Sequential Scheduling of Dataflow Graphs for Memory Peak Minimization

Pascal Fradet
Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG
Grenoble, France
pascal.fradet@inria.fr

Alain Girault
Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG
Grenoble, France
alain.girault@inria.fr

Alexandre Honorat
Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG
Grenoble, France
alexandre.honorat@inria.fr

Abstract

Many computing systems are constrained by their fixed amount of shared memory. Modeling applications with task or Synchronous DataFlow (SDF) graphs makes it possible to analyze and optimize their memory peak. The problem studied by this paper is the memory peak minimization of such graphs when scheduled sequentially. Regarding task graphs, former work has focused on the Series-Parallel Directed Acyclic Graph (SP-DAG) subclass and proposed techniques to find the optimal sequential algorithm w.r.t. memory peak. In this paper, we propose task graph transformations and an optimized branch and bound algorithm to solve the problem on a larger class of task graphs. The approach also applies to SDF graphs after converting them to task graphs. However, since that conversion may produce very large graphs, we also propose a new suboptimal method, similar to Partial Expansion Graphs, to reduce the problem size. We evaluate our approach on classic benchmarks, on which we always outperform the state-of-the-art.

CCS Concepts: • **Theory of computation → Streaming models**.

Keywords: dataflow, task graph, SDF, memory peak, sequential scheduling

ACM Reference Format:
Pascal Fradet, Alain Girault, and Alexandre Honorat. 2023. Sequential Scheduling of Dataflow Graphs for Memory Peak Minimization. In *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3589610.3596280

LCTES '23, June 18, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0174-0/23/06…$15.00
https://doi.org/10.1145/3589610.3596280

1 Introduction

Memory footprint is an important constraint to consider when developing software applications. In the domain of artificial intelligence, some neural networks require hundreds of GigaBytes (as for GPT-3 [5]) to be stored and they cannot entirely fit in the memory of a single GPU. The modeling of neural networks with dataflow graphs [11] allows the memory requirements to be analyzed and optimized. This research topic is still very active today (see, *e.g.*, [2, 3] for recent results).

Similarly, embedded systems are subject to strict memory constraints, and, here again, dataflow graphs help to efficiently schedule the tasks w.r.t. their consumed and produced amounts of data [4].

The problem we address is to find a *sequential schedule* that minimizes the *memory peak* of a dataflow graph. This is directly useful for single processor applications as found frequently in the embedded context, but also for massively parallel applications where a GPU executes, using the Single Program Multiple Data model, the same sequence of tasks.

We take as input a dataflow graph with memory costs attached to edges and/or nodes. We consider two variants of dataflow graphs: *task graphs* and the more expressive *Synchronous DataFlow (SDF)* graphs [9]. In a task graph, each node is a task, *i.e.*, a piece of code executed atomically, and each edge between two nodes is a FIFO buffer. An SDF graph refines the task graph model by attaching two rates to each edge: the amount of data produced by its source node and the amount of data consumed by its destination node. An SDF graph can be transformed into a task graph, but this transformation potentially entails an exponential blow-up in the number of nodes in the SDF graph. In both cases, our goal is to find a static and sequential schedule of the dataflow graph whose memory peak is minimal among all possible schedules. The memory peak is the maximum shared memory needed to execute the dataflow graph.

The memory peak minimization problem of task graphs is a variation of the pebble game [15], which is NP-complete. A naive method to solve it is to generate all the linear extensions of the graph, but it has at least factorial complexity [12].

The heart of our contribution is a polynomial-time preprocessing of the task graph where we apply several graph transformations that reduce the parallelism by fusing nodes

such that: (i) the resulting graph's set of schedules is a subset of the initial graph's set of schedules, and (ii) the resulting graph's minimal memory peak is the same as the initial graph's. After this pre-processing, we apply classical algorithms to compute the memory peak, but on a much smaller task graph.

Our contributions consist of:

1. local task graph transformations that compress the given task graph while preserving a schedule with the optimal memory peak;
2. a proof that these transformations always compress Series-Parallel Directed Acyclic Graphs (SP-DAGs) to a single node representing their optimal schedule;
3. an optimized branch and bound (B&B) algorithm able to find optimal schedules for medium sized (30-50 nodes) task graphs;
4. a sub-optimal algorithm to reduce the size of the conversion of SDF graphs into task graphs;
5. experimental results that show that our transformations and algorithms outperform the state of the art for dataflow applications for which sub-optimal memory peaks are known.

The article is organized as follows. Sec. 2 presents the two considered models of computation: task graphs and SDF graphs. Sec. 3 defines the terminology and notations used in the subsequent sections. Sec. 4 presents a collection of transformations compressing task graphs while preserving the existence of an optimal schedule (optimal w.r.t. the memory peak). It also proves how these transformations are sufficient to compress any SP-DAG into a single node representing (one of) its optimal schedule. Sec. 5 presents our optimized B&B algorithm to find optimal schedules of task graphs. Sec. 6 describes techniques to apply our approach to the more general model of SDF graphs. We provide benchmark comparisons and other experimental results in Sec. 7. Finally, Sec. 8 presents related work and Sec. 9 concludes.

2 Task Graphs and SDF Graphs

Our goal is to find a static sequential schedule with the optimal (lowest) memory peak for task or SDF graphs. We present the characteristics of these two models in turn.

2.1 Task Graphs

The task graph model we consider consists of a Directed Acyclic Graph (DAG) where vertices represent tasks and edges represent FIFO communication buffers between tasks. Let A be a task in the graph G, then $\mathrm{Succ}(A)$ returns the set of its immediate successors, $\mathrm{Pred}(A)$ its set of immediate predecessors, $\mathrm{Succ}^+(A)$ and $\mathrm{Pred}^+(A)$ the sets of all its successors and predecessors in the transitive closure of G respectively.

The atomic execution of a task, referred to as a *firing*, consumes data from all its incoming edges (its inputs) and produces data to all its outgoing edges (its outputs). The data unit is called a *token*, and the same unit is used for all measures (production, consumption, memory peak). The number of tokens consumed (resp. produced) on a given edge at each firing is indicated on the edge and is called the consumption (resp. production) *rate*. A task can fire only when all its input edges contain enough tokens. It then consumes and produces a number of tokens from its ingoing and on its outgoing edges equal to the rate corresponding to each edge.

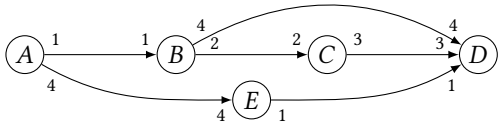


Figure 1. A simple task graph example.

Fig. 1 presents a simple task graph with five tasks A, B, C, D, and E. When fired, task B consumes 1 token on its input edge and produces 2 and 4 tokens on its two output edges, which will be eventually consumed by C and D respectively, when they will fire. Task A does not have any ingoing edges: it is a pure producer called a *source*. Task D is a pure consumer and is called a *sink*. There are three possible schedules for this graph. In general, for a connected graph of n tasks, there can be up to $(n-1)!$ schedules.

The memory occupied during execution by a task graph is the sum of all the tokens present on all its edges, *i.e.*, buffers. We assume that the FIFO buffers are allocated in the same global shared memory. Compared to implementations where each buffer is allocated independently in memory, the model we consider, called the *shared buffer model* [13], minimizes memory requirement. The memory peak of a schedule is the maximum memory needed at any execution step of that schedule.

There exist two local memory models for tasks:

- the consumed-before-produced (CBP) model where a task first consumes its input tokens, then executes its block of code, and finally produces its output tokens;
- the produced-before-consumed (PBC) model where a task keeps its input tokens to execute and produce its result before consuming (freeing) them.

The memory peak of a schedule depends on the chosen task memory model. For the task graph of Fig. 1, the minimal memory peak is 8 for the CBP model and 10 for the PBC model. Both are obtained for the schedule $A;E;B;C;D$. Our approach can take those two local task models (and others) into account.

2.2 SDF Graphs

The SDF [9] dataflow model generalizes task graphs by allowing different consumption and production rates on a given edge. Nodes are called *actors* and have the same rules as tasks for firing.

Heterogeneous rates imply that each actor must be fired multiple times to balance the production and consumption of tokens on each edge. For instance, in the SDF graph $A \xrightarrow{2}{1} B$, each firing of A produces 2 tokens and each firing of B consumes 1 token; hence, each firing of A must be eventually followed by two firings of B.

Consistent SDF graphs ensure on each edge $A \xrightarrow{r}{s} B$ the *balance equation* (1), with $\#X$ denoting the number of firings of actor X.

$$\#A \cdot r = \#B \cdot s \tag{1}$$

In other words, there exists a set of firings, called an *iteration*, which consumes exactly all tokens produced on each edge. For the basic graph $A \xrightarrow{2}{1} B$, the minimal solution of the balance equation is $\#A = 1$ and $\#B = 2$ and the minimal iteration is $\{A, B, B\}$ (see Fig. 7 in Sec. 6 for another SDF example). Consistent SDF graphs can be executed indefinitely with bounded memory and, as in most work, we consider only consistent SDF graphs.

Finally, consistent SDF graphs can be converted into task graphs (also called Single-Rate SDF (SRSDF) or Homogeneous SDF (HSDF) [4]) by duplicating each actor into n tasks, n being its number of firings in the minimal iteration. This conversion produces a number of tasks equals to the length of the iteration which can be, in pathological cases, exponential in terms of number of actors.

3 Preliminaries and Notations

We first define more precisely schedules and the key notions of *memory peak* and *memory impact*. We then introduce *schedule graphs*, the representation of task graphs that we use in our analyses and transformations.

For a given task graph G, a sequential schedule is either a single task A (a node of G) or the sequence of two schedules as formalized by the following grammar:

$$S ::= A \mid S_1; S_2$$

For our purposes, a schedule S has two key attributes:

- its *peak*, *i.e.*, the maximal memory peak reached during its execution;
- its *impact*, *i.e.*, the final number of tokens added or removed after its execution.

These attributes are either denoted directly on the schedule as $S^{\binom{p}{i}}$ or referred to as p_s and i_s when the context makes it unambiguous. For any $S^{\binom{p}{i}}$, we have $p \geq i$ and $p \geq 0$ whereas i can be negative.

A schedule's peak and impact are expressed independently of the state of the memory before its execution. For instance, let π be the current memory peak already reached and μ the current number of tokens in memory just before starting $S^{\binom{p}{i}}$ then, after its execution, the new peak will be $\max(\pi, \mu + p)$ and the final number of tokens will be $\mu + i$.

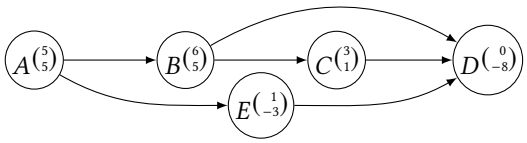


Figure 2. A schedule graph for the task graph of Fig. 1 in the PBC model.

A node A that produces r tokens and consumes s tokens is represented by:

- $A^{\binom{\max(0,r-s)}{r-s}}$ in the CBP model;
- $A^{\binom{r}{r-s}}$ in the PBC model.

Both models entail the same impact, but the peak of a task in the PBC model is always the number of tokens it produces whereas, in the CBP model, the number of consumed tokens is first subtracted and the peak might be null. Applications where some tasks require more memory to perform their computation could be expressed as well by adjusting their peaks accordingly.

We represent task graphs as *schedule graphs* where each node represents a schedule (a single task being a schedule of length 1) decorated with its peak and impact, as in Fig. 2.

The memory peak of a complete schedule can be computed using the following associative operation:

$$A^{\binom{p_a}{i_a}}; B^{\binom{p_b}{i_b}} = (A; B)^{\binom{max(p_a, p_b + i_a)}{i_a + i_b}} \tag{2}$$

The impact of the sequential execution of nodes A and B is the sum of their impacts, whereas its memory peak is the maximum of the peak reached during A and the peak reached during B starting at the impact left by A.

For example, the peak of the schedule $A; E; B; C; D$ of the graph of Fig. 2 can be computed as:

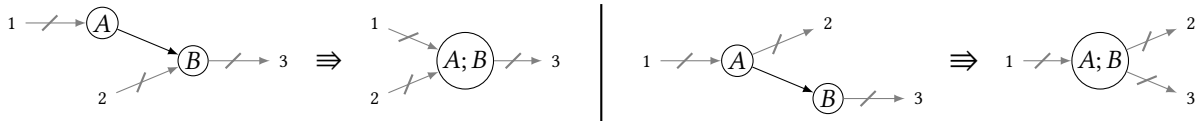
$$\begin{aligned} A^{\binom{5}{5}}; E^{\binom{1}{-3}}; B^{\binom{6}{5}}; C^{\binom{3}{1}}; D^{\binom{0}{-8}} &= (A; E)^{\binom{6}{2}}; (B; C)^{\binom{8}{6}}; D^{\binom{0}{-8}} \\ &= (A; E; B; C)^{\binom{10}{8}}; D^{\binom{0}{-8}} \\ &= (A; E; B; C; D)^{\binom{10}{0}} \end{aligned}$$

Each intermediate step in the above computation can be represented as a schedule graph as well. For instance, the node $(A; E)^{\binom{6}{2}}$ is a schedule with two outgoing edges towards $(B; C)$ and D. A task graph or a consistent SDF graph consumes exactly the number of tokens it produces: its global schedule starts with a null memory peak and impact and completes with an arbitrary peak but always with a null impact. In the previous schedule, the peak 10 is reached while executing task C.

In the next section, we use the following order relation:

$$S_1^{\binom{p_1}{i_1}} \geq_p S_2^{\binom{p_2}{i_2}} \Leftrightarrow p_1 \geq p_2 \tag{3}$$

Our goal is to find minimal schedules according to this order.



$\text{Succ}(A) = \{B\} \ \wedge \ (i_A \geq 0) \ \wedge \ (p_B + i_A \geq p_A)$ (C1)

$\text{Pred}(B) = \{A\} \ \wedge \ (i_B \leq 0) \ \wedge \ (p_A \geq p_B + i_A)$ (C2)

Figure 3. Clustering rules for single successor (left) and single predecessor (right). Striked through arrows ⇢ represent 0 or more incoming/outgoing edges.

4 Optimal Schedule Graph Compression

We present transformations that simplify the schedule graph (in number of nodes and edges) while preserving the optimality of the memory peak analysis. We reduce, cluster, and sequentialize the graph but ensure that there still exists a schedule with the *same minimal memory peak* than on the original graph.

Combined and applied repetitively, these local transformations compress substantially most task graphs. They even compress many graphs (and, among them, all SP-DAGs) to a single node representing an optimal schedule. In the general case, compressed graphs may remain large but have less schedules than the original ones.

We first present the reduction transformation in Sec. 4.1, followed by clustering in Sec. 4.2, and sequentialization in Sec. 4.3. The global compression algorithm is presented in Sec. 4.4 and its application to SP-DAGs in Sec. 4.5.

4.1 Transitive Reduction

The first graph transformation suppresses useless edges. It does not directly remove schedules but makes the subsequent transformations more effective.

In our schedule graphs, edges do not carry other information than dependencies: removing one does not modify the impact nor the peak of the nodes that it connects. However, removing an edge must not modify the scheduling constraints. Consider the subgraph B, C, D in Fig. 2. The edge $B \longrightarrow D$ can be removed since it does not add any scheduling constraint. In any case, those three nodes must be executed in the order $B; C; D$ (possibly interleaved with some subschedules), the edge $B \longrightarrow D$ being present or not.

This is generalized by suppressing any edge between two nodes A and B that are connected via another path made of multiple edges. This transformation is the classic transitive reduction [1]. It returns the graph with the fewest possible edges that keeps the same reachability relation.

4.2 Node Clustering

Let A and B be two nodes such that $\text{Succ}(A) = \{B\}$ and $\text{Pred}(B) = \{A\}$. The $A \longrightarrow B$ dependency implies that any sequential schedule of the graph is of the form $\ldots; A; S; B; \ldots$ with S a schedule made of nodes that do not depend on A nor B. Therefore, S could also be executed before A or after B.

If we can prove that, for any S, the peak of $A; S; B$ is greater or equal than the peak of $S; A; B$ or $A; B; S$, then there is no gain in interleaving S between A and B. In that case, A and B can be clustered in a single node $(A; B)$. The resulting graph has much less schedules since we got rid of schedules that could not lead to a strictly better peak.

This simple case is generalized by the two transformations Rules (C1) and (C2) of Fig. 3 that are proved correct by the two following properties.

• $\text{Succ}(A) = \{B\}$. Here, B is the unique successor of A, but B may have several predecessors. Here, all nodes that can be executed between A and B can also be executed before A. In general, they cannot all be executed after B since B has predecessors that must be executed before. The following property states the precise arithmetic conditions to allow clustering A and B when $\text{Succ}(A) = \{B\}$.

Property 1. *Let A and B be two nodes of a schedule graph.*

$$\forall S, \ i_a \geq 0 \ \wedge \ p_b + i_a \geq p_a \ \Leftrightarrow \ (A; S; B) \geq_p (S; A; B)$$

Proof.

($\Rightarrow$) $i_a \geq 0 \wedge p_b + i_a \geq p_a$ is a sufficient condition. Indeed,

$$\begin{cases} i_a \geq 0 & \Rightarrow \quad p_s + i_a \geq p_s \\ p_b + i_a \geq p_a & \Rightarrow \quad p_b + i_a + i_s \geq p_a + i_s \end{cases}$$

therefore

$$\max\{p_a, p_s + i_a, p_b + i_a + i_s\} \geq \max\{p_s, p_a + i_s, p_b + i_s + i_a\}$$

which is the expanded form of $(A; S; B) \geq_p (S; A; B)$ according to Eqs. (2) and (3).

($\Leftarrow$) We show that if the condition does not hold, *i.e.*,

$$(i_a < 0) \ \vee \ (p_a > p_b + i_a)$$

then there exists a schedule S which must be executed between A and B to minimize the peak, *i.e.*,

$$(S; A; B) >_p (A; S; B)$$

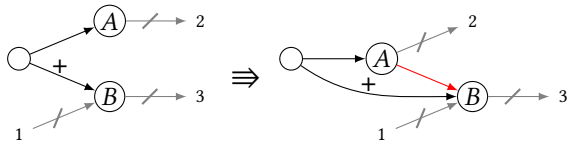
∘ **Case** $i_a < 0$. Let S be a node of the schedule graph s.t.

$$(i_s = 0) \ \wedge \ (p_s > p_a) \ \wedge \ (p_s > p_b + i_a)$$

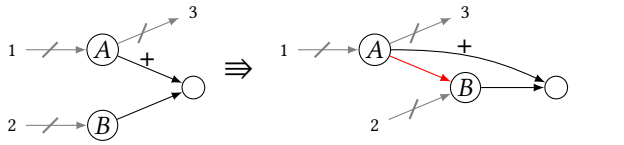
It follows that $p_s = \max\{p_s, p_a + i_s, p_b + i_s + i_a\}$ and $p_s > \max\{p_a, p_s + i_a, p_b + i_a + i_s\}$, hence $(S; A; B) >_p (A; S; B)$.

∘ **Case** $i_a \geq 0$ and $p_a > p_b + i_a$. Let S be a node of the schedule graph s.t.

$$(p_s = i_s) \ \wedge \ (i_s > 0)$$



$\text{Pred}(A) \subseteq \text{Pred}^+(B) \wedge (i_A \leq 0) \wedge (p_B \geq p_A)$ (S1)



$\text{Succ}(B) \subseteq \text{Succ}^+(A) \wedge (i_B \geq 0) \wedge (p_A + i_B \geq p_B + i_A)$ (S2)

Figure 4. Sequentialization rules for same predecessors (left) and same successors (right). The red arrow from A to B is added by the transformation. Striked through arrows $\not\rightarrow$ represent 0 or more incoming/outgoing edges.

Since peaks are positive, $p_a > p_b + i_a \Rightarrow p_a > i_a$. It is easy to check that $p_a + i_s = \max\{p_s, p_a + i_s, p_b + i_s + i_a\}$ and $p_a + i_s > \max\{p_a, p_s + i_a, p_b + i_a + i_s\}$ so $(S; A; B) \succ_p (A; S; B)$. □

• $\text{Pred}(B) = \{A\}$. A is the unique predecessor of B, but A may have several successors. This case is the dual of the previous one: all nodes that can be executed between A and B can also be executed after B. In general they cannot all be executed before A since A has other successors that must be executed after. The following property provides the condition to ensure that A and B can be clustered when $\text{Pred}(B) = \{A\}$.

Property 2. *Let A and B be two nodes of a schedule graph.*

$$\forall S,\ i_b \leq 0 \ \wedge\ p_a \geq p_b + i_a \ \Leftrightarrow\ (A; S; B) \geq_p (A; B; S)$$

The proof is similar to the proof of Prop. 1.

It can also be shown that clustering is an associative operation. If A and B can be clustered as well as B and C, then starting by clustering $A; B$ or $B; C$ does not matter; we end up with $(A; B; C)$ in both cases.

4.3 Node Sequentialization

Another useful transformation is the sequentialization of nodes that could be executed in any order in the original graph. The interest of sequentialization is twofold: it suppresses useless schedules and creates new clustering opportunities. Of course, this comes with conditions to ensure that sequentializing their execution cannot suppress a schedule that minimizes the memory peak.

The sequentialization of two nodes A and B is performed by the two Rules (S1) and (S2) of Fig. 4 which are proved correct by the two following properties.

• $\text{Pred}(A) \subseteq \text{Pred}^+(B)$. This topological condition ensures that every schedule that can be executed after B and before A can also be executed *after* A and B. The following property gives the exact arithmetic conditions for the sequentialization of A and B.

Property 3. *Let A and B be two nodes of a schedule graph.*

$$\forall S,\ i_A \leq 0 \ \wedge\ p_B \geq p_A \ \Leftrightarrow\ (B; S; A) \geq_p (A; B; S)$$

This entails that all schedules of the form $\ldots; B; S; A; \ldots$ cannot lead to a strictly smaller peak than schedules of the form $\ldots; A; B; S; \ldots$, and therefore we can execute A before B. This is not a sufficient condition to cluster A and B because interleaving some schedule between them might be beneficial w.r.t. the memory peak.

• $\text{Succ}(B) \subseteq \text{Succ}^+(A)$. This topological condition ensures that every schedule that can be executed after B and before A can also be executed *before* A and B. The following property gives the exact arithmetic conditions for the sequentialization of A and B.

Property 4. *Let A and B be two nodes of a schedule graph.*

$$\forall S,\ i_B \geq 0 \ \wedge\ p_A + i_B \geq p_B + i_A \ \Leftrightarrow\ (B; S; A) \geq_p (S; A; B)$$

This entails that all schedules of the form $\ldots; B; S; A; \ldots$ cannot lead to a strictly smaller peak than schedules of the form $\ldots; S; A; B; \ldots$, and therefore we can execute A before B.

The proofs of Props. 3 and 4 are similar to the proof of Prop. 1.

4.4 The Compression Algorithm

The three above transformations are combined according to their complexity into the compression algorithm sketched in Alg. 1.

Algorithm 1: Global compression algorithm (sketch)

```
/* Takes a schedule graph G and compresses it until
   none of the transformations apply                    */
changed := false;
repeat
    repeat
        repeat
            clustering(G); ▷ O(n)
        until ¬ changed;
        basic_sequentialization(G); ▷ O(n²)
    until ¬ changed;
    complete_sequentialization(G); ▷ O(n³)
    transitive_reduction(G); ▷ O(n³)
until ¬ changed;
```

The main procedure, which updates the graph G and sets the *changed* boolean to *true*, involves three nested loops.

1. clustering. It traverses the whole graph and tries to apply the two clustering Rules (C1) and (C2) (linear-time complexity). When a clustering creates a transitive edge, a local reduction is applied. It also computes the set of neighbours of the clustered nodes. Indeed, only those nodes may be candidates to further clustering. The subsequent steps of clustering are done on this set (which may evolve) until no further clustering can be done.

2. basic_sequentialization. This procedure is a simplified version of sequentialization where the topological conditions for the two rules respectively become $\text{Pred}(A) = \text{Pred}(B)$ and $\text{Succ}(B) = \text{Succ}(A)$. These are the most common cases and they are less costly to detect (quadratic-time) than the complete ones. If this step changes the graph, a new round of clustering is performed.

3. complete_sequentialization. This procedure detects the most general conditions of sequentialization (S1 and S2) and applies the corresponding transformation rules. It is followed by a step of transitive reduction, provided by a standard library. These two procedures have a worst case cubic-time complexity which may be problematic for large graphs. If these two steps have changed the graph a new round of clustering plus basic_sequentialization is performed.

The algorithm stops when no clustering, sequentialization, or reduction are possible. Its global worst case complexity is quartic-time.

4.5 The Case of Series-Parallel Graphs

The compression algorithm applies to any DAG and usually reduces significantly its size. SP-DAGs (see Fig. 5 for a simple example) are a well-know and important class of DAG for which the compression algorithm is particularly effective. For this class, it can be shown that the four Rules (C1), (C2), (S1), and (S2) can compress any SP-DAG to a single node. This single node is the clustering of (one of) its optimal schedule.

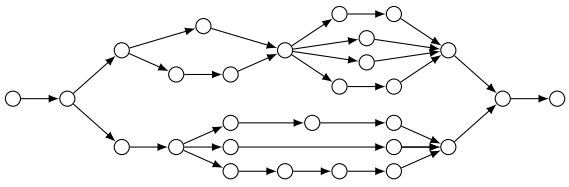


Figure 5. An example of an SP-DAG.

Definition 5. *SP-DAGs are graphs with a single source and sink nodes which can be build fusing the following rules [18]. Let A and B be two nodes and G_1 and G_2 be two SP-DAGs then*

- (base case) *the two nodes connected $A \longrightarrow B$ is an SP-DAG;*
- (sequential composition) *identifying the sink of G_1 with the source of G_2 makes a new SP-DAG;*
- (parallel composition) *identifying the source of G_1 with the source of G_2 and the sink of G_1 with the sink of G_2 makes a new SP-DAG.*

We first prove a property on linear chains, a sub-class of SP-DAG, where each node has a single predecessor and successor except the first and last nodes which have a single successor and predecessor respectively.

Property 6. *Any linear chain can be compressed by using Rules (C1) and (C2) into a chain of the form:*

$$N_1 \longrightarrow \ldots \longrightarrow N_n \longrightarrow P_1 \longrightarrow \ldots \longrightarrow P_m \quad \text{with} \quad n, m \geq 0$$

where the N_i are nodes with a negative impact and the P_i nodes with a positive impact. They are referred to as sorted chains.

Proof. In the following we refer to nodes with a negative and positive impact as *negative* and *positive nodes*. Any dependency $P \longrightarrow N$ with P a positive node and N a negative one, will be clustered into the single schedule $(P; N)$. Indeed, either one of the Rule (C1) or (C2) conditions is satisfied in this case. Therefore, by applying repetitively the clustering rules, no positive node can be followed by a negative node. □

This property directly implies that task graph applications represented by a linear chain can be clustered into a single node. Indeed, in a dataflow application, a task can only consume tokens what has been produced before: the size of memory occupied during execution cannot be negative. Therefore, the impact of any prefix of a linear chain representing a complete dataflow application is positive. Such a chain always starts with a positive node (a producer) and if it is followed by a negative one, their combined impact remains positive.

Corollary 7. *Any linear chain whose prefixes have all a positive impact can be compressed using Rules (C1) and (C2) into a single node.*

The property on SP-DAG is expressed as follows.

Property 8. *Any SP-DAG can be compressed using Rules (C1), (C2), (S1), and (S2) into a sorted chain.*

Proof. By induction based on the Def. 5 of SP-DAGs .
∘ **Base case**. The basic dag $A \longrightarrow B$ is sorted (possibly into a single node) using Rules (C1) and (C2) (Prop. 6).
∘ **Sequential case**. The sequential composition of two SP-DAGs G_1 and G_2 supposes that the sink of G_1, say A, is also the source of G_2. By induction hypothesis, G_1 and G_2 can be compressed into two sorted chains. If A has been clustered into G_1 or G_2 we extract it: a clustered node $(X; A)$ or $(A; X)$ can be unclustered into the chains $X \longrightarrow A$ and $A \longrightarrow X$. The chains are joined in A in a single chain which can be sorted.
∘ **Parallel case**. The parallel composition of two SP-DAGs G_1 and G_2 supposes that the source and sink of G_1, say A and B, are also the source and sink of G_2. By induction hypothesis, G_1 and G_2 can be compressed into two sorted chains. If A or B have been clustered into G_1 or G_2 we can extract them

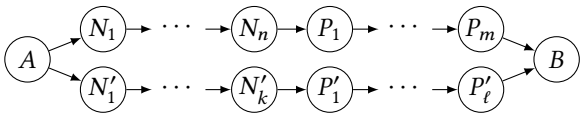


Figure 6. Parallel composition of two sorted chains.

as before. The two chains are joined in A and B to get a graph of the form depicted in Fig. 6. If A is connected to two negative nodes N_1 and N'_1 then Rule (S1) always applies. We can sequentialize N_1 or N'_1 depending on which has the greater peak and suppress the edge of $A \to N'_1$ or $A \to N_1$ by (local) transitive reduction. Similarly, if the two predecessors of B are positive nodes then Rule (S2) always applies. In both cases, one of the parallel branches get shorter: new nodes are serialized before A and after B to form eventually a linear chain. The problematic case is when A has a positive successor and a negative one and B has a positive predecessor and a negative one. In this case, the conditions to apply Rules (S1) and (S2) may not be satisfied. However, since the two chains are sorted, this case can occur only when one chain is made of only positive nodes $P_1 \dots P_n$ while the other is composed of only negative ones $N_1 \dots N_m$. The two chains being maximally clustered, we know that:

- the peak condition of Rule (C1) is false on all the P_i, which entails that $p_{p_2} + i_{p_1} < p_{p_1}, p_{p_3} + i_{p_2} < p_{p_2}, \dots$ and since all impacts are positive: $p_{p_m} < p_{p_1}$;
- the peak condition of Rule (C2) is false on all the N_i, which entails that $p_{n_1} < p_{n_2} + i_{n_1}, p_{n_2} < p_{n_3} + i_{n_2}, \dots$ and since all impacts are negative: $p_{n_1} < p_{n_m}$.

Now, assume that N_1 and P_1 cannot be sequentialized because the condition of Rule (S1) is not satisfied *i.e.*, because $p_{p_1} < p_{n_1}$. Then, we just pointed out that $p_{p_m} < p_{p_1} < p_{n_1} < p_{n_m}$ and since i_{p_m} is positive and i_{N_m} is negative we have $p_{n_m} + i_{p_m} > p_{p_m} + i_{n_m}$ which is the condition of Rule (S2) to sequentialize P_m. Therefore, in any case, there always exists node that can be sequentialized. The parallel composition is eventually transformed into a linear chain which can be sorted by Rules (C1) and (C2) if needed. □

According to Prop. 8 any SP-DAG representing a dataflow application can be compressed into a linear chain representing a schedule. Being a dataflow application, all prefixes of this chain have a positive impact and Corollary 7 applies.

Theorem 9. *Any SP-DAG representing a dataflow application can be compressed using Rules (C1), (C2), (S1), and (S2) into a single node.*

Even if Alg. 1 works well enough on SP-DAGs, those could be compressed by a simpler recursive algorithm expressed along Def. 5 decomposition. Global nested iterations and transitive reduction become useless and sequentialization can be specialized to deal only with the case of Fig. 6.

5 Branch and Bound Algorithm

On some DAGs, the previous compression techniques do not manage to compress the graph into a single node containing the optimal schedule achieving the minimal memory peak. Instead, they produce a (smaller) DAG whose optimal memory peak has yet to be found. As already mentioned, this can be done by computing all the linear extensions of the DAG and their memory peak. This naive approach is costly and, in our experience, can deal only with small graphs ($\leq$ 15 tasks/nodes).

We propose instead to use a B&B algorithm that explores the tree of all schedules in a depth-first manner. The depth of this tree is equal to the number of tasks and each branch corresponds to a decision to schedule one node of the DAG, chosen in the so-called ready list that contains all the nodes having all their predecessors already scheduled. Initially, the ready list contains the source nodes of the DAG.

Even though the theoretical time complexity of B&B remains exponential in the size of the DAG, since we compute the *minimum* memory peak, we expect that branches that will be cut in the tree will be closer to the root, resulting in a better practical time complexity. Our algorithm is therefore a classical B&B algorithm tuned with several optimizations. A simple one is to start the algorithm with an initial upper bound provided by the user or by a heuristic. We present two other optimizations in the following sections.

5.1 Fast Backtracking

Our first optimization takes advantage of the minimization nature of our problem. When building the sequential schedule, we retain the node X that caused the last memory peak value. When we reach the end of the scheduling tree with a full schedule, we backtrack directly to the immediate predecessor of X in the tree. Indeed, since the memory peak of this schedule was already reached when scheduling X, the entire sub-tree starting at X cannot result in a smaller memory peak.

This direct and long backtracking significantly reduces the computation time by pruning the tree of schedules. If we assume a constant branching factor b and a number of k remaining nodes to be scheduled from the backtracking point, then the number of pruned schedules is b^k.

5.2 Optimization for Negative Nodes

Our second optimization takes advantage of the peak/impact information to prune the ready list present at each node. Assuming we are at step n with the ready list Rdy_n, the current memory used $MemCurr_n$, and the current memory peak $MemPeak_n$, pruning can occur when Rdy_n contains at least one negative node. We select the negative node A ($i_a \leq 0$) with the smallest peak p_a. Two cases are possible:

- If A does not increase the memory peak ($MemCurr_n + p_a \leq MemPeak_n$), then we only keep A in the ready

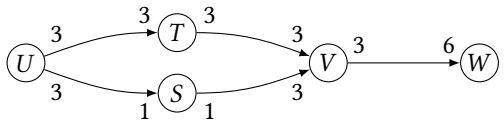


Figure 7. SDF example: the simple graph $\mathcal{G}$.

list. The reason is that scheduling at this step any other node can only result in a higher memory peak.

- If the above condition is not met, then we remove from the ready list all nodes B such that $p_b \geq p_a$. Indeed, since A does not increase the current memory, no gain can be expected by scheduling at this step a node having a larger peak. However, nodes with a positive impact and a smaller peak can eventually lead to a better peak and are kept in Rdy_n.

In both cases, the nodes removed for the ready list at this step remain to be scheduled and will be in the ready lists of subsequent steps. Pruning nodes from Rdy_n yields a computational benefit when backtracking to this step n. If we assume a constant branching factor b and a number of k remaining tasks to be scheduled after step n, then removing a single node from Rdy_n prunes b^{k-1} schedules.

Our B&B algorithm finds the optimal memory peak of applications up to 50 nodes/tasks. Combined with the compression algorithm, many task graph applications can be analyzed optimally. Furthermore, a timeout always permits to retrieve at least an over-estimation of the peak.

6 Extension to SDF Graphs

As described in Sec. 2, SDF graphs are a strict extension of tasks graphs. Our technique can be applied by first converting an SDF graph into an SRSDF graph [4], which is precisely a task graph. This standard transformation can be applied to *any* (even cyclic) consistent and live SDF graph and produces a task graph encoding a minimal iteration of the initial SDF graph. Despite the potential exponential blow up of this conversion, this approach remains effective to analyze the memory peak of many SDF graphs. There are nonetheless some SDF applications for which it produces too large task graphs or schedules.

We present a technique to reduce the number of firings of each SDF actor so that the task graph obtained after conversion is analyzable optimally and/or the schedule is compact enough. Fig. 8 summarizes the different transformation steps from the input SDF graph $\mathcal{G}$ to a reduced SDF graph $\mathcal{G}^{\mathcal{R}}$ that is converted to an SRSDF graph G, compressed into G^C and analyzed by the B&B algorithm. The graphs $\mathcal{G}$ and $\mathcal{G}^{\mathcal{R}}$ have the same number of actors, but the total number of firings expressed in $\mathcal{G}^{\mathcal{R}}$ is lower than in $\mathcal{G}$.

We first present a coarse but well-known conversion that produces G with the same number of nodes as actors in $\mathcal{G}$, *i.e.*, $|\mathcal{G}| = |G|$. Such coarse conversion allows the analysis of

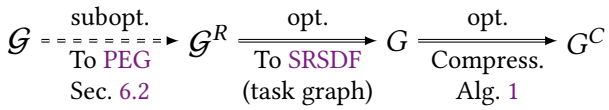


Figure 8. Graph transformation chain for an SDF graph $\mathcal{G}$.

any SDF graph but the schedules have a much larger memory peak than the optimal. We then present a more refined technique inspired from the work on Partial Expansion Graphs (PEGs) [19] which can incrementally reach an objective expressed in terms of a maximum number of firings.

6.1 Flat Single Appearance Schedules

To reduce the number of nodes in the expanded graph, a drastic approach is to ensure that each actor X executes its $\#X$ firings in a row. To enforce this policy, each edge rate is set to the total production or consumption of an iteration as in Eq. (1). Each node executes once and the graphs $\mathcal{G}$ and G have the same number of nodes. The schedules of such a graph correspond to what is called flat Single Appearance Schedules (SASs) in [4].

For instance, solving the balance equations of the SDF graph $\mathcal{G}$ in Fig. 7 yields the set of firings $\{\#U{=}2, \#T{=}2, \#S{=}6, \#V{=}2, \#W{=}1\}$ for the minimal iteration. Regrouping all the firings of each actor is enforced by changing the production and consumption rates of all edges to 6. Denoting X^k the actor that executes X k times in a row, all actors in the resulting graph $\mathcal{G}^R$ are of the form $X^{\#X}$. For Fig. 7, the corresponding schedule of $\mathcal{G}^R$ is $U^2; T^2; S^6; V^2; W$, which is a flat SAS of $\mathcal{G}$.

A flat SAS schedule usually has a much higher memory peak than the optimal one obtained for the SRSDF expansion of the same SDF graph. The flat SAS schedule prevents from interleaving different node firings, in particular firings of negative nodes between firings of positive nodes. For example, the previous flat SAS schedule gives a peak of 18 tokens, against only 12 tokens for the optimal.

6.2 Partial Expansion Graphs

Instead of grouping all the firings of an actor, a more refined technique is to consider any possible grouping of its firings according to its integer divisors. For example, the actor S in Fig. 7 could be transformed either into the actor S^2, S^3, or S^6, respectively appearing thrice, twice, or once in the schedule.

Figure 9. Firing reduction of actor A by divisor d.

Our PEG transformation is based on this idea: we modify the graph progressively by setting a greater firing divisor to

every actor. Each modification leads to a suboptimal schedule, and iterating this process eventually produces a flat SAS.

When grouping the firings of an actor A, its input (resp. output) rates s (resp. r) are updated accordingly, as depicted in Fig. 9. This modification is *local* to the considered actor A and does *not* modify the balance equations of the graph. As it may increase the memory peak, we define a score criterion to always modify the actor with the smallest memory requirement. Let $\{d_1 = 1, \ldots, d_z = \#A\}$ be the ordered set of divisors of $\#A$. Each divisor d_k yields the corresponding input (resp. output) rates $s_k = d_k \cdot s$ (resp. $r_k = d_k \cdot r$). The score of A currently at divisor k is defined by Eq. (4).

$$\begin{aligned} \text{score}(A,k) = & \sum_{A,k \xrightarrow{r_k\ s_\ell} X,\ell} \left\lceil \frac{s_\ell}{r_{k+1}} \right\rceil r_{k+1} - \left\lceil \frac{s_\ell}{r_k} \right\rceil r_k \\ & + \sum_{X,\ell \xrightarrow{r_\ell\ s_k} A,k} \left\lceil \frac{s_{k+1}}{r_\ell} \right\rceil r_\ell - \left\lceil \frac{s_k}{r_\ell} \right\rceil r_\ell \end{aligned} \tag{4}$$

We take as the memory requirement of actor A the sum of tokens needed to be produced by its predecessors for a single firing of A, and of tokens needed to be produced by A to enable a single firing of each of its successors. Our criterion score is defined as the difference between the requirements before and after the modification using the next divisor d_{k+1} of $\#A$; its unit is in tokens.

Consider actor S in Fig. 7 and its first divisor 2; in this case $\text{score}(S, 1) = 1$: indeed, U must be fired once (as before) to enable S^2, but S^2 must be fired twice to produce 4 tokens to enable V to fire, whereas S needed to produce only 3 tokens to enable V. For the second divisor 3, the memory requirement does not change and $\text{score}(S, 2) = 0$: indeed, only 3 tokens must be produced by A and then by V as before.

Our PEG algorithm takes an objective expressed as a maximum of firings. Until the objective is reached, it repeats the local transformation of Fig. 9 by selecting at each step the actor A and its next divisor d_{k+1} that minimizes Eq. (4). Then, the score needs to be updated only for node A and its immediate neighbors. If the objective in terms of number of firings is equal or less than the number of actors, the result will be the same as in Sec. 6.1.

7 Experiments

We compare the memory peaks obtained with previously known results. Our experimental setup is a regular laptop (Intel® Core™ i5-8265U @1.60GHz processor, 16 GB of RAM), with Linux Ubuntu 22.04. We use Python 3.10 with the graph library Networkx 2.8.8. All the results presented here assume the PBC model.

Concerning task graphs, most existing work focus on tree-shaped graphs or SP-DAGs but do not provide benchmarks. For such graphs, we always find the optimal memory peak very quickly, even for very large ones.

Table 1. Memory peaks for the satellite application.

satellite	$\lvert G\rvert$	[14]	[13]	[8]	[ours]	sec.
flat SAS	22	1,920	—	**1,680**	**1,680**	0.002
SDF	4,515	—	991	**960**	**960**	7.7

Table 1 presents the results for satellite, one of the first SDF applications used to analyze the memory peak. We handle it as a task graph with 22 nodes (by analyzing only its flat SAS schedules), and as an SDF graph that is converted into a task graph with 4,515 nodes. Ritz *et al.* [14] consider only flat SAS schedules. The problem, which consists of scheduling 22 tasks, was expressed as an Integer Linear Programming (ILP) problem and took 4 days to provide an over-estimation (in 1995). Murthy *et al.* [13] consider the SDF application and use heuristics to produce memory efficient schedules whose peak is evaluated afterwards. The technique presented in [8] finds optimal memory peaks for SP-DAGs and over-estimated peaks for general task graphs. Even if the two versions of satellite (flat SAS and SDF) are not SP-DAGs, [8] finds the optimal memory peaks for both. Our graph transformations compress both versions to a single node, giving the optimal peaks in a short runtime (last two columns in Table 1).

Table 2. Memory peaks for different qmf filterbank benchmark applications in [13].

filterbank	$\lvert G\rvert$	[13]	[8]	[ours]	sec.
qmf23_2d	78	22	18	**13**	0.007
qmf23_3d	324	63	53	**31**	0.06
qmf23_5d	4,536	492	405	**247**	6.7
qmf12_2d	40	9	10	**7**	0.003
qmf12_3d	112	16	20	**11**	0.009
qmf12_5d	704	58	79	**35**	0.1
qmf235_2d	190	55	45	**22**	0.03
qmf235_3d	1,300	240	133	**47**	0.7
qmf235_5d	50,000	5,690	1,190	**272**	802.5

Table 2 presents results obtained on Quadrature Mirror Filterbanks (QMFs), a well-known tree-structured class of signal processing applications often used in benchmarks for SDF. Different versions of filterbank exist: the SDF graph topology remains the same but the number of nodes (nesting depth) and the rates vary. We evaluate the memory peak on the nine filterbank versions used in [13], where qmf[ij]_[l]d (resp. qmf[ijk]_[l]d) refers to a version with rates i and j (resp. i, j, and k) and a depth ℓ. The $\lvert G\rvert$ column shows the total number of nodes of the corresponding SRSDF graph. The next column gives the suboptimal results obtained by [13] based on the same heuristics as for satellite. The SRSDF extensions of filterbank are not SP-DAGs and the technique

of [8] only finds over-estimations. The results obtained by our algorithm Alg. 1 are given in the last two columns of Table 2 where we indicate the memory peak found and the total runtime. On all filterbank versions, our graph transformations always compress the corresponding SRSDF graph to a single node with optimal memory peak.

Table 3. PEG reduction on filterbank applications in Table 1 with more than 4,000 tasks.

filterbank	PEG (increasing objective from flat SAS)				
	188	400	800	1,600	3,200
qmf23_5d	729	351	283	267*	253†
qmf235_5d	9,375	4,750	3,625*	3,625*	3,275†

The PEG reduction is useful when graph transformations cannot be completed in a reasonable time and/or to reduce the size of schedules. In Table 3, we evaluate the PEG algorithm presented in Sec. 6 to get shorter schedules for qmf23_5d and qmf235_5d (for which we found optimal schedules but respectively 4,536 and 50,000 tasks long).

In Table 3, we present the memory peaks found for different PEGs objectives expressed as number of firings: 188 (which is the flat SAS), 400, 800, 1,600, and 3,200. In five cases (marked by $*$ and †), the corresponding task graph is not reduced to a single node and the B&B algorithm is applied. In only two of those cases (marked by †), the resulting schedule graph is too large (123 and 225 nodes for qmf23_5d and qmf235_5d respectively) and the B&B, stopped by its 600 sec. timeout, returns an over-estimated result. These results show the trade-off between the schedules size and the memory peak. For qmf235_5d, dividing the number of firings by ~ 266 (from 50,000 to 188, which is the flat SAS size) multiplies the memory peak by ~ 34 (from 272, the optimal peak, to 9,375). Then, subsequent increases of the number of firings decrease the memory peak.

8 Related Work

Much work aiming at minimizing memory requirements for SDF graphs assume that buffers are allocated independently in memory (nonshared model) [4, 7, 16]. The analyses search for the minimal size of each buffer to ensure live executions.

The more memory efficient *shared buffer model* assume that buffers are allocated in the same global shared memory. The analyses focus on minimizing the gobal memory needed *i.e.*, the memory peak. The problem of sequential scheduling of dataflow graphs for memory peak minimization has been studied on two main aspects: optimal solutions for a subclass of task graphs and suboptimal solutions for SDF graphs.

Regarding optimal solutions, previous work presented algorithms to find the optimal schedules in $O(n^2)$ for tree-shaped task graphs [10], and in $O(n^3)$ for SP-DAGs [8]. Our graph transformations presented in Sec. 4 solve optimally the same classes of graphs with the same time complexity. Our graph transformations are more general though since they apply to any task and SDF graph, and along with an optimized B&B algorithm they find optimal schedules for a larger class of task graphs.

Regarding over-estimated solutions, previous work focused on SDF graphs and reduced the problem complexity by considering only SASs schedules. While [14] considered only flat SASs, [13] relaxed this restriction to non flat SASs schedules. In both cases, we outperform their benchmarks, as seen in Sec. 7. A relaxed class of SASs was also studied in [17] but we have not been able to compare with their results because neither their code nor their benchmarks are available to our knowledge.

9 Conclusion

In this article, we have proposed simple task graph transformations to reduce the problem size of sequential memory peak minimization for dataflow graphs. These transformations are local, they are proved to preserve the minimal memory peak, and they can be applied on all task graphs. The transformations alone permit to solve the problem for a larger class of task graphs than before. This class is not formally characterized, but we proved that it includes SP-DAGs. When transformations do not compress a task graph to a single node, an optimized B&B algorithm explores the (reduced) task graph to find its optimal memory peak with the corresponding schedule. Still, the conversion of SDF produce very large task (SRSDF) graphs. To deal with those, we designed a dedicated PEG transformation to reduce the schedule size and so the problem size. Together, the algorithms we have proposed significantly outperform the state of the art and reveal new optimal and better sub-optimal bounds on benchmarks.

Within an embedded context, the size of schedules can be an issue of concern. For example, there is an optimal schedule of size 5,346 task firings in our benchmarks. As future work, we intend to study this problem. Such schedules comes from a small number of actors whose individual firing is represented by a task. A first approach is to reuse work on string compression [6] to get shorter optimal schedules. A second approximated approach would be to take into account trade-offs between shorter schedules and higher memory peaks as initiated by our PEG transformation.

A natural extension of this work is to consider parallel executions with shared memory. Note that the optimal memory peak found by our approach for a sequential execution already provides a lower bound for all its parallel versions.

Another, more speculative, avenue for further research is to investigate the application of our transformation-based approach to the analysis of other task graph properties.

References

[1] Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. 1972. The Transitive Reduction of a Directed Graph. *SIAM J. Comput.* 1, 2 (1972), 131–137.

[2] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. 2021. Efficient Combination of Rematerialization and Offloading for Training DNNs. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 23844–23857. https://proceedings.neurips.cc/paper/2021/file/c8461bf13fca8a2b9912ab2eb1668e4b-Paper.pdf

[3] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. 2021. Pipelined Model Parallelism: Complexity Results and Memory Considerations. In *27th International European Conference on Parallel and Distributed Computing*. Lisbon, Portugal. https://hal.inria.fr/hal-02968802

[4] S.S. Bhattacharyya, P.K. Murthy, and E.A. Lee. 1996. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Pub., Hingham, MA.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[6] Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. 2010. Extracting powers and periods in a string from its runs structure. In *SPIRE (LNCS, 6393)*, Edgar Chavez and Stefano Lonardi (Eds.). Springer, Los Cabos, Mexico, 258–269. https://doi.org/10.1007/978-3-642-16321-0_27

[7] Marc Geilen, Twan Basten, and Sander Stuijk. 2005. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings of the 42nd Design Automation Conference, DAC 2005, San Diego, CA, USA, June 13-17, 2005*, William H. Joyner Jr., Grant Martin, and Andrew B. Kahng (Eds.). ACM, 819–824.

[8] Enver Kayaaslan, Thomas Lambert, Loris Marchal, and Bora Uçar. 2018. Scheduling series-parallel task graphs to minimize peak memory. *Theoretical Computer Science* 707 (2018), 1–23. https://doi.org/10.1016/j.tcs.2017.09.037

[9] E.A. Lee and D.G. Messerschmitt. 1987. Synchronougs data flow. *Proc. IEEE* 75, 9 (Sept 1987), 1235–1245. https://doi.org/10.1109/PROC.1987.13876

[10] Joseph W. H. Liu. 1987. An Application of Generalized Tree Pebbling to Sparse Matrix Factorization. *SIAM Journal on Algebraic Discrete Methods* 8, 3 (1987), 375–395. https://doi.org/10.1137/0608031 arXiv:https://doi.org/10.1137/0608031

[11] Svetlana Minakova, Erqian Tang, and Todor Stefanov. 2020. Combining Task- and Data-Level Parallelism for High-Throughput CNN Inference on Embedded CPUs-GPUs MPSoCs. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Alex Orailoglu, Matthias Jung, and Marc Reichenbach (Eds.). Springer International Publishing, Cham, 18–35.

[12] Rolf H. Möhring. 1989. *Computationally Tractable Classes of Ordered Sets*. Springer Netherlands, Dordrecht, 105–193. https://doi.org/10.1007/978-94-009-2639-4_4

[13] P.K. Murthy and S.S. Bhattacharyya. 2001. Shared buffer implementations of signal processing systems using lifetime analysis techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20, 2 (2001), 177–198. https://doi.org/10.1109/43.908427

[14] Sebastian Ritz, Markus Willems, and Heinrich Meyr. 1995. Scheduling for optimum data memory compaction in block diagram oriented software synthesis. In *1995 International Conference on Acoustics, Speech, and Signal Processing*, Vol. 4. 2651–2654 vol.4. https://doi.org/10.1109/ICASSP.1995.480106

[15] Ravi Sethi. 1973. Complete Register Allocation Problems. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing* (Austin, Texas, USA) *(STOC '73)*. Association for Computing Machinery, New York, NY, USA, 182–195. https://doi.org/10.1145/800125.804049

[16] Tae-ho Shin, Hyunok Oh, and Soonhoi Ha. 2011. Minimizing buffer requirements for throughput constrained parallel execution of synchronous dataflow graph. In *Proceedings of the 16th Asia South Pacific Design Automation Conference, ASP-DAC 2011, Yokohama, Japan, January 25-27, 2011*. IEEE, 165–170.

[17] Wonyong Sung, Junedong Kim, and Soonhoi Ha. 1998. Memory Efficient Software Synthesis Form Dataflow Graph. In *Proceedings of the 11th International Symposium on System Synthesis* (Hsinchu, Taiwan, China) *(ISSS '98)*. IEEE Computer Society, USA, 137–142.

[18] Krishnaiyan Thulasiraman and M. N. S. Swamy. 1992. *Graphs - theory and algorithms*. Wiley.

[19] George F. Zaki, William Plishker, Shuvra S. Bhattacharyya, and Frank Fruth. 2017. Implementation, Scheduling, and Adaptation of Partial Expansion Graphs on Multicore Platforms. *Journal of Signal Processing Systems* 87, 1 (Apr 2017), 107–125. https://doi.org/10.1007/s11265-016-1107-8

Received 2023-03-16; accepted 2023-04-21