Automatic control flow generation for open VX graphs

van Son, S.W.H.

Award date:
2016

Automatic control flow generation for OpenVX graphs

Master Thesis

S.W.H. van Son
Student ID: 0726604
s.w.h.v.son@student.tue.nl

Graduation supervisor:	Prof. dr. H. Corporaal (ES)
Graduation tutor:	M. Popp M.Sc. (Intel Eindhoven)
Committee member:	Dr. ir. P.J.L. Cuijpers (SAN)

Master Embedded Systems
Eindhoven University of Technology
The Netherlands
May 30, 2016

Abstract

The active field of computer vision- and imaging for mobile platforms requires energy efficient high performance applications. These applications typically feature task- and data parallelism. This can be exploited in an energy efficient manner by using specialized hardware instead of general purpose processors. Although most mobile platforms contain such accelerators, this hardware is often not directly targetable by application developers and applications are typically not created with only one platform in mind, e.g. Android applications.

There are several industry standards, for example OpenCL, OpenCV and OpenVX, that enable usage of specialized hardware. In this thesis a platform template of a programmable image signal processor, consisting of multiple VLIW cores, a shared DMA and a scalar processor (SP) for control, is targeted by an application specified as an OpenVX graph. The advantage of OpenVX is that knowledge of the aggregation of functions into a graph also enables system-level optimizations.

A novel configuration time scheduling heuristic is proposed that exploits inter-processor communication (IPC) aggregation and kernel pipelining, yielding a fast, compact schedule. A control program is generated from this schedule, which will run on the SP. It takes care of loading the kernels and executing the schedule. Evaluation and comparison to alternative approaches shows that significant speedups can be achieved for very limited scheduling time budgets, i.e. without degrading the user experience.

Contents

Chapter 1

Introduction

In the field of computer vision- and imaging for mobile platforms, the combination of low power and high performance is very important. One of the main goals in embedded systems of today is to improve performance without increasing energy consumption. This can be done by exploiting parallelism and dividing execution over multiple cores, possibly running at a lower frequency through voltage-frequency scaling. Another common method is to use specialized hardware instead of general purpose processors, which can perform specific computations more efficiently, e.g. vector operations.

In computer vision, computational photography and imaging applications, the algorithms typically use stencil operations. These algorithms often allow for *tiling* the input data. Processing these tiles in parallel can be done to increase performance. By using programmable Image Signal Processors (ISPs), in the form of VLIWs with issue slots featuring vector operations, performance and energy efficiency are also increased. Imaging applications also often consist of a sequence of standard functions, such as color space conversion, filtering, scaling, bitwise operations or edge detection, that can potentially be executed in parallel or allow for pipelining. This task parallelism can also be exploited in case there are multiple processing elements (PEs).

To efficiently use specialized hardware, manual porting is required. This requires thorough knowledge of the platform. Even if this is possible, it makes porting applications to specialized hardware a cumbersome process. With the growing number of available platforms, e.g. Android apps target a variety of architectures and product generations, this is no longer practically feasible.

Another solution is to define a standard that provides a generic API for the developer for the construction of imaging applications. The hardware vendor can then create a library that implements the functionality of the API. Since the hardware vendor has thorough knowledge of the platform, he can optimize the implementation for the specific platform. This allows the application developer to utilize specialized hardware on all devices that support the standard. A well-known example of such a standard, that is adopted by the industry, is OpenCL.

OpenVX is a more recent standard that allows the application developer to specify a computer vision application as a graph of imaging functions, also referred to as *kernels*, from a predefined set. This application specifies the relation between input and output, but leaves the execution completely up to the OpenVX runtime. The OpenVX runtime can exploit kernel- and system-level

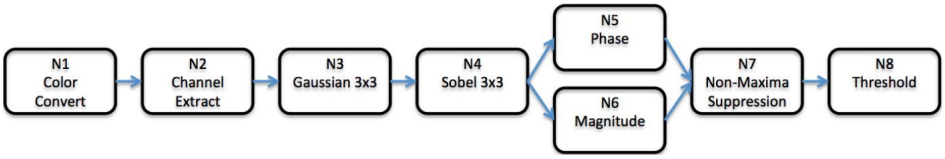


Figure 1.1: An example of an OpenVX graph [1] in which the intermediate data objects are omitted. The nodes represent an image function that is performed on the data and the arrows denote data dependencies.

optimizations, by targeting specialized hardware and applying optimizations, such as pipelining kernels and aggregating data transfers. These optimizations are not possible with other standards, like OpenCL and OpenCV, since kernels are executed on an individual basis.

An example of an application graph can be seen in figure 1.1. It consists of eight nodes, representing kernels. Node N1 takes one input image and N8 outputs the resulting image of the graph. In case these kernels are to be executed on specialized hardware, they need to be compiled online. This is normally not possible for platforms with multiple VLIW cores, because a compiler is typically not available and the compilation time would be unacceptably long. This means that the kernels should be available as precompiled binaries. Kernel-level optimizations at runtime are therefore not possible.

If there are multiple cores available it is obvious that kernels N5 and N6 of figure 1.1 can be executed in parallel. The advantage of knowing the whole graph before execution is that more parallelism can be exploited. For example, if N5 and N6 do not require the complete output of N4 to be ready, N4, N5 and N6 can be executed in a pipelined manner.

Besides enabling parallel execution, pipelining enables another important optimization; interprocess communication (IPC) aggregation. Since full images are typically too large to fit in local ISP memories, kernels read their inputs from external memory and write their output back to external memory. When executing in a pipelined fashion, the tiles of the images can fit in the local memories. This allows to avoid the roundtrip to external memory by writing from local- to local memory. First of all local memories have smaller latencies and a higher bandwidth (a factor 40 according to [2]), which results in faster transfers. A second advantage is that roughly a factor 40 is saved in energy consumption, according to the numbers for SRAM and DRAM accesses in [3]. Other optimizations and more detailed explanations of the possible optimizations are given in section 2.2.1.

When data transfers are processed by a shared DMA, performance depends on the ordering of transfers and kernel executions, i.e. the schedule. Also, the amount of transfers is dependent on the *mapping* of the nodes with a data dependency at a specific moment in time:

- When kernels are loaded at the same time on the same PE, the data can be kept in local memory, so no transfers are required.
- When kernels are loaded at the same time on different PEs, the data can be transferred from local to local memory.
- When kernels are not loaded at the same time, the intermediate data that is communicated between these kernels does not fit in the local memory. Therefore it must be written to external memory. Later, when the kernel

that requires this data as input is loaded, the intermediate data must be transferred from external memory to local memory again.

The **goal** of this thesis is to determine a schedule for a graph-based imaging application on a homogeneous multi-core accelerator with a shared DMA and limited local memory sizes. The quality of the schedule is measured by the *makespan* of the schedule of the complete execution of the graph. Since the platform (and sometimes the final application graph) are not known at compile time, mapping and scheduling can only be done at *configuration time*. Configuration time is the phase of execution of an application, in which the final graph is known but executing the graph has not yet started. For this reason the user is directly impacted by waiting times. To avoid degrading the user experience the scheduler must *always* find a valid schedule and transform this into actual execution of the graph, given a tight *scheduling time budget*.

The remainder of this section lists the contributions of this work and explains the structure of this thesis.

1.1 Contributions

The contributions of this thesis are:

- Design and implementation of a configuration time scheduling heuristic for generating a compact schedule for an acyclic imaging graph on a homogeneous multiprocessor platform with shared resources and limited local memories, optimizing for minimal makespan.
- A generic method for exploiting parallelism in graph-based applications, by using pipelining of sequential kernels, inter-process communication (IPC) aggregation and node (in)dependencies.
- Allowing application developers to target specialized hardware for their graph-based imaging applications, without requiring manual implementations for specific hardware.
- A proof of concept for the complete application-graph-to-control-program flow by simulation.
- Evaluation of the performance of the scheduling heuristic, compared to a naive list-scheduling approach and an approach similar to the state-of-the-art solution suggested in [4], which is based on tiling.
- A Satisfiability Modulo Theories (SMT) problem formulation for obtaining the optimal solution of the same solution space as the heuristic.

1.2 Structure of the report

This thesis starts with an explanation of the required background knowledge in chapter 2. This includes a description of the platform, graph-based imaging applications with OpenVX, theory and concepts of cyclo-static- and synchronous dataflow and scheduling. In chapter 3, a formalization of the problem is given together with the formal problem statement. Chapter 4 gives an overview of the related work.

The approach to this problem, in the form of a heuristic, is explained and motivated in chapter 5. The results of the heuristic are evaluated in chapter 6, where it is compared to alternative approaches. Also the design parameters

of the heuristic are evaluated here. In chapter 7 a conclusion is drawn and the future work is specified.

Chapter 2

Background

This section provides background information and theory for formulating the problem and solution. First the platform template, for which a schedule and mapping must be determined, is described. After that the concept of graph-based imaging applications, including an introduction to OpenVX, is given.

The required background knowledge on scheduling is summarized in section 2.3. Synchronous dataflow (SDF) will be used for modeling the graphs. In section 2.4 the semantics and useful properties of SDF are described, including a motivation for its usage and how OpenVX graphs and kernels can be modelled using dataflow.

In section 2.5 information on acceptable response times for end users is given and section 2.6 describes the big O notation that is used for describing algorithm complexity.

2.1 Platform template

The platform is based on the template shown in figure 2.1. It consists of:

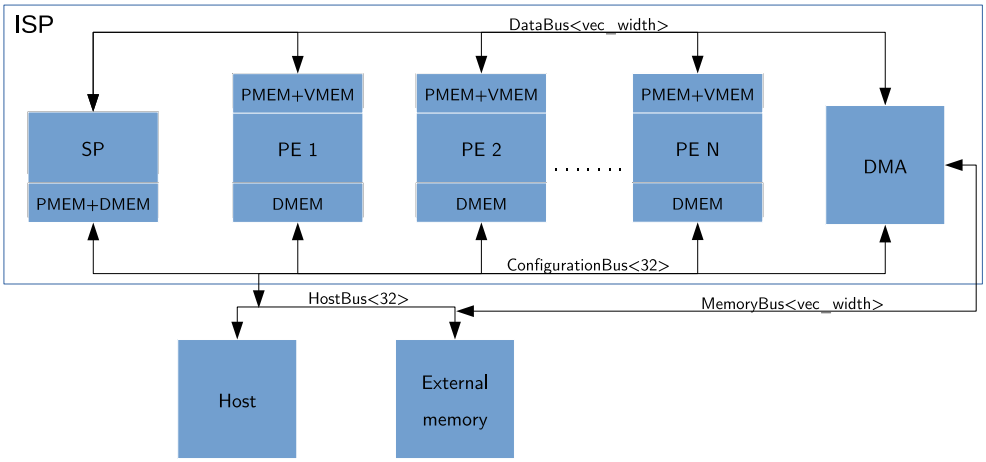


Figure 2.1: Platform template consisting of the host processor, external memory and the ISP. The ISP consists of multiple equivalent processing elements (PEs), a scalar processor (SP) for control and a DMA controller.

- A host processor with external memory. This external memory is large enough to store the buffers for inputs and outputs, but accessing it is 40x [2] slower and cost approximately 40 times more energy [3] than for the local PE memories.
- An Image Signal Processor (ISP) consisting of:
  - Multiple processing elements (PEs). These are VLIW cores that support vector operations and are used to process the image data. These PEs are identical, meaning that the execution time of a kernel is independent of the PE it is run on. Every PE features its own local vector memory (VMEM), in which input and output data are stored, and program memory (PMEM).
  - A single scalar processor (SP). This processor is capable of running a control program; a program that starts and synchronizes execution of kernels on the PEs and initiates and synchronizes data transfers between local memories and external memories. The scalar processor itself has a local data- and program memory.
  - A Direct Memory Access controller (DMA), which allows to perform the data transfers without involving a processor. The DMA is used for transferring input and output data, loading kernel binaries into the PE's program memory (PMEM) and can be configured over the configuration bus.

The connections between busses of different widths are made using bridges that transform streams of 32-bit data to vector width and vice versa.

2.2 Application graphs

Often applications can be specified as a graph of kernels, which are connected by data dependencies. A simple example of an imaging application that is specified as a graph, taken from the OpenVX launch presentation [5], is shown in figure 2.2. The rectangular blocks represent image processing functions and the edges denote the data dependencies. This section discusses the basics of OpenVX and the optimization opportunities it enables.

2.2.1 OpenVX

At the end of 2014 the open standard OpenVX has been released by the Khronos Group. The Khronos Group is a consortium that specifies open standards for accelerating various kinds of media processing. It is funded by its members from industry, which are for example NVIDIA, Apple, Samsung and Intel. Well known examples of standards developed by Khronos Group that have been widely adopted are OpenCL and OpenGL.

In the new standard, OpenVX, the main concept is the graph. An OpenVX graph is a directed acyclic graph (DAG) in which the nodes are instances of computer vision functions, called kernels. These kernels are either taken from a predefined set, of which the implementation is mandated by the standard (the base kernels), or are user kernels. Examples of base kernels are histogram computation, Canny edge detection, image pyramid computation as well as more basic kernels, such as color space conversion, color channel extraction, filters, integer arithmetic (multiplication, absolute difference) and bitwise operations.

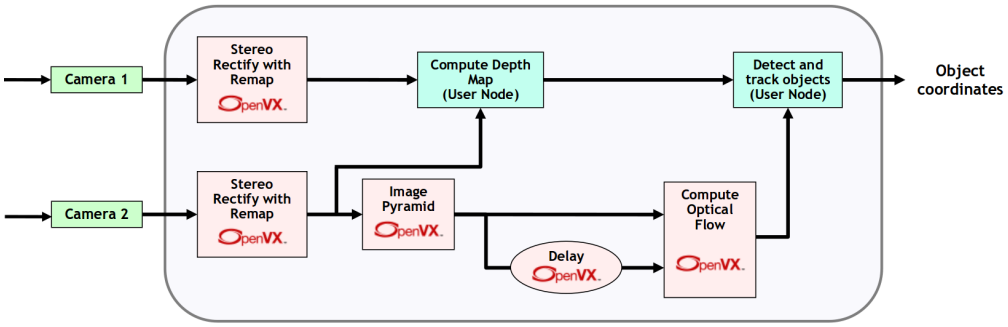


Figure 2.2: An example imaging graph for stereo machine vision. [5]

User kernels are part of the OpenVX framework to allow for more flexibility in application design. The developer has to deliver the implementation of the kernel, besides functions such as input and output validators. The *user kernel tiling* extension[1] allows users to specifiy the data dependencies of an output pixel. This can be used to determine if the user kernel can be tiled. However, since online compilation for the specialized hardware requires a compiler on the mobile device, running user kernels on the specialized hardware is excluded. OpenVX does not mandate anything about the targets of user kernels, but says they "will typically be loaded and executed on High Level OS/CPU compatible targets, not on remote processors or other accelerators". Nevertheless, if the developer implements the user kernel using for example OpenCL, it can still be accelerated.

The nodes are connected by intermediate data objects (not displayed in figure 2.2). OpenVX features three types of data objects; normal, virtual and delay. Normal objects should be present and accessible outside the graph scope. Therefore these data objects typically correspond to graph in- and outputs. In contrast, virtual data objects are opaque and cannot be accessed. Intermediate results that are not used outside the graph scope should be specified as virtual objects. This way they are susceptible for optimizations, such as tiling. Delay objects are data objects that are dependent on previous execution(s) of the graph.

The application developer has to follow three steps, for which functions are implemented in the OpenVX API; construct the graph, verify the graph and execute it. At the verification step the graph is not only verified, e.g. checking for matching kernel in and outputs, but also any preparation for execution, such as mapping and scheduling, takes place here. After verification, the graph becomes immutable. This means that if a change to the graph is made it needs to be reverified, before it can be executed again.

An important aspect of the OpenVX standard is that it does not specify any methodology or techniques that an implementation of the runtime must conform to, except for the functionality. This means that anything that can execute the kernel as specified can be a target, including fixed function hardware, ISPs, GPUs and OpenCL kernels. The OpenVX implementation, which is supplied by the hardware vendor, can process the data in any desired way as long as the

[1] https://www.khronos.org/registry/vx/specs/1.0/html/d0/d84/page_design.html#sub_node_tiling_ext, accessed at April 8th 2016

output is equal to the input processed by the specified kernels in the specified order.

Optimization opportunities

When kernels are executed individually, only *kernel-level* optimizations are enabled. Examples of this are running the kernel on an accelerator or replacing it by a highly-optimized version. OpenCV[2] is an example of a standard that enables individual execution of optimized versions of computer vision functions.

By specifying an application as a graph, also *system-level* optimizations are enabled. An overview of potential optimizations that OpenVX enables is given in [1]. An interesting subset is listed here to give an idea of the possibilities:

- **Aggregate function replacement**: If an efficient implementation exists for a subgraph of the OpenVX graph then multiple kernels can be replaced by a single, more efficient, kernel.
- **IPC aggregation**: Interprocess(or) communication can be reduced in many different ways. One of the main optimizations applied in this thesis is avoiding unnecessary transfers (e.g. roundtrips to external memory) by smart mapping decisions, that allow to keep data in local memory.
- **Compilation strategies**: In case online compilation is possible, compiler strategies such as loop unrolling and inlining can be applied for combinations of kernels.
- Executing **independent nodes** in parallel: Nodes for which there is no data dependency can be executed in parallel.
- **Tile processing**: When a specific part of the output only depends on a specific part of the input, the data can be divided into smaller tiles that can be processed by different PEs.
- **Pipelining**: Data can be passed through a pipeline of different functions if the data is tilable. Pipelining multiple invocations of a graph is currently not supported in OpenVX version 1.0.

2.3 Scheduling

This section starts with a concise description of different types of scheduling strategies in section 2.3.1. Since a compact, but high performance, schedule is desired, pipelined scheduling will be introduced here too, in section 2.3.2. Also an introduction to list scheduling is given here in section 2.3.3.

2.3.1 Scheduling strategies

Scheduling strategies range from fully-static to fully-dynamic scheduling. Fully-static scheduling is only possible if good estimates of execution times are available. This way a schedule can be determined before starting any execution. The advantage is that run-time overhead is reduced to a minimum.

The opposite of static scheduling is fully-dynamic scheduling in which all decisions are made at run-time. This results in high run-time overhead and

[2]OpenCV contains a library of more than 2500 optimized algorithms and its usage is widespread [6]. More information can be found on `opencv.org`.

implementation complexity, but is more flexible and therefore more generally applicable.

There are multiple strategies between fully-static and fully-dynamic. In the self-timed strategy only the processor assignment and ordering of executions on the same processor are fixed. Typically, first a fully-static schedule is determined, with the known execution time estimates, after which the exact start and end times of tasks are discarded.

A more detailed and extensive overview can be found in chapter 5 of [7].

2.3.2 Pipelined scheduling

In pipelined scheduling execution is divided into stages. Resources are assigned and schedules are derived for these stages. In the schedule these stages of different iterations can then overlap. An example of this can be seen in figure 2.3. In this figure a task consisting of three stages is executed in a pipelined manner.

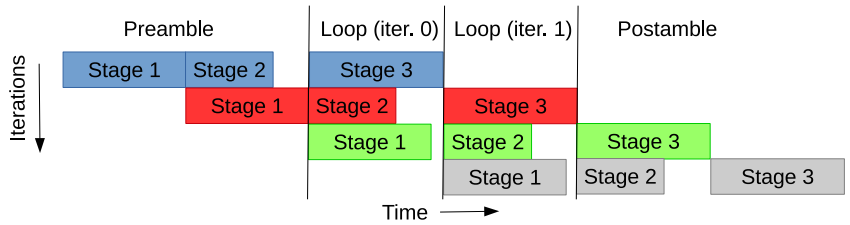


Figure 2.3: Example of pipelined execution. The preamble, postamble and two iterations of the repetitive loop body are shown here. Pipelining allows for overlapping the processing stages of different iterations.

It can be seen that all three stages of the different iterations can be executed in parallel. The point in the schedule where all three stages are executed in parallel can be continued with new iterations of the same task. This is part of the schedule is called the **loop body**. It is also sometimes referred to as the kernel, but this name is already used for OpenVX functions and therefore avoided in this work. The length of the loop body is referred to as the initiation interval or period and is equal to the length of the longest stage.

The parallelism achieved in the loop body cannot be achieved from the start. As can be seen in figure 2.3, it is required to execute stage 1 and 2 of the first task iteration and stage 1 for the second, to satisfy all dependencies of the loop body. This part of the schedule is referred to as the **preamble**, also known as the prolog.

Similarly to the preamble, the pipeline needs to be cleared after all iterations have been performed. In this example only stage 1 of the last iteration has been performed and only stage 1 and 2 of the second last iteration have been performed. Finishing the remaining stages is done in the **preamble**, or epilog.

The total execution time t_{total} is given by:

$$t_{total} = t_{pre} + (i - (s - 1)) \cdot t_{loop} + t_{post} \tag{2.1}$$

In which t_{pre}, t_{loop} and t_{post} are the execution times of the preamble, loop body and postamble, i is the number of iterations of the task and s is the number of

pipeline stages. With this equation the average time per task can be calculated:

$$t_{total}/i = \frac{t_{pre} + t_{post}}{i} + t_{loop} - \frac{(s-1) \cdot t_{loop}}{i} \tag{2.2}$$

What can be seen from this is that when the number of iterations of the task grows, the duration of the preamble and postamble become less important ($\lim_{i \to \infty} t_{total}/i = t_{loop}$).

When the stages share resources, for example a DMA for data transfers, parallel execution becomes more complex, since the length of the loop body can be dependent on the ordering of accesses. A well-known approach for determining pipelined schedules that can take this into account is **modulo scheduling**.

Modulo scheduling

Modulo scheduling is a framework that specifies a set of constraints that must be satisfied in order to obtain a modulo schedule [8]. In modulo scheduling the goal is to find schedule that can be repeated at regular intervals. This regular interval is called the initiation intervals (II) and corresponds to the period in pipelined scheduling. This means that the operations that are executed at t_1 and other operations executed at t_2 cannot use the same resource if $t_1 mod II$ equals $t_2 mod II$. The name "modulo scheduling" comes from this property. What this also means is that a valid modulo schedule is found if a schedule for a single II is found, such that all intra- and inter-iteration dependencies are satisfied.

This framework was originally intended for deriving instruction-level software pipelined schedules. The main difficulty in modulo scheduling is to derive a schedule for the repeating pattern. Since this problem is NP-hard ([7]) heuristics are normally used. In [8] a heuristic is presented, called Iterative Modulo Scheduling. It was designed for scheduling instructions given resource constraints.

Using modulo scheduling for task-level pipelined scheduling of SDF graphs on multiprocessors is done before ([9]). It identifies the requirement for explicitly scheduling data transfers and compares the speedup by a greedy heuristic to the exact solution of the SMT formulation. Task-level pipelining is often referred to as **coarse-grained** modulo scheduling.

2.3.3 List scheduling

In list scheduling a resource and start and end times are assigned to tasks as soon as they are **ready**. Ready means that all preceding tasks have an end time smaller than or equal to the start time of current task. If multiple tasks are ready, then a decision is made based on a priority. In list scheduling this priority list is a static ordering of all actors in the graph. In ready-list scheduling this static priority list is omitted and determined dynamically. The most well-known prioritization methods (discussed in chapter 6 of [7]) are the highest-level first with estimated times (HLFET), earliest task first (ETF) and the dynamic level scheduling (DLS) algorithm.

In HLFET [10] the tasks are sorted by their **level** in the graph. The level of a node is defined as the longest path, weighted by node execution times, to a sink node. In the ETF algorithm [11] the task that is scheduled at every step is the task that can start at the earliest point in time. This earliest start

time is influenced by multiple factors, which are the end time of preceding tasks and communication delays (mapping can be determined during scheduling). The DLS algorithm [12] also does not make any assumptions on mappings and reevaluates the priorities after every scheduling step. This dynamic ordering is based on the identification of the fact that the quality of the schedule also depends on the current scheduling state (besides the longest paths).

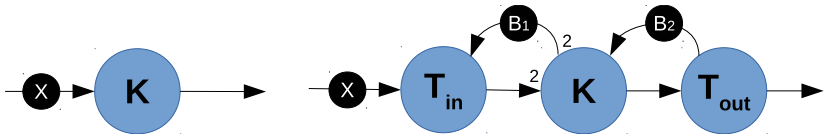
List scheduling can be combined with a predetermined processor assignment, i.e. mapping. This imposes limitations or completely omits processor assignment during start and end time assignment.

2.4 Synchronous dataflow

Synchronous Dataflow (SDF) is a concept introduced by Lee and Messerschmitt [13]. It is a model of computation that abstracts from functionality and is often used for analyzing and scheduling digital signal processing applications. A description of the fundamental knowledge for understanding the remainder of this thesis is given here. A formalization can be found in chapter 3.

An SDF graph consists of actors, often called **nodes**, and **edges**. An actor typically represents a computation and an edge models a FIFO channel. **Tokens** represent data objects and are produced and consumed by actors from and to edges. An actor can fire, i.e. execute, if there are enough tokens available on all of its incoming edges. When processing is done, tokens are produced on all of its outgoing edges. If actors have an execution time associated with them, *timed* SDF is considered. Timed SDF will be used throughout this thesis. The amount of tokens on an edge that is consumed and produced, depends on the consumption and production **rates**, associated with that edge. These rates are denoted by a number near the source or sink of an edge. If no rate is present, it is implicitly defined as one. If all rates are equal to one, the graph is called a Homogeneous Dataflow Graph (HDFG).

The behavior of OpenVX kernels, and the data dependencies when combined into imaging graphs (section 2.2), match the semantics of SDF. In its simplest form a kernel can be modelled as a single node, as in figure 2.4a. In this figure, X denotes the number of tokens. A token can represent a full input image, which means that X equals one. Executing node K then represents executing the full kernel (including data transfers), outputting a full processed output image.



(a) Basic dataflow model of a kernel.

(b) Line-based kernel model with DMA transfers

Figure 2.4: Modeling kernels in synchronous dataflow. X is the number of input tokens.

The model of figure 2.4a can also be used differently; node K can represent processing a line of input data and therefore a token represents a line of input data. In this case X equals the number of lines of the input image. Node K must be executed the same amount of times as there are input tokens, to produce the

full output image. This has the advantage of potentially enabling pipelined execution of multiple adjacent nodes or parallel execution of different iterations of the same node.

This model can be refined to the model of figure 2.4b, where the data transfers are no longer part of the kernel node, but modeled as separate tasks. The advantage of making this refinement in the model is that it explicitly allows to fire the actors (transfers and execution) in parallel. This also allows scheduling the individual data transfers, that might use a shared DMA and can therefore impact performance. Buffers are also modelled in this figure. This is done by adding backedges with an amount of tokens (B_1 and B_2 on it equal to the buffer size.

Cylco-static dataflow

SDF requires rates to be constant throughout execution. Sometimes this restriction precludes modeling the practical behavior. In that case cyclo-static dataflow (CSDF) can be used, which does not require rates to be constant. Instead of denoting the amount of produced or consumed tokens by a single rate near the sink or source of an edge, the rate is denoted by a sequence of rates between square brackets. As an example, [2, 0] at the sink of an edge means that the first execution of the sink node consumes two tokens and the second execution zero. After that it repeats, so the third execution consumes two tokens again.

Now consider a 3x3 filter. This can be implemented in a line-based manner; based on three input lines, $l_{in,i-1}$, $l_{in,i}$ and $l_{in,i+1}$, one output line, $l_{out,i}$, can be calculated. For calculating the next output line, $l_{out,i+1}$, the kernel requires input lines $l_{in,i}$, $l_{in,i+1}$ and $l_{in,i+2}$. As can be seen there is an overlap of two lines, line $l_{in,i}$ and $l_{in,i+1}$, which do not require to be transferred into an input buffer again. Effectively one new input line is required, and one input buffer can be released, which means that it can be modeled with an input rate of 1.

The problem is that this rate is not correct for the first and last iteration. For the first execution, where output line $l_{out,0}$ is calculated, input lines $l_{in,0}$ and $l_{in,1}$ need to be loaded into input buffers, indicating a rate of two. This is were CSDF is required. Figure 2.5 shows the cyclo-static dataflow model, which takes into account the following too:

- The last iteration of a node equals its input height minus one. For producing line $l_{out,h-1}$, where h is the image height, no additional input buffers are required, since $l_{in,h-2}$ and $l_{in,h-1}$ were already loaded for calculating output line $l_{out,h-2}$.
- Since the two input lines that are used for producing the first output line are also required for the second output line, no buffers are released on the backedge after the first iteration. After the last execution, the two input lines that were used for calculating it are released.

Since OpenVX graphs are acyclic, the number of executions of each node is finite and can easily be determined. The ellipsis, i.e. the three dots, in the rate indicate a constant value for the rate, equal to the value preceding and following it. This means that the first consumption rate of node K at the non-buffer edge equals two and the last execution consumes zero tokens. All executions in between consume one token.

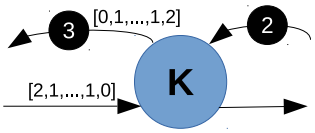


Figure 2.5: Modeling a filter kernel using CSDF. The number of rates in the rate sequence equals the total number of executions of the node. The ellipsis indicates a constant rate, equal to the values preceding and following it. Only the first and last iteration of node K have a different rate than one.

2.4.1 Repetition vector

An SDFG can be represented by its **topology matrix** Γ [13]. Every column corresponds to a different actor and every row corresponds to a different edge. In the topology matrix, every entry (i, j) corresponds to the number of tokens produced by actor j on edge i. If tokens are consumed by an actor, this value is negative. The topology matrix of an SDFG that has **consistent** sample rates has $rank(\Gamma) = |V| - 1$, where $|V|$ denotes the cardinality of the set of vertices (nodes). If this holds, the repetition vector q can be found by solving the equation $\Gamma \cdot q = \bar{0}$. The proof for this can be found in [13].

Following definition 3.1 of [7], a repetition vector q is defined as a column vector of size $|V|$, such that there is one entry for each actor. It has the property that if every actor i is fired $q(i)$ times, the number of tokens on every edge remains unchanged. Also the repetition vector is the smallest integer vector for which this holds.

Since OpenVX graphs are allowed to consist of multiple connected components (for the definition see section 3.1.1), the requirement $rank(\Gamma) = |V| - 1$ must hold separately for every connected component in the graph.

2.4.2 Retiming

One technique applied for obtaining high throughput schedules is called retiming. It is a technique that was originally used to optimize the throughput of cyclic HDFGs [14]. It does this by reducing the longest (in terms of node execution times) zero delay paths. The retiming relation is formalized in equation 3.1 in section 3.1.2. The retiming specifies an amount of executions for every node and relates this to the number of tokens on each edge. By applying a retiming, tokens are redistributed in the model. A good retiming results in a shorter period in which every node is executed $q(n)$ times, where q is the reptition vector of section 2.4.1.

In this thesis it is used with a similar goal, but since an OpenVX graph is acyclic it redistributes the tokens of the graph's input edges to the other edges. This can only be done when kernels process tiles, since the retiming cannot exceed the total number of executions of a node. Using the number of executions in the repetition vector, a loop body with a shorter period can be achieved by applying a retiming first. This means that the retiming can be interpreted as the number of executions of nodes in the preamble.

The advantage of using retiming for the preamble is that the number of tokens on every edge after the preamble, can be determined without actually deriving start and end times for the nodes' executions. A disadvantage is that

the schedule is limited to yield a **strictly periodic** schedule. This means that executions cannot overlap between different iterations of the loop body, i.e. they start and end in the same iteration.

2.4.3 Buffer sizing

Determining the required buffer sizes, to guarantee deadlock-freedom, is another aspect that can be analyzed using dataflow. Buffer size minimization is important, when there are limited local memories available for buffers. In [15] a polynomial time algorithm is described that minimizes buffer constraints for maximal throughput, which claims optimality. Later, in [16], it is shown that the algorithm does not guarantee optimality, by providing a counterexample. In fact, [16] proves that the problem is NP-complete. For this reason, an efficient algorithm that guarantees optimality does not exist. In [16] also the error of [15] is quantified. The overestimation of the buffer sizes is typically small, but the paper also claims that the exact solution is efficient in practice, most solutions found in a few hundred milliseconds.

2.4.4 Motivation

The SDF (and CSDF) semantics match the processes described in OpenVX graphs and are commonly used in signal processing applications. Execution of a kernel intuitively maps to an actor, edges are the input and output buffers and tokens represent the image data. Kernels can start execution only when sufficient input data and output buffers are available. Also, the processing is not data dependent. This makes (C)SDFGs suitable for modeling the behavior and interesting properties of imaging graphs for scheduling, as described in figures 2.4 and 2.5.

Deriving schedules for SDF graphs is a thoroughly researched topic. It makes use of the property of dataflow that execution is deterministic, modeling buffer capacities is straight-forward using backedges and deriving bounds on throughput can be done analytically. Over time many techniques have been presented such as graph unfolding and the usage of repetition vectors and retiming.

Limitations

By selecting (C)SDF as a model of computation also some restrictions are introduced. The main limitations are that execution times are fixed. This means that, if no good approximations can be made for execution times of nodes, schedule quality is decreased.

Another assumption that was made in this section is that kernels can be implemented as a (multi)line-based kernel, when adding transfers as in figure 2.4b. When looking at the list of the OpenVX kernels of specification v1.0 it seems to hold for all kernels, except for histogram equalization, (Lukas-Kanade) optical flow computation, remapping and warping. For the statistical functions (mean, standard deviation, min-max locations) only the input can be pipelined, since their outputs are scalar values.

2.5 User experience

A topic that is not exact, but which is important in this thesis, is the theory of end user acceptance of response times. [17] states that 10 seconds is the amount of time that the user's attention is kept. When considering a mobile application implementing an OpenVX graph, this means that the amount of time available for scheduling is in the order of a few seconds.

There are cases where a graph is fixed and once a schedule is found it can be stored on the mobile device and reloaded the next time the application is loaded. However, this is not always the case. Especially in cases where the end users can adapt the graphs, for example by having variable input sizes or options that alter kernel parameters, the time required for scheduling becomes important. Remember that every time an OpenVX graph is altered it must be reverified and therefore potentially rescheduled.

2.6 Big O notation

When designing algorithms an interesting property is the growth rate of the algorithm. It relates the size of the input to the execution time of the algorithm and allows for classifying and comparing algorithms. This classification is typically formulated using the big O notation. The growth rate is also called the *order* of a function and for this reason the symbol O is used.

The big O notation has the following meaning; $f(x) = O(g(x))$ means that there exists a constant value C and a value for x_0, such that forall $x \in [x_0, \infty]$ it holds that $|f(x)| \leq C * |g(x)|$. In other words this means that $g(x)$ is an upper bound to the growth rate of $f(x)$.

Of course it is desired to keep the growth rate as small as possible. Some time complexity names, increasing in complexity, are constant time ($O(1)$), logarithmic time ($O(log(n))$), linear time ($O(n)$), polynomial time ($O(n^k)$ for any non-negative integer value of k) and exponential time ($O(2^{poly(n)})$, where $poly(n)$ is a polynomial function of n).

Some properties, which can be easily proven given the definition above, for simplifying expressions using the big O notation are:

- In case the function is a sum of other functions the growth rate equals the largest growth rate of these functions; $x^2 + 100x = O(x^2)$.
- $O(k \cdot g(x)) = O(g(x))$ for every non-zero value of k, follows directly from the definition.
- The product rule for the big O notation yields $f(x) = O(g_1(x)) \wedge h(x) = O(g_2(x)) \Rightarrow f(x)h(x) = O(g_1(x)g_2(x))$.

Chapter 3

Problem formulation

This chapter gives a formal description of the used concepts and constraints of the problem in section 3.1. An overview of the problem and the goal of this work is given afterwards in section 3.2.

3.1 Formalization

This section gives an overview of the concepts that are used to describe the problem and formalizes them. In the remainder of this section $\mathbb{N}$ is the set of non-negative integers.

3.1.1 Directed graphs

A directed graph is a tuple (V, E) where V is a set of vertices, often referred to as *nodes*, and E is a set of *directed* edges. An edge is a tuple (u, v) with $u, v \in V$, which indicates an edge from **source** node u to **sink** node v. For an edge $e = (u, v)$, the source and sink vertices are denoted by respectively $src(e)$ and $snk(e)$. Furthermore, e is called an **input edge** of vertex v if $snk(e) = v$ and it is called an **output edge** of v if $src(e) = v$. If there exists an edge (u, v) then vertex u and v are **adjacent**. When edges do not have a source node they are referred to as **source edges** and similarly **sink edges** are defined as edge that do not have a sink node. These are the model input and output buffers. All edges that are not sink or source edges are called **internal** edges.

A **subgraph** $G' = (V', E')$ is a directed graph such that $V' \subseteq V$ and every $e \in E$ for which $src(e) \in V' \vee snk(e) \in V'$ holds is in E'. A subgraph can be denoted by its subset of vertices as $subgraph(V)$.

A **path** in a directed graph is a finite sequence of edges $e \in E$, such that $src(e_i) = snk(e_{i-1})$ for all e_i in $(e_1, e_2, ..., e_n)$ and $1 < i \leq n$. A **cycle** is a path $(e_i, ..., e_n)$ where $src(e_i) = snk(e_n)$. A graph is a **directed acyclic graph** (DAG) if it is a directed graph in which there exist no cycles. We say that a vertex u **precedes** vertex v if there exists a path from u to v.

A **chain** is a sequence of vertices $(v_1, ..., v_n)$, where every vertex v_i and its subsequent vertex v_{i+1} in the sequence are adjacent. A graph is **connected** if there exists a chain between every vertex pair $u, v \in V$. A **connected component** of graph $G = (V, E)$ is a subgraph $G' = subgraph(V')$ in which subset

$V' \subseteq V$, such that there does not exist a chain between any vertices $u \in V'$ and $v \in V \backslash V'$.

3.1.2 Synchronous Dataflow

Synchronous dataflow has been formalized often before (e.g. in chapters 3 and 4 of [7]). In *timed* SDF a notion of time is added in the form of execution times. A timed SDFG can be defined as a tuple (V, E, p, c, d, t) where:

- V is the set of vertices, which in SDF are referred to as nodes or actors.
- $E \subseteq V \times V$ the set of directed edges.
- The valuation $p : E \to \mathbb{N}$, where $p(e)$ represents the number of tokens produced by the source node of an edge e.
- The valuation $c : E \to \mathbb{N}$, where $c(e)$ denotes the number of tokens consumed by the sink node of edge e.
- The valuation $d : E \to \mathbb{N}$ represents the number of initial tokens on edge e, which are called delays.
- The valuation $t : V \to \mathbb{N}$ gives the actors execution time of actor v.

An execution of an actor is called a **firing**. An actor n can fire if on all its input edges e_i there are $c(e_i)$ tokens available. The actor consumes these tokens when firing, i.e. the tokens are removed from the input edges. After $t(n)$ time units, execution is done and $p(e_o)$ tokens are produced on all of the actor's output edges e_o.

The retiming of an SDF graph [14] is also a column vector of size $|V|$, which specifies the number of executions for each node. This gives a relation for the number of delays on each edge e:

$$d(e) = d_0(e) + p(e)r(src(e)) - c(e)r(snk(e)) \tag{3.1}$$

In which $d_0(e)$ denotes the initial tokens at $t = 0$.

Cyclo-Static Dataflow

In cyclo-static dataflow (CSDF) rates do not have to be constant, but can be dependent on the firing iteration as described in section 2.4. If for a node a sequence of three rates $r = [r_1, r_2, r_3]$ is specified, then it means that iteration i of that node consumes/produces $r[i \mod |r|]$ tokens, where $|r|$ denotes the number of rates in sequence r. We denote the number of consumed or produced tokens for iteration i of the corresponding node as $c(e, i)$ and $p(e, i)$, respectively.

For reasons of conciseness $p(e)$ is used in the remainder of this chapter. If there are differences between SDF and CSDF, then this is explicitly noted. An example where this is the case is in the retiming relation of equation 3.1. This now becomes:

$$d(e) = d_0(e) + \sum_{i=0}^{r(src(e))-1} p(e, i) - \sum_{i=0}^{r(snk(e))-1} c(e, i) \tag{3.2}$$

3.1.3 Communication unaware kernel graph

A communication unaware kernel graph (CUKG) is the input for the problem. It is a (C)SDF graph with an extra valuation $type : V \to kernel$, which denotes which kernel it implements. An edge in a CUKG has two valuations, $p_{in} : E \to \mathbb{N}$

and $p_{out} : E \to \mathbb{N}$, that denote the input and output port of respectively the sink and source node to which the edge is connected. It has some restrictions, based on the properties as specified in the OpenVX graph formalization [1]. The CUKG contains no actors representing communication, only actors for processing the imaging kernels.

For a CUKG to be valid it must hold that:

- It is a directed acyclic graph.
- All its connected components are *consistent*.
- Every node v is of a specific kernel type $type(v)$. The following properties have a value that is dictated by the kernel type:
  - Number of input edges and their corresponding input rates.
  - The output rates. Output edges do not have to be present for all outputs of a kernel, since they might not be used.
  - Execution time.
  - Kernel size. The amount of program memory the binary would occupy.
- Only one edge can be connected to the same input port.

3.1.4 Transformation from a CUKG to a CAKG

To analyze the problem the CUKG is transformed into a communication aware kernel graph (CAKG). In this transformation, actors representing DMA transfers are added with their corresponding edges. The transformation from CUKG G to CAKG G' is formally defined as follows:

- For every node $v \in V$ of G there is a corresponding node $v' \in V$ of G', with $t(v') = t(v)$ and $type(v') = type(v)$. We denote the relation between v and v' as $v_{cor}(v) = v'$ and say v' **corresponds** to v. We define $v_{cor}(\varnothing) = \varnothing$. We call these added nodes **processing nodes**, because they represent a PE processing data, and we denote the set of processing nodes as V_p.
- For every edge $e \in E$ of G a node v' is added to V of G', with $t(v') = t_{t,in}(type(snk(e)), p_{in}(e)) = t_{t,out}(type(src(e)), p_{out}(e))$, where $t_{t,in}$ and $t_{t,out}$ denote the time it takes to transfer one input buffer. These added nodes are called **transfer nodes** because they represent a DMA transfer. The set of transfer nodes is denoted as V_T.

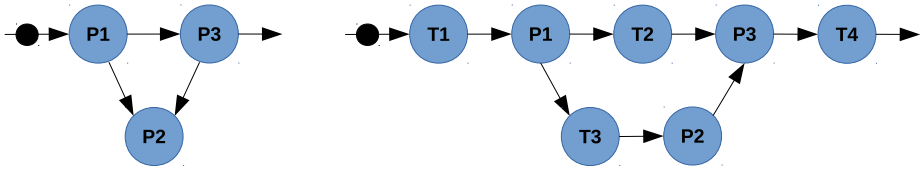
  For this edge $e \in E$ of G, also two edges $e_1, e_2 \in E$ of G' are added:
  - $e_1 = (v_{cor}(src(e)), v')$ with $p(e_1) = p(e)$ and $c(e_1) = 1$.
  - $e_2 = (v', v_{cor}(snk(e)))$ with $p(e_2) = 1$ and $c(e_1) = c(e)$.

  All added edges have zero initial tokens on them, i.e. $d_0(e_1) = d_0(e_2) = 0$, except for source edges. Source edges hold the graphs input data and therefore $d_0(e_1) = d(e)$.

  In case it holds that, for an edge e, both $src(e)$ and $snk(e)$ are mapped to the same PE, then data can be kept in local memory and no data transfers are needed. Hence, no transfer node is added for this edge and an edge e_1, equal to the CUKG e is added to the CAKG.

A simple example of a transformation from a CUKG to a CAKG is shown in figure 3.1. In this situation node P2 and P3 are mapped to the same PE and node P1 to a different PE. Therefore no transfer is present between node P2 and P3.

[1] https://www.khronos.org/registry/vx/specs/1.0/html/d0/d84/page_design.html#sub_graphs_rules



(a) The CUKG **(b)** The corresponding CAKG

Figure 3.1: CUKG to CAKG transformation. In this case all processing nodes are mapped to different PEs. P- and T-nodes represent processing and transfer nodes, respectively. Node P2 and P3 are mapped to the same PE and node P1 to another PE.

3.1.5 Platform template

We formalize the platform template as a tuple (Π, Φ), where Π is the set of PEs and Φ is the set of DMAs. Transfer nodes can only be mapped to DMAs and processing nodes can only be mapped to PEs. The available local (vector) memory for buffers on a PE is denoted by mem_{PE} and the available program memory on a PE is denoted by $pmem_{PE}$. Similarly mem_{SP} and $pmem_{SP}$ correspond to the memory sizes of the scalar processor.

3.1.6 Schedule

For defining schedules, the schedule functions $start(v,k)$ and $sync(v,k)$ are used. They denote the *start* and *synchronization* time of the k^{th} execution, or *iteration*, of actor v. Following the dataflow semantics it must hold that $sync(v,k) \geq start(v,k) + t(v)$. Synchronization is blocking and therefore variations in execution time are allowed.

According to [18], a schedule is *admissible* if for every edge $e = (u,v) \in E$ the following condition is satisfied:

$$start(v,k) \geq sync(u, \left\lceil \frac{(k+1)c(e) - d(e) - p(e)}{p(e)} \right\rceil) \tag{3.3}$$

Rederiving this relation for CSDF cannot be simplified to the same compact form as equation 3.3, but yields:

$$\sum_{i_u=0}^{k_u} p(e,i_u) \geq \sum_{i_v=0}^{k_v} c(e,i_v) - d(e) \rightarrow start(v,k_v) \geq sync(u,k_u) \tag{3.4}$$

Since the CAKG is acyclic, there is a limit to the value of k for every node v. We denote this maximum iteration value as $k_{max}(v)$. With this limit we can define the **makespan** T of the schedule as:

$$T = \max_{v \in V} sync(v, k_{max}(v)) \tag{3.5}$$

We define the *mapping function* $\sigma : V \rightarrow \Pi \cup \Phi$, for which $\sigma(v)$ denotes the hardware that node v is mapped to. Since there are no edges in the CAKG for modeling a shared resource it must be explicitly noted that there is no overlap of task executions on the same hardware:

$$\begin{aligned} \forall v_1, v_2 \in V,\ k_1 \in [0, k_{max}(v_1)], k_2 \in [0, k_{max}(v_1)] : v_1 \neq v_2 \wedge \sigma(v_1) = \sigma(v_2) \mid \\ start(v_1,k_1) \geq sync(v_2,k_2) \vee start(v_2,k_2) \geq sync(v_1,k_1) \end{aligned} \tag{3.6}$$

3.1.7 Instructions

The schedule must be transformed into a list of instructions that is used for executing the graph. An instruction is a tuple $(type, v, k)$, with the valuation $type : I \to \{load, start_P, sync_P, start_T, sync_T\}$. These types have the following meaning:

- *load* represents the activity of loading a kernel.
- $start_P$ represents the activity of starting the execution of a kernel.
- $sync_P$ represents the activity of waiting for a PE to become idle, i.e. waiting for an execution to be finished if it is not already.
- $start_T$ represents the activity of starting a transfer.
- $sync_T$ represents the activity of waiting for a DMA transfer to be finished.

The valuation $v : I \to V$ indicates the node to which the instruction corresponds and $k : I \to \mathbb{N}$ denotes the iteration of the instruction starting at 0, so $k(i) = 2$ indicates that this is the third execution of this type for actor v.

The transformation from a schedule to an ordered sequence of instructions is defined as follows:

- For every node v and every iteration $k \in [0, k_{max}(v)]$ there are the instructions:
  - If v is a *processing node* then two present instructions:
    - i_1, with $type(i_1) = start_P$, $v(i_1) = v$ and $k(i_1) = k$.
    - i_2, with $type(i_2) = sync_P$, $v(i_2) = v$ and $k(i_2) = k$.
  - If v is a *transfer node* then two present instructions:
    - i_1, with $type(i_1) = start_T$, $v(i_1) = v$ and $k(i_1) = k$.
    - i_2, with $type(i_2) = sync_T$, $v(i_2) = v$ and $k(i_2) = k$.
- For every processing node there is one *load* instruction i, with $type(i) = load$, $v(i) = v$ and $k(i) = 0$.

We denote the position of an instruction i with $type(i) = t$, $v(i) = v$ and $k(i) = k$ in a sequence of instructions L as $pos(t, v, k)$. The length of instruction list L, i.e. the highest defined value of $pos(t, v, k)$, is denoted by $length(L)$. The list of instructions is valid if:

- All instructions adhere to the order of an admissible schedule:

$$event(v_1, k_1) < event(v_2, k_2) \to pos(t_1, v_1, k_1) < pos(t_v, v_2, k_2) \tag{3.7}$$

  Where *event* can be replaced by any combination of $start_T, start_P, sync_T$ or $sync_P$.
- A kernel must always be loaded before it can be started:

$$pos(load, v, 0) < pos(start_E, v, k) \tag{3.8}$$

3.1.8 Buffer sizes

The required buffer size can be formalized as the maximum number of buffers that is used on each edge during execution. Since every edge corresponds to an input, output, or inout buffer we use the edge to identify the buffer. We denote the number of used buffers on edge e after executing the instruction at position i in an instruction list as $used(e, i)$.

All buffers are initialized with their initial tokens, so $tok(e, -1) = d(e)$. The number of used buffers can be determined for all instructions that are present in list L:

- When starting to process or transfer data, buffer space for the output must be available. Therefore the used buffer size equals the used buffer size before starting the instruction plus the produced tokens.

$$\begin{aligned} start_X(src(e),\, k) \in L \wedge idx = pos(start_X, src(e), k) \\ \rightarrow used(e, idx) = used(e, idx-1) + p(e) \end{aligned} \tag{3.9}$$

  Where $start_X$ is either a $start_T$- or $start_P$-type instruction.
- When a PE is done processing or a DMA is done transferring data, the processed input buffers are no longer needed. This means that, after synchronization, the number of used input buffers is decreased by the number of consumed tokens.

$$\begin{aligned} sync_X(snk(e),\, k) \in L \wedge idx = pos(sync_X, snk(e), k) \\ \rightarrow used(e, idx) = used(e, idx-1) - c(e) \end{aligned} \tag{3.10}$$

  Where $sync_X$ is either a $sync_T$- or $sync_P$-type instruction.

If none of these rules apply then $tok(e, i) = tok(e, i-1)$.

The required buffer sizes for every edge $bs(e)$ can then be described by the maximum value:

$$bs(e) = \max_{0 \leq i < length(L)} tok(e, i) \tag{3.11}$$

The total memory requirement for a PE is then the sum of all its input and output buffers, where we denote size of a token on an edge e in bytes as $size(e)$. so the *memory requirements* for a processing element p is:

$$mr(p) = \sum_{\{e \in E \mid (src(e)=v \vee snk(e)=v) \wedge \sigma(v)=p\}} bs(e) \cdot size(e) \tag{3.12}$$

3.2 Problem statement

Given a platform P and an application in the form of a Communication Unaware Kernel Graph G find a schedule such that:

- The schedule is **admissible** (equations 3.3 and 3.4).
- The buffers fit in the **local memories** of the processing elements:

$$\forall p \in \Pi \mid mr(p) \leq mem_{PE} \tag{3.13}$$

- The kernels loaded at the same time on a processing element must fit in the **program memory** of that processing element.

The objective is to minimize the makespan of the graph (equation 3.5). Optimality can be formalized as a solution for which it holds that:

$$\max_{v \in V} sync(v, k_{max}(v)) = \min_{s \in S} \max_{v \in V} sync(v, k_{max}(v)) \tag{3.14}$$

Where S is the set of all admissible schedules. The schedule must be transformed into a list of instructions (as defined in section 3.1.7). Besides these constraints on the schedule it is required that:

- The list of instructions is executed by a **control program** running on the scalar processor. This means that the schedule must fit in the SP's local memory.

- It is **guaranteed** that a solution is found. Being unable to find a schedule means that the application cannot be executed at all and this clearly means failure.
- The schedule is derived **at configuration time**. This results in a constraint on the execution time of the scheduler since the user has to wait for it (see section 2.5). Since execution time of the scheduler is dependent on the device that it is executed on, this will come in the form of a scheduling **time budget**.
- **Online compilation** of kernels for the VLIWs is excluded. This is the case because (i) running the compiler for the ISP of this thesis on a mobile device is not practically feasible, and (ii) compiling takes too much time from a user perspective ($\gg$ 10 seconds).

Chapter 4

Related work

Besides the references in the background section of this thesis, literature has been studied concerning the topics in this chapter. The work that is related to the automatic scheduling of graphs, is described in section 4.1. Also, papers concerning partitioning approaches have been studied (section 4.2).

4.1 Automatic scheduling of graphs

Already in 1989, Printz e.a. proposed methods to automatically schedule graph-based signal processing applications on parallel architectures [19], in their case the Warp platform. It identifies three *parallel styles*, which are systolic, data partitioned and serial. *Systolic* corresponds to pipelined execution and *serial* corresponds to executing independent kernels in parallel. Data partitioned execution means that the input data is partitioned into *tiles*. These tiles are then processed in parallel on multiple PEs.

However, the Warp platform did not have *communication agents* (nowadays DMAs), and letting the (VLIW) cores handle communication was very inefficient. Therefore, the implementation only applied the data partitioned execution style (when the serial style is not mandated by the function). This resulted in low processor utilization when the number of processors was increased [20], because the communication bandwidth did not scale with the number of cores.

The systolic style was included later in the implementation of a FORTRAN compiler for the iWarp platform [20]. The compiler made decisions on the parallel style, based on user hints and compiler analysis. If the systolic style was chosen, then communication resources were dedicated to the task to achieve good performance.

Tiling for OpenVX graphs

In the work of [4] the data partitioned style of [19] is used for exploiting data parallelism in OpenVX graphs on cluster-based many-core accelerators. The OpenVX runtime is implemented using OpenCL. One of the key bottlenecks that is identified in this work is the bandwidth of the external memory.

The platform that is used is the STHORM SoC, which has multiple clusters (although only one cluster is used in testing) consisting of 16 Multiple Program Multiple Data (MPMD) processing elements. Every cluster has one local (L1)

memory and a dual-channel DMA. These clusters are connected to external memory by an asynchronous bus.

The tile sizes are chosen such that the required buffers fit in the local memories of a cluster. The tile sizes are determined for multiple consecutively executed kernels. For example, for calculating a $w \times h$ output tile for a 3x3 box filter, the input tile must have a size of $w \times h$ *plus* a border of one pixel around it. Because the output tiles must together form the complete output image, these input tiles must have overlap in case of filters. These overlapping zones are called *ghost zones*. Determining the tile sizes (including ghost zones), for multiple consecutively executed kernels, is referred to as *tile size propagation*. They claim that their experiments have shown that the impact of redundantly transferred pixels is not a problem.

Besides graphs with kernels that do not allow for pipelining (as identified in section 2.4.4), there are graphs for which the buffer sizing algorithm fails to fit all buffers in L1 memory. In that case the graph needs to be partitioned. How this partitioning is performed is not explained, but intermediate data is stored in external memory. The methodology is implemented in the ADRENALINE framework [21].

In chapter 6 also the approach of this work is used for comparison. Although it has been designed for a different platform, the set of constraints is roughly the same.

4.2 Partitioning

As identified in [4], graphs sometimes need to be partitioned. Partitioning a graph into *blocks* is a thoroughly researched field, but most research targets min- and max-cut problems. The cost-function in the problem in this thesis depends on the number of blocks and the connections between tasks of different blocks. Besides that, the cost function also depends on the clustering/mapping of the tasks *inside* a block.

This is an unconventional cost-function and no existing solutions to the problem are known. However, this does not mean that there are no standard approaches to min-cost partition problems [22]. Since the graph partition problem is typically an NP-hard problem, the common approach again is to use a heuristic. A classical min-cut bipartitioning heuristic is [23], where in every iteration a single node is moved from one block to another, based on a local gain. Evaluation of this heuristic shows that making locally optimal decisions can often yield (near-)optimal results.

Chapter 5

Heuristic

This section describes and motivates the use of a novel heuristic that solves the problem described in chapter 3. The heuristic uses a static scheduling strategy, since all information is known before the actual execution starts and execution times are constant for almost[1] all OpenVX kernels.

The heuristic divides the graph into subgraphs, called *gangs*, based on a cost function that can be approximated efficiently. This is referred to as *partitioning* the graph. The gangs in the partition then have to be *scheduled*, taking into account the constraints as described in chapter 3.

The main design decisions concerning the heuristic are given and motivated in section 5.1. After that the outer level algorithm, for partitioning the graph into gangs, is described in section 5.2. The inner level algorithm, used for scheduling the gangs, is explained in section 5.3. In both sections the complexity of the algorithms is described using the big O notation (section 2.6). The complexity of the complete heuristic is discussed in section 5.4. The last section (5.5) describes how an optimal solution can be determined.

5.1 Design decisions

To meet the requirements of section 3.2 for the platform of section 2.1, certain decisions on the form of the solution are made. The decisions are:

- The schedule is executed by a **control program**.
- The heuristic will generate a **pipelined schedule** in order to obtain a compact schedule.
- Kernels will only be **loaded once** per node.
- The partitioning and scheduling phase are decoupled into a **two-level approach**.
- The multi-rate graph will **not** be **transformed** to a **HDFG**.
- An **incremental approach** is chosen that starts from a situation that is **guaranteed** to result in an admissible schedule.
- The heuristic will be implemented in **C**.

These decisions are elucidated in the remainder of this first section.

[1] e.g. Canny edge detection has an edge tracing step. However, the ISP cannot execute these kind of (sub)functions efficiently anyway.

Control program

The generated schedule must be executed on the platform, enforcing the ordering specified by the schedule. This is done by running a control program on the scalar processor (SP). The control program is responsible for loading kernels on PEs and starting and synchronizing data transfers and kernel executions in the correct order. The advantages of running control on the SP are:

- Kernel code is not made unnecessarily complex, compared to kernels handling all control.
- Kernels have very limited control overhead.
- A central point of synchronization reduces synchronization communication.
- The control program can be run in parallel to kernel execution.
- The SP is not interfered by other processes, which can be the case if the host processor is used for control.
- The latency of communication, when using the SP, is smaller than for the host processor.

The control program can be implemented as either an interpreter program that can execute the instructions of the schedule, or the schedule can be transformed into control program code. For the evaluation of the schedule this is not important, but the tradeoff here is simple. The advantage of a generated control program is that it results in less overhead for the scalar processor at runtime, but it needs to be compiled at runtime. For an interpreter program this is the other way around.

Since a code generator for generating control programs already partly exists, it is extended and used to transform the schedule into actual execution. Implementing the interpreter and evaluating the impact of overhead is deemed future work.

Pipelined schedule

Either a generated control program or an interpreter program is used for control. In case of a generated control program, the control program size depends on the schedule length, which must fit in the SP's program memory.

In case an interpreter program is used the instructions must fit in the data memory of the SP. A simple calculation of running a simple single-input, single-output, line-based function on a 4K (3840 x 2160) input image with an equally sized output, shows that it would require 2160 start and synchronization instructions for kernel execution, as well as for the input and output transfers. For this single node graph this means already $2160 \times 2 \times 3 = 12960$ instructions, if the schedule would have an individual instruction for every iteration of each task.

Both cases require that the schedule is denoted in a compact form. Generating the schedule in the form of a preamble, loop body and postamble is sufficiently compact and still allows to achieve a high performance. For this reason the theory of pipelined and modulo scheduling (see section 2.3.2) is used.

No kernel reloading

Loading kernels requires usage of the DMA. This means that no data can be transferred to or from PEs, while a kernel is being loaded to another PE, which

can block execution. Also a PE cannot process data during context switches. For this reason loading kernels is kept to a minimum; one kernel load per node. A comparison to a solution that does allow kernel reloading is made in chapter 6.

Two-level approach

Because of the assumption of no kernel reloading, the graph is naturally divided into subgraphs, called gangs, because of two reasons. Either (i) the total size of the kernels can be too large to fit in the program memories or (ii) the required buffer sizes do not fit in the limited vector memory. These gangs are executed sequentially and read and write all their input and output to external memory. This means that these gangs can be scheduled individually.

The decisions on the partition obviously change the nodes in a gang. Combined with a decision on the mapping, this also changes the required amount and type of communication:

- When kernels are loaded at the same time on the same PE, the data can be kept in local memory, so no transfers are required.
- When kernels are loaded at the same time on different PEs, the data can be transferred from local- to local memory.
- When kernels are not loaded at the same time, the intermediate data is too large to be stored in local memory. Therefore it must be written to external memory. Later, when the kernel that requires this data is loaded, it must be transferred from external memory to local memory again.

This means that every decision can completely alter the scheduling problem for every gang. However, a good approximation on the length of the schedule of a gang can be calculated efficiently. For this reason it is intuitive and efficient to explicitly decouple the partitioning and scheduling phase.

No HDFG transformation

There are existing solutions for parts of our scheduling problems, but they require homogeneous dataflow graphs. There are algorithms that describe how to transform cyclo-static and synchronous dataflow graphs into their single-rate equivalents ([24]). The transformation is avoided for the same reasons for which [24] avoids it for their scheduling solution.

The main reason is that, when a (C)SDF graph is transformed into its single-rate equivalent, $q(n)$ copies of every node n are introduced (including the transfer nodes). This can cause the model size to explode. The worst case growth rate for the size of the model in terms of nodes is exponential, but it is mainly dependent on input and output rates of the kernels in the graph.

By avoiding the transformation to HDFG, combined with the decision to avoid kernel reloading, all iterations of the same node are mapped to the same PE. This reduces the solution space significantly and might therefore exclude potentially superior schedules. However, when looking at the set of mainly primitive OpenVX kernels and related work, it is clear that the tiling approach is not suitable for higher numbers of cores, due to the external memory bandwidth bottleneck. Therefore the heuristic focusses on pipelined execution, in which communication with the external memory is minimized. The potential loss of

performance in cases where memory bandwidth is not the bottleneck can be compensated, by a method that is described in the future work section 7.1.

Incremental approach

Not being able to find an admissible schedule results in not being able to execute the application. Since this means failure, finding an admissible schedule must be guaranteed. It is assumed that the kernels can be executed at least individually. Therefore, if the starting point of the heuristic is a solution where all kernels are executed one by one, a valid schedule is guaranteed to be found.

By making locally optimal decisions for moving a processing node from one gang to another, the makespan of the total schedule is strictly decreasing and schedulability is guaranteed by rescheduling only the two gangs that are altered by moving the node. This converging behavior is preferable over the exhaustive search, which might not terminate before the user runs out of patience, resulting in failure.

Heuristic

As far as known to the author there exists no scheduling heuristic that considers the combination of the constraints as described in 3 and generates pipelined schedules. Most scheduling heuristics do not take into account limited program memory (or program loading costs) and therefore the combination of the partitioning and scheduling problem. Since a partition is only valid if the gangs are actually schedulable, the scheduling problem has to be solved for every gang, for every decision on a partition.

Typically, scheduling algorithms only take into account the runtime of a single execution of the algorithm. Also static scheduling algorithms normally do not assume time budgets of a few seconds, since they are typically executed at compile time. For this reason other algorithms are fast enough for achieving a high performance schedule, given the tight scheduling budget. For this reason a new heuristic is proposed that is faster than combinations of known algorithms, by exploiting the characteristic properties of this scheduling problem:

- The optimal period can be efficiently calculated in case the buffer size constraints are neglected and the mapping is known. This period is used by the cost function of the partition problem and is used as a target period for the scheduling problem.
- The number of data transfers is dependent on the partition and mapping. Combined with the first property, this can be used to make important decisions in an early stage, based on efficiently calculated expectations.
- The target schedule consists of a preamble, loop body and postamble. Since the classical iterative modulo scheduling algorithm ([8]) requires transformation to an equivalent HDFG, this heuristic does not use it, but works with the original SDFG. Deriving a pipelined schedule requires (i) determining the number of executions in the preamble and (ii) finding a schedule for one period. This greatly reduces the scheduling time, compared to full graph unrolling.

The bottom line of creating a novel heuristic is that, as far as known to the author of this thesis, there is no algorithm that takes into account all constraints of the problem of this thesis. Based on problem characteristics and the coherency

between the partition decisions and schedule quality, a heuristic can be designed that uses this information. For this reason it is expected to be faster than combining several existing methods, which is required given the tight scheduling budget.

Programming language

The output of the scheduler will be a text file with the schedule. Scripting languages, like Python or Ruby, are very convenient for text generation, in contrast to a programming language like C. However, compiler optimizations result in relatively superior performance for programs written in C/C++. Since scheduling budget/speed of the heuristic is very important in this thesis, the decision here is driven by performance and not by programming convenience. For this reason C is chosen for the implementation of the scheduler.

5.2 Partitioning

In the outer level of the heuristic decisions on the partition and mapping are made. Before describing the constraints and algorithms, an explanation of the nomenclature and definitions, including the cost function, are given in section 5.2.1. Section 5.2.2 lists the constraints that have to be taken into account, and a description of the algorithm itself is given in section 5.2.3.

5.2.1 Nomenclature and definitions

The graph is divided into subgraphs, which are called **gangs**. Every node v resides in exactly one gang that is denoted as $gang(v)$. For every gang a pipelined schedule is derived and the gangs are executed sequentially. If a gang g_x is executed before gang g_y, this is denoted as $g_x < g_y$.

The initial partition is changed by performing locally optimal **moves**. A move is the event of removing a node from one gang and adding it to another. The gang from which the node is removed is referred to as the **source gang** and the gang to which the node is added is called the **target gang**.

The goal of the heuristic is to minimize the total execution time of the graph. The exact execution time of a gang cannot be determined efficiently in the partitioning step, since it requires the lengths of preamble, loop body and postamble. These lengths are only known after scheduling the gang. Therefore, an approximation of the *cost* of a gang is used. With the decisions made in section 5.1, a lower bound on the length of the loop body can be determined. Combining the information of:

- A known number of executions of the nodes (based on the repetition vector $q(n)$).
- Multiple iterations of the same node are mapped to the same PE (or DMA).
- No overlap of executions on the same hardware (for PEs and the DMA).

allows us to define the *optimal cost* of a gang as:

$$cost(g) = \max_{h \in \Pi \cup \Phi} \sum_{n \in nodes | hw(n)=h} q(n) t_{exe}(n) \tag{5.1}$$

Which says that the best achievable period, i.e. the **optimal cost**, equals the maximum of the total execution time on all hardware (PE or DMA). This total execution time on a hardware component equals the sum of $q(n) \cdot t_{exe}(n)$ for all nodes that are mapped to the same hardware. This cost is referred to as the optimal cost, since it equals the best period that can be achieved for the gang. After a gang is scheduled the real period is known. The real period is referred to as the **real cost** of a gang.

The cost of the whole partition is defined as the sum of the costs of all gangs. Since the number of executions of all nodes in a period are taken from the repetition vector, this sum of costs is directly proportional to the makespan of the whole graph. For this it is implicitly assumed that the impact of the execution times of the preamble and postamble on the average period is negligible. The validity of this assumption can afterwards be checked with equation 2.2.

A **gain** is associated with every move. This gain is defined with the definition of the cost of equation 5.1. If there are two gangs g_1 and g_2 and moving a node from g_1 to g_2 is considered, which results in the new gangs g_1' and g_2', then the gain is defined as:

$$gain = (cost(g_1) + cost(g_2)) - (cost(g_1') + cost(g_2')) \tag{5.2}$$

If the costs of g_1' and g_2' are based on the optimal cost then the gain is referred to as the **expected gain**. If the costs of g_1' and g_2' are real costs then this is called the **real gain**.

5.2.2 Constraints

The constraints that have to be taken into account when making a decision on the partition are:

- The size of the kernel binaries of nodes that are in the same gang and mapped to the same PE, should not exceed the program memory size of a PE.
- Data dependencies should be taken into account. Since gangs are executed sequentially it means that if a there is an edge (u, v), then node v must be in the same gang or in a gang that is executed after the gang with node u, because v depends on the output of u. We denote this constraint as $gang(u) \leq gang(v)$.
- Some kernels are not suitable for pipelining, due to the relation between required input pixels for calculating the output pixels. For this reason the kernels mentioned in section 2.4.4 are currently executed separately.
- The other kernels mentioned in section 2.4.4, for which only the input can be pipelined (mean and histogram computation, etc.), can be moved like the other kernels, but with one restriction: Since their output is a scalar value that is only available after all input lines have been processed, kernels that depend on it can only start their execution in the next gang or later.

5.2.3 Algorithm

Section 5.2.1 already explained that a partition is constructed by a sequence of locally optimal moves. This section explains the algorithm that is used for this in more detail and motivates the design decisions. As explained in section 5.1,

the algorithm starts with a partition with one node per gang, to guarantee that an admissible schedule has been found when the scheduler runs out of budget. From this starting point, locally optimal moves are performed that reduce the cost of the partition.

Algorithm 1 shows the pseudocode for the partitioning algorithm. First the main structure and execution are explained. Afterwards motivations are given for the design decisions. These decisions are marked with a number ([1]) in the textual explanation, which correspond to the in-depth explanations afterwards.

```
order[ ] = determine_order(G)                                   O(X_1)
timer = init_timer()                                            O(1)
partition = create_initial_partition()                          O(n)
idx = 0
cur_gang = get_gang(order[idx++])                               O(n)
while update_timer(timer) ∨ not_improving() do                  O(1)
    cur_gang = get_next_gang(order, idx)                        O(n)
    moves[ ] = get_possible_moves(cur_gang)                     O(X_2)
    move = get_best_move(moves)                                 O(1)
    while move.gain ≥ 0 do                                      O(m)
        new_source_gang = remove_node(get_source_gang(move))    O(n)
        new_target_gang = add_node(get_target_gang(move))       O(n)
        source_gang_cost = schedule(new_source_gang)            O(S)
        target_gang_cost = schedule(new_target_gang)            O(S)
        real_gain = calc_real_gain()                            O(1)
        if real_gain ≥ 0 then
            commit()                                            O(1)
            break
        else
            move = get_next_move(moves)                         O(1)
        end
    end
end
```

Algorithm 1: Partitioning algorithm for the outer-level of the heuristic. The symbols used for denoting the complexity are explained in section 5.2.5.

The first step is to determine an ordering of the nodes of the graph G. The implementation currently supports three different sorting methods; topological, level, and execution time sorting. More details on this can be found in section 5.2.4.

After that a timer is initialized, which is used for keeping track of the time spent since starting the heuristic. Then the initial partition is created, including the scheduling of the single node gangs. The current gang (*cur_gang*) is set by getting the gang in which the first node of the ordering is present.

In the outer while loop, the target gang is updated[1] and the moves, with their associated expected gains, that are possible *towards* this gang are determined[2]. The outer while loop repeats until the scheduler runs out of budget or until it is detected that the result will not further improve[3].

In the inner while loop the move with the highest expected gain is selected. Based on this move the source and target gang are created. These are CAKGs.

Both of these gangs are scheduled, which returns the real costs of both. With these real costs the real gain is calculated (see equation 5.2, calc_real_gain() is used for brevity). If this gain is greater than or equal to zero[4] the move is *committed*. This means that the old source and target gangs are replaced by the newly created gangs. If the real gain is smaller than zero, the move is not applied and the move with the next best expected gain is selected.

(1) Updating the target gang

Determining which target gang is taken is done based on:

1. If a move was committed for the last target gang. If this was the case then the target gang remains the same. This is done because new moves can be enabled by the last move.
2. An ordering of the nodes of the graph (see section 5.2.4) and the last target gang. If no move was committed then the next node in the ordering is selected. This next node is the first node in the ordering that is not in the same gang as last target gang, since this gang was already checked for moves in the last iteration of the outer while loop.

(2) Early termination

It makes no sense to run the heuristc until the budget is exceeded, when it is known that the cost is not going to be improved anymore (*not_improving*() in the algorithm). For this reason information about the past partitioning decisions and their corresponding gains is kept. If we define one iteration of checking moves towards all gangs as a cycle, then *after one cycle without a gain* the heuristic is terminated and the final schedule is generated.

(3) Determining the moves

The best move is determined in two steps. First the list of allowed moves towards the current target gang is generated. This is done based on the constraints in section 5.2.2, except for the program memory constraint. A recursive function with complexity $O(n)$ is used for this. After that, the expected gain is determined based on the optimal cost of the source and target gangs resulting from the move. For this a mapping needs to be determined.

The mapping is currently determined by an exhaustive search. It selects the mapping based on the cost. If the cost is equal for different mappings then the mapping with the smallest transfer time is selected. This is done since it is deemed easier for scheduling, because it means more freedom on the shared resource (DMA). The complexity of the exhaustive mapping equals $O(m|\Pi|^n)$, where m equals the number of allowed moves and n is the number of *processing* nodes that need to be mapped.

Exponential complexity is something you typically want to avoid. However, n is the number of nodes *in a gang*, which is bounded by the program memory size in reality. In practice this mapping function does not result in scheduling time problems, since the implementation avoids checking symmetrical mappings and breaks early, when the program memory constraint is violated or the best cost found up till then is exceeded. This means that the worst-case complexity of $O(m|\Pi|^n)$ is not realistic. However, in case this heuristic is used for platforms

and kernels for which it holds that the program memory size $\gg$ kernel size it might become a problem. In that case a heuristic can be used for the mapping too, but this is deemed future work because in the situations considered in this thesis this is not a problem.

After getting the expected gains of all moves (for which the mapping did satisfy the program memory constraint) an ordering of the moves is determined. Of course the node with the highest expected gain is the first node in this ordering.

(4) Gains of zero

In case a node, for which performance is limited by data transfers, is added to a gang of which the period is also bandwidth bounded, the expected gain is zero. Initially it might not make sense to move nodes with a gain of zero, because it results in occupying more program memory and requires an extra run of of the scheduling algorithm. However, during implementation and testing of the heuristic, it was identified that applying a move that has a zero gain enables other moves that can have a gain higher than zero. The impact of this decision is evaluated in section 6.3.1.

5.2.4 Design parameters

There are two design parameters that can be easily set in the implementation. Their function is described here.

Sorting

The target gangs are selected in an order following the ordering of the processing nodes. The implementation supports three different types of sorting:

- *Topological sorting.* The nodes are sorted in topological order where a source node has the highest priority. It is implemented using a recursive function with complexity $O(e)$, where e is the number of edges.
- *Level sorting.* Similar to topological sorting, but now the level is used. It is equal to the definition of level in HLFET list scheduling, except for the fact that execution time is taken equal to one for every node. The complexity is for sorting equals $O(n_{snk} \cdot n)$, where n is the number of nodes that are sorted and n_{snk} is the number of sink nodes. This means that the worst-case complexity is $O(n^2)$.
- *Execution time sorting.* The nodes are sorted by $q(n) \cdot t\_exe(n)$, from high to low.

The main advantages of topological and level sorting are that as many nodes as possible, at either a source or a sink, are grouped into a gang, not leaving *leftover* nodes. Leftover nodes are nodes, which cannot be moved to another gang due to the data dependency constraint and the PMEM constraint. The idea behind execution time sorting is that the highest gains are achieved when nodes with large execution times are moved to the same gang. Therefore this type of sorting might converge faster. These three different types of sorting are compared in section 6.3.2.

Search depth

Another design parameter is the search depth for determining the moves. As described in section 5.2.3, the best move is determined by checking all possible moves and ordering them by the associated expected gains. However, to determine the best move it is also interesting to know what the gains are of moving nodes that are enabled by applying the first move. This means that if there are multiple nodes, the best move is not necessarily the move with highest gain for the first move.

This same reasoning can continue for the gain after the second move and this the number of subsequent moves that is checked is called the search depth. Informally said a search depth of four means that the gains are calculated for up to four consecutive moves (if possible, given the constraints).

The impact of the search depth, on both the final makespan and the scheduling time, is evaluated in section 6.3.3.

5.2.5 Complexity

In the description of algorithm 1 the complexity of each line is denoted on the right. For this the big O notation (see section 2.6) is used. In the annotations the n represents the number of nodes of the full graph and m the number of moves. The other letters are used for:

- S denotes the complexity of the scheduling algorithm, which is determined in the next section.
- X_1 equals the complexity of sorting the nodes. Since there are multiple options here it depends on the selected option.
- X_2 is the complexity of generating the list of moves including the associated mappings and gains.

The worst case complexity of the number of moves m is $O(n)$. However, it is very unlikely that this will be the case in practice. First of all, after a move is committed the inner while loop breaks. Typically, the first move is committed. Also the number of possible moves is limited by the graph structure combined with the constraints of section 5.2.2. $O(n)$ can only be achieved when all nodes are separate connected components, i.e. there are no edges between any pair of nodes. For this reason the practical complexity is $O(1)$.

The complexity of the initialization part (before the outer while loop) has complexity $O(X_1) + O(1) + O(n) + O(n) = O(X_1 + 1 + n + n) = O(X_1 + n)$. Since the complexity of sorting is higher than $O(n)$ this reduces to $O(X_1)$.

The complexity of the inner while loop similarly reduces $O(m(S+n))$. Since the complexity of the scheduling algorithm is higher than $O(n)$ (will be determined in the next section), the complexity reduces to $O(m \cdot S)$. The complexity in the outer loop is therefore $O(X_2 + m \cdot S) = O(m|\Pi|^{n_g} + m \cdot S)$, where n_g is the number of nodes in a gang.

The number of iterations of the outer while loop is hard to determine. In section 5.4, where the complexity of scheduling and partitioning is combined, the usability of the complexity analysis is discussed. An estimate on the required number of executions of the outer while loop, after which *not_improving*() returns true, is given there.

5.3 Scheduling

This section describes how the gangs, resulting from the partitioning phase, are scheduled. The gangs are modeled as acyclic (C)SDF graphs, for which the mapping is also known from the partitioning phase. The tokens on the source edges of the gang model are initialized as if all preceding nodes in the complete graph have finished all their executions.

As motivated in section 5.1, a schedule consisting of a preamble, loop body and postamble must be determined, in terms of instructions (see section 3.1.7). How this is done is described first in section 5.3.1. After that, in section 5.3.2, the approach to determine the retiming and buffer sizes is given. Also the list scheduling algorithm that is used is explained there.

5.3.1 Pipelined scheduling

The schedule consists of a preamble, a repeating loop body, and a postamble. The original modulo scheduling approach as described in [8] is not applicable, since it requires HDFGs. The main problem is finding a schedule for the loop body.

Some tools determine optimal throughput by list scheduling and detecting subsequent equal states. The numbers on scheduling time for tools, such as UPPAAL [25], are in the order of 100 ms for simple models. This is too long given the fact that scheduling must be performed multiple times at configuration time. On top of this, also buffer sizes have to be taken into account. Therefore a different approach is chosen.

For the scheduling, non-preemptive execution is assumed, because when executing simple kernels on tiles of data the costs of context switching are relatively high.

Preamble

The purpose of a preamble has already been described in section 2.3.2. In our dataflow graph, initially there are only tokens on the gang's input edges. All internal non-buffer edges have no tokens on them. By executing a preamble, the tokens are distributed onto internal edges. For describing the relation between the executions in the preamble and the number of tokens on the internal edges, the definition of the retiming relation (equation 3.1) is suitable.

For filter kernels this relation is extended to equation 3.2. Since only the first and last rates of a node's executions can differ, this can be easily taken into account when calculating the tokens produced and consumed in the preamble and postamble. However, to have the same situation at the beginning and at the end of scheduling the executions in a loop body the rates must be constant. Therefore we restrict the executions in which this deviating rates are applicable to the preamble and postamble.

The advantage of using retiming for the preamble is that the state, i.e. the tokens on every edge, after the preamble can be determined, without actually deriving start and end times for the nodes' executions. A disadvantage is that the schedule is limited to yield a *strictly periodic* schedule.

Determining schedulability

With the repetition vector q it is known how many iterations of each node are performed in one execution of the loop body. Since the length of the loop body is directly proportional to the makespan of the gang, the optimization problem can be reduced to minimizing the period of one loop body. Determining schedulability now means:

- Determining a value for the retiming of every node.
- Determining a start and synchronization time for $rv(n)$ iterations for every node n, relative to the start of the period.
- Determining buffer sizes.

such that all platform constraints are met. As described in the partitioning section (5.2.1), the optimal period can be determined with equation 5.1. Once this period is achieved during the scheduling of a gang, improving is impossible and scheduling can terminate.

5.3.2 Algorithm

Scheduling the loop body is performed in two similar phases. In the *first phase* a retiming is determined with which the optimal period can be achieved. In this first phase no backedges are present, which means that buffer sizes are infinite. In the *second phase* the required number of buffer tokens is determined, given the retiming. While doing this, the memory size available for the buffers must be taken into account.

Both phases have the same basic structure:

1. Determine a minimal initial value (retiming or buffer sizes).
2. Perform list scheduling.
3. If the optimal period is achieved then scheduling is done. Otherwise, make use of the information of the last schedule for updating the model (retiming or buffer sizes) and reschedule.

Because of the repeating pattern of list scheduling and updating the model based on the previous schedule we will refer to this method as *iterative list scheduling*.

Shared resources are handled by following the list scheduling approach. If multiple nodes are ready for execution on an available hardware component, the node with the highest priority (based on level) is executed.

List scheduling

The list scheduling algorithm that is used is algorithm 2. First, two arrays are initialized. The order array holds the priorities and is sorted based on level. The iter array holds the number of executed iterations of a specific node. Also a time is kept, for all PEs, that equals the point in time at which the current execution on it will finish.

After initialization a while loop is entered in which the time is increased (starting at 0). In this while loop, first tokens are produced for node executions that finished at the current point in time (apply_productions()). After that, the following is checked for all nodes, ordered by the priorities in the order array, if:

- An execution is desired. Every node must be executed the amount specified in the repetition vector for the node.

```
order[ ] = sort_nodes_by_level()                                   O(n_snk · n)
iter[ ] = init_iterations()                                        O(n)
while update_timer() do                                            O(|Π|)
    apply_productions()                                            O(|Π|)
    for p in 0..num_nodes-1 do                                     O(n)
        n = order[p]
        if iter[n] < rv[n] ∧ hw_ready(mapping[n]) then             O(1)
            if enough_tokens(n) then                               O(1)
                execute(n)                                         O(1)
                iter[n] += 1
            else if enough_nonbuffer_tokens(n) then                O(1)
                store_blocking_buffer_info()                       O(1)
        end
    end
end
```

Algorithm 2: List scheduling algorithm for scheduling a graph G.

- The hardware is available (*hw_ready*(*mapping*[*n*])). This is checked by looking at the time at which the PE that the node is mapped to is idle again. This time is set when a node's execution starts. This also means that execution of another node with a higher priority can have started on this hardware during this iteration of the while loop.
- If there are enough tokens on all input edges for starting execution, denoted by enough_tokens(n).

If all these conditions are satisfied then a node is executed. When a node is executed the following is done:

- The time at which the PE that the node is mapped to is idle again, is set to the current time plus the execution time of the node.
- In case the PE was idle before starting execution, i.e. the point in time at which the PE would become idle is earlier than the current point in time, there was slack on this PE. *Information on this slack is stored as profiling information for updating the retiming in phase one.*
- A production is added to a list of productions. A production is a pair of a time value with a node ID, indicating that the node with the node ID will produce tokens at the specified time.

After starting execution for a node, its number of iterations is increased.

As can be seen, the check for enough tokens on the input edges is not in a conjunction of conditions with the other two requirements for execution. This is done to be able to determine information on **blocking buffers** for phase two. A blocking buffer is a buffer that is preventing execution, so if execution is desired, the hardware is idle and there are enough tokens on the non-buffer edges, a blocking buffer is encountered. Information on this is stored for updating the model in phase two.

After all nodes have been checked, the timer is updated (*update_timer*()) to the first point in time at which a node's execution finishes, because this is the next earliest point in time at which a node can potentially be executed.

The complexity of every operation of this list scheduling algorithm is added

in the notation of algorithm 2. In these annotations, n is the number of nodes *in the gang* and $|\Pi|$ the number of *used* PEs. The number of iterations of the while loop in the worst case, equals the total number of firings $n' = \sum_{n \in nodes} q(n)$. This means that the complexity of list scheduling is $O(n + n'(|\Pi| + n * 1)) = O(n'n)$, because the number of used PEs never exceeds the number of nodes in a gang.

Retiming phase

In the first phase the required retiming, for being able to achieve the optimal period, is determined. In this phase the (C)SDF model without buffer edges is used. This mimics infinite buffers, since execution of nodes is not prevented by the lack of tokens on an incoming buffer edge.

The required retiming is determined in two main steps. In the first step the amount of required **preparations** is determined. When a node requires one preparation it means that, at the start of the period, there must be enough tokens on all input edges of the node to execute once. The amount of tokens required for an execution is given by the input rate.

This results in an array, holding the required preparations of all nodes. This is converted into the required retiming by a recursive function. This function is started from all sink edges and continues through all input edges of the current node. The retiming of its source node is determined for every edge, based on the tokens that are required. The required amount of tokens is determined by the required preparations of the sink node and the consumed tokens, based on the previously determined retiming of the sink node. This relation is more formally given by:

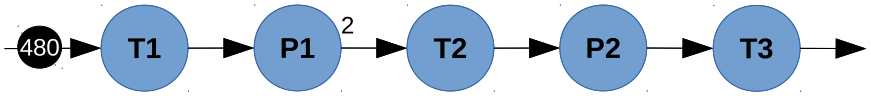
$$r(src(e)) = \left\lceil \frac{(prep(snk(e)) + r(snk(e))) * c(e)}{p(e)} \right\rceil \tag{5.3}$$

An example of this can be seen in figure 5.1. Assume that node P2 is a bottleneck and that therefore P2 and T3 must be able to execute at the start of the loop body, without waiting for their predecessors to finish execution. Also assume that it turned out that T2 requires a preparation too. Then the required preparations are as shown in table 5.1. We start at the sink edge and continue through the input edges of T3 (the retiming of T3 equals zero, since no requirements on the number of output tokens are present). Applying equation 5.3 on edge $(P2, T3)$ gives $r(P2) = \left\lceil \frac{(1+0)*1}{1} \right\rceil = 1$. Continuing through P2's input edges gives $r(T2) = \left\lceil \frac{(1+1)*1}{1} \right\rceil = 2$, $r(P1) = \left\lceil \frac{(1+2)*1}{2} \right\rceil = 2$ and $r(T1) = \left\lceil \frac{(0+2)*1}{1} \right\rceil = 2$.

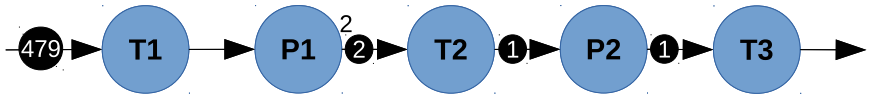
Applying this retiming to figure 5.1a results in 5.1b. What can be seen here is that T2, P2 and T3 now have enough tokens available to start one execution at the start of the loop body. One token would be sufficient for node T2 on edge $(P1, T2)$, but P1 has an output rate of 2. This is the reason for the ceiling function in equation 5.3.

After applying the retiming, the model is scheduled using the list scheduling algorithm (algorithm 2). In case the optimal period is not achieved, the generated schedule is used to updated the required preparations and retiming.

This updating is done based on the *slack*, i.e. the idle time between tasks on the same hardware. The focus is on PEs on which execution exceeds the



(a) Gang model.



(b) Gang model after applying retiming.

Figure 5.1: Determining the retiming based on the number of preparations.

Node	Required preparations	Resulting retiming
T1	0	2
P1	0	2
T2	1	2
P2	1	1
T3	1	0

Table 5.1: Required preparations and the resulting values for the retiming corresponding to figure 5.1.

optimal period. What slack in the schedule means, is that there is a task for which the hardware was available, but that was not used due to a lack of tokens for a task. By increasing the required preparations for the node that is executed after the slack, it is assured that in the next iteration of the list scheduler the required token(s) are available. The longest slack is checked first since the gains are the highest there. Of course this can only be done if buffer size constraints remain satisfied.

There is one extra condition for increasing the required preparation for a node (except buffer size constraints). If another node, which produces the required input tokens, also encounters slack before its execution and this slack is longer, this node's required preparations are increased. This is checked recursively. This way the maximum amount of slack is reduced with the minimum amount of tokens.

Buffer sizing phase

In the second phase the buffer sizes must be determined. The approach is the same as for determining the retiming:

1. Determine the initial required buffer size. This is done based on the minimum amount of tokens to avoid deadlocks, on the number of "absorbed" tokens by filter kernels, and on the final values for the required preparations and retiming determined in the previous phase. The number of preparations is taken into account, since in the previous step it was determined that there should be enough initial tokens on these node's inputs. This means also enough available output buffers must be present.
2. Perform list scheduling for a schedule of the period on the model with the backedges. If the optimal period is achieved then the buffer sizes are sufficient. Otherwise, information gathered during the list scheduling is

used to update the buffer sizes and reschedule.

The model is updated based on the information on blocking buffers (see the list scheduling description). As in the updating of the retiming, here also the slack is sorted by size, such that the maximal amount of slack is reduced with the tokens.

Example

To illustrate this approach, a more detailed explanation of the updates on the model after every iteration can be found in appendix B. It includes Gantt charts and visualizations of the graphs. Also an excerpt of the control flow in the form of a list of instructions is included there.

Complexity

Both phases increase their values until the optimal period is reached, or until no buffer size is left for adding tokens. It is hard to analytically determine the worst-case number of iterations for this. Intuitively it is $O(1)$, because when updating the model the amount of tokens added is based on the amount of time that the optimal period is exceeded. It should therefore not dependent on the number of nodes in a gang. However, from practice it has become clear that the number of iterations for determining the retiming is relatively small, but values for the number of iterations for determining the buffer sizes are larger. This performance will be quantified in the next chapter in section 6.4.4.

The complexity of backtracking slack in the retiming phase has a worst case complexity of $O(n' \cdot n_g)$. Iteratively updating and list scheduling in the retiming phase therefore equals $O(i_r(n' \cdot n + n' \cdot n)) = O(i_r \cdot n' \cdot n)$, where i_r is the number of iterations after which the optimal retiming is achieved.

The complexity of the current implementation of updating the buffer size is $O(n^2)$, but since this complexity results from sorting a list it can be implemented more efficiently. However, similar to the retiming phase the complexity equals $O(i_b(n' \cdot n + n^2)) = O(i_b \cdot n' \cdot n)$, because $O(n) = O(n')$ (but $O(n) \neq O(n')$!). Here i_b is the required number of iterations for updating the buffer sizes before the optimal period is achieved.

The complete scheduling algorithm executes these two phases sequentially. Therefore the scheduling complexity S equals $O(i_r \cdot n' \cdot n) + O(i_b \cdot n' \cdot n) = O((i_r + i_b)n' \cdot n)$.

5.4 Complexity

The complexity of the partitioning phase inside the outer while loop was already determined to be $O(|\Pi|^{n_g} + m \cdot S)$, where n_g is the number of nodes in a gang and S is the scheduling complexity. The scheduling complexity was determined to equal $O((i_r + i_b)n' \cdot n)$. Combining these complexities results in $O(|\Pi|^{n_g} + (i_r + i_b)m \cdot n'_g \cdot n_g)$.

As stated in the complexity analysis of section 5.2.5, the number of iterations of the outer while loop of the partitioning phase is not yet considered. The performance depends on the number of moves that can be performed within thie scheduling budget, but to determine the complexity of the heuristic it must be approximated how this required number of moves scales with model size.

Since nodes are not moved around multiple times, intuitively the number of moves scales linearly with the number of nodes.

Running the heuristic and measuring the number of *non-zero* moves, the result of table 5.2 is obtained. From this it seems clear that, at least in practice, the number of moves after which the final period is achieved, scales linearly with the model size.

Graph:	**HDR**	**HDRx2**	**CE**	**Hist**	**Verif.**	**Diff**
Model size n	20	40	8	8	7	7
Moves	22	45	8	8	6	7

Table 5.2: The number of iterations of the outer while loop. The count is stopped when the final makespan is achieved.

With this result, the complexity of the heuristic equals $O(n \cdot m \cdot |\Pi|^{n_g} + (i_r + i_b)n \cdot m \cdot n'_g \cdot n_g)) = O(n \cdot m \cdot |\Pi|^{n_g})$, so the bottleneck when scaling is the determination of the optimal mapping. As motivated before this is not a problem in our case, because the number of nodes in a gang n_g in practice (for our platform) is bounded by the program memory size and other partitioning constraints.

5.5 Optimality

To determine the optimal result and to be able to verify the correctness of a solution found by the heuristic, the problem can be specified as a Satisfiability Modulo Theories (SMT) problem. SMT problems are decision problems where the satisfiability of a set of constraints, expressed in first-order logic with respect to background theories such as integers, reals or arrays, must be determined. How to formulate the SMT problem based on a graph specification is described in appendix A.

An implementation, which automatically generated input for SMT solvers from a graph specification, was created and tested. Due to multiple required changes, in the structure of the library of kernels for the scheduler and the inclusion of filter kernels, this implementation is no longer in an executable state.

An older, working version of the implementation showed that the solving time becomes too long for larger graphs. The time the solver spent on proving optimality of a partition, *only based on optimal costs*, of a 15-node graph, was 3 hours. When including the scheduling problem for determining the real costs of gangs, the solver was terminated after 1 week of execution.

Therefore, the usefullness of the SMT problem formulation is questionable. It is interesting to see if the performance can be improved by adding symmetry breaking constraints. On the other hand it shows that using a heuristic is required, when scheduling budgets are in the order of seconds and graphs become larger.

Chapter 6

Results

In this chapter the performance of the heuristic is evaluated and compared to other solutions. The main figures of merit are the achieved *makespan* and the *scheduling time*, i.e. the time it takes to generate the schedule.

First, the set of benchmark graphs is described in section 6.1. To prove that a graph can be transformed into correct execution, the whole control flow generation is performed and tested using a simulator. This is described in section 6.2. In section 6.3 the impact of different values of the design parameters is analyzed, using the set of benchmark graphs. Based on this, a final configuration is chosen. With this configuration, the final results of the heuristic are obtained and these results are compared to alternative approaches. The description of the other approaches and the comparison can be found in section 6.4.3. In this section also the performance of the scheduler is evaluated.

The results in this section are obtained with the following environment and variables, unless stated otherwise:

- The scheduler is executed on a platform with an Intel® Core™ i5-5200U CPU and 4GB of RAM.
- A scheduling budget of one second is used throughout this chapter. The limit for keeping the user's attention is 10 seconds (section 2.5), but more needs to be done (e.g. control program compilation) in this timespan and the platform mentioned at the previous bullet is faster than mobile platforms.
- The number of processing elements is four.
- The used image size is 1920x1080 (full HD resolution).
- DDR is modeled with a bandwidth that is a factor 40 smaller than the local memories [2].
- All values on scheduling time are averaged over 10 runs.

6.1 Benchmark graphs

For evaluating the heuristic and comparing it to alternative approaches, a set of application graphs is used that consists of the following graphs:

1. A *Histogram equalization* graph from [26], where histogram equalization is split into separate histogram computation and equalization nodes. It includes kernels with a very low arithmetic intensity, such as color channel

extraction kernels, and non-pipelineable histogram computation kernels. It consists of 9 nodes.

2. A *Difference highlighting* graph. It takes two grayscale images as input and determines the absolute differences of pixel values. This difference is thresholded and the resulting pixels are dilated three times and added as black pixels to one of the original images. In total, the graph consists of 7 pipelineable nodes.
3. A *Canny edge detector* graph from [21], extended with color conversion. In total it consists of 8 pipelineable nodes.
4. A *High-dynamic-range* (HDR) graph. It takes three input images and combines these into one output image. It includes non-pipelineable nodes, image pyramids and a mixture of bandwidth- and compute-limited kernels. In total it features 20 nodes. It must be noted that not all kernels in the graph are OpenVX kernels, but it shows the performance for a graph consisting of more compute-bounded kernels.
5. The graph used for *verification* (see section 6.2), consisting of 7 nodes.
6. For checking scalability, also a graph referred to as *double HDR* is used, which equals two copies of the HDR graph. What this practically means is that, in case of HDR video, two output frames can be calculated at the same time at the cost of additional latency.

6.2 Verification

To check the correctness of the approach in practice, a verification graph is used. It is constructed in such a way that it contains a variety of graph properties and node types. The schedule that is generated by the implementation of the heuristic, is transformed into a control program by a code generator. This code generator creates a control program by combining standard parameterized code blocks in the order of the instructions in the schedule. This code is compiled and executed using a simulator.

The graph can be seen in figure 6.1. It consists of filter kernels (F1 and F2), up- and down-scaling kernels (P1 and P2), regular kernels that implement addition (K1 and K2) and a node that is non-pipelineable and produces a scalar output. Because linking multiple kernels into one binary is not implemented as part of the flow, the kernel sizes are taken equal to the program memory size. This way only one kernel per gang can be mapped to the a PE.

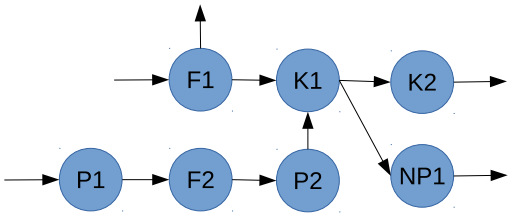


Figure 6.1: The graph used for verification.

The kernel implementations and the input data are constructed in a way that the availability of the correct data and the ordering of transfers and executions can be checked by inspecting the output. Besides checking the correctness of

the schedule and the transformation into a correct control program, this also allows us to verify that execution times of these kernels are close to constant. A third reason for executing this verification graph, is that it gives realistic values on execution times and program memory usage. These values can also be used to estimate execution times of other kernels, without implementing them.

The automatic control flow generation from graph to execution is implemented and the correctness is verified for this graph. Of course this is not a complete verification of the heuristic, but it can be seen as a proof of concept and relates the theoretical schedule to practice with the execution times and program memory requirements.

6.3 Design parameter evaluation

In this section, the influence of the following design parameters and decisions is evaluated:

- Committing moves with a *gain of zero*.
- The *sorting method* for determining the target gang.
- The relation between scheduling time and makespan for different *search depths*.

Based on the results of this section, the best configuration of these parameters is determined in section 6.3.4. This configuration is used in section 6.4.3 to compare the heuristic to other approaches.

6.3.1 Zero gain

As mentioned in section 5.2.3, it was identified during tests with synthetic graphs, that performing moves with a gain (equation 5.2) of zero could be benificial. The reasoning behind this is that it can enable moves of other nodes to the gang that would have a gain greater than zero.

When running the heuristic on the benchmark graphs, with and without committing zero-gain moves, there are no significant differences in the makespans (all $< 0.1\%$), except for one specific case. When scheduling the verification graph, with target gangs sorted by level, the makespan is improved by 56% when zero-gain moves are enabled. The reason is that there is a zero-gain move, that enables a beneficial move. The problematic situation in case zero-gain moves are not performed, is illustrated with figure 6.2.

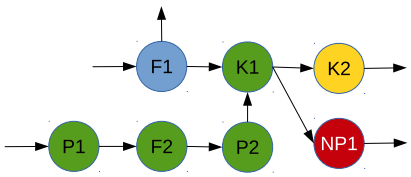


Figure 6.2: Situation where committing a zero-gain move, results in a better makespan. The nodes that have the same color are in the same gang.

Due to level sorting, the first target gang is the gang that contains node P1. Moving node F1 is a zero-gain move because the performance of the gang with P1 is bandwidth-limited and F1 does not result in any reduction in transfers. Therefore in three subsequent moves, node F2, P2 and K1 are moved towards

the gang with P1, resulting in the situation of figure 6.2. In this figure, nodes with the same color are in the same gang.

Since no more moves can be performed towards the target gang (one node per PE for the verification graph), the next target gang is selected. According to the level-sorted nodes, this is the gang with node F1. Due to data dependencies only a move K1 is allowed. This move is a zero-gain move, because the data that is transferred between F1 and K1, and P2 and K1 is equally sized. Performing this zero-gain move, would enable subsequent moves resulting in a partition consisting of two gangs ({P1,F2,P2},{F1,K1,K2,NP1}). If this move is not applied then the final partition consists of three gangs; ({F1},{P1,F2,P2,K1},{K2,NP1}). This extra gang means extra communication between local and external memories, resulting in the large increase in makespan.

Another important measure is the scheduling time. The scheduling time is only affected by disabling zero-gain moves for the HDR and the double HDR graph. The scheduling time is decreased by a factor of less than 1.5 for the normal HDR graph, but the time it takes to finish the heuristic for the double HDR graph is decreased on average by a factor 4 for all sorting types. A decrease in scheduling time was expected, when the amount of allowed moves is limited, but a factor 4 is higher than expected.

The main reason for this large decrease is that, when a partition with the final makespan is achieved, the heuristic terminates faster, because no zero-gain moves are scheduled anymore. To quantify this, the scheduling time with zero-gain moves enabled and disabled are 28 and 8 ms, respectively, for the double HDR graph with topological sorting. The time after which the final makespan is achieved are 17 and 7 ms. This means that 11 ms are spent on performing zero-gain moves until the scheduler finishes. This can also be seen in figure 6.4. For the HDR and double HDR graph the curve is horizontal after the final makespan is achieved.

This 17 vs 7 ms is still is a significant difference, but it seems to be only present for the double HDR graph. The reason the scheduling times of other graphs are not (or less) impacted has to do with the presence of zero-gain moves. Zero-gain moves are only present when there is no aggregation of IPC resulting from the move, and the execution time of the target gang is bandwidth-limited. Since external memory bandwidth is typically the bottleneck and most OpenVX kernels are simple stencil operations, performance of target gangs is typically bandwidth-limited. Since the double HDR graph consists of two connected components, there are many moves without IPC aggregation, i.e. zero-gain moves, possible.

In general this means that, if a graph consists of multiple connected components, the difference in scheduling time will be relatively large. However, disabling zero-gain moves might prevent high gain moves, which can result in makespans that are far from optimal.

6.3.2 Sorting type

The target gangs are selected based on a sorted list of the nodes. The gang of the node with the highest priority is selected first, and moves towards this node are checked. In section 5.2.3, three different types of sorting were mentioned and motivated, which are topological sorting, level sorting and sorting by execution

time. When nodes are sorted topologically or by level, nodes near the graphs input have the highest priority.

These three types of sorting are extended with two other types that we will call *reversed level* and *reversed topological* sorting. They are equal to level and topological sorting, but the ordering is reversed. The resulting makespans, for the different types of sorting for every benchmark graph, are shown in figure 6.3. What can be seen is that the sorting type only influences the makespans for HDR, double HDR and verification. The largest difference, is an increase of the makespan by 18%, for the double HDR graph.

When taking a closer look at the resulting partitions, the differences are the result of making locally optimal decisions and not of the differences between topological and level sorting. For example, for the verification graph of figure 6.1, reversed topological sort starts at node K2 and reversed level sort starts at node NP1. The different makespan results from the fact that moving node NP1, to a gang with nodes K1 and K2, has a lower gain than the other moves. This results in node NP1 being a leftover node, while moving node K2 to a gang with K1 and NP1 does have a higher gain. This difference in final makespan is the result of indeterminism in the sorting methods, since both starting nodes could have been the highest priority node for both types of sorting.

The advantage of starting near a source or a sink of the graph is to avoid leftover nodes, but due to the graph structure and node properties, leftover nodes can still occur. Leftover nodes are also the reason that the largest differences are present for the double HDR graph. Since it consists of two connected components, early moves, where nodes of different connected components are placed in the same gang, are relatively less restricted due to data dependencies.

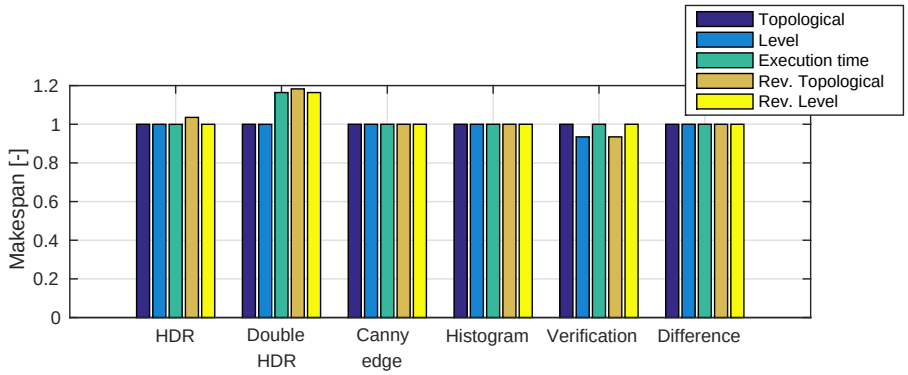


Figure 6.3: The makespans for the benchmark graphs for different sorting methods, relative to the result with topological sorting.

Another expected result is that sorting by execution time will converge faster to the final makespan, because gains are higher when nodes with larger execution times are moved to the same gang. Figure 6.4 shows the makespan versus the scheduling time. In the figures, reversed level sorting is omitted, since the curves for this type of sorting closely match the curves of reversed topological sorting and omitting this type of sorting improves the readability.

What can be seen from this figure is that, when sorting by execution time, the heuristic indeed has high initial gains. It must be noted that for the HDR

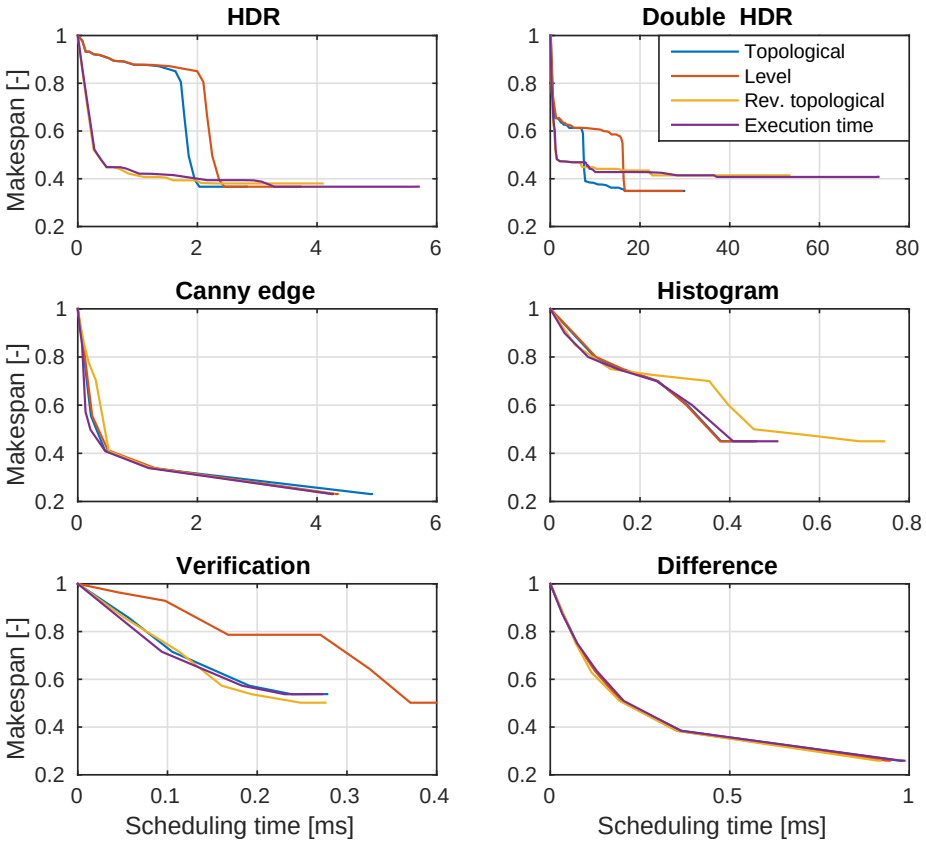


Figure 6.4: Makespan versus scheduling time for all benchmark graphs for different sorting methods.

graph the node with the highest execution time is at the sink of the graph. For this reason the reversed sorting methodologies converge similar to execution time sorting. What can be seen for the single and double HDR graph is that topological and level sorting results in small gains initially. This is the case because at the three inputs, there are non-pipelineable kernel nodes (histogram computation), for which moves do not result in IPC aggregation. The reason that topological sorting improves earlier is that the ordering of nodes is less strict. This means that nodes that are further from the source node can be used for determining the target gang earlier.

6.3.3 Search depth

Increasing the search depth can also result in achieving better results, by looking at the moves that are possible after performing the current move. The downside is that determining the gains of moves requires mapping and this must be performed roughly m^{sd} times, where m is the number of possible moves and sd is the search depth.

The results for a search depth of up to five, relative to the makespan at a search depth of one, are given in table 6.1. These are the results obtained with level sorting, but the results show the same possibilities as for other types of sorting too.

Graph	**SD = 1**	**2**	**3**	**4**	**5**
HDR	1.00 (3)	1.04 (5)	1.04 (8)	1.00 (22)	1.00 (86)
Double HDR	1.00 (28)	0.98 (41)	0.98 (119)	2.84 (1001)	2.84 (1001)
Canny edge	1.00 (4)	1.00 (6)	1.00 (7)	1.00 (10)	1.00 (13)
Histogram	1.00 (0)	1.00 (0)	1.00 (0)	1.00 (1)	1.00 (1)
Verification	1.00 (0)	1.00 (0)	1.00 (0)	1.00 (0)	1.00 (0)
Difference	1.00 (0)	1.00 (1)	1.00 (1)	1.00 (1)	1.00 (2)

Table 6.1: The makespans relative to the makespan at a search depth (SD) of 1. The values between the brackets are the corresponding scheduling times in milliseconds.

The double HDR graph shows that improvements can be achieved, when increasing the search depth, but that the scheduling budget of 1 second is soon exceeded. The reason that this is especially problematic for the double HDR graph, is (i) its large number of nodes and (ii) the fact that it consists of multiple connected components. For this reason the number of possible moves is higher than for the other graphs and this results in exceeding the scheduling budget for a depth of 4. Obviously, performing moves based on more information does not outweigh performing less moves. Exceeding the scheduling budget must therefore be avoided at all cost.

What also can be noted is that, for a single HDR graph, the makespans for search depths of 2 and 3 are worse than for a search depth of 1. The reason for this is that moves are still locally optimal and that a higher gain for moving a node towards a gang can result in a worse total partition.

The reason that some graphs are not affected by it, both in terms of makespan and scheduling time, is the size of the graphs. A search depth of 5 is only useful if 5 subsequent moves with a zero or positive gain towards a gang are possible and if the final partition is not going to consist of the same gangs anyway.

6.3.4 Best configuration

A combination of the parameters of the previous sections must be chosen, which will be used for comparing the heuristic to the other approaches. Since the heuristic is fast, design space exploration can be done exhaustively. A design space of the five different methods of sorting, with or without zero-gain moves enabled, for a search depth of one and two, is explored. The average difference on normalized makespans is approximately 2.8% and the total scheduling time ranges from 16 to 85 ms.

Since "overfitting" of the parameters for the set of benchmarks is undesired, when comparing to other approaches, the best result of the fast design space exploration is not picked. Instead the choice is motivated by the findings in the previous sections:

- Because performing zero-gain moves may enable high-gain moves, and because the cost in terms of scheduling time is limited and only significant for specific graphs, zero-gain moves are *enabled*.

- Level sorting gives the best overall results, but as was shown in section 6.3.2, the differences are small for other types of sorting and are highly dependent on the graph structure. To avoid overfitting and give a fair comparison in the next section, *topological sorting* is selected, which corresponds to average performance.
- The improvements when increasing the search depth are disappointing and can come at the cost of an exponential growth of the scheduling time. Therefore, a *search depth of one* is used.

Since the graph structure is known at the start of scheduling, the parameters can also be selected based on the graph structure. For example, the search depth can be made a function of the number of nodes and whether or not the graph consists of multiple connected components. However, this is considered future work.

6.4 Comparison

This section compares the results of the heuristic to other approaches. These approaches are:

- The **sequential** approach. This is the simplest approach in which all nodes are executed individually. Processing and transfers are performed in parallel, but all input data is read from external memory and all (intermediate) output data is written back to external memory. This approach corresponds to executing a sequence of OpenCV kernels and results are obtained in the form of the initial partition.
- The **tiling** approach. The data is divided over the different PEs, to compare to approaches such as [4]. How this approach is implemented is explained in section 6.4.1.
- A **naive list scheduling** approach. How it determines the order of execution is described in more detail in section 6.4.2. The goal of comparing to this approach, is to show the effect of reloading kernel binaries.

6.4.1 Tiling approach

In [4], a cluster-based many-core platform is targeted and the input is divided into tiles. Tiles are loaded from external memory into the local memory of the cluster, after which multiple cores process the tile. The output is written back to external memory again and potentially reloaded in a later gang. External memory contention was identified to be a bottleneck.

The tiling approach that is evaluated here, uses a strategy similar to [4] and the data partitioned style of [19]. Instead of dividing pipelining stages over the PEs, the data is divided over the PEs. The input data is divided into as many parts as there are PEs. Every PE then executes the same sequence of kernels.

It is implemented using the heuristic. A partition and schedule are generated for the graph for a *single* PE. This way some IPC can be aggregated. Processing can be done in parallel and therefore the time spent on processing is divided by the number of PEs. Also the amount of time spent on transferring data between external memory and local memory is determined. Data transfers cannot be performed in parallel, since they require shared resources. This results in the following approximation for the makespan of the schedule:

$$t_{total} = \sum_{g \in gangs} \max(\frac{t_{single}(g)}{|\Pi|}, tt_{ext}(g)) \tag{6.1}$$

Where $|\Pi|$ is the number of PEs and $t_{single}(g)$ denotes the makespan of the schedule of gang g for a single PE. The value of $tt_{ext}(g)$ equals the time spent on transfers from and to external memory in gang g. What the equation shows, is that the makespan of the complete schedule, t_{total}, equals the sum of the makespans of the individual gangs. For every gang, the makespan is the maximum of either time spent on processing or on transfers.

What can be seen from this equation is that it is only advantageous to increase the number of PEs when the processing time exceeds the time required for data transfers. This is in contrast with the proposed method in this thesis, where IPC aggregation is an important method for reducing the makespan.

The value that is found with this equation is an approximation, since no scheduling of the transfers is performed and input might not be perfectly dividable by the number of PEs. Also, filter kernels would require extra transfers for the required overlap. The significance of this is expected to be small. A 5x5 filter and division over four PEs would result in an overlap of 16 lines, which is 1.5% when considering a 1080x1920 image.

6.4.2 Naive list scheduling approach

One of the main decisions for the heuristic is that a node is mapped to one PE, which is equal for every iteration of a node. Also, a kernel is loaded only once per node resulting in a partitioning into gangs. This is deemed better than loading kernels multiple times, because the loading requires the use of the DMA, which has two disadvantages. First of all, during kernel loading the DMA is unavailable for transferring data, which can block execution of kernels on other PEs. Second, the PE cannot be used for executing kernels during this kernel loading.

To quantify how the chosen solution compares to a solution that does allow this dynamic kernel loading, a naive list scheduling approach is implemented. Different iterations of a node can be executed on different PEs. Kernels are executed as soon as there is enough input data available. The usage of the DMA is arbitrated by the following rules, ordered from high to low priority:

1. Transfer the input for the lowest level node, that is currently loaded to a PE. This can only be done if the input is available in the local memory of another PE or external memory and if there are sufficient free buffers.
2. Load the kernel of the lowest level node, for which input is available, to an unused PE.
3. If on all PEs a kernel is loaded then check if a PE is idle. If this is the case then replace it if all its buffers are empty. If there is data in an output buffer then transfer that first to DDR.
4. If none of the above can be done then input tokens of an idle PE are transferred to DDR. This way loading another kernel is enabled and deadlocks are avoided.

For the buffer sizes, the decision is made to assume triple input buffering. This is expected to have better performance than double buffering, because it allows for having more available input when the DMA is blocked due to context

switching of another PE. For filter kernels the first rate, i.e. the "absorbed" tokens, is added to the amount of input buffers. The rest of the buffer space is used for output buffers.

6.4.3 Results

Figure 6.5 shows the results for all benchmark graph. It shows the four different approaches and also indicates the fraction of the makespan that is used for external transfers. The yellow remaining parts comprehense the time spent on local to local memory transfers and the processing that is not overlapped by transfers to or from external memory. The scheduling times can be read from figure 6.4 for topological sorting.

Sequential approach

In the tiling approach all intermediate data makes the roundtrip to external memory. For this reason the time spent on external transfers is maximal. What can be seen from figure 6.4 is that all single-node gangs for all graphs have bandwidth-limited performance, except for some of the heavier kernels of the HDR graph.

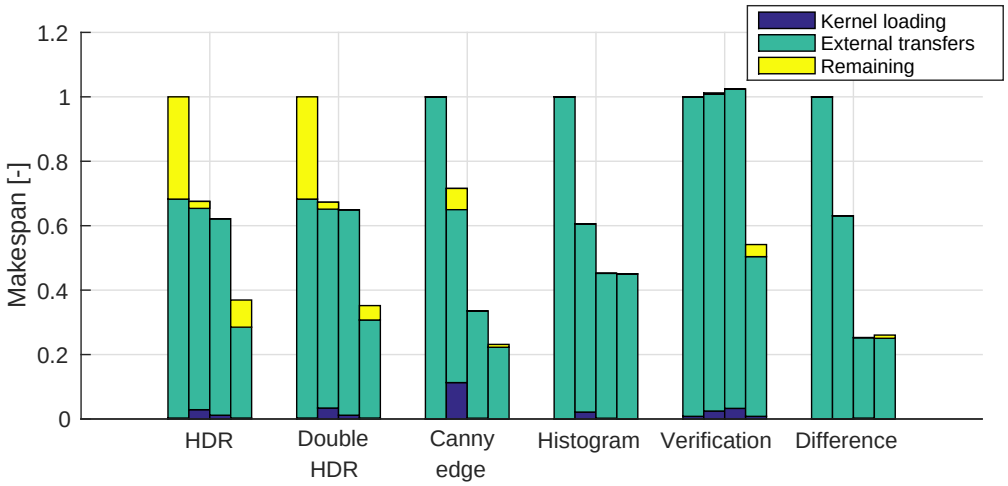


Figure 6.5: Relative makespans for all benchmark graphs. The four bars correspond, from left to right, to the sequential, naive list scheduling, tiling and heuristic approach. The remaining fraction consists of the processing and internal transfers that are not overlapped by the external transfers and kernel loading.

Tiling approach

It was shown (with equation 6.1) that the tiling approach only results in better performance, when external memory bandwidth is not the bottleneck. The performance is improved by (i) a decrease in the makespan when scheduled for a single PE using the heuristic and (ii) dividing the processing over multiple cores in case single PE execution is not bandwidth-limited. The breakdown of the achieved speedups for this approach are given in table 6.2. What can be

seen in this table is that the speedup for the tiling approach is mainly the result of the IPC aggregation for the single PE schedule and that none of the graphs benefit from adding a third or fourth PE.

Graph:	**HDR**	**HDRx2**	**CE**	**Hist**	**Verif.**	**Diff**
Single PE speedup	1.05	1.07	2.68	2.22	1.00	2.84
$\mathbf{1}/tt_{ext}$	1.54	1.46	1.11	1.00	1.00	1.41
Speedup	1.6	1.6	3.0	2.2	1.0	4.0

Table 6.2: Breakdown of the speedup of the tiling approach. Here tt_{ext} is the fraction of the makespan spent on transfers from and to external memory for the single PE schedule.

For the verification graph the tiling approach is even worse than the sequential approach. The reason for this is that there is no IPC aggregation for the single PE schedule, due to the single kernel per PE for this graph. On top of that, four times the loading time is incurred because of the four PEs.

The performance for the histogram equalization is equal, and for the difference highlighting graph even slightly better than the heuristic. The reason for this is that in the case of the histogram graph, due to the non-pipelineable nodes the partition consists of two small gangs of which all kernels fit in the program memory of a single PE. For this reason the amount of external memory transfers is equal for both approaches and since this is the bottleneck, performance is equal.

Something similar holds for the difference highlighting graph. All kernels fit in the program memory of a single PE, resulting in a single gang for which the computation time exceeds the transfer time. In the tiling approach this compute-bounded single PE schedule results in a 1.4x speedup when it is executed on 2 PEs or more. The heuristic performs the processing in kernels in parallel by mapping kernels to different PEs. This result in extra transfers for the local to local transfer, which results in a slightly larger makespan.

Naive list scheduler

The performance of the naive list scheduler is approximately equal to the performance of the tiling approach for HDR, double HDR and the verification graph. The reason for this is that in these graphs the kernel sizes are relatively large and little IPC aggregation can be achieved for a single PE. Therefore the DMA is not used excessively for kernel reloading, because the DMA can be kept busy with transferring input and output.

For graphs where the tiling approach benefits from IPC aggregation on a single PE (the higher values for single PE speedup in table 6.2), the performance of the naive list scheduler is clearly worse.

Heuristic

The differences between the heuristic and the tiling approach have already been discussed. On average, the heuristic results in a schedule with a makespan that is 33% better than the result for the tiling approach.

In general, the heuristic performs better than the tiling approach in case the number of gangs in the final partition is smaller. This results in smaller makespans, because the external memory bandwidth is the bottleneck, and

these external transfers exist for gang boundaries in the graph. The number of gangs is smaller when there is a chain of pipelineable nodes that does not fit in the program memory of a single PE.

In case such a chain does fit in the program memory of a single PE, similar performance can be achieved by balancing the processing on different PEs. If the gang is compute-bounded then the makespan of the gang is reduced by mapping nodes to different PEs. This can come at the cost of extra internal transfers and therefore results can be slightly worse than for the tiling approach approximation.

In case the kernels are strongly compute-bounded, the tiling approach will perform better than the heuristic, but given the set of OpenVX kernels that mainly consists of basic stencil operations, this is unlikely for the platform considered in this evaluation. This is supported by table 6.2, in which the speedup due to parallelization does not exceed 1.54.

6.4.4 Scheduler evaluation

In the scheduler, the retiming and buffer size are determined by performing multiple iterations of list scheduling and updating the retiming or buffer sizes. This is repeated until the optimal period is achieved or until it is determined that the optimal period cannot be achieved, given the buffer sizes. Since in the complexity analysis of section 5.2.3 there could not be put a bound on these numbers of iterations, it is evaluated here.

For the set of benchmark graphs it is measured how many iterations are needed to achieve the optimal period. Also the average amount of time it takes to execute this phase is measured for the number of iterations. The results are shown in figure 6.6. What can be seen from this is that the retiming phase is done after two iterations in 90% of the cases and that the buffer sizing phase is done after 3 iterations in 71% of the cases.

The graphs for which higher amounts of iterations are required, are the HDR and especially the double HDR graph. The number of iterations therefore seems to be dependent on the size of the gang. This can also be seen from the scheduling times, because the scheduling time seems to grow faster than linear for the number of iterations. With this data the total time spent on scheduling can be calculated. For the double HDR graph in total 12.8 ms of the 33 ms is spent on scheduling.

Another interesting statistic is that for only 1 of the 420 gangs the optimal period could not be achieved. For this gang it was determined after 4 iterations in the buffer sizing stage, that the optimal period could not be achieved. When increasing the image size to 4K resolution (3840x2160), resulting in more constrained buffer sizes, then in total 3 of the 435 gangs cannot achieve the optimal period. When increasing the image size to three times the width and height of full HD resolution, initial partitions can no longer be scheduled. What this means is that buffer size was insufficient for scheduling nodes indivdually, due to a lack of buffer space.

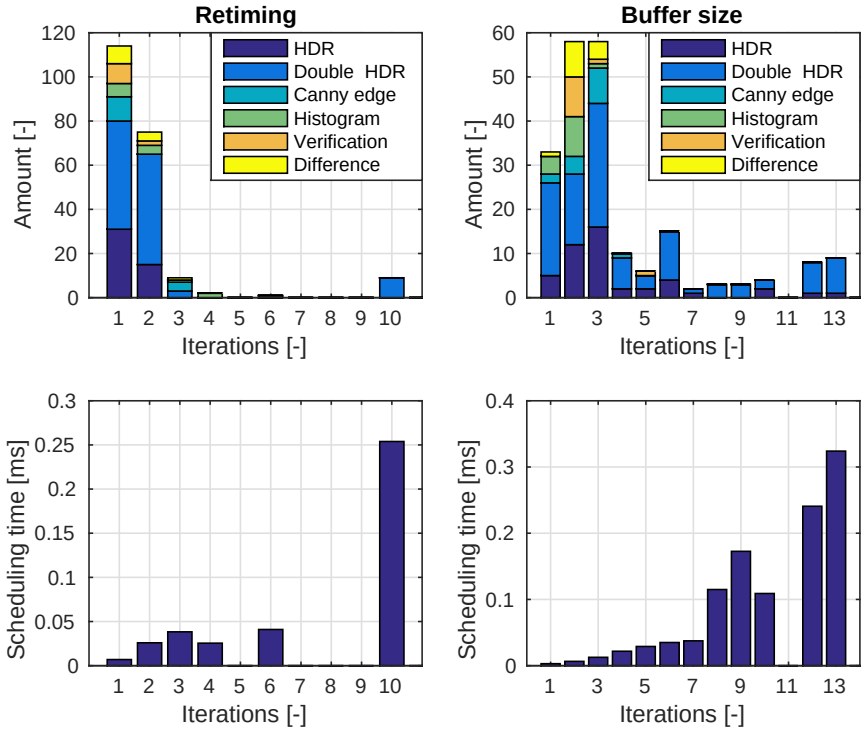


Figure 6.6: Overview of how many iterations are required in the retiming phase (left) and the buffer size (phase). The bottom figures show the average execution time of the phase for that amount of iterations, corresponding the bar graphs above them.

Chapter 7

Conclusion

This chapter recapitulates the properties of the proposed solution to the problem of this work and summarizes the most important results. An overview of the possibilities to continue and improve this work is given in section 7.1.

The main goal of this thesis is to enable application developers to accelerate their imaging and computer vision applications, in the form of OpenVX graphs, on a programmable ISP platform, without requiring manual porting. By specifying the application as an OpenVX graph, system-level optimizations are enabled.

The control flow for execution of the graph is generated. Control flow generation consists of two main steps. The first step is to generate the control flow in the form of an ordered list of instructions, i.e. scheduling. The second step is to transform this schedule into a control program.

The main performance bottleneck is the bandwidth of the external memory. The reason for this is that most kernels in the OpenVX specification are simple stencil operations, with a low compute-to-communication ratio, such as bitwise operations, absolute difference calculation and filters. Since the ISP supports vector operations, these computations can be performed efficiently.

Because no existing work is known that takes into account the full set of constraints, a new heuristic is designed and implemented. Without scheduling individual gangs, a good estimation on the reduction of IPC can be made. The system-level optimizations that resulted in improved performance are task-level pipelining and IPC aggregation, by making smart decisions during partitioning and mapping of the nodes. This reduced the amount of external transfers, which is the key to reducing the makespan of the complete graph.

The performance of the scheduling heuristic has been evaluated for different settings of the design parameters and showed strong dependencies on graph structure due to the locally optimal decisions. The results, obtained with the chosen configuration of design parameters, are compared to alternative approaches for a set of 6 benchmark graphs. One of these approaches is similar to [4], in which the data is divided over multiple PEs that run the same kernels.

Because most of the kernels that are currently in the OpenVX specification have a very low compute-to-communication ratio, most benchmark graphs were limited by external memory bandwidth. This does not show the performance for more compute-bounded OpenVX graphs that could be constructed with new kernels in future versions of OpenVX. The HDR graph, which includes

some heavier non-OpenVX kernels, has displayed good performance for such graphs.

The scheduling heuristic showed an average makespan reduction of 33%, compared to the tiling approach, and even higher reductions when compared to the other solutions. Besides that, the scheduling time is acceptable for end users (at most 54 ms) due to the fast iterative list scheduling approach.

What must be noted on this result is that some graphs showed equal performance for the tiling approach and the heuristic. This is the case for graphs with non-pipelineable nodes, smaller graphs and kernels with low program memory requirements. The reason for this is that the number of gangs cannot be reduced, compared to the tiling approach. For this reason it makespan reductions are only present for graphs that do not have these properties, although performance is not worse than any of the other approaches for such graphs.

To be able to verify the schedule and to determine optimality of the final makespan found by the heuristic, also an SMT formulation of the problem was described. In the end it is not used for determining optimality, because there was no time for updating the implementation of the SMT problem generator with some required recent changes.

To verify the correctness of the ordering of the generated control flow, the schedule of a verification graph has been transformed into a control program. This control program has been simulated for the platform and the correctness of the control flow was determined. With this proof of concept, also realistic values for kernel execution times were obtained.

A weak part of the novel heuristic is the mapping algorithm, that is currently determined by an algorithm that has exponential complexity. Until now this has not resulted in problems, but when the program memory-kernel size ratio grows the number of nodes in a gang can grow and mapping might become a bottleneck.

7.1 Future work

Although an implementation and evaluation of the heuristic have shown good results, there are aspects that can be improved. Also other parts of this thesis could be extended. The list of work that is deemed interesting for future research and improvement is:

- *Replacing the exponential-time mapping algorithm by a heuristic.* It is expected to be possible to determine an effective polynomial-time algorithm that finds this mapping, based on IPC aggregation. This way, scalability of the heuristic is improved for platforms where more kernels fit in the program memory.
- *Combining the tiling approach of [4] with the heuristic.* In its simplest form a pipelined schedule for two cores can be generated that is loaded on two pairs of PEs. This way the advantages of both approaches are combined.
- *Implementation and evaluation of an interpreter control program.* Because a control program code generator was already partly implemented, this was used for the proof of concept. In practice compiling the control program at configuration time can be the bottleneck in the generation of the control flow and therefore an interpreter program might be preferable.

- This scheduler only targets the ISP. It is very interesting to see how this scheduler could be used in control flow generation for executing the graph on *multiple (different) accelerators*. However, the external memory bandwidth bottleneck will still remain.
- The *implementation of the SMT problem generator should be extended* with the described constraints for filter kernels. With this implementation it can be checked if the gangs, for which the scheduling heuristic did not achieve the optimal period, cannot be scheduled or if the heuristic is failing.

Appendices

Appendix A

Satisfiability Modulo Theories formulation

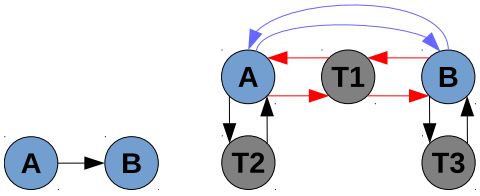
Knowledge of the optimal solution is interesting for quantifying how well the heuristic performs relative to optimal. The optimal solution can be determined by formulating the problem as a Satisfiability Modulo Theories (SMT) problem and solving it. SMT problems are decision problems where the satisfiability of a set of constraints, expressed in first-order logic with respect to background theories such as integers, reals or arrays, must be determined. Having the SMT formulation of the problem also allows us to verify the solution found by the heuristic. The SMT problem formulation can then be solved by SMT solvers that support the required background theories.

This appendix first describes how graphs are represented. Then the variables and constraints, for the combined scheduling and partitioning problem, are given. After that it is shortly described how the two subproblems, partitioning a graph and scheduling a gang, can be formulated individually and how the optimal result can be found using existing SMT solvers.

A.1 Graph representation

Since the existence of a transfer is dependent on both the mapping and partition, a representation that can capture this behavior must be used. For this reason the CUKG will be expanded into a representation that consists of all nodes from the CUKG, plus all *possible* transfer nodes. This is achieved by adding three transfer nodes and the corresponding edges for every edge in the CUKG. This is illustrated in figure A.1, where a CUKG edge (A.1a) is expanded into the edges and nodes as shown in figure A.1b. Node T1 represents the internal transfer and T2 and T3 represent transfers between local and external memory. By making the constraints, implied by the nodes and edges, dependent on the partitioning and mapping, the situations as displayed in figure A.2 can be achieved:

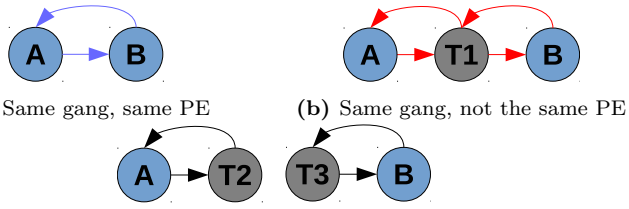
- In case node A and B of figure A.1a are in the same gang and mapped to the same PE, the constraints on the blue edges of figure A.1b apply. This results in the situation of figure A.2a.
- In case node A and B of figure A.1a are in the same gang, but mapped to different PEs, the constraints on the red edges and on node T1, of figure



(a) Nodes with a dependency in the CUKG

(b) Created nodes and edges for the CAKG

Figure A.1: Modelling the tranformation from CUKG to CAKG in the SMT problem



(a) Same gang, same PE

(b) Same gang, not the same PE

(c) Not in the same gang

Figure A.2: Resulting nodes and edges from disabling nodes and edges according to the mapping and gang assignment

A.1b apply. This results in the situation of figure A.2b.

- In case node A and B of figure A.1a are not in the same gang the constraints on the black edges, and on the nodes T2 and T3 of figure A.1b apply. This results in the situation of figure A.2c. Note that in this case the temporal ordering is captured in the fact that the gang of node A is executed before the gang of node B.

A transfer node or edge that is present due to the mapping and partition of A and B is called an **enabled** node or edge.

The properties that are used to formulate the constraints are, for every node n and edge e, listed in tables A.1 and A.2 respectively. The values assigned to the new transfer nodes result from the kernels that nodes A and B implement. The execution time $t_{exe}(n)$ of the transfer nodes, represents the time it takes to transfer the data that one token represents.

The repetition vector $q(n)$ of the added nodes is defined according to the definition of the repetition vector in section 2.4.1. Since transfer nodes do not require any program memory, the program size $size(n)$ is 0. The property $is\_transfer(n)$ equals true for both added nodes and false for the CUKG nodes. The properties of the added edges follow from the CUKG to CAKG transformation, as described in section 3.1.4. Also note that the gang, in which the added nodes are executed, matches the gang of node A and B for T1, the gang of A for T2 and the gang of B for T3, if the nodes are enabled.

Also some constants are used to describe the platform parameters:

- #PEs; the number of processing elements that the platform consists of.
- PMEM_SIZE; the size of the program memory of each PE.
- BUF_SIZE; the size of the local memory of each PE where the buffers are allocated.

Property	Meaning
$t_{exe}(n) \rightarrow int$	Execution time of the node.
$is\_transfer(n) \rightarrow bool$	True or false corresponding to if the node represents a data transfer or kernel execution.
$q(n) \rightarrow int$	Corresponding value in the repetition vector.
$size(n) \rightarrow int$	Size of the program in bytes.

Table A.1: Constant node properties

Property	Meaning
$snk(e) \rightarrow int$	The sink node of edge e.
$src(e) \rightarrow int$	The source node of edge e.
$p(e) \rightarrow int$	The number of tokens produced after one execution of the source node.
$c(e) \rightarrow int$	The number of tokens consumed by one execution of the sink node.
$is\_buffer(e) \rightarrow bool$	Whether or not the edge is a buffer edge or not.
$tok\_size(e) \rightarrow int$	The size of a token in bytes.
$pipelineable(e) \rightarrow bool$	Indicates whether or not this output can be pipelined.

Table A.2: Constant edge properties

A.2 Problem formulation

The input for the problem is the expanded CUKG as explained in section A.1. The goal is to determine if a partition, mapping and values for all start and synchronization times exist, such that all platform and dependency constraints are met, and a given cost is achieved. First the list of variables, to which a value must be assigned, is given. Afterwards the constraints on these variables are listed.

A.2.1 Variables

The input of the problem consists of all nodes and edges, including backedges that represent buffers, for which all constant properties are specified. The variables and *uninterpreted functions*, which are equivalent to arrays of variables, that are used in the problem formulation are listed in table A.3. Finding a solution means assigning a value to the variables in this table that do not violate the constraints of the next section.

Note that the specified ranges are implemented as constraints on the values, but that these constraints are not listed in the next section because they are trivial. If no range is specified in the table it means that the range of the variable is limited by the constraints of the next section.

Some constraints on transfer nodes and edges should only hold if the edge or node is actually present, i.e. enabled. For example, constraints on transfer nodes of type T1 of figure A.1, and on its input and output edges, only apply if node A and node B are in the same gang and mapped to the same hardware. In the constraints, whether a node or edge is enabled is denoted as $en_n(n)$ or

Variable	Meaning	Range
$start(n,k) \rightarrow int$	The starting time of a iteration k of node n, with $k \in [0, q(n)-1]$.	≥ 0
$sync(n,k) \rightarrow int$	The synchronization time of a iteration k of a node n, with $k \in [0, q(n)-1]$.	≥ 0
$d(e) \rightarrow int$	The number of tokens on an edge e at the start of the repeating pattern.	
$r(n) \rightarrow int$	The retiming of a node n.	≥ 0
$num\_buf(e) \rightarrow int$	The number of tokens on an edge e before any task execution, i.e. the number of buffers.	≥ 0
$req\_buf(n) \rightarrow int$	The total buffer size required for a node n.	≥ 0
$hw(n) \rightarrow int$	The hardware that the node n is mapped to.	
$gang(n) \rightarrow int$	The gang in which the node n is executed.	
$cost(g) \rightarrow int$	Cost of a gang g.	≥ 0
$en_e(e) \rightarrow bool$	Indicates if edge e is enabled.	
$en_n(n) \rightarrow bool$	Indicates if node n is enabled.	

Table A.3: Variables in the SMT problem.

$en_e(e)$, respectively.

A.2.2 Constraints

The logical formulas describing the problem mostly correspond to the formalization of the problem in section 3.1. First the constraints that determine the initial amounts of tokens are defined, followed by the constraints on the partition, mapping and memory requirements. Then the typical scheduling restrictions due to data dependencies are given. After that, the constraints that relate the total cost and the individual cost of gangs to the other constraints, are described.

For notation the following convention will be used:

$$\forall x \in X : C \mid E$$

Which reads as "for all elements x in set X, such that condition C is satisfied, expression E" must hold. The condition C, which selects a subset of X, is optional and could also be implemented by an implication inside expression E. The convention that is used in the rest of this section is that, when a condition can be evaluated at problem generation time, e.g. a condition $is\_transfer(n)$, it is placed at the position of C. If the condition is not known at problem generation time it is denoted using implication in expression E, for example $en_e(e)$. An example of a constraint featuring both is constraint A.2.

Initial token constraints

First the constraints that define the situation at the start of the repeating pattern are given. This mainly concerns the amount of tokens on all edges.

The number of buffer tokens is zero for every non-buffer edge by definition. In table A.3 the range for buffer edges was already constrained to any integer ≥ 0, since negative amounts of tokens cannot exist in dataflow.

$$\forall e \in edges : \neg is\_buffer(e) \mid num\_buf(e) = 0 \tag{A.1}$$

Also disabled buffer edges are forced to have zero buffers. Although this is not strictly necessary to check satisfiability, it makes the assignment of values match reality, since allocating buffers for non-existing transfers (if there happens to be free buffer space) has no function and might lead to confusion.

$$\forall e \in edges : is\_buffer(e) \mid \neg en_e(e) \rightarrow num\_buf(e) = 0 \tag{A.2}$$

Constraint A.1 defines the amount of tokens on edges *before* executing the preamble. The amount of tokens *after* the preamble, $d(e)$, can be determined with equation 3.1, which describes the retiming relation. The constraint on $d(e)$ is then given by:

$$\forall e \in edges \mid d(e) = p(e)r(src(e)) - c(e)r(snk(e)) + num\_buf(e) \tag{A.3}$$

Note that $num\_buf = 0$ for every non-buffer edge. Also note that no range on the number of delays is given in table A.3. These should intuitively all be greater than or equal to 0. However, in combination with constraint A.3 this would imply unnecessary (and too restricting) constraints on the retiming of nodes that are not enabled. Therefore the range of the delay tokens is only enforced for enabled nodes:

$$\forall e \in edges \mid en_e(e) \rightarrow d(e) \geq 0 \tag{A.4}$$

For CSDF, constraint A.3 needs to be slightly adapted. Recall from section 2.4 that only the first and the last rates deviate from a constant rate for filter kernels. In section 5.3.1 it was already determined that the executions with a rate that deviates from the constant rate, are limited to execution in the preamble and postamble. This means that the retiming of a CSDF node n has a constraint on its range $r(n)$ (≥ 1 for a 3x3 box filter), and the number of tokens follows equation 3.2. The rates are known at the moment that the SMT problem formulation is generated. As an example, for a non-buffer input edge of a 3x3 filter node this results in:

$$\forall e \in edges \mid d(e) = p(e)(r(src(e))) - (1 + c(e)(r(snk(e)) - 1)) + num\_buf(e)$$

Partitioning constraints

In table A.3 no range is specified for gangs. This is not done since the range is different for transfer nodes and processing nodes. The maximum number of gangs can be determined with the number of processing nodes $\#n_{proc}$, since in the worst case scenario all nodes have to be executed in a separate gang.

Processing nodes will always be in a gang. Transfer nodes are in a gang dependent on the adjacent processing nodes. Disabled nodes are in dummy gang 0, on which no further constraints that affect the cost are applied. Disabled nodes are forced to be in gang 0, because not constraining their gangs would leave the possibility of assigning it to a gang in the range $[1, \#n_{proc}]$. This is not a problem when trying to prove (un)satisfiability, but it could give a model with unrequired transfer nodes when inspecting the assigned values in case of a satisfiability.

$$\begin{aligned} &\forall n \in nodes : \neg is\_transfer\_node(n) \mid gang(n) \in [1, \#n_{proc}] \\ &\forall n \in nodes : is\_transfer\_node(n) \mid \neg en_n(n) \rightarrow gang(n) = 0 \end{aligned} \tag{A.5}$$

Following figure A.2, all nodes of type T1 are in the same gang as node A and B, if node A and B are in the same gang and mapped to the same hardware. Otherwise the node is disabled.

$$\begin{aligned}&\forall e_1, e_2 \in edges : src(e_2) = snk(e_1) = n_T \subseteq \text{T1} \wedge n_1 = src(e_1) \wedge n_2 = snk(e_2) \mid \\ &\quad (gang(n_1) = gang(n_2) \wedge hw(n_1) \neq hw(n_2) \rightarrow gang(n_T) = gang(n_1)) \wedge \\ &\neg(gang(n_1) = gang(n_2) \wedge hw(n_1) \neq hw(n_2)) \rightarrow gang(n_T) = 0\end{aligned} \tag{A.6}$$

Where T1 is the set of all nodes of type T1. Similarly for nodes of type T2 and T3 it must hold that node A and B are not in the same gang. Otherwise the nodes are disabled.

$$\begin{aligned}&\forall e_1, e_2 \in edges : src(e_2) = snk(e_1) = n_T \subseteq \text{T2} \cup \text{T3} \wedge n_1 = src(e_1) \wedge n_2 = snk(e_2) \mid \\ &\quad ((gang(n_1) \neq gang(n_2) \rightarrow gang(n_T) = gang(n_2)) \wedge \\ &\quad\ gang(n_1) = gang(n_2) \rightarrow gang(n_T) = 0\end{aligned} \tag{A.7}$$

Where T2 and T3 are the sets of all nodes of type T2 and T3 respectively.

The gangs are sorted chronologically, meaning that gang g precedes all gangs with an ID $> g$. This means that if there is a directed edge (data dependency), then the sink node cannot be in a gang preceding the gang in which the source node is executed. Since intermediate transfer nodes can be in gang 0 (constraint A.5), this constraint will only be applied to the source and sink nodes of original edges of the CUKG $edges_{CUKG}$.

$$\forall e \in edges_{CUKG} \mid gang(src(e)) \leq gang(snk(e)) \tag{A.8}$$

If the data is transferred over an edge that does not allow for pipelining, then the processing nodes cannot be in the same gang. Again this constraint is only applied to edges of the CUKG.

$$\forall e \in edges_{CUKG} : \neg pipelineable(e) \mid gang(src(e)) \neq gang(snk(e)) \tag{A.9}$$

Mapping

It is decided that the unique hardware IDs, to which nodes are mapped, of PEs are in the range $\Pi = [1, \#\text{PEs}]$.

$$\forall n \in nodes : \neg is\_transfer\_node(n) \mid hw(n) \in \Pi \tag{A.10}$$

The ID of the DMA equals $\Phi = \#\text{PEs} + 1$. All transfer nodes are mapped to this node if they are enabled.

$$\forall n \in nodes : is\_transfer\_node(n) \mid en_n(n) \rightarrow hw(n) \in \Phi \tag{A.11}$$

Disabled nodes are mapped to hardware with ID 0, on which no cost affecting constraints are applied, similar to constraint A.5.

$$\forall n \in nodes : is\_transfer\_node(n) \mid \neg en_n(n) \rightarrow hw(n) = 0 \tag{A.12}$$

Enabled nodes and edges

Now that the representation of the mapping and partition is explained the constraints, on when an edge or node is enabled, can be given.

Checking if a transfer node is enabled is done by gang. If a node n is disabled then $gang(n) = 0$ as a result of constraints A.6 and A.7. All processing nodes are enabled by default. For the transfer nodes the gang is compared to 0.

$$\begin{aligned} &\forall n \in nodes : \neg is\_transfer(n) \mid en_n(n) = true \\ &\forall n \in nodes : is\_transfer(n) \mid en_n(n) = \neg(gang(n) = 0) \end{aligned} \tag{A.13}$$

For all edges where the sink or source node is a transfer node, the edge is enabled if both the sink and source node are in the same gang. This is enough, since either the sink or source node is a processing node, which are always enabled. If the transfer node is in the same gang, then it must be enabled.

$$\begin{aligned} &\forall e \in edges : is\_transfer(src(e)) \vee is\_transfer(snk(e)) \mid \\ &\quad en_e(e) = (gang(src(e)) = gang(snk(e))) \end{aligned} \tag{A.14}$$

The blue edges from figure A.1b should only be enabled when both the sink and source nodes, which are both processing nodes, are in the same gang and mapped to the same hardware.

$$\begin{aligned} &\forall e \in edges : \neg is\_transfer(src(e)) \wedge \neg is\_transfer(snk(e)) \mid \\ &\quad en_e(e) = (gang(src(e)) = gang(snk(e)) \wedge hw(src(e)) = hw(snk(e))) \end{aligned} \tag{A.15}$$

Memory constraints

The sum of the program binary sizes, of all nodes in a gang that are mapped to the same hardware, must fit in the program memory.

$$\forall g \in gangs, p \in \Pi \mid \text{PMEM\_SIZE} \geq \sum_{n \in nodes \mid hw(n)=p \wedge gang(n)=g} size(n) \tag{A.16}$$

The total number of buffers on all in- and outgoing edges of a processing node, multiplied by their size, gives the required buffer size for a processing node. Note that $num\_buf(e) = 0$ for disabled edges.

$$\begin{aligned} &\forall n \in nodes : \neg is\_transfer\_node(n) \mid \\ &\quad req\_buf(n) = \sum_{\substack{e \in edges \mid (src(e) = n \vee snk(e) = n) \wedge \\ \neg(src(e) = n \wedge \neg is\_transfer(snk(e))}} num\_buf(e) * tok\_size(e)) \end{aligned} \tag{A.17}$$

The sum of the required buffer sizes of all nodes in the same gang that are mapped to the same hardware should be smaller than the available buffer size.

$$\forall g \in gangs, p \in \Pi \mid \sum_{n \in nodes \mid gang(n)=g \wedge hw(n)=p} req\_buf(n) < \text{BUF\_SIZE} \tag{A.18}$$

Note that with the combination of constraints A.17 and A.18 the buffers on the blue edges of figure A.1b are accounted for by both node A and node B. This means that the expected buffer size is too large. To overcome this problem buffer tokens on edges between two processing nodes are only accounted for by the source node.

Data dependency constraints

The constraints that enforce the ordering of actor firings according to synchronous dataflow semantics, resulting in a valid schedule, are given here.

The time between start and synchronization of a task is greater than or equal to the execution time of the kernel or transfer:

$$\forall n \in nodes, k \in [0, q(n) - 1] \mid sync(n, k) \geq start(n, k) + t_{exe}(n) \tag{A.19}$$

The implemented constraint is $sync(n, k) = start(n, k) + t_{exe}(n)$, because it reduces the search space (and therefore the solver's execution time) without excluding satisfiability. The reason for this is that the time of synchronization does not matter, as long as it is before the start of the executions that depend on it by shared resource or produced (buffer)tokens, and after finishing the task's execution. Since synchronizing as early as possible is always at least as good as synchronizing later, satisfiability is never excluded, when synchronizing as soon as possible.

Ordering of all iterations of nodes, between which there are enabled edges, must adhere to the dependency constraint, given by equation 3.3. This constraint is also applied to the buffer edges, which results in the backpressure in the SMT formulation.

$$\begin{aligned} &\forall e \in edges \mid (\forall k \in [0, q(src(e)) - 1] \mid en_e(e) \rightarrow \\ &\quad start(snk(e), k) \geq sync(src(e), \left\lceil \frac{(k+1)c(e) - d(e) - p(e)}{p(e)} \right\rceil)) \end{aligned} \tag{A.20}$$

Again, this is slightly different for CSDF nodes. For filter nodes the relation between the iteration of the start and the synchronization is given by equation 3.4.

There can be no overlap of executions of processing nodes in the same gang that are mapped to the same processing element.

$$\begin{aligned} \forall n_1, n_2 \in &nodes : \neg is\_transfer\_node(n_1) \wedge \neg is\_transfer\_node(n_2) \wedge n_2 > n_1 \mid \\ &(\forall k_1 \in [0, q(n_1) - 1], k_2 \in [0, q(n_2) - 1] \mid hw(n_1) = hw(n_2) \rightarrow \\ &(start(n_1, k_1) \geq sync(n_2, k_2) \vee start(n_2, k_2) \geq sync(n_1, k_1))) \end{aligned} \tag{A.21}$$

The same holds for transfer nodes that are in the same gang and mapped to the DMA.

$$\begin{aligned} \forall n_1, n_2 \in &nodes : is\_transfer\_node(n_1) \wedge is\_transfer\_node(n_2) \wedge n_2 > n_1 \mid \\ &(\forall k_1 \in [0, q(n_1) - 1], k_2 \in [0, q(n_2) - 1] \mid hw(n_1) = hw(n_2) \rightarrow \\ &(start(n_1, k_1) \geq sync(n_2, k_2) \vee start(n_2, k_2) \geq sync(n_1, k_1))) \end{aligned} \tag{A.22}$$

Constraint A.21 and A.22 are not combined since the hardware IDs do not overlap. This way, many useless constraints are avoided.

Cost

The cost of a gang equals the best achievable period of the repeating pattern in the gang. This can also be denoted the other way around, resulting in less constraints; the synchronization time of the latest iteration of each node restricts the cost of the gang that the node is in.

$$\forall n \in nodes \mid cost(gang(n)) \geq sync(n, q(n) - 1) \tag{A.23}$$

The total cost is then given by the sum of the costs of all gangs.

$$total\_cost = \sum_{g \in gangs} cost(g) \tag{A.24}$$

This total cost is checked against a specified value.

$$total\_cost \leq \text{OBJECTIVE\_VALUE} \tag{A.25}$$

A.3 Solving the subproblems

Solving the combined scheduling- and partitioning problem easily becomes impractical when the number of nodes of a graph grows, and values for the repetition vector become higher. Solving the subproblems individually can still give useful leads to how close the result of the heuristic is to optimal. For example, when the heuristic finds a schedule with a cost that is equal to the optimal cost of the partitioning-only SMT problem then optimality is still proven, because the SMT subproblem uses the lower-bound for the costs (i.e. periods) of the gangs. Also, after finding this partition the optimal period of a gang can be found by solving the scheduling problem separately.

This section explains how the subproblems can be constructed in an efficient manner.

A.3.1 Scheduling

The goal of the scheduling subproblem is to find values for all start- and synchronization times *for one gang*, such that all platform and data dependency constraints are met. The most straightforward adaptation of the combined problem formulation to achieve this is to add constraints that enforce that all processing nodes are in the same gang.

Although this does the job, it can be made more efficient. For example; in the separate scheduling problem all nodes are in the same gang by definition, the black edges and nodes T2 and T3 of figure A.1b, and all constraints associated with them, are no longer required and can be removed.

Also, all conditions using the $gang(n)$-variable can be removed even though it is to be expected that the solvers can derive these simplifications internally, when they are set to a fixed value.

A.3.2 Partitioning

The goal of the partitioning subproblem is to find a partitioning plus mapping, with which a cost is associated that is less than or equal to a specified value, if such an assignment exists given the platform constraints.

This subproblem abstracts from the actual scheduling, meaning that the cost is no longer the latest synchronization time of an execution in the gang. Instead, the cost of a gang is now equal to the optimal period, which is greater than or equal to the sum of the execution times, multiplied by the repetition vector entry, of all nodes in the same gang mapped to the same hardware (PE or DMA).

$$\forall g \in gangs, h \in \Pi \cup \Phi \mid cost(g) \geq \sum_{n \in nodes | gang(n)=g \wedge hw(n)=h} q(n)\ t_{exe}(n) \tag{A.26}$$

Abstracting from scheduling means that constraints A.1 through A.4 and A.17 through A.23 are no longer required.

A.4 Solving the SMT problem

Now that the problem is formulated as an SMT problem it can be checked for satisfiability. Since the problem consists of equations mostly concerning integers, but also one equation concerning reals (constraint A.20), a solver supporting the theories of real- and integer arithmetic is required.

Some well-known SMT solvers that support these background theories are CVC4[27] (joint project of New York University and University of Iowa), Z3[28] (Microsoft Research) and Yices[29, 30] (SRI International), which are all participating in the International Satisfiability Modulo Theories Competition[1] (SMT-COMP). This annual competition is part of the Conference on Computer Aided Verification (CAV). There are more possible solvers, but an extensive comparison is not the purpose of this work. Therefore, these three solvers have been used to solve a testbench and Yices, which has been used before for solving modulo scheduling problems before [9], was determined to be the most suitable (fastest) for solving the problem presented here.

The output of the solvers is whether the problem is satisfiable or not. If the problem is satisfiable then a list of the values that are assigned to the variables is also part of the output. A result of the nature of SMT problems is that the solvers cannot directly find the optimal value of a specific variable. However, optimality can be proven if two consecutive objective values are found, for which one yields unsatisfiable and the other one yields satisfiable.

[1] http://smtcomp.sourceforge.net/2015/index.shtml, Website of SMT-COMP 2015, consulted on February 26th 2016

Appendix B

Scheduling example

In this appendix an explanation of the steps that are taken in the scheduling heuristic are shown. It is deemed most interesting to inspect the scheduling of a gang that:

- Needs multiple iterations for determining the retiming and buffer sizes.
- Is compute-bounded. Strongly communication-bound gangs have a very low PE utilization and often uses only one PE to maximize IPC aggregation.
- Does not consist of too many node iterations to keep it understandable.

For this reason the gang model of figure B.1 is taken from an iteration of the HDR graph, which requires 2 iterations in the retiming phase and 5 iterations in the buffer sizing phase. In this figure the bold numbers in the nodes are the node IDs and the values below them are execution times. The nodes with the more intense colors are processing nodes. The lighter color nodes are the transfers nodes. Different colors mean that nodes are mapped to different PEs (except for the transfers nodes), so in this case all processing nodes are mapped to different PEs. The tokens with the "i" in it are the number of input tokens. This value is 1080 in this case for a full HD input image, but this does not influence the scheduling.

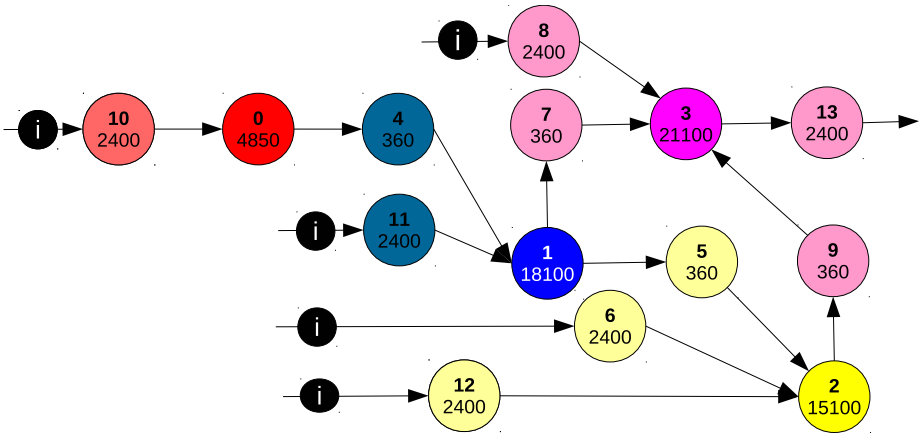


Figure B.1: Determining the retiming based on the number of preparations.

In this gang model all rates are equal to one. For this reason the repetition vector equals one for every node and each node is executed once in the loop body. For this reason the bottleneck can easily be determined. The maximum processing time on a PE is 21100 time units. The total transfer time equals 15840 time units, so 21100 time units is the optimal period.

The remainder of this section follows the steps that the scheduler takes in chronological order. It shows the updated model after every iteration of the retiming phase. It will not go into detail concerning the list scheduler.

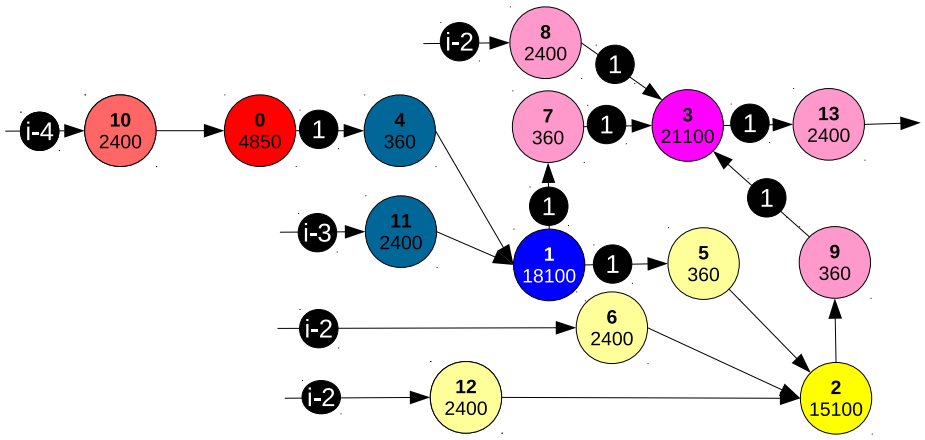
B.1 Retiming phase

In the first phase the retiming with which the optimal period can be achieved is determined. An initial required number of preparations is determined by looking for paths that exceed the optimal period. An explanation of preparations and converting them to retiming is given in the retiming phase description in section 5.3.2.

The first step is to get a starting point for the required preparations. Since node 3's execution time equals the optimal period, node 3 and 13 must be able to execute at the start of the period to be able to achieve the optimal period. Also nodes 4, 5 and 7 need a preparation to break the other paths with sums of execution times that exceed the optimal period. These required preparations are converted to the retiming and the values are shown in figure B.2a. This results in the model of figure B.2b.

ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13
prep	0	0	0	1	1	1	0	1	0	0	0	0	0	1
ret	4	3	2	1	3	2	2	2	2	2	4	3	2	0

(a) The table showing the required preparations and corresponding retimings for each node.



(b) The graph after applying the retiming.

Figure B.2: The preparations and retiming (a) and the graph with the applied retiming (b) for the first iteration of the retiming phase.

Scheduling this graph results in the Gantt chart of figure B.3. This Gantt chart is generated with freeware called Timedoctor. This tool does not flexible

in adapting the time labels. Therefore it has to be taken into account that 1 second in the Gantt chart corresponds to 10 time units.

To link the Gantt charts to the SDF model the colors are used. The processing nodes, with the more intense colors are executed on ISP0 to ISP3 and the nodes with a lighter color on the DMA.

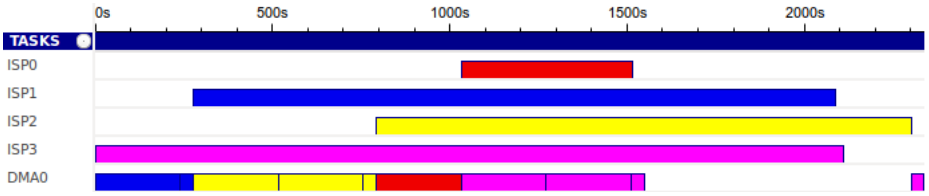


Figure B.3: The Gantt chart of the schedule of the loop body after the first iteration of the retiming phase.

What can be seen here is that all tasks finish within the optimal period of 21100 time units, except for the yellow node 2 and the pink node 9. Since the optimal period is not achieved, the preparations are updated. Updating the preparations is done by looking at the PEs on which task end times exceed the optimal period. For ISP2 this is the case and the maximum slack on this PE occurs before node 2 (because it is the only node on this PE). It then starts backtracking the slack. It does this by checking node 5 (the short yellow task on the DMA), which enabled the execution of node 2. No slack is present before node 5, so a preparation for node 2 is required.

After that the DMA is checked. The only node before which there is slack is node 9. Since it is dependent on the execution of node 2 backtracking is started through this node. It is found out that the slack before node 2 is already removed by adding a preparation for it. Because the slack before node 2 was higher than the required slack reduction on the DMA, no preparation for node 9 is required.

The resulting schedule of the loop body in figure B.4 shows that the optimal period is achieved now for the model of figure B.5. Therefore the next phase, where the buffer sizes are determined, can be started.

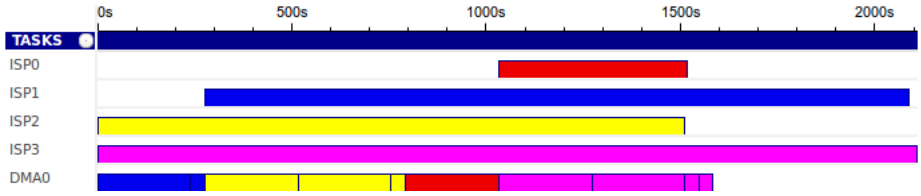
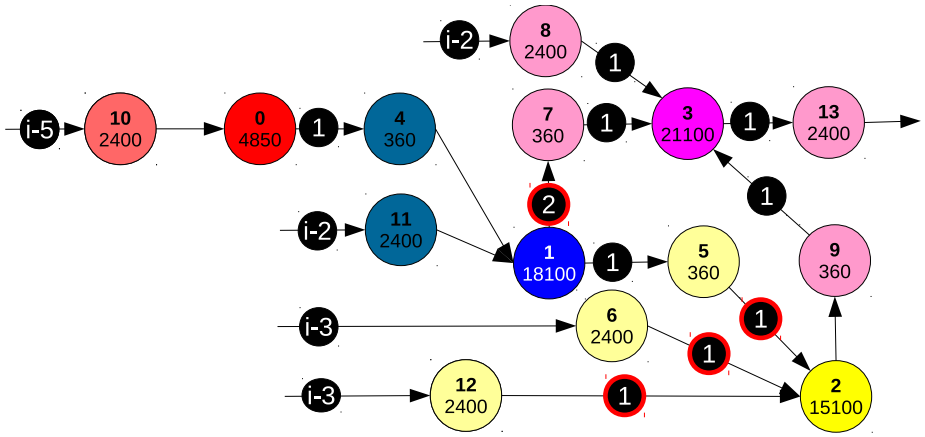


Figure B.4: The Gantt chart of the schedule of the loop body after the second iteration of the retiming phase.

ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13
prep	0	0	**1**	1	1	1	0	1	0	0	0	0	0	1
ret	5	4	2	1	4	3	3	2	2	2	5	4	3	0

(a) The table showing the required preparations and corresponding retimings for each node.



(b) The graph after applying the retiming of (a).

Figure B.5: The preparations and retiming (a) and the graph with the applied retiming (b) for the second iteration of the retiming phase.

B.2 Buffer sizing phase

In this second phase the buffer sizes are determined, such that the optimal period can be achieved. The retiming is fixed after the first phase. Similar to the first phase, an initial minimum buffer size is determined. Afterwards, the model is scheduled and updated, if the optimal period is not achieved.

For every non-source and non-sink edge a backedge is added. The initial amount of tokens is first set equal to the minimum required token sizes based on the rates at the sink and source. After this the preparations of the last retiming iteration are used to determine if there should be added more buffers initially. If the number of required preparations is larger than 0, this means that at the start of the loop period enough tokens should be available for starting the execution of the node. This also holds for the tokens on the backedges, i.e. the free buffer space.

After this, the model is checked for deadlocks by executing each node until the amount specified in the retiming plus the number of executions in the repetition vector is reached. If there are deadlocks, tokens are added to the buffer edges causing them. This results in the model of figure B.6 and the corresponding Gantt chart of the first iteration of figure B.7.

As can be seen in figure B.7, the optimal period is not achieved after the first iteration. Therefore the model is updated, which is done as follows. As described in section 5.3.2, information on blocking buffers is gathered during scheduling. Based on this the model is updated similarly to the retiming phase. For all hardware it is checked if the tasks that execute on it exceed the optimal period. If this is the case then the buffer edge, on which there is a lack of tokens, that causes the largest slack on the PE is determined. The buffers on this edge

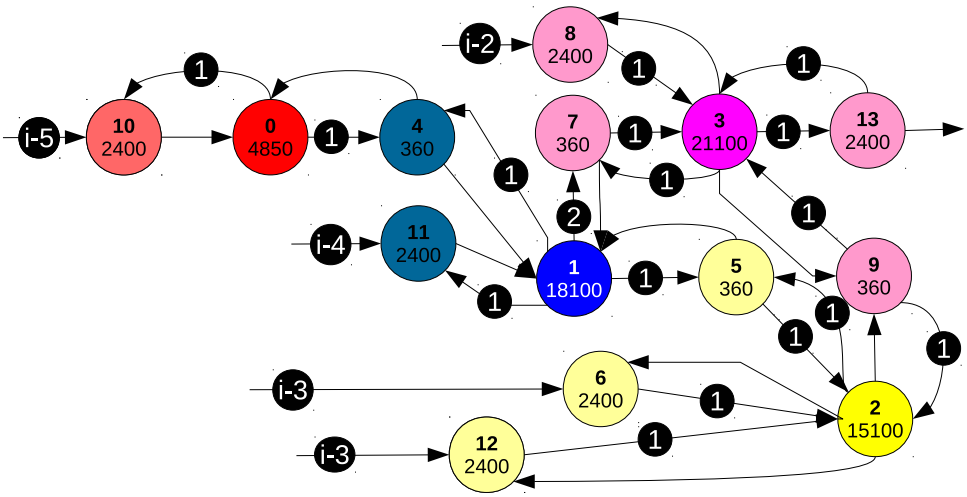


Figure B.6: The graph with the initial buffer sizes.

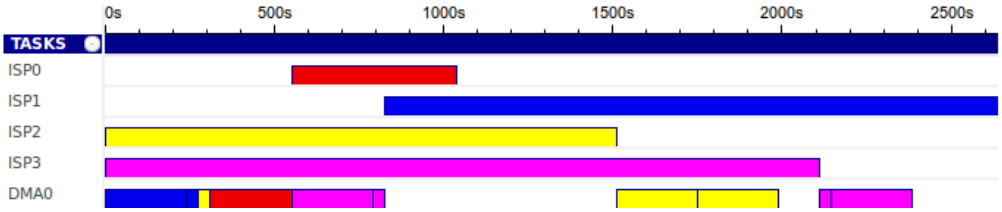


Figure B.7: The Gantt chart of loop body after the first iteration fo the buffer sizing phase.

are increased if the memory constraints allow for it. Again backtracking through tasks is performed to determine the best location to add buffers.

For the first iteration (see figure B.7) the optimal period is exceeded for ISP1 and the DMA. On ISP1, node 1 cannot execute because there is no buffer available on the backedge of node 7 to node 1. Since there is no slack before node 7 (the short pink task on the DMA that is executed before the start of node 1) a buffer is added on edge $(7, 1)$.

On the DMA the optimal period is also exceeded. The largest slack is encountered before the execution of node 6, which is caused by a lack of buffers on edge $(6, 2)$. Since the enabling execution of node 2 is not preceded by slack, this buffer size is increased. The length of the slack before the execution of node 6 is larger than the required decrease in slack. Therefore updating the buffers is finished and scheduling results in the Gantt chart of figure B.8.

Again it can be seen that the optimal period is not achieved. Adding a buffer for node 1 resulted in an earlier start time, but another blocking buffer is encountered now. There is a lack of tokens on edge $(5, 1)$. Backtracking does not lead to a better buffer to increase, so a buffer is added to this edge. Similar to the previous iteration, an extra buffer is added on edge $(2, 12)$ to enable the execution of node 12. After this, scheduling is performed again resulting in the schedule of figure B.9.

Now the optimal period is achieved on all PEs, but the tasks that are scheduled on the DMA are still exceeding it. The same process is repeated twice

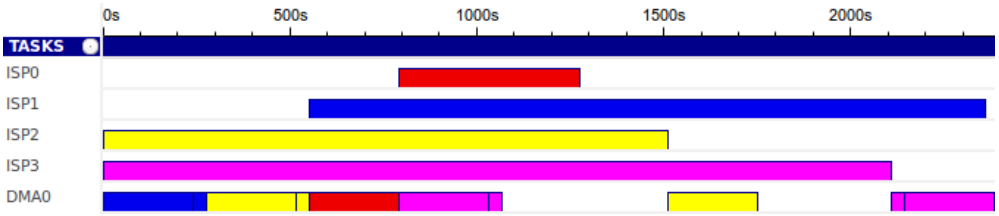


Figure B.8: The Gantt chart of loop body after the second iteration fo the buffer sizing phase.

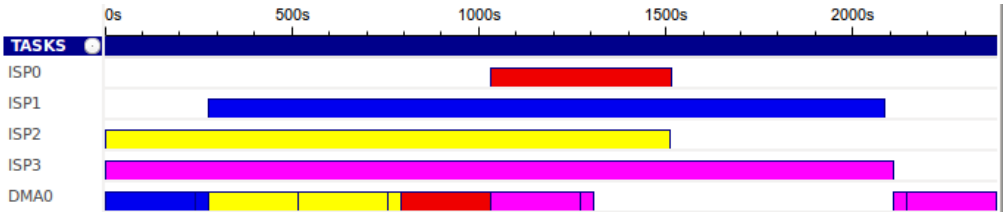


Figure B.9: The Gantt chart of loop body after the third iteration fo the buffer sizing phase.

more. First the buffer size of edge $(3, 9)$ is increased by one resulting in the schedule of figure B.10. After the fourth iteration, the buffer size of edge $(3, 8)$ is increased and now the optimal period can be achieved, as can be seen in figure B.11. The resulting SDF graph can be found in figure B.12.

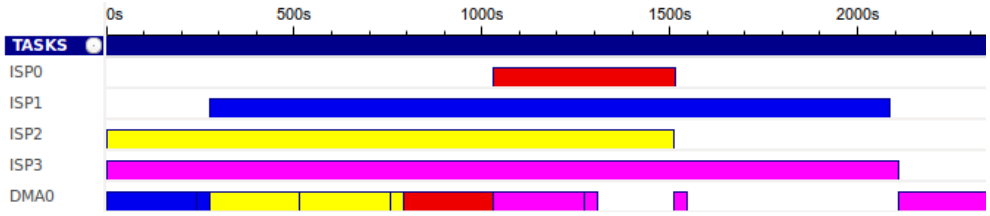


Figure B.10: The Gantt chart of loop body after the fourth iteration fo the buffer sizing phase.

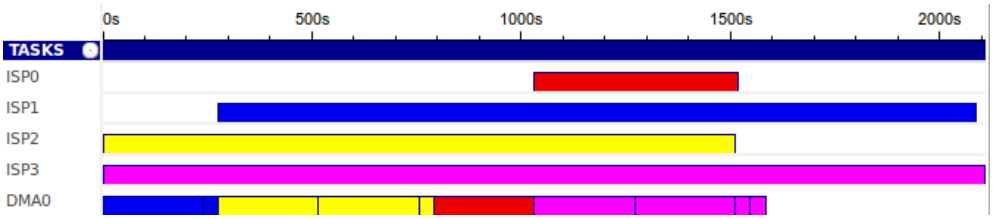


Figure B.11: The Gantt chart of loop body after the fifth iteration fo the buffer sizing phase.

B.3 Scheduler output

If this gang is part of the final partition then also a preamble and postamble are generated for it and this is written to a file called programScript.txt. To give an

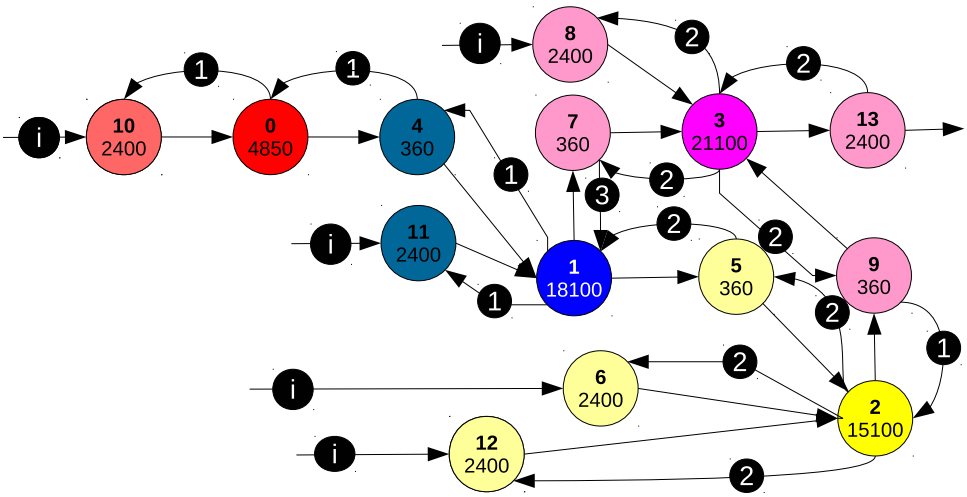


Figure B.12: The final model with the updated buffer sizes with which the optimal period can be achieved. The retiming is not applied.

idea of what this output looks like, fractions of the program script for this gang are shown below. This output is the input for the control program generator that transforms in the C code that can be compiled for the scalar processor.

Information about buffers, consisting of a memory address, width, height and number of buffers.

```
#buffers
BUFFER  ddr:input_6_0       ddr_address5    1920    1080    1
...
BUFFER  ddr:output_13_0     ddr_address20   1920    1080    1
BUFFER  cell0:output_0_0    cell_address0   1920    1       1
...
BUFFER  cell3:output_3_0    cell_address13  1920    1       2
```

Kernel loading instructions.

```
#kernels
LOAD_KERNEL warping cell0   cell_address10  cell_address0
...
LOAD_KERNEL chromaproc cell3    cell_address6   cell_address7   ce...
```

The preamble instructions

```
#preamble
DMA_TRANSFER    channel1:   ddr:input_11_0[0]    -> cell1:input_1_1[0]
WAIT_DMA        channel1
DMA_TRANSFER    channel1:   ddr:input_6_0[0]     -> cell2:input_2_1[0]
WAIT_DMA        channel1
DMA_TRANSFER    channel1:   ddr:input_6_0[1]     -> cell2:input_2_1[1]
WAIT_DMA        channel1
```

```
DMA_TRANSFER    channel2:   ddr:input_12_0[0]    -> cell2:input_2_2[0]
WAIT_DMA        channel2
DMA_TRANSFER    channel2:   ddr:input_12_0[1]    -> cell2:input_2_2[1]
WAIT_DMA        channel2
DMA_TRANSFER    channel0:   ddr:input_10_0[0]    -> cell0:input_0_0[0]
WAIT_DMA        channel0
START_KERNEL    cell0
DMA_TRANSFER    channel3:   ddr:input_8_0[0]     -> cell3:input_3_1[0]
WAIT_DMA        channel3
DMA_TRANSFER    channel3:   ddr:input_8_0[1]     -> cell3:input_3_1[1]
WAIT_DMA        channel3
WAIT_KERNEL     cell0
...
DMA_TRANSFER    channel4:   cell0:output_0_0[0]  -> cell1:input_1_0[0]
WAIT_DMA        channel4
START_KERNEL    cell1
DMA_TRANSFER    channel0:   cell1:output_1_0[2]  -> cell2:input_2_0[0]
START_KERNEL    cell0
WAIT_DMA        channel0
WAIT_KERNEL     cell0
WAIT_KERNEL     cell1
```

The loop body instructions.

```
#loop
LOOP i 5    ->  1080
START_KERNEL    cell3
DMA_TRANSFER    channel1:   ddr:input_11_0[i-1]  -> cell1:input_1_1[0]
START_KERNEL    cell2
WAIT_DMA        channel1
DMA_TRANSFER    channel4:   cell0:output_0_0[0]  -> cell1:input_1_0[0]
WAIT_DMA        channel4
START_KERNEL    cell1
DMA_TRANSFER    channel1:   ddr:input_6_0[i-2]   -> cell2:input_2_1[i-2]
WAIT_DMA        channel1
DMA_TRANSFER    channel2:   ddr:input_12_0[i-2]  -> cell2:input_2_2[i-2]
WAIT_DMA        channel2
DMA_TRANSFER    channel0:   cell1:output_1_0[i-2]    -> cell2:input_2_0[i-2]
WAIT_DMA        channel0
DMA_TRANSFER    channel0:   ddr:input_10_0[i]    -> cell0:input_0_0[0]
WAIT_DMA        channel0
START_KERNEL    cell0
DMA_TRANSFER    channel3:   ddr:input_8_0[i-3]   -> cell3:input_3_1[i-3]
WAIT_DMA        channel3
DMA_TRANSFER    channel3:   cell3:output_3_0[i-5]    -> ddr:output_13_0[i-5]
WAIT_KERNEL     cell2
WAIT_DMA        channel3
DMA_TRANSFER    channel4:   cell2:output_2_0[0]  -> cell3:input_3_2[i-3]
WAIT_KERNEL     cell0
WAIT_DMA        channel4
```

```
DMA_TRANSFER    channel2:   cell1:output_1_0[i-3]    -> cell3:input_3_0[i-3]
WAIT_DMA        channel2
WAIT_KERNEL     cell1
WAIT_KERNEL     cell3
ENDLOOP
```

The postamble instructions.

```
#postamble
START_KERNEL    cell3
DMA_TRANSFER    channel1:   ddr:input_11_0[1079]     -> cell1:input_1_1[0]
START_KERNEL    cell2
WAIT_DMA        channel1
DMA_TRANSFER    channel4:   cell0:output_0_0[0]  -> cell1:input_1_0[0]
WAIT_DMA        channel4
START_KERNEL    cell1
DMA_TRANSFER    channel1:   ddr:input_6_0[1078]  -> cell2:input_2_1[0]
WAIT_DMA        channel1
DMA_TRANSFER    channel2:   ddr:input_12_0[1078]     -> cell2:input_2_2[0]
...
WAIT_KERNEL     cell2
DMA_TRANSFER    channel4:   cell2:output_2_0[0]  -> cell3:input_3_2[1]
WAIT_DMA        channel4
WAIT_KERNEL     cell3
START_KERNEL    cell3
DMA_TRANSFER    channel3:   cell3:output_3_0[0]  -> ddr:output_13_0[1078]
WAIT_DMA        channel3
WAIT_KERNEL     cell3
DMA_TRANSFER    channel3:   cell3:output_3_0[1]  -> ddr:output_13_0[1079]
WAIT_DMA        channel3
```

Bibliography

[1] E. Rainey, J. Villarreal, G. Dedeoglu, K. Pulli, T. Lepley, and F. Brill. Addressing system-level optimization with openvx graphs. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2014 IEEE Conference on*, pages 658–663, June 2014.

[2] E. Bolotin, D. Nellans, O. Villa, M. O'Connor, A. Ramirez, and S. W. Keckler. Designing efficient heterogeneous memory architectures. *IEEE Micro*, 35(4):60–68, July 2015.

[3] Jonathan Ragan-Kelley. *Decoupling Algorithms from the Organization of Computation for High Performance Image Processing*. Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2014.

[4] G. Tagliavini, G. Haugou, and L. Benini. Optimizing memory bandwidth in openvx graph execution on embedded many-core accelerators. In *Design and Architectures for Signal and Image Processing (DASIP), 2014 Conference on*, pages 1–8, Oct 2014.

[5] Khronos Group. Khronos group openvx overview. `https://www.khronos.org/assets/uploads/developers/library/overview/openvx-overview.pdf`, 6 2015. Accessed: 2015-10-20.

[6] I. Culjak, D. Abram, T. Pribanic, H. Dzapo, and M. Cifrek. A brief introduction to opencv. In *MIPRO, 2012 Proceedings of the 35th International Convention*, pages 1725–1730, May 2012.

[7] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., New York, NY, USA, 1st edition, 2000.

[8] B. Ramakrishna Rau. Iterative module scheduling: an algorithm for software pipelining loops. In *Microarchitecture, 1994. MICRO-27. Proceedings of the 27th Annual International Symposium on*, pages 63–74, Nov 1994.

[9] Yuan Lin, Manjunath Kudlur, Scott Mahlke, and Trevor Mudge. Hierarchical coarse-grained stream compilation for software defined radio. In *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '07, pages 115–124, New York, NY, USA, 2007. ACM.

[10] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Commun. ACM*, 17(12):685–690, December 1974.

[11] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18(2):244–257, April 1989.

[12] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, Feb 1993.

[13] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, Jan 1987.

[14] C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[15] Qi Ning and Guang R. Gao. A novel framework of register allocation for software pipelining. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 29–42, New York, NY, USA, 1993. ACM.

[16] O. Moreira, T. Basten, M. Geilen, and S. Stuijk. Buffer sizing for rate-optimal single-rate data-flow scheduling revisited. *IEEE Transactions on Computers*, 59(2):188–201, Feb 2010.

[17] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[18] R. Govindarajan and G. R. Gao. A novel framework for multi-rate scheduling in dsp applications. In *Application-Specific Array Processors, 1993. Proceedings., International Conference on*, pages 77–88, Oct 1993.

[19] Harry Printz, H. T. Kung, Todd Mummert, and Paul Scherer. Automatic mapping of large signal processing systems to a parallel machine, 1989.

[20] T. Gross, S. Hinrichs, G. Lueh, D. O'Hallaron, J. Stichnoch, and J. Subhlok. Compiling task and data parallel programs for iwarp. *SIGPLAN Not.*, 28(1):32–35, January 1993.

[21] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini. Adrenaline: An openvx environment to optimize embedded vision applications on many-core accelerators. In *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on*, pages 289–296, Sept 2015.

[22] Jan Arne Telle and Andrzej Proskurowski. Algorithms for vertex partitioning problems on partial k-trees. *SIAM J. Discret. Math.*, 10(4):529–550, August 1997.

[23] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Design Automation, 1982. 19th Conference on*, pages 175–181, June 1982.

[24] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, pages 3255–3258 vol.5, May 1995.

[25] W. Ahmad, R. d. Groote, P. K. F. Hlzenspies, M. Stoelinga, and J. v. d. Pol. Resource-constrained optimal scheduling of synchronous dataflow graphs via timed automata. In *Application of Concurrency to System Design (ACSD), 2014 14th International Conference on*, pages 72–81, June 2014.

[26] Thierry Lepley. Tegra x1 and visionworks/openvx: Computer vision on a chip. `http://www.gipsa-lab.grenoble-inp.fr/thematic-school/gpu2015/presentations/GIPSA-Lab-GPU2015-T-Lepley.pdf`, December 2015. Accessed: 2016-16-4.

[27] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[28] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] L. De Moura and B. Dutertre. Yices 1.0: An efficient smt solver. *The Satisfiability Modulo Theories Competition (SMT-COMP)*, 2006.

[30] B. Dutertre. Yices 2.2. *CAV, Vienna*, 2014.