Graph Partitioning with Acyclicity Constraints

Orlando Moreira[1], Merten Popp[2], and Christian Schulz[3]

1 **Intel Corporation, Eindhoven, The Netherlands**
orlando.moreira@intel.com
2 **Intel Corporation, Eindhoven, The Netherlands**
merten.popp@intel.com
3 **Karlsruhe Institute of Technology, Karlsruhe, Germany; and University of Vienna, Vienna, Austria**
christian.schulz@{kit.edu, univie.ac.at}

Abstract

Graphs are widely used to model execution dependencies in applications. In particular, the NP-complete problem of partitioning a graph under constraints receives enormous attention by researchers because of its applicability in multiprocessor scheduling. We identified the additional constraint of acyclic dependencies between blocks when mapping streaming applications to a heterogeneous embedded multiprocessor. Existing algorithms and heuristics do not address this requirement and deliver results that are not applicable for our use-case. In this work, we show that this more constrained version of the graph partitioning problem is NP-complete and present heuristics that achieve a close approximation of the optimal solution found by an exhaustive search for small problem instances and much better scalability for larger instances. In addition, we can show a positive impact on the schedule of a real imaging application that improves communication volume and execution time.

1998 ACM Subject Classification G.2.2 Graph Theory

Keywords and phrases Graph Partitioning, Computer Vision and Imaging Applications

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.30

1 Practical Motivation

The context of this research is the development of computer vision and imaging applications at Intel Corporation. These applications have high demands for computational power but often need to run on embedded devices with severely limited compute resources and a tight thermal budget. Our target platform is a heterogeneous multiprocessor for advanced imaging and computer vision and is currently used in Intel processors. It is designed for low power and has small local program and data memories. To cope with the memory constraints, the application developer currently has to *manually* break the application, which is given as a directed dataflow graph, into smaller blocks that are executed one after another. The quality of this partitioning has a strong impact on communication volume and performance. However, for large graphs this is a non-trivial task that requires detailed knowledge of the hardware. Hence, the task should be handled by a well-designed algorithm instead.

There are many existing heuristics for partitioning graphs into blocks of nodes of roughly equal size. However, our platform has the requirement that there must not be a cycle in the dependencies between the blocks because they have to be executed one after another.

The contributions of this work are the identification of a new variation of the graph partitioning problem, proofs it is NP-complete and hard to approximate, as well as the implementation and evaluation of heuristics that address this problem. First, we present

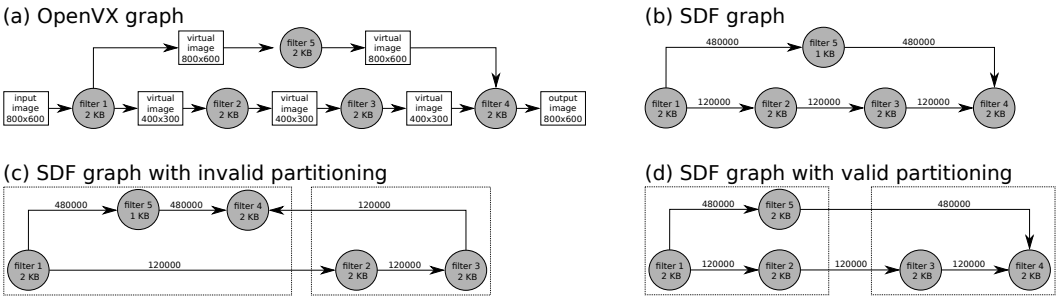


Figure 1 Depiction of an imaging application graph. The nodes represent processing kernels and are annotated with the size of the program binaries. The OpenVX graph as specified by the developer using the API standard is shown in (a), virtual images are intermediate data. The initial SDF representation is shown in (b) where the edges are annotated with the buffer size, (c) shows an invalid partition with minimal edge cut, but a bidirectional connection between blocks and thus a cycle in the quotient graph. A valid partitioning with minimal edge cut is shown in (d).

all necessary background information on the application graph and hardware and explain our additional constraint in Section 2. We then continue to briefly introduce all basic concepts and related work in Section 3. We have not been able to identify works that address the aforementioned constraint of our problem variant, which originates from the hardware platform and disallows cycles between blocks. The proofs are found in Section 4 and the proposed heuristic algorithms in Section 5. We perform a number of experiments in Section 6, where we use small graphs to compare our heuristics against an optimal algorithm that uses exhaustive search. We then evaluate our heuristics for larger graphs in the context of a real-world imaging application and estimate the impact on the application. In addition, we demonstrate the scalability of our heuristics with a set of large graphs. Finally, we conclude in Section 7.

2 Background

Computer vision and imaging applications can often be expressed as stream graphs where nodes represent tasks that process the stream data and edges denote the direction of the dataflow. The widely-accepted industry standard OpenVX [11] released in 2014 by the Khronos group uses a graph-based execution model. With the OpenVX API the developer can specify the data flow of the application as a graph independent of hardware constraints. The hardware vendor on the other hand can provide an API implementation that uses advanced optimizations [21] like specialized hardware, parallelized and pipelined node execution, overlapped computation and data transfers and aggregated data transfers to avoid a round trip to external memory.

The application is specified as a Directed Acyclic Graph (DAG) in OpenVX. The nodes of the DAG are either kernels (small, self-contained functions) or data objects. Edges denote data dependencies and always connect exactly one kernel with one data object. No cycles or feedback loops are allowed [11]. The need of some imaging algorithms to access previous data (e.g. video stabilization) is addressed by special OpenVX delay objects that hold data of previous graph executions. Our existing tool flow converts the OpenVX graph in linear time into an Synchronous Dataflow (SDF) graph. SDF is a model of computation that abstracts from functionality and enables several prevalent analysis and scheduling techniques [15]. In this representation, nodes represent processing and the directed edges represent FIFO

buffers. The SDF nodes are annotated with the program size for each kernel in the OpenVX graph. If two kernels are linked by a data object in the OpenVX graph, the SDF nodes are connected by a directed edge annotated with the size of the data object. The resulting graph is a DAG. An example of this conversion is shown in Figure 1a and 1b.

In this work, we address a graph partitioning problem that arises when mapping the nodes of a DAG to the processing elements of a heterogeneous embedded multiprocessor. The processing elements (PEs) of this platform have a private local data memory and a separate program memory. A direct memory access controller is used to transfer data between the local memories and the external DDR memory of the system. The data memories have a size in the order of hundreds of kilobytes and can thus only store a small portion of the image. Therefore the input image is divided into *tiles*. The mode of operation of this hardware usually is that the nodes in the application graph are assigned to PEs and process the tiles one after the other. In most cases this can be pipelined such that while the PEs process the current tile, the direct memory access controller concurrently loads the next tile to the local memories and writes the processed tile from the previous iteration back to main memory.

However, this is only possible if the program memory size of the PEs is sufficient to store all kernel implementations. For the hardware platform under consideration it was found that this is not the case for more complex applications such as a Local Laplacian filter [18]. Therefore a gang scheduling [7] approach is used where the kernels are divided into groups of kernels (referred to as gangs) that do not violate memory constraints. Gangs are executed one after another on the target platform. After each execution, the kernels of the next gang are loaded. At no time any two kernels of different gangs are loaded in the program memories of the processors at the same time. Thus all intermediate data that is produced by the current gang but is needed by a kernel in a later gang needs to be transferred to external memory.

Since memory transfers, especially to external memories, are expensive in terms of power, the assignment of nodes to gangs is crucially important. There are many graph partitioning algorithms for the problem of dividing a graph into blocks of nodes under certain conditions [5]. However, in this case we require a strict ordering of gangs. Data objects may only be consumed in the same gang where they were produced and in gangs that are scheduled later. If this does not hold, there is no valid order in which the gangs can be executed on the platform. A valid order is a topological ordering of the graph that represents data dependencies between gangs. This graph can be created by taking the original DAG of the application and contracting all nodes that are assigned to the same gang into a single node. In order for a valid gang execution order to exist, the resulting graph therefore must be a DAG itself. An example for an incorrect assignment is shown in Figure 1c and a correct assignment in Figure 1d.

3 Preliminaries

In this section, we introduce the mathematical notation used throughout this paper, give the formal definition of the graph partitioning problem and show its relation to multiprocessor scheduling as a whole.

3.1 Basic Concepts

Let $G = (V = \{0, \ldots, n-1\}, E, c, \omega)$ be an directed graph with edge weights $\omega : E \to \mathbb{R}_{>0}$, node weights $c : V \to \mathbb{R}_{\geq 0}$, $n = |V|$, and $m = |E|$. We extend c and ω to sets, i.e., $c(V') := \sum_{v \in V'} c(v)$ and $\omega(E') := \sum_{e \in E'} \omega(e)$. We are looking for *blocks* of nodes $V_1, \ldots, V_k$

that partition V, i.e., $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. We call a block V_i *underloaded* [*overloaded*] if $c(V_i) < L_{\max}$ [if $c(V_i) > L_{\max}$]. If a node v has a neighbor in a block different of its own block then both nodes are called *boundary nodes*. An abstract view of the partitioned graph is the so-called *quotient graph*, in which nodes represent blocks and edges are induced by connectivity between blocks. The *weighted* version of the quotient graph has node weights which are set to the weight of the corresponding block and edge weights which are equal to the weight of the edges that run between the respective blocks.

3.2 Problem Definition

The partitions that we are looking for have to satisfy two constraints: a balancing constraint and a acyclicity constraint. The *balancing constraint* demands that $\forall i \in \{1..k\} : c(V_i) \leq L_{\max} := (1+\epsilon)\lceil \frac{c(V)}{k} \rceil$ for some imbalance parameter $\epsilon \geq 0$. The *acyclicity constraint* mandates that the quotient graph is acyclic. The objective is to minimize the total *cut* $\sum_{i,j} w(E_{ij})$ where $E_{ij} := \{(u,v) \in E : u \in V_i, v \in V_j\}$. The *directed graph partitioning problem with acyclic quotient graph (DGPAQ)* is then defined as finding a partition $\Pi := \{V_1, \ldots, V_k\}$ that satisfies both constraints while minimizing the objective function.

3.3 Relation to Scheduling

The balancing constraint ensures that the size of the programs in a scheduling gang does not exceed the program memory size of the platform and thus is an important constraint for scheduling. Reducing the edge cut reduces the amount of data transfers between gangs and thus improves the memory bandwidth requirements of the application. Note that an application is either compute-limited (processors are always occupied) or bandwidth-limited (processors wait for data). Thus a minimization of transfers does not guarantee an optimal schedule. However, especially in embedded systems, the memory bandwidth is often the bottleneck and a schedule requiring a large amount of transfers will neither yield a good throughput nor good energy efficiency [17]. Therefore, we address the problem of minimizing the edge cut under the given constraints in isolation and do not solve a scheduling problem in this work. We provide linear-time heuristics that can later be employed as subroutines in broader scheduling algorithms to reduce data transfers.

3.4 Related Work

There has been a vast amount of research on graph partitioning so that we refer the reader to [23, 4, 5] for most of the material. Here, we focus on issues closely related to our main contributions. All general-purpose methods that are able to obtain good partitions for large real-world graphs are based on the multilevel principle. The basic idea can be traced back to multigrid solvers for systems of linear equations [24] but more recent practical methods are based on mostly graph theoretical aspects, in particular edge contraction and local search. There are many ways to create graph hierarchies such as matching-based schemes [27, 14, 19] or variations thereof [1] and techniques similar to algebraic multigrid, e.g. [16]. We refer the interested reader to the respective papers for more details. Well-known software packages based on this approach include Jostle [27], KaHIP [22], Metis [14] and Scotch [6]. However, none of these tools can partition directed graphs under the constraint that the quotient graph is a DAG. We are not aware of any related work that is able to satisfy this constraint.

Gang scheduling was originally introduced to efficiently schedule parallel programs with fine-grained interactions [7]. In recent work, this concept has been applied to schedule parallel applications on virtual machines in cloud computing [25] and extended to include hard

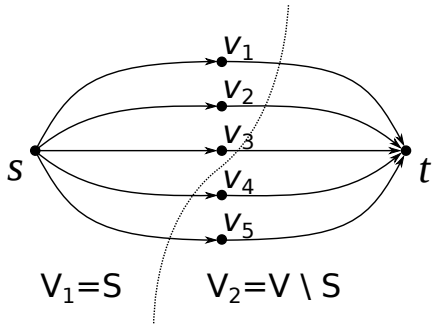


Figure 2 Reduction: subset sum problem is reduced to DGPAQ by creating a node for each a_i (the nodes in the center) and adding a source and sink node with edges as shown.

real-time tasks [10]. An important difference to our work is that in gang scheduling all tasks that exchange data with each other are assigned to the same gang, thus there is no communication between gangs. In our work, the limited program memory of embedded platforms does not allow to assign all kernels to the same gang. Therefore, there is communication between gangs which we aim to minimize by employing graph partitioning methods.

4 Hardness Results

In this section, we show that the problem under consideration is NP-complete when restricted to the case $k = 2$ and $\epsilon = 0$, and also hard to approximate with a finite approximation factor for $k \geq 3$. A given solution for an instance of DGPAQ can be verified in linear time by constructing the quotient graph $\mathcal{Q}$, checking the balance constraint and checking $\mathcal{Q}$ for acyclicity. The last task can be done in linear time in the size of $\mathcal{Q}$ using Kahn's algorithm [13]. We now reduce the subset sum problem to our problem. The proof is inspired by the reduction used in [20] which shows that the most balanced minimum cut problem is NP-complete.

▶ **Theorem 1.** *The DGPAQ problem is NP-complete for the bi-partitioning case with* $\epsilon = 0$.

Proof. We reduce the NP-complete [9] subset sum problem to DGPAQ. The decision version of the subset problem is stated as follows: Given a set of integers $\{a_1, \ldots, a_n\}$, is there a non-empty subset $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$ holds? The construction of an equivalent instance of DGPAQ is as follows: We construct a DAG $G = (V, E, c)$ with nodes $s, t \in V$ as well as a node $v_i \in V$ for each $i \in \{1, \ldots, n\}$. Then we set $A := \sum_i 2a_i$ and define the node weights as $c(s), c(t) := A$, $c(v_i) := 2a_i$. Afterwards, we insert edges (s, v_i) $\forall i$ and (v_i, t) $\forall i$. The graph is a DAG – an example topological ordering puts s first, t last and the remaining nodes at arbitrary positions in between. Figure 2 illustrates the construction. By definition, $L_{\max} = 3A/2$ for this instance of DGPAQ. Note that by construction A is divisible by 2. The construction can be done in polynomial time. Note that all balanced partitions $(S, V \backslash S)$ cut n edges, and due to the balance constraint s and t can never be in the same block. This ensures that there cannot be any edge (u, v) with $u \in V \backslash S$ and $v \in S$ and hence the quotient graph is acyclic. If the subset sum instance is a yes instance, then there is perfectly balanced bipartition and vice versa. ◀

The following theorem shows that it is not possible to find a finite factor approximation algorithm for our general problem where k is not a constant. The proof is a modification

of the proof by Andreev and Räcke [2] which shows this for the classical graph partitioning problem, i.e. no acyclicity constraint and for undirected inputs. Hence, we follow the proof of [2] closely with the difference being that the inputs that we construct are DAGs.

▶ **Theorem 2.** *The directed graph partitioning problem with acyclic quotient graph has no polynomial time approximation algorithm with a finite approximation factor for* $\epsilon = 0, k \geq 3$ *unless* $P = NP$.

Proof. The 3-Partition problem is defined as follows. Given $n = 3k$ integers $a_1, \ldots, a_n$ and a threshold A such that $A/4 < a_i < A/2$ and $\sum_i a_i = kA$, decide whether the numbers can be partitioned into triples such that each triple adds up to A. This problem is *strongly* NP-complete [9], i.e. the problem remains NP-complete if all numbers a_i and A are polynomially bounded.

Now suppose we have an approximation algorithm for the directed graph partitioning problem with acyclic quotient graph for $\epsilon = 0$. We can use this algorithm to decide the 3-Partition problem with polynomially bounded numbers. To do so, we construct a graph G that contains n subgraphs. Subgraph i has a_i nodes. All weights are set to 1. We make each of the subgraphs a directed clique, i.e. all edges (u, v) with $u < v$ are inserted into the subgraph. By construction G is a DAG. This is the main difference to [2] in which the subgraphs are undirected cliques. Also since all numbers are polynomially bounded, the construction takes polynomial time.

Now, if the 3-Partition instance can be solved, the k-DGPAQ problem in G can be solved without cutting any edge. Note that this solution also fulfills the acyclicity constraint. If the 3-Partition instance cannot be solved, then the optimum solution to the k-DGPAQ problem will cut at least one edge. An approximation algorithm with finite approximation factor has to differentiate between these two cases. Hence, it can solve the 3-Partition problem. ◀

5 Heuristic Algorithms

In this section we present simple yet effective construction and local search heuristics to tackle the problem. Our general approach is as follows: First create an initial solution based on a topological ordering of the input graph and then apply a local search strategy to improve the objective of the solution while maintaining both constraints. We start the section with the construction algorithm and then present different local search heuristics.

5.1 Construction Algorithm

All of our local search heuristics start with an initial partitioning that fulfills both constraints, i.e. the quotient graph is acyclic and the balance constraint is satisfied. Our algorithm does this by computing a random topological ordering of the nodes using a modified version of Kahn's algorithm with randomized tie-breaking. More precisely, the algorithm initializes a list S with all nodes that have indegree zero and an empty list T. It then repeats the following steps until the list S is empty: Select a node from S uniformly at random and remove it from the list. Add the node to the tail of T. Remove all outgoing edges of the node. If this reduces the indegree of another node to zero, add it to S. When the algorithm terminates, the list T is a topological ordering of all nodes unless the graph has a cycle. Using list T, we can now derive initial solutions by dividing the graph into blocks of consecutive nodes w.r.t. the ordering. Due to the properties of the topological ordering there is no node in a block V_j that has an outgoing edge ending in a block V_i with $i < j$. Hence, the quotient graph of our solution is cycle-free. In addition, the blocks are chosen

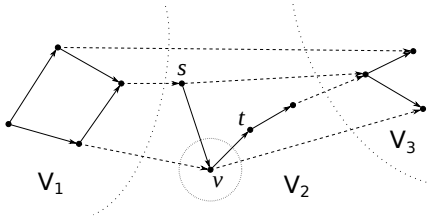


Figure 3 A DAG divided into three blocks. Internal edges are solid, external edges are dashed. Node v is a node that has non-zero internal and external cost for both C_{in} and C_{out}. Because of $(s, v) \in E \Rightarrow C_{in}(v, 2) > 0$, the node cannot be moved to V_1. Because of $(v, t) \in E \Rightarrow C_{out}(v, 2) > 0$, the node cannot be moved to V_3 either.

such that the balance constraint is fulfilled. There is obviously a large number of possible divisions. Our algorithm generates a balanced initial partitioning by dividing the ordering into blocks of size $\lfloor \frac{c(V)}{k} \rfloor$ or $\lceil \frac{c(V)}{k} \rceil$ uniformly at random. Since the construction algorithm is randomized, we run the heuristics ℓ times with different initial partitionings and pick the best solution afterwards.

5.2 Local Search Algorithms

Our local search heuristics take a given initial solution and move nodes between the blocks in order to decrease the edge cut. The reduction of the edge cut after a move is called the *gain* of the move. To compute the gain when moving node v, we define two functions:

$$C_{in}(v, i) := \omega(\{(u, v) \in E : u \in V_i\})$$
$$C_{out}(v, i) := \omega(\{(v, u) \in E : u \in V_i\})$$

Roughly speaking, C_{in} is the combined weight for all edges that start in nodes of block V_i and end in v. Analogously, C_{out} is the combined weight of all edges that start in v and connect to nodes in the block V_i. If $v \in V_i$, these costs are the weights of *internal* edges. These edges will become external edges and increase the objective if we move v to a different block. If $v \in V_j, j \neq i$, then these costs are weights of *external* edges, which will become internal and thus reduce the edge cut if v is moved to V_i. Figure 3 shows an example of internal and external edges.

We have multiple local search heuristics that differ in the size of the local search neighborhood: Simple Moves, Advanced Moves, Global Moves as well as FM moves. We found that the heuristics can often yield better results with a different initial partitioning. In order to compare the different heuristics, we will give each heuristic the same time budget and will restart the heuristics for different initial partitionings until it is exhausted.

5.2.1 Simple Moves (SM)

Simple moves start by picking a node v and moving it to a different block if this does not violate the constraints and improves the objective. Our simple move heuristic only considers to move a node $v \in V_i$ to adjacent blocks V_{i-1} and V_{i+1}. This is because there is a fast algorithm to check the *acyclicity constraint*. Assuming that the given solution is feasible with respect to both constraints, it is sufficient to check whether $C_{out}(v, i) = 0$ in the case that we want to move v to V_{i+1} and $C_{in}(v, i) = 0$ in the case that we want to move v to

V_{i-1}. The gain of a node movement depends on the block and is calculated as:

$$\begin{cases} C_{in}(v, i-1) - C_{out}(v, i) & \text{when moving } v \text{ to } V_{i-1} \\ C_{out}(v, i+1) - C_{in}(v, i) & \text{when moving } v \text{ to } V_{i+1}. \end{cases}$$

A block is eligible if the move does not create a cycle and does not overload the block. In addition, the gain has to be positive or zero but the balance of the partitioning is improved. If there is such a block, we move v to it. In the case that both blocks are eligible for the move and have the same gain, the heuristic selects one uniformly at random.

We repeat the process for all nodes. Our heuristic stops if there is no node with positive gain or balance cannot be improved. Hence, our heuristic terminates when a local minimum is found with respect to the local search neighborhood defined above. Note that even though the edge cut is not *strictly* monotonically decreasing, the combination of edge cut and difference in block weight is. In one pass, the heuristic considers the in- and outgoing edges of all nodes. Thus, each edge is considered exactly twice to calculate the gain for all nodes and the complexity of the heuristic is $O(m)$ per round.

5.2.2 Advanced Moves (AM)

This algorithm increases the local search neighborhood of the Simple Moves algorithm by considering more target blocks for a move. For the node $v \in V_i$ under consideration, all incoming edges are checked to find the node $u \in V_A$ where A is maximal. Also all outgoing edges are checked to find the node $w \in V_B$ where B is minimal. Since the original partition was obtained from a topological ordering, $A \leq i \leq B$ must hold, otherwise there would be back edges in the ordering and thus it would not be a topological ordering. If $A = i = B$, then the node v has in- and outgoing edges in its own block and cannot be moved. If $A < i$, then the node can be moved to blocks preceding V_i up to and including V_A in the topological ordering without creating a cycle. This is because all incoming edges of the node will either be internal to block V_A or are forward edges starting from blocks preceding V_A. Therefore it is still a topological ordering. However, when the node is moved to a block preceding V_A, the edge starting in this block becomes a back edge and the ordering is not a topological ordering anymore. Similar, if $i < B$, the node can be moved to blocks succeeding V_i up to and including V_B. Thus moving the node to V_j with $j \in \{A, \ldots, B\} \setminus \{i\}$ will preserve the topological ordering of blocks. This is a sufficient condition to ensure the acyclicity constraint and is not computationally expensive to check. However, since it is not a necessary condition, it might prevent the heuristic from testing some possible moves. The Global Moves heuristic does not have this limitation, but has a higher computational complexity.

The gain of the moves to all allowed V_j is computed with the cost functions described in the previous section as $C_{in}(v, j) - C_{out}(v, i) + C_{out}(v, j) - C_{in}(v, i)$. In each iteration, the move with the largest gain such that the constraints are maintained is selected. Tie-breaking and gains of zero are handled in the same way as in Simple Moves.

This heuristic considers each edge exactly twice in order to calculate the gain when moving the node to any other block. Afterwards, a block yielding maximal gain is selected, which can be done in time proportional to the degree of a node. Thus, the complexity of this heuristic is $O(m)$.

5.2.3 Global Moves (GM)

With this algorithm, we increase the local search neighborhood even further by considering all other blocks. Starting from the initial partition, the algorithm computes the adjacency

lists of the quotient graph. Throughout the algorithm the quotient graph is kept up-to-date. When moving a node we update the adjacency information of the quotient graph and record whether a new edge has been created. If this is the case we check the quotient graph for acyclicity by using Kahn's algorithm and undo the last movement if it created a cycle.

The calculation of the gain values can be done in $O(m)$ as for the other heuristics. For a node, the heuristic needs to check the acyclicity constraint for all considered moves/blocks in the worst case. Since Kahn's algorithm checks the quotient graph for acyclicity, the total complexity of this heuristic is $O(m(m_Q+k))$ where m_Q is the number of edges in the quotient graph. If the quotient graph is sparse, i.e. m_Q is $O(k)$, we get a complexity of $O(km)$.

5.2.4 FM Moves (FM)

This heuristic combines the quick check for acyclicity of the Advanced Moves heuristic with an adapted Fiduccia-Mattheyses algorithm [8] which gives the heuristic the ability to climb out of a local minimum. The initial partitioning is improved by exchanging nodes between a pair of blocks even if the gain is negative. The partition with the best objective that was seen during the pass will be returned. A pass starts with two blocks A and B, where A precedes B in the topological ordering of blocks. The algorithm will then calculate the gain for moving *enabled* boundary nodes to the other block. Using the same criterion to guarantee acyclicity as the Advanced Moves heuristic, we say that a boundary node is enabled if it is in A and does not have outgoing edges to nodes that precede B or it is in B and does not have incoming edges from nodes that follow A. The candidate moves, consisting of a gain and a node identifier, are inserted into a priority queue. The queue is a binary heap where the total order on the elements is implemented by comparing the gain of the moves and, if the gain is the same, a random number that is generated upon insertion.

In a loop that runs until the priority queue is depleted, the first move is extracted from the queue. If the selected move would overload the target block or is not enabled because it was disabled in a previous loop iteration, the heuristic continues with the next iteration. Otherwise, the move will be committed even if the gain is negative. The node is then locked, i.e. it cannot be moved again during this pass. This prevents thrashing and guarantees the termination of the algorithm. Unlike the Fiduccia-Mattheyses algorithm, a move in this scenario does not change the gain, it disables and enables other moves. For example, if a node w is moved from A to B, the heuristic will disable all nodes v in block B with $(w, v) \in E$ since they do not fulfill the condition for acyclicity anymore and moving any of them to A would introduce a back edge in the topological ordering of blocks. This does not necessarily mean that the quotient graph would become cyclic, however, assuring this would require a more expensive check like Kahn's algorithm. Note that the gain of the moves does not need to be re-calculated since w was locked and thus all nodes v will not be enabled again in this pass. On the other hand, moving w enables nodes in A if they are connected with an outgoing edge to w and if after the move they do not have other outgoing edges to blocks preceding B. The heuristic will calculate the gain for these nodes, enable and insert them into the priority queue. A move from B to A will enable and disable moves correspondingly. The loop will continue to move nodes between the blocks until the priority queue is depleted, which occurs when all nodes are either disabled or locked. Since the number of loop iterations is hard to predict due to the reinsertion of moves, it is limited to $2n/k$ which did not have a measurable impact on the quality of obtained partitionings. The best objective that was achieved in the pass is recorded. In the final step, the last moves are undone if required to reach the corresponding partitioning. This terminates the inner pass of the heuristic.

The outer pass of the heuristic will repeat the inner pass for randomly chosen pairs of blocks. At least one of these blocks has to be "active". Initially, all blocks are marked as "active". If and only if the inner pass results in movement of nodes, the two blocks will be marked as active for the next iteration. The heuristic stops if there are no more active blocks.

The overall time to compute gain values is $O(m)$. We now analyze the running time for a pair of blocks. In the worst case, all nodes of both blocks are enabled in the beginning and initializing the priority queue with $2n/k$ nodes requires $O(\frac{n}{k})$ time. Note that we cannot use a bucket priority queue, since the weights associated with the edges can be more or less arbitrarily distributed. Removing a node with the best gain from the queue takes $O(\log \frac{n}{k})$ time. If a move is committed in an iteration, the heuristic needs to calculate the gain of adjacent nodes. However, the heuristic will never calculate the gain of a move twice during a pass. Thus the total complexity of the inner pass is $O(\frac{n}{k} \log \frac{n}{k})$. Note that the inner pass needs to be performed for all pairs of blocks which yields overall time $O(m + m_{\mathcal{Q}} \frac{n}{k} \log \frac{n}{k})$ per round of the algorithm, or $O(m + n \log \frac{n}{k})$ if the quotient graph is sparse.

6 Experimental Evaluation

In this section we evaluate the performance of our algorithms. We start by presenting methodology and the systems we use for the evaluation. Then we evaluate the solution quality on small instances by comparing with the optimal solutions and evaluate our algorithms on complex imaging filters. We finish with testing the scalability of our algorithms.

Methodology. We have implemented the algorithms described above using C++. All programs have been compiled using g++ 4.8.0 and 32 bit index data types. The system we use is equipped with two Intel Xeon X5670 Hexa-Core processors (Westmere) running at a clock speed of 2.93 GHz. The machine has 128GB main memory, 12MB L3-Cache and 6×256KB L2-Cache. All instances described in this section will be made available on request.

Comparison with Optimal Solutions. This section compares the results of our heuristics against the optimal solution obtained by a non-polynomial time algorithm that performs an exhaustive search. We create a set of random graphs that are close to instances from typical applications. Our generation algorithm works by consecutively adding new graph levels with a random number of nodes. Each of the new nodes is connected to a random number of nodes in previous levels. Because the application domain of this work is imaging, we use a small number of input and output nodes (between one and three) which is typically the case for imaging and vision kernels (compare library of OpenVX vision functions [12]). Since the weight of nodes is representing the program size, we select a random value between the size of the smallest and the largest kernel in an implementation of the Local Laplacian filter for our target platform. The weight of edges is uniformly chosen between 1 and 100 to account for different sizes for intermediate buffers between the functions.

Because the following parameters have a major impact on the structure of the graph, we use two different values for each and generate 25 graphs for each of the eight resulting parameter combinations:

- The maximum size of a graph level is either set to a high value ($\sqrt{n}$) which results in a graph that can in extreme cases have $\sqrt{n}$ levels with about $\sqrt{n}$ nodes each, meaning that there is a high amount of data parallelism, and low values ($\sqrt[4]{n}$) such that the graph resembles more a long chain of nodes and thus represents the classical imaging pipeline with low data parallelism on kernel level.

Table 1 Each cell shows the averaged result of the heuristic for the current combination of block count k and imbalance ϵ. The value is the increase in cost compared to the optimal solution.

k	ϵ	SM	AM	GM	FM
2	20 %	3.41 %	3.41 %	3.41 %	0.26 %
	30 %	11.94 %	11.91 %	11.90 %	0.33 %
	40 %	14.71 %	14.78 %	14.58 %	1.29 %
	50 %	23.32 %	23.36 %	23.04 %	1.21 %
4	20 %	1.89 %	1.27 %	1.33 %	0.74 %
	30 %	4.03 %	3.22 %	3.25 %	0.67 %
	40 %	5.09 %	3.65 %	3.69 %	0.44 %
	50 %	6.50 %	4.04 %	4.19 %	0.31 %

Table 2 Table comparing the results of the manual implementation with the solution found by the heuristic.

	man.	SM	AM	GM	FM
number programs	20	22	16	16	17
number gangs	5	7	4	4	6
1-level edge cut	11,4	10,9	8,8	8,9	10,4
2-level edge cut	8,9	6,2	5,0	4,7	6,1
relative execution time	1,00	1,09	0,89	1,04	1,31

- The maximum number of edges is either set to the lowest number that ensures that inner nodes have at least one incoming and one outgoing edge and that the graph is connected or to $\sqrt{n}$ per node such that the number of edges scales with the problem size. This reflects applications with few and many data dependencies between functions.
- The maximum distance in terms of node indices, over which new nodes are connected to preceding nodes in the graph, is either set to a low value that results in a graph where nodes only have incoming edges from the closest preceding levels or it is set to n which means that there is no restriction on where edges can start. The first case models application where data is short-lived and only needed for the next step in a pipeline while the second case represents scenarios with a long data lifetime.

These 200 different problems instances were generated for problem sizes in the range of $n \in [10, \ldots, 20]$ nodes each. Table 1 shows the averaged approximation factor of the four heuristics when using a time budget of 10 milliseconds. The results show a good approximation of the optimal solution. The quality of SM, AM and GM degrades with large ϵ since they can get trapped in a local minimum, FM moves on the other hand shows a close and consistent approximation. The heuristics generally perform better on graphs that were created with more, unconstrained edges, presumably because there are more legal moves available of which the heuristic can pick the best one. We also found that the running time for a single pass of the heuristics is consistent across the instances while it varies drastically between milliseconds and several days for the exhaustive search. This emphasizes the need for a heuristic.

Local Laplacian Filter. The Local Laplacian filter is an edge-aware image processing filter. A detailed description of the algorithm and theoretical background is given in [18].

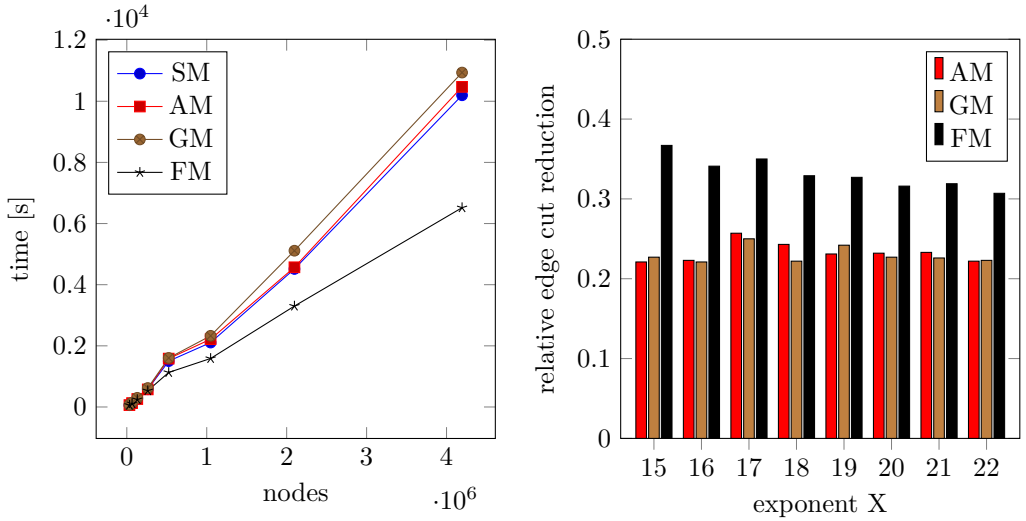
The algorithm uses concepts of *Gaussian pyramids* and *Laplacian pyramids* as well as a point-wise remapping function in order to enhance image details without creating arti-

facts. We model the data flow of the filter as a DAG where nodes represent simple function primitives, e.g. upsampling, downsampling and gaussian filtering for both image dimensions for each level of the pyramid generation. If there is a direct data dependency between two nodes, they are connected by an edge with a weight of the number of pixels of the corresponding buffer. The node weight is set to the program memory used by the primitive. The DAG has 72 nodes and 93 edges in total in our configuration. In an existing implementation, the primitives were grouped in a functional way (e.g. pyramid generation) by the developer into programs and then assigned to a total of five scheduling gangs. To evaluate the heuristics, we use a first pass with $L_{\max}$ set to the size of the program memory to find a good composition of function primitives into programs. The resulting quotient graph is then used in a second pass where $L_{\max}$ is set to the total number of PEs in order to find scheduling gangs that minimize external memory transfers. In this second step the acyclicity constraint is crucially important. In both passes, empty blocks were explicitly permitted to allow the heuristics to reduce the number of gangs. The time budget given to each heuristic is one minute. We also found that due to (desired) compiler optimizations, the final program memory size of a program can be smaller than the sum of its primitives. Since the entire process of partitioning, code generation and compilation is automated, we took advantage of this by slowly increasing ϵ until the programs became too large. The results are shown in Table 2. The 1-level edge cut shows the amount of communication between programs, the 2-level edge cut between gangs, both in megapixels.

If our heuristics are able to reduce bandwidth requirements of the application, the execution time will improve on all platforms where bandwidth is the limiting factor. Although this is the case for the majority of embedded platforms, we wanted to know what happens if we take bandwidth out of the equation. We obtained cycle counts for each program with a cycle-true compiled simulator of the hardware platform. Since context switching is synchronized, the longest execution time of a program in a gang equals the gang execution time assuming the programs never have to wait for data from external memories. The table shows the sum of this optimistic execution time for all gangs relative to the manual implementation.

All heuristics improve the edge cut by at least 30%, thus the schedule will be superior on all platforms where the manual implementation is bandwidth-limited. In addition, by reducing edge cut, the AM and GM heuristics find partitionings that require fewer gangs. For AM, this improves execution time, so the schedule will be superior even if bandwidth is not the limiting factor. For GM, the heuristic makes an additional choice that reduces edge cut even further, but does not balance compute-intensive programs well and thus execution time is not improved. In conclusion, we see a strong improvement in bandwidth demand and at the same time, with the exception of FM, an execution time that is on a par with or better than a manual implementation even if bandwidth is not a concern. We conclude that our pure edge cut reducing heuristics are a good starting point for the development of gang scheduling algorithms.

Random Geometric Graphs. We now look at the scalability of our heuristics. We do this on *random geometric graphs* where nodes represent random points in the unit square and edges connect nodes whose Euclidean distance is below $0.55\sqrt{\ln n/n}$. This threshold was chosen in order to ensure that the graph is almost connected. These graphs were taken from [3] and were initially undirected. We convert them into DAGs by directing edges from smaller to larger node ids. The graph rgg$X$ has 2^X nodes. We vary $X \in [15, \ldots, 22]$. The allowed imbalance was set to 3% since this is one of the values used in [26]. Figure 4 shows



(a) execution time averaged over 100 passes **(b)** edge cut relative to Simple Moves

Figure 4 Graph showing the execution time of each heuristic and the relative edge cut on directed random geometric graphs rggX.

the averaged time required for 100 passes of each heuristic and the relative improvement in edge cut that was found for $k = 8$ by the more advanced heuristics in comparison to the Simple Moves heuristic. The figure shows a linear growth in running time of our heuristics respective to number of nodes. The worst case complexity of FM moves was shown to be superlinear since it had to be assumed that all nodes are boundary nodes, which is not the case here. In fact, that FM only considers boundary nodes appears to improve the execution time compared to the other heuristics. We conclude that our algorithms scale well to large problems.

In another small experiment, we evaluated the quality of the solution found by the initial partitioning only. As expected, the best edge cut is always a fair amount larger than the one found by the heuristics, for example 29% compared to SM for the largest random geometric graph.

7 Conclusion

In this work we designed, implemented and evaluated new heuristics that partition streaming application graphs under constraints that are important for multiprocessor scheduling. It was shown that the constrained problem is NP-complete and that the heuristics yield good approximations of the optimal solution for small problem instances and have a linear growth for larger instances. In a simulation we could show the positive impact on communication volume of a real application for all heuristics and in one case even a reduction of execution time when bandwidth is not the limiting factor due to a reduction in number of scheduling gangs. The running time of the heuristics w.r.t. problem size is sufficient for this application domain, especially since the algorithms only need to run at compile-time. Also the communication volume was reduced by an extent that suggests that for future work it will be more rewarding to introduce additional objectives that help to improve gang execution time by better balancing compute-intensive kernels.

References

1 A. Abou-Rjeili and G. Karypis. Multilevel Algorithms for Partitioning Power-Law Graphs. In *Proc. of 20th IPDPS*, 2006.

2 K. Andreev and H. Räcke. Balanced Graph Partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

3 D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*, to appear.

4 C. Bichot and P. Siarry, editors. *Graph Partitioning.* Wiley, 2011.

5 A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent Advances in Graph Partitioning. In *Algorithm Engineering – Selected Topics,* to app., *ArXiv:1311.3144*, 2014.

6 C. Chevalier and F. Pellegrini. PT-Scotch. *Parallel Computing*, 34(6-8):318–331, 2008. doi:10.1016/j.parco.2007.12.001.

7 D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and distributed Computing*, 16(4):306–318, 1992.

8 C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *Proceedings of the 19th Conference on Design Automation*, pages 175–181, 1982.

9 M. R. Gary and D. S. Johnson. Computers and intractability: A guide to the theory of np-completeness, 1979.

10 J. Goossens and P. Richard. Optimal scheduling of periodic gang tasks. *Leibniz transactions on embedded systems*, 3(1):04–1, 2016.

11 Khronos Group. The OpenVX API. https://www.khronos.org/openvx/.

12 Khronos Group. The OpenVX Specification: Vision Functions. https://www.khronos.org/registry/OpenVX/specs/1.0/html/da/db6/group__group__vision__functions.html.

13 A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

14 G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

15 E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

16 H. Meyerhenke, B. Monien, and S. Schamberger. Accelerating Shape Optimizing Load Balancing for Parallel FEM Simulations by Algebraic Multigrid. In *Proc. of 20th IPDPS*, 2006.

17 P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 6(2):149–206, 2001.

18 S. Paris, S. W. Hasinoff, and J. Kautz. Local laplacian filters: edge-aware image processing with a laplacian pyramid. *ACM Trans. Graph.*, 30(4):68, 2011.

19 F. Pellegrini. Scotch Home Page. http://www.labri.fr/pelegrin/scotch.

20 J. C. Picard and M. Queyranne. On the Structure of All Minimum Cuts in a Network and Applications. *Mathematical Programming Studies*, 13:8–16, 1980.

21 E. Rainey, J. Villarreal, G. Dedeoglu, K. Pulli, T. Lepley, and F. Brill. Addressing system-level optimization with openvx graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 644–649, 2014.

22 P. Sanders and C. Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *Proc. of the 19th European Symp. on Algorithms*, volume 6942 of *LNCS*, pages 469–480. Springer, 2011.

23 K. Schloegel, G. Karypis, and V. Kumar. Graph Partitioning for High Performance Scientific Simulations. In *The Sourcebook of Parallel Computing*, pages 491–541, 2003.

24 R. V. Southwell. Stress-Calculation in Frameworks by the Method of "Systematic Relaxation of Constraints". *Proc. of the Royal Society of London*, 151(872):56–95, 1935.

25 G. L. Stavrinides and H. D. Karatza. Scheduling different types of applications in a saas cloud. In *Proceedings of the 6th International Symposium on Business Modeling and Software Design (BMSD'16)*, pages 144–151, 2016.

26 C. Walshaw and M. Cross. Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm. *SIAM Journal on Scientific Computing*, 22(1):63–80, 2000.

27 C. Walshaw and M. Cross. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. American Mathematical Society, 2007.