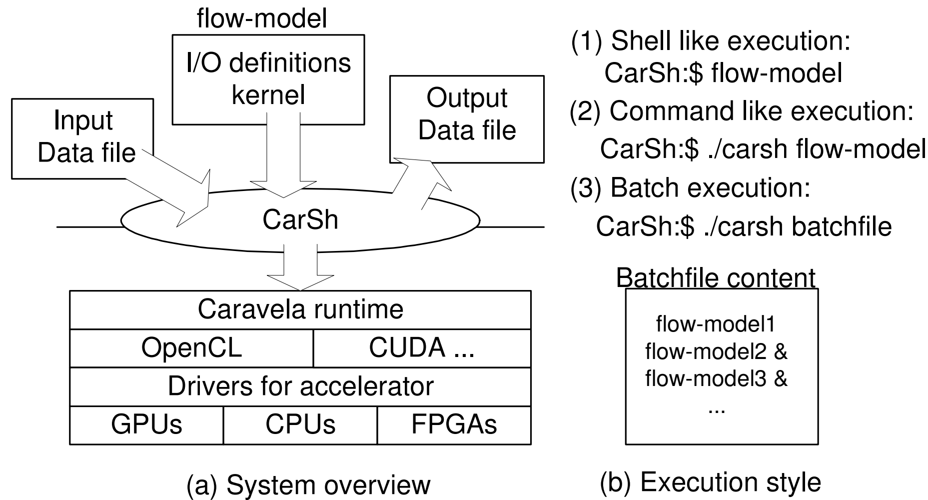

Figure 3: (a) CarSh system overview and (b) the commandline interface for (1) shell-like, (2) command-like and (3) batch executions.

although the programmer wants to write programs executed on the accelerator side only. We need to address this difficulty by an innovative programming interface for the manycore accelerators.

2.3 Commandline-based Stream Computing

We have developed a tool for commandline-based stream computing called *CarSh* [41]. It overcomes the double programming difficulty of the conventional runtimes for accelerators. When a programmer uses CarSh, one never needs to make a control program on the host CPU side. CarSh receives an *executable* file. CarSh reads/writes the input/output data from/to the CSV files according to the I/O data assignments in the executable. The executable file packs the flow-model of the Caravela framework and the I/O definitions. The definition links the arguments in the kernel program into the I/Os. The linked I/O information is defined in the executable as illustrated in Figure 3(a). The CarSh extracts the flow-model and the I/O linking information from the executable file. CarSh passes those to the Caravela runtime library to execute the flow-model. The Caravela library executes the flow-model by selecting the best fit runtime of the target accelerator. To accept the executable file, CarSh provides a shell-like interface as illustrated in Figure 3(b). It accepts an executable files and an *batch* file. The batch file includes a scenario with the executables and commands regarding I/O

```
<?xml version="1.0" encoding ="ASCII"?>
<CarshEx>
 <ModelFile>Flow-model XML file name</ModelFile>
 <Input>
  <Name>Input valuable name in kernel function</Name>
  <DataFile>CSV file name</DataFile>
 </Input>
 <Input>
  ....
 </Input>
 <Output>
  <Name>Output valuable name in kernel function</Name>
  <DataFile>CSV file name</DataFile>
 </Output>
 <SwapPair>
  <Input>Input valuable name in kernel function</Input>
  <Output>Output valuable name in kernel function</Output>
  <NumSwap>Number of swaps</NumSwap>
 </SwapPair>
</CarshEx>
```

(a) CarSh executable format in XML

```
<?xml version="1.0" encoding ="ASCII"?>
<CarshBat>
virtbuf create int 1024 buf_a
virtbuf create int 1024 buf_b
virtbuf fill sample 1_a.csv buf_a
virtbuf fill sample 1_b.csv buf_b
virtbuf create int 1024 buf_c
sample_virt &
sync
repeat 10 sample_virt.xml
sync
virtbuf dump c _tmp.csv buf_c
virtbuf swap buf_a buf_c
virtbuf delete buf _a
virtbuf delete buf _b
virtbuf delete buf _c
exit
</CarshBat>
```

(b) CarSh batch format in XML

Figure 4: The XML definitions of (a) CarSh executable and (b) batch files.

buffers. Here CarSh supports the background execution of the executable file (i.e. the background execution of a flow-model) or the batch file by appending "&" in the last of the commandline of CarSh.

The XML description of the executable is listed in Figure 4(a). The I/O data is mapped by the CSV file. Moreover, the *virtual buffers* can be assigned to the definition. The virtual buffers are allocated by the commands in the batch. The virtual buffer saves the intermediate output data and can pass the buffer name to the next executable. We can write the batch file as depicted in Figure 4(b).

CarSh provides the control commands for flow-model execution. CarSh supports *ps* and *kill* commands for process management. The *virtubuf* command supports the virtual buffer management interface. The command *sync* synchronizes all executables or batches invoked before the command. *repeat* command repeats an executable or a batch for specified number of times. Using these commands in an executable or a batch file, a programmer focuses on only making a processing order of a pipelined processing flow.

2.4 GUI-based stream-based computing

We have developed a GUI of Caravela platform as shown in Figure 5. The programmer just defines the flow-models and connection among those flow-models graphically. It generates executables and batches needed for CarSh. The GUI is very helpful for automatic programming for accelerators. However, it is very hard to generate an execution scenario for the processing pipeline. For example, given a processing pipeline as depicted in Figure 6 (a), it is easy for us to identify the execution order. While the input data for *flowmodel1* is given successively, the overall calculation is invoked in the order from *flowmodel1* to *flowmodel4*.

When we consider the concurrency of the execution of multiple flow-models, it is also conceivable for us by writing the execution order with the considerations: 1) The *flowmodel1* and *flowmodel2* are not executable in parallel because the connected I/O are interfered with each other because it is assigned to the same physical buffer. 2) The *flowmodel2* must be executed after the *flowmodel1* because the input data must be prepared (we call this *initialized*) before the flow-model execution. Such as the right side of Figure 6 (a), after the executions of *flowmodel1* and *flowmodel2*, we can think intuitively that the combinations of *flowmodel2 & 4* and *flowmodel1 & 3* are executed concurrently.

However, given the processing flow of Figure 6 (b), how do you specify which is the flow-model initially executed when the first input data is given to *flowmodel1*? How is the parallelism if multiple flow-models could be invoked concurrently? It is impossible to implement the perfect GUI-based programming method before solving these problems. As an objective of this paper, we propose an algorithm that mechanically exploits the deterministic execution order and the parallelism from any kind of pipeline execution flow.

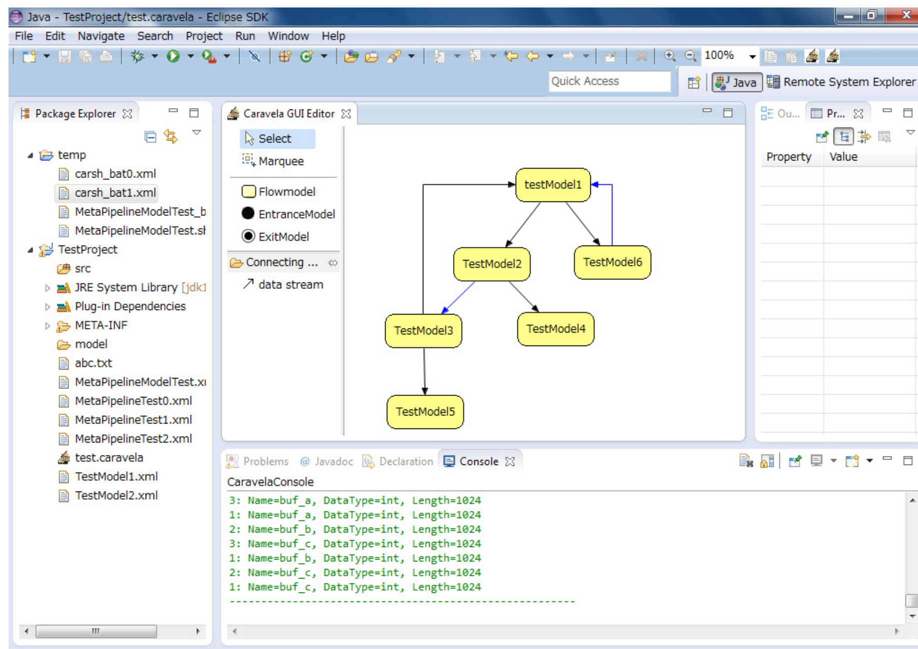

Figure 5: The Caravela GUI implemented as an Eclipse plugin that generates XML files of the flow-model, CarSh executables and batches.

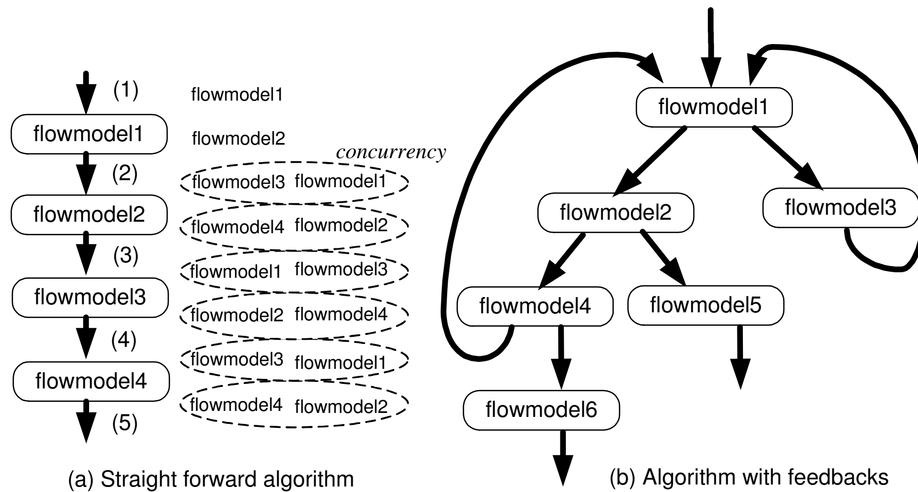

Figure 6: Pipeline flow examples of a) straight forward algorithm and b) algorithm with feedbacks.

Algorithm 1 Depth-first spanning tree (DFST) algorithm.

```
void DFST(int x){
  pre_num ++;
  NPre[x] = pre_num;
  for(int y=0; y<N; y++){
    if(edges[x][y].connect){
      if(NPre[y] == 0){
        //x → y is a tree edge
        DFST(y);
      }
      else if(Npre[x] < Npre[y]) { // x→y is an advancing edge }
      else if(NRPost[x] == 0){  //x→ y is an retreating edge }
      else { // x →  y is an cross edge }
    }
  }
  NRPost[x] = rpost_num;
  rpost_num --;
}
```

2.5 Spanning Tree

Challenges for execution ordering and exploiting parallelism from a program are studied in the decades by researchers of compilers. The High Performance Fortran (HPF) is a well-known solution to exploit potential parallelism from a numerical program description. *Spanning Tree* introduced in [38] is a technique to determine the control flow of a program description.

Given a directed graph $G(V,E)$, where V is a set of vertices (nodes) of G, and E is the set of edges of G, $S(V,T)$ is called the $G(V,E)$'s *spanning tree*, when a subset of edges T satisfies $T \subseteq E$ and the graph $S(V,T)$ forms a tree. Here, $S(V,T)$ does not include loops. When spanning tree is defined, the edges in G is categorized into four types: 1) *Tree edges* form the spanning tree. 2) *Advancing edges* are a set of edges of $X \rightarrow Y$ that are not the tree edges. Y is a descendant of X. The edges jump to vertices in the lower structure of the tree. 3) *Retreating edges* are also a set of edges of $X \rightarrow Y$ that are not the tree edges. Y is an ancestor of X. That is, the edges jump to vertices in the upper structure of the tree. Finally, 4) *Cross edges* are the rest of the edges which are not belong to 1) - 3). Those are a set of edges of $X \rightarrow Y$, where Y is neither ancestor nor a descendant of X. Here, there exists a condition regarding the root vertex of the spanning tree. From a root vertex, the graph must have a reachable path to all other vertices. If the path does not include all vertices of G, it does not have a spanning tree. However, note that a spanning tree generated from a node is uniquely found in G. Therefore, a different root constructs a different spanning tree from the same graph.

Algorithm 1 lists the steps to categorize the edges of a directed graph into four categories above, when a root vertex is given to the `DFST` function. This algorithm is developed based on the Tarjan's depth-first search algorithm [36]. In this meaning, we call it *Depth-First Spanning Tree* (DFST) algorithm. Regarding the tree edges generated by the algorithm, we can find the spanning tree.

The algorithm is a combination of the preorder numbering $NPre()$ and the reverse post numbering $NRPost()$. In the former numbering, if $NPre(X) < NPre(Y)$, X is either a preorder ancestor of Y in the tree, or the left of Y. In the latter numbering, if $NRPost(X) < NRPost(Y)$, X is either a preorder ancestor of Y in the tree, or the right of Y. First, in the step performing the preorder numbering , the DFST is checking the preorder number of the next connected node after the current node. If it is zero, it is detected as a tree edge. When the search reaches a leaf of the tree, it performs the reverse post numbering, returning to the tree edges. In the backward searching, it marks one of retreating, advancing and cross edges.

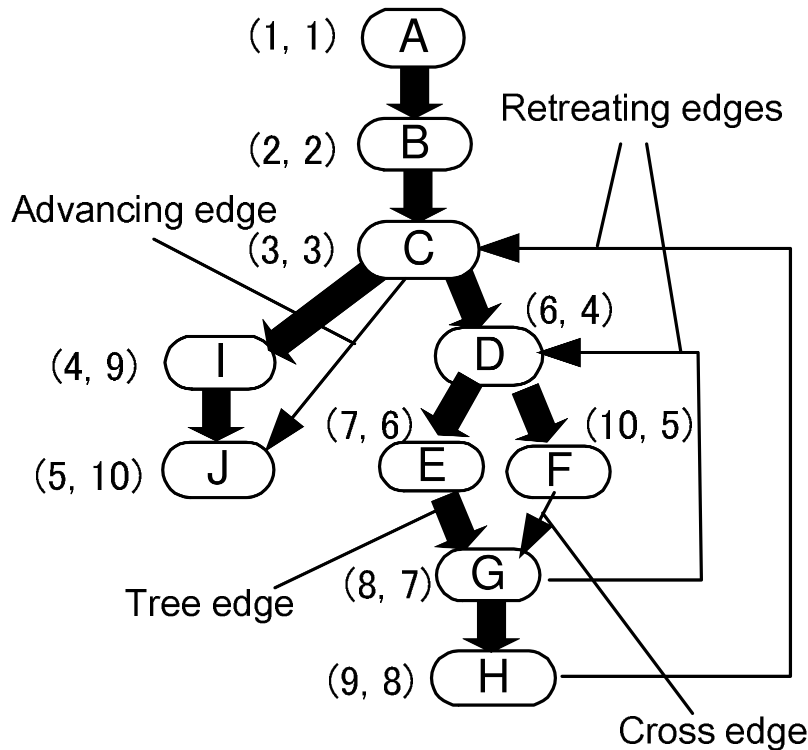

Figure 7: Spanning tree examples.

Figure 7 shows an example of a spanning tree generated by the DFST. The pairs of numbers in the figure are $(NPre, NRPost)$. We select A as the root vertex. First the preorder numbering is performed in the order of $A \rightarrow B \rightarrow C \rightarrow I \rightarrow J \rightarrow D \rightarrow E \rightarrow G \rightarrow H \rightarrow F$. During the numbering of the backward searching, the reverse post numbering is performed in the order of $J \rightarrow I \rightarrow H \rightarrow G \rightarrow E \rightarrow F \rightarrow D \rightarrow C \rightarrow B \rightarrow A$. In the former step, the tree edges are found like the path marked with the thick arrows. The retreating edges and the crossing edges are also found during the backward search.

2.6 Discussion

Spanning tree method is also applied in the communication network field. The Spanning Tree Protocol (STP) is a network protocol that ensures a loop-free topology for any bridged Ethernet-based local area network. The basic function of STP is to prevent bridge loops and the broadcast radiation that results from them. The spanning tree also allows a network design to include spare (redundant) links to provide automatic backup paths if an active link fails, without the danger of bridge loops, or the need for manual enabling/disabling of these backup links. Spanning Tree Protocol (STP) is standardized as IEEE 802.1D. As the name suggests, it creates a spanning tree within a network of connected layer-2 bridges (typically Ethernet switches) and disables those links that are not part of the spanning tree, leaving a single active path between any two network nodes [31]. Perlman applied STP to a routing algorithm in a communication path, considering the network connections among switches and communication nodes as edges and nodes of a directed graph [32]. The algorithm finds the shortest network path with the smallest number of links that eliminates loops.

To break loops in the LAN while maintaining access to all LAN segments, the program in each bridge that allows it to determine how to use the protocol is known as the spanning tree algorithm. The algorithm is specifically constructed to avoid bridge loops (multiple paths linking one segment to another, resulting in an infinite loop situation). The algorithm is responsible for a bridge using only the most efficient path when faced with multiple paths. If the best path fails, the algorithm recalculates the network and finds the next best route.

The Dijkstra's algorithm [7] is also another well-known graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree [23]. For a given source node in the graph, the algorithm finds the shortest path and the reaching cost between a vertex and another vertex. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, the Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path first is widely used in network routing protocols, most notably IS-IS [13] and OSPF (Open Shortest Path First) [24].

In the Dijkstra's algorithm, the node at which it starts is called the initial node, and the distance of node Y is the one from the initial node to the node Y. The algorithm will assign some initial distance values and will try to improve them step by step. The steps of the algorithm are summarized below; 1) assigning every node a temporary distance value, 2) marking all nodes unvisited, for the current node, 3) considering all of its unvisited neighbors and calculating their temporary distances, 4) marking the current node as visited and removing it from the unvisited set, and 5) finishing the algorithm if the destination node has been marked as visited or if the smallest temporary distance among the nodes in the unvisited set is infinity, 6) selecting the unvisited node that is marked with the smallest temporary distance, and setting it as the new current node then go back to step 3).

The Dijkstra's algorithm is usually the working principle behind link-state routing protocols, OSPF and IS-IS being the most common ones. The process that underlies the Dijkstra's algorithm is similar to the greedy process used in Prim's algorithm [33]. Prim's purpose is to find a minimum spanning tree [11] that connects all nodes in the graph. Meanwhile, the Dijkstra's is concerned only between two nodes. The Prim's does not evaluate the total weight of the path from the starting node, only the individual path.

The spanning tree algorithm is to discover a loop-free subset of the topology dynamically. For example, if a network topology does not change, the network is not partitioned temporarily while nodes switch over to other routes. If the backup root is very near the old root, the topology will not change significantly even when the old root dies. On the other hand, in the case of the Dijkstra's algorithm, when any cost is changed, it inevitably needs to recalculate it. Therefore, the search result of Dijkstra's algorithm is not adaptive to the dynamic route path.

According to the discussion above, the spanning tree algorithm is powerful for finding a route path between the nodes in a tree. When we consider a pipeline with processing tasks (i.e. flow-models) connected by the I/O data streams, it can be treated as a tree. Therefore, the spanning tree algorithm can be applied to define a unique processing order. Additionally it also finds the feedback I/Os. In the spanning tree, the nodes with the same depth from the root node do not have edges among them. This means that those nodes (flow-models) can be independently executed. The groups of the nodes induce a definition of stages in the processing pipeline because we can define an execution order of the groups. Thus, the spanning tree algorithm can define an effective pipeline order with all tasks included in the processing flow. This paper will focus on the characteristics of the pipeline flow exploited by the spanning tree algorithm and proposes a novel algorithm using the spanning tree that extracts the best parallelism.

3 Parallelism extraction algorithm with spanning tree

In order to address the execution order and finds the concurrency, we need to develop an algorithm for 1) finding a flow-model executed first, 2) finding a deterministic execution order without I/O buffer collisions and 3) exploiting an available concurrency from the processing flow. We also define the CarSh batch scenario from the algorithm. Let us map the processing flow to a directed graph. The flow-models correspond to the nodes. The I/O data streams correspond to the edges.

3.1 Finding the first execution flow-model

First, we define an *executable node* and a *root node*. When all edges point to a node that is given, we call the node *executable*. The edges that come out from the node are *initialized* after the execution of the node. Here, the *initialization* means that all of the input data streams entering into the node are ready. The *root node* is defined as an executable node that all input edges into it are initialized before execution of any node in a graph (i.e. at the initial status).

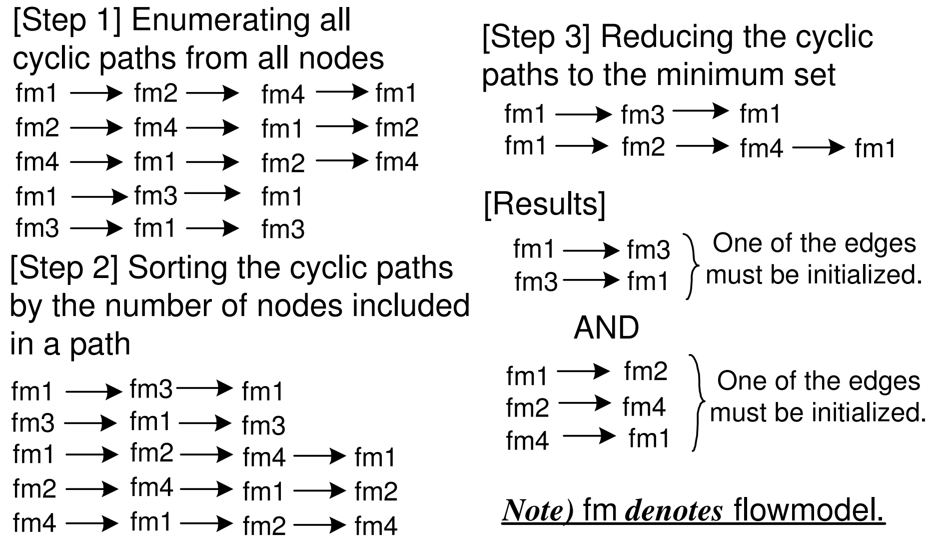

Figure 8: An example of finding a set of cyclic paths and the edges that needs initialization to avoid deadlock applying the algorithm from [40] to the graph of Figure 6(b).

When a graph includes any retreated edge such as the one from *flowmodel4* to *flowmodel1* as depicted in Figure 6(b), the pipeline execution of the graph absolutely deadlocks. In this case one of the edges in the *cyclic path* that is *flowmodel1→flowmodel2→flowmodel4→flowmodel1* must be initialized. The algorithm to find the minimal set of the cyclic paths in a directed graph has been defined already in the reference [40] by the author of this paper. The steps to make a set of the minimum cyclic paths are summarized below:

1. Enumerating all cyclic paths from all nodes
   For example, in Figure 6(b), all cyclic paths from all nodes are enumerated (Figure 8[Step 1]).
2. Sorting the cyclic paths by the number of nodes included in a path
   Comparing the number of nodes included in each cyclic path, it sorts the set of all paths (Figure 8[Step 2]).
3. Reducing the cyclic paths to the minimum set
   During comparisons of the paths from the one with the smallest number of nodes to the one with largest number of nodes, if the path compared is the same as the current one, the former is removed. Then the remaining set of paths is the minimum cyclic path. Regarding all paths in the set, one of the edges of each path must be initialized to avoid deadlock of execution (Figure 8[Results]).

After all cyclic paths are found and any edge of any cyclic path is initialized, one or more root nodes exist in the directed graph. The graph with any root node is called an *executable directed graph*.

An executable directed graph must have executable node(s). Those nodes correspond to root nodes of the graph. Now let us to explain the processing pipeline with flow-models. The root node(s) is equivalent to the flow-models executed at the beginning of the pipeline execution. According to this initialization strategy, we can find the first executable flow-model in the processing pipeline.

Now we can define the root node in a directed graph using the algorithm explained above. Selecting one of the nodes in the minimum cyclic paths as the root node, we can find a spanning tree. It includes a unique path of the tree when the execution steps follow the tree edges from the root node to downward. This path becomes the execution flow of the directed graph of the flow-models.

3.2 Extracting parallelism and determining execution order

The number of nodes in a cyclic path is not only one in general. Any node in a minimum cyclic path can be the root node of the spanning tree. Therefore, it is available for all nodes to be selected as the root node in the spanning tree. Here, we assume that the processing flow builds a strongly connected directed graph.

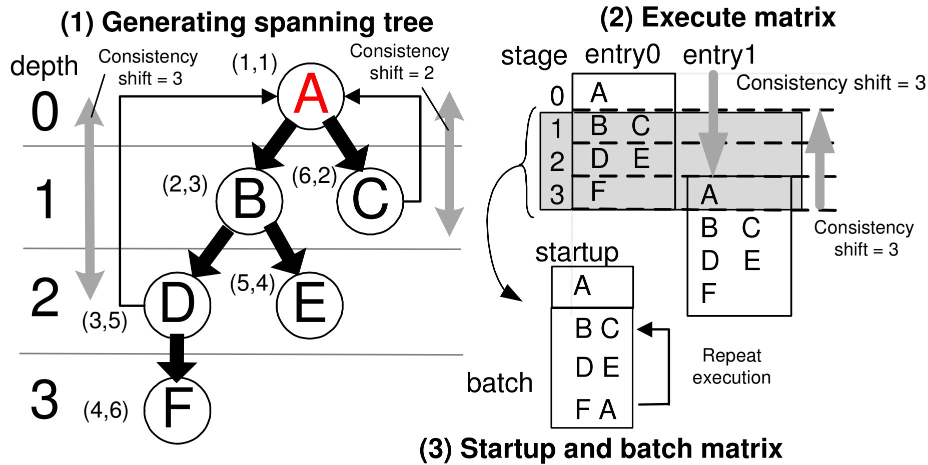

Figure 9: Processing steps of the PEA-ST.

The goal of this paper is to exploit parallelism of the processing flow. With the parallelism, we perform a processing order of concurrent execution of multiple flow-models in a pipeline manner by using the parallelism extraction algorithm with spanning tree (PEA-ST). Here, let us consider the case of Figure 9. We assume that the edges $D \to A$ and $C \to A$ are initialized. Then we select A as the root node. The spanning tree of the graph becomes like Figure 9(1). The initialized edges are categorized as the retreating edges. Others are the tree edges.

To exploit the parallelism from the spanning tree, we define *depth* and *stage* of a processing pipeline. The depth of a node is defined as the number of tree edges from the node to the root node.The stage is defined as a set of nodes (kernel programs of flow-models) which can be invoked concurrently without the I/O buffer conflicts. Let us generate the depths and stages of the spanning tree in Figure 9. Here, we define that a tree edge has a single depth of the pipeline. For instance, the depth of E is 2 because there are two tree edges in the path of $A \to B \to E$. As another example, the depth of F is 3 because there are three tree edges in the path of $A \to B \to D \to F$. Thus, the total depth of the example graph becomes four.

According to the definitions of the stage and the depth, it is obviously to find that the nodes with the same depth can be added to the same stage. For example, the nodes B and C or the ones D and E can be invoked concurrently because there is no I/O conflict among those nodes in the same stage. Therefore, we can build the pipeline as $A \to (B, C) \to (D, E) \to F$. Shifting a stage, we can organize a pipeline like $A \to (B, C, A) \to (D, E, B, C) \to (F, D, E) \to (A, F) \to (B, C, A)$...

In the explanation above, we ignore the retreating edges. Here, we consider the effect of the retreating edges (i.e. loops) in the graph. If we completely ignore a retreating edge, the pipeline works incorrectly due to I/O conflicts caused by the loops. For example, the stage (B, C, A) can not be executed correctly because A must be invoked after C. Moreover, A must be invoked after D in the next execution. To resolve the problems we define *consistency shift*. First we find a retreating edge. In the loop where the two nodes connected by that retreating edge, we calculate the number of nodes connected by tree edges. In the example case, the edge $D \to A$ includes three nodes (i.e. A, B and D). The edge $C \to A$ includes two nodes (i.e. A and C). The consistency shift is that number we just calculated. For example, if the consistency shift is two, the pipeline becomes $A \to (B, C) \to (D, E, A) \to (F, B, C) \to (A, D, E)$... by shifting two stages. However A be invoked before D which causes I/O conflict. Therefore, it must be maximized. Taking the largest number of consistency shift among all retreating edges. Thus, if the stages are shifted by the max consistency shift, all nodes related to retreating edges are invoked correctly. The correct pipeline should become $A \to (B, C) \to (D, E) \to (F, A) \to (B, C) \to (D, E) \to (F, A)$...

Although the condition that a graph has no retreating edge, we need to consider the consistency shift. For example, we consider a straightforward processing flow $A \to B \to C$. The pipeline would become $A \to (B, A) \to (C, B) \to (A, B)$... However, this is wrong because may occur I/O conflict. To resolve this case, we assume that nodes connected by an edge also have a retreating edge. Therefore, the default number

Algorithm 2 Parallelism extraction algorithm applying spanning tree (PEA-ST).

```
struct edge {
    bool connect;
} edges[N][N];

struct node {
  char name;
  int depth;
  bool update;
} nodes[N];

void DFST_Modified(int x){
  pre_num ++;
  NPre[x] = pre_num;
  for(int y=0; y<N; y++){
    if(edges[x][y].connect){          <- Tree edge
      if(NPre[y] == 0){
        nodes[y].depth=nodes[x].depth+1;
        DFST_Modified(y);
      }
      else if(NPre[x] < NPre[y])      <- Advancing edge
        edges[x][y].connect = false;
      else if(NRPost[x] == 0){        <- Retreating edge
        If(max_retreat_offset<nodes[x].depth -  nodes[y].depth)
          max_retreat_offset=nodes[x].depth - nodes[y].depth +1;
        edges[x][y].connect = false;
      }
      else  Edges[x][y].connect = false;   <- Cross edge
    }
  }
  NRPost[x] = rpost_num;
  rpost_num --;
}

void main(){
  for(int result=0; result<N; result++){
    node_init();
    edge_init();
    root_test(result);
    if(node can reach all other nodes){
      node[result].depth=0;
       DFST_Modified(result);
    }
    else  continue;
    if(there is no retreating edges)
      consistency_shift=2;
    else
      consistency_shift = max_retreat_offset;
    for(i=0;i*move<max_stage;i++){
      for(int j=0; j<N; j++){
        stage= nodes[j].depth+i*move;
        if(stage < max_stage){
          excute[stage][count[stage]] = nodes[j].name;
          count[stage]++;
        }
      }
    }
    if(Parallelism is better than the previous one)
      update(startup and batch);
    else continue;
  }
}
```

of the consistency shift must be two.

In the correct pipeline, we can find two parts. One is the initial stage(s) executed once called *startup*. The other is the contiguous repeating stage(s) called *repeat batch*. In the example case, the stage of (A) is the former. In the case of Figure 9, a set of stages of $(B, C) \rightarrow (D, E) \rightarrow (F, A)$ is the latter. The startup stages include the beginning ones of the pipeline after the consistency shift is performed. Therefore, the number of stages of the startup is calculated by $depth - max_consistency_shift$. In the example case, it is $4 - 3 = 1$. Therefore the first stage with A is included in the startup. Other three stages organize the repeat batch. Thus, while the correct pipeline is configured, the startup is invoked once. Then the repeat batch is repeated.

Regarding the cross edges, PEA-ST ignores them because the edge can be treated as tree edge. Cross edge has two or more ancestors. An edge from one of the ancestors should become tree edge.

Let us summarize the processing steps of the PEA-ST explained above. First, one of the root nodes is selected. This makes a graph executable. Second, a spanning tree is created from the root node. Only a tree is generated. Third, the consistency shift is calculated from the retreating edges. Finally, the startup and the repeat batch are generated.

3.3 Implementation

Algorithm 2 shows the PEA-ST written by a C-like code. The *main* function processes 1) spanning tree creation, 2) consistency shift calculation, 3) startup and batch creation and finally 4) updating the startup and the repeat batch according to the user-defined requirements. The function checks all available spanning trees by selecting available root nodes. The process 4) will break the loop of selecting the root nodes and return the best startup and repeat batch.

The spanning tree creation is performed by *DFST_Modified* function. In the function all edges are categorized to the ones defined by the spanning tree. When an edge is categorized, it updates the *edges* matrix that obtains the tree edge. If an edge is categorized as a retreating edge, it updates the *max_retreat_offset* that obtains the largest offset of stages in the retreating edge (i.e. consistency shift).

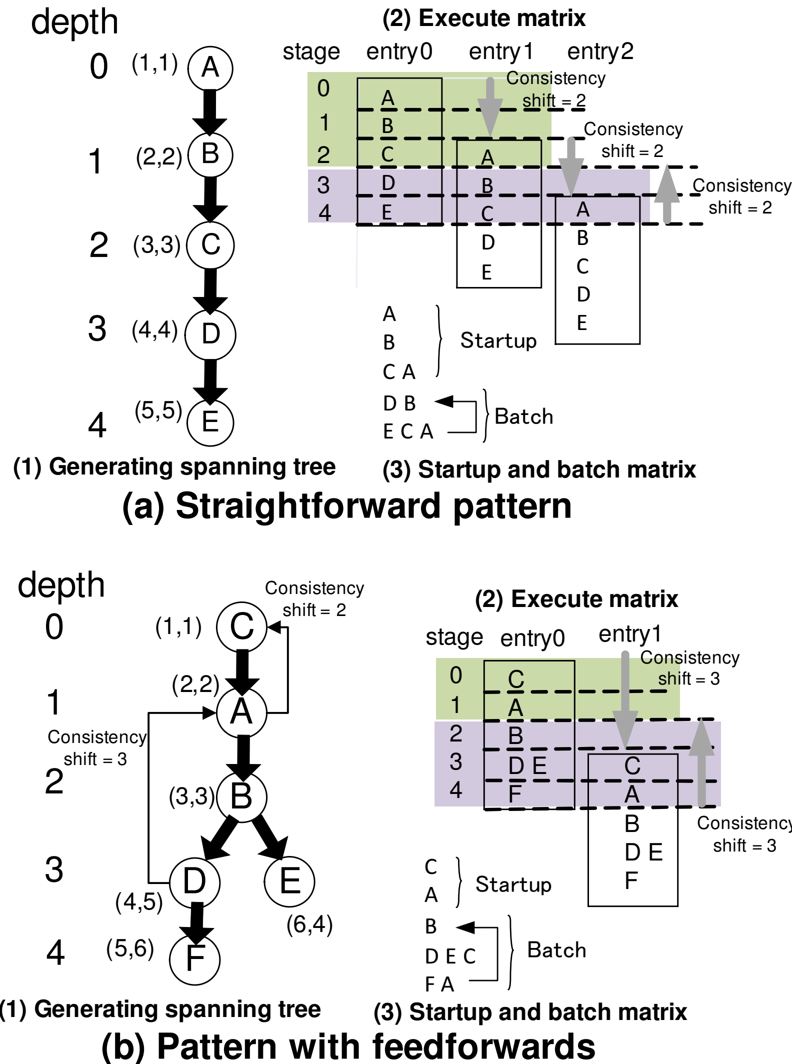

Figure 10: Examples of batch generation using the PEA-ST. A straight forward pattern (a) and the pattern with feedforward edges of Figure 9 when C is selected to the root vertex(b).

After the spanning tree creation, the *main* function calculates the consistency shift. If any consistency shift exists, the *max_retreat_offset* is used. If not, it is two because of the case of straightforward processing flow. Then finally the execute matrix is created. It obtains the stages and the node combinations at each stage. Using the example of Figure 9, let us explain how to calculate the startup and the repeat batch. At the first iteration of *execute* matrix creation, it fills each stage with the nodes of the same depth illustrated by *entry0* in the figure. In the second iteration, it shifts the column index by the consistency shift and fills the nodes of stages again like *entry1*. If the shifted index is larger than the depth of the graph, the iteration ends. The final *execute* matrix is the startup and the repeat batch. The repeat batch is the last n rows of the matrix where n is the consistency shift. The remaining row(s) are the startup.

Finally, user-defined conditions are checked. The conditions depend on the maximum/minimum parallelism, the average parallelism, the number of stages and the smallest consistency shift, etc. In our implementation, we apply the condition to find the largest parallelism of $MIN_AVR < parallelism < MAX_AVR$ and $parallelism < MAX$, where MIN_AVR and MAX_AVR are the minimum and maximum parallelisms respectively, and where the MAX is the maximum parallelism. MAX is given by the programmer because of the limitation of the resources (the number of accelerators).

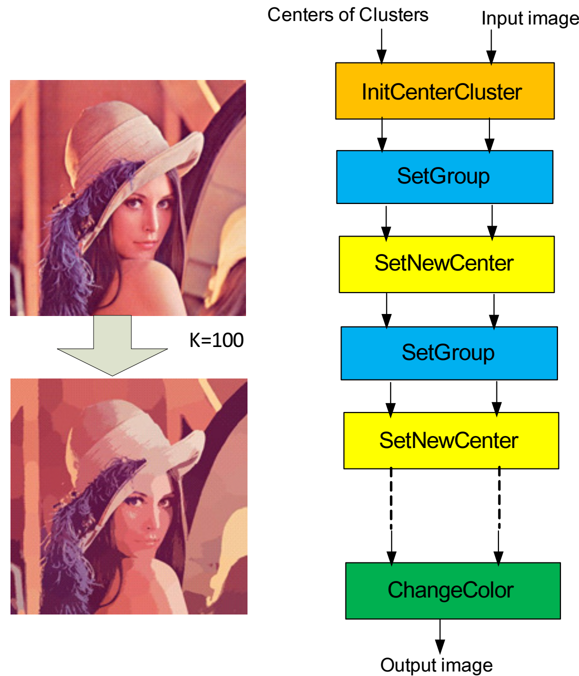

Figure 11: k-means example applying PEA-ST.

Regarding the complexity of the PEA-ST, it is similar to the spanning tree algorithm. The preorder and the reverse post order numberings takes $O(NE)$ time respectively, where N is the number of nodes and E is the number of edges in the graph. However, the PEA-ST needs to try all available spanning tree creations of the root nodes. Therefore, it becomes $O(NEM)$ where M is the number of available root nodes.

3.4 Examples

Here, let us introduce two additional examples as depicted in Figure 10. The straightforward pattern shown in Figure 10(a) needs three iterations to generate an *execute* matrix. The maximum parallelism becomes three. Figure 10(b) shows another example of the same processing flow as Figure 9. But the root node is C. This case shows that the maximum parallelism becomes three. It is two when we selected A as the root node. Thus, PEA-ST is very flexible to select an ideal parallelism defined by a programmer according to the limited number of accelerator resources.

4 Performance evaluation

To validate the PEA-ST, we have made two applications related to image processing. The first one shows the validity of the PEA-ST. The second one shows the impact on the performance aspect. Both applications have the potential ability of achieving small latency to output the final results when the PEA-ST is applied and the processing flows are modified to pipeline flows with parallel executions of multiple tasks (flow-models).

4.1 Color image quantization

The first example is a color image quantization. We will show an realistic example when the PEA-ST is applied to a processing flow graph. Color quantization is a task of reducing the color palette of an image to a fixed number of colors k. The k-means algorithm can easily be used for this task and produces competitive results. Figure 11 shows the processing flow. It consists of four kinds of flow-models. The `InitCenterCluster` performs the initialization of the clusters' centers which are randomly selected.

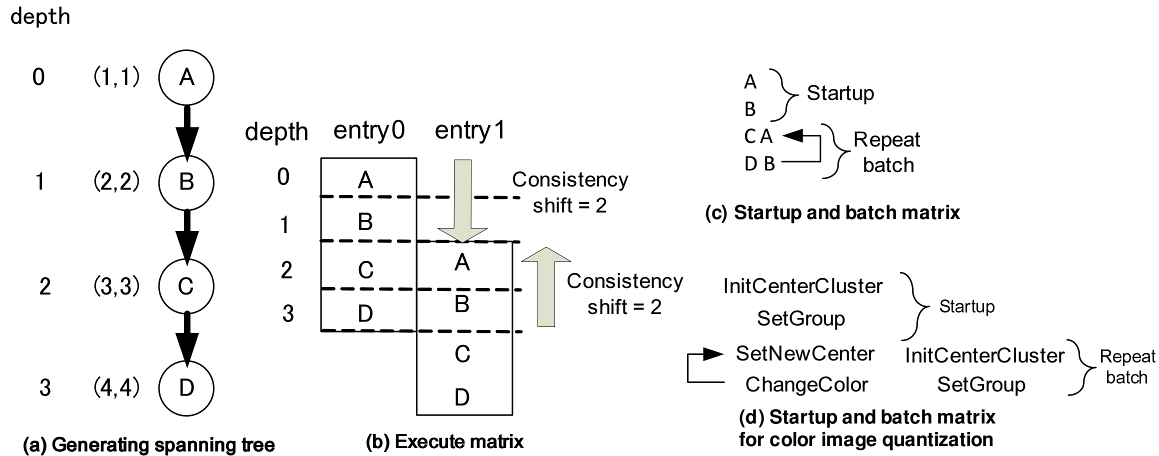

Figure 12: The startup and the repeat batch of the color image quantization application.

The `SetGroup` divides all pixels into k clusters. It calculates the distances from the pixels (RGB) to each center and assigns each pixel to the nearest cluster. The `SetNewCenter` calculates the new centers. Then it compares the old and the new center. If they are different, it continues iterating the loop of the processing steps with the `SetGroup` and the `SetNewCenter` flow-models. Contrarily, if they are the same, it stops the loop and executes the subsequent flow-models. The `ChangeColor` performs a transformation of the color (RGB) of the original image.

Figure 12 shows the pipelined processing flow generated by PEA-ST. Using the figure, let us explain the steps of how to generate pipelined processing flow of k-means example by PEA-ST in detail. Here, we can simplify the structure of the four kinds of flow-models in Figure 11 into the graph of Figure 12(a). Because only the node A can become the root node, only one spanning tree is derived. In other words, the k-means example has just one result of pipeline processing flow according to the PEA-ST. First, the spanning tree generated by the DFST Modified function is shown in Algorithm 2. In the resulting spanning tree, there exists a retreating edge $C \rightarrow B$.Also, we can see that the `max_retreat_offset` of this retreating edge is 2. The `main` function calculates the consistency shift, which is equal to the `max_retreat_offset`. Then the PEA-ST algorithm generates the *execute matrix* with consistency shift. There are four stages and two entries in the execute matrix in Figure 12(b). And at the first iteration of the creation process for the execute matrix, the algorithm assigns the corresponding offset incremented from 0 for the depth to each stage of the nodes from A to D illustrated by *entry0*. In the second iteration, it shifts the depth by the consistency shift of 2, and assigns the nodes of stages again from the shifted depth number as shown in *entry1*. Because the maximum depth is 3 (the original depth of the spanning tree is 3), the nodes is placed in the depth which is larger than the maximum depth. Therefore, the nodes C and D in the *entry1* is canceled. Finally, the startup and the repeat batch are generated from the execute matrix. As shown in Figure 12(c), the repeat batch is the last 2 rows of the matrix because of the consistency shift and it contains all the flow-models of k-means example. And the remaining first two rows are regarded as the startup. The final pipelined result of k-means example is shown in the Figure 12(d). The startup that contains the `SetGroup` and the `SetNewCenter` flow-models is executed once at first. Then the repeat batch is repeated.

In this example, the `ChangeColor` outputs the result regarding a single input image. Therefore, if the flow is executed in serial, each flow-model is executed by blocking after the execution because of the I/O data dependencies. Therefore, the total execution time takes the elapsed time of all four flow-models in the execution flow. However, after it is pipelined, the processing flow has been parallelized with two flow-models in a stage. If multiple accelerators are available, the execution time will be improved at most to the one of a single flow-model (`ChangeColor`). Therefore, the PEA-ST approach is effective modification of the processing flow achieving the consistency of the I/O data dependency. Let us see the performance impact on the optimization by the PEA-ST algorithm in the next example.

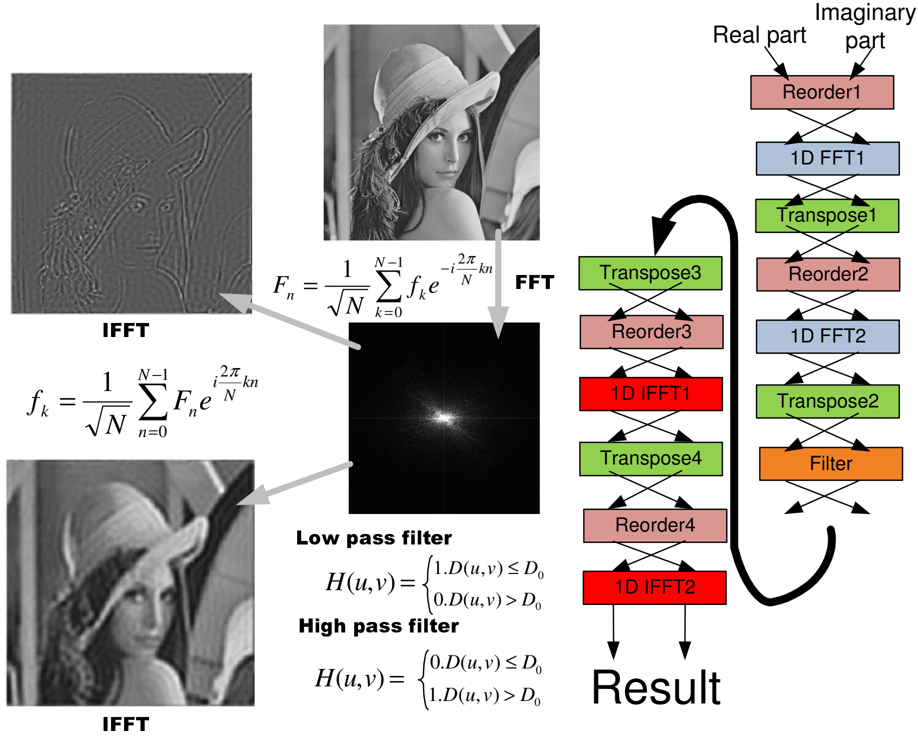

Figure 13: FFT example using PEA-ST.

4.2 2D FFT

To evaluate the performance of the PEA-ST, we compare between the serialized and the pipelined processing flows of a common image filtering. Figure 13 shows the processing flow. It consists of four kinds of flow-models. The reorder performs butterfly operation, the FFT and IFFT calculate 1-dimentional FFT and IFFT respectively. The transpose performs a transpose of the 2-dimentional matrix data. The filter is a high-pass or low-pass filter. A CarSh executable of the flow-model is downloaded to accelerator via Caravela runtime. And finally, it is executed by the OpenCL runtime. Figure 14 shows the pipelined processing flow generated by PEA-ST which is packed in two CarSh batches; one is the startup and another is the repeat batch. Totally 13 flow-models are executed in a pipeline manner. The maximum parallelism of the processing flow exploited by the PEA-ST becomes seven as shown in Figure 14.

This application also includes the potential performance improvement function after applying PEA-ST algorithm to the original flow with 13 flow-models. The final IFFT2 output the result of the input single image. Therefore, the original flow graph shows 13 steps to execute all flow-models and it need the latency to execute all the flow-models to calculate a single result. However, after pipelining, the latency to calculate the result from IFFT2 becomes small because IFFT2 is executed at every execution of a stage.

The execution time until the final IFFT is measured with/without parallelization. In the case of the serialized version, the 13 flow-models are executed in serial. On the other hand, the pipelined version generates the final image output after the execution of the stage with IFFT2. Therefore, the actual execution time per final result equals to the elapsed time of the repeat batch. The experimental platform is a PC with a Core i7 2.8GHz with 12 GByte DDR2 memory in which a Tesla C2050 is connected via PCI Express bus. Varying the input image size from 128^2 to 1024^2 using OpenCL runtimes on GPU and CPU. We apply the width of input image to the number of the threads at each kernel program. The execution times until the IFFT2 generates the final image are shown in Table 1. Due to the OpenCL runtime overhead, the speedup (serialized/parallelized) of the GPU case is about 25%. The execution time of each kernel program is small. Therefore, the parallelized version promotes the intensive utilization of GPU resource. On the other hand, in the case of CPU, the parallelized version achieves about 4 times higher performance because the parallel

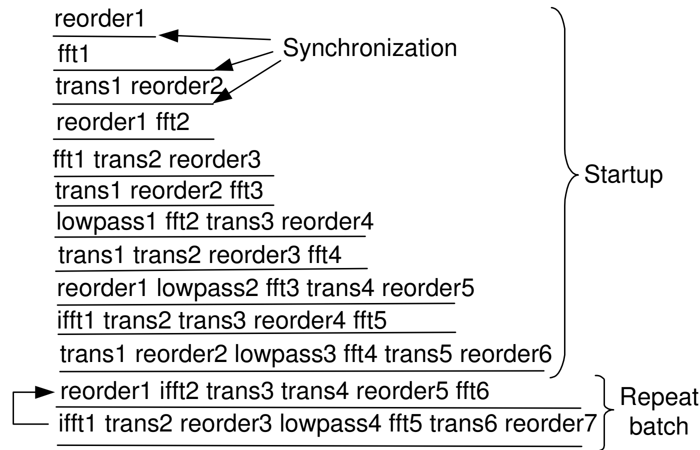

Figure 14: The startup and the repeat batch of the image filtering application resolved by the PEA-ST in the case when the maximum parallelism is seven.

Table 1: Execution times by resulting IFFT comparison among the serialized and the pipelined versions invoked on CPU and GPU. The startup time shows the elapsed times of the startup part of the batch. The repeat time shows the ones of the repeat batch.

Serialized	128^2	256^2	512^2	1024^2
GPU	1.64 sec	1.67 sec	1.77 sec	2.11 sec
CPU	4.19 sec	4.25 sec	4.38 sec	4.77 sec
Pipelined (startup)	128^2	256^2	512^2	1024^2
GPU	1.79 sec	2.18 sec	2.15 sec	1.85 sec
CPU	2.38 sec	2.40 sec	2.46 sec	2.46 sec
Pipelined (repeat)	128^2	256^2	512^2	1024^2
GPU	1.39 sec	1.40 sec	1.41 sec	1.41 sec
CPU	1.12 sec	1.12 sec	1.13 sec	1.14 sec
Speedup	128^2	256^2	512^2	1024^2
GPU	1.18	1.19	1.26	1.50
CPU	3.74	3.79	3.87	4.18

threads for different kernels are working concurrently. Therefore, in any accelerator we have confirmed that the parallelized version generated by PEA-ST achieves better performance than the serialized version.

5 Related work

Data parallelism is a kind of parallelism method of computing across multiple processors in parallel computing environments. It focuses on distributing data to different parallel computing nodes. It is a contrast to *task parallelism* as another form of parallelism.

The history of data-parallel processors began with the efforts to implement vector machines. Much of the early work on both hardware and data-parallel algorithms was pioneered at companies such as MasPar, Tera, and Cray. Now a variety of fine-grained or data-parallel programming environments are available. Many of these have achieved the recent visibility by supporting GPUs. They can be categorized as older languages(MPL [10], Co-Array Fortran [27], Cilk [4], etc.), newer languages(XMT-C [37], CUDA, CAL [8], etc.), array-based languages(RapidMind [6], Microsoft Accelerator [35], etc.) and graphics APIs(OpenGL, Direct3D). [5]

Programming languages have been designed for fine-grained parallel programming and vector processing.

OpenCL supports data parallelism and task parallelism, but does not support the pipeline parallelism for large amounts of data oriented in the case of multiple execution of the program. Also, these APIs do not fully actualize the automatic parallelization. Programmers need to write how to schedule the parallel processing and also data distribution in multiple processors.

On the other hand, the task parallelism (also known as *function parallelism* and *control parallelism*) is a form of parallelism method of computer code across multiple processors in parallel computing environments. The task parallelism focuses on allocating execution processes (threads) across different parallel computing nodes.

Examples of (fine-grained) task-parallel languages can be found in the Hardware Description Languages like Verilog and VHDL, which can also be considered as a ”code static” software paradigm where the program has a static structure and the data is changing - as against a ”data static” model where the data is not changing (or changing slowly) and the processing (applied methods) change (e.g. database search).

In our research, a flow-model is regarded as a task. These tasks (i.e. flow-models) will be parallelized in the manner of the PEA-ST. The PEA-ST algorithm is designed for the ”code static” applications. In a parallel programming language, it is necessary for programmers to note exactly which part is to be parallelized. Therefore, most automatic parallelization mechanisms perform incompletely parallelization. Although there is few programing language that does not need to specify the detailed parallelization, such as SISAL [22], Parallel Haskell [16], Mitrion-C [18]. Due to the inherent difficulties in fully automatic parallelization, several easier approaches exist to get a parallel program in higher performance. One is allowing programmers to add ”hints” to their programs to guide compiler parallelization, such as HPF for distributed memory systems and OpenMP or OpenHMPP [2] for shared memory systems. Another one is building an interactive system between programmers and parallelizing tools/compilers such as Vector Fabrics’ Pareon [9], SUIF Explorer [21], the Polaris compiler [3], and ParaWise [15]. Hardware-supported speculative multithreading is also an easier approach for automatic parallelization.

On the other hand the software pipelining is another technique that reforms the loop so that a faster execution rate is realized. Currently most effective known compiler technique for generating software pipelined loops is *modulo scheduling*. The objective of modulo scheduling is to repeat the intervals that have no intra- or inter-iteration dependence and also that no resource usage conflict arises between operations of either the same or distinct iterations. Rau and Glaeser developed the first compiler with software pipelining for the polycyclic architecture [34] that had a novel crossbar whose cross-points provided a programmable form of delay to support software pipelining. Moreover, Lam’s technique, the *modulo renaming* is widely used in practice. This method is mainly as this. First, instead of relying on specialized hardware support like a polycyclic interconnect, this method shows that the same effect can be achieved by using modulo variable expansion and unrolling the generated code a small number of times. Second, unlike hardware pipelining, dependences may cross iteration boundaries in a loop, thus creating cycles in the dependence graph. This method shows how scheduling one instruction in a strongly connected component can severely constrain the schedule of all other instructions in the same component. A heuristic technique must take this fact into consideration to be efficient. Third, this method handles conditional statements in a loop using hierarchical scheduling [20]. A related recent study is about an automatic approach to thread extraction, called Decoupled Software Pipelining (DSWP). DSWP exploits the fine-grained pipeline parallelism lurking in most applications to extract long-running, concurrently executing threads. Use of the non-speculative and truly decoupled threads produced by DSWP can increase execution efficiency and provide significant latency tolerance, mitigating design complexity by reducing inter-core communication and per-core resource requirements [29]. However, instead of the instruction level, the PEA-ST algorithm is like a task-level software pipeline method. The automatic scheduling is also the goal of our research.

Moreover, many problems can be expressed as a set of tasks where the number of tasks is input-dependent. Tasks denote an independent unit of work that can be executed in parallel with other tasks. A flow-model can be treated as a task. Therefore, finding the order of flow-models and task scheduling are somewhat similar, if ignoring the data dependency between them. Many applications have large amounts of tasks and large compute demands that we need to exploit for efficient parallel execution. In such applications, the number of tasks and sometimes the length of each task can vary dynamically. In order to effectively parallelize such applications and exploit concurrency, we must take care of issues that influence scalability. Good parallel scaling requires load balancing among the participating processors. Task queue is a well-known mechanism that is primarily designed to address the load imbalance problem [19]. The Task Queue is defined as a

mechanism to synchronously distribute a sequence of tasks among parallel threads of execution. The main problem is broken down into tasks and the tasks are enqueued into the queue. Parallel threads of execution pull tasks from the queue and perform computations on the tasks. The runtime system is responsible for managing thread accesses to the tasks in the queue as well as ensuring proper queue usage. Tasks come from using loop-level parallelism model where a set of iterations of the loop can represent a task, or using the task-parallelism model to represent parallel entities of execution, also using the memory-parallelism model to share system resources effectively for increasing data locality and performance of the application. However, compared to these task queue based scheduling algorithms, PEA-ST determines the order of flow-models (i.e. task) statically. In the PEA-ST, before running multiple tasks, it discovers potential parallelism between all tasks, and determines the execution order of all tasks. Finally, these flow-models (i.e. task) will be run by in that order.

Comparing the PEA-ST with these approaches, programmers can design processing flows without considering its parallelism of the connected flow-models. The final goal of our study is the fully automatic parallelization. The PEA-ST also takes advantage of data parallelism and it extracts the task parallelism among flow-models in a processing flow.

6 Conclusion

To address the difficulty to find a deterministic execution order of a pipeline execution flow with multiple stream-based kernel programs, we have proposed a novel algorithm called PEA-ST with the spanning tree. It determines the processing order in a pipeline manner, and also exploits the flexible parallelism from any processing flow. According to the performance evaluation, we have confirmed that the utilization ratio in a stage of pipeline becomes high. Also, the overall performance to get the result in a pipeline manner becomes higher than the serialized execution of the flow.

For the future plans, we are considering to devise a mechanism to find the best combination of the initialized edges for the data streams in flow-models. It is an indispensable guide for programmers to extract the best performance from the target accelerator. Additionally, we plan to improve the parallelism applying PEA-ST recursively to the spanning sub-tree of the nodes included in the retreating edge.

Acknowledgment

This work is partially supported by the Japan Science Technology Agency (JST) PRESTO program. And also this work is partially supported by KAKENHI (24300020) Grant-in-Aid for Scientific Research (B).

References

[1] http://www.altera.com/products/software/opencl/opencl-index.html.

[2] OpenHMPP, new hpc open standard for many-core, April 2013.

[3] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: The next generation in parallelizing compilers. In *PROCEEDINGS OF THE WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING*, pages 10–1. Springer-Verlag, Berlin/Heidelberg, 1994.

[4] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[5] Chas. Boyd. Data-parallel computing. *Queue*, 6(2):30–39, March 2008.

[6] Iris Christadler and Volker Weinberg. Facing the multicore-challenge. chapter RapidMind: Portability Across Architectures and Its Limitations, pages 4–15. Springer-Verlag, Berlin, Heidelberg, 2010.

[7] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[8] J. Eker and J. W. Janneck. Cal language report specification of the cal actor language. Technical Report UCB/ERL M03/48, EECS Department, University of California, Berkeley, 2003.

[9] Vector Fabrics. Pareon, 2012.

[10] Robert Fourer, David M. Gay, and Brian W. Kernighan. Ampl: A mathematical programming language. Technical report, MANAGEMENT SCIENCE, 1989.

[11] Michael L. Fredman and Dan E. Willard. Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. *J. Comput. Syst. Sci.*, 48(3):533–551, June 1994.

[12] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing*. Addison Wesley, 2003.

[13] Hannes Gredler and Walter Goralski. *The Complete IS-IS Routing Protocol*. SpringerVerlag, 2004.

[14] Intel OpenCL SDK. http://software.intel.com/en-us/articles/download-intel-opencl-sdk/.

[15] Stephen Johnson, Emyr Evans, Haoqiang Jin, and Constantinos Ierotheou. The parawise expert assistant – widening accessibility to efficient and scalable tool generated openmp code. In *Proceedings of the 5th International Conference on OpenMP Applications and Tools: Shared Memory Parallel Programming with OpenMP*, WOMPAT'04, pages 67–82, Berlin, Heidelberg, 2005. Springer-Verlag.

[16] Simon Peyton Jones and Satnam Singh. A tutorial on parallel and concurrent programming in haskell. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, pages 267–305, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufman, 2010.

[18] Tomasz Kryjak and Marek Gorgoń. Parallel implementation of local thresholding in mitrion-c. *Int. J. Appl. Math. Comput. Sci.*, 20(3):571–580, September 2010.

[19] Sanjeev Kumar, Christopher J Hughes, and Anthony Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 162–173. ACM, 2007.

[20] Monica S. Lam. Software pipelining: An effective scheduling technique for vliw machines. *SIGPLAN Not.*, 39(4):244–256, April 2004.

[21] Shih-Wei Liao, Amer Diwan, Robert P. Bosch, Jr., Anwar Ghuloum, and Monica S. Lam. Suif explorer: An interactive and interprocedural parallelizer. *SIGPLAN Not.*, 34(8):37–48, May 1999.

[22] J. Mcgraw, S. Skedzielewski, S. Allan, Oldehoeft Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. *SISAL: Streams and iteration in a single assignment language, language reference manual version 1.2*. Lawrence-Livermore-National-Laboratory, March 1985.

[23] Thomas J. Misa and Philip L. Frana. An Interview with Edsger W. Dijkstra. *Commun. ACM*, 53(8):41–47, August 2010.

[24] J. Moy. OSPF Version 2. Internet RFC 2328, April 1998.

[25] Aaftab Munshi, Benedict R. Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide*. Addison Wesley, 2011.

[26] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, first edition, 2007.

[27] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[28] NVIDIA Corporation. CUDA: Compute Unified Device Architecture programming guide, http://developer.nvidia.com/cuda.

[29] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *In Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*, pages 105–118. IEEE Computer Society, 2005.

[30] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[31] Radia Perlman. An Algorithm for Distributed Computation of a SpanningTree in an extended LAN. In *Proceedings of the ninth symposium on Data communications*, SIGCOMM '85, pages 44–53. ACM, 1985.

[32] Radia Perlman. *Interconnections (2nd Ed.): Bridges, Routers, Switches, and Internetworking Protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[33] R. C. Prim. Shortest Connection Networks and some Generalizations. *The Bell Systems Technical Journal*, 36(6):1389–1401, 1957.

[34] B Ramakrishna Rau and Christopher D Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *ACM SIGMICRO Newsletter*, volume 12, pages 183–198. IEEE Press, 1981.

[35] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using data parallelism to program gpus for general-purpose uses. Technical Report MSR-TR-2005-184, Microsoft Research, October 2006.

[36] Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

[37] Uzi Vishkin, Shlomit Dascal, Efraim Berkovich, and Joseph Nuzman. Explicit multi-threading (xmt) bridging models for instruction parallelism (extended abstract). In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 140–151, New York, NY, USA, 1998. ACM.

[38] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, 1996.

[39] Shinichi Yamagiwa and Leonel Sousa. Caravela: A Novel Stream-Based Distributed Computing Environment. *IEEE Computer*, 40(5):70–77, 2007.

[40] Shinichi Yamagiwa and Leonel Sousa. Modelling and Programming Stream-based Distributed Computing based on the Meta-pipeline Approach. *Int. J. Parallel Emerg. Distrib. Syst.*, 24(4):311–330, 2009.

[41] Shinichi Yamagiwa and Shixun Zhang. CarSh: A Commandline Execution Support for Stream-based Acceleration Environment. In *Procedia Computer Science, Proceedings of the International Conference on Computational Science, ICCS 2013*. ELSEVIER, 2013.