ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

StencilFlow: Stencil Dataflow on Reconfigurable Hardware

Bachelor Thesis

A. Kuster

August 20, 2019

Advisors: Prof. Dr. T. Hoefler, J. de Fine Licht
Department of Computer Science, ETH Zürich

Abstract

Accurate and reliable weather forecast is of vital importance for a broad field of industries and the general public. Highly regular and statically analyzable stencil operators on structured grids are used to numerically solve the partial differential equations of such weather prediction models. This allows optimizations for data re-use while minimizing the high demand of memory bandwidth. Since technology is hitting the power-wall of approximately $1\frac{Watt}{mm^2}$ for air-cooled CMOS fabrics, while spending a great fraction of the energy for data movement and caching, future high-performance architectures must increase the fraction of energy spent on computations to continue scaling. By implementing custom dataflow architectures on FPGAs, there is a potential to greatly reduce control and data movement overhead. We hope to move a step toward this goal of being more energy efficient compared to the Von-Neuman load/store architecture.

In this thesis, we introduce StencilFlow, a framework for mapping stencil programs to FPGAs, offering a complete toolchain from input data analysis, optimization, and simulation, to generation of optimal code for re-programmable devices. We formalize the input program as a directed acyclic graph of streaming modules, and optimize for maximum utilization of on-chip memory and off-chip memory bandwidth.

To gain insight into performance metrics and debugging, we develop a model to simulate the stencil program in software. The optimized stencil program is mapped onto FPGA hardware using the DaCe framework. As a realistic use case, we study the COSMO numerical weather forecasting model, currently running on a multi-core and hybrid CPU-GPU architecture. With our framework, high-level users can rapidly implement, optimize, debug, and synthesize arbitrary, large stencil programs onto FPGAs.

Contents

Chapter 1

Introduction

Accurate and reliable weather forecast is of vital importance for a broad field of industries, as well as the general public. Highly regular and statically analyzable stencil operators [44] on structured grids are used to numerically solve the partial differential equations of such weather prediction models. This allows optimizations for data re-use while minimizing the high demand of memory bandwidth. Our collaboration with MeteoSwiss [6] enables us to apply our theoretical optimization findings to the numerical weather prediction and regional climate model COSMO [24, 2]. By cooperating closely with Swiss National Supercomputing Center (CSCS) [17] and the University of Paderborn we gain access to computational hardware resources and intend to figure out together if FPGAs (field-programmable gate arrays) can serve as the next generation accelerator for high-performance weather prediction simulations.

1.1 Objectives and Goal

Nowadays, numerical weather prediction simulations are executed on Von-Neumann architecture [23] based CPU or GPU clusters [39]. Since technology is hitting the power-wall of approximately $1\frac{Watt}{mm^2}$ for air-cooled CMOS fabrics [27], high-performance designs should spend a significant amount of that energy for actual computations. Research in the computational field has shown that FPGAs could be a competitive technology in comparison to CPUs [35], GPUs [37] and even ASICs [28]. By streaming data through the FPGA and therefore greatly reducing control and data movement overhead [27, 30], we estimate to be about 8 times (see Appendix A) more energy efficient compared to the load/store architecture. Thus, we seek to implement the numerical model on a state-of-the-art FPGA, the Intel Stratix 10 [7] using OpenCL [15] as high-level synthesis (HLS) tool [32, 29] for increased productivity.

1.2 Methods of Investigation / Implementation

Our theoretical optimization focus lies in the analysis and optimal allocation of resources to reduce the memory bandwidth bottleneck [41, 42, 26]. By formalizing the input program as a directed graph of streaming modules and formulating it as an optimization problem, we seek to solve it for optimal bandwidth and fast memory usage. To gain insight into performance metrics and for debugging, we develop a model to simulate the stencil program in software. The final step will be to transform this theoretically optimal design onto the FPGA. Studies [38, 25, 32, 31, 21] have shown that optimization of HLS code is crucial for maximal performance. Therefore, the project is carried out in close collaboration with experts in stencil optimization for different heterogeneous systems (CPU and GPU) [41, 42, 33, 43, 40, 34, 36] at ETH.

1.3 Collaborations

The outcome of this project highly depends on the symbiosis of partners willing to share their experience, expertise and resources with us. On one hand, we tackle a real-world problem that can help us finding out if our approach can be efficiently scaled up to production scale problems and program sizes. On the other hand, the hardware resources for the synthesis and testing of FPGA designs are very demanding, which is why we have a partner who can provide us with this service. Furthermore, having expert knowledge from other groups involved in high performance FPGA designs helps us a lot to share valuable problem solutions and optimization hints.

1.3.1 MeteoSwiss

MeteoSwiss [10], the Federal Office of Meteorology and Climatology located in Zurich, Geneva, Locarno and Payerne is part of the federal administration of Switzerland. Their main task is to constantly create weather forecasts to inform authorities and the general public about upcoming storms and other strong weather phenomena. In addition to that, they provide weather services for civil and the military.
MeteoSwiss, together with CSCS are very progressive partners making use of modern, cutting-edge technology. This effort made them the first major national weather service to deploy a GPU-accelerated supercomputer to improve its daily weather forecasts [12].
We both share the common interest of finding out if FPGAs could be the next technological step forward, which is why we are closely collaborating. This collaboration gives us access to the source code of their current implementation and allows us to directly communicate with experts from MeteoSwiss that are actively involved in the current developments.

1.3.2 CSCS

CSCS [4], the Swiss National Supercomputing Center, is the national high-performance computing center of Switzerland. It operates the Piz Daint, one of the world's largest supercomputers (rank 6 in TOP500 at the time of writing, [18]) in addition to some dedicated computing resources for specific projects and offices of the federal government of Switzerland.
Their interest and the associated early purchase of the Stratix 10 FPGA allowed us to do early hands-on test with the hardware and getting valuable insights. In addition to that, high-level synthesis from our high-level OpenCL design to the actual bitstream of the FPGA is a time and resource consuming task. CSCS provides us with Greina and Ault, two large memory cluster computers with integrated FPGA hardware, the required resources for efficient testing and evaluation.

1.3.3 University of Paderborn

The University of Paderborn, Germany, is one of the leading European university in the use of reconfigurable device for high performance applications. With the inauguration of Noctua [13], their new cluster computer incorporates 16 nodes with two Intel Stratix 10 FPGA (Bittware 520N) cards each. These cards are interconnected through a optical circuit switch, which gives us the flexibility of potentially splitting the problem and make tests of running it on multiple devices while streaming the necessary data via the dedicated 40Gb/s fiber interconnect.
Furthermore, we get expert knowledge from the scientist working in this environment and can share experiences, problems and performance improving design hints. This is especially valuable, since FPGAs are not as main-stream in HPC as CPU and GPUs are. The lack of forum and human resources makes the problem solving more complex. This makes a close collaboration with other people from the same field especially valuable.

1.4 Acknowledgement

I would like to thank my mentor Johannes de Fine Licht for his extraordinary support in this thesis process. Advice given by Professor Torsten Hoefler during the meetings has been of great help in staying on the right track throughout the project. Assistance provided by Carlos Osuna from MeteoSwiss for COSMO related questions was greatly appreciated. I wish to acknowledge the help provided by Hussein Nasser from CSCS by setting up the FPGA cards on Greina and Ault and helping us in case of failure. I am particularly grateful for the assistance given by Tobias Kenter from University of Paderborn in high-level synthesis related questions. Last but not

least, Tobias Gysi from ETH Zurich provided me with very valuable inputs from his prior work on the COSMO weather model.

Chapter 2

Weather Simulation at MeteoSwiss

2.1 Introduction

COSMO, the Consortium for Small-scale Modeling [3], is an association seeking to develop, improve and maintain a non-hydrostatic limited-area atmospheric model, the COSMO-model. This is the base for many national weather prediction services in Europe.
The consortium was established in 1998 by DWD (Germany) and MeteoSwiss (Switzerland). Over the years, weather services and organizations such as ZGeoBw (Germany), ITAF ReMet/CIRA/ARPAE, ARPA Piemont (Italy), HNMS (Greece), IMGW (Poland), IMS (Israel), NMA (Romania) and RHM (Russia) have joined. In addition to meteorological services, there are also academic communities involved, namely the Climate Limited-Area Modeling (CLM) that extended the model for long-term simulations and Universities maintaining their own extensions. The Karlsruhe Insitute for Technology (KIT) extended the model to account for the interactions of gases and aerosols within the state of the atmosphere.
The COSMO-model was initially based on DWDs "Lokal-Modell" (Local Model) and further refined and improved in joint collaboration by the COSMO members.

Structure A time integration cycle illustrated in figure 2.1 can be broken down into different tasks and execution phases [19]. The first steps prepares the input data for processing. The physics step accounts for the physical processes that are not resolved by the 3-dimensional numerical grid (vertical diffusion (turbulences), cloud and precipitation formation (condensation), convection, radiation and soil processes).
The dynamical core, consisting of the dynamics and relaxations phase, is the main computational challenge of the weather model. It is formulated in terms of stencil programs applied on the highly regular, three dimensional

data grid. Therefore, we focus on this part of the model.
The nudging step takes care of the convergence of the computed model values and the actual observations from the observation stations. Finally there is post-processing going on and a cleanup to be ready for the next iteration step.

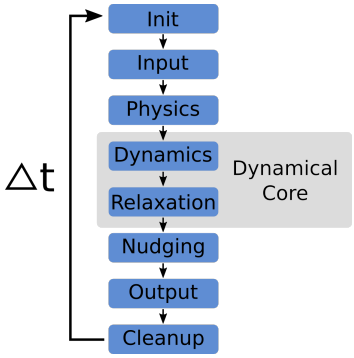


Figure 2.1: The COSMO Model: Structure of a the time integration interval.

Model Instances (MeteoSwiss Specific) MeteoSwiss does not rely on a single instance of the weather model, but rather follows the approach of subdivisions of the forecast into different set of problems [11]. There are various grid resolution instances and simulation durations shown in figure 2.2, which gives them the ability to better predict longer-term, mid-term and short-term weather constellations in the very challenging topological Alpine region. These grid resolutions define the computational domain of the problem since all stencil operators are applied to each individual grid point.

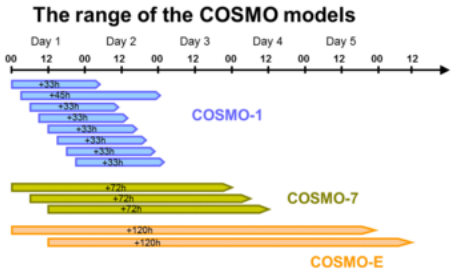


Figure 2.2: Range of COSMO models with their forecast horizon and number of runs per day. *meteoswiss.admin.ch*

COSMO-1 The COSMO-1 instance is a high-resolution version with a grid-box size of 1.1km (0.01°) spread over the Alpine region. It is able to predict precipitation, temperature, wind as seen in figure 2.3 and numerous other meteorological parameters. The number of grid points is 1158x774 horizontally and 80 in the vertical dimension. The model topography reaches 4268 meters above sea level. Every grid point is updated in a time integration interval of 10 seconds.

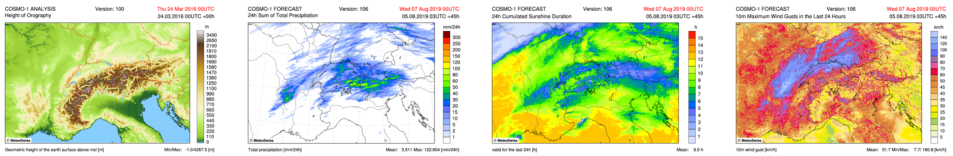


Figure 2.3: COSMO-1 example model parameters: Height of Orography, Sum of total Precipitation (Prediction), 24h Sum of Sunshine Hours (Prediction), 10m Above Ground Maximum Wind Guts (Past) (from left to right), *meteoswiss.admin.ch*

COSMO-E The COSMO-E models purpose is to improve the reliability of the short to medium range forecast for highly localized weather events. It is a collection of 21 different forecasts that are being executed twice a day.
It covers the whole Alpine region, but with a lower resolution in comparison to the COSMO-1 instance. It uses grid box sizes of 2.2km and 582x390 horizontal grid points and 60 vertical layers.

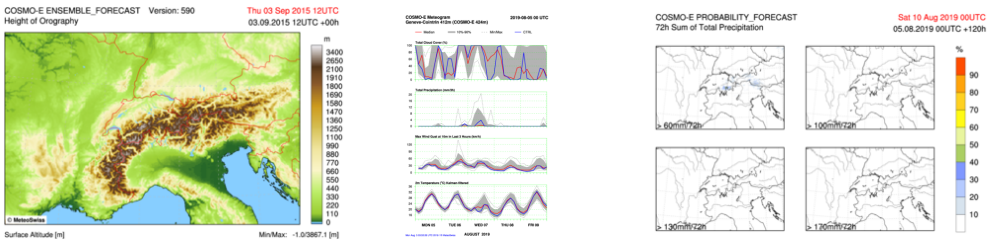


Figure 2.4: COSMO-E example model parameters: Height of Orography, Meteogram for Geneva-Cointrin, Probability Forecast (from left to right), *meteoswiss.admin.ch*

COSMO-7 The COSMO-7 instance with a grid box size of 6.6km (0.06°) is a low-resolution version with the aim of covering central and western Europe. The predictions are being run four times a day over a grid of size 393x338 horizontally and 60 vertical units.

Specialized Seasonal Instances There exists also seasonal model instances for specific problems, such as the COSMO-ART, which computes the pollen

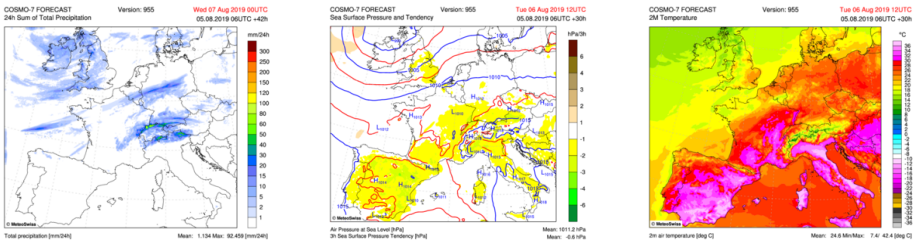


Figure 2.5: COSMO-7 example model parameters: Height of Orography, Air Pressure at Sea Level, 2m Air Temperature (from left to right), *meteoswiss.admin.ch*

concentration of alder, birch, grasses and ragweed pollen, and is in service from February till end of September).

2.1.1 Compute Hardware of MeteoSwiss

History Since simulation-dependent scientific areas can improve their prediction accuracies by increasing model complexity and resolution, they were able to benefit a lot from the exponential increase in compute power over the last couple of decades. This can be well observed by the increase in node and core counts of MeteoSwiss' cluster compute infrastructure [5].

- 1999–2007: "Prometeo", NEC SX-5, vector processor, 16 nodes
- 2007-2012:
  - "La Dôle", Cray XT4, AMD Opteron quad-core Barcelona 2.3 GHz, 160 nodes (640 cores)
  - Piz Buin Cray XT4 AMD Opteron quad-core Barcelona 2.3 GHz, 264 nodes (1,056 cores)
- from 2012
  - "Monte Lema", Cray XE6, AMD Opteron 12-core Magny-Cours 2.1 GHz, 336 nodes (4,032 cores)
  - "Albis", Cray XE6, AMD Opteron 12-core Magny-Cours 2.1 GHz, 144 nodes (1,728 cores)
- from 2015
  - "Kesch" and "Es-cha", Cray CS-Storm, Intel Haswell-EP + Nvidia Tesla K80 GPUs (heterogenous system)

Today MeteoSwiss together with CSCS are very ambitious and progressive institutions by keeping up with the pace of technology development. This can be observed very well by the hardware details of Kesch and Es-cha, which was the first cluster computer of a national weather forecast service

using a heterogeneous combination of CPUs and GPU compute cards for performance increase [14].

- "Kesch and Es-cha": 12 hybrid compute nodes consisting of [9]:
  - 2 Intel Haswell E5-2690v3 2.6 GHz 12-core CPUs per node (total of 24 E5-2690v3 processors)
  - 256 GB 2133 MHz DDR4 memory per node (total of 3 TB)
  - 8 NVIDIA Tesla K80 GPU devices per node (total of 192 GPUs)

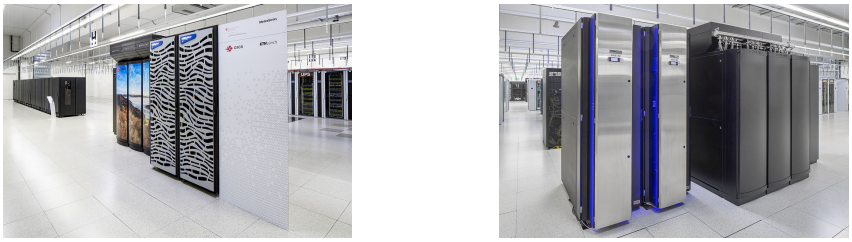


Figure 2.6: Kesch and Es-cha, the two compute clusters dedicated for the MeteoSwiss weather prediction. *cscs.ch*

2.1.2 Current Limitation

Even thought the transistor density and the number of computations per time unit is still increasing, the memory subsystem can not keep up. Stencil computations on structured grids, which COSMOs dynamical core consists of are heavily memory bound. Figure 2.7 illustrates the abstraction levels from the dynamical core as a stencil program down to the actual compute stencils with their corresponding data field accesses. This means that the ratio of compute operations compared to the number of data field accesses is very low. This fact limits the effectiveness of the upgrade of compute resources, since many compute elements are idling and waiting for data to be transfered to them.
STELLA [41] and MODESTO [42] try to solve or mitigate this problem by optimizing the stencil programs to optimally use the cache hierarchy and therefore reducing the stress on the actual memory bandwidth.

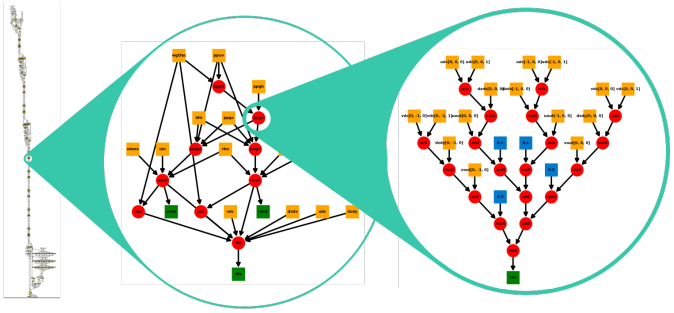


Figure 2.7: Overview of the total complexity and number of memory accesses given the different hierarchy layers of abstraction.

Benefit of FPGA This is exactly where the strength of FPGAs lies in. The freedom of not having fixed, hard-wired cache hierarchies, but rather access to spread the fast memory resources in a fine-grained manner such that the overall design profits the most. This increases the actual number of computations per unit time. The next chapter is dedicated to our solution approach and how we intend to implement this in practice.

Chapter 3

StencilFlow Architecture Model

One the one hand, field programmable gate arrays give us much higher flexibility compared to traditional computed devices such as central progressing units or graphics processing units. On the other hand, a good strategy and knowledge about the underlying hardware is required in order to make use of this additional degrees of freedom.
This section is dedicated to outline theoretical aspects of the overall strategy we are proposing. This includes the transformation from the abstracted high-level theoretical input to the mapping onto the actual hardware resources.

3.1 Application Specific Dataflow for Weather Codes

Kernel computations on structured grids usually impose high stress on data movement since the amount of actual computations is rather low. Therefore we seek to transform the problem to a single, deep pipeline. This allows us to directly forward the intermediate result of a compute unit to the next unit, instead of transferring it back to slow memory (or some intermediate cache) and later fetch it again. This optimization technique is widely used in hardware designs and not only increases the efficient usage of the resources, but can also save energy by not involving the lower memory hierarchy.
As we will see later on, larger designs with more complex dependencies might impose constraints such that we are not always able to directly forward the result to the next processing unit. Therefore we have to statically analyze where and in which size we have to allocate buffers for temporary storing the intermediate results. Since fast memory is limited, we have to find a good strategy to decide whether or not to use fast memory for a buffer allocation or sacrifice memory bandwidth and swap the buffer out to slow memory.

3.1.1 Example Walk-through

To better understand what this means in practice, we will go through the transformation of a simple two kernel example stencil program and its transformation to a fully pipelined design.
The stencil program illustrated in figure 3.1 consists of three input data arrays (inA, inB, inC), a first kernel kernelA reading from inA and inB a second kernel kernelB reading from inC and kernelA. The final result is the output of kernelB.

```
inputs: inA, inB, inC
outputs: kernelB
kernels:
kernelA[i,j,k] = (inA[i,j,k]+inA[i,j,k+1])*inB[i,j,k]
kernelB[i,j,k] = (1.0-inC[i,j,k])*kernelA[i,j,k]
```

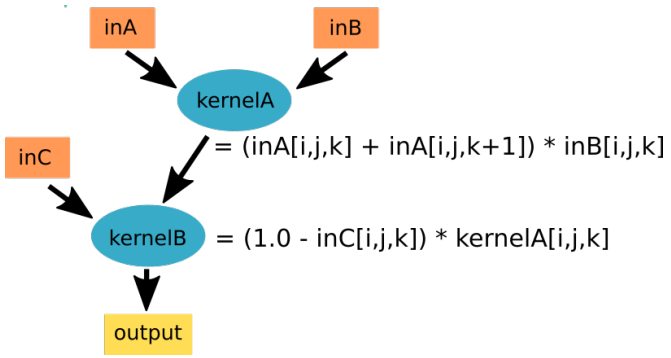


Figure 3.1: Example stencil program in high-level data flow representation.

To avoid storing results of kernelA back to memory, we can directly forward the result to kernelB as illustrated in listing.

```
inputs: inA, inB, inC
outputs: kernelB
kernels:
kernelB[i,j,k] = (1.0 - inC[i,j,k])*(inA[i,j,k] +
                 inA[i,j,k+1])*inB[i,j,k]
```

This expression can then be laid down in a pipeline manner by following the mathematical precedence rule of the operation shown in figure 3.2. Since we are reading two fields from inA with an offset of one, we have to add a stage where we buffer the element at [i,j,k] first in order to be able to read both data elements simultaneously.

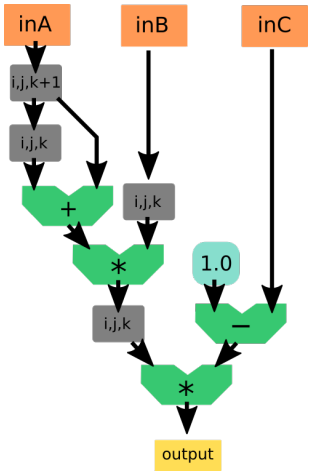


Figure 3.2: Example stencil program visualized as a pipeline.

When we look from a higher level at the pipeline again, illustrated in figure 3.3, we can see the major components:

- Inputs: the input data arrays: inA, inB, inC
- Kernels: the two compute kernels: kernelA, kernelB
- Output: the output data store with the compute result: output
- a channel that transfers data from the first to the second kernel

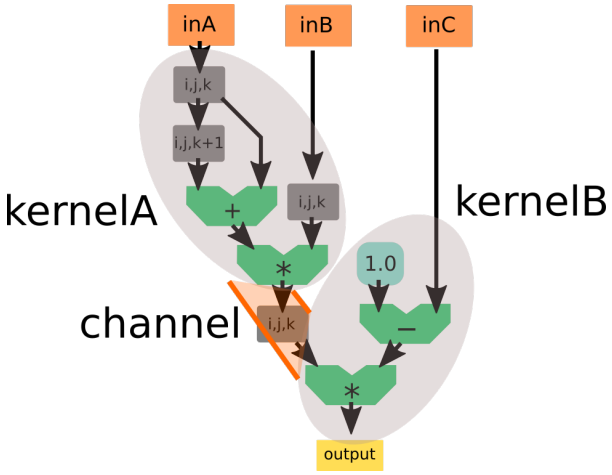


Figure 3.3: Example stencil program with additional annotation about the structure of kernels and channels connecting the kernels.

As we will see later on, this structure builds the foundation for further analysis.

3.1.2 Reason

There are several reasons why we think that deep pipelining is the right choice. First, todays FPGAs are very well suited and optimized to generate highly efficient and higher-frequency designs compared to earlier reprogrammable hardware logic designs [8, 22]. In addition, the incorporated hard compute units have a raw peak performances of 10 TFLOP/s of 32-bit single precision floating point compute performance.
Furthermore, pipelining can save us a lot of data movements since we can directly forward the result of the previous computation to the next compute unit. This can significantly increase performance in case of memory bottlenecks and the idling time associated with it.

We conclude that the actual problem is about optimal allocation of the different resources for maximizing the overall throughput. This brings us to the question of, given an stencil program, which and how much resources are required to implement it efficiently in hardware. The next section is dedicated to this input problem analysis.

3.2 Analysis

This section is dedicated to find and formalize the key metrics for transforming an input problem to a fully pipelined design. We reason about how different patterns of the input program require certain resources such as buffer space and generalize this in order to make these analysis steps fully automatic by StencilFlow.
As we have seen in the previous section, it naturally makes sense to split the stencil program into inputs, kernels and outputs. We will therefore first look at the individual kernels and later on use the information and insights gained from this to reason about the whole stencil program.

3.2.1 Kernel

The base of a kernel consists of a kernel string or mathematical expression, the actual stencil. We are dealing with shift-invariant stencils that can read from a neighborhood of the center element [i,j,k] to produce the result for the data element at [i,j,k]. This stencil pattern is then applied for the elements in the grid.
The key metrics we are interested in is the latency of a single computation given the latency of each individual operation and the amount of buffer space required such that we do not have to re-fetch the same data element for the stencil twice. We will refer to this as the internal buffer and derive the formula for its size in this section.

Stencil Shift-invariant stencils are operators that perform element-wise computation of a function F over a fixed neighborhood S. Therefore they can be element-wise applied to each location of the data grid as illustrated in figures 3.4 and 3.5.

```
for i=1..N
  for j=1..M
    out(i,j) = F(S(i,j))
```

$$S(i,j) = \text{in}(u,v), \text{(u,v)} \in \{(i,j),(i,j-1),(i,j+1),(i-1,j),(i+1,j)\} \quad (3.1)$$

$$\text{F: } \mathbb{R}^{|S|} \rightarrow \mathbb{R} \quad (3.2)$$

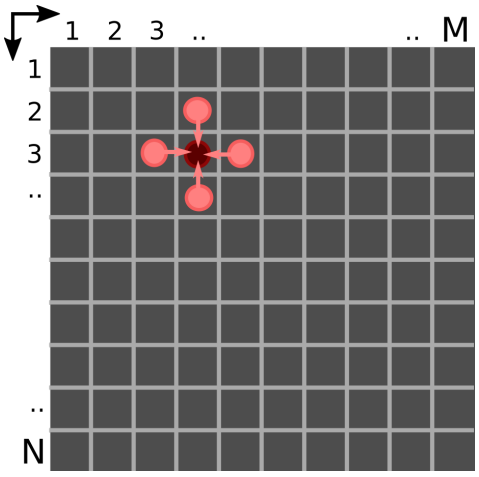


Figure 3.4: A two dimensional stencil laid out on a two dimensional grid.

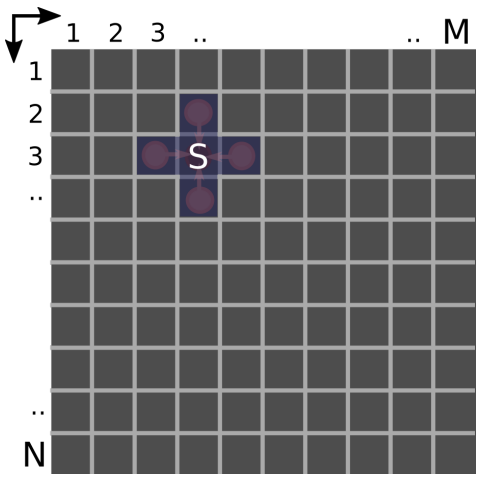


Figure 3.5: A two dimensional stencil laid out on a two dimensional grid with emphasized neighborhood S.

Boundary Conditions At the boundary of the grid shown in figure 3.6, some accesses of the neighborhood might be out-of-bound. For this special case, there are different strategies that can be applied to pad the grid shown in figure 3.7.
The most commonly used ones are constant, copy and interpolation boundary conditions. The constant boundary condition assigns a pre-defined constant value to each out-of-bound data access. The copy boundary condition copies the value of the last valid access field and the interpolation boundary condition interpolates the value from some neighborhood around the boundary (e.g. polynomial of degree one or two).

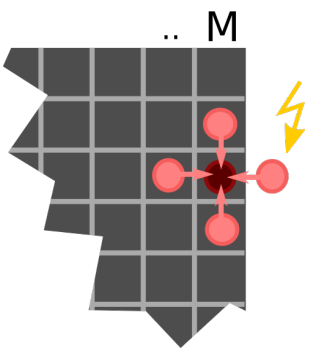


Figure 3.6: Example case of a field access outside of the data array.

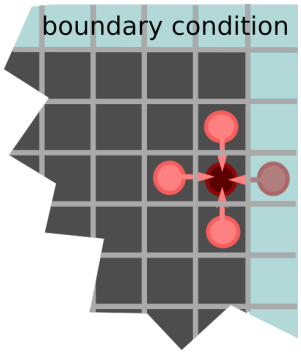


Figure 3.7: Example case of a field access outside of the data array with emphasized boundary condition.

Compute Graph We represent these computations as an abstract syntax tree graph consisting of input nodes, operation nodes and output nodes. The input nodes feed data into the compute graph and are either constant numerals, variables or array field accesses. The operation nodes represent the actual compute units. They can be binary operations such as addition and multiplication, but also unary operations (e.g. negation), function calls (e.g. sin/cos) or the ternary operator. At the end (or root of the tree) of the compute graph is always an output node that carries the actual computation result as shown in figure 3.8.

```
res = (1.0 > c*d) ? (a-b):(c*d)
```

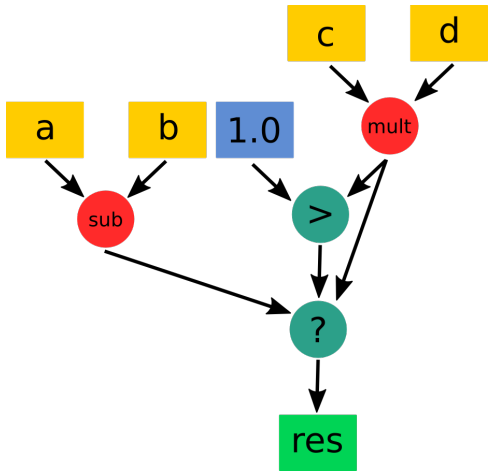


Figure 3.8: Example Compute Graph.

Latency In order to compute the global stencil program latency, we are interested in the critical path latency of each individual stencil. Using the

graph representation of the AST and a static mapping of each operation to the number of cycles it takes to perform a single operation, we can simply walk up the tree from the result and sum up the latency of each computation node till an input node is being reached. The highest latency value at some input corresponds to the overall critical path latency of the stencil.

Internal Buffer We refer to the internal buffer as the buffer space required such that each input data element has to be fetched or read only once. In other words, we want to store all data elements once they have been used for the first time by this specific stencil till they retire (not being required any longer) as shown in figure 3.10.
To get a good understanding, we derive the formula of this calculation by looking at some examples. Without loss of generality, we assume that we have a single three dimensional input data grid that we are looping over (figure 3.9) using our stencil where the center element of the stencil is the field we are producing the result for. The derived formula can be extended to multiple different input arrays by splitting the field accesses for each input array and solve the problem for each of them. Furthermore, we iterate over the array in the following dimensional order:

```
for i=1..Z
  for j=1..Y
    for k=1..X
```

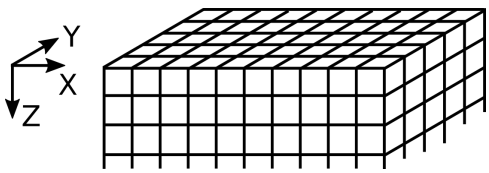


Figure 3.9: Data array.

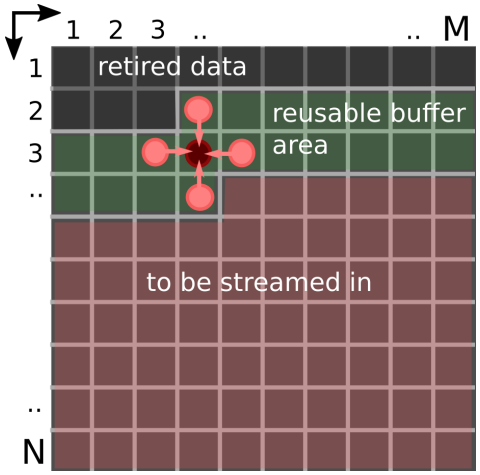


Figure 3.10: Internal Buffer of a Stencil.

Example 1: Single Element Our first example is a kernel that simply multiplies the only input element by a factor of 7.0. Furthermore, the only field

access is at the same location on the grid as we write the output to (at [i,j,k]).

$$\text{out}(i,j,k) = 7.0 \cdot \text{in}(i,j,k) \tag{3.3}$$

This leads to a stencil consisting only of a single element (namely the center element itself) as illustrated in figure 3.11.
Figure 3.12 shows the internal buffer of such a stencil operator, which is one data element wide. As soon as the data element has been read, it will not be used anymore and it can therefore be discarded.

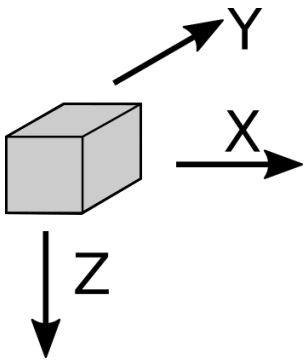


Figure 3.11: Example 1: Stencil Operator.

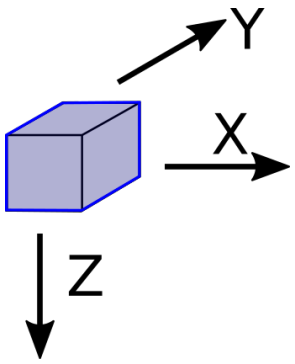


Figure 3.12: Example 1: Internal Buffer laid over the Stencil.

This number can also be derived by computing the difference between the highest and the lowest access index and adding one.

$$\text{max index} - \text{min index} + 1 = [i,j,k] - [i,j,k] + 1 = [0,0,0] + 1 \tag{3.4}$$

We will check with more complex examples if the formula also holds for them.

Example 2: 1D Stencil Our second example consists of a stencil that sums over the center element and its predecessor and successor element in X direction.

$$\text{out}(i,j,k) = \text{in}(i,j,k-1) + \text{in}(i,j,k) + \text{in}(i,j,k+1) \tag{3.5}$$

Therefore, the stencil consists of three cubes (figure 3.13) laid out next to each other in the direction of the X axis.
The internal buffer size of such a stencil operator is three data element wide. This can be verified by hand since we are in the innermost loop and the stencil therefore moves forward each iteration by one element in X direction.

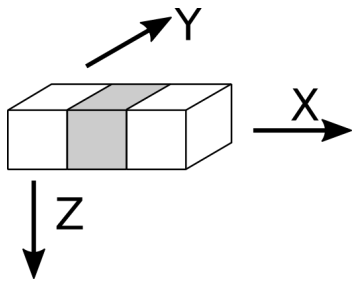


Figure 3.13: Example 2: Stencil Operator.

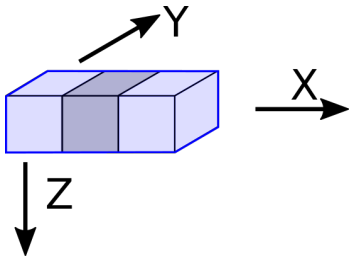


Figure 3.14: Example 2: Internal Buffer laid over the Stencil.

Therefore, the element read by the field access [i,j,k+1] can be discarded after the last stencil access ([i,j,k-1]) made use of it.

This number can also be derived by computing the difference between the highest and the lowest access index and adding one.

$$\text{max index} - \text{min index} + 1 = [i,j,k+1] - [i,j,k-1] + 1 = [0,0,2] + 1 \tag{3.6}$$

Example 3: 2D Stencil Our third example consists of a stencil that sums up its direct neighbors in the X-Y plane.

$$\text{out}(i,j,k) = \text{in}(i,j,k) + \text{in}(i,j,k-1) + \text{in}(i,j,k+1) + \text{in}(i,j-1,k) + \text{in}(i,j+1,k) \tag{3.7}$$

Therefore, the stencil looks like a cross in the X-Y plane (figure 3.15).
The internal buffer size of such a stencil is not a static number of elements anymore, but depends on the actual dimension sizes since the buffer is spread over a whole dimension. Using the graphic, you can verify that the buffer size of this specific stencil is 2*X + 1.

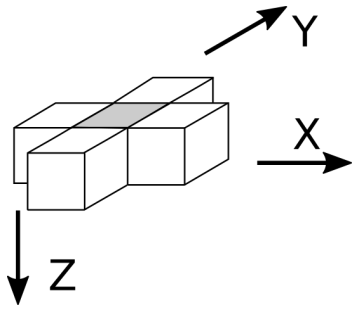


Figure 3.15: Example 3: Stencil Operator.

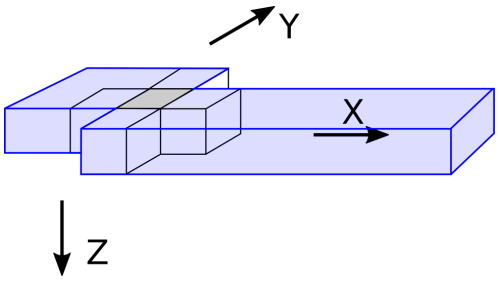


Figure 3.16: Example 3: Internal Buffer laid over the Stencil.

This number can also be derived by computing the difference between the

highest and the lowest access index and adding one.

$$\text{max index} - \text{min index} + 1 = [i, j+1, k] - [i, j-1, k] + 1 = [0, 2, 0] + 1 \tag{3.8}$$

Example 4: 3D Stencil Our fourth example consists of a stencil that adds the center element with its neighbor in the Z dimension, which represents the outermost loop.

$$\text{out}(i, j, k) = \text{in}(i, j, k) + \text{in}(i+1, j, k) \tag{3.9}$$

Therefore, the corresponding stencil looks like two cubes stacked above each other.
This time the buffer is even spread over two dimensions (i.e. layer of the X-Y plane. And has the size of X*Y + 1.

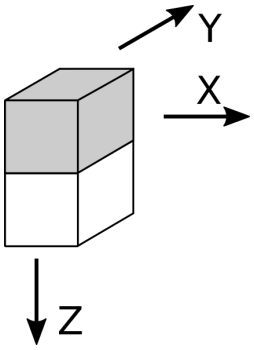


Figure 3.17: Example 4: Stencil Operator.

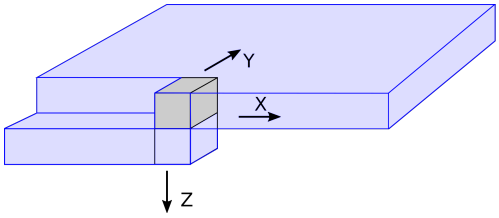


Figure 3.18: Example 4: Internal Buffer laid over the Stencil.

This number can also be derived by computing the difference between the highest and the lowest access index and adding one.

$$\text{max index} - \text{min index} + 1 = [i+1, j, k] - [i, j, k] + 1 = [1, 0, 0] + 1 \tag{3.10}$$

We can conclude that the internal buffer size can be computed in general using the following formula:

$$\text{max index} - \text{min index} + 1 \tag{3.11}$$

Dimensional Notation We refer to this notation of square brackets as the dimensional notation. This format is internally used in StencilFlow since it has the great benefit of being independent of the actual problem dimensions. The relation from the dimensional form to an absolute value is given by:

```
given:
input dimensions [X,Y,Z]
value in dimensional format: [a,b,c]
absolute size: x + X*b + X*Y*a = c + X*(b  + Y*a)
```

3.2.2 Stencil Program

The stencil program graph represents the highest level of abstraction of the stencil program. It is required to represent dependencies between different kernels and computes global properties combined from the individual stencils.

Stencil Program Graph The graph consists of input, kernel and output nodes. The input nodes represents the actual data array inputs, the kernel nodes represent stencil computations and the output nodes store the result of the computation. Edges between them are called channels and provide forwarding and buffer functionality for the pipeline.

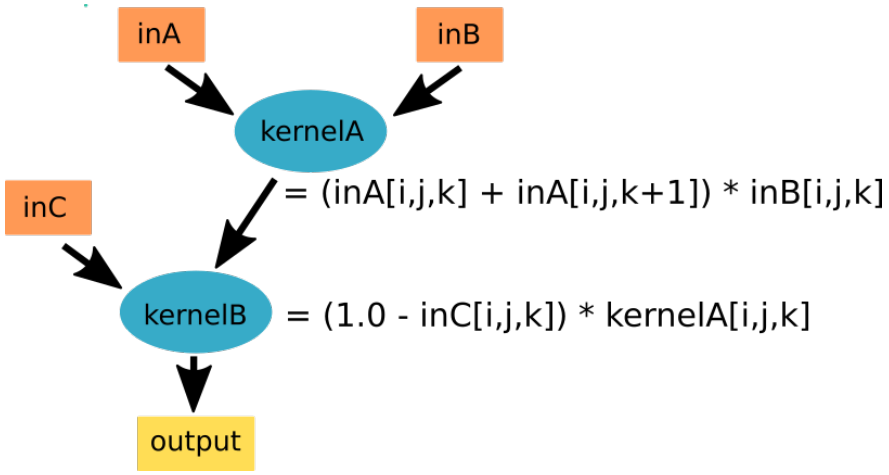


Figure 3.19: Example Kernel Chain Graph.

Latency The critical path latency of the stencil program graph is an important metric to compute the actual runtime of the program in hardware. Given the design frequency, the (theoretical) runtime is: $\frac{(critical\_path\_length + X*Y*Z)}{frequency}$. The critical path computation is identical to the calculation for the compute graph. We can assign latency zero to the output nodes and walk up the tree to the inputs while adding the critical kernel latency computed by the ComputeGraph.

Delay Buffer We call the second type of buffering required to run a fully pipelined design delay buffer (beside the internal buffer described in the kernel section 3.2.1). As the name suggest, its purpose is to latch intermediate data till the kernel is ready to read from the channel. This data dependency arises at the inter-kernel-level, which is why the stencil program graph has to take care of this. The following example illustrates a case where this is required.
The output of kernelA for kernelD is ready some time before the data stream from kernelC to kernelD arrives. Since kernelD cannot start processing the input data from kernelA, but rather has to wait for the data from kernelC, we have to introduce a delay buffer at the edge from A to D. The size of this delay buffer is (assuming field accesses of D from C and A are identical e.g kernelD[i,j,k] = kernelA[i,j,k] + kernelC[i,j,k]) given by the latency of the path kernelA-kernelB-kernelC-kernelD.

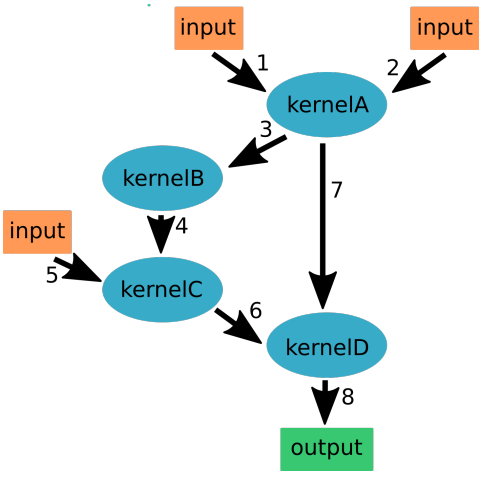


Figure 3.20: Example stencil program.

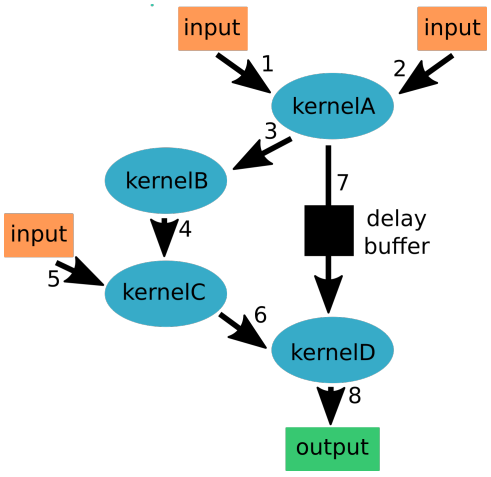


Figure 3.21: Example stencil program with delay buffer.

3.3 FPGA Model

Since we are dealing with a memory bound problem and seek to optimize for maximal throughput, our model is very data-centric, especially since todays state-of-the-art FPGAs offer such a huge amount of hardened digital signal processors (DSPs) (e.g. 10 TFLOP/s peak 32-bit floating point operations on a Stratix 10). Therefore we assume that we are limited by either on-chip memory capacity or off-chip memory bandwidth.

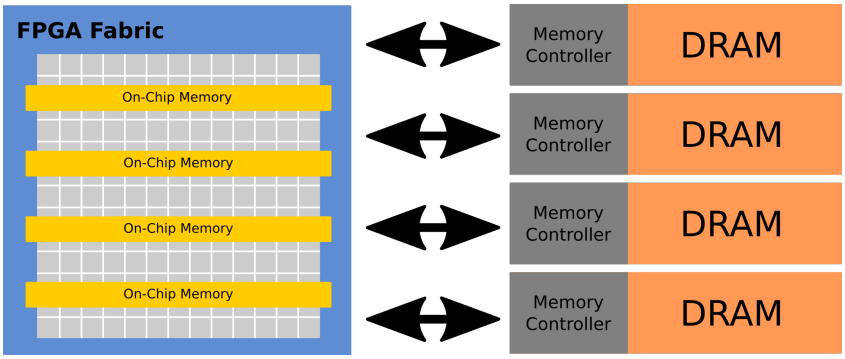


Figure 3.22: Basic memory model: Fast on-chip memory, slow DRAM memory and memory bandwidth.

3.3.1 Fast and Slow Memory

Since we fully pipeline the whole stencil program, we need to lay out all the internal and delay buffers in memory. While the fast on-chip memory of the FPGA has very low latency/high throughput and a low energy footprint compared to the slow DRAM memory (including the data path to the DRAM), it is a limited resource and we have to carefully choose which buffers we allocate in fast memory and which we better swap out to the slow memory.

3.3.2 Memory Bandwidth

Furthermore, even before having filled up the whole DRAM with buffers, we might run into the issue that we run out of bandwidth between the DRAM and the FPGA fabric. Therefore, memory bandwidth is a key metric too, which we have to take into account in the optimization process.

3.4 Hardware Mapping: DaCe

Despite our high-level optimization, it has been shown that FPGA designs can gain significant performance increase [20] if the HLS design has been further optimized and equipped with the suitable pragmas and formulations. Therefore, we decided to decouple the high-level representation from the lower-level and use DaCe [1], which is able to compile FPGA code with high utilization and performance from the intermediate representation, the so called Stateful DataFlow multiGraph (SDFG) [20].
The SDFG intermediate representation is a directed graph of directed acyclic multi graphs, where the nodes represent computations and storage while the edges represent data movement. Figure 3.23 summaries all graph primi-

[1] https://github.com/spcl/dace

tives of the stateful dataflow multigraph.

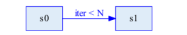
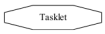
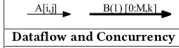
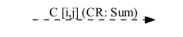
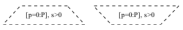
Primitive	Description
Turing-Complete Construction	
	**State**: State machine element.
	**Data**: N-dimensional array container.
	**Tasklet**: Fine-grained computational block.
	**Memlet**: Data movement descriptor.
Dataflow and Concurrency	
	**Stream**: Streaming data container.
	**Map**: Parametric graph abstraction for parallelism.
	**Conflict Resolution**: Defines behavior during conflicting writes.
Nesting and Subgraph Aliases	
	**Invoke**: Call a nested SDFG.
	**Reduce**: Reduction of one or more memlet axes.
	**Consume**: Dynamic mapping of computations on stream elements.

Figure 3.23: SDFG primitives syntax and description. [20]

In order to get a better understanding we look at the transformation of vector addition from python code to the SDFG graph representation.

```
def vector_add(A, B):
  return list(map(lambda x,y: x+y, A, B))
```

The corresponding graph starts with the two N-dimensional array containers A and B, which are connected by a data movement descriptor (memlet) through a map with the fine-grained computational block (tasklet). The map primitive splits the N-dimensional problem into N parallel 1-dimensional sub-problems in order to exploit the implicit parallelism. This is illustrated in figure 3.25 for the case N=3. The tasklet describes this element-wise computation:

$$C[i] = A[i] + B[i] \tag{3.12}$$

The resulting is then mapped back to the N-dimensional form and finally stored in the output data container C as shown in figure 3.24.

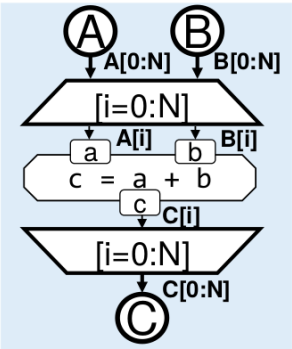


Figure 3.24: SDFG in parametric form. [20]

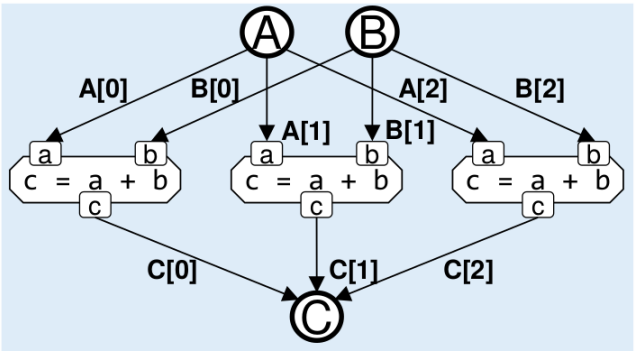


Figure 3.25: SDFG expanded for N=3. [20]

This general graph representation builds the foundation for the transformation and optimization to different hardware platforms, which makes DaCe a very powerful toolbox. It incorporates many optimization techniques such as tiling and fusion in order to generate efficient code for the specific platform. Figure 3.26 shows the complete transformation process form the domain specific high-level program to the hardware-specific optimized design.

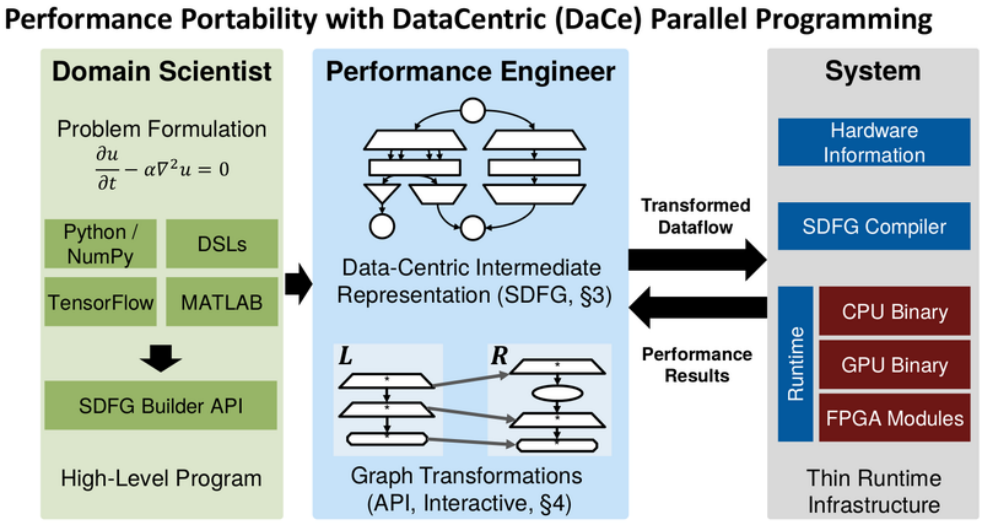


Figure 3.26: *Presentation Slide: Data-Centric Parallel Programming, Torsten Hoefler, invited talk at ISC'19, Frankfurt, Germany*

3.5 Conclusion

So far, we have seen that the actual problem to solve is trading the different compromise in resource allocation against each other to find a feasible and well performing design. The foundation for such an optimization has been laid down in this chapter by formalizing the analysis of resource requirements. In the next chapter, we will have a look on how we can formalize a objective and find the (theoretical) optimal solution within our specific assumptions.

Chapter 4

Optimizer

The aim of mathematically modeling stencil programs for reconfigurable hardware is to have a formal way of arguing about the optimal and efficient usage of the available resources. This section is dedicated to explaining what we are optimizing for and how we are achieving a maximized objective.

4.1 Objective

In stencil programs, each compute operation usually depends on multiple data field accesses which makes them very memory heavy applications. The re-programmable nature of FPGAs allows us to make use of the fine grained access to fast on-chip memory in order to speed up or shorten the data path a data element has to take from its storage location to the actual compute unit. The objective we seek to achieve is to make optimal use of the very limited resource of fast memory, in conjunction with the available memory bandwidth and slow memory. By caching resources for re-use and exploitation of the efficient pipelining nature of modern FPGAs we try to get a fully pipelined design with a minimal amount of pipeline stalls due to waiting for data to arrive.
We mathematically formalize the problem by formulating it as an linear program.

Objective 1: minimize fast memory usage: $\min_{(X,Y)} F(X,Y)$

Objective 2: minimize communication volume: $\min_{(X,Y)} COM(X,Y)$

Objective 3: optimize for ratio: $\min_{(X,Y)}(\text{RATIO} - \frac{F(X,Y)}{COM(X,Y)})$

subject to:

$F(X,Y) = \sum_{(i,j)} D(X_{ij}) * (1 - X_{ij}) + \sum_{(i,j)} D(Y_{ij}) * (1 - Y_{ij}) \leq$ FAST_MEMORY_BOUND

$$S(X,Y) = \sum_{(i,j)} D(X_{ij}) * X_{ij} + \sum_{(i,j)} D(Y_{ij}) * Y_{ij} \leq \text{SLOW\_MEMORY\_BOUND}$$

$$C(X,Y) = \sum_{(i,j)} \text{comm\_vol}(X_{ij}) + \sum_{(i,j)} \text{comm\_vol}(Y_{ij}) \leq \text{COMMUNICATION\_VOLUME\_BOUND}$$

where the variables are denoted as:

$$X_{ij} = \begin{cases} 1, & \text{if j-th part of the internal buffer of kernel i allocated in slow memory} \\ 0, & \text{otherwise} \end{cases}$$

$$Y_{ij} = \begin{cases} 1, & \text{if the delay buffer between kernel i and j is allocated in slow memory} \\ 0, & \text{otherwise} \end{cases}$$

$\forall X_{ij} : D(X_{ij}) =$ size of j-th part of the internal buffer of kernel i

$\forall Y_{ij} : D(Y_{ij}) =$ size of the delay buffer between kernel i and j

In the next section we will show the optimization approach to achieve this goal and further explain how the function comm_vol is derived.

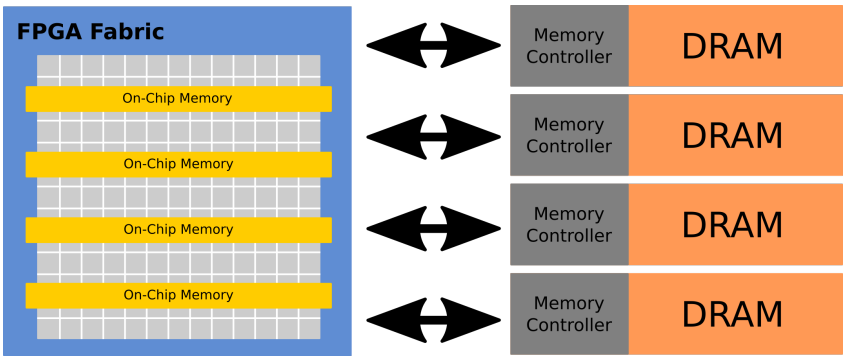


Figure 4.1: High level overview of the abstracted FPGA with the key resources: fast/slow memory and memory bandwidth.

4.2 Approach

We start our algorithm by putting all buffers (internal and delay) into fast memory. Then, we swap out the buffer that uses the memory *most inefficiently* to slow memory. We repeat this till we reach the goal of the strategy chosen. We will walk through an example to get an intuition of the decision for the right choice of swapping out.

Most Inefficient Buffer We move the buffer with the highest metric $\frac{\text{memory size(buffer)}}{\text{communication volume(buffer)}}$) from fast to slow memory. In other words, we "pay" communication volume and slow memory (we do not bother about the slow memory, since this is usually orders of magnitudes larger than the fast memory) for getting rid of some amount of buffer space. The buffer with the highest metric value get is optimal to swap out, since we "pay" the least amount compared to the size of the buffer we can remove from fast memory.
Since the communication volume of a buffer depends on the predecessor and successors location in memory, we will walk through an example to derive the rule for calculating the actual value.

Initial state In figure 4.2 you can see the channel from the input kernel to the next kernel, split up into the delay buffer and the junks of the internal buffer (each part is from one field access to the next). For example a kernel *out[i] = in[i-2] + in[i] + in[i+10]* would have one delay buffer and the internal buffer split into two parts of size 2 and 10. At the moment, they are all allocated in fast memory. We will analyze the communication volume for the darker part of the internal buffer dependent on the location of its predecessor and successor.
We can observe that the communication volume imposed by our buffer is zero if the predecessor and successor are in located in fast memory too.

Swap Out of Marked Buffer We moved out the marked buffer to slow memory (figure 4.3), which imposes an additional communication volume of $2 * X * Y * Z$

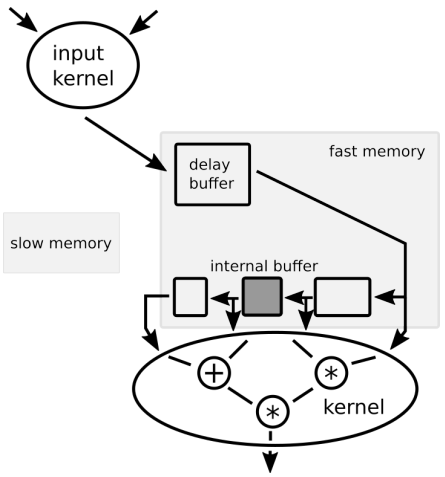


Figure 4.2: Scenario 1: All buffers are allocated in fast memory.

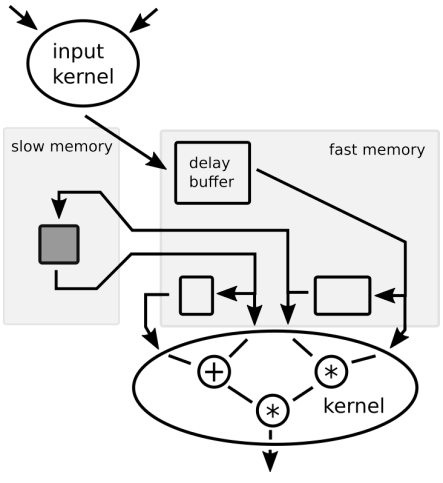


Figure 4.3: Scenario 2: The marked internal buffer is in slow memory but its predecessor and successor are in fast memory.

Swap Out Successor This time, the successor of the marked buffer was already allocated in slow memory while the predecessor is still allocated in fast memory as shown in figure 4.4. Moving out the marked buffer to slow memory imposes an additional communication volume of $1 * X * Y * Z$.

Swap Out Predecessor and Successor Figure 4.5 shows a situation where the predecessor and the successor have both already been swapped out. Swapping out the marked buffer in this situation actually does not impose any additional communication volume. The only thing changing is the direction of data movement (The output of the marked buffer was going from fast to slow, and now it goes into opposite direction)

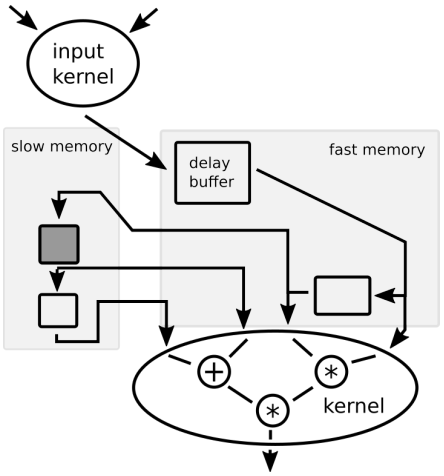


Figure 4.4: Scenario 3: The marked internal buffer is in slow memory with its successor. The predecessor is allocated in fast memory.

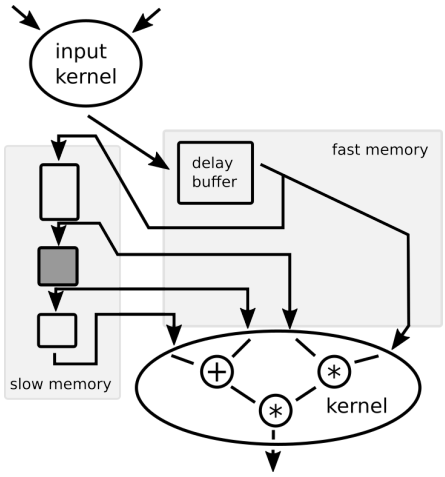


Figure 4.5: Scenario 4: The marked internal buffer is in slow memory together with its predecessor and successor.

Conclusion The general rule for swapping the buffer out as explained in the previous example is given by: pre=predecessor suc=successor global array size = X*Y*Z = C

- pred and succ in fast mem: 2*C
- pred in fast, succ in slow mem: C
- pred in slow, succ in fast mem: C
- pred and succ in slow mem: 0

Optimizer On a high level, the optimizer works as follow:

Algorithm 1 Optimizer

```
while objective_not_reached do
    find buffer with highest metric value
    swap the buffer out
    update communication volume metric values for
    its neighbors
end while
```

4.3 Strategies

Even though we have a fixed objective and a clear understanding on how we optimize, there are several scenarios with different requirements we can optimize for.

Minimization of Communication Volume Reduction of communication volume implies maximal usage of the fast on-chip memory resources and reduces data movement which is a natural choice of optimization.

Minimization of Fast Memory Minimization of fast memory is a favorable strategy for example if one wants to use the remaining fast memory for other purposes. One such scenario would be tiling.

Optimization to Ratio This optimization strategy tries to optimize the buffer allocations in the best way to fit the ratio $\frac{\text{fast memory}}{\text{communication volume}}$ while staying below the upper bound of communication volume. This is a very useful metric if we assume that the DRAM (slow memory) is not our limiting factor and want to find out how many devices are required to fit the whole design onto it. If we choose the ratio of $\frac{\text{fast memory}}{\text{communication volume}}$ for a particular hardware device, we can reason about the number of devices required to implement our design as: number of required devices $= \frac{\text{fast memory size after optimization}}{\text{fast memory on a single device}}$

Chapter 5

Simulator

We will go through a few arguments to underline that this is a good investment for time saving of future development.

Debugging / Design Test Everybody that has already run FPGA designs in hardware knows that it is really hard to properly debug them in case of errors happening (e.g. deadlocks, etc.), since there is no notion of setting a breakpoint and look at the intermediate values as we are used to do this in software. What you can do is setting up a debug channel, which enables you to see at least what is happening on the device or where it breaks, but the capabilities remain very limited.
That's why we want the functionality to simulate the design in software first in order to eliminate as many bugs as possible. In case of problems, we can detect and debug them in the integrated development environment which gives us access to the current state of the whole design. This significantly reduces the error rate and increases the certainty of running on the actual hardware without issues.

Correctness Furthermore, changes in the code generator, which will often occur in the future while doing the low-level performance optimization, might result in a working design, that produces invalid output data. Running the same input data on the simulator and compare the computation result to the one from the hardware gives us the opportunity to check the correctness of the hardware design without having to write a dedicated golden model implementation for every problem.

Performance Metrics Running the design in software gives us the benefit of being able to collect whatsoever information or metric we are interested in. This means that we cannot only test the design for correctness, but we can

for example find out if the buffer size estimate is tight, in other words, that is the size exact and not just large enough.

5.1 FPGA Execution Model

The software execution model of the abstracted FPGA device is modular built up and transitions multiple phases, which we will have a look at now.

5.1.1 Initialization

In the initialization phase, we set up and load all input data into their corresponding allocated location. This is similar to the transfer of the input data from the host memory to the reconfigurable device.

5.1.2 Step Execution

After initialization we can start with the actual execution process. A single run of the so called step execution represents a clock cycle on the actual hardware. This is repeated till the stop condition is met, which will end the execution and initiates the transition over to the finalization process.

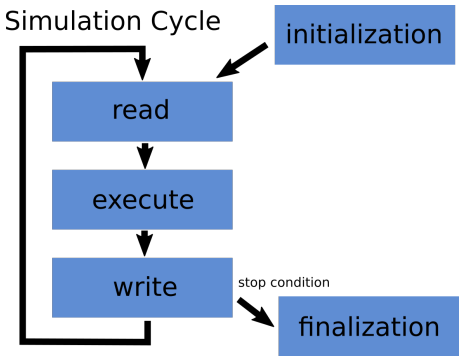


Figure 5.1: Simulation Cycle: Step by step execution.

Read In the read phase, all kernels check if data is available from all input channels. If this is the case they can consume them.

Execute In the execution phase, all kernels that have successfully read data elements from all input channels can compute the output value according to the kernel function. In order to account for the operation latencies, the result is not directly written to the output channel, but rather hold back for *latency* cycles.

Write In the write phase, all kernels with valid output data write the result to the output channels, which are either input channels of other kernels or the channel collecting the final result for the transfer back to the host memory.

Stop Condition The problem size respectively the number of data elements to process is given in the problem definition. Therefore we know after every kernel has written that many elements to the output channels, we can safely switch to the finalization phase.

Finalization The finalization process takes care of writing the final result to the filing system.

5.2 Performance Metrics

Buffer Utilization We are not only interested in the information of having large enough buffers, but also for example if they are actually filled 100% at some instance in time. If this would not be the case we could reduce their size and therefore save buffer space. Furthermore, we might, at some point, come up with a stencil program or kernel merging strategy that might reduce the peak buffer size usage and therefore could reduce the overall memory consumption.

Start and End Time Another interesting per-kernel metric is the timing of the actual start (program counter time) and end time. This allows us to see if there are any stalls happening for this kernel.

5.3 Error Handling

The buffer optimization goal is to make all buffers (internal and delay buffers) as large as necessary, but as small as possible. In case of a kernel producing a valid output data element, but any of the output channels is already full indicates that the buffer was too small. This holds true since we are running the design synchronously. In such a case, we can output the current state of the simulation (buffer data, performance metrics, etc) which gives us the possibility to resolve these issues in software.

5.4 Conclusion

The software simulation gives us great flexibility and insight into understanding what is happening on the hardware itself and helps us to prevent and detect bugs in an early stage where we still have the freedom to see

into the global simulator state. Furthermore, we can gain great insights into performance measures which might lead to optimization and better use of the hardware resources.

Chapter 6

StencilFlow Implementation

6.1 Software Overview

StencilFlow consists of many modules, some of them are very specific and dependent on others while we tried to keep many modules as general as possible for reuse within the tool. Figure 6.1 gives you an overview how the different classes are nested and where they are instantiated. This section is

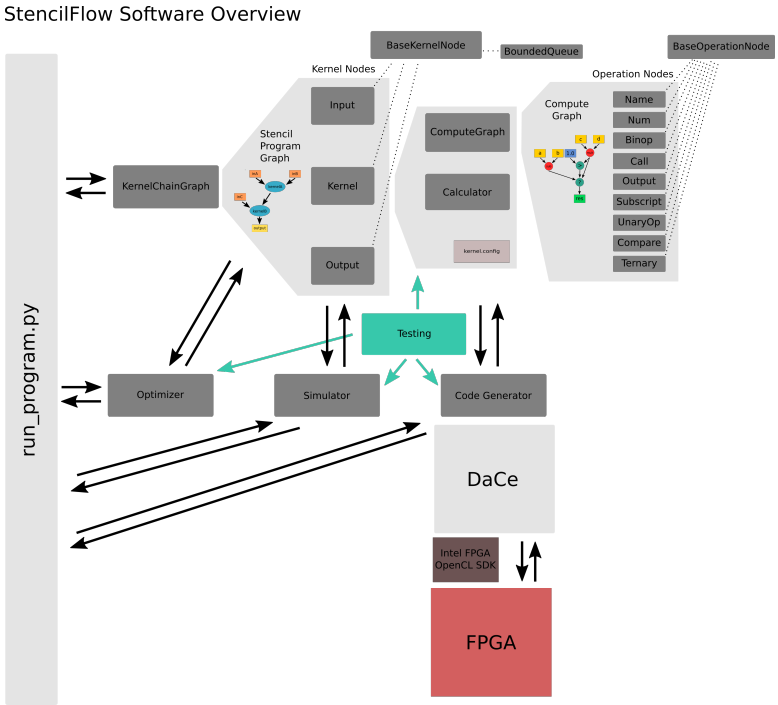


Figure 6.1: Overview of the StencilFlow class structure and their interaction.

dedicated to give you a detailed insight into the actual implementation with additional clarification why and how we implemented certain functionality.

This code snippet gives you an overview how the main classes interact with each other:

```
# instantiate the KernelChainGraph
chain = KernelChainGraph(path=args.stencil_file,
                         plot_graph=args.plot,
                         log_level=args.log_level)

# instantiate the Optimizer
opt = Optimizer(kernels=chain.kernel_nodes,
                dimensions=chain.dimensions,
                log_level=chain.log_level)
opt.optimize_to_ratio(1e-2)

# instantiate the Simulator
sim = Simulator(input_config_name=args.stencil_file
                chain=chain,
                write_output=False,
                log_level=args.log_level)
sim.simulate()

chain.report(args.stencil_file)
sim.report()
```

6.2 Input File

The input json file contains all information required to analyze, optimize and run the given stencil program. We will go through its sections and explain them in detail.

```
{
"inputs": {
    "inA": {
        "data": [
            19.30100000000000000e+00,
            19.43700000000000000e+00,
            19.30100000000000000e+00,
            19.03700000000000000e+00,
            19.36400000000000000e+00,
            19.31100000000000000e+00,
            19.93600000000000000e+00,
```

```
            20.10300000000000000e+00,
            20.24600000000000000e+00,
            20.03800000000000000e+00,
            20.36800000000000000e+00,
            20.03900000000000000e+00,
            20.24700000000000000e+00,
            20.03400000000000000e+00,
            20.10300000000000000e+00,
            20.56400000000000000e+00,
            20.94800000000000000e+00,
            19.19300000000000000e+00,
            19.36700000000000000e+00,
            19.98700000000000000e+00,
            20.43200000000000000e+00,
            20.95200000000000000e+00,
            21.34400000000000000e+00,
            21.07300000000000000e+00
        ],
        "data_type": "float64"
    },
    "inB": {
        "data": "inB.csv",
        "data_type": "float64"
    },
    "inC": {
        "data": "inC.dat",
        "data_type": "float32"
    }
},
"outputs": [
    "kernelB"
],
"dimensions": [
    2,
    3,
    4
],
"program": {
    "kernelA": {
        "computation_string": "out = 3.14 * (inB[i,j,k]-inB[i,j-1,k]);
                               res = (inA[i,j,k] + out) * cos(out);",
        "boundary_condition": {
            "inA": {
               "type": "constant",
```

```
                "value": 1.0
            },
            "inB": {
                "type": "constant",
                "value": 7.5
            }
        },
        "data_type": "float64"
    },
    "kernelB": {
        "computation_string": "res = kernelA[i,j,k] + inC[i,j,k];",
        "boundary_condition": {
            "kernelA": {
                "type": "copy"
            },
            "inC": {
                "type": "copy"
            }
        },
        "data_type": "float64"
    }
}
}
```

Inputs The inputs section is dedicated to the input data arrays. For each input data array, there exists an entry that either contains data directly or references to an external file in binary or csv format. Furthermore, it specifies the data type and precision of the data.

Outputs The output section contains all kernel names of which data should not be discarted, but rather written back to the host computer as the output or computation result. This list can contain a single item or multiple entries.

Dimensions The dimensions value represents the global problem size and should correspond to the dimensions of the data arrays too (e.g. dimX*dimY*dimZ = size(input array)).

Program The program section contains information about the kernels of the stencil program. The computation_string represents the kernel computation. The boundary condition subsection defines the condition per input in case of an out-of-bound data read. The data type specifies the precision of the computation.

Note: The precision value of data_type might have severe performance impact due to the impact in buffer requirements and the usage of hardened floating point operators.

6.3 Bounded Queue (bounded_queue.py)

The BoundedQueue class represents the basic building block when it comes to handle data in a FIFO manner. It can act as a feed to push data into the pipeline (input nodes), connect stencil chains as channels (internal and delay buffer), it does latency simulation within the kernel and collects all final results to store them on disk. We will go through the implementation details and explain for what the functionalities are useful.

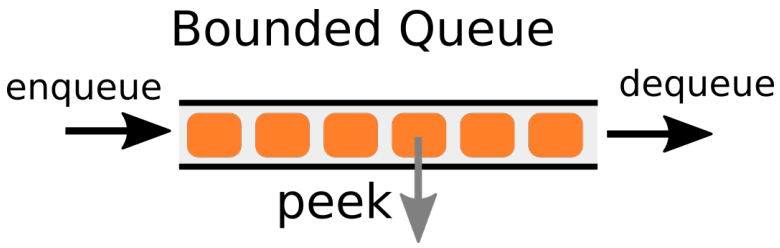


Figure 6.2: Working principle of the bounded queue implementation.

Generic There are several short-cut implementations for generic functionality necessary to work with these data structures such as getting the actual size, check if the data structure is full respectively empty or print the basic instance info in a well-arranged way.

dequeue / try_dequeue Removing and retrieving a single element from the BoundedQueue can be achieved using the dequeue method. The difference between the two implementations lies in the error handling. In case of an empty queue, dequeue raises an exception while the try_dequeue method simply returns False instead of a data element in case of an empty queue.

There are valid use cases for both of them. For example in case of a kernel program trying to read data from input channels, that might or might not be available yet, try_dequeue might be the better choice to avoid having to handle the exception.

enqueue / try_enqueue Adding a single element to the BoundedQueue can be achieved using the enqueue method. The difference between the two implementations lies in the error handling. In case of a full queue, enqueue raises an exception while the try_enqueue method simply discards the data item and indicates success or failure by the boolean return value.

peek / try_peek_last The peeking functionality allows to retrieve items add arbitrary positions in the queue without actually removing them from the queue. Especially try_peek_last is very useful for the simulator to read the next element that is being feed into the kernel without actually touching the data.

import_data / export_data For the initial setup and cleanup we usually have to add many elements to a queue or retrieve all data elements from it. These functionalities are provided by the import_data respectively export_data methods.

6.4 Calculator (calculator.py)

The Calculator class provides the service of given a mathematical expression and a map from all variables to their corresponding values, to compute the numerical result of the expression, even for conditional expressions in the form of *if(condition) true_expression else false_expression*.

Internals The functionality is implemented by first parsing the expression into an abstract syntax tree. This tree is used later on to walk through and compute the intermediate values leading to the final result by having custom AST tree walker functions implemented.

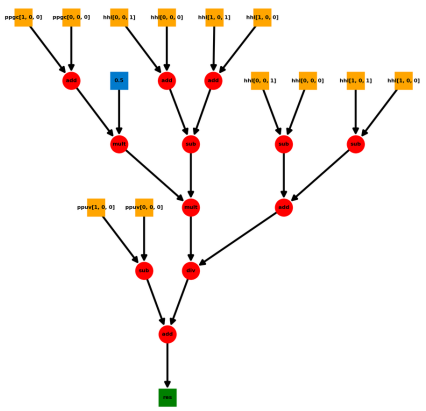


Figure 6.3: Example of the high level visualization of a abstract syntax tree compute graph.

eval_expr The Calculator class provides a single method for evaluation of an expressing. Given a mathematical expression and a map of all variable names to its corresponding value, the calculator computes the final result.

This functionality is required for the simulator to be able to compute the kernel result, given all input values.

Example Usage The following example should illustrate the functionality of the Calculator class.

```
variables = dict()
variables["a"] = 7
variables["b"] = 2

for var in variables:
  print("name: {}, value: {}"
        .format(var, str(variables[var])))

computation = "cos(-a + b) if (a > b) else (a + 5)
calculator = Calculator()
result = calculator.eval_expr(variables, computati
print("{} = {}".format(computation, str(result)))
```

Returns:

```
name: a, value: 7
name: b, value: 2
cos(-a + b) if (a > b) else (a + 5) * b = 0.28366
```

6.5 Compute Graph Nodes (compute_graph_nodes.py)

The ComputeGraphNodes implement the BaseOperationNodeClass and are nodes of the ComputeGraph. Their main functionality beside have different nodes types in the three is to generate names and symbols for visualization from the given abstract syntax tree node.

Name: The Name node class represents the variable name node in the computation tree.

Num: The Num node class represents the numeral node (constant values) in the computation tree.

Binop: The Binop node class represents the binary operations (e.g. addition, subtraction, etc.) in the computation tree.

Call: The Call class represents the function calls (e.g. sin/cos,..) nodes in the computation tree. We support functions with single arguments, which

could be extended to multiple argument functions if necessary.

Output: The Output class represents the output node in the computation tree.

Subscript: The Subscript class represents the array field access nodes in the computation tree. (e.g. inX[i,j,k+1]). It does convert the [i,j-2,k+1] index to relative number indices i.e. [0,-2,1]

Ternary: The Ternary operator class represents ternary operation of the form: expression_true if comparison_expression else expression_false

Compare: The Comparison operator class represents the comparison of two expression by a comparator (e.g $=, >=, ..$).

UnaryOp: The UnaryOp operator class represents unary operations. In our case we only support negation (mathematical - sign) as unary operation yet.

6.6 Compute Graph (compute_graph.py)

The ComputeGraph class represents the actual computation of a kernel instance. It analyses the field access patterns and metrics such as the critical path latency. Furthermore, it generates a high-level graph representation from the input fields through the computational nodes to the output, which can be visualized nicely.

Configuration Performance metrics such as latency greatly depends on the underlying FPGA hardware. This is why we did not hard-code such values, but rather provide the flexibility to load the configuration of operation latency values at runtime.

```
{
"op_latency": {
        "add": 16,
        "sub": 16,
        "mult": 16,
        "div": 128,
        "inv": 16,
        "sin": 128,
        "cos": 128,
        "tan": 128,
        "sinh": 128,
```

```
        "coshh": 128,
        "comparison": 16,
        "conditional": 16,
        "neg": 16
        }
}
```

Nodes The computation graph contains several different node types, each of them represents either a data source, a data destination or some type of computation.

- Name: variable names
- Num: numeral (constant number)
- Binop: binary operation (e.g. addition, subtraction etc.)
- Call: function call (e.g. sin(), cos(), etc.)
- Output: output/result node
- Subscript: array access (e.g. inX[i,j,k+1])
- Ternary: ternary (conditional) operation node of the form: if_expr if condition else else_expr
- Compare: part of the ternary operator, tests the condition
- UnaryOp: unary operations (e.g. negation)

setup_internal_buffers This method takes care of computing the internal buffer size. This can be achieved by first finding the highest and the lowest access index per field and compute the difference which is exactly the internal buffer size plus one. internal buffer size(field) = max_field_access(field) - min_field_access(field) + 1.

Data Reuse, Multiple Expressions There are many cases where part of the kernel expression is occurring more than one time. For efficiency reasons, it makes sense to compute them only once. We provide this functionality by allowing to split the expression into subexpression, which can be referenced multiple times. This can be achieved using the following syntax: ”res = -a if (a+1 $>$ b-c) else b; b = d + e” where b is being substituted twice by (d + e). From the implementation perspective, we have to account for the processing of both parts of the expression as well as contracting the output node b (from b = d + e) with the input node b (in res = -a if (a+1 $>$ b-c) else b;), which is being achieved by the contract_edge method.

generate_graph The generate_graph method creates a high-level networkx graph consisting of OperationNodes for further visualization and analysis. The expression is parsed using the abstract syntax tree framework which gives us the ability to walk through the AST tree in order to extract all the required information. The ast_tree_walk method is a recursive method that does the following steps:

1. create a new node from node and the node number and add it to the graph
2. depending on the type of node, call ast_tree_walk recursively for left child (node number: 2*parent + 1) and right child (node number: 2*parent)
3. add edge from the new node to the recursive child/children
4. return new node

plot_graph Visualizing the problem instances greatly helps to get a good understanding of the problem and seeing what is going or what we could further optimize. This method accounts for that by plotting a well-arranged overview of the ComputeGraph.

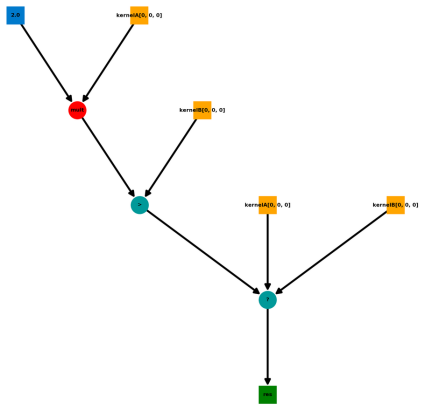


Figure 6.4: Example of the high level visualization of a abstract syntax tree compute graph.

calculate_latency The latency of the critical path is a key metric for our problem question and part of the ComputeGraphs functionality. This can be achieved by a recursive tree walk from the output node up to the field accesses while carrying out the latency propagation i.e. latency_to_outp(child) = max{latency_to_outp(child), latency_to_outp(self) + operation_latency(self)}
Remark: The reason for using the maximum of both values is that the tree is not a

classical mathematical tree, but rather a directed acyclic graph. It can happen due to the allowance of substitution/reuse of identical part of the computation to have a cycle if we look at it as an undirected graph. This behavior can be seen by the example ComputeGraph above.

6.7 Base Node (base_node_class.py)

The base node class is not a class by itself, but rather contains the abstract base nodes for the ComputeGraph and the KernelChainGraph.

BaseKernelNodeClass The BaseKernelClass provides all the basic fields and functionality for its subclasses which are the Input, Kernel and Output classes. These are nodes of of the KernelChainGraph.
Furthermore, it provides a fall-back implementation of the generate_label functionality.

BaseOperationNodeClass The BaseOperationNodeClass class provides all the basic fields and methods for its subclasses (Num, Subscript,..). These are the nodes of the ComputeGraph.
Furthermore, it provides a fall-back implementation of the generate_label and functionality. In addition, it contains an method generate_name to generate the node name from the AST node, which all subclasses have to implement.

6.8 Input (input.py)

The Input class is a subclass of the BaseKernelNodeClass and represents an Input node in the KernelChainGraph. Its purpose is to feed input data into the pipeline/data flow design.

init_queues This method sets up a dedicated queue for each output channel. This is necessary since some of the kernels might have to wait for other inputs to get available till they start reading from this channel.

try_write This method is called by the simulator in the step execution cycle. The input node loops over all output channels and tries to add a data element to them. If the corresponding data queue is already empty (all elements already feed) it inserts a ”bubble” (None) in order to keep the data flow running. If the output channel is full, it does nothing.

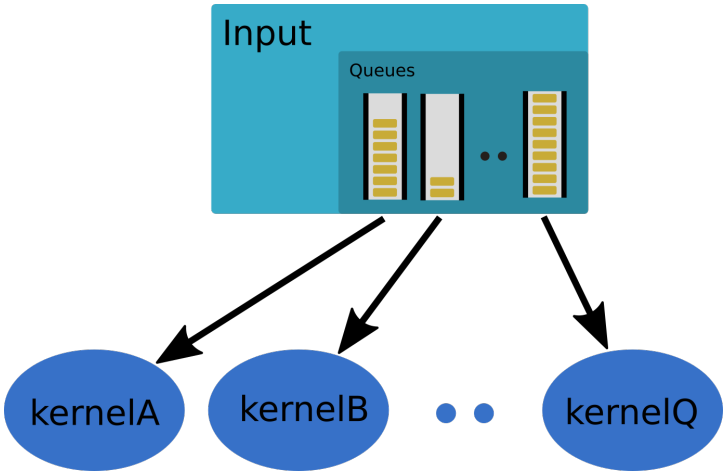


Figure 6.5: The functionality of the Input node. Feed each channel individually.

init_input_data The init_input_data method initializes the internal queues with data from the config file or the file system. It currently supports the following file types: implicit (array defined in the input config file), binary (.dat, .bin, .data) , h5 and csv.

6.9 Output (output.py)

The Output class is a subclass of the BaseKernelNodeClass and represents an Output node in the KernelChainGraph. Its purpose is to store data coming from the pipeline/dataflow design.

try_read This method is called by the simulator in the step execution cycle. It simply tries to read from the input channel and stores the element into the data_queue on success.

write_result_to_file The write_result_to_file method writes the content of the internal queue with the computation results to the binary file at *results/INPUT_CONFIG_NAME/SELF.NAME_simulation.dat*. This can be used for example to compare the simulator values with the result coming from the actual hardware implementation.

6.10 Kernel (kernel.py)

Overview The Kernel class is used for four different scenarios:

1. Initialization: setup and computation of buffer sizes and latencies.
2. Optimization: set/reset swap out buffer flags

3. Simulation: run data through the data flow graph by repeatedly calling try_read/try_execute and try_write
4. Code Generation: extraction of equations and buffer swap out flag

Initialization The initialization phase looks like:

1. instantiate Calculator class
2. instantiate ComputeGraph class with given compute_string
3. generate compute graph (networkx data structure)
4. compute compute graph latency (critical path)
5. compute and set up internal buffers
6. set up latency simulation out delay queue
7. compute access distance from furthest access to the center

Optimization The optimization framework takes kernels as input and sets or resets the field *swap_out* of the delay buffer or internal buffer parts to indicate whether or not the buffer should stay in fast memory.

Simulation The simulator makes us of the pre-defined step execution calls *try_read*, *try_execute* and *try_write*.

Code Generation The code generator makes use of the function generate_relative_access_kernel_string that transforms the input compute_string with given global dimensions to a single dimensional index based string.

```
dimensions are: [100, 100, 100]
input: res = a[i+1,j+1,k+1] + a[i+1,j,k] +
             a[i-1,j-1,k-1] + a[i+1,j+1,k] +
             (-a[i,j,k])
relative_access_kernel_string: res = ((((a[0] +
             a[-101]) + a[-20202]) + a[-1]) +
             (-a[-10101]))
```

print_kernel_performance This methods pretty-prints the performance metrics (maximum buffer usage, average buffer usage, total execution time, etc) gathered during the simulation.

update_performance_metric This method is called by the simulator every step execution in order to gather and process internal state metrics.

set_up_dist_to_center Computes for all fields/channels the distance from the furthest field access to the center of the stencil ([0,0,0]).
This is necessary and covers the following edge case: Suppose we have the kernel: *out[i,j,k] = in[i,j,k-2] + in[i,j,k+2]* without accessing the center element (in[i,j,k]) from the input. We look at the time two step execution cycles before the first actual execution of this kernel with valid input data. The front access is already in the position to read valid data while the back access is in the boundary condition which can lead to an invalid execution (if no center access is used it seems to be valid but in fact is not). In order to circumvent this, we introduce the distance-to-center and decrement it every step to make sure we start as soon as the center element is in the first position.

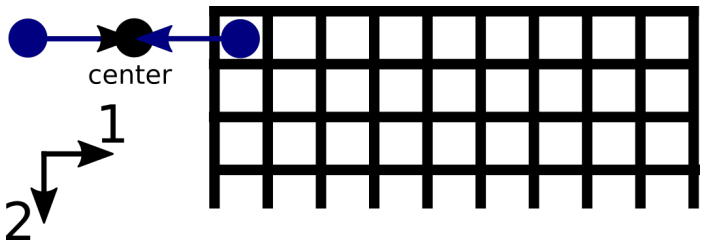


Figure 6.6: Edge Case: Distance to center.

iter_comp_tree This method iterates through the computation tree in order to generate the kernel string (according to some properties e.g. relative to center or replace negative index. The pseudo code for the function is: *Relative to center: The absolute array index value is either relative to the center*

Algorithm 2 iter_comp_tree(node)

get all predecessors pred
check what type of Node node is
depending on the node type, recursively call iter_comp_tree for the predecessor(s)
return the formated string e.g. for binary operation: LHS op RHS

node e.g. center is at 0 and [i,j,k+10] is at [10] or relative to the front access e.g. the center is at [-10] and [i,j,k+10] is at [0].
Replace negative index: In order to get a valid variable name string, we have to replace the negative sign '-' by 'n' otherwise it gets interpreted as a mathematical symbol (for the Calculator class) e.g. arrA_-20 gets replaced by arrA_n20

generate_relative_access_kernel_string Generates the relative (either to the center or to the furthest field access) access kernel string which is necessary for the code generator HLS tool.

reset_old_compute_state This is a helper function for the simulator to make sure that no artifacts from any previous simulation step is remaining. It clears the variable to value map and all execution related flags such as read_success.

convert_3d_to_1d Converts [i,j,k] to a flat 1D array index using the given dimensions [dimZ, dimY, dimX] using the formula: index = i*dimY*dimX + j*dimX + k = (i*dimY + j)*dimX + k (dimensional to absolute value formula from the model part).

remove_duplicate_accesses The list of field accesses must be cleaned from duplicate entries, since they can be handled concurrently and could increase the complexity of the computations of StencilFlow.

setup_internal_buffers This method creates and splits the internal buffers according to the pipeline model. Each internal buffer part has the length from one access of the field to the next. This is done in the following way:

Algorithm 3 setup_internal_buffers

```
sort all buffer accesses in the list
case single buffer access: internal buffer is size 1
case multiple buffer accesses: iterate over the list
compute difference from current to next access
create buffer of this size and add it to the buffers
```

buffer_number Computes the index position of the internal buffer part from the access position.
Example: out[i,j,k] = in[i,j,k-10] + in[i,j,k] + in[i,j,k+10] has two internal buffers (form k-10 to k, and from k to k+10). The data item for the access [i,j,k+10] comes from the delay buffer, the one for [i,j,k] form the first internal buffer and for [i,j,k-10] from the second internal buffer. If we call buffer_position([i,j,k+10]) we get '-1' which indicates the delay buffer. For buffer_position([i,j,k]) we get '0' and for buffer_position([i,j,k-10]) we get '1'

get_global_kernel_index Return the current position (simulator, program counter) within the computation as a list of the form [i,j,k]. This is required for the simulation to test if the data at a specific index is out-of-bound or

not.
Example: global problem size: [2,3,4].
(1) Current program counter index: 3, global kernel index: [0,0,3]
(2) Current program counter index: 7, global kernel index: [0,1,3]

is_out_of_bound Checks whether the current access is within bounds or not. This is required for the simulation to check if boundary conditions must be applied or regular data should be read.

get_data Returns data of current stencil access (could be real data or boundary condition)

test_availability This method checks if all accesses are available (and ready for execution). In addition, the method returns all accesses that are not available yet. Checks are different for different data types:

Numeral (Constant Values) They are always available and therefore nothing has to be checked.

Subscript (array) data accesses The buffers have to be checked if data is ready yet:

Algorithm 4 setup_internal_buffers

```
peek the last element of the delay buffer
peek the last element of all internal buffer junks
if none of the is 'None' then
    all data elements are available
else
    not all items are available yet
end if
```

move_forward Moves all items within the internal and delay buffer one element forward to free up space for a new data element. The furthest element is getting discarded since it is not used anymore.

decrement_center_reached This counter (individual to each field) gets decremented every time there is a data item at the output of the delay buffer (kernel access index position [0,0,0]). The counter reaching zero indicates that the kernel reached center for this field.

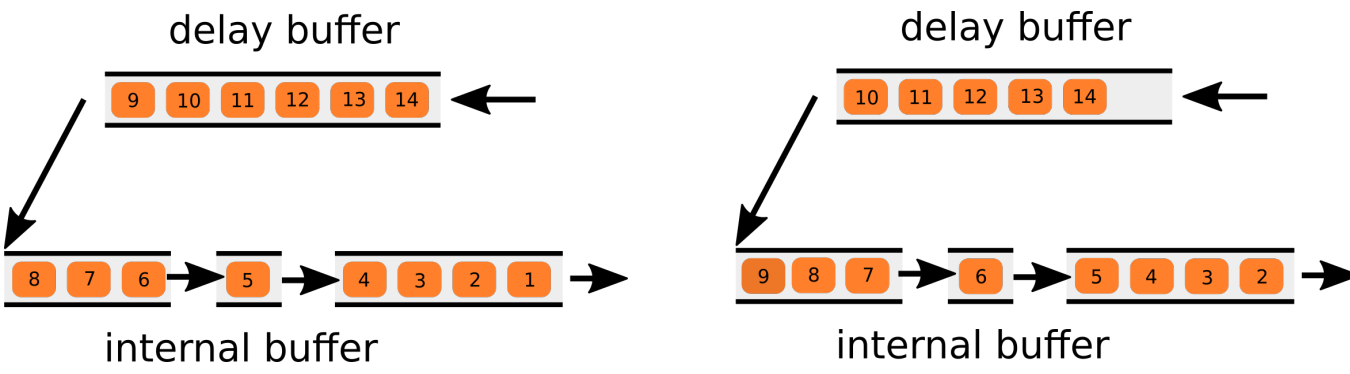


Figure 6.7: Move data forward.

try_read This is the implementation of the kernel reading functionality of the simulator. It tries to read from all input channels and indicates if this has been done with success. It does the following procedure written in pseudo code:

Algorithm 5 try_read

check if all inputs are available
case success: read all data items into the variable to value map var_map and set the read_success flag
case unsucessful: reset the read_success flag
decrement the center counter for all fields that have data at the output of the delay queue
check if all counter reached 0
case true: continue
case false: move all delay/internal buffer forward for the fields that did not reach center yet

try_execute This is the implementation of the kernel execution functionality of the simulator. It executes the stencil computation for the current variable mapping that was set up by the try_read() function. It does the following procedure written in pseudo code:

try_write This is the implementation of the kernel write functionality of the simulator. It does the following procedure written in pseudo code:

diagnostics The diagnostics method summarizes the internal kernel state. It either gets called upon an exception or at the end for the final report. It provides the following information:

Algorithm 6 try_execute

if check if center has been reached, read_success is set and the internal program counter is between 0 and PROBLEM_SIZE **then**
  get the relative access kernel string
  compute the result of the expression together with the variable map using the calculator
  add result to out_delay_queue
  increment the program counter
else
  add a None (bubble) to the out_delay_queue
end if

Algorithm 7 try_write

dequeue last element from the out_delay_queue
add it to the input channel of all kernel outputs respectively successor nodes

- program counter
- all_available flag
- center_reached
- exception traceback
- Inputs/Outputs:
  - delay buffer maximal size
  - delay buffer current size
  - delay buffer content
  - internal buffer content
- content of latency simulation buffer (out_delay_queue)

out_delay_queue The out_delay_queue's purpose is to simulate the computation latency of the computation graph. This is achieved by having a FIFO queue of length *max_latency* and by adding each result to the queue while sending out the last item from the queue each cycle instead of directly sending it out.

6.11 Kernel Chain Graph (kernel_chain_graph.py)

The KernelChainGraph class represents the whole pipelined data flow graph consisting of input nodes (real data input arrays, kernel nodes and output nodes (storing the result of the computation).

Call The KernelChainGraph can be called in the following way:

```
python3 kernel_chain_graph.py -stencil_file stencils/simulator12.json -plot -simulate
    -report -log-level 2
```

Initialization The following procedure is called to initially setup the internal structure and analyze the input configuration:

1. set up all internal data structures (dicts, lists,..)
2. read input config files
3. create all kernels
4. compute kernel latencies
5. connect kernels in the graph
6. compute delay buffer sizes
7. add channels to the graph edges
8. plot all graphs (if flag set)

plot_graph Visualizing the stencil program instances greatly helps to get a good understanding of the problem and seeing what is going or what we could further optimize. This method accounts for that by plotting a well-arranged overview of the StencilChainGraph.

connect_kernels This method connects all networkx nodes to a directed acyclic graph by matching the input names with the corresponding kernel names.

add_channels Add_channels packs all required information (name, delay buffer queue, internal buffer queue and data type) into a dictionary and adds it to the edge between the corresponding two nodes of the graph.

```
channel = {
  "name": name,
  "delay_buffer": self.kernel_nodes[dest.name].
                  delay_buffer[src.name],
  "internal_buffer": dest.internal_buffer[src.name],
  "data_type": src.data_type
}
```

import_input This helper method reads the input file and adds each part to the corresponding field in the class.

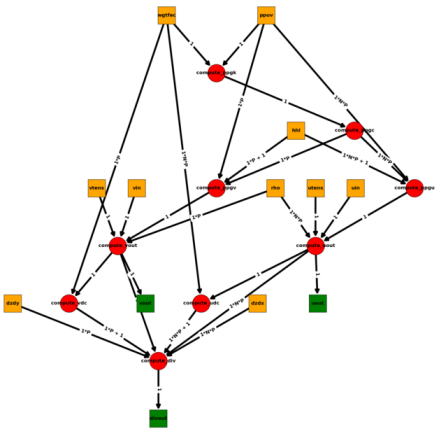


Figure 6.8: Example stencil chain graph.

total_elements Reduces the global problem size to a single scalar value.

create_kernels Instantiates the kernels and adds them to the networkx graph.

compute_kernel_latency Fills a global dictionary of the individual critical kernel computation paths.

compute_delay_buffer Computes the delay buffer sizes in the graph by propagating all paths from the input arrays to the successors in topological order. Delay buffer entries should be of the format: kernel.input_paths: "in1": [[a,b,c, pred1], [d,e,f, pred2], ...], "in2": [...], ... where inX are input arrays to the stencil chain and predY are the kernel predecessors/inputs. This is being achieved in the following way:

compute_critical_path_dim Computes the maximum latency critical path through the graph in dimensional format.
Note: Since we know the output nodes as well as the path lengths the critical path is simply max { latency(node) + max { path_length(node) — node in output nodes }}

Algorithm 8 compute_delay_buffer

sort the graph in topological order
go over all nodes in the topological order
process delay buffers (no new delay buffers can occur since its is topologically sorted)
max delay = maximum entry for this field
compute delay buffer for all field by subtracting it from the maximum one (the maximum one does not require a delay buffer since it is ready as the last one)
propagate the path length as own path length + internal delay from computation and delay buffers

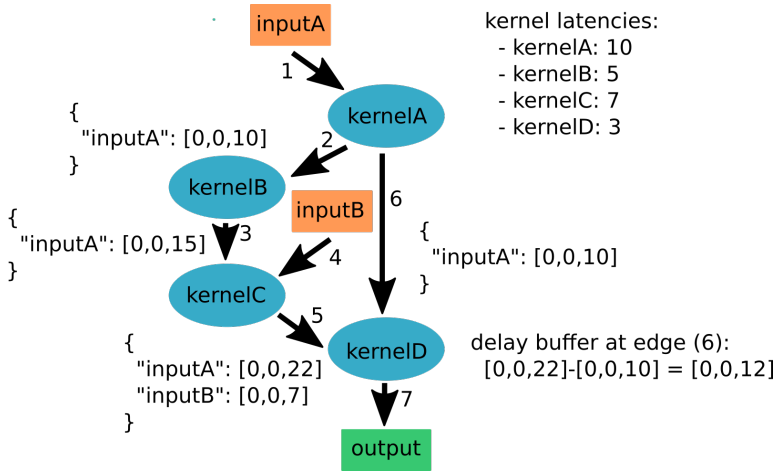


Figure 6.9: Example delay buffer computation using the path accesses list.

report The report summarizes the stencil program information and analysis and reports the following information:

- dimensions of data array (problem size)
- channels
- data field accesses
- internal buffer size
- internal buffer chunks
- delay buffer size
- path lengths (from the input to the node, per field)
- latency
- total buffer

- input kernel string
- relative access kernel string
- slow and fast memory buffer size

6.12 Simulator (simulator.py)

The Simulator class handles the simulation of our high level model FPGA functionality of the stencil chain design as described in the model section (figure 5.1).

simulate The simulate orchestrates the simulator and does the following:

Algorithm 9 simulate

```
initialize: read all data into input nodes
set global program counter PC=0
while not all input/kernel/output nodes done do
  step execution()
  PC = PC + 1
end while
finalize: write all output data to file
```

step_execution A single step of the step_execution represents a single cycle on an FPGA, split up into sub-steps, which are being called into the Input, Kernel and Output classes as follows:

Algorithm 10 step_execution

```
try_read all output and kernel nodes
try_execute all kernel nodes
try_write all input and kernel nodes
```

initialize The initialization method does the pre-processing before the actual simulation. It takes cares care of importing the input data.

finalize The finalization method does the post-processing after successful simulation. It writes the result data to a file and outputs the performance metrics if the log-level is at least 'basic'.

get_result The get_result functionality allows to extract the simulation result of the stencil program to be extracted.

all_done This method checks if all nodes have finished their execution and are idling. This is achieved by adding a local, node-internal program counter that increments whenever there has been a successful execution of an operation. For the input nodes it means: problem_size - largest queue = program counter or in other words, as soon as the last queue is empty, the program counter reached the value of dimX*dimY*dimZ. For the kernel nodes a increment of the program counter happens after a successful execution. A program counter increment for the output node happens after a successful data read.
As soon as all the program counter reached the value of the problem size, we can safely terminate. Since this is a sufficient and required property, this is an efficient implementation without doing any unnecessary idle steps.

diagnostics Diagnostics in the simulator is not only an exception handler, but is actually of great importance for the whole StencilFlow tool since most of the exceptions happen because of wrong input assumptions or wrong buffer size calculations.
Therefore, the diagnostics exception handler does not only raise the exception, but rather starts a procedure where the current state of all nodes (input, kernel and output) is being post-processed and logged. The diagnostics function of these nodes is being called and an output log including current program counter, internal state properties (e.g. for kernel: all_available, center_reached, etc) and channel/buffer content is displayed. Which helps for later analysis and finding the root cause of the error.

report If the log-level is at least 'basic', a report about the internal node state is generated after the simulation successfully completed, exactly as in the diagnostic method but just without an actual exception.

6.13 Optimizer (optimizer.py)

The Optimizer class is a key piece of StencilFlow and takes care of the optimal allocation and usage of the optimization parameters: fast memory, slow memory and bandwidth.

single_comm_volume This method returns the number of bytes necessary to copy a whole data array from or to the FPGA. This number actually corresponds to: dimX*dimY*dimZ * 4 bytes.

add_buffers_to_metric Creates the buffer data structure for optimization. It not only stores the information about the individual buffer, its communication volume and data type, but also maintains a doubly linked list to its predecessor and successor for updating the communication volume in case of a swap_out of a buffer.

```
{
    "queue": entry,
    "comm_vol": 2*self.single_comm_volume(
        self.kernels[kernel].data_type.bytes),
    "type": "internal",
    "datatype_size": self.kernels[kernel].data_type.bytes,
    "prev": prev,
    "next": None
}
```

max_metric This method finds the buffer entry in the metric_data list with the maximum metric value i.e. the buffer with the highest ratio: $\frac{buffersize}{communicationvolume}$ that is still located in fast memory.

ratio Computes the ratio between fast memory and the communication volume of the current optimization state. This is useful for the *optimize_to_ratio* optimization strategy.

update_comm_vol This method is being called for both neighbors if we moved a fast memory buffer to the slow memory. It updates the communication volume according to the formula derived in the model section according to the following rule:

```
How to determine the necessary communication volume?
(predecessor, successor):
case (fast, fast): 2*C
case (fast, slow): C
case (slow, fast): C
case (slow, slow): 0
where C:= communication volume to stream single data
    array in or out of fast memory (=SIZE(data array))
```

Note: pred of delay buffer is always fast memory since the stream from the predecessor kernel is being compute on the FPGA die next to the fast memory.

minimize_comm_vol This optimization strategy optimizes the problem to minimize the communication volume between fast and slow memory. In other words, it uses as little bandwidth as possible by a given amount of

fast and slow memory. This implies that this strategy makes maximal use of the fast memory. The pseudo code for this function is:

Algorithm 11 minimize_comm_vol(fast_memory_bound, slow_memory_bound)

```
mark all buffers to be allocated in fast memory
while metric_data list is not empty and fast_memory_use ¿ fast_memory_bound do
  swap max metric buffer to slow memory
  update neighbors
  remove buffer from metric_data
  update fast memory, slow memory and bandwidth use
  if check if slow_memory_use ¿ slow_memory_bound then
    Exception
  end if
end while
```

Note: The above exception occurs if the total buffer memory consumption is higher than sum of the slow memory and fast memory bound, which means that there cannot be found any solution for this optimization problem.

minimize_fast_mem This optimization strategy minimizes the usage of fast memory by a given amount of communication volume from/to slow memory. This implies that strategy makes maximal use of the available memory bandwidth and slow memory. The pseudo code for this function is:

Algorithm 12 minimize_fast_mem(communication_volume_bound, slow_memory_bound)

```
mark all buffers to be allocated in fast memory
while metric_data list is not empty and comm_volume_use ¿ comm_volume_bound and slow_memory_use ¡ slow_memory_bound do
  swap max metric buffer to slow memory
  update neighbors
  remove buffer from metric_data
  update fast memory, slow memory and bandwidth use
end while
```

optimize_to_ratio This optimization strategy optimizes for the given ratio value between the fast memory usage and the amount of available communication volume. The pseudo code for this function is given by:

Algorithm 13 optimize_ratio(ratio)

```
mark all buffers to be allocated in fast memory
while metric_data list is not empty and current_ratio ¿ ratio do
  swap max metric buffer to slow memory
  update neighbors
  remove buffer from metric_data
  recompute current_ratio
  update fast memory, slow memory and bandwidth use
end while
```

report The optimizer report gives an overview of the important metric values such as:

- swap status: which buffers are swapped out to slow memory and which are allocated in fast memory
- fast memory usage (for the strategy we optimized for)
- slow memory usage (for the strategy we optimized for)
- communication volume usage (for the strategy we optimized for)

6.14 SDFG Generator (sdfg_generator.py)

The Stateful DataFlow multiGraph (SDFG) Generator is the interface between the high-level representation in StencilFlow and the low-level and hardware design generator in DaCe [20]. This part has been implemented by my mentor Johannes de Fine Licht.

6.15 Run Program (run_program.py)

Run_program is a wrapper program that instantiates all required components and classes in order to run the input stencil program with the given input data arrays in software or hardware as specified by the arguments. Furthermore, we can simulate the design using StencilFlow and check if the output matches.

1. read program input file
2. create the kernel chain graph (including analysis of buffer sizes, latencies, etc)
3. execute the optimizer
4. execute the simuator and store the result
5. generate HLS OpenCl code from the high-level representation (using SDFG)

6. compile the code
7. read input data arrays
8. run the design in hardware (or in the Intel SDK Simulation Mode)
9. compare the result to the output from the simulator and check if they match

Note: In order to run the design on the actual hardware, we have to do the synthesis from the OpenCL code to the actual hardware bitstream. This is being done separately since we have to do this only once and since it takes many hours and should therefore not block the very limited number of nodes with hardware FPGAs installed.

6.16 Helper (helper.py)

The helper name space does not contain any specific classes, but rather very helpful and highly reused utility functions.
The most important ones are:
str_to_dtype In order to have a simpler interaction with DaCe, we use their data type class throughout the project. This method provides the functionality to convert generic strings of data type names (e.g. "float64") to DaCe data types.

parse_json Reads the input file from disk and parses the json as a python dictionary including some pre-processing (data type conversion) and sanity checks (e.g. file exist).

max_dict_entry_key Returns the key of the largest value entry of the input dictionary.

list_add_cwise Adds all items of two lists component-wise.

list_subtract_cwise Subtracts all items of two list component-wise.

dim_to_abs_val Computes the scalar number from the dimensional input using the global array dimensions (problem size). i.e. dim [X, Y, Z], size [a, b, c] $->$ 1*c + X*(b + Y*a) = [a, b, c] * transpose([Z*Y, Z, 1]).

load_array Loads arrays of various input formats.

save_array Stores numpy arrays to file.

load_input_arrays / save_output_arrays Basic functionality for loading actual data from file for transfer to the FPGA and store it back to the filing system.

arrays_are_equal Checks if two arrays are equal (up to some tolerance). This is used e.g. for verification of simulator output and the actual hardware output.

unique Removes duplicates from the given iterable.

6.17 Log-Level (log_level.py)

As the tool chain was growing and the combination of different functionalities accumulated it became apparent that we need a clever scheme for managing the verbosity of the output console log messages. We decided to implement this using log levels as follows:

- level 0: no logging
- level 1: basic logging (output what the program is doing)
- level 2: moderate logging (output what the program is doing and basic information about the actual content it is processing)
- level 3: full logging (output what the program is doing and full information disclosure, including e.g. program counter and buffer content of every stencil program during simulation)

Each lower log level is a subset of the higher log level. It is usually helpful to get a good trade-off between these levels depending on the purpose, since e.g. simulation performance suffers a lot under log-level 3 performance-wise.

6.18 Testing (testing.py)

As the project progressed, software complexity increased significantly which makes it prone to bugs, especially while refactoring. Furthermore, the integration of DaCe and the parallel work of my mentor Johannes de Fine Licht and me on related software made it necessary to ensure proper code functionality in the master branch. Therefore and because we want to achieve good software practices, we decided to do extensive testing for the low level functionality (e.g. bounded queue) and basic function testing for the remaining parts. This way, we can ensure that changes did not break anything before actually checking them in potentially preventing someone from being able to work.

6.19 Dynamical Core Estimate (dycore_estimate.py)

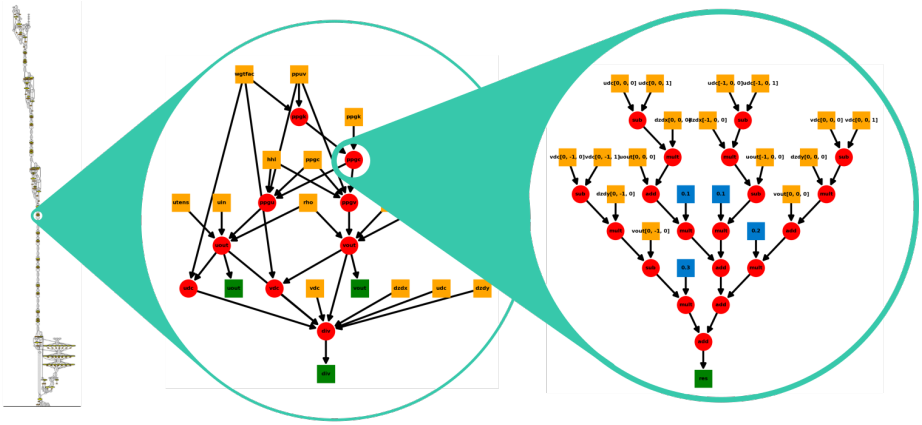
The dynamical core estimate program is not part of StencilFlow used for internal functionality, but has rather been used to do a half-automatic walk-

through of computing the required numbers for the estimate. It instantiates the KernelChainGraph for the three stencil programs (fastwaves, advection and diffusion) we used in order to compute the mean as well as for getting the delay buffer estimate for the full dynamical core.

Chapter 7

Feasibility of Full COSMO Weather Forecast Model on an FPGA Cluster

Since we are unaware of the existance any usage of re-programmable hardware in large-scale simulations in practice at the time of writing, from the first day on we have been pursuing the legitimate question of feasibility. This section is dedicated to an early-on general resource usage estimation for COSMO and check if these constraints are satisfied by the, at the time of writing, state of the art FPGA, the Intel Stratix 10 FPGA. Since the complexity arises from the high number of abstraction layers imposed it is impractical to reason about the actual feasibility manually. That's why we started with the implementations and did the first feasibility study as soon as we progressed far enough with the StencilFlow framework to get substantial work done automatically. The following figure gives you an idea of the complexity and size of the COSMO dynamical core model.



Dynamic Core Stencil Program Single Kernel

Figure 7.1: The levels of abstraction: From the Dynamical Core to Stencil Programs and Kernels.

7.1 Estimation

Assumptions Before we start with the derivation we want to summarize all the assumptions we made and justify their validity.

Data Type According to discussions with experts from MeteoSwiss, most of the modeled equations converge by using floating point data types of 32 or 64 bit precision. Therefore we will assume that 32bit floating point precision (IEEE 754) is reasonable. The StencilFlow framework incorporates the functionality to choose a per-kernel data type, which would allow to choose the necessary precision on a per-equation basis.

Grid Size The current highest resolution, in-production variant has about 1158x774 horizontal and 80 vertical grid points (COSMO-1, COSMO-E: 582x390x60, COSMO-7: 393x338x60) [11]. Together with experts from MeteoSwiss, we decided that grid sizes of 1024x1024x64, which is in the order of their current highest-resolution (1.1km) variant, would be a good choice.

Iteration Space Order The dimension of the finite grid is not equally spaced in all three dimensions. The delay and internal buffer usage greatly depends on the order of iteration direction over the grid. Since there are no cycles in the data flow we assume that we can iterate over whichever direction is optimal. This can be achieved technically by either stream the transposed input data arrays to the FPGA or do it on the fly in a preprocessor transpose kernel. This modification saves us factor $\frac{1024}{64} = 16$ of buffer space.

```
for (i = 0; i < X; i++)
  for(j = 0; j < Y; j++)
    for(k = 0; k < Z; k++)
      out[i,j,k] = f(S(in[i,j,k]))
```

```
for (i = 0; i < X; i++)
  for(j = 0; j < Y; j++)
    for(k = 0; k < Z; k++)
      out[i,k,j] = f(S(in[i,k,j]))
```

Change the iteration direction order from X,Y,Z to X,Z,Y.

Dynamical Core To model the size, shape, critical path and delay buffer dependencies of the dynamical core of the weather forecast model, we used the high-level representation of the dependency graph (Runge Kutta time integration scheme). These values have been derived manually (total number of kernel programs) and partially automatically (critical path length, delay buffer size) by modeling the stencil programs as very simplistic stencils in the shape of the dependency graph. This lead to the following metrics,

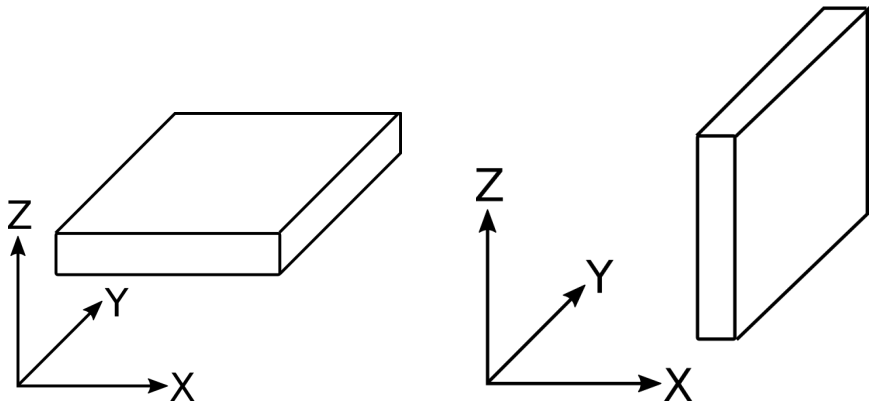


Figure 7.2: Transposition of the input array to optimize buffer size.

which we can later use with the average stencil program values to extrapolate to the full size.

- Critical path length (in kernel programs): 252
- Total number of kernel programs: 131
- Delay buffer: 184 * buffer size of an average stencil program

Average and Total Critical Path Length The critical path length is the longest path from any input to the final output latency-wise.

- fastwaves: [3, 1, 54]
- diffusion: [3, 0, 24]
- advection: [7, 4, 26]
- mean: $mean([[3,1,54]],[3,0,24],[7,4,26]) = [\lceil\frac{3+3+7}{3}\rceil,\lceil\frac{1+0+4}{3}\rceil,\lceil\frac{54+24+26}{3}\rceil] = [5,2,35]$
- total critical path length = dycore critical path length * mean path length $= 252 * [5,2,35] = [1260,504,8820]$

Average and Total Buffer Size In order to estimate the buffer space of the whole model, we manually converted three common stencil programs of COSMO, namely fastwaves, advection and diffusion, to the StencilFlow input format and let the tool analyze their buffer usage. We use the average of them for extrapolation.

- fastwaves:
  - internal buffer: [8, 7, 6]
  - delay buffer: [32, -12, 289]

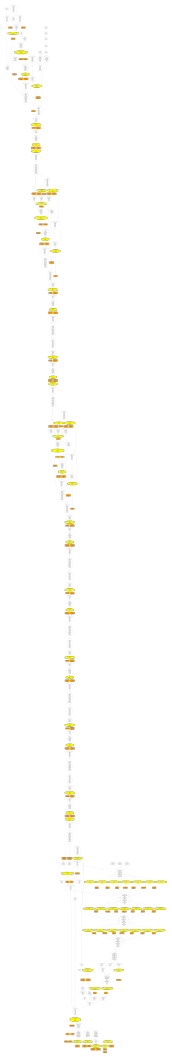


Figure 7.3: Overview over the stencil program of the whole dynamical core of COSMO. *MeteoSwiss*

  - combined: [40, -5, 295]
- diffusion:
  - internal buffer: [16, 13, 0]
  - delay buffer: [20, 8, 192]
  - combined: [36, 21, 192]
- advection:
  - internal buffer: [8, 8, 0]
  - delay buffer: [10, 2, 57]

  - combined: [18, 10, 57]

- mean delay buffer: $mean([32, -12, 289], [20, 8, 192], [10, 2, 57]) = [\lceil \frac{32+20+10}{3} \rceil, \lceil \frac{-12+8+2}{3} \rceil, \lceil \frac{289+192+57}{3} \rceil] = [21, 0, 180]$
- mean combined buffer: $mean([40, -5, 295], [36, 21, 192], [18, 10, 57]) = [\lceil \frac{40+36+18}{3} \rceil, \lceil \frac{-5+21+10}{3} \rceil, \lceil \frac{295+192+57}{3} \rceil] = [32, 9, 182]$
- total delay buffer = dycore delay buffer estimate (unit: size of average stencil program buffer size) * average combined buffer size = $184 * [32, 9, 182] = [5888, 1656, 33488]$

Average i-th Largest Buffer For the sake of simplicity, we use the natural heuristic optimization strategy for the manual estimate. First, we assume all buffers to be allocated in the fast memory of the FPGA, and we swap out the largest buffer of each stencil program to slow memory till we reach the memory bandwidth bottleneck. In order to achieve this, we manually looked at our three stencils and calculated their buffer sizes in Megabytes for the largest 10 buffers and computed the mean. The values are as follow: $[\approx 3.7MB, \approx 0.8MB, \approx 0.6MB, \approx 0.4MB, \approx 0.4MB, \approx 0.4MB, \approx 0.3MB, \approx 0.2MB, \approx 0.2MB]$. This gives us the following fast memory savings per stencil program, if we stream the data to slow memory (swap out).

- 1 buffer evicted: $\approx 3.7MB$
- 2 buffers evicted: $\approx 4.5MB$
- 3 buffers evicted: $\approx 5.1MB$
- 4 buffers evicted: $\approx 5.5MB$
- 5 buffers evicted: $\approx 5.9MB$
- 6 buffers evicted: $\approx 6.3MB$
- 7 buffers evicted: $\approx 6.6MB$
- 8 buffers evicted: $\approx 6.8MB$
- 9 buffers evicted: $\approx 7.1MB$
- 10 buffers evicted: $\approx 7.3MB$

Remark: Depending on the actual hardwares bandwidth capacity, we might go a lot higher than swapping out 10 buffers evicted per stencil program, but it is infeasible to go ahead manually, since the complexity of extraction increases.

Clock Frequency In contrast to modern CPU and GPU architectures, the FPGA frequency is flexible per design (in a range of tens or hundreds of Megahertz), but fixed after the synthesis has been finished. Expertise from previous projects have shown that 200-300Mhz is a reasonable assumption that should be achievable with todays FPGA devices [16].

Pipeline Theory We assume to deal with a ideal pipeline, which means we will have a saturation phase a full utilization phase and a draining phase, where we are able to produce one output per clock cycle in the latter two phases.

Derivation We will split the derivation into two subproblems. First we want to find out how many cycles it takes from the feed of the first data element till the last result exits the pipeline. This gives us an upper bound on the communication volume available. In a second step, we want to find out how much communication volume is required to move all data elements to respectively from the FPGA.

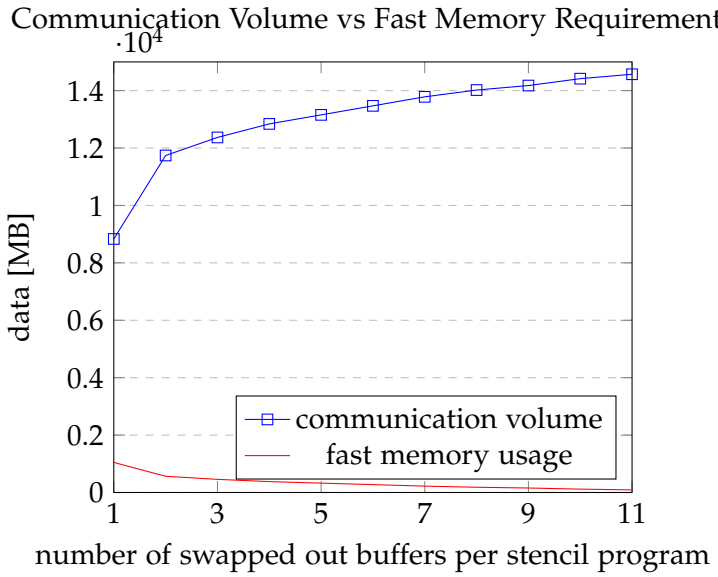
Latency The critical path length of the dynamical core is 252 stencil programs. Since every stencil program itself has an average latency of [5, 2, 35], we end up with a total latency of $252 * [5, 2, 35] = [1260, 504, 8820]$
The grid is 1024x1024x64 (longitude x latitude x altitude) in size. We would like to optimize the total buffer usage by reordering the dimensions to [64, 1024, 1024] = [X,Y,Z]. This leads to the absolute latency value (in cycles) of: $(8820 * 1 + 504 * X + 1260 * X * Y) = (8820 + 504 * 64 + 1260 * 64 * 1024) =$ 82616436 cycles.
Under the assumption that we can produce one result every cycle after the pipeline is saturated, we et the following latency:

- total run time (cycles): $latency + X * Y * Z = 82616436 + 64 * 1024 * 1024 = 149725300$ cycles
- total run time (seconds): $\frac{149725300 cycles}{200 MHz} = 0.7486265$ seconds

Communication Volume We will derive the total communication volume for the scenario of swapping out all delay buffers to slow memory since they are a lot larger in the analyzed designs (*Remark: The actual size might still be smaller, since almost every field access imposes internal buffer, but only very few do impose delay buffers.*) compared to the internal buffers. Beside that, we move the 10 largest buffers of each stencil program from fast memory to slow memory while residing the remaining buffer allocations in fast on-chip memory.
From the dycore delay buffer estimate (184) and the average stencil program buffer size, we derive the total amount of delay buffer: $184 * [32, 9, 182] = [5888, 1656, 33488]$ in dimensional form. Using the transposed input dimensions and the float32 assumption, we get: $184 * 4 bytes * [32, 9, 182] = 4 bytes * 184 * (182 + 64 * 9 + 32 * 64 * 1024) = 184 * 8391640 bytes \approx 184 * 8.39 MB = 1544061760 bytes \approx 1544 MB$. Since we have to move that data from the fast memory to the slow memory and back, we have to account twice for the communication volume: $2 * 1544 MB = 3088 MB$

Buffering the 10 largest channels in slow memory gives us an approximate saving of 7.3MB per stencil program, which is half of the total communication volume (we have to bring the data elements out and in again), which leads to: $2 * 131 * 7.3MB = 1912.6MB$
This gives us a total communication volume: $3088MB + 1912.6MB = 5000.6MB$
The next step will be now to look at the specifications of a state of the art FPGA to find out if these numbers are feasible.

Fast and Slow Memory We finally need to derive the fast and slow memory requirements imposed by the above optimization strategy.
The slow memory size is equal to the combination of all the delay buffers and the 10 largest internal buffers: $(131 + 184) * 8.39MB = 2642.85MB$.
The fast memory contains all the internal buffers that are smaller then the 10 largest ones: $(8.39MB - 7.3MB) * 131 = 142.79MB$



7.2 Case Study: Intel Stratix 10 FPGA

This section is dedicated to the important technical aspects necessary to check if the above constraints can be fulfilled by a state of the art FPGA. We have chosen the Intel Stratix 10 FGPA since it is one of the most advanced re-programmable hardware on the market at the time of writing and we do have lots of such devices available at CSCS and the University of Paderborn. Nevertheless, this walk through is representative for other devices too.

Parameters The key metrics for our optimization objective are:

- fast on-chip memory: 25MB
- slow dram memory: 64GB
- memory bandwidth between slow and fast memory: 86.4GB/s
- clock frequency: theory: up to 1GHz, observed in synthesized stencil programs: 250-350Mhz

The reason we rely only on these very few parameters for estimating the feasibility is that we are following the basic assumption that our stencil programs are heavily memory bound. In addition to that, the DSP blocks on these devices offer a huge amount of parallel available operation processing power that we do not think this will constrain us. The Stratix 10 can deliver up to 10^12 floating point operations per second.

Bandwidth A single run of the program takes 0.749 seconds, which allows us at a memory bandwidth rate of 86.4GB/s to transfer $\frac{86.4GB}{s} * 0.749s = 64.7136GB$ of data. Since we only need 5000.6MB of communication volume, we are within the constraints, even with a high level of communication overhead.

Slow Memory The derivation has shown that we need 2642.85MB of slow memory to allocate all the swapped out buffers. Since we have 64GB available, we are perfectly within the constraints.

Fast Memory The derivation has shown that we need 142.79MB of fast memory if we use the naive optimization scheme from above. This would mean that we need $\lceil \frac{142.79MB}{25MB} \rceil = 6$ FPGA devices to fit the whole design. *Remark: The high excess of memory bandwidth and slow memory would suggest to further swap out buffers, which would potentially make the design fit on fewer or even a single FPGA device. This will be part of the automatic optimization framework to find a good balance between these constraints.*

7.3 Conclusion

We could show that even with a naive optimization strategy, we should be able to fit the whole dynamical core of the COSMO weather prediction model onto six Intel Stratix 10 FPGA devices. Further optimization might significantly reduce this number, since there is lot of headroom for the bandwidth and slow memory utilization. In addition to that, multi FPGA designs are not only possible in theory, but are rather something we consider to achieve in the future either for fitting larger problems or to accelerate the current design. Especially the shape with large depth and small width of the COSMO weather models looks promising for a simple splitting of design

without imposing a lot of data streams between the devices. The collaboration with the University of Paderborn and their configuration of 32 Stratix 10 FPGAs connected through 40Gbps direct optical fiber links will help us to achieve our goal [1].

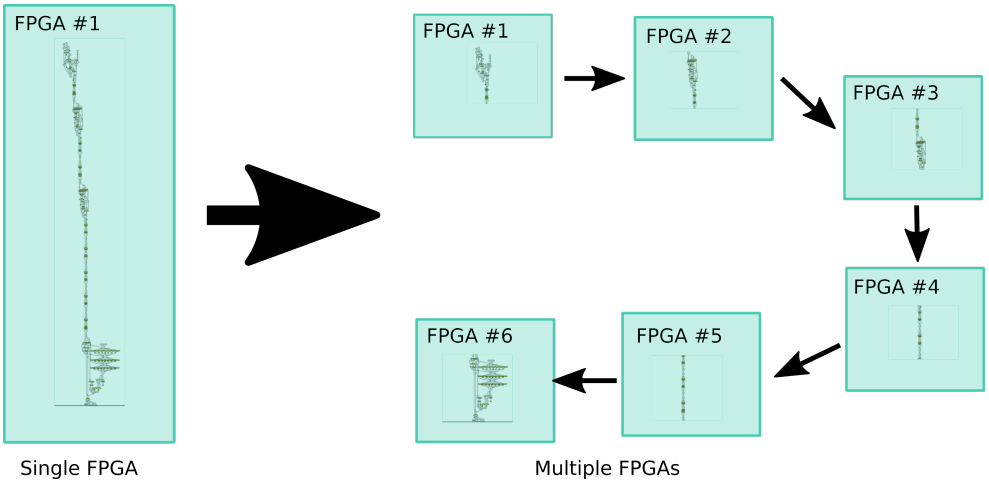


Figure 7.4: Split the problem into equal sized subproblems which fit on instanced of the reprogrammable device. Stream the required data fields between them.

Chapter 8

Evaluation of StencilFlow in Hardware

We evaluate the performance of StencilFlow on two sets of stencil programs. The first set consists of simple and highly regular kernels, which makes it easy to reason about the measurement and scaling while the second set consists of results from stencil programs of the actual dynamical core.

8.1 Experimental Setup

We performed the evaluation on ault12 consisting of an Intel E5-2630 v4 @ 2.20GHz host CPU with 64GB of RAM and a Nallatech 520N (Intel Stratix 10 GX 2800) FPGA compute card interconnected via PCI-Express.
The key specifications of the Nallatech 520N device illustrated in figure 8.1 are:

- Fast Memory: 25MBytes
  - 11721 M20K (20Kbit) blocks
  - 23796 MLAB (640bit) blocks
- Host Interface
  - 16-lane PCI-Express Gen 3.0
- DDR4 SDRAM Memory
  - Four banks of DDR4 SDRAM x 72 bits
  - 8GB per bank (32GB total)
  - Transfer Rate: 2400 MT/s

[1]

We measured all results 20 times and report the median.

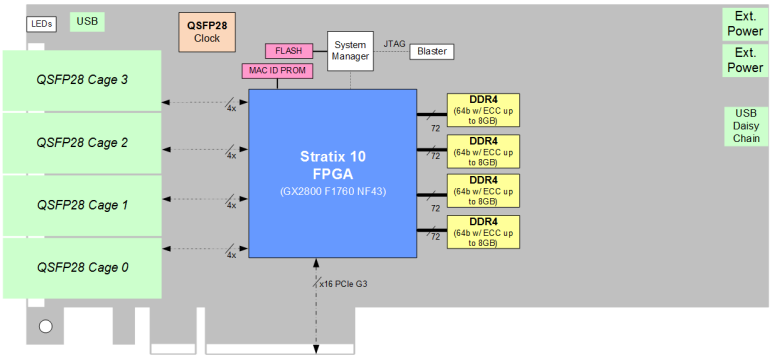


Figure 8.1: Intel Stratix 10 520N Block Diagram, *bittware.com*

8.2 Metrics

The goal of the evaluation at this point is to find out if our abstracted high-level model of the FPGA corresponds to the actual hardware in different aspects.

8.2.1 Resource Report

As part of the evaluation, we compare the theoretical usage findings with the values provided in the Intel OpenCL FPGA HLS Report after synthesis of the design.

Single vs K Iteration In order to better understand the the actual scaling of the design, we constructed the Jacobian examples that can easily be scaled in two directions:

1. Problem Size: Scale up the input problem size

```
input: a[1..N]
res = 0.25 * (a[j-1,k] + a[j+1,k] + a[j,k-1] + a[j,k+1])
```

2. Depth: Add many identical stencils after each other. e.g. for K=3:

```
input: a[1..N]
b[i,j] = 0.25 * (a[j-1,k] + a[j+1,k] + a[j,k-1] + a[j,k+1])
c[i,j] = 0.25 * (b[j-1,k] + b[j+1,k] + b[j,k-1] + b[j,k+1])
res = 0.25 * (c[j-1,k] + c[j+1,k] + c[j,k-1] + c[j,k+1])
```

Frequency We expect the design frequency to be almost constant throughout the different designs, but since we are not able to directly influence this value, we will add the frequency figures without interpretation if there are no specific outliers.

LUT The number of look-up tables approximately stays constant while scaling up the problem size. We do not model this resource parameter because we assume that LUTs are not the restricting factor.

FF The interpretation of the synthesis report suggests that the number of flip-flops required to build the design is almost constant for a specific design, even when we scale the problem size. We assume that this resource is not the restricting factor.

MLAB The number of memory logic array blocks stays constant for a specific design even when we scale the problem size. In addition, it scales linearly with the depth of the problem. We assume that this resource will not limit our design.

RAM The random access memory resource is the primary source of fast memory on the device, known as M20K blocks in the Stratix 10 architecture. Since the current implementation of the SDFG generator does not support the actual swap-out of buffers to slow memory, we expect the following relation to hold true:

$$\text{total buffer} = \text{internal buffer} + \text{delay buffer} = \text{RAM} \tag{8.1}$$

This means that the RAM requirements should scale as follows:

$$\text{buffer requirements in dimensional form} = [a, b, c] \tag{8.2}$$

$$\text{problem size: } [X,Y,Z] \tag{8.3}$$

$$x + X * b + X * Y * a = c + X * (b + Y * a) \tag{8.4}$$

For the K-Iteration scenario, the buffer requirements are:

$$\text{buffer of K iterations} = K * \text{single iteration} \tag{8.5}$$

DSP The digital signal processor blocks are processing the actual computation. Since our design is fully pipelined without any parallel execution pipelines, we expect the number of DSP units to stay equal even if we scale the problem dimensions.
In the scenario of K-Iterations, we add K such pipelines after each other. Therefore, we expect the number of DSP units to scale linear in K.

8.2.2 Execution Time

In addition to the resource usage, we are interested in how the execution time scales again compared to the problem size and the depth of the problem.

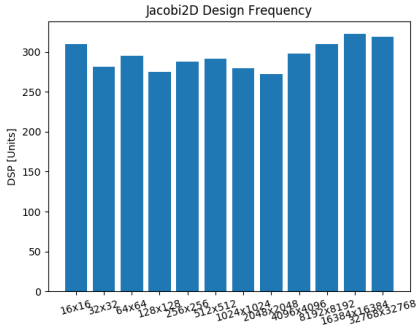


Figure 8.2: Frequency of the high-level synthesized design.

Single Iteration In general, the execution time of a pipeline expressed in cycles is given by

$$\text{execution time} = D + (N - 1) * I \tag{8.6}$$

where D denotes the depth of the pipeline (latency), I is the number of cycles required to produce an output and N is the actual problem size.

K-Iteration The only difference of the K-iteration design is that the depth of the complete pipeline is $K * D$ instead of D. Therefore, the formula looks like:

$$\text{execution time} = K * D + (N - 1) * I \tag{8.7}$$

8.3 Simple Stencils

8.3.1 Jacobi2D

The first stencil program we analyze consists of a single two-dimensional Jacobian kernel.

$$res = 0.25 * (a[j - 1, k] + a[j + 1, k] + a[j, k - 1] + a[j, k + 1]) \tag{8.8}$$

Frequency The design frequency of all problem sizes are between 270Mhz and 325Mhz (figure 8.2) as we expected.

DSP The amount of DSP units is constant 3, which is the number of operations (3 additions and 1 multiplication) of our Jacobian stencil if we assume that one DSP is a fused Add-Multiply unit.

RAM The buffer requirements of Jacobi2D in dimensional form consists of the internal buffer of the kernel only, since a single kernel cannot impose any delay buffers. This is given by max index − min index + 1 = [0, 2, 3] which leads to a buffer requirement of

$$3 + X * 2 \tag{8.9}$$

This trend of a factor of two by increasing the X dimension by a factor of two too can be seen well for larger problem sizes in the plot 8.3

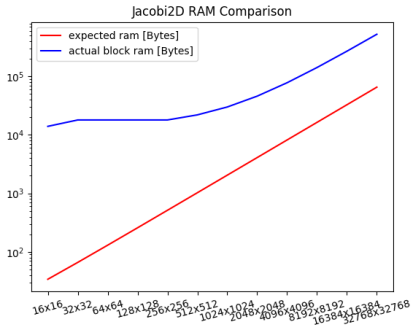


Figure 8.3: Expected buffer size vs block ram usage.

Execution Time In general, the execution time of a pipeline expressed in cycles is given by

$$\text{execution time} = D + (N - 1) * I \tag{8.10}$$

By assuming $I = 1$ (producing a result every cycle) and using D as the latency computed by StencilFlow which, our estimate is almost perfectly equal to the measurements, especially for larger problem sizes where we can measure more exact.

8.3.2 Jacobi3D

The second stencil program we analyze consists of a single three-dimensional Jacobian kernel.

$$res = \frac{1}{6} * (a[i-1,j,k] + a[i+1,j,k] + a[i,j-1,k] + a[i,j+1,k] \\ + a[i,j,k-1] + a[i,j,k+1]) \tag{8.11}$$

Frequency The design frequency of all problem sizes and iteration depths are between 260Mhz and 330Mhz as we expected.

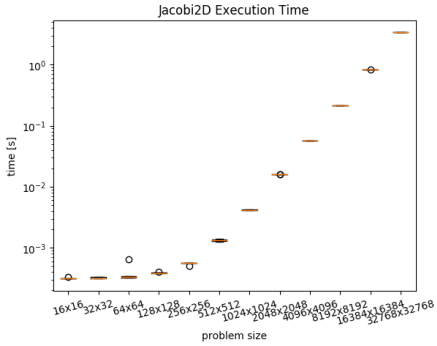


Figure 8.4: Execution time of Jacobi2D.

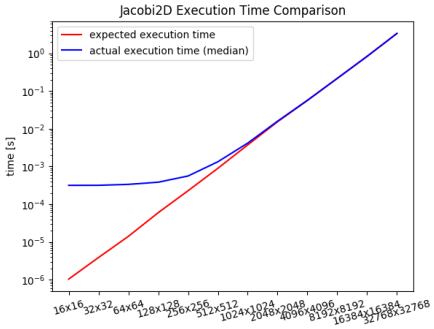


Figure 8.5: Expected vs actual execution time.

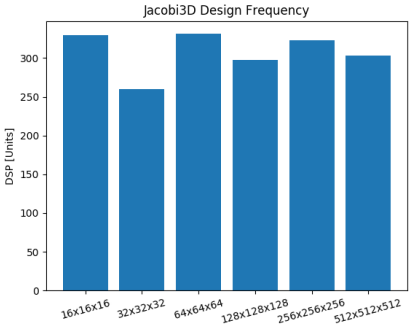


Figure 8.6: Frequency of the high-level synthesized design.

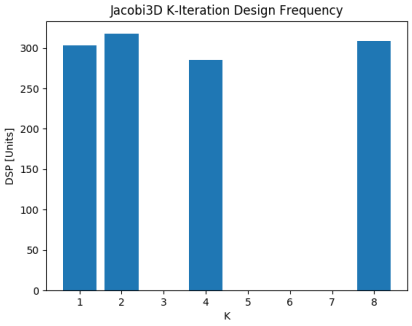


Figure 8.7: Frequency of the high-level synthesized design.

DSP The amount of DSP units is constant six, which is exactly the number of operations (5 additions and 1 multiplication) of our Jacobian stencil. Furthermore, the value is problem size independent as expected.

On the other hand, for the K-Iteration case, the number of DSP units increases perfectly linear with K i.e. $K * 6$ as shown in figure XY.

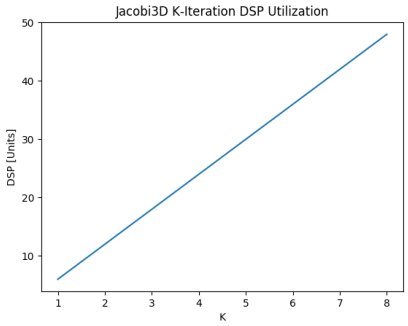


Figure 8.8: DSP utilization of k iterations.

RAM The buffer requirements of Jacobi2D in dimensional form consists of the internal buffer of the kernel only, since a single kernel cannot impose any delay buffers. This is given by max index − min index + 1 = [2, 0, 1] which leads to a buffer requirement of

$$1 + X * Y * 2 \tag{8.12}$$

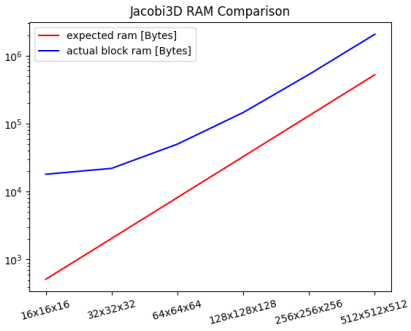


Figure 8.9: Expected buffer size vs block ram usage.

For the K-Iteration case, the RAM amount should grow linear in K by the number of a single Jacobi3D kernel. This is the case as you can see in figure XY.

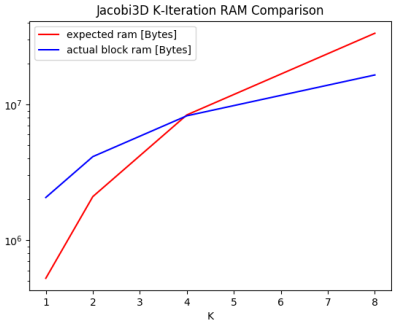


Figure 8.10: Expected buffer size vs block ram usage.

Execution Time In general, the execution time of a pipeline expressed in cycles is given by

$$\text{execution time} = D + (N - 1) * I \tag{8.13}$$

By assuming $I = 1$ (producing a result every cycle) and using D as the latency computed by StencilFlow which , our estimate is almost perfectly

equal to the measurements, especially for larger problem sizes where we can measure more exact.

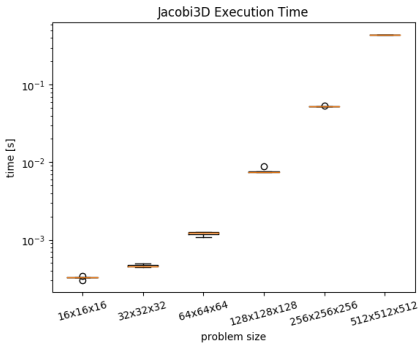


Figure 8.11: Execution time of Jacobi3D.

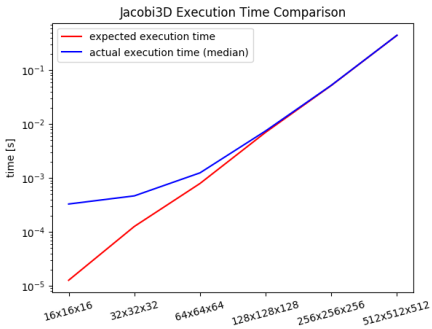


Figure 8.12: Expected vs actual execution time.

The same applies for the K-Iteration case with the modified formula:

$$\text{execution time} = K * D + (N - 1) * I \tag{8.14}$$

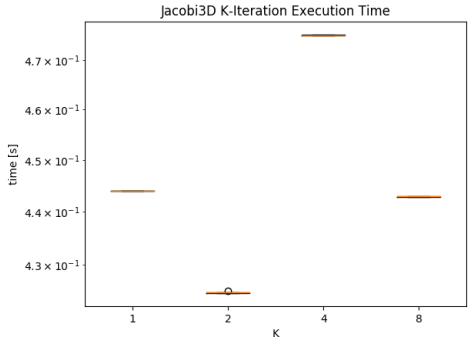


Figure 8.13: Execution time of K-Iteration Jacobi3D.

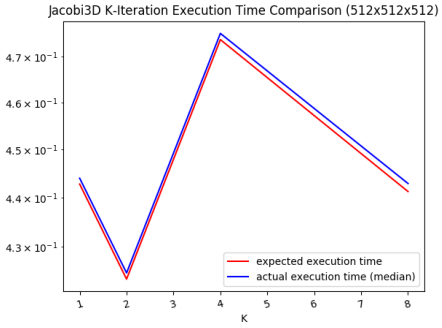


Figure 8.14: Expected vs actual execution time.

Note: The spikes in figure 8.13 and 8.14 are occurring because of the different HLS design frequencies which accounts much higher compared to the little additional depth the iterations add.

8.4 COSMO Stencil Chains

8.4.1 Fastwaves

The fastwaves stencil chain is part of the dynamical core of COSMO. Due to non-disclosure agreement restrictions, we cannot publish the kernel expres-

sions, but we will use the output of StencilFlow for comparison with the measurements from the report and timings from the actual executions.

Problem	Expected Buffer Size [MB]	Expected Latency [cycles]
fastwaves_16x16x16	51543	1438
fastwaves_32x32x32	169767	3742
fastwaves_64x64x64	641223	12958
fastwaves_128x128x128	2524167	49822

Figure 8.15: Expected Buffer and Latency from StencilFlow.

Frequency The design frequency of all problem sizes are between 220Mhz and 260Mhz as we expected. Since these stencil programs have a higher degree of complexity compared to the Jacobi examples, we also expected the frequency to be a bit lower.

DSP The reported number of DSP units is 66 throughout all problem sizes of fast waves, which is what we expected. Since there are 44 additions/-subtractions and 28 multiplications/divisions in the kernels of this stencil program, this number is again lower than the sum of 44 and 28. Since the single precision DSP blocks incorporate an add and a multiply unit and shown in figure 8.16, they might be used concurrently.

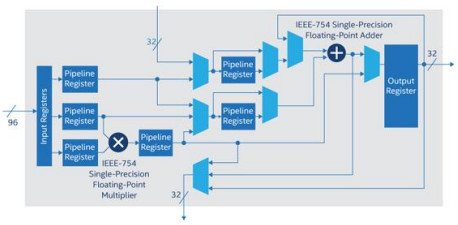


Figure 8.16: Single-Precision Floating Point DSP Block *intel.com*

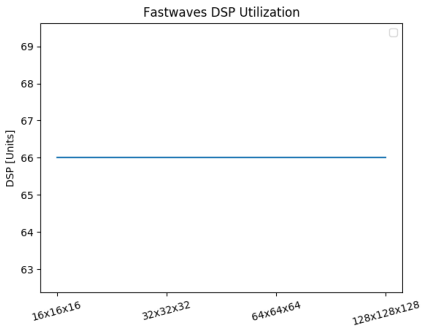


Figure 8.17: DSP utilization.

RAM This design imposes internal and delay buffers of various sizes, which is automatically handled and computed by StencilFlow. We can therefore directly compare the theoretical value to the number we are getting in the report.

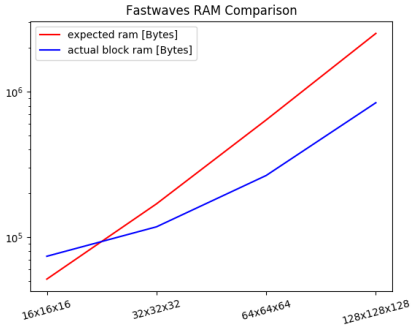


Figure 8.18: Expected buffer size vs block ram usage.

Execution Time In general, the execution time of a pipeline expressed in cycles is given by

$$\text{execution time} = D + (N - 1) * I \tag{8.15}$$

By assuming $I = 1$ (producing a result every cycle, shown as approximative value in the report) and using D as the latency computed by StencilFlow, our estimate is about 20-30% lower.

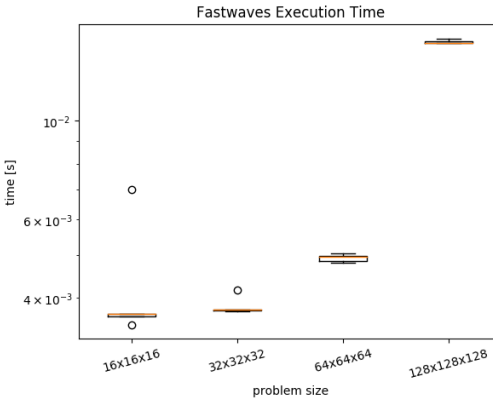


Figure 8.19: Execution time of Fastwaves.

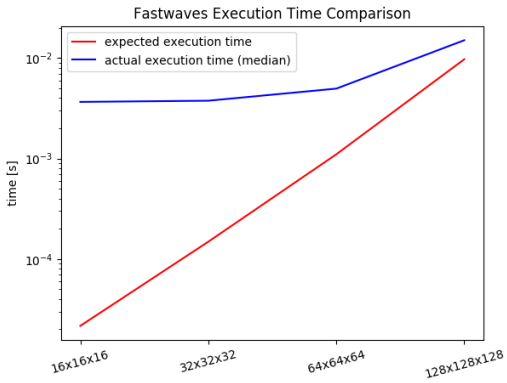


Figure 8.20: Expected vs actual execution time.

8.4.2 Diffusion

The diffusion stencil chain is part of the dynamical core of COSMO. We will use the output of StencilFlow for comparison with the measurements from the report and timings from the actual executions.

Frequency The design frequency of all problem sizes are between 260Mhz and 270Mhz as we expected. Since these stencil programs have a higher

Problem	Expected Buffer Size [MB]	Expected Latency [cycles]
diffusion_16x16x16	189696	758
diffusion_32x32x32	683264	2294
diffusion_64x64x64	2653440	8438
diffusion_128x128x128	10525952	33014

Figure 8.21: Expected Buffer and Latency from StencilFlow.

degree of complexity compared to the Jacobi examples, we also expected the frequency to be a bit lower.

DSP The reported number of DSP units is 56 throughout all problem sizes of diffusion. Since there are 50 additions/subtractions and 17 multiplications/divisions and 9 ternary operator in the kernels of this stencil program, this number is again lower than the sum of 50 and 17. Since the single precision DSP blocks incorporate an add and a multiply unit and shown in figure 8.16, we conclude that they might be in concurrent use.

RAM This design imposes internal and delay buffers of various sizes, which is automatically handled and computed by StencilFlow. We can therefore directly compare the theoretical value to the number we are getting in the report.

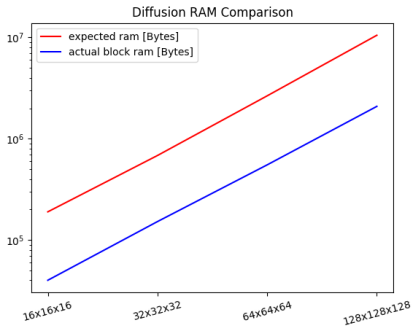


Figure 8.22: Expected buffer size vs block ram usage.

Execution Time By assuming $I = 1$ (producing a result every cycle, shown as approximative value in the report) and using D as the latency computed by StencilFlow, our estimate is about 20-30% lower.

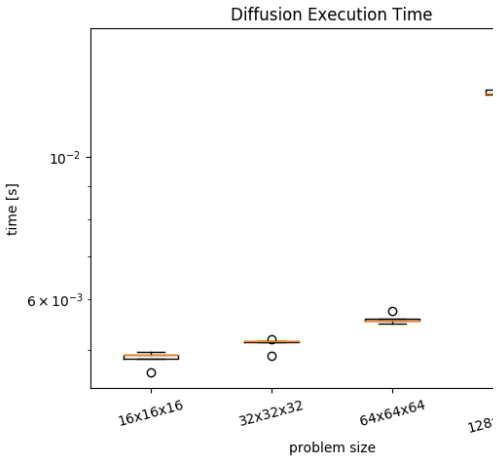


Figure 8.23: Execution time of diffusion.

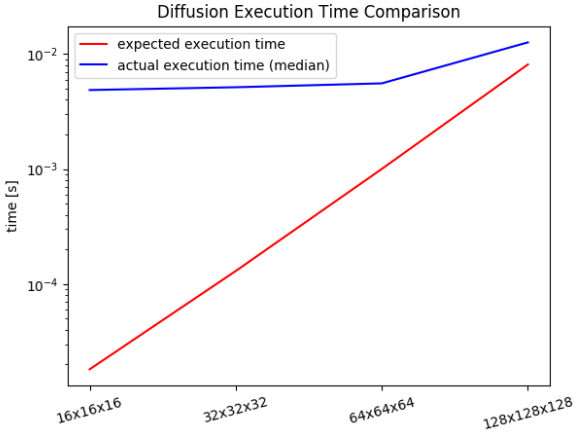


Figure 8.24: Expected vs actual execution time.

8.5 Conclusion

Most of our theoretical findings correlate very well with the measurements and reports from the real hardware. This builds a good foundation for further investigation into deeper analysis and optimization for the specific platforms.

Chapter 9

Future Work

The StencilFlow framework and the theoretical findings of this Bachelor thesis provide the foundation for future work. We seek to further develop this toolbox with a strong focus on performance optimization and the evaluation of a real-world sized problem on the re-programmable hardware architecture.

9.1 Generalization and Optimization for Different Domains

Despite that our motivation for the development of StencilFlow was the COSMO weather forecasting model, we generalized the problem definition in order to be valid for general stencil programs on structured grids. This gives us the opportunity to look into further fields of application. This could give researchers and software developers the opportunity to exploit the benefits of FPGAs as a high level tool while having the implementation details and optimization tweak abstracted.

9.2 Implementation of the Full Dynamical Core

The result of the feasibility study gave us confidence to put this idea into action. The ongoing collaboration with MeteoSwiss enables us to port the full dynamical core of the COSMO weather forecast model gives us the chance to see if FPGAs are part of the next generation of compute accelerator in the high performance computing sector.

9.3 Hardware Optimization

Beside optimizing for the theoretical objective of fast/slow memory and bandwidth usage, studies [38, 25, 32, 31, 21] have shown that optimization of high level synthesis code is crucial for maximal performance. By using the knowledge of field experts from ETH Zurich and the University of Paderborn, we seek to improve the code generator to get a higher yield of the available compute resources.

9.4 FPGA Performance

The rapid advance in on-chip resources such as memory bandwidth, floating point IPs, etc justifies a tool that can automatically adapt and generate efficient code for the new constraints. Furthermore, the current technology allows us to use the classical high performance scaling approach to not only increase the compute capacity locally, but to split the work by using multiple devices. The usage of the Noctua cluster at the University of Paderborn, equipped with 32 Intel Stratix 10 FPGAs [13] with four integrated 40Gbit/s transceivers connected through an fiber optical switch with configurable topology, enables us to find an optimal strategy of dividing the design for optimum area usage while keeping the communication overhead minimal.

Appendix A

Appendix

Appendix A

The approximation is based on the 45nm technology described in *Computing's Energy Problem (and what we can do about it)* [27].
We used the following assumptions:

- Scenario
  - 1000 iterations over 200 data elements by applying floating-point multiplication
  - data has to be loaded from DRAM first and is later fetched from the first-level cache (CPU, 95% cache hit rate) or fast on-chip memory (FPGA)
- Energy partitioning
  - Floating-Point Multiplication: 4pJ
  - Memory:
    - DRAM access: 2nJ
    - Cache (8KB): 10pJ
  - Control overhead:
    - CPU (instruction decoding): 30pJ
    - FPGA (only pipeline-stall logic, assumption): 5pJ

Calculation:
CPU energy consumption: $200 \cdot (2 \cdot 2nJ + 4pJ + 30pJ + 10pJ + 999 \cdot (0.95 \cdot 3 \cdot 10pJ + 0.05 \cdot (2 \cdot 2nJ + 10pJ) + 4pJ + 30pJ)) \approx 53356nJ$
FPGA energy consumption: $200 \cdot (2nJ + 4pJ + 5pJ + 10pJ + 999 \cdot (10pJ + 4pJ + 5pJ + 10pJ)) \approx 6198nJ$
Energy efficiency of FPGA over CPU: $\frac{53356nJ}{6198nJ} \approx 8.6$

Bibliography

[1] Bittware Stratix 10 520N. https://www.bittware.com/fpga/520n/, last accessed on 12/08/19.

[2] Consortium for Small-scale Modeling. www.cosmo-model.org, last accessed on 02/08/19.

[3] COSMO, Consortium. http://www.cosmo-model.org/content/consortium/default.htm, last accessed on 12/08/19.

[4] CSCS. https://en.wikipedia.org/wiki/Swiss_National_Supercomputing_Centre, last accessed on 16/08/19.

[5] CSCS. https://en.wikipedia.org/wiki/Swiss_National_Supercomputing_Centre, , last accessed on 02/08/19.

[6] Federal Office of Meteorology and Climatology MeteoSwiss. www.meteosuisse.admin.ch, last accessed on 16/08/19.

[7] Intel Stratix 10 FPGA. www.intel.com/content/www/us/en/products/programmable/fpga/stratix-10.html, last accessed on 10/08/19.

[8] Intel Stratix 10 High-Performance Design Handbook. https://www.intel.com/content/www/us/en/programmable/documentation/jbr1444752564689.h last accessed on 16/08/19.

[9] Kesch and Es-cha. https://www.cscs.ch/computers/kesch-escha-meteoswiss/, last accessed on 02/08/19.

[10] MeteoSwiss. https://en.wikipedia.org/wiki/MeteoSwiss, last accessed on 02/08/19.

[11] MeteoSwiss COSMO Forecasting System. https://www.meteoswiss.admin.ch/home/measurement-and-forecasting-systems/warning-and-forecasting-systems/cosmo-forecasting-system.html, last accessed on 16/08/19.

[12] MeteoSwiss GPU. https://blogs.nvidia.com/blog/2015/09/15/gpus-weather/, last accessed on 02/08/19.

[13] Nonctua, Univeristy of Paderborn. https://pc2.uni-paderborn.de/hpc-services/available-systems/noctua/, last accessed on 12/08/19.

[14] Nvidia Article MeteoSwiss GPU. https://blogs.nvidia.com/blog/2015/09/15/gpus-weather/, last accessed on 02/08/19.

[15] OpenCL. www.khronos.org/opencl, last accessed on 10/08/19.

[16] Stratix 10 Hyperflex Architecture. https://www.intel.com/content/dam/www/programm 01220-hyperflex-architecture-fpga-socs.pdf.

[17] Swiss National Supercomputing Center. www.cscs.ch, last accessed on 16/08/19.

[18] TOP500. https://en.m.wikipedia.org/wiki/TOP500, last accessed on 02/08/19.

[19] Pascal Spörri Andrea Arteaga. COSMO C++ Dynamical Core Documentation. 2017.

[20] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler. Stateful dataflow multigraphs: A data-centric model for high-performance parallel programs. *CoRR*, abs/1902.10345, 2019.

[21] Sunil Shukla David F. Bacon, Rodric Rabbah. FPGA Programming for the Masses. *Communications of the ACM*, 2013.

[22] Jeffrey Chromczak David Galloway Ben Gamsa Valavan Manohararajah Ian Milton Tim Vanderhoek David Lewis, Gordon Chiu and John Van Dyken. The Stratix™ 10 Highly Pipelined FPGA Architecture. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016.

[23] D.J.Kuck R.H.Kuhn D.D.Gajski, D.A.Padua. A Second Opinion on Data Flow Machines and Languages. *IEE 1982*, 1982.

[24] Doris Folini. Climate, weather, space weather: model development in an operational context. *J. Space Weather Space Clim. 2018*, 2018.

[25] Tomofumi Yuki Steven Derrien Sanjay Rajopadhye Gaël Deest, Nicolas Estibals. Towards Scalable and Efficient FPGA Stencil Accelerators. *IMPACT 2016*, 2016.

[26] Robert G. Clapp Haohuan Fu. Eliminating the Memory Bottleneck: An FPGA-based Solution for 3D Reverse Time Migration. *FPGA11*, 2011.

[27] Mark Horowitz. Computing's Energy Problem (and what we can do about it). *ISSCC 2014*, 2014.

[28] Jonathan Rose Ian Kuon. Measuring the Gap between FPGAs and ASICs. *FPGA 06*, 2006.

[29] Stephen Neuendorffer Juanjo Noguera Kees Vissers Zhiru Zhang Jason Cong, Bin Liu. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEE 2011*, 2011.

[30] Patrick Cooke Greg Stitt Jeremy Fowers, Greg Brown. A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-Window Applications. *FPGA12*, 2012.

[31] Torsten Hoefler Johannes de Fine Licht, Michaela Blott. Designing scalable FGPA architectures using high-level synthesis. *PPoPP18*, 2018.

[32] Torsten Hoefler Johannes de Fine Licht, Simon Meierhans. Transformations of High-Level Synthesis Codes for High-Performance Computing. *arXiv.org*, 2018.

[33] Perry Cheng Rodric Rabbah Joshua Auerbach, David F.Bacon. Lime: a Java-Compatible and Synthesizable Language for Heterogeneous Architectures. *SPLASH10*, 2010.

[34] Vasily Volkov Samuel Williams Jonathan Carter Leonid Oliker David Patterson John Shalf Katherine Yelick Kaushik Datta, Mark Murphy. Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures. *IEEE 2008*, 2008.

[35] K. Scott Hemmert Keith D. Underwood. Closing the gap: CPU and FPGA Trends in sustainable floating-point BLAS performance. *FCCM 04*, 2004.

[36] Helmar Burkhart Matthias Christen, Olaf Schenk. PATUS: A Code Generation and Autotuning Framework For Parallel Iterative Stencil Computations on Modern Microarchitectures. *IEEE 2011*, 2011.

[37] Berten Digital Signal Processing. GPU vs FPGA Performance Comparison. www.bertendsp.com.

[38] Huiyang Zhou Qi Jia. Tuning Stencil Codes in OpenCL for FPGAs. *IEEE 2016*, 2016.

[39] Junichi Ishida Kohei Kawano Chiashi Muroi Takashi Shimokawabe, Takayuki Aoki. 145 TFlops Performance on 3990 GPUs of TSUBAME 2.0 Supercomputer for an Operational Weather Prediction. *ICCS 2011*, 2011.

[40] Naoyuki Onodera Takashi Shimokawabe, Takayuki Aoki. High-productivity Framework for Large-scale GPU/CPU Stencil Applications. *ICCS 2016*, 2016.

[41] Oliver Fuhrer Mauro Bianco Thomas C. Schulthess Tobias Gysi, Carlos Osuna. STELLA: A Domain-specific Tool for Structured Grid Methods in Weather and Climate Models. *SC 15*, 2015.

[42] Torsten Hoefler Tobias Gysi, Tobias Grosser. MODESTO: Data-centric Analytic Optimization of Complex Stencil Programs on Heterogeneous Architectures. *ICS 15*, 2015.

[43] Oliver Fuhrer Xavier Lapillonne Carlos E. Osuna Robert Pincus Jon Rood William Sawyer Valentin Clement, Sylviane Ferrachat. The CLAW DSL: Abstractions for Performance Portable Weather and Climate Models. *PASC 18*, 2018.

[44] Saturo Yamamoto Wang Luzhou, Kentaro Sano. Domain-Specific Language and Compiler for Stencil Computation on FPGA-based Systolic Computational-Memory Array. *ARC'12*, 2012.

jacobi2d.json

```
{
        "inputs": {
                "a": {
                        "data": "stencils/data/zeros_16x16_fp64.dat",
                        "data_type": "float64"
                }
        },
        "outputs": ["b"],
        "dimensions": [16, 16],
        "program": {
                "b": {
                        "computation_string": "res = 0.25 * (a[j-1,k] + a[j+1,k] + a[j,k-1] + a[j,k+1])",
                        "boundary_condition": {
                                "a": {
                                        "type": "constant",
                                        "value": 1.0
                                }
                        },
                        "data_type": "float32"
                }
        }
}
```

jacobi3d.json


```
{
        "inputs": {
                "a": {
                        "data": "stencils/data/zeros_16x16x16_fp64.dat",
                        "data_type": "float64"
                }
        },
        "outputs": ["b"],
        "dimensions": [16, 16, 16],
        "program": {
                "b": {
                        "computation_string": "res = 0.16666666 * (a[i-1,j,k] + a[i+1,j,k] + a[i,j-1,k] + a[i,j+1,k] + a[i,j,k
-1] + a[i,j,k+1])",
                        "boundary_condition": {
                                "a": {
                                        "type": "constant",
                                        "value": 1.0
                                }
                        },
                        "data_type": "float64"
                }
        }
}
```

bounded_queue.py

```
import collections
from typing import List

import numpy as np


class BoundedQueue:
    """
        The BoundedQueue class represents or models the buffers within the data flow design of our memory optimization model
        for the stencil operators on FPGA.

        Notes:
            - implementation:               Uses two stacks as underlying data structure to ensure overall complexity of O(1)
                                            for appendleft() and pop().
            - maxsize for bounded queue:    Default behaviour is to remove oldest element of queue, therefore we have to check
                                            it and raise an exception.
            - reference:                    https://docs.python.org/3/library/collections.html#deque-objects
    """

    def __init__(self,
                 name: str,
                 maxsize: int,
                 swap_out: bool = False,
                 collection: List = [],
                 verbose: bool = False) -> None:
        """
        Create new BoundedQueue with given initialization parameters.
        :param name: name of the queue
        :param maxsize: maximum number of elements the queue can hold at a time
        :param swap_out: set whether or not the buffer is swapped out (might get overridden by the optimizer)
        :param collection: initial data in queue
        :param verbose: flag for console output logging
        """
        # save params
        self.maxsize: int = maxsize if maxsize > 0 else 1  # maxsize must be at least 1 to correctly forward data
        self.name: str = name
        # create queue
        self.queue: collection.dequeue = collections.deque(collection, self.maxsize)
        # init current size
        self.current_size: int = len(collection)
        # indication of where the buffer is located (slow memory or fast memory)
        self.swap_out = swap_out
        # flag for verbose console output
        self.verbose = verbose

    def __repr__(self):
        # override default implementation to return nice output e.g. if a collection of queue is being printed
        return str(self)
```

bounded_queue.py

```
def __str__(self):
    # return an useful and human readable info string from the queue
    return "BoundedQueue: {}, current size: {}, max size: {}".format(self.name, self.current_size, self.maxsize)

def import_data(self, data):
    """
    Add data elements to queue.
    :param data: initial data in queue
    :return: nothing
    """
    if self.maxsize < len(data):
        raise RuntimeError("max size of queue ({}) is smaller than the data collection size ({})"
                           .format(self.maxsize, len(data)))
    else:
        self.queue: collections.deque = collections.deque(data, self.maxsize)
        self.current_size = len(data)

def export_data(self):
    """
    Return the current content of the hole queue.
    :return: numpy data array
    """
    return np.array(self.queue)[::-1]

def try_peek_last(self):
    """
    Return last data element (next element that gets dequeued) without removing it from the queue.
    :return: last data element on success, False otherwise
    """
    if self.current_size > 0:  # check bound
        return self.queue[self.current_size-1]
    else:
        return False

def size(self) -> int:
    """
    Get number of data items the queue currently contains.
    :return: current queue size
    """
    return self.current_size

def is_empty(self) -> bool:
    """
    Test if queue is empty.
    :return: if queue is empty
    """
    return self.size() == 0

def is_full(self) -> bool:
```

```
    """
    Test if queue is full.
    :return: if queue is full
    """
    return self.size() == self.maxsize

def enqueue(self, item) -> None:
    """
    Add data element to queue, causes an exception if queue is full.
    :param item: data element
    :return: None
    """
    if self.current_size >= self.maxsize:  # check bound
        raise RuntimeError("buffer {} overflow occurred".format(self.name))
    # add a new item to the left side
    self.queue.appendleft(item)
    # adjust counter
    self.current_size += 1

def dequeue(self):
    """
    Remove and return data element from queue, causes an exception if queue is empty.
    :return: data element
    """
    if self.current_size > 0:  # check bound
        # adjust size
        self.current_size -= 1
        # return and remove the rightmost item
        return self.queue.pop()
    else:
        raise RuntimeError("buffer {} underflow occurred".format(self.name))

def try_enqueue(self, item) -> bool:
    """
    Add data element to queue..
    :param item: data item
    :return: True: successful, False: unsuccessful
    """
    # check bound, do not raise exception in case of an overflow
    if self.current_size >= self.maxsize:
        # report: unsuccessful
        return False
    # add a new item to the left side
    self.queue.appendleft(item)
    # adjust counter
    self.current_size += 1
    # report: successful
    return True
```

```
    def try_dequeue(self):
        """
        Remove and return data item from queue.
        :return: data item: successful, False: unsuccessful
        """
        # check bound, do not raise exception in case of an underflow
        if self.current_size > 0:
            # adjust size
            self.current_size -= 1
            # return and remove the rightmost item
            return self.queue.pop()
        else:
            # report: unsuccessful
            return False

    def peek(self, index: int):
        """
        Returns data item at position 'index' without removal, causes an exception if index > BoundedQueue.current_size
        :param index: queue position of peeking element
        :return: data item
        """
        # check bound
        if self.current_size <= index:
            raise RuntimeError("buffer {} index out of bound access occurred".format(self.name))
        else:
            return self.queue[index]


if __name__ == "__main__":
    """
        Simple debugging example
    """
    # create dummy queue
    queue = BoundedQueue(name="debug",
                         maxsize=5,
                         collection=[1, 2, 3, 4, 5])
    # do some basic function calls and check if it crashes
    try:
        print("Enqueue element into full queue, should throw an exception.")
        queue.enqueue(6)
        print("Peek element at pos=3, value is: " + str(queue.peek(3)))
    except Exception as ex:
        print("Exception has been thrown.\n{}".format(ex.__traceback__))
```

calculator.py

```
import ast
import math
import operator
from typing import Dict


class Calculator:
    """
        The Calculator (wrapper) class can evaluate a (python) mathematical expression string in conjunction with a
        variable-to-value mapping and compute its result.
    """

    def __init__(self, verbose: bool = False) -> None:
        # save params
        self.verbose = verbose
        # create ast calculator object
        self.calc = self.Calc()

    """
        Mapping between ast operation object and operator operation object.
    """
    _OP_MAP: Dict[type(ast), type(operator)] = {
        ast.Add: operator.add,
        ast.Sub: operator.sub,
        ast.Mult: operator.mul,
        ast.Div: operator.truediv,
        ast.Invert: operator.neg,
        ast.USub: operator.sub
    }

    """
        Mapping between ast comparison object and operator comparison object.
    """
    _COMP_MAP: Dict[type(ast), type(operator)] = {
        ast.Lt: operator.lt,
        ast.LtE: operator.le,
        ast.Gt: operator.gt,
        ast.GtE: operator.ge,
        ast.Eq: operator.eq
    }

    """
        Mapping between mathematical functions (string) and mathematical objects.
    """
    _CALL_MAP: Dict[str, type(math)] = {
        "sin": math.sin,
        "cos": math.cos,
        "tan": math.tan,
        "sinh": math.sinh,
```

```
    "cosh": math.cosh
}

def eval_expr(self, variable_map: Dict[str, float], computation_string: str) -> float:
    """
    Given a mapping from variable names to values and a mathematical (python) expression, it evaluates the
    expression.
    :param variable_map: a dictionary map containing all variables of the computation_string
    :param computation_string: a python-syntax-compatible input string
    :return: the result of the expression
    """
    return self.calc.evaluate(variable_map, computation_string)

"""
    Internal Calc class for the actual calculation.
"""

class Calc(ast.NodeVisitor):

    def __init__(self) -> None:
        """
        Initializes the actual expression evaluator.
        """
        # init variable map
        self.var_map: Dict[str, float] = dict()

    def visit_BinOp(self, node: ast) -> float:
        """
        Binary operation evaluator.
        :param node: ast tree node
        :return: result of binary operation (LHS op RHS)
        """
        left = self.visit(node.left)
        right = self.visit(node.right)
        return Calculator._OP_MAP[type(node.op)](left, right)

    def visit_Num(self, node: ast) -> float:
        """
        Numeral evaluator.
        :param node: ast tree node
        :return: numeral value
        """
        return node.n

    def visit_Expr(self, node: ast) -> float:
        """
        Expression evaluator.
        :param node: ast tree node
        :return: value of the expression evaluated with the given variable map
```

```
    """
    return self.visit(node.value)

def visit_IfExp(self,
                node: ast) -> float:  # added for ternary operations of the (python syntax: a if expr else b)
    """
    Ternary operator evaluator.
    :param node: ast tree node
    :return: value of if clause if comparison evaluates to true, value of else clause otherwise
    """
    if self.visit(node.test):  # evaluate comparison
        return self.visit(node.body)  # use left
    else:
        return self.visit(node.orelse)  # use right

def visit_Compare(self, node: ast) -> bool:  # added for ternary operations (python syntax: a if expr else b)
    """
    Comparison evaluator.
    :param node: ast tree node
    :return: whether the comparison evaluates to true of false
    """
    left = self.visit(node.left)
    right = self.visit(node.comparators[0])
    return Calculator._COMP_MAP[type(node.ops[0])](left, right)

def visit_Name(self, node: ast) -> float:
    """
    Variable evaluator.
    :param node: ast tree node
    :return: variable value
    """
    return self.var_map[node.id]

def visit_Call(self, node: ast) -> float:
    """
    Function evaluator.
    :param node: ast tree node
    :return: value of the evaluated mathematical function
    """
    return Calculator._CALL_MAP[node.func.id](self.visit(node.args[0]))

def visit_UnaryOp(self, node: ast) -> float:
    return Calculator._OP_MAP[type(node.op)](0.0, self.visit(node.operand))

@classmethod
def evaluate(cls,
             variable_map: Dict[str, float],
             expression: str) -> float:
    """
```

```
            Entry point for calculator.
            :param variable_map: mapping from value names to values
            :param expression: mathematical expression in string format
            :return: result of the evaluated string
            """
            # remove LHS of the equality sign e.g. 'res=...' --> '...'
            if "=" in expression:
                expression = expression[expression.find("=") + 1:]
            # parse tree
            tree = ast.parse(expression)
            # create calculator
            calc = cls()
            # add the variable value mapping
            calc.var_map = variable_map
            # evaluate expression tree and return result
            return calc.visit(tree.body[0])


'''
    safe (in contrast to evaluate()) python expression evaluator class
        -input:
            - map: variable name -> value
            - computation string (must be python syntax, e.g. for ternary operations)
        - output: resulting value

    credits: https://stackoverflow.com/questions/33029168/how-to-calculate-an-equation-in-a-string-python

'''

if __name__ == "__main__":

    '''
        simple example for debugging purpose
    '''

    variables = dict()
    variables["a"] = 7
    variables["b"] = 2

    for var in variables:
        print("name: {}, value: {}".format(var, str(variables[var])))

    computation = "cos(-a + b) if (a > b) else (a + 5) * b"
    calculator = Calculator()
    result = calculator.eval_expr(variables, computation)
    print("{} = {}".format(computation, str(result)))
```

compute_graph_nodes.py

```
import ast
import operator
from typing import List, Dict

from base_node_class import BaseOperationNodeClass
from calculator import Calculator


class Name(BaseOperationNodeClass):
    """
        The Name class is a subclass of the BaseOperationNodeClass and represents the variable name node in the computation
        tree.
    """

    def __init__(self,
                 ast_node: ast,
                 number: int) -> None:
        """
        Create new Name node with given initialization parameters.
        :param ast_node: abstract syntax tree node of the computation
        :param number: tree walk numbering
        """
        # initialize superclass
        super().__init__(ast_node, number)

    def generate_name(self,
                      ast_node: ast) -> str:
        """
        Variable name implementation of generate_name.
        :param ast_node: abstract syntax tree node of the computation
        :returns generated name
        """
        return ast_node.id


class Num(BaseOperationNodeClass):
    """
        The Name class is a subclass of the BaseOperationNodeClass and represents the numeral node in the computation tree.
    """

    def __init__(self,
                 ast_node: ast,
                 number: int) -> None:
        """
        Create new Num node with given initialization parameters.
        :param ast_node: abstract syntax tree node of the computation
        :param number: tree walk numbering
        """
        # initialize superclass
```

```
        super().__init__(ast_node, number)

    def generate_name(self,
                      ast_node: ast) -> str:
        """
        Numeral implementation of generate_name.
        :param ast_node: abstract syntax tree node of the computation
        :returns generated name
        """
        return ast_node.n


class Binop(BaseOperationNodeClass):
    """
        The Name class is a subclass of the BaseOperationNodeClass and represents the binary operation node in the
        computation tree.
    """

    def __init__(self,
                 ast_node: ast,
                 number: int) -> None:
        """
        Create new Binop node with given initialization parameters.
        :param ast_node: abstract syntax tree node of the computation
        :param number: tree walk numbering
        """
        # initialize superclass
        super().__init__(ast_node, number)

    """
        Mapping between ast mathematical operations and the string name of the operation.
    """
    _OP_NAME_MAP: Dict[type(ast), str] = {
        ast.Add: "add",
        ast.Sub: "sub",
        ast.Mult: "mult",
        ast.Div: "div",
        ast.Invert: "neg"
    }

    """
        Mapping between the string name of the operation and its symbol.
    """
    _OP_SYM_MAP: Dict[str, str] = {
        "add": "+",
        "sub": "-",
        "mult": "*",
        "div": "/",
        "neg": "-"
```

```
    }

    def generate_name(self,
                      ast_node: ast) -> str:
        """
        Binary operation implementation of generate_name.
        :param ast_node: abstract syntax tree node of the computation
        :returns generated name
        """
        return self._OP_NAME_MAP[type(ast_node.op)]

    def generate_op_sym(self) -> str:
        """
        Generates the symbol of the mathematical operation out of the operation string (e.g. add, sub, ..).
        :returns generated symbol
        """
        return self._OP_SYM_MAP[self.name]


class Call(BaseOperationNodeClass):
    """
        The Call class is a subclass of the BaseOperationNodeClass and represents the function calls (e.g. sin/cos,..) node
        in the computation tree.
    """

    def __init__(self,
                 ast_node: ast,
                 number: int) -> None:
        """
        Create new Function (call) node with given initialization parameters.
        :param ast_node: abstract syntax tree node of the computation
        :param number: tree walk numbering
        """
        # initialize superclass
        super().__init__(ast_node, number)

    def generate_name(self,
                      ast_node: ast) -> str:
        """
        Function call implementation of generate_name.
        :param ast_node: abstract syntax tree node of the computation
        :returns generated name
        """
        return ast_node.func.id


class Output(BaseOperationNodeClass):
    """
        The Output class is a subclass of the BaseOperationNodeClass and represents the output node in the computation tree.
```

```
    """

    def __init__(self,
                 ast_node: ast,
                 number: int) -> None:
        """
        Create new Output node with given initialization parameters.
        :param ast_node: abstract syntax tree node of the computation
        :param number: tree walk numbering
        """
        # initialize superclass
        super().__init__(ast_node, number)

    def generate_name(self,
                      ast_node: ast) -> str:
        """
        Output implementation of generate_name.
        :param ast_node: abstract syntax tree node of the computation
        :returns generated name
        """
        return ast_node.targets[0].id


class Subscript(BaseOperationNodeClass):
    """
        The Subscript class is a subclass of the BaseOperationNodeClass and represents the array field access node in the
        computation tree.
    """

    def __init__(self,
                 ast_node: ast,
                 number: int) -> None:
        """
        Create new Subscript node with given initialization parameters.
        :param ast_node: abstract syntax tree node of the computation
        :param number: tree walk numbering
        """
        # initialize superclass
        super().__init__(ast_node, number)
        # initialize local fields
        self.index: List[int] = list()
        self.create_index(ast_node)

    """
        Mapping between the index of the operation and its position (actually always 0).
    """
    _VAR_MAP: Dict[str, int] = {
        "i": 0,
        "j": 0,
```

```
        "k": 0
    }

    """
        Mapping between the operation and its symbol.
    """
    _OP_SYM_MAP: Dict[type(ast), str] = {
        ast.Add: "+",
        ast.Sub: "-"
    }

    def create_index(self,
                     ast_node: ast) -> None:
        """
        Create the numerical index of the array field access e.g. convert [i+2, j-3, k] to [2,-3,0]
        :param ast_node: abstract syntax tree node of the computation
        :returns generated name
        """
        # create index
        self.index = list()
        for slice in ast_node.slice.value.elts:
            if isinstance(slice, ast.Name):
                self.index.append(self._VAR_MAP[slice.id])
            elif isinstance(slice, ast.BinOp):
                # note: only support for index variations [i, j+3,..]
                # read index expression
                expression = str(slice.left.id) + self._OP_SYM_MAP[type(slice.op)] + str(slice.right.n)
                # convert [i+1,j, k-1] to [1, 0, -1]
                calculator = Calculator()
                self.index.append(calculator.eval_expr(self._VAR_MAP, expression))

    def generate_name(self,
                      ast_node: ast) -> str:
        """
        Subscript (array field access) implementation of generate_name.
        :param ast_node: abstract syntax tree node of the computation
        :returns generated name
        """
        return ast_node.value.id

    def generate_label(self) -> str:
        """
        Subscript (array field access) implementation of generate_label.
        :returns generated label
        """
        return str(self.name) + str(self.index)


class Ternary(BaseOperationNodeClass):
```

```
    """
        The Ternary operator class is a subclass of the BaseOperationNodeClass and represents ternary operation of the
        form: expression_true if comparison_expression else expression_false
    """

    def __init__(self,
                 ast_node: ast,
                 number: int) -> None:
        """
        Create new Ternary node with given initialization parameters.
        :param ast_node: abstract syntax tree node of the computation
        :param number: tree walk numbering
        """
        # initialize superclass
        super().__init__(ast_node, number)

    def generate_name(self,
                      ast_node: ast) -> str:
        """
        Ternary operator implementation of generate_name.
        :param ast_node: abstract syntax tree node of the computation
        :returns generated name
        """
        return "?"


class Compare(BaseOperationNodeClass):
    """
        The Comparison operator class is a subclass of the BaseOperationNodeClass and represents the comparison of two
    """

    def __init__(self,
                 ast_node: ast,
                 number: int) -> None:
        """
        Create new Compare node with given initialization parameters.
        :param ast_node: abstract syntax tree node of the computation
        :param number: tree walk numbering
        """
        # set comparison operator field
        self.op: operator = self._COMP_MAP[type(ast_node.ops[0])]
        # initialize superclass
        super().__init__(ast_node, number)

    """
        Mapping between the abstract syntax tree (python) comparison operator and the operator comparison operator.
    """
    _COMP_MAP: Dict[type(ast), type(operator)] = {
        ast.Lt: operator.lt,
```

```
        ast.LtE: operator.le,
        ast.Gt: operator.gt,
        ast.GtE: operator.ge,
        ast.Eq: operator.eq
    }

    """
        Mapping between the operator comparison operator and its mathematical string symbol.
    """
    _COMP_SYM: Dict[type(operator), str] = {
        operator.lt: "<",
        operator.le: "<=",
        operator.gt: ">",
        operator.ge: ">=",
        operator.eq: "=="
    }

    def generate_name(self,
                      ast_node: ast) -> str:
        """
        Comparison operator implementation of generate_name.
        :param ast_node: abstract syntax tree node of the computation
        :returns generated name
        """
        return self._COMP_SYM[self.op]


class UnaryOp(BaseOperationNodeClass):
    """
         The UnaryOp operator class is a subclass of the BaseOperationNodeClass and represents unary operations. In our
         case we only support negation (mathematical - sign) as unary operation.
     """

    def __init__(self,
                 ast_node: ast,
                 number: int) -> None:
        """
        Create new unary operation node with given initialization parameters.
        :param ast_node: abstract syntax tree node of the computation
        :param number: tree walk numbering
        """
        # set unary operator field
        self.op: operator = self._UNARYOP_MAP[type(ast_node.op)]
        # initialize superclass
        super().__init__(ast_node, number)

    """
        Mapping between the ast unary operation and the operator operation.
    """
```

compute_graph_nodes.py

```
_UNARYOP_MAP: Dict[type(ast), type(operator)] = {
    ast.USub: operator.sub
}

"""
    Mapping between the operator unary operator and its mathematical string.
"""
_UNARYOP_SYM: Dict[type(operator), str] = {
    operator.sub: "neg"
}

"""
     Mapping between the mathematical string and its symbol.
 """
_UNARYOP_SYM_NAME = {
    "neg": "-"
}

def generate_name(self,
                  ast_node: ast) -> str:
    """
    Unary operator implementation of generate_name.
    :param ast_node: abstract syntax tree node of the computation
    :returns generated name
    """
    return self._UNARYOP_SYM[self.op]

def generate_op_sym(self) -> str:
    """
    Generates the symbol of the mathematical operation out of the operation string.
    :returns generated symbol
    """
    return self._UNARYOP_SYM_NAME[self.name]
```

compute_graph.py

```
import ast
from typing import List, Dict, Set

import networkx as nx

import helper
from base_node_class import BaseOperationNodeClass
from compute_graph_nodes import Name, Num, Binop, Call, Output, Subscript, Ternary, Compare, UnaryOp


class ComputeGraph:
    """
        The ComputeGraph class manages the inner data flow of a single computation respectively a single kernel
        including its properties e.g. latency, internal buffer sizes and field accesses.

        Notes:
            - Creation of a proper graph representation for the computation data flow graph.
            - Credits for node-visitor: https://stackoverflow.com/questions/33029168/how-to-calculate-an-equation-in-a-string-
python
            - More info: https://networkx.github.io/

        Note about ast structure:
            tree.body[i] : i-th expression
            tree.body[i] = Assign: of type: x = Expr
            tree.body[i].targets          ->
            tree.body[i].value = {BinOp, Name, Call}
            tree.body[i] = Expr:
            tree.body[i].value = BinOp    -> subtree: .left, .right, .op {Add, Mult, Name, Call}
            tree.body[i].value = Name     -> subtree: .id (name)
            tree.body[i].value = Call     -> subtree: .func.id (function), .args[i] (i-th argument)
            tree.body[i].value = Subscript -> subtree: .slice.value.elts[i]: i-th parameter in [i, j, k, ...]
    """

    def __init__(self,
                 verbose: bool = False) -> None:
        """
        Create new ComputeGraph with given initialization parameters.
        :param verbose: flag for console output logging
        """
        # set parameter variables
        self.verbose = verbose
        # read static parameters from config file
        self.config: Dict[str, int] = helper.parse_json("compute_graph.config")
        # initialize internal data structures
        self.graph: nx.DiGraph = nx.DiGraph()  # networkx (library) compute graph with compute_graph_nodes as nodes
        self.tree: type(ast) = None  # abstract syntax tree (python) data structure
        self.max_latency: int = -1  # (non-valid) initial value for the maximum latency (critical path) of the
        # computational tree
        self.inputs: Set[BaseOperationNodeClass] = set()  # link to all nodes that feed input into this computation
```

```
    self.outputs: Set[BaseOperationNodeClass] = set()  # link to all nodes this computation feeds data to
    self.min_index: Dict[str, List] = dict()  # per input array the last access index of the stencil
    self.max_index: Dict[str, List] = dict()  # per input array the furthest access index of the stencil
    self.buffer_size: Dict[str, List] = dict()  # size (dimensional) from the last to the first access (determines
    # the internal buffer size)
    self.accesses: Dict[str, List[List]] = dict()  # dictionary containing all field accesses for a specific
    # resource e.g. {"A":{[0,0,0],[0,1,0]}} for the stencil "res = A[i,j,k] + A[i,j+1,k]"

@staticmethod
def create_operation_node(node: ast,
                          number: int) -> BaseOperationNodeClass:
    """
    Create operation node of the correct type.
    :param node: abstract syntax tree node
    :param number: tree numbering
    :return: corresponding operation node
    """
    if isinstance(node, ast.Name):  # variables or array access
        return Name(node, number)
    elif isinstance(node, ast.Num):  # static value
        return Num(node, number)
    elif isinstance(node, ast.BinOp):  # binary operation
        return Binop(node, number)
    elif isinstance(node, ast.Call):  # function (e.g. sin, cos,..)
        return Call(node, number)
    elif isinstance(node, ast.Assign):  # assign operator (var = expr;)
        return Output(node, number)
    elif isinstance(node, ast.Subscript):  # array access (form: arr[i,j,k])
        return Subscript(node, number)
    elif isinstance(node, ast.IfExp):  # if/else clause of ternary operation
        return Ternary(node, number)
    elif isinstance(node, ast.Compare):  # comparison of ternary operation
        return Compare(node, number)
    elif isinstance(node, ast.UnaryOp):  # negation of value ('-' sign)
        return UnaryOp(node, number)
    else:
        raise Exception("Unknown AST type {}".format(type(node)))

def setup_internal_buffers(self,
                           relative_to_center=True) -> None:
    """
    Set up minimum/maximum index and accesses for the internal data structures.
    :param relative_to_center: if true, the center of the stencil is at position [0,0,0] respecively 0, if false,
    the furthest element is at position [0,0,0] and all other accesses on the same input field are relative to that
    (i.e. negative)
    """
    # init dicts
    self.min_index = dict()  # min_index["buffer_name"] = [i_min, j_min, k_min]
    self.max_index = dict()
```

```
    self.buffer_size = dict()  # buffer_size["buffer_name"] = size
    # find min and max index
    for inp in self.inputs:
        if isinstance(inp, Subscript):  # subscript nodes only
            if inp.name in self.min_index:
                if inp.index < self.min_index[inp.name]:  # check min
                    self.min_index[inp.name] = inp.index
                if inp.index >= self.max_index[inp.name]:  # check max
                    self.max_index[inp.name] = inp.index
            else:  # first entry
                self.min_index[inp.name] = inp.index
                self.max_index[inp.name] = inp.index
            if inp.name not in self.accesses:  # create initial list
                self.accesses[inp.name] = list()
            self.accesses[inp.name].append(inp.index)  # add entry
    # set buffer_size = max_index - min_index
    for buffer_name in self.min_index:
        self.buffer_size[buffer_name] = [abs(a_i - b_i) for a_i, b_i in zip(self.max_index[buffer_name],
                                                                            self.min_index[buffer_name])]
    # update access to have [0,0,0] for the max_index (subtract it from all)
    if not relative_to_center:
        for field in self.accesses:
            updated_entries = list()
            for entry in self.accesses[field]:
                updated_entries.append(helper.list_subtract_cwise(entry, self.max_index[field]))
            self.accesses[field] = updated_entries

def determine_inputs_outputs(self) -> None:
    """
    Fill up internal input and output data structures with the corresponding nodes.
    """
    # create empty sets
    self.inputs = set()
    self.outputs = set()
    # idea: do a tree-walk: all nodes with cardinality(predecessor)=0 are inputs, all nodes with cardinality(
    # successor)=0 are outputs
    for node in self.graph.nodes:
        if len(self.graph.pred[node]) == 0:
            self.inputs.add(node)
        if len(self.graph.succ[node]) == 0:
            self.outputs.add(node)

def contract_edge(self,
                  u: BaseOperationNodeClass,
                  v: BaseOperationNodeClass) -> None:
    """
    Contract node v into node u.
    :param u: contractor node
    :param v: contracted node
```

```
    """
    # add edges of node v to node u
    for edge in self.graph.succ[v]:
        self.graph.add_edge(u, edge)
    for edge in self.graph.pred[v]:
        self.graph.add_edge(edge, u)
    # remove node v
    self.graph.remove_node(v)

def generate_graph(self,
                   computation_string: str) -> nx.DiGraph:
    """
    Create networkx graph of the mathematical computation given in the computation_string.
    :param computation_string:
    :return networkx (library) graph of the computation with nodes from compute_graph_nodes
    """
    # generate abstract syntax tree
    self.tree = ast.parse(computation_string)
    # iterate over all equations (e.g. res=a+b;b=c+d; ...)
    for equation in self.tree.body:
        # check if base node is of type Expr or Assign
        if isinstance(equation, ast.Assign):
            lhs = self.create_operation_node(equation, 0)  # left hand side equation
            rhs = self.ast_tree_walk(equation.value, 1)  # right hand side of equation
            self.graph.add_edge(rhs, lhs)
    # merge ambiguous variables in tree (implies: merge of ast.Assign trees into a single tree)
    outp_nodes = list(self.graph.nodes)
    for outp in outp_nodes:
        if isinstance(outp, Name):
            inp_nodes = list(self.graph.nodes)
            for inp in inp_nodes:
                if isinstance(outp, Subscript) and outp is not inp and outp.name == inp.name and \
                        outp.index == inp.index:
                    # only contract if the indices and the names match
                    self.contract_edge(outp, inp)
                elif isinstance(outp, Name) and outp is not inp and outp.name == inp.name:
                    # contract nodes if the names match
                    self.contract_edge(outp, inp)
    # test if graph is now a single component (for directed graph: each non-output must have at least one successor)
    for node in self.graph.nodes:
        if not isinstance(node, Output) and len(self.graph.succ[node]) == 0:
            raise RuntimeError("Kernel-internal data flow is not single component (must be connected in the sense "
                               "of a DAG).")
    return self.graph

def ast_tree_walk(self,
                  node: ast,
                  number: int) -> BaseOperationNodeClass:
    """
```

compute_graph.py

```
Recursively walk through the abstract syntax tree structure.
:param node: current node
:param number: tree numbering
:return: reference of current node
"""
# create node
new_node = self.create_operation_node(node, number)
# add node to graph
self.graph.add_node(new_node)
# node type specific implementation of the tree walk
if isinstance(node, ast.BinOp):
    # do tree-walk recursively and get references to children (to create the edges to them)
    lhs = self.ast_tree_walk(node.left, ComputeGraph.child_left_number(number))  # left hand side
    rhs = self.ast_tree_walk(node.right, ComputeGraph.child_right_number(number))  # right hand side
    # add edges from parent to children
    self.graph.add_edge(lhs, new_node)
    self.graph.add_edge(rhs, new_node)
elif isinstance(node, ast.Call):
    # do tree-walk for all arguments
    if len(node.args) > 2:
        raise NotImplementedError("Current implementation does not support more than two arguments due"
                                  " to the binary tree numbering convention")
    # process first argument
    first = self.ast_tree_walk(node.args[0], ComputeGraph.child_left_number(number))
    self.graph.add_edge(first, new_node)
    # check if second argument exist
    if len(node.args) >= 2:
        second = self.ast_tree_walk(node.args[1], ComputeGraph.child_right_number(number))
        self.graph.add_edge(second, new_node)
elif isinstance(node, ast.Name):
    # nothing to do
    pass
elif isinstance(node, ast.Num):
    # nothing to do
    pass
elif isinstance(node, ast.Compare):
    # do tree-walk recursively and get references to children (to create the edges to them)
    lhs = self.ast_tree_walk(node.left, ComputeGraph.child_left_number(number))  # left hand side
    rhs = self.ast_tree_walk(node.comparators[0], ComputeGraph.child_right_number(number))  # right hand side
    # add edges from parent to children
    self.graph.add_edge(lhs, new_node)
    self.graph.add_edge(rhs, new_node)
elif isinstance(node, ast.IfExp):
    # do tree-walk recursively and get references to children (to create the edges to them)
    test = self.ast_tree_walk(node.test, 0)  # test clause
    true_path = self.ast_tree_walk(node.body, ComputeGraph.child_left_number(number))
    false_path = self.ast_tree_walk(node.orelse, ComputeGraph.child_right_number(number))
    # add edges from parent to children
    self.graph.add_edge(true_path, new_node)
```

```
        self.graph.add_edge(false_path, new_node)
        self.graph.add_edge(test, new_node)
    elif isinstance(node, ast.UnaryOp):
        # do tree-walk recursively and get references to child
        operand = self.ast_tree_walk(node.operand, ComputeGraph.child_right_number(number))
        # add edges form parent to child
        self.graph.add_edge(operand, new_node)
    return new_node

@staticmethod
def child_left_number(n: int) -> int:
    """
    Default tree numbering (number from root right to left downward per level).
    :param n: parent number
    :return: left child number
    """
    return 2 * n + 1

@staticmethod
def child_right_number(n: int) -> int:
    """
    Default tree numbering (number from root right to left downward per level).
    :param n: parent number
    :return: right child number
    """
    return 2 * n

def plot_graph(self,
               save_path: str = None) -> None:
    """
    Plot the compute graph graphically.
    :param save_path: filename of the output image, if none: do not save to file
    """
    # create drawing area
    import matplotlib.pyplot as plt  # import matplotlib only if graph plotting is set to true
    plt.figure(figsize=(20, 20))  # define drawing size. NOTE: must probably be a function of the number of nodes at
    # some point (for large graphs)
    plt.axis('off')
    # generate positions
    positions = nx.nx_pydot.graphviz_layout(self.graph, prog='dot')
    # divide nodes into different lists for colouring purpose
    nums = list()
    names = list()
    ops = list()
    outs = list()
    comp = list()
    for node in self.graph.nodes:
        if isinstance(node, Num):  # numerals
            nums.append(node)
```

compute_graph.py

```
    elif isinstance(node, Name) or isinstance(node, Subscript):  # variables
        names.append(node)
    elif isinstance(node, Binop) or isinstance(node, Call) or isinstance(node, UnaryOp):  # operations
        ops.append(node)
    elif isinstance(node, Output):  # outputs
        outs.append(node)
    elif isinstance(node, Ternary) or isinstance(node, Compare):  # comparison
        comp.append(node)
# create dictionary of the labels and add all of them
labels = dict()
for node in self.graph.nodes:
    labels[node] = node.generate_label()
# add nodes and edges
# name nodes
nx.draw_networkx_nodes(G=self.graph,
                       pos=positions,
                       nodelist=names,
                       node_color='orange',
                       node_size=3000,
                       node_shape='s',  # square
                       edge_color='black')
# output nodes
nx.draw_networkx_nodes(G=self.graph,
                       pos=positions,
                       nodelist=outs,
                       node_color='green',
                       node_size=3000,
                       node_shape='s')
# numeral nodes
nx.draw_networkx_nodes(G=self.graph,
                       pos=positions,
                       nodelist=nums,
                       node_color='#007acc',
                       node_size=3000,
                       node_shape='s')
# ternary operator nodes
nx.draw_networkx_nodes(G=self.graph,
                       pos=positions,
                       nodelist=comp,
                       node_color='#009999',
                       node_size=3000,
                       node_shape='o')  # circle
# operation nodes and edges between all nodes
nx.draw_networkx(G=self.graph,
                 pos=positions,
                 nodelist=ops,
                 node_color='red',
                 node_size=3000,
                 node_shape='o',
```

```
                    font_weight='bold',
                    font_size=16,
                    edge_color='black',
                    arrows=True,
                    arrowsize=36,
                    arrowstyle='-|>',
                    width=6,
                    linwidths=1,
                    with_labels=False)
    # add labels
    nx.draw_networkx_labels(G=self.graph,
                            pos=positions,
                            labels=labels,
                            font_weight='bold',
                            font_size=16)
    # save plot to file if save_path has been specified
    if save_path is not None:
        plt.savefig(save_path)
    # plot it
    plt.show()

def try_set_max_latency(self,
                        new_val: int) -> bool:
    """
    Update the maximum latency of the compute graph.
    :param new_val: new maximum latency candidate
    :return: whether or not the candidate is the new maximum
    """
    if self.max_latency <= new_val:
        self.max_latency = new_val
        return True
    else:
        return False

def calculate_latency(self) -> None:
    """
    Find critical path in the computation tree.
    """
    # idea: do a longest-path tree-walk (since the graph is a DAG (directed acyclic graph) we can do that
    for node in self.graph.nodes:
        if isinstance(node, Output):  # start at the output nodes and walk the tree up to the input nodes
            node.latency = 1
            self.try_set_max_latency(node.latency)
            self.latency_tree_walk(node)

def latency_tree_walk(self,
                      node: BaseOperationNodeClass) -> None:
    """
    Computation tree walk for latency calculation.
```

```
        :param node: current node
        """
        # check node type
        if isinstance(node, Name) or isinstance(node, Num) or isinstance(node, Subscript):  # variable or numeral:
            # no additional latency
            # copy parent latency to children
            for child in self.graph.pred[node]:
                child.latency = node.latency
                self.latency_tree_walk(child)
        elif isinstance(node, Binop) or isinstance(node, Call):  # function calls: additional latency of the function
            # added
            # get op latency from config
            op_latency = self.config["op_latency"][node.name]
            # add latency to children
            for child in self.graph.pred[node]:
                child.latency = max(child.latency, node.latency + op_latency)
                self.latency_tree_walk(child)
        elif isinstance(node, Output):  # output: no additional latency
            # copy parent latency to children
            for child in self.graph.pred[node]:
                child.latency = node.latency
                self.latency_tree_walk(child)
        elif isinstance(node, Ternary):  # function calls: additional latency of the conditional operator added
            # get op latency from config
            op_latency = self.config["op_latency"]["conditional"]
            # add latency to children
            for child in self.graph.pred[node]:
                child.latency = max(child.latency, node.latency + op_latency)
                self.latency_tree_walk(child)
        elif isinstance(node, Compare):  # comparison: additional latency of the comparison operator added
            # get op latency from config
            op_latency = self.config["op_latency"]["comparison"]
            # add latency to children
            for child in self.graph.pred[node]:
                child.latency = max(child.latency, node.latency + op_latency)
                self.latency_tree_walk(child)
        elif isinstance(node, UnaryOp):  # unary operator
            # get op latency from config
            op_latency = self.config["op_latency"][node.name]
            # add latency to children
            for child in self.graph.pred[node]:
                child.latency = max(child.latency, node.latency + op_latency)
                self.latency_tree_walk(child)
        else:
            raise NotImplementedError("Node type {} has not been implemented yet.".format(type(node)))
        self.try_set_max_latency(node.latency)


if __name__ == "__main__":
```

compute_graph.py

```
"""
    simple debugging example
"""
computation = "res = -a if (a+1 > b-c) else b; b = d + e"
graph = ComputeGraph()
graph.generate_graph(computation)
graph.calculate_latency()
# graph.plot_graph("compute_graph_example.png")  # write graph to file
graph.plot_graph()
```

compute_graph.config

```
{
  "op_latency": {
    "add": 16,
    "sub": 16,
    "mult": 16,
    "div": 128,
    "inv": 16,
    "sin": 128,
    "cos": 128,
    "tan": 128,
    "sinh": 128,
    "coshh": 128,
    "comparison": 16,
    "conditional": 16,
    "neg": 16
  }
}
```

base_node_class.py

```
# from __future__ import annotations  # support return type of its own class
import ast
from abc import ABCMeta, abstractmethod
from enum import Enum
from typing import List, Dict

import dace.types

from bounded_queue import BoundedQueue


class BoundaryCondition(Enum):
    """
        The BoundaryCondition Enumeration works as an adapter between the input string representation and the
        programmatically more useful enumeration. It defines the strategy used for out-of-bound stencil accesses on
        the data arrays we iterate over.
    """

    CONSTANT = 1  # use a fixed (static) value for all out-of-bound accesses
    COPY = 2  # copy the last within-bound-value for out-of-bound accesses

    @staticmethod
    def to_bc(text: str):  # -> BoundaryCondition:
        if text == "const":
            return BoundaryCondition.CONSTANT
        elif text == "copy":
            return BoundaryCondition.COPY
        else:
            raise Exception("{} is not a valid boundary condition string".format(text))


class BaseKernelNodeClass:
    """
        The BaseKernelClass provides all the basic fields and functionality for its subclasses which are the Input,
        Kernel and Output classes. These are nodes of of the KernelChainGraph.
    """

    __metaclass__ = ABCMeta

    def __init__(self, name: str,
                 data_queue: BoundedQueue,
                 data_type: dace.types.typeclass,
                 verbose: bool = False) -> None:
        """
        Create new BaseKernelNodeClass with given initialization parameters.
        :param name: name of the node
        :param data_queue: queue containing the input (Input) or final output (Output) data
        :param data_type: set whether or not the buffer is swapped out (might get overridden by the optimizer)
        :param verbose: flag for console output logging
```

```
        """
        # save params
        self.name: str = name
        self.data_queue: BoundedQueue = data_queue
        self.data_type = data_type
        if not isinstance(data_type, dace.types.typeclass):  # check type of input
            raise TypeError("Expected dace.types.typeclass, got: " + type(data_type).__name__)
        self.verbose = verbose
        # define basic node structures
        self.input_paths: Dict[str, List] = dict()  # contains all paths to the source arrays
        self.inputs: Dict[str, Dict] = dict()  # contains all predecessors
        self.outputs: Dict[str, BoundedQueue] = dict()  # contains all successors
        self.delay_buffer: Dict[str, BoundedQueue] = dict()  # contains the delay buffers for all inputs
        self.program_counter = 0  # progress program counter for simulation

    def generate_label(self) -> str:  # wrapper for customizations
        """
        Base class basic implementation of the generate_label method.
        :returns generated label
        """
        return self.name


class BaseOperationNodeClass:
    """
        The BaseOperationNodeClass class provides all the basic fields and methods for its subclasses (Num,
        Subscript,..). These are the nodes of the ComputeGraph .
    """

    __metaclass__ = ABCMeta

    def __init__(self,
                 ast_node: ast,
                 number: int,
                 verbose: bool = False) -> None:
        """
        Create new BaseOperationNodeClass with given initialization parameters.
        :param ast_node: abstract syntax tree (python) entry
        :param number: node number (tree numbering)
        :param verbose: flag for console output logging
        """
        # save params
        self.number: int = number
        self.name: str = self.generate_name(ast_node)
        self.verbose = verbose
        # set initial latency to a value distinguishable from correct values
        self.latency: int = -1

    @abstractmethod
```

base_node_class.py

```
def generate_name(self,
                  ast_node: ast) -> str:  # every subclass must implement this
    """
    Base class basic implementation of the generate_label method.
    :returns generated label
    """
    return str(self.name)

def generate_label(self) -> str:  # subclass can, if necessary, override the default implementation
    """
    Generates the node label.
    :returns generated label
    """
    return str(self.name)
```

input.py

```python
import numpy as np
from dace.types import typeclass

from base_node_class import BaseKernelNodeClass
from bounded_queue import BoundedQueue


class Input(BaseKernelNodeClass):
    """
        The Input class is a subclass of the BaseKernelNodeClass and represents an Input node in the KernelChainGraph.
        Its purpose is to feed input array data into the pipeline/dataflow design.
    """

    def __init__(self,
                 name: str,
                 data_type: typeclass,
                 data_queue: BoundedQueue = None) -> None:
        """
        Initialize the Input node.
        :param name: node name
        :param data_type: data type of the data
        :param data_queue: BoundedQueue containing the input data
        """
        # initialize superclass
        super().__init__(name=name, data_queue=data_queue, data_type=data_type)
        # set internal fields
        self.queues = dict()
        self.dimension_size = data_queue.maxsize
        self.init = False  # flag for internal initialization (must be done later when all successors are added to the
        # graph)

    def init_queues(self):
        """
        Create individual queues for all successors in order to feed data individually into the channels.
        """
        # add a queue for each successor
        self.queues = dict()
        for successor in self.outputs:
            self.queues[successor] = BoundedQueue(name=successor,
                                                  maxsize=self.data_queue.maxsize,
                                                  collection=self.data_queue.export_data())
        self.init = True  # set init flag

    def reset_old_compute_state(self):
        """
        Reset compute-specific internal state (only for Kernel node).
        """
        pass  # nothing to do
```

input.py

```
def try_read(self):
    """
    Read data from predecessor (only for Kernel and Output node).
    """
    pass  # nothing to do

def try_write(self):
    """
    Feed data to all successor channels.
    :return:
    """
    # set up all individual data queues
    if not self.init:
        self.init_queues()
    # feed data into pipeline inputs (all kernels that feed from this input data array)
    for successor in self.outputs:
        if self.queues[successor].is_empty() and not self.outputs[successor]["delay_buffer"].is_full():  # no more
            # data to feed, add bubble
            self.outputs[successor]["delay_buffer"].enqueue(None)  # insert bubble
        elif self.outputs[successor]["delay_buffer"].is_full():  # channel full, skip
            pass
        else:  # feed data into channel
            data = self.queues[successor].dequeue()
            self.outputs[successor]["delay_buffer"].enqueue(data)
            self.program_counter = self.dimension_size - max([self.queues[x].size() for x in self.queues])

def init_input_data(self, inputs):
    """
    Initialize internal queue i.e. read data from config or external file.
    :param inputs:
    :return:
    """
    # check if data is in the config or in a separate file
    if isinstance(inputs[self.name]["data"], list):  # inline
        self.data_queue.import_data(inputs[self.name]["data"])
    elif isinstance(inputs[self.name]["data"], str):  # external file
        coll = None
        if inputs[self.name]["data"].lower().endswith(('.dat', '.bin', '.data')):  # general binary data file
            coll = np.fromfile(inputs[self.name]["data"], inputs[self.name]["data_type"].type)
        if inputs[self.name]["data"].lower().endswith('.h5'):  # h5 file
            from h5py import File
            f = File(inputs[self.name]["data"], 'r')
            coll = np.array(list(f[list(f.keys())[0]]), dtype=inputs[self.name]["data_type"].type)  # read data
            # from first key
        elif inputs[self.name]["data"].lower().endswith('.csv'):  # csv file
            coll = list(np.genfromtxt(inputs[self.name]["data"], delimiter=',',
                                      dtype=inputs[self.name]["data_type"].type))
        # add data to queue
        self.data_queue.import_data(coll)
```

```
else:
    raise Exception("Input data representation should either be implicit (list) or a path to a csv file.")
```

output.py

```
import functools
import operator
import os
from typing import List

from dace.types import typeclass

import helper
from base_node_class import BaseKernelNodeClass
from bounded_queue import BoundedQueue


class Output(BaseKernelNodeClass):
    """
        The Output class is a subclass of the BaseKernelNodeClass and represents an Ouput node in the KernelChainGraph.
        Its purpose is to store data coming from the pipeline/dataflow design.
    """

    def __init__(self,
                 name: str,
                 data_type: typeclass,
                 dimensions: List[int],
                 data_queue=None) -> None:
        """
        Initializes the Output class with given initialization parameters.
        :param name: name of the output node
        :param data_type: data type of the data feed into output
        :param dimensions: global problem dimensions
        :param data_queue: dummy
        """
        # init superclass with queue of size: global problem size
        super().__init__(name=name, data_type=data_type, data_queue=BoundedQueue(name="output",
                                                                                 maxsize=functools.reduce(operator.mul,
                                                                                                          dimensions),
                                                                                 collection=[]))

    def reset_old_compute_state(self) -> None:
        """
        Reset compute-specific internal state (only for Kernel node).
        """
        pass  # nothing to do

    def try_read(self) -> None:
        """
        Read data from predecessor.
        """
        # check for single input
        assert len(self.inputs) == 1  # there should be only a single one
        for inp in self.inputs:
```

```
        # read data
        if self.inputs[inp]["delay_buffer"].try_peek_last() is not False and self.inputs[inp]["delay_buffer"]\
                .try_peek_last() is not None:
            self.data_queue.enqueue(self.inputs[inp]["delay_buffer"].dequeue())
            self.program_counter += 1
        elif self.inputs[inp]["delay_buffer"].try_peek_last() is not False:
            self.inputs[inp]["delay_buffer"].dequeue()  # remove bubble

def try_write(self) -> None:
    """
    Feed data to all successor channels (for Input and Kernel nodes)
    """
    pass  # nothing to do

def write_result_to_file(self,
                         input_config_name: str) -> None:
    """
    Write internal queue with computation result to the file results/INPUT_CONFIG_NAME/SELF.NAME_simulation.dat
    :param input_config_name: the config name, used to determine the save path
    """
    # join the paths
    output_folder = os.path.join("results", input_config_name)
    # create (recursively) directories
    os.makedirs(output_folder, exist_ok=True)
    # store the data
    helper.save_array(self.data_queue.export_data(), "{}/{}_{}.dat".format(output_folder, self.name, 'simulation'))
```

kernel.py

```
import functools
import operator
from typing import List, Dict

import dace.types

import helper
from base_node_class import BaseKernelNodeClass, BaseOperationNodeClass
from bounded_queue import BoundedQueue
from calculator import Calculator
from compute_graph import ComputeGraph
from compute_graph import Name, Num, Binop, Call, Output, Subscript, Ternary, Compare, UnaryOp


class Kernel(BaseKernelNodeClass):
    """
        The Kernel class is a subclass of the BaseKernelNodeClass and represents the actual kernel node in the
        KernelChainGraph. This class is able to read from predecessors, process it according to the stencil expression
        and write the result to the successor channels. In addition it analyses the buffer sizes and latencies of the
        computation according  to the defined latencies.
    """

    def __init__(self,
                 name: str,
                 kernel_string: str,
                 dimensions: List[int],
                 data_type: dace.types.typeclass,
                 boundary_conditions: Dict[str, Dict[str, str]],
                 plot_graph: bool = False,
                 verbose: bool = False) -> None:
        """

        :param name: name of the kernel
        :param kernel_string: mathematical expression representing the stencil computation
        :param dimensions: global dimensions / problem size (i.e. size of the input array
        :param data_type: data type of the result produced by this kernel
        :param boundary_conditions: dictionary of the boundary condition for each input channel/field
        :param plot_graph: flag indicating whether the underlying graph is being drawn
        :param verbose: flag for console output logging
        """
        # initialize the superclass
        super().__init__(name, BoundedQueue(name="dummy", maxsize=0), data_type)
        # store arguments
        self.kernel_string: str = kernel_string  # raw kernel string input
        self.dimensions: List[int] = dimensions  # input array dimensions [dimX, dimY, dimZ]
        self.boundary_conditions: Dict[str, Dict[str, str]] = boundary_conditions  # boundary_conditions[field_name]
        self.verbose = verbose
        # read static parameters from config
        self.config: Dict = helper.parse_json("kernel.config")
```

```
    self.calculator: Calculator = Calculator()
    # set simulator initial parameters
    self.all_available = False
    self.not_available = set()
    # analyze input
    self.graph: ComputeGraph = ComputeGraph()
    self.graph.generate_graph(kernel_string)  # generate the ast computation graph from the mathematicl expression
    self.graph.calculate_latency()  # calculate the latency in the compuation tree to find the critical path
    self.graph.determine_inputs_outputs()  # sort out input nodes (field accesses and constant values) and output
    # nodes
    self.graph.setup_internal_buffers()
    # set plot path (if plot is set to True)
    if plot_graph:
        self.graph.plot_graph(name + ".png")
    # init sim specific params
    self.var_map: Dict[
        str, float] = dict()  # mapping between variable names and its (current) value: var_map[var_name] =
    # var_value
    self.read_success: bool = False  # flag indicating if read has been successful from all input nodes (=> ready
    # to execute)
    self.exec_success: bool = False  # flag indicating if the execution has been successful
    self.result: float = float('nan')  # execution result of current iteration (see program counter)
    self.outputs: Dict[str, BoundedQueue] = dict()
    # output delay queue: for simulation of calculation latency, fill it up with bubbles
    self.out_delay_queue: BoundedQueue = BoundedQueue(name="delay_output",
                                                      maxsize=self.graph.max_latency + 1,
                                                      collection=[None] * self.graph.max_latency)
    # setup internal buffer queues
    self.internal_buffer: Dict[str, BoundedQueue] = dict()
    self.setup_internal_buffers()
    # this method takes care of the (falsely) executed kernel in case of not having a field access at [0,0,0]
    # present and the implication that there might be only fields out of bound s.t. there is a result produced,
    # but there should not be a result yet (see paper example ref# TODO)
    self.dist_to_center: Dict = dict()
    self.set_up_dist_to_center()
    self.center_reached = False
    # add performance metric fields
    self.max_del_buf_usage = dict()
    # for mean
    self.buf_usage_sum = dict()
    self.buf_usage_num = dict()
    self.init_metric = False
    self.PC_exec_start = helper.convert_3d_to_1d(self.dimensions, self.dimensions)  # upper bound
    self.PC_exec_end = 0  # lower bound

def print_kernel_performance(self):
    """
    Print performance metric data.
    """
```

```
    print("##############################")
    for input in set(self.inputs).union(set(self.outputs)):
        print("##############################")
        print("input buffer name: {}".format(input))
        print("max buffer usage: {}".format(self.max_del_buf_usage[input]))
        print("average buffer usage: {}".format(self.buf_usage_sum[input] / self.buf_usage_num[input]))
    print("total execution time (from first exec to last): {}".format(self.PC_exec_end - self.PC_exec_start))

def update_performance_metric(self):
    """
    Update buffer size values for performance evalution purpose.
    """
    # check if dict has been initialized
    if not self.init_metric:
        # init all keys
        for input in self.inputs:
            self.max_del_buf_usage[input] = 0
            self.buf_usage_num[input] = 0
            self.buf_usage_sum[input] = 0
        for output in self.outputs:
            self.max_del_buf_usage[output] = 0
            self.buf_usage_num[output] = 0
            self.buf_usage_sum[output] = 0
    # update maximum delay buf usage
    # inputs
    for input in self.inputs:
        buffer = self.inputs[input]
        self.max_del_buf_usage[input] = max(self.max_del_buf_usage[input],
                                            len([x for x in buffer['delay_buffer'].queue if x is not None]))
        self.buf_usage_num[input] += 1
        self.buf_usage_sum[input] += len([x for x in buffer['delay_buffer'].queue if x is not None])
    # outputs
    for output in self.outputs:
        buffer = self.outputs[output]
        self.max_del_buf_usage[output] = max(self.max_del_buf_usage[output],
                                             len([x for x in buffer['delay_buffer'].queue if x is not None]))
        self.buf_usage_num[output] += 1
        self.buf_usage_sum[output] += len([x for x in buffer['delay_buffer'].queue if x is not None])

def set_up_dist_to_center(self):
    """
    Computes for all fields/channels the distance from the furthest field access to the center of the stencil
    ([0,0,0,]).
    """
    for item in self.graph.accesses:
        furthest = max(self.graph.accesses[item])
        self.dist_to_center[item] = helper.dim_to_abs_val(furthest, self.dimensions)

def iter_comp_tree(self,
```

```
                node: BaseOperationNodeClass,
                index_relative_to_center=True,
                replace_negative_index=False,
                python_syntax=False) -> str:
"""
Iterate through the computation tree in order to generate the kernel string (according to some properties
e.g. relative to center or replace negative index.
:param node: current node in the tree
:param index_relative_to_center: indication wheter the zero index should be at the center of the stencil or the
furthest element
:param replace_negative_index: replace the negativ sign '-' by n in order to create variable names that are not
being split up by the python expression parser (Calculator)
:return: computation string of the subgraph
"""
# get predecessor list
pred = list(self.graph.graph.pred[node])
# differentiate cases for each node type
if isinstance(node, Binop):  # binary operation
    # extract expression elements
    lhs = pred[0]  # left hand side
    rhs = pred[1]  # right hand side
    # recursively compute the child string
    lhs_str = self.iter_comp_tree(lhs, index_relative_to_center, replace_negative_index, python_syntax)
    rhs_str = self.iter_comp_tree(rhs, index_relative_to_center, replace_negative_index, python_syntax)
    # return formatted string
    return "({} {} {})".format(lhs_str, node.generate_op_sym(), rhs_str)
elif isinstance(node, Call):  # function call
    # extract expression element
    expr = pred[0]
    # recursively compute the child string
    expr_str = self.iter_comp_tree(expr, index_relative_to_center, replace_negative_index, python_syntax)
    # return formatted string
    return "{}({})".format(node.name, expr_str)
elif isinstance(node, Name) or isinstance(node, Num):
    # return formatted string
    return str(node.name)  # variable name
elif isinstance(node, Subscript):
    # compute correct indexing according to the flag
    if index_relative_to_center:
        dim_index = node.index
    else:
        dim_index = helper.list_subtract_cwise(node.index, self.graph.max_index[node.name])
    # break down index from 3D (i.e. [X,Y,Z]) to 1D
    word_index = self.convert_3d_to_1d(dim_index)
    # replace negative sign if the flag is set
    if replace_negative_index and word_index < 0:
        return node.name + "[" + "n" + str(abs(word_index)) + "]"
    else:
        return node.name + "[" + str(word_index) + "]"
```

```
    elif isinstance(node, Ternary):  # ternary operator of the form true_expr if comp else false_expr
        # extract expression elements
        compare = [x for x in pred if type(x) == Compare][0]  # comparison
        lhs = [x for x in pred if type(x) != Compare][0]  # left hand side
        rhs = [x for x in pred if type(x) != Compare][1]  # right hand side
        # recursively compute the child string
        compare_str = self.iter_comp_tree(compare, index_relative_to_center, replace_negative_index, python_syntax)
        lhs_str = self.iter_comp_tree(lhs, index_relative_to_center, replace_negative_index, python_syntax)
        rhs_str = self.iter_comp_tree(rhs, index_relative_to_center, replace_negative_index, python_syntax)
        # return formatted string
        if python_syntax:
            return "(({}) if ({}) else ({}))".format(lhs_str, compare_str, rhs_str)
        else:  # C++ ternary operator syntax
            return "(({}) ? ({}) : ({}))".format(compare_str, lhs_str, rhs_str)
    elif isinstance(node, Compare):  # comparison
        # extract expression element
        lhs = pred[0]
        rhs = pred[1]
        # recursively compute the child string
        lhs_str = self.iter_comp_tree(lhs, index_relative_to_center, replace_negative_index, python_syntax)
        rhs_str = self.iter_comp_tree(rhs, index_relative_to_center, replace_negative_index, python_syntax)
        # return formatted string
        return "{} {} {}".format(lhs_str, str(node.name), rhs_str)
    elif isinstance(node, UnaryOp):  # unary operations e.g. negation
        # extract expression element
        expr = pred[0]
        # recursively compute the child string
        expr_str = self.iter_comp_tree(node=expr, index_relative_to_center=index_relative_to_center,
                                       replace_negative_index=replace_negative_index, python_syntax=python_syntax)
        # return formatted string
        return "({}{})".format(node.generate_op_sym(), expr_str)
    else:
        raise NotImplementedError("iter_comp_tree is not implemented for node type {}".format(type(node)))

def generate_relative_access_kernel_string(self,
                                           relative_to_center=True,
                                           replace_negative_index=False,
                                           python_syntax=False) -> str:
    """
    Generates the relative (either to the center or to the furthest field access) access kernel string which
    is necessary for the code generator HLS tool.
    :param relative_to_center: if true, the center is at zero, otherwise the furthest access is at zero
    :param replace_negative_index: if true, all negative access signs e.g. arrA_-20 gets replaced by n e.g.
    arrA_n20 in order to be correctly recognised as a single variable name.
    :return: the generated relative access kernel string
    """
    # format: 'res = vdc[index1] + vout[index2]'
    res = []
    # treat named nodes
```

```
    for n in self.graph.graph.nodes:
        if isinstance(n, Name):
            res.append(n.name + " = " + self.iter_comp_tree(
                list(self.graph.graph.pred[n])[0], relative_to_center, replace_negative_index, python_syntax))
    # treat output node(s)
    output_node = [
        n for n in self.graph.graph.nodes if isinstance(n, Output)
    ]
    if len(output_node) != 1:
        raise Exception("Expected a single output node")
    output_node = output_node[0]
    # concatenate the expressions
    res.append("res = " + self.iter_comp_tree(node=list(self.graph.graph.pred[output_node])[0],
                                              index_relative_to_center=relative_to_center,
                                              replace_negative_index=replace_negative_index,
                                              python_syntax=python_syntax))
    return "; ".join(res)

def reset_old_compute_state(self) -> None:
    """
    Reset the internal kernel simulator state in order to be prepared for the next iteration.
    """
    self.var_map = dict()
    self.read_success = False
    self.exec_success = False
    self.result = None

def convert_3d_to_1d(self,
                     index: List[int]) -> int:
    """
    Convert [i,j,k] to flat 1D array index using the given dimensions [dimX, dimY, dimZ]
    :param index: index array to be converted to 1D
    :return: scalar value of the computation i*dimY*dimZ + j*dimZ + k = (i*dimY + j)*dimZ + k
    """
    # do computation: index = i*dimY*dimZ + j*dimZ + k = (i*dimY + j)*dimZ + k if the array is not empty
    if not index:
        return 0  # empty list
    return helper.dim_to_abs_val(index, self.dimensions)

def remove_duplicate_accesses(self,
                              inp: List) -> List:
    """
    Remove duplicate accesses of the given input array.
    :param inp: List with duplicates.
    :return: List without duplicates.
    """
    tuple_set = set(tuple(row) for row in inp)
    return [list(t) for t in tuple_set]
```

```
def setup_internal_buffers(self) -> None:
    """
    Create and split the internal buffers according to the pipline model (see paper example ref# TODO)
    :return:
    """
    # remove duplicate accesses
    for item in self.graph.accesses:
        self.graph.accesses[item] = self.remove_duplicate_accesses(self.graph.accesses[item])
    # slice the internal buffer into junks of accesses
    for buf_name in self.graph.buffer_size:
        # create empty list and sort the accesses according to their relative position
        self.internal_buffer[buf_name]: List[BoundedQueue] = list()
        list.sort(self.graph.accesses[buf_name], reverse=True)
        # split according to the cases
        if len(self.graph.accesses[buf_name]) == 0:  # empty list
            pass
        elif len(self.graph.accesses[buf_name]) == 1:  # single entry list
            # this line would add an additional internal buffer for fields that only have a single access
            self.internal_buffer[buf_name].append(BoundedQueue(name=buf_name, maxsize=1, collection=[None]))
        else:  # many entry list
            # iterate through all of them and split them into correct sizes
            itr = self.graph.accesses[buf_name].__iter__()
            pre = itr.__next__()
            for item in itr:
                curr = item
                # calculate size of buffer
                diff = abs(helper.dim_to_abs_val(helper.list_subtract_cwise(pre, curr), self.dimensions))
                if diff == 0:  # two accesses on same field
                    pass
                else:
                    self.internal_buffer[buf_name].append(
                        BoundedQueue(name=buf_name, maxsize=diff, collection=[None] * diff))
                pre = curr

def buffer_position(self,
                    access: BaseKernelNodeClass) -> int:
    """
    Computes the offset position within the buffer list
    :param access: the access index we want to know the buffer position
    :return: the offset from the access
    """
    return self.convert_3d_to_1d(self.graph.min_index[access.name]) - self.convert_3d_to_1d(access.index)

def index_to_ijk(self,
                 index: List[int]):
    """
    Creates a string of the access (for variable name generation).
    :param index: access
    :return: created string
```

```
    """
    # current implementation only supports 3 dimension (default)
    if len(index) == 3:
        """
        # v1:
        return "[i{},j{},k{}]".format(
            "" if index[0] == 0 else "+{}".format(index[0]),
            "" if index[1] == 0 else "+{}".format(index[1]),
            "" if index[2] == 0 else "+{}".format(index[2])
        )
        # v2:
        return "_{}_{}_{}".format(index[0], index[1], index[2])
        """
        # compute absolute index
        ind = helper.convert_3d_to_1d(self.dimensions, index)
        # return formatted string
        return "_{}".format(ind) if ind >= 0 else "_n{}".format(abs(ind))
    else:
        raise NotImplementedError(
            "Method index_to_ijk has not been implemented for |indices|!=3, here: |indices|={}".format(len(index)))

def buffer_number(self,
                  node: Subscript):
    """
    Computes the index within the internal buffer array for accessing the input node.
    :param node: input node
    :return: index (-1: delay buffer, >= 0: internal buffer index)
    """
    # select all matching inputs
    selected = [x.index for x in self.graph.inputs if x.name == node.name]
    # remove duplicates
    selected_unique = self.remove_duplicate_accesses(selected)
    # sort them to have them ordered by the access
    ordered = sorted(selected_unique, reverse=True)
    # get the position within the sorted list
    result = ordered.index(node.index)
    return result - 1

def get_global_kernel_index(self) -> List[int]:
    """
    Return the current position (simulator, program counter) within the comutation as a list of the form
    [i,j,k].
    :return: current global kernel position as [i,j,k]
    """
    # get dimensions and PC
    index = self.dimensions
    number = self.program_counter
    # convert the absolute value (PC) to its corresponding position in the given 3D space.
    n = len(index)
```

```
    all_dim = functools.reduce(operator.mul, index, 1) // index[0]  # integer arithmetic
    output = list()
    for i in range(1, n + 1):
        output.append(number // all_dim)
        number -= output[-1] * all_dim
        if i < n:
            all_dim = all_dim // index[i]
    return output

def is_out_of_bound(self,
                    index: List[int]) -> bool:
    """
    Checks whether the current access is within bounds or not.
    :param index: access index
    :return: true: within bounds, false: otherwise
    """
    # check all dimensions boundary
    for i in range(len(index)):
        if index[i] < 0 or index[i] >= self.dimensions[i]:
            return True
    return False

def get_data(self,
             inp: Subscript,
             global_index: List[int],
             relative_index: List[int]):
    """
    Returns data of current stencil access (could be real data or boundary condition)
    :param inp: array field access
    :param global_index: center location of current stencil
    :param relative_index: offset from center of stencil
    :return: data
    """
    # get the access index
    access_index = helper.list_add_cwise(global_index, relative_index)
    """
        Boundary Condition
    """
    # check if it is within bounds
    if self.is_out_of_bound(access_index):
        if self.boundary_conditions[inp.name]["type"] == "constant":
            return self.boundary_conditions[inp.name]["value"]
        elif self.boundary_conditions[inp.name]["type"] == "copy":
            raise NotImplementedError("Copy boundary conditions have not been implemented yet.")
        else:
            raise NotImplementedError("We currently do not support boundary conditions of type {}".format(
                self.boundary_conditions[inp.name]["type"]))
    """
        Data Access
```

```
    """
    # get index position within the buffers
    pos = self.buffer_number(inp)
    if pos == -1:  # delay buffer
        return self.inputs[inp.name]["delay_buffer"].try_peek_last()
    elif pos >= 0:  # internal buffer
        return self.inputs[inp.name]["internal_buffer"][pos].try_peek_last()

def test_availability(self):
    """
    Check if all accesses are available (=> ready for execution). In addition to that, the method delivers all
    accesses that are not available yet.
    :return: true: all available, false: otherwise
    """
    # set initial value and init set
    all_available = True
    self.not_available = set()
    # iterate through all inputs
    for inp in self.graph.inputs:
        # case split for types
        if isinstance(inp, Num):  # numerals are always available
            pass
        elif len(self.inputs[inp.name]['internal_buffer']) == 0:  # no internal buffer
            pass
        elif isinstance(inp, Subscript):  # normal subscript access
            # get current internal state position in [i,j,k] format
            gki = self.get_global_kernel_index()
            # check bound, out of bound is handled by the boundary condition automatically (always available for
            # constant)
            if self.is_out_of_bound(helper.list_add_cwise(inp.index, gki)):
                pass
            else:  # within bounds
                # get position and check if the value (not None) is available
                index = self.buffer_number(inp)
                if index == -1:  # delay buffer
                    if self.inputs[inp.name]['delay_buffer'].try_peek_last() is None or \
                            self.inputs[inp.name]['delay_buffer'].try_peek_last() is False:
                        all_available = False
                        self.not_available.add(inp.name)
                elif 0 <= index < len(self.inputs[inp.name]['internal_buffer']):  # internal buffer
                    if self.inputs[inp.name]['internal_buffer'][index].try_peek_last() is False \
                            or self.inputs[inp.name]['internal_buffer'][index].try_peek_last() is None:
                        all_available = False
                        self.not_available.add(inp.name)
                else:
                    raise Exception("index out of bound: {}".format(index))

    return all_available
```

```
def move_forward(self,
                 items: Dict[str, Dict]) -> None:
    """
    Move all items within the internal and delay buffer one element forward.
    :param items:
    :return:
    """
    # move all forward
    for name in items:
        if len(items[name]['internal_buffer']) == 0:  # no internal buffer
            pass
        elif len(self.inputs[name]['internal_buffer']) == 1:  # single internal buffer
            items[name]['internal_buffer'][0].dequeue()
            items[name]['internal_buffer'][0].enqueue(items[name]['delay_buffer'].dequeue())
        else:  # many internal buffers
            # iterate over them and move all one forward
            index = len(items[name]['internal_buffer']) - 1
            pre = items[name]['internal_buffer'][index - 1]
            next = items[name]['internal_buffer'][index]
            next.dequeue()
            while index > 0:
                next.enqueue(pre.dequeue())
                next = pre
                index -= 1
                pre = items[name]['internal_buffer'][index - 1]
            items[name]['internal_buffer'][0].enqueue(items[name]['delay_buffer'].dequeue())

def decrement_center_reached(self):
    """
    Decrement counter for reaching the center. As soon as this counter reaches zero, the computed output values
    are valid and should be forwarded to the successors channels.
    """
    # decrement all
    for item in self.dist_to_center:
        if self.inputs[item]['delay_buffer'].try_peek_last() is not None:
            self.dist_to_center[item] -= 1

def try_read(self) -> bool:
    """
    This is the implementation of the kernel reading functionality of the simulator. It tries to read from all
    input channels and indicates if this has been done with success.
    """
    # check if all inputs are available
    self.all_available = self.test_availability()
    # get all values and put them into the variable map
    if self.all_available:
        for inp in self.graph.inputs:
            # read inputs into var_map
            if isinstance(inp, Num):  # case numerals
```

```
                self.var_map[inp.name] = float(inp.name)
            elif isinstance(inp, Name):  # case variable names
                # get value from internal_buffer
                try:
                    # check for duplicate
                    if not self.var_map.__contains__(inp.name):
                        self.var_map[inp.name] = self.internal_buffer[inp.name].peek(self.buffer_position(inp))
                except Exception as ex:  # do proper diagnosis
                    self.diagnostics(ex)
            elif isinstance(inp, Subscript):  # case array accesses
                # get value from internal buffer
                try:
                    name = inp.name + self.index_to_ijk(inp.index)
                    if not self.var_map.__contains__(name):
                        self.var_map[name] = self.get_data(inp=inp,
                                                           global_index=self.get_global_kernel_index(),
                                                           relative_index=inp.index)
                except Exception as ex:  # do proper diagnosis
                    self.diagnostics(ex)
    # set kernel flag indicating the the read has been successful
    self.read_success = self.all_available
    # test center reached
    self.decrement_center_reached()
    self.center_reached = True
    for item in self.dist_to_center:
        if self.dist_to_center[item] >= 0:
            self.center_reached = False
    # either move all inputs forward or those that are not available yet
    if self.center_reached:
        if self.all_available:
            self.move_forward(self.inputs)
        else:
            not_avail_dict = dict()
            for item in self.not_available:
                not_avail_dict[item] = self.inputs[item]
            self.move_forward(not_avail_dict)
    else:
        not_reached_dict = dict()
        for item in self.dist_to_center:
            if self.dist_to_center[item] >= 0:
                not_reached_dict[item] = self.inputs[item]
        self.move_forward(not_reached_dict)
    return self.all_available

def try_execute(self):
    """
    This is the implementation of the kernel execution functionality of the simulator. It executes the stencil
    computation for the current variable mapping that was set up by the try_read() function.
    """
```

```
    # check if read has been succeeded
    if self.center_reached and self.read_success and \
            0 <= self.program_counter < functools.reduce(operator.mul, self.dimensions, 1):
        # execute calculation
        try:
            # get computation string
            computation = self.generate_relative_access_kernel_string(relative_to_center=True,
                                                                      replace_negative_index=True,
                                                                      python_syntax=True) \
                .replace("[", "_").replace("]", "").replace(" ", "")
            # compute result and
            self.result = self.data_type(self.calculator.eval_expr(self.var_map, computation))
            # write result to latency-simulating buffer
            self.out_delay_queue.enqueue(self.result)
            # update performance metric
            self.PC_exec_start = min(self.PC_exec_start, self.program_counter)
            self.PC_exec_end = max(self.PC_exec_end, self.program_counter)
            # increment the program counter
            self.program_counter += 1
        except Exception as ex:  # do proper diagnosis upon an exception
            self.diagnostics(ex)
    else:
        # write bubble to latency-simulating buffer
        self.out_delay_queue.enqueue(None)

def try_write(self):
    """
    This is the implementation of the kernel write functionality of the simulator. It writes the output element to
    its successor channels.
    """
    # read last element of the delay queue
    data = self.out_delay_queue.dequeue()
    # write result to all output queues
    for outp in self.outputs:
        try:
            self.outputs[outp]["delay_buffer"].enqueue(data)  # use delay buffer to be consistent with others,
            # delay buffer is used to write to the output data queue here
        except Exception as ex:  # do proper diagnosis upon an exception
            self.diagnostics(ex)

def diagnostics(self,
                ex: Exception) -> None:
    """
    Interface for error overview reporting (gets called in case of an exception)

    - goal:
            - get an overview over the whole stencil chain state in case of an error
                - maximal and current size of all buffers
                - type of phase (saturation/execution)
```

```
                - efficiency (#execution cycles / #total cycles)
        :param ex: the exception that arose
        """
        print("#####################################")
        print("Diagnosis output of kernel {}".format(self.name))
        print("Program Counter: {}".format(self.program_counter))
        print("All inputs available? {}".format(self.all_available))
        print("Center reached? {}".format(self.center_reached))
        print("Exception traceback:")
        if ex is not None:
            import traceback
            try:
                raise ex
            except Exception:
                print(traceback.format_exc())  # inputs
        for input in self.inputs:
            buffer = self.inputs[input]
            print("Buffer info from input {}".format(input))
            # delay buffer
            print("Delay buffer max size: {}, current size: {}".format(buffer['delay_buffer'].maxsize,
                                                                       buffer['delay_buffer'].size()))
            print("Delay buffer data: {}".format(buffer['delay_buffer'].queue))
            # internal buffer
            data = list(map(lambda x: x.queue, buffer['internal_buffer']))
            print("Internal buffer data: {}".format(data))
        # latency sim buffer
        print("Latency simulation buffer data: {}".format(self.out_delay_queue.queue))
        # output
        for output in self.outputs:
            buffer = self.outputs[output]
            print("Buffer info from output {}".format(output))
            # delay buffer
            print("Delay buffer max size: {}, current size: {}".format(buffer['delay_buffer'].maxsize,
                                                                       buffer['delay_buffer'].size()))
            print("Delay buffer data: {}".format(buffer['delay_buffer'].queue))
            # internal buffer
            data = list(map(lambda x: x.queue, buffer['internal_buffer']))
            print("Internal buffer data: {}".format(data))


if __name__ == "__main__":
    """
        simple test kernel for debugging
    """
    # global dimensions
    dim = [100, 100, 100]
    # instantiate kernel
    kernel = Kernel(name="dummy",
                    kernel_string="res = a[i+1,j+1,k+1] + a[i+1,j,k] + a[i-1,j-1,k-1] + a[i+1,j+1,k] + (-a[i,j,k])",
```

```
                dimensions=dim,
                boundary_conditions={"a": {
                    "type": "constant",
                    "value": 0.0}},
                data_type=dace.types.float64)
print("Kernel string conversion:")
print("dimensions are: {}".format(dim))
print(kernel.kernel_string)
print(kernel.generate_relative_access_kernel_string(relative_to_center=False))
print()
```

kernel_chain_graph.py

```
import argparse
import functools
import operator
import os
import re
from typing import List, Dict

import networkx as nx

import helper
from bounded_queue import BoundedQueue
from input import Input
from kernel import Kernel
from log_level import LogLevel
from output import Output


class KernelChainGraph:
    """
    The KernelChainGraph class represents the whole pipelined data flow graph consisting of input nodes (real data input
    arrays, kernel nodes and output nodes (storing the result of the computation).
    """

    def __init__(self,
                 path: str,
                 plot_graph: bool = False,
                 log_level: int = 0) -> None:
        """
        Create new KernelChainGraph with given initialization parameters.
        :param path: path to the input file
        :param plot_graph: flag indication whether or not to produce the graphical graph representation
        :param log_level: flag for console output logging
        """
        if log_level >= LogLevel.BASIC.value:
            print("Initialize KernelChainGraph.")
        # set parameters
        # absolute path
        self.path: str = os.path.join(os.path.dirname(os.path.dirname(os.path.realpath(__file__))), path)  # get valid
        self.log_level: int = log_level
        # init internal fields
        self.inputs: Dict[str, Dict[str, str]] = dict()  # input data
        self.outputs: List[str] = list()  # name of the output fields
        self.dimensions: List[int] = list()  # global problem size
        self.program: Dict[str, Dict[str, Dict[str, Dict[
            str, str]]]] = dict()  # mathematical stencil expressionos:program[stencil_name] = stencil expression
        self.kernel_latency = None  # critical path latency
        self.channels: Dict[str, BoundedQueue] = dict()  # each channel is an edge between two nodes
        self.graph: nx.DiGraph = nx.DiGraph()  # data flow graph
        self.input_nodes: Dict[str, Kernel] = dict()  # Input nodes of the graph
```

```
    self.output_nodes: Dict[str, Kernel] = dict()  # Output nodes of the graph
    self.kernel_nodes: Dict[str, Kernel] = dict()  # Kernel nodes of the graph
    self.config = helper.parse_json("stencil_chain.config")
    self.name = os.path.splitext(os.path.basename(self.path))[0]  # name
    self.kernel_dimensions = -1  # 2: 2D, 3: 3D
    # trigger all internal calculations
    if self.log_level >= LogLevel.BASIC.value:
        print("Read input config files.")
    self.import_input()  # read input config file
    if self.log_level >= LogLevel.BASIC.value:
        print("Create all kernels.")
    self.create_kernels()  # create all kernels
    if self.log_level >= LogLevel.BASIC.value:
        print("Compute kernel latencies.")
    self.compute_kernel_latency()  # compute their latencies
    if self.log_level >= LogLevel.BASIC.value:
        print("Connect kernels.")
    self.connect_kernels()  # connect them in the graph
    if self.log_level >= LogLevel.BASIC.value:
        print("Compute delay buffer sizes.")
    self.compute_delay_buffer()  # compute the delay buffer sizes
    if self.log_level >= LogLevel.BASIC.value:
        print("Add channels to the graph edges.")
    self.add_channels()  # add all channels (internal buffer and delay buffer) to the edges of the graph
    # plot kernel graphs if flag set to true
    if plot_graph:
        if self.log_level >= LogLevel.BASIC.value:
            print("Plot kernel chain graph.")
        # plot kernel chain graph
        self.plot_graph()
        # plot all compute graphs
        if self.log_level >= LogLevel.BASIC.value:
            print("Plot computation graph of each kernel.")
        for compute_kernel in self.kernel_nodes:
            self.kernel_nodes[compute_kernel].graph.plot_graph()
    # print sin/cos/tan latency warning
    for kernel in self.program:
        if "sin" in self.program[kernel]['computation_string'] or "cos" in self.program[kernel][
            'computation_string'] or "tan" in self.program[kernel]['computation_string']:
            print("Warning: Computation contains sinusoidal functions with experimental latency values.")
    # print report for moderate and high verbosity levels
    if self.log_level >= LogLevel.MODERATE.value:
        self.report(self.name)

def plot_graph(self,
               save_path: str = None) -> None:
    """
    Draw the networkx (library) KernelChainGraph graphically.
    :param save_path: path to save the image
```

kernel_chain_graph.py

```
"""
# create drawing area
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
fig.set_size_inches(25, 25)
ax.set_axis_off()
# generate positions of the node (for pretty visualization)
positions = nx.nx_pydot.graphviz_layout(self.graph, prog='dot')
# divide nodes into different lists for colouring purpose
nums = list()
names = list()
ops = list()
outs = list()
# add nodes to the corresponding list
for node in self.graph.nodes:
    if isinstance(node, Kernel):
        ops.append(node)
    elif isinstance(node, Input):
        names.append(node)
    elif isinstance(node, Output):
        outs.append(node)
# create dictionary of labels
labels = dict()
for node in self.graph.nodes:
    labels[node] = node.generate_label()
# add nodes and edges with distinct colours and shapes
nx.draw_networkx_nodes(
    self.graph,
    positions,
    nodelist=names,
    node_color='orange',
    node_size=3000,
    node_shape='s',
    edge_color='black')
nx.draw_networkx_nodes(
    self.graph,
    positions,
    nodelist=outs,
    node_color='green',
    node_size=3000,
    node_shape='s')
nx.draw_networkx_nodes(
    self.graph,
    positions,
    nodelist=nums,
    node_color='#007acc',
    node_size=3000,
    node_shape='s')
nx.draw_networkx(
```

```
        self.graph,
        positions,
        nodelist=ops,
        node_color='red',
        node_size=3000,
        node_shape='o',
        font_weight='bold',
        font_size=16,
        edge_color='black',
        arrows=True,
        arrowsize=36,
        arrowstyle='-|>',
        width=6,
        linwidths=1,
        with_labels=False)
    nx.draw_networkx_labels(
        self.graph,
        positions,
        labels=labels,
        font_weight='bold',
        font_size=16)
    # save plot to file if save_path has been specified
    if save_path is not None:
        fig.savefig(save_path)
    else:
        # plot it
        fig.show()

def connect_kernels(self) -> None:
    """
    Connect the nodes to a directed acyclic graph by matching the input name with kernel names.
    """
    # loop over all node tuples
    for src in self.graph.nodes:
        for dest in self.graph.nodes:
            if src is not dest:  # skip src == dest case
                if isinstance(src, Kernel) and isinstance(dest, Kernel):  # case: KERNEL -> KERNEL
                    for inp in dest.graph.inputs:
                        if src.name == inp.name:
                            # add edge
                            self.graph.add_edge(src, dest, channel=None)
                            break
                elif isinstance(src, Input) and isinstance(dest, Kernel):  # case: INPUT -> KERNEL
                    for inp in dest.graph.inputs:
                        if src.name == inp.name:
                            # add edge
                            self.graph.add_edge(src, dest, channel=None)
                            break
                elif isinstance(dest, Output):  # case: INPUT/KERNEL -> OUTPUT
```

```
                if src.name == dest.name:
                    # add edge
                    self.graph.add_edge(src, dest, channel=None)
            else:
                # pass all other source/destination pairs
                pass

def add_channels(self) -> None:
    """
    Assemble channels (internal buffers and delay buffer) and add them to src, dest and the global dict.
    """
    # create initial empty dictionary
    self.channels = dict()
    # loop over all node tuples
    for src in self.graph.nodes:
        for dest in self.graph.nodes:
            if src is not dest:  # skip src == dest
                if isinstance(src, Kernel) and isinstance(dest, Kernel):  # case: KERNEL -> KERNEL
                    for inp in dest.graph.inputs:
                        if src.name == inp.name:
                            # create channel
                            name = src.name + "_" + dest.name
                            channel = {
                                "name": name,
                                "delay_buffer": self.kernel_nodes[dest.name].delay_buffer[src.name],
                                "internal_buffer": dest.internal_buffer[src.name],
                                "data_type": src.data_type
                            }
                            # add channel reference to global channel dictionary
                            self.channels[name] = channel
                            # add channel to both endpoints
                            src.outputs[dest.name] = channel
                            dest.inputs[src.name] = channel
                            # add channel to edge
                            self.graph[src][dest]['channel'] = channel
                            break
                elif isinstance(src, Input) and isinstance(dest, Kernel):  # case: INPUT -> KERNEL
                    for inp in dest.graph.inputs:
                        if src.name == inp.name:
                            # create channel
                            name = src.name + "_" + dest.name
                            channel = {
                                "name": name,
                                "delay_buffer": self.kernel_nodes[dest.name].delay_buffer[src.name],
                                "internal_buffer": dest.internal_buffer[src.name],
                                "data_type": src.data_type
                            }
                            # add channel reference to global channel dictionary
                            self.channels[name] = channel
```

```
                        # add channel to both endpoints
                        src.outputs[dest.name] = channel
                        dest.inputs[src.name] = channel
                        # add to edge
                        self.graph[src][dest]['channel'] = channel
                        break
                elif isinstance(dest, Output):  # case: INPUT/KERNEL -> OUTPUT
                    if src.name == dest.name:
                        # create channel
                        name = src.name + "_" + dest.name
                        channel = {
                            "name": name,
                            "delay_buffer": self.output_nodes[dest.name].delay_buffer[src.name],
                            "internal_buffer": {},
                            "data_type": src.data_type
                        }
                        # add channel reference to global channel dictionary
                        self.channels[name] = channel
                        # add channel to both endpoints
                        src.outputs[dest.name] = channel
                        dest.inputs[src.name] = channel
                        # add to edge
                        self.graph[src][dest]['channel'] = channel
                else:
                    # pass all other source/destination pairs
                    pass

    def import_input(self) -> None:
        """
        Read all sections of the program input file.
        """
        inp = helper.parse_json(self.path)
        # get dimensions
        self.kernel_dimensions = len(inp["dimensions"])
        if self.kernel_dimensions == 1:  # 1D
            self.program = inp["program"]
            for entry in self.program:
                self.program[entry]["computation_string"] = \
                    self.program[entry]["computation_string"].replace("[", "[i,j,")  # add two extra indices
            self.inputs = inp["inputs"]
            self.outputs = inp["outputs"]
            self.dimensions = [1, 1] + inp["dimensions"]  # add two extra dimensions
        elif self.kernel_dimensions == 2:  # 2D
            self.program = inp["program"]
            for entry in self.program:
                self.program[entry]["computation_string"] = self.program[entry]["computation_string"]\
                    .replace("[","[i,")  # add extra index
            self.inputs = inp["inputs"]
            self.outputs = inp["outputs"]
```

```
        self.dimensions = [1] + inp["dimensions"]  # add extra dimension
    else:  # 3D
        self.program = inp["program"]
        self.inputs = inp["inputs"]
        self.outputs = inp["outputs"]
        self.dimensions = inp["dimensions"]

def total_elements(self) -> int:
    """
    Reduction of the global problem size to a single scalar.
    :return: the global problem size as a scalar value
    """
    return functools.reduce(operator.mul, self.dimensions, 1)  # foldl (*) 1 [...]

def create_kernels(self) -> None:
    """
    Create the kernels and add them to the networkx (library) graph.
    """
    # create all kernel objects and add them to the graph
    self.kernel_nodes = dict()
    for kernel in self.program:
        new_node = Kernel(name=kernel,
                          kernel_string=str(self.program[kernel]['computation_string']),
                          dimensions=self.dimensions,
                          data_type=self.program[kernel]['data_type'],
                          boundary_conditions=self.program[kernel]['boundary_condition'])
        self.graph.add_node(new_node)
        self.kernel_nodes[kernel] = new_node
    # create all input nodes (without data, we will add data in the simulator if necessary)
    self.input_nodes = dict()
    for inp in self.inputs:
        new_node = Input(name=inp,
                         data_type=self.inputs[inp]["data_type"],
                         data_queue=BoundedQueue(name=inp,
                                                 maxsize=self.total_elements(),
                                                 collection=[None] * self.total_elements()))
        self.input_nodes[inp] = new_node
        self.graph.add_node(new_node)
    # create all output nodes
    self.output_nodes = dict()
    for out in self.outputs:
        new_node = Output(name=out,
                          data_type=self.program[out]["data_type"],
                          dimensions=self.dimensions,
                          data_queue=BoundedQueue(name="dummy", maxsize=0))
        self.output_nodes[out] = new_node
        self.graph.add_node(new_node)

def compute_kernel_latency(self) -> None:
```

kernel_chain_graph.py

```
    """
    Fill global dictionary of the individual kernel critical computation paths.
    """
    # create dict
    self.kernel_latency = dict()
    # compute kernel latency of each kernel
    for kernel in self.kernel_nodes:
        self.kernel_latency[kernel] = self.kernel_nodes[kernel].graph.max_latency

def at_least_one(self,
                 value: int) -> int:
    """
    This function returns the input value or at least one if it is less.
    :param value: input value
    :return: at least 1
    """
    return value if value > 0 else 1

def compute_delay_buffer(self) -> None:
    """
    Computes the delay buffer sizes in the graph by propagating all paths from the input arrays to the successors in
    topological order. Delay buffer entries should be of the format: kernel.input_paths:{
                                                                        "in1": [[a,b,c, pred1], [d,e,f, pred2],
                                                                        ...],
                                                                        "in2": [ ... ],
                                                                        ...
                                                                    }
    where inX are input arrays to the stencil chain and predY are the kernel predecessors/inputs
    """
    # get topological order for top-down walk through of the graph
    try:
        order = nx.topological_sort(self.graph)
    except nx.exception.NetworkXUnfeasible:
        raise ValueError("Cycle detected, cannot be sorted topologically!")
    # go through all nodes
    for node in order:
        # process delay buffer (no additional delay buffer will appear because of the topological order)
        for inp in node.input_paths:
            # compute maximum delay size per input
            max_delay = max(node.input_paths[inp])
            max_delay[2] += 1  # add an extra delay cycle for the processing in the kernel node
            # loop over all inputs and set their size relative to the max size to have data ready at the exact
            # same time
            for entry in node.input_paths[inp]:
                name = entry[-1]
                max_size = helper.convert_3d_to_1d(self.dimensions,
                                                   helper.list_subtract_cwise(max_delay[:-1], entry[:-1]))
                node.delay_buffer[name] = BoundedQueue(name=name, maxsize=max_size)
                node.delay_buffer[name].import_data([None] * node.delay_buffer[name].maxsize)
```

kernel_chain_graph.py

```
        # set input node delay buffers to 1
        if isinstance(node, Input):
            node.delay_buffer = BoundedQueue(name=node.name, maxsize=1, collection=[None])
        # propagate the path lengths (from input arrays over all ways) to the successors
        for succ in self.graph.successors(node):
            # add input node to all as direct input (=0 delay buffer)
            if isinstance(node, Input):
                # add emtpy list dictionary entry for enabling list append()
                if node.name not in succ.input_paths:
                    succ.input_paths[node.name] = []
                successor = [0] * len(self.dimensions)
                successor = successor + [node.name]
                succ.input_paths[node.name].append(successor)
            # add kernel node to all, but calculate the length first (predecessor + delay + internal, ..)
            elif isinstance(node, Kernel):  # add KERNEL

                # add latency, internal_buffer, delay_buffer
                internal_buffer = [0] * 3
                for item in node.graph.accesses:
                    internal_buffer = max(node.graph.accesses[item]) if max(
                        node.graph.accesses[item]) > internal_buffer else internal_buffer
                # latency
                latency = self.kernel_nodes[node.name].graph.max_latency
                # compute delay buffer and create entry
                for entry in node.input_paths:
                    # the first entry has to initialize the structure
                    if entry not in succ.input_paths:
                        succ.input_paths[entry] = []
                    # compute the actual delay buffer
                    delay_buffer = max(node.input_paths[entry][:])
                    # merge them together
                    total = [
                        i + d
                        for i, d in zip(internal_buffer, delay_buffer)
                    ]
                    # add the latency too
                    total[-1] += latency
                    total.append(node.name)
                    # add entry to paths
                    succ.input_paths[entry].append(total)

            else:  # NodeType.OUTPUT: do nothing
                continue

    def compute_critical_path_dim(self) -> List[int]:
        """
        Computes the max latency critical path through the graph in dimensional format.
        Note: Since we know the output nodes as well as the path lengths the critical path is just
        max { latency(node) + max { path_length(node) | node in output nodes }
```

```
    :return:
    """
    # init critical path length with zero
    critical_path_length = [0] * len(self.dimensions)
    # loop through all and update if our path with the extra kernel latency is larger then the largest that is
    # already stored
    for output in self.outputs:
        a = self.kernel_nodes[output].graph.max_latency
        b = max(self.kernel_nodes[output].input_paths)
        c = max(self.kernel_nodes[output].input_paths[b])
        c[2] += a
        critical_path_length = max(critical_path_length, c)
    # return final result
    return c[:-1]

def compute_critical_path(self) -> int:
    """
    Computes the max latency critical path through the graph in scalar format.
    """
    return helper.dim_to_abs_val(self.compute_critical_path_dim(), self.dimensions)

def report(self, name):
    print("Report of {}\n".format(name))

    print("dimensions of data array: {}\n".format(self.dimensions))

    print("channel info:")
    for u, v, channel in self.graph.edges(data='channel'):
        if channel is not None:
            print("internal buffers:\n {}".format(channel["internal_buffer"]))
            print("delay buffers:\n {}".format(channel["delay_buffer"]))
    print()

    print("field access info:")
    for node in self.kernel_nodes:
        print("node name: {}, field accesses: {}".format(node, self.kernel_nodes[node].graph.accesses))
    print()

    print("internal buffer size info:")
    for node in self.kernel_nodes:
        print("node name: {}, internal buffer size: {}".format(node,
                                                               self.kernel_nodes[node].graph.buffer_size))
    print()

    print("internal buffer chunks info:")
    for node in self.kernel_nodes:
        print("node name: {}, internal buffer chunks: {}".format(node,
                                                                 self.kernel_nodes[node].internal_buffer))
    print()
```

kernel_chain_graph.py

```
print("delay buffer size info:")
for node in self.kernel_nodes:
    print("node name: {}, delay buffer size: {}".format(node, self.kernel_nodes[node].delay_buffer))
print()

print("path length info:")
for node in self.kernel_nodes:
    print("node name: {}, path lengths: {}".format(node, self.kernel_nodes[node].input_paths))
print()

print("latency info:")
for node in self.kernel_nodes:
    print("node name: {}, node latency: {}".format(node, self.kernel_nodes[node].graph.max_latency))
print()

print("critical path info:")
print("critical path length is {}\n".format(self.compute_critical_path()))

print("total buffer info:")
total = 0
for node in self.kernel_nodes:
    for u, v, channel in self.graph.edges(data='channel'):
        if channel is not None:
            total_delay = 0
            for item in channel["internal_buffer"]:
                total_delay += item.maxsize
            total_internal = 0
            total_delay += channel["delay_buffer"].maxsize
            total += total_delay + total_internal
print("total buffer size: {}\n".format(total))

print("input kernel string info:")
for node in self.kernel_nodes:
    print("input kernel string of {} is: {}".format(node, self.kernel_nodes[node].kernel_string))
print()

print("relative access kernel string info:")
for node in self.kernel_nodes:
    print("relative access kernel string of {} is: {}".format(node, self.kernel_nodes[node].
                                                              generate_relative_access_kernel_string()))

print("instantiate optimizer...")
from optimizer import Optimizer
opt = Optimizer(self.kernel_nodes, self.dimensions)
bound = 12001
opt.minimize_fast_mem(communication_volume_bound=bound)
print("optimize fast memory usage with comm volume bound= {}".format(bound))
print("single stream comm vol for float32 is: {}".format(opt.single_comm_volume(4)))
```

```
        print("total buffer info:")
        total = 0
        for node in self.kernel_nodes:
            for u, v, channel in self.graph.edges(data='channel'):
                if channel is not None:
                    total_fast = 0
                    total_slow = 0
                    for entry in channel["internal_buffer"]:
                        if entry.swap_out:
                            print("internal buffer slow memory: {}, size: {}".format(entry.name, entry.maxsize))
                            total_slow += entry.maxsize
                        else:
                            print("internal buffer fast memory: {}, size: {}".format(entry.name, entry.maxsize))
                            total_fast += entry.maxsize
                    entry = channel["delay_buffer"]
                    if entry.swap_out:
                        print("delay buffer slow memory: {}, size: {}".format(entry.name, entry.maxsize))
                        total_slow += entry.maxsize
                    else:
                        print("delay buffer fast memory: {}, size: {}".format(entry.name, entry.maxsize))
                        total_fast += entry.maxsize
        print("buffer size slow memory: {} \nbuffer size fast memory: {}".format(total_slow, total_fast))


if __name__ == "__main__":

    """
        simple test stencil program for debugging

        usage: python3 kernel_chain_graph.py -stencil_file stencils/simulator12.json -plot -simulate -report -log-level 2
    """
    # instantiate the argument parser
    parser = argparse.ArgumentParser()
    parser.add_argument("-stencil_file")
    parser.add_argument("-plot", action="store_true")
    parser.add_argument("-log-level")
    parser.add_argument("-report", action="store_true")
    parser.add_argument("-simulate", action="store_true")
    args = parser.parse_args()
    # instantiate the KernelChainGraph
    chain = KernelChainGraph(path=args.stencil_file,
                             plot_graph=args.plot,
                             log_level=int(args.log_level))
    # simulate the design if argument -simulate is true
    if args.simulate:
        from simulator import Simulator

        sim = Simulator(input_config_name=re.match("[^\.]+", os.path.basename(args.stencil_file)).group(0),
```

```
                    input_nodes=chain.input_nodes,
                    input_config=chain.inputs,
                    kernel_nodes=chain.kernel_nodes,
                    output_nodes=chain.output_nodes,
                    dimensions=chain.dimensions,
                    write_output=False,
                    log_level=int(args.log_level))
    sim.simulate()

# output a report if argument -report is true
if args.report:
    chain.report(args.stencil_file)
    if args.simulate:
        sim.report()
```

stencil_chain.config

```
{
  "eps": 1e-10
}
```

```
import functools
import operator
from typing import List, Dict

from log_level import LogLevel


class Simulator:
    """
        The Simulator class handles the simulation of our high level model of the fpga functionality of the stencil chain
        design (see paper example ref# TODO).

        interface for FPGA-like execution (gets called from the scheduler)

        - read:
                - saturation phase: read unconditionally
                - execution phase: read all inputs iff they are available
        - execute:
                - saturation phase: do nothing
                - execution phase: if input read, execute stencil using the input
        - write:
                - saturation phase: do nothing
                - execution phase: write result from execution to output buffers
                    --> if output buffer overflows: assumptions about size was wrong!
    """

    def __init__(self,
                 input_config_name: str,
                 input_nodes: Dict,
                 input_config: Dict,
                 kernel_nodes: Dict,
                 output_nodes: Dict,
                 dimensions: List,
                 write_output: bool,
                 log_level: int) -> None:
        """
        Create new Simulator class with given initialization parameters.
        :param input_config_name: name of the input file
        :param input_nodes: dict of all input nodes
        :param input_config: input config dict
        :param kernel_nodes: dict of all kernel nodes
        :param output_nodes: dict of all output nodes
        :param dimensions: global problem size dimensions
        :param write_output: flag for defining whether or not to write the result to a file
        :param log_level: flag for console output logging
        """
        # save params
        self.input_config_name: str = input_config_name
        self.dimensions: List = dimensions
```

```
    self.input_nodes: Dict = input_nodes
    self.input_config: Dict = input_config
    self.kernel_nodes: Dict = kernel_nodes
    self.output_nodes: Dict = output_nodes
    self.write_output: bool = write_output
    self.log_level: int = log_level

def step_execution(self):
    """
    Execute one step/cycle (read & execute & write).
    """
    """
        try to read all kernel inputs
    """
    # read output nodes
    for output in self.output_nodes:
        try:
            self.output_nodes[output].try_read()
        except Exception as ex:  # error
            self.diagnostics(ex)
    # read kernel nodes
    for kernel in self.kernel_nodes:
        try:
            # reset the internal state to be ready for a new computation
            self.kernel_nodes[kernel].reset_old_compute_state()
            self.kernel_nodes[kernel].try_read()
        except Exception as ex:  # error
            self.diagnostics(ex)
    """
        try to execute all kernels
    """
    # execute kernel nodes
    for kernel in self.kernel_nodes:
        try:
            self.kernel_nodes[kernel].try_execute()
        except Exception as ex:
            self.diagnostics(ex)

    """
        try to write all kernel outputs
    """
    # write input nodes
    for input in self.input_nodes:
        try:
            self.input_nodes[input].try_write()
        except Exception as ex:
            self.diagnostics(ex)
    # write kernel nodes
    for kernel in self.kernel_nodes:
```

```
            try:
                self.kernel_nodes[kernel].try_write()
            except Exception as ex:
                self.diagnostics(ex)
        """
            update performance metrics
        """
        for kernel in self.kernel_nodes:
            try:
                self.kernel_nodes[kernel].update_performance_metric()
            except Exception as ex:
                self.diagnostics(ex)

    def initialize(self):
        """
        Initialize the input nodes with data given in the config file.
        """
        # loop over all input nodes
        if self.log_level >= LogLevel.BASIC.value:
            print("Initialize simulator input arrays.")
        for input in self.input_nodes:
            # import data
            self.input_nodes[input].init_input_data(self.input_config)

    def finalize(self):
        """
        Do the necessary post-processsing after the simulator completed the step execution.
        """
        # check if write flag set
        if self.write_output:
            # save data to files
            for output in self.output_nodes:
                self.output_nodes[output].write_result_to_file(self.input_config_name)
        # output kernel performance metric
        if self.log_level >= LogLevel.BASIC.value:
            for kernel in self.kernel_nodes:
                self.kernel_nodes[kernel].print_kernel_performance()

    def get_result(self):
        """
        Returns the result stored in all output nodes.
        """
        # add all output node data to the dictionary
        result_dict = dict()
        for output in self.output_nodes:
            result_dict[output] = self.output_nodes[output].data_queue.export_data()
        return result_dict

    def all_done(self) -> bool:
```

```
    """
    Check if all nodes completed their execution.
    :return:
    """
    # compute the total problem size
    total_elements = functools.reduce(operator.mul, self.dimensions)
    # check if all input nodes completed their write
    for input in self.input_nodes:
        if self.input_nodes[input].program_counter < total_elements:
            return False
    # check if all kernel nodes completed their execution
    for kernel in self.kernel_nodes:
        if self.kernel_nodes[kernel].program_counter < total_elements:
            return False
    # check if all output nodes completed their read
    for output in self.output_nodes:
        if self.output_nodes[output].program_counter < total_elements:
            return False
    return True

def simulate(self):
    """
    Run the main simulation loop
    """
    # init
    if self.log_level >= LogLevel.BASIC.value:
        print("Initialize simulation.")
    self.initialize()
    # run simulation
    if self.log_level >= LogLevel.BASIC.value:
        print("Run simulation.")
    PC = 0
    while not self.all_done():
        if self.log_level >= LogLevel.FULL.value:  # output program counter of each node
            print("Execute next step. Current global program counter: {}.".format(PC))
        # execute
        self.step_execution()
        # increment program counter
        PC += 1
        if self.log_level >= LogLevel.FULL.value:  # output program counter of each node
            for input in self.input_nodes:
                print("input:{}, PC: {}".format(input, self.input_nodes[input].program_counter))
            for kernel in self.kernel_nodes:
                print("kernel:{}, PC: {}".format(kernel, self.kernel_nodes[kernel].program_counter))
            for output in self.output_nodes:
                print("output:{}, PC: {}".format(output, self.output_nodes[output].program_counter))
    # write completion message
    if self.log_level >= LogLevel.BASIC.value:
        print("Simulation completed after {} cycles.".format(PC))
```

```
    # finalize the simulation
    if self.log_level >= LogLevel.BASIC.value:
        print("Finalize simulation.")
    self.finalize()
    if self.log_level >= LogLevel.BASIC.value:
        print("Create simulator report.")
        self.report()
    if self.log_level >= LogLevel.BASIC.value:
        print("Write result report.")
        for out in self.output_nodes:
            print("name: {}, data: {}".format(out, self.output_nodes[out].data_queue.export_data()))

def report(self):
    if self.log_level >= LogLevel.BASIC.value:
        print("Create simulator report.")
    self.diagnostics(None)

def diagnostics(self, exception):
    if exception is not None:
        print("Error: Exception {} has been risen. Run diagnostics.".format(exception.__traceback__))
    if self.log_level >= LogLevel.BASIC.value:
        print("Run diagnostics of {}.".format(self.input_config_name))
    # print info about all inputs
    for input in self.input_nodes:
        print("input:{}, PC: {}".format(input, self.input_nodes[input].program_counter))
    # call debug diagnostics output of all kernels
    for kernel in self.kernel_nodes:
        self.kernel_nodes[kernel].diagnostics(exception)
    # print info about all outputs
    for output in self.output_nodes:
        print("output:{}, PC: {}".format(output, self.output_nodes[output].program_counter))
```

```
import operator
from functools import reduce
from typing import List, Dict

import helper
from kernel import Kernel
from log_level import LogLevel


class Optimizer:
    """
        This is the Optimizer class to optimize the usage of fast on-chip memory vs slow ddr memory vs bandwidth between
        them.

        Optimization strategy:
            - initial state: all buffers are in fast memory, there is no communication volume used for transferring data
              between slow and fast memory
            - optimize for:
                - minimize_comm_vol
                - minimize_fast_mem
                -
    """

    def __init__(self,
                 kernels: Dict[str, Kernel],
                 dimensions: List[int],
                 log_level: int = 0):
        """
        Create new BoundedQueue with given initialization parameters.
        :param kernels: all kernels
        :param dimensions: global dimensions / problem size (i.e. size of the input array
        :param verbose: flag for console output logging
        """
        if log_level >= LogLevel.BASIC.value:
            print("Initialize Optimizer.")
        # save params
        self.kernels = kernels
        self.dimensions: List[int] = dimensions
        self.log_level = log_level
        # init local fields
        self.fast_memory_use: int = 0
        self.slow_memory_use: int = 0
        self.metric_data: List[Dict] = list()
        self.config = helper.parse_json("stencil_chain.config")
        self.eps = self.config["eps"]  # machine precision (i.e. used for division by (almost) zero)
        # run init methods
        if self.log_level >= LogLevel.MODERATE.value:
            print("Add all buffers to the metric.")
        self.add_buffers_to_metric()
```

```
    if self.log_level >= LogLevel.MODERATE.value:
        print("Reset old state.")
    self.reset()

def reinit(self):
    """
    Clean old state to run a new optimization round.
    """
    # reinit local fields
    self.fast_memory_use: int = 0
    self.slow_memory_use: int = 0
    self.metric_data: List[Dict] = list()
    # run init methods
    self.add_buffers_to_metric()
    self.reset()

def minimize_comm_vol(self,
                      fast_memory_bound: int,
                      slow_memory_bound: int) -> None:
    """
    This optimization strategy optimizes the problem to minimize the communication volume between fast and slow
    memory. In other words, it uses as few bandwidth as possible by a given amount of fast and slow memory.
    :param fast_memory_bound: maximum available fast on-chip memory
    :param slow_memory_bound: maximum available slow dram memory
    """
    if self.log_level >= LogLevel.BASIC.value:
        print(
            "Run optimizer in mode: Minimize Communication Volume with fast memory bound: {} and slow memory "
            "bound: {}.".format(fast_memory_bound, slow_memory_bound))
    self.reinit()
    # optimize for minimal communication volume use / maximal fast memory use
    opt = self.max_metric()
    while not self.empty_list(self.metric_data) and self.fast_memory_use > fast_memory_bound:
        self.fast_memory_use -= opt["queue"].maxsize * opt["datatype_size"]
        self.slow_memory_use += opt["queue"].maxsize * opt["datatype_size"]
        opt["queue"].swap_out = True
        self.update_neighbours(opt)
        self.metric_data.remove(opt)
        opt = self.max_metric()
    if self.slow_memory_use > slow_memory_bound:
        raise Exception("Optimization failed, slow memory bound: {}, slow memory necessary to hold fast memory "
                        "constraint: {}".format(slow_memory_bound, self.slow_memory_use))
    if self.log_level >= LogLevel.MODERATE.value:
        self.report()

def minimize_fast_mem(self,
                      communication_volume_bound: int) -> None:
    """
    This optimization strategy minimizes the usage of fast memory by a given amount of communication volume from/to
```

```
    slow memory.
    :param communication_volume_bound: maximum available data volume between fast and slow memory
    """
    if self.log_level >= LogLevel.BASIC.value:
        print("Run optimizer in mode: Minimize Fast Memory Usage with comm volume bound: {}".format(
            communication_volume_bound))
    self.reinit()
    # optimize for minimal fast memory use / maximum communication volume use
    opt = self.max_metric()
    while not self.empty_list(self.metric_data) and opt["comm_vol"] < communication_volume_bound:
        communication_volume_bound -= opt["comm_vol"]
        self.fast_memory_use -= opt["queue"].maxsize * opt["datatype_size"]
        self.slow_memory_use += opt["queue"].maxsize * opt["datatype_size"]
        opt["queue"].swap_out = True
        self.update_neighbours(opt)
        self.metric_data.remove(opt)
        opt = self.max_metric()
    if self.log_level >= LogLevel.MODERATE.value:
        self.report()

def optimize_to_ratio(self,
                      ratio: float) -> None:
    """
    This optimization strategy optimizes for the given ratio value between the fast memory usage and the amount of
    available communication volume.
    :param ratio: ratio = #fast_mem / #comm_vol
    """
    if self.log_level >= LogLevel.BASIC.value:
        print("Run optimizer in mode: Optimize to Ratio with ratio: {}".format(ratio))
    self.reinit()
    # optimize for the ratio of #fast_memory/communication_volume
    opt = self.max_metric()
    while not self.empty_list(self.metric_data) and self.ratio() > ratio:
        self.fast_memory_use -= opt["queue"].maxsize * opt["datatype_size"]
        self.slow_memory_use += opt["queue"].maxsize * opt["datatype_size"]
        opt["queue"].swap_out = True
        self.update_neighbours(opt)
        self.metric_data.remove(opt)
        opt = self.max_metric()
    if self.log_level >= LogLevel.MODERATE.value:
        self.report()

@staticmethod
def empty_list(lst: List) -> bool:
    """
    Check if collection is empty.
    :param lst: collection
    """
    return len(lst) == 0
```

optimizer.py

```
def ratio(self):
    """
    Get the ratio between fast and the communication volume of the current optimization state.
    :return: ratio: fast_mem / comm_vol
    """
    total_com = 0
    for item in self.metric_data:
        total_com += item["comm_vol"]
    return self.fast_memory_use / (total_com + self.eps)

def reset(self) -> None:
    """
    Reset the internal optimizer state to start a new round of optimization.
    """
    # initially put all buffers into fast memory
    for item in self.metric_data:
        item["queue"].swap_out = False
    # count fast memory usage
    self.fast_memory_use = 0
    for item in self.metric_data:
        if not item["queue"].swap_out:
            self.fast_memory_use += item["queue"].maxsize
    # reset slow memory usage
    self.slow_memory_use = 0

def max_metric(self) -> Dict:
    """
    Return the entry with the highest ratio between its size and the necessary communication volume (i.e. the "worst"
    node).
    :return: worst node (best candidate to swap out to slow memory)
    """
    if self.empty_list(self.metric_data):
        return dict()  # empty dict
    else:
        return max(self.metric_data, key=lambda x: x["queue"].maxsize / x["comm_vol"])

def update_neighbours(self,
                      buffer: Dict) -> None:
    """
    After moving the buffer from/to slow/fast memory, the communication volume to the direct neighbours must get
    updated.
    :param buffer: node that is being moved
    """
    if buffer["prev"] is not None:
        self.update_comm_vol(buffer["prev"])
    if buffer["next"] is not None:
        self.update_comm_vol(buffer["next"])
```

```
def update_comm_vol(self,
                    buffer: Dict) -> None:
    """
        Recompute the communication volume.

        How to determine the necessary communication volume?
        (predecessor, successor):
        case (fast, fast): 2C
        case (fast, slow): C
        case (slow, fast): C
        case (slow, slow): 0
            where C:= communication volume to stream single data array in or out of fast memory (=SIZE(data array))
        Note:
        pred of delay buffer is always fast memory
    :param buffer: node that has to be recomputed
    """
    # determine predecessor fast/slow
    if buffer["type"] == "delay":
        pre_fast = True
    elif buffer["prev"]["queue"].swap_out:
        pre_fast = False
    else:
        pre_fast = True
    # determine successor fast/slow
    if buffer["next"] is None:
        succ_fast = True
    elif buffer["next"]["queue"].swap_out:
        succ_fast = False
    else:
        succ_fast = True
    # set comm vol accordingly
    if pre_fast and succ_fast:  # case (fast, fast)
        buffer["comm_vol"] = 2 * self.single_comm_volume(buffer["datatype_size"])
    elif (pre_fast and not succ_fast) or (not pre_fast and succ_fast):  # case (fast, slow) or (slow, fast)
        buffer["comm_vol"] = 1 * self.single_comm_volume(buffer["datatype_size"])
    else:  # case (slow, slow)
        buffer["comm_vol"] = self.eps

def add_buffers_to_metric(self):
    """
    Create buffer data structure for optimization (contain buffers, type of buffer as well as predecessor/successor.
    """
    # loop over all kernels
    for kernel in self.kernels:
        # loop over all delay buffers
        for buf in self.kernels[kernel].delay_buffer:
            # get delay buffer first
            del_buf = {
                "queue": self.kernels[kernel].delay_buffer[buf],
```

```
                "comm_vol": 2 * self.single_comm_volume(self.kernels[kernel].data_type.bytes),
                "type": "delay",
                "datatype_size": self.kernels[kernel].data_type.bytes,
                "prev": None,
                "next": None}
            self.fast_memory_use += del_buf["queue"].maxsize * del_buf["datatype_size"]
            self.metric_data.append(del_buf)
            # get internal buffers next
            prev = del_buf
            for entry in self.kernels[kernel].internal_buffer[buf]:
                curr = {
                    "queue": entry,
                    "comm_vol": 2 * self.single_comm_volume(self.kernels[kernel].data_type.bytes),
                    "type": "internal",
                    "datatype_size": self.kernels[kernel].data_type.bytes,
                    "prev": prev,
                    "next": None}
                prev["next"] = curr
                self.fast_memory_use += curr["queue"].maxsize * curr["datatype_size"]
                self.metric_data.append(curr)
                prev = curr

def single_comm_volume(self,
                       datatype_size: int):
    """
    # Returns the number of bytes necessary to copy a whole array from or to the fpga.
    :param datatype_size: size in bytes of the kernel data type e.g. 4 for float32
    :return:
    """
    return reduce(operator.mul, self.dimensions) * datatype_size

def report(self):
    print("Optimization report:")
    # sum up data values
    total_fast, total_slow, total_comm = 0, 0, 0
    for kernel in self.kernels:
        # loop over all delay buffers
        for buf in self.kernels[kernel].delay_buffer:
            # get delay buffer
            if self.kernels[kernel].delay_buffer[buf].swap_out:
                print("Delay buffer: {} {}: swapped out to slow memory".format(kernel, buf))
                total_slow += self.kernels[kernel].delay_buffer[buf].maxsize * self.kernels[kernel].data_type.bytes
            else:
                print("Delay buffer: {} {}: kept in fast memory".format(kernel, buf))
                total_fast += self.kernels[kernel].delay_buffer[buf].maxsize * self.kernels[kernel].data_type.bytes
            # get internal buffers
            for entry in self.kernels[kernel].internal_buffer[buf]:
                if entry.swap_out:
                    print("Internal buffer: {} {} index {}: swapped out to slow memory"
```

```
                                .format(kernel, buf, self.kernels[kernel].internal_buffer[buf].index(entry)))
                            total_slow += entry.maxsize * self.kernels[kernel].data_type.bytes
                        else:
                            print("Internal buffer: {} {} index {}: kept in fast memory".format(kernel, buf, self.kernels[
                                kernel].internal_buffer[buf].index(entry)))
                            total_fast += entry.maxsize * self.kernels[kernel].data_type.bytes
        for item in self.metric_data:
            total_comm += self.metric_data["comm_vol"]
        print("Total fast memory usage: {} bytes".format(total_fast))
        print("Total slow memory usage: {} bytes".format(total_slow))
        print("Total communication volume usage: {} bytes".format(total_comm))


if __name__ == "__main__":
    opt = Optimizer()
    print()
```

sdfg_generator.py

```
import argparse
import collections
import functools
import itertools
import operator
import os
import re

import dace
import dace.codegen.targets.fpga
import numpy as np
from dace.graph.edges import InterstateEdge
from dace.memlet import Memlet
from dace.sdfg import SDFG
from dace.types import ScheduleType, StorageType, Language

import helper
from kernel_chain_graph import Kernel, Input, Output, KernelChainGraph

ITERATORS = ["i", "j", "k"]


def make_iterators(dimensions, halo_sizes=None):
    def add_halo(i):
        if i == len(dimensions) - 1 and halo_sizes is not None:
            return " + " + str(-halo_sizes[0] + halo_sizes[1])
        else:
            return ""
    if len(dimensions) > 3:
        return collections.OrderedDict(
            [("i" + str(i), "0:" + str(d) + add_halo(i))
             for i, d in enumerate(dimensions)])
    else:
        return collections.OrderedDict(
            [(ITERATORS[i], "0:" + str(d) + add_halo(i))
             for i, d in enumerate(dimensions)])


def make_stream_name(src_name, dst_name):
    return src_name + "_to_" + dst_name


def relative_to_buffer_index(buffer_size, index):
    return buffer_size - 1 - abs(index)


def generate_sdfg(name, chain):

    sdfg = SDFG(name)
```

sdfg_generator.py

```
pre_state = sdfg.add_state("initialize")
state = sdfg.add_state("compute")
post_state = sdfg.add_state("finalize")

sdfg.add_edge(pre_state, state, InterstateEdge())
sdfg.add_edge(state, post_state, InterstateEdge())

def add_pipe(sdfg, edge):

    stream_name = make_stream_name(edge[0].name, edge[1].name)

    sdfg.add_stream(
        stream_name,
        edge[0].data_type,
        buffer_size=edge[2]["channel"]["delay_buffer"].maxsize,
        storage=StorageType.FPGA_Local,
        transient=True)

def add_input(node):

    # Host-side array, which will be an input argument
    sdfg.add_array(node.name + "_host", chain.dimensions, node.data_type)

    # Device-side copy
    sdfg.add_array(
        node.name,
        chain.dimensions,
        node.data_type,
        storage=StorageType.FPGA_Global,
        transient=True)
    access_node = state.add_read(node.name)

    iterators = make_iterators(chain.dimensions)

    # Copy data to the FPGA
    copy_host = pre_state.add_read(node.name + "_host")
    copy_fpga = pre_state.add_write(node.name)
    pre_state.add_memlet_path(
        copy_host,
        copy_fpga,
        memlet=Memlet.simple(
            copy_fpga,
            ", ".join(iterators.values()),
            num_accesses=functools.reduce(operator.mul, chain.dimensions,
                                          1)))

    entry, exit = state.add_map(
        "read_" + node.name, iterators, schedule=ScheduleType.FPGA_Device)
```

```
    # Sort to get deterministic output
    outputs = sorted([e[1].name for e in chain.graph.out_edges(node)])

    out_memlets = ["_" + o for o in outputs]

    tasklet_code = "\n".join(
        ["{} = memory".format(o) for o in out_memlets])

    tasklet = state.add_tasklet("read_" + node.name, {"memory"},
                                out_memlets, tasklet_code)

    state.add_memlet_path(
        access_node,
        entry,
        tasklet,
        dst_conn="memory",
        memlet=Memlet.simple(
            node.name, ", ".join(iterators.keys()), num_accesses=1))

    # Add memlets to all FIFOs connecting to compute units
    for out_name, out_memlet in zip(outputs, out_memlets):
        stream_name = make_stream_name(node.name, out_name)
        write_node = state.add_write(stream_name)
        state.add_memlet_path(
            tasklet,
            exit,
            write_node,
            src_conn=out_memlet,
            memlet=Memlet.simple(stream_name, "0", num_accesses=1))

def add_output(node):

    # Host-side array, which will be an output argument
    sdfg.add_array(node.name + "_host", chain.dimensions, node.data_type)

    # Device-side copy
    sdfg.add_array(
        node.name,
        chain.dimensions,
        node.data_type,
        storage=StorageType.FPGA_Global,
        transient=True)
    write_node = state.add_write(node.name)

    iterators = make_iterators(chain.dimensions)

    # Copy data to the FPGA
    copy_fpga = post_state.add_read(node.name)
```

```
    copy_host = post_state.add_write(node.name + "_host")
    post_state.add_memlet_path(
        copy_fpga,
        copy_host,
        memlet=Memlet.simple(
            copy_host,
            ", ".join(iterators.values()),
            num_accesses=functools.reduce(operator.mul, chain.dimensions,
                                          1)))

    entry, exit = state.add_map(
        "write_" + node.name, iterators, schedule=ScheduleType.FPGA_Device)

    src = chain.graph.in_edges(node)
    if len(src) > 1:
        raise RuntimeError("Only one writer per output supported")
    src = next(iter(src))[0]

    in_memlet = "_" + src.name

    tasklet_code = "memory = " + in_memlet

    tasklet = state.add_tasklet("write_" + node.name, {in_memlet},
                                {"memory"}, tasklet_code)

    stream_name = make_stream_name(src.name, node.name)
    read_node = state.add_read(stream_name)

    state.add_memlet_path(
        read_node,
        entry,
        tasklet,
        dst_conn=in_memlet,
        memlet=Memlet.simple(stream_name, "0", num_accesses=1))

    state.add_memlet_path(
        tasklet,
        exit,
        write_node,
        src_conn="memory",
        memlet=Memlet.simple(
            node.name, ", ".join(iterators.keys()), num_accesses=1))

def add_kernel(node):

    # Extract fields to read from memory and fields to buffer
    memory_accesses = []
    buffer_sizes = collections.OrderedDict()
    buffer_accesses = collections.OrderedDict()
```

sdfg_generator.py

```
for field, relative in node.graph.accesses.items():
    # Deduplicate, as we can have multiple accesses to the same index
    abs_indices = helper.unique(
        [helper.dim_to_abs_val(i, node.dimensions) for i in relative] +
        ([0]
         if node.boundary_conditions[field]["type"] == "copy" else []))
    max_access = max(abs_indices)
    min_access = min(abs_indices)
    buffer_size = max_access - min_access + 1
    memory_accesses.append(field)
    buffer_sizes[field] = buffer_size
    # (indices relative to center, buffer indices, buffer center index)
    buffer_accesses[field] = (relative,
                              [i - min_access for i in abs_indices],
                              -min_access)

# Create a initialization phase corresponding to the highest distance
# to the center
init_sizes = [
    buffer_sizes[key] - 1 - val[2]
    for key, val in buffer_accesses.items()
]
init_size_max = np.max(init_sizes)

iterators = make_iterators(chain.dimensions)

# Manually add pipeline entry and exit nodes
pipeline_range = dace.properties.SubsetProperty.from_string(', '.join(
    iterators.values()))
pipeline = dace.codegen.targets.fpga.Pipeline(
    "read_" + node.name,
    list(iterators.keys()),
    pipeline_range,
    ScheduleType.FPGA_Device,
    False,
    init_size=init_size_max,
    init_overlap=False,
    drain_size=init_size_max,
    drain_overlap=True)
entry = dace.codegen.targets.fpga.PipelineEntry(pipeline)
exit = dace.codegen.targets.fpga.PipelineExit(pipeline)
state.add_nodes_from([entry, exit])

# Sort to get deterministic output
outputs = sorted(e[1].name for e in chain.graph.out_edges(node))

# Add nested SDFG to do 1) shift buffers 2) read from input 3) compute
nested_sdfg = SDFG(node.name, parent=sdfg)
nested_sdfg_tasklet = state.add_nested_sdfg(
```

```
    nested_sdfg,
    sdfg, ([name + "_in" for name in memory_accesses] +
           [name + "_buffer_in" for name, _ in buffer_sizes.items()]),
    ([name + "_out" for name in outputs] +
     [name + "_buffer_out" for name, _ in buffer_sizes.items()]),
    name=node.name)

# Shift state, which shifts all buffers by one
shift_state = nested_sdfg.add_state(node.name + "_shift")

# Update state, which reads new values from memory
update_state = nested_sdfg.add_state(node.name + "_update")

# TODO: data type per field
dtype = node.data_type

######################################################################
# Implement boundary conditions
######################################################################

boundary_code = ""
# Loop over each field
for (field, (accesses, accesses_buffer,
             center)) in buffer_accesses.items():
    # Loop over each access to this field
    for indices, offset_buffer in zip(accesses, accesses_buffer):
        # Loop over each index of this access
        cond = []
        for i, offset in enumerate(indices):
            if offset < 0:
                cond.append(ITERATORS[i] + " < " + str(-offset))
            elif offset > 0:
                cond.append(ITERATORS[i] + " >= " +
                            str(node.dimensions[i] - offset))
        if len(cond) == 0:
            boundary_code += "{} {}_{} = _{}_{};\n".format(
                dtype.ctype, field, offset_buffer, field,
                offset_buffer)
        else:
            if node.boundary_conditions[field]["type"] == "copy":
                boundary_val = "_{}_{}".format(field, center)
            elif node.boundary_conditions[field]["type"] == "constant":
                boundary_val = node.boundary_conditions[field]["value"]
            boundary_code += (
                "{} {}_{} = ({}) ? ({}) : (_{}_{});\n".format(
                    dtype.ctype, field, offset_buffer,
                    " || ".join(cond), boundary_val, field,
                    offset_buffer))
```

sdfg_generator.py

```
########################################################################
# Only write if we're in bounds
########################################################################

write_code = (
    ("if (!{}) {{\n".format("".join(pipeline.init_condition()))
     if init_size_max > 0 else "") + ("\n".join([
         "write_channel_intel({}_inner_out, res);".format(output)
         for output in outputs
     ])) + ("\n}" if init_size_max > 0 else "\n"))

########################################################################
# Tasklet code generation
########################################################################

code = boundary_code + "\n" + "\n".join([
    dtype.ctype + " " + expr.strip() + ";"
    for expr in node.generate_relative_access_kernel_string(
        relative_to_center=False).split(";")
])
# Replace array accesses with memlet names
pattern = re.compile("(([a-zA-Z_][a-zA-Z0-9_]*)\[([^\]]+)\])")
for full_str, field, index in re.findall(pattern, code):
    buffer_index = relative_to_buffer_index(buffer_sizes[field],
                                            int(index))
    if int(index) > 0:
        raise ValueError("Received positive index " + full_str + ".")
    code = code.replace(full_str, "{}_{}".format(field, buffer_index))
code += "\n" + write_code

########################################################################
# Create DaCe compute state
########################################################################

# Compute state, which reads from input channels, performs the compute,
# and writes to the output channel(s)
compute_state = nested_sdfg.add_state(node.name + "_compute")
compute_inputs = list(
    itertools.chain.from_iterable(
        [["_{}_{}".format(name, offset) for offset in offsets]
         for name, (_, offsets, _) in buffer_accesses.items()]))
compute_tasklet = compute_state.add_tasklet(
    node.name + "_compute",
    compute_inputs, [name + "_inner_out" for name in outputs],
    code,
    language=Language.CPP)

# Connect the three nested states
nested_sdfg.add_edge(shift_state, update_state, InterstateEdge())
```

```
nested_sdfg.add_edge(update_state, compute_state, InterstateEdge())

for in_name, (field_name, size), init_size in zip(
        memory_accesses, buffer_sizes.items(), init_sizes):

    stream_name = make_stream_name(in_name, node.name)

    # Outer memory read
    read_node_outer = state.add_read(stream_name)
    state.add_memlet_path(
        read_node_outer,
        entry,
        nested_sdfg_tasklet,
        dst_conn=in_name + "_in",
        memlet=Memlet.simple(stream_name, "0", num_accesses=-1))

    # Create inner memory pipe
    stream_name_inner = field_name + "_in"
    stream_inner = sdfg.arrays[stream_name].clone()
    stream_inner.transient = False
    nested_sdfg.add_datadesc(stream_name_inner, stream_inner)

    buffer_name_outer = "{}_{}_buffer".format(node.name, field_name)
    buffer_name_inner_read = "{}_buffer_in".format(field_name)
    buffer_name_inner_write = "{}_buffer_out".format(field_name)

    # Create buffer transient in outer SDFG
    # TODO: use data type from edge
    if size > 1:
        desc_outer = sdfg.add_array(
            buffer_name_outer, [size],
            dtype,
            storage=StorageType.FPGA_Local,
            transient=True)
    else:
        desc_outer = sdfg.add_scalar(
            buffer_name_outer,
            dtype,
            storage=StorageType.FPGA_Registers,
            transient=True)

    # Create read and write nodes
    read_node_outer = state.add_read(buffer_name_outer)
    write_node_outer = state.add_write(buffer_name_outer)

    # Outer memory read
    state.add_memlet_path(
        read_node_outer,
        entry,
```

```
    nested_sdfg_tasklet,
    dst_conn=buffer_name_inner_read,
    memlet=Memlet.simple(
        buffer_name_outer, "0:{}".format(size), num_accesses=-1))

# Outer memory write
state.add_memlet_path(
    nested_sdfg_tasklet,
    exit,
    write_node_outer,
    src_conn=buffer_name_inner_write,
    memlet=Memlet.simple(
        write_node_outer.data,
        "0:{}".format(size),
        num_accesses=-1))

# Inner copy
desc_inner_read = desc_outer.clone()
desc_inner_read.transient = False
desc_inner_read.name = buffer_name_inner_read
desc_inner_write = desc_inner_read.clone()
desc_inner_write.name = buffer_name_inner_write
nested_sdfg.add_datadesc(buffer_name_inner_read, desc_inner_read)
nested_sdfg.add_datadesc(buffer_name_inner_write, desc_inner_write)

# Make shift state if necessary
if size > 1:
    shift_read = shift_state.add_read(buffer_name_inner_read)
    shift_write = shift_state.add_write(buffer_name_inner_write)
    shift_entry, shift_exit = shift_state.add_map(
        "shift_{}".format(field_name),
        {"i_shift": "0:{} - 1".format(size)},
        schedule=ScheduleType.FPGA_Device,
        unroll=True)
    shift_tasklet = shift_state.add_tasklet(
        "shift_{}".format(field_name),
        {"{}_shift_in".format(field_name)},
        {"{}_shift_out".format(field_name)},
        "{field}_shift_out = {field}_shift_in".format(
            field=field_name))
    shift_state.add_memlet_path(
        shift_read,
        shift_entry,
        shift_tasklet,
        dst_conn=field_name + "_shift_in",
        memlet=Memlet.simple(
            shift_read.data, "i_shift + 1", num_accesses=1))
    shift_state.add_memlet_path(
        shift_tasklet,
```

```
            shift_exit,
            shift_write,
            src_conn=field_name + "_shift_out",
            memlet=Memlet.simple(
                shift_write.data, "i_shift", num_accesses=1))

    # Begin reading according to this field's own buffer size, which is
    # translated to an index by subtracting it from the maximum buffer
    # size
    begin_reading = (init_size_max - init_size)
    end_reading = (functools.reduce(operator.mul, chain.dimensions, 1)
                   + init_size_max - init_size)

    update_read = update_state.add_read(stream_name_inner)
    update_write = update_state.add_write(buffer_name_inner_write)
    update_tasklet = update_state.add_tasklet(
        "read_wavefront", {"wavefront_in"}, {"buffer_out"},
        "if ({it} >= {begin} && {it} < {end}) {{\n"
        "buffer_out = read_channel_intel(wavefront_in);\n"
        "}} else {{\n"
        "buffer_out = -1000;\n"
        "}}\n".format(it=pipeline.iterator_str(), begin=begin_reading,
        end=end_reading),
        language=Language.CPP)
    update_state.add_memlet_path(
        update_read,
        update_tasklet,
        memlet=Memlet.simple(update_read.data, "0", num_accesses=-1),
        dst_conn="wavefront_in")
    update_state.add_memlet_path(
        update_tasklet,
        update_write,
        memlet=Memlet.simple(
            update_write.data,
            "{} - 1".format(size) if size > 1 else "0",
            num_accesses=1),
        src_conn="buffer_out")

    # Make compute state
    compute_read = compute_state.add_read(buffer_name_inner_read)
    for offset in buffer_accesses[field_name][1]:
        compute_state.add_memlet_path(
            compute_read,
            compute_tasklet,
            dst_conn="_" + field_name + "_" + str(offset),
            memlet=Memlet.simple(
                compute_read.data, str(offset), num_accesses=1))

for out_name in outputs:
```

```
            # Outer write
            stream_name_outer = make_stream_name(node.name, out_name)
            write_node_outer = state.add_write(stream_name_outer)
            state.add_memlet_path(
                nested_sdfg_tasklet,
                exit,
                write_node_outer,
                src_conn=out_name + "_out",
                memlet=Memlet.simple(
                    write_node_outer.data, "0", num_accesses=-1))

            # Create inner stream
            stream_name_inner = out_name + "_out"
            stream_inner = sdfg.arrays[stream_name_outer].clone()
            stream_inner.transient = False
            nested_sdfg.add_datadesc(stream_name_inner, stream_inner)

            # Inner write
            write_node_inner = compute_state.add_write(stream_name_inner)
            compute_state.add_memlet_path(
                compute_tasklet,
                write_node_inner,
                src_conn=out_name + "_inner_out",
                memlet=Memlet.simple(
                    write_node_inner.data, "0", num_accesses=-1))

    for link in chain.graph.edges(data=True):
        add_pipe(sdfg, link)

    for node in chain.graph.nodes():
        if isinstance(node, Input):
            add_input(node)
        elif isinstance(node, Output):
            add_output(node)
        elif isinstance(node, Kernel):
            add_kernel(node)
        else:
            raise RuntimeError("Unexpected node type: {}".format(
                node.node_type))

    return sdfg


if __name__ == "__main__":

    parser = argparse.ArgumentParser()
    parser.add_argument("stencil_file")
    parser.add_argument("--plot-graph", dest="plot-graph", action="store_true")
```

```
parser.add_argument("--plot-sdfg", dest="plot-sdfg", action="store_true")

args = parser.parse_args()

name = os.path.basename(args.stencil_file)
name = re.match("[^\.]+", name).group(0)

chain = KernelChainGraph(args.stencil_file)

if getattr(args, "plot-graph"):
    chain.plot_graph(name + ".pdf")

sdfg = generate_sdfg(name, chain)

if getattr(args, "plot-sdfg"):
    chain.plot_graph(name + ".pdf")

sdfg.compile()
```

run_program.py

```
#!/usr/bin/env python3
import argparse
import itertools
import os
import re
import subprocess as sp

import dace
import numpy as np

import helper
from kernel_chain_graph import KernelChainGraph
from sdfg_generator import generate_sdfg
from simulator import Simulator

parser = argparse.ArgumentParser()
parser.add_argument("stencil_file")
parser.add_argument("mode", choices=["emulation", "hardware"])
parser.add_argument("-log-level", choices=["0", "1", "2", "3"], default=3)
parser.add_argument("-plot", action="store_true")
parser.add_argument("-simulation", action="store_true")
parser.add_argument("-skip-execution", dest="skip_execution", action="store_true")
parser.add_argument("--print-result", dest="print_result", action="store_true")
args = parser.parse_args()

# Load program file
program_description = helper.parse_json(args.stencil_file)
name = os.path.basename(args.stencil_file)
name = re.match("[^\.]+", name).group(0)

# Create SDFG
print("Create KernelChainGraph")
chain = KernelChainGraph(path=args.stencil_file,
                         plot_graph=args.plot,
                         log_level=int(args.log_level))

# do simulation
if args.simulation:
    print("Run simulation.")
    sim = Simulator(input_config_name=re.match("[^\.]+", os.path.basename(args.stencil_file)).group(0),
                    input_nodes=chain.input_nodes,
                    input_config=chain.inputs,
                    kernel_nodes=chain.kernel_nodes,
                    output_nodes=chain.output_nodes,
                    dimensions=chain.dimensions,
                    write_output=False,
                    log_level=int(args.log_level))
    sim.simulate()
    simulation_result = sim.get_result()
```

run_program.py

```
print("Generate sdfg")
sdfg = generate_sdfg(name, chain)

# Configure and compile SDFG
print("Configure sdfg")
dace.config.Config.set("compiler", "fpga_vendor", value="intel_fpga")
# dace.config.Config.set("compiler", "use_cache", value=True)
dace.config.Config.set("optimizer", "interface", value="")
if args.mode == "emulation":
    dace.config.Config.set("compiler", "intel_fpga", "mode", value="emulator")
else:
    dace.config.Config.set("compiler", "intel_fpga", "mode", value="hardware")
print("Compile sdfg")
program = sdfg.compile()

# Load data from disk
print("Loading input arrays...")
input_arrays = helper.load_input_arrays(program_description)

# Initialize output arrays
print("Initializing output arrays...")
output_arrays = {
    arr_name: np.zeros(
        program_description["dimensions"],
        dtype=program_description["program"][arr_name]["data_type"].type)
    for arr_name in program_description["outputs"]
}

# Compile program (if running emulation)
build_folder = os.path.join(".dacecache", name, "build")
if args.mode == "emulation":
    print("Compiling emulation kernel...")
    sp.run(
        ["make", "intelfpga_compile_" + name + "_emulator"],
        cwd=build_folder,
        check=True)
elif args.mode == "hardware":
    if not os.path.exists(os.path.join(build_folder, name + "_hardware.aocx")):
        raise FileNotFoundError("Hardware kernel has not been built.")

if args.skip_execution:
    exit()

# Run program
dace_args = {
    key + "_host": val
    for key, val in itertools.chain(input_arrays.items(),
                                    output_arrays.items())
```

```
}
print("Executing DaCe program...")
program(**dace_args)
print("Finished running program.")

if args.print_result:
    for key, val in output_arrays.items():
        print(key + ":", val)

# Write results to file
output_folder = os.path.join("results", name)
os.makedirs(output_folder, exist_ok=True)
helper.save_output_arrays(output_arrays, output_folder)
print("Results saved to " + output_folder)

# Compare simulation result to fpga result
if args.simulation:
    print("Comparing the results.")
    all_match = True
    for outp in output_arrays:
        print("fpga:")
        print(np.ravel(output_arrays[outp]))
        print("simulation")
        print(np.ravel(simulation_result[outp]))
        if not helper.arrays_are_equal(np.ravel(output_arrays[outp]), np.ravel(simulation_result[outp])):
            all_match = False
    if all_match:
        print("Output matched!")
    else:
        print("Output did not match!")
```

helper.py

```
import collections
import functools
import json
import operator
import os.path
import warnings
from functools import reduce
from typing import List, Dict

import dace
import numpy as np

"""
    This file contains many helper methods that are being re-used in multiple classes and do not specifically belong to
    a class.
"""


def deprecated(func):
    """
    This is a decorator which can be used to mark functions as deprecated. It will result in a warning being emitted
    when the function is used.
    """
    @functools.wraps(func)
    def new_func(*args, **kwargs):
        warnings.simplefilter('always', DeprecationWarning)  # turn off filter
        warnings.warn(
            "Call to deprecated function {}.".format(func.__name__),
            category=DeprecationWarning,
            stacklevel=2)
        warnings.simplefilter('default', DeprecationWarning)  # reset filter
        return func(*args, **kwargs)
    return new_func


def str_to_dtype(dtype_str: str) -> dace.types.typeclass:
    """
    Conversion from the data type name (string) to its type defined in dace.
    :param dtype_str: string data type
    :return: dace data type
    """
    if not isinstance(dtype_str, str):  # type check
        raise TypeError("Expected string, got: " + type(dtype_str).__name__)
    try:
        return getattr(dace.types, dtype_str)  # match type
    except AttributeError:
        pass
    raise AttributeError("Unsupported data type: " + dtype_str)  # missmatch
```

helper.py

```
def parse_json(config_path: str) -> Dict:
    """
    Read input file from disk and parse it.
    :param config_path: path to the file
    :return: parsed file
    """
    # check file exists
    if not os.path.isfile(config_path):
        relative = os.path.join(
            os.path.dirname(os.path.realpath(__file__)), config_path)
        if not os.path.isfile(relative):
            raise RuntimeError("file {} does not exists.".format(config_path))
        config_path = relative
    # open file in with-clause, to ensure proper file closing even in the event of an exception
    with open(config_path, "r") as file_handle:
        # try to parse it
        config = json.loads(file_handle.read())  # type: dict
    # Save the path to the config
    config["path"] = os.path.dirname(os.path.abspath(config_path))
    def walk(d):  # replace string data type in config
        for key, val in d.items():
            if isinstance(val, dict):
                walk(val)
            else:
                if key == "data_type":
                    d[key] = str_to_dtype(val)
    walk(config)
    # return dict
    return config


def max_dict_entry_key(dict1: Dict[str, List[int]]) -> str:
    """
    Get key of largest value entry out of the input dictionary.
    :param dict1: a dictionary with keys as names and values as buffer sizes
    :return: key of buffer entry with maximum size
    """
    # check type
    if not isinstance(dict1, dict):
        raise Exception("dict1 should be of type {}, but is of type {}".format(
            type(dict), type(dict1)))
    # extract max value entry
    return max(dict1, key=dict1.get)


def list_add_cwise(list1: List,
                   list2: List) -> List:
    """
```

```
    Merge two lists by component-wise addition.
    :param list1: input list: summand
    :param list2: input list: summand
    :return: merged list
    """
    # check type
    if not isinstance(list1, list):
        raise Exception("list1 should be of type {}, but is of type {}".format(
            type(list), type(list1)))
    if not isinstance(list2, list):
        raise Exception("list2 should be of type {}, but is of type {}".format(
            type(list), type(list2)))
    # do map lambda operation over both lists
    return list(map(lambda x, y: x + y, list1, list2))


def list_subtract_cwise(list1: List,
                        list2: List) -> List:
    """
    Merge two lists by component-wise subtraction.
    :param list1: input list: minuend
    :param list2: input list: subtrahend
    :return: merged list
    """
    # check type
    if not isinstance(list1, list):
        raise Exception("list1 should be of type {}, but is of type {}".format(
            type(list), type(list1)))
    if not isinstance(list2, list):
        raise Exception("list2 should be of type {}, but is of type {}".format(
            type(list), type(list2)))
    # do map lambda operation over both lists
    return list(map(lambda x, y: x - y, list1, list2))


def dim_to_abs_val(input: List[int],
                   dimensions: List[int]) -> int:
    """
    Computes scalar number out of independent dimension unit.
    :param input: vector to evaluate
    :param dimensions: vector of global array dimensions
    :return: scalar value
    """
    # dim [X, Y, Z], size [a, b, c] -> 1*c + X*(b + Y*a) = [a, b, c] * transpose([Z*Y, Z, 1])
    vec = [
        reduce(operator.mul, dimensions[i + 1:], 1)
        for i in range(len(dimensions))
    ]
    return reduce(operator.add, map(operator.mul, input, vec))  # inner product
```

helper.py

```
def load_array(source_config: Dict, search_path=None):
    """
    Load array from file or list into numpy array.
    :param source_config: External data input file config.
    :param search_path: Path to search as relative paths.
    :return: Data stored in a numpy array.
    """
    # get path to either source file or direct to the embedded array
    data = source_config["data"]
    if isinstance(data, str):  # source file
        path = data
        if not os.path.isfile(path):
            if search_path is not None:
                path = os.path.join(search_path, path)
            if not os.path.isfile(path):
                raise RuntimeError("File {} does not exists.".format(data))
        if path.endswith(".csv"):
            return np.genfromtxt(path, float, delimiter=',')
        elif path.endswith(".dat"):
            return np.fromfile(path, float)
        else:
            raise ValueError("Invalid file type: " + path)
    elif isinstance(data, np.ndarray):  # embedded array: already numpy array
        return data
    else:
        return np.array(data, dtype=source_config["data_type"].type) # embedded array: collection item -> convert to
        # np array


def load_input_arrays(program: Dict) -> Dict:
    """
    Loads input arrays for the passed program into memory.
    :param program: Program tree as generated by parse_json.
    :return: Dictionary of input names to input data as numpy arrays.
    """
    # add all input arrays to the dict
    input_arrays = dict()
    for arr_name, source in program["inputs"].items():
        input_arrays[arr_name] = load_array(source, program["path"])
    return input_arrays


def save_array(array, path):
    """
    Saves array to a binary file.
    :param array: numpy array to save.
    :param path: Path to binary file where data will be saved.
```

```
    """
    array.tofile(path)


def save_output_arrays(outputs: Dict, output_dir=str()):
    """
    Saves output arrays to individual files.
    :param outputs: Dictionary of array names to numpy arrays.
    :param output_dir: Folder to store files in.
    """
    # store all arrays in the output directory path
    for arr_name, arr_data in outputs.items():
        path = os.path.join(output_dir, arr_name + ".dat")
        save_array(arr_data, path)


def arrays_are_equal(reference, result, tolerance=1e-3):
    """
    Check if two arrays are equal within a tolerance.
    :param reference: numpy array or path to file.
    :param result: numpy array or path to file.
    :param tolerance: Maximum relative (fractional) difference between arrays.
    """
    if not isinstance(reference, np.ndarray):
        reference = load_array(reference)
    if not isinstance(result, np.ndarray):
        result = load_array(result)
    # tolerate zeroes by adding epsilon to the divisor
    relative_diff = (np.abs(reference - result) / (np.maximum.reduce([reference, result]) + np.finfo(np.float64).eps))
    return np.all(relative_diff <= tolerance)


def unique(iterable):
    """
    Removes duplicates in the passed iterable.
    :param iterable: iterable with potential duplicates
    :return iterable without duplicates
    """
    try:
        return type(iterable)([i for i in sorted(set(iterable), key=lambda x: iterable.index(x))])
    except TypeError:
        return type(iterable)(collections.OrderedDict(
            zip(map(str, iterable), iterable)).values())


def convert_3d_to_1d(dimensions: List[int], index: List[int]) -> int:
    """
    Convert the size of form [a, b, c] to absolute values according to the global dimensions of the problem.
    :param dimensions: problem size (e.g. size of the input arrays)
```

```
    :param index: 3d value of the size
    :return the 1d value of the size
    """
    # convert [i, j, k] to flat 1D array index using the given dimensions [dimX, dimY, dimZ]
    # index = i*dimY*dimZ + j*dimZ + k = (i*dimY + j)*dimZ + k
    if not index:
        return 0  # empty list
    return dim_to_abs_val(index, dimensions)


if __name__ == "__main__":
    """
        Basic helper function test. Comprehensive testing is implemented in 'testing.py'.
    """
    example_list = [[1, 2, 2], [1, 2, 3], [3, 2, 1], [2, 3, 1]]
    print("properties of list {}:\nmin: {}\nmax: {}\n".format(
        example_list, min(example_list), max(example_list)))

    example_dict = {
        "small": [0, 10, 10],
        "very small": [0, 1, 0],
        "extra large": [12, 1, 2],
        "large": [10, 10, 10]
    }
    print("max value entry key of dict {} is:\n\'{}\'".format(
        example_dict, max_dict_entry_key(example_dict)))
```

log_level.py

```
from enum import Enum


class LogLevel(Enum):
    NO_LOG = 0
    BASIC = 1
    MODERATE = 2
    FULL = 3
```

testing.py

```
import unittest

from bounded_queue import BoundedQueue


class BoundedQueueTest(unittest.TestCase):

    def test_import(self):
        # init
        queue = BoundedQueue(name="test",
                             maxsize=5)
        # init_queue
        collection = [1.0, 2.0, 3.0, 4.0, 5.0]
        queue.import_data(collection)
        # check size
        self.assertEqual(queue.size(), len(collection))
        # check if data added in the right order
        self.assertEqual(queue.try_peek_last(), collection[len(collection) - 1])
        # check exception for overfilling queue
        self.assertRaises(RuntimeError, queue.import_data, 6 * [1.0])

    def test_enq_deq(self):
        # init
        queue = BoundedQueue(name="test",
                             maxsize=1,
                             collection=[1.0])
        # check size
        self.assertEqual(queue.size(), 1)
        # empty queue, check element value
        self.assertEqual(queue.dequeue(), 1.0)
        # check size
        self.assertEqual(queue.size(), 0)
        # check size
        self.assertTrue(queue.is_empty())
        # check exception on underflow
        self.assertRaises(RuntimeError, queue.dequeue)
        # enqueue element
        queue.enqueue(1.0)
        # check size
        self.assertTrue(queue.is_full())
        # check exception on overflow
        self.assertRaises(RuntimeError, queue.enqueue, 2.0)

    def test_try_enq_deq(self):
        # init
        queue = BoundedQueue(name="test",
                             maxsize=1,
                             collection=[1.0])
        # check size
```

```
        self.assertEqual(queue.size(), 1)
        # empty queue, check element value
        self.assertEqual(queue.try_dequeue(), 1.0)
        # check size
        self.assertEqual(queue.size(), 0)
        # check size
        self.assertTrue(queue.is_empty())
        # dequeue from emtpy queue, check return value
        self.assertFalse(queue.try_dequeue())
        # enqueue, into non-full list, check return value
        self.assertTrue(queue.try_enqueue(1.0))
        # check size
        self.assertTrue(queue.is_full())
        # enqueue into full queue, check return value
        self.assertFalse(queue.try_enqueue(1.0), 2.0)

    def test_peek(self):
        # init
        queue = BoundedQueue(name="test",
                             maxsize=2,
                             collection=[1.0, 2.0])
        # check value at index 0
        self.assertEqual(queue.peek(0), 1.0)
        # check value at index 1
        self.assertEqual(queue.peek(1), 2.0)
        # check value at last location
        self.assertEqual(queue.try_peek_last(), 2.0)
        # empty queue
        queue.dequeue()
        queue.dequeue()
        # peek on empty queue, check return value
        self.assertFalse(queue.try_peek_last())


from calculator import Calculator
from numpy import cos


class CalculatorTest(unittest.TestCase):

    def test_calc(self):
        # init vars
        variables = dict()
        variables["a"] = 7.0
        variables["b"] = 2.0
        # init calc
        computation = "cos(a + b) if (a > b) else (a + 5) * b"
        calculator = Calculator()
        # do manual calculation and compare result
```

```
        result = cos(variables["a"] + variables["b"]) if (variables["a"] > variables["b"]) else (variables["a"] + 5) * \
                                                                                                 variables["b"]
        self.assertEqual(calculator.eval_expr(variables, computation), result)


class RunProgramTest(unittest.TestCase):

    def test(self):
        pass  # not a general test case, since dace and intel fgpa opencl sdk has to be installed and configured


import helper
import numpy as np


class HelperTest(unittest.TestCase):

    def test(self):
        # check max_dict_entry_key
        self.assertEqual(helper.max_dict_entry_key({"a": [1, 0, 0], "b": [0, 1, 0], "c": [0, 0, 1]}), "a")
        # check list_add_cwise
        self.assertEqual(helper.list_add_cwise([1, 2, 3], [3, 2, 1]), [4, 4, 4])
        # check list_subtract_cwise
        self.assertEqual(helper.list_subtract_cwise([1, 2, 3], [1, 2, 3]), [0, 0, 0])
        # check dim_to_abs_val
        self.assertEqual(helper.dim_to_abs_val([3, 2, 1], [10, 10, 10]), 321)
        # check convert_3d_to_1d
        self.assertEqual(helper.convert_3d_to_1d([10, 10, 10], [3, 2, 1]), 321)
        # check load_array
        self.assertListEqual(list(helper.load_array({"data": "testing/helper_test.csv", "data_type": "float64"})),
                             [7.0, 7.0])
        self.assertListEqual(list(helper.load_array({"data": "testing/helper_test.dat", "data_type": "float64"})),
                             [7.0, 7.0])
        # check save_array / load_array
        out_data = np.array([1.0, 2.0, 3.0])
        file = {"data": "test.dat", "data_type": "float64"}
        helper.save_array(out_data, file["data"])
        in_data = helper.load_array(file)
        self.assertTrue(helper.arrays_are_equal(out_data, in_data))
        os.remove(file["data"])
        # check unique
        not_unique = [1.0, 2.0, 1.0]
        self.assertListEqual(sorted(helper.unique(not_unique)), [1.0, 2.0])


from compute_graph import ComputeGraph
import json
import os
```

testing.py

```
class ComputeGraphTest(unittest.TestCase):

    def test(self):
        # define example computation
        computation = "out = cos(3.14);res = A[i,j,k] if (A[i,j,k]+1 > A[i,j,k]-B[i,j,k]) else out"
        # instantiate ComputeGraph and generate internal state
        graph = ComputeGraph()
        graph.generate_graph(computation)
        graph.calculate_latency()
        # load operation latency manually to compare result
        with open('compute_graph.config') as json_file:
            op_latency = json.load(json_file)
        # check if latencies match
        self.assertEqual(op_latency["op_latency"]["cos"] + op_latency["op_latency"]["add"] + 1, graph.max_latency)
        # save plot
        filename = "compute_graph_unittest.png"
        graph.plot_graph(filename)  # write graph to file
        # delete plot
        os.remove(filename)


from kernel import Kernel
import dace.types


class KernelTest(unittest.TestCase):

    def test(self):
        # define global problem size
        dimensions = [100, 100, 100]
        # instantiate example kernel
        kernel = Kernel(name="dummy",
                        kernel_string="SUBST = a[i,j,k] + a[i,j,k-1] + a[i,j-1,k] + a[i-1,j,k]; res = SUBST + a[i,j,k]",
                        dimensions=dimensions,
                        data_type=dace.types.float64,
                        boundary_conditions={"a": {"type": "constant", "value": 1.0}})
        # check if the string matches
        self.assertEqual(kernel.generate_relative_access_kernel_string(),
                         "SUBST = (((a[0] + a[-1]) + a[-100]) + a[-10000]); res = (SUBST + a[0])")


from kernel_chain_graph import KernelChainGraph


class KernelChainGraphTest(unittest.TestCase):

    def test(self):
        chain = KernelChainGraph(path='stencils/simple_input_delay_buf.json', plot_graph=False)
```

```
        # Note: Since e.g. the delay buffer sizes get tested using different cases (e.g. through the simulator), we only
        # add a basic (no exception) case in here for the moment.


from optimizer import Optimizer


class OptimizerTest(unittest.TestCase):

    def test(self):
        # instantiate example KernelChainGraph
        chain = KernelChainGraph(path='stencils/simple_input_delay_buf.json', plot_graph=False)
        # instantiate the Optimizer
        opt = Optimizer(chain.kernel_nodes, chain.dimensions)
        # define bounds
        com_bound = 10000
        fast_mem_bound = 1000
        slow_mem_bound = 100000
        ratio = 0.5
        # run all optimization strategies
        opt.minimize_fast_mem(communication_volume_bound=com_bound)
        opt.minimize_comm_vol(fast_memory_bound=fast_mem_bound, slow_memory_bound=slow_mem_bound)
        opt.optimize_to_ratio(ratio=ratio)


from simulator import Simulator


class SimulatorTest(unittest.TestCase):

    def test(self):
        # set up all sample configs with their (paper) result
        samples = {
            "sample1": {
                "file": "stencils/simulator.json",
                "res": [5.14, 4.14, 5.14, 11.14, 7.14, 8.14]
            },
            "sample2": {
                "file": "stencils/simulator2.json",
                "res": [3., 4., 3., 4., 5., 4., 3., 4., 3.]
            },
            "sample3": {
                "file": "stencils/simulator3.json",
                "res": [4., 7., 8., 13., 20., 19., 16., 25., 20.]
            },
            "sample4": {
                "file": "stencils/simulator4.json",
                "res": [3., 3., 3., 3., 3., 3., 3., 3., 3.]
            },
```

```
    "sample5": {
        "file": "stencils/simulator5.json",
        "res": [7., 9., 7., 9., 11., 9., 7., 9., 7.]
    },
    "sample6": {
        "file": "stencils/simulator6.json",
        "res": [14., 18., 14., 18., 22., 18., 14., 18., 14.]
    },
    "sample7": {
        "file": "stencils/simulator7.json",
        "res": [20.25, 20.25, 19.25, 20.25, 20.25, 19.25, 16.25, 16.25, 16.25]
    },
    "sample8": {
        "file": "stencils/simulator8.json",
        "res": [4., 8., 12., 16., 20., 24., 28., 32., 36.]
    },
    "sample9": {
        "file": "stencils/simulator9.json",
        "res": [3., 5., 7., 9., 11., 13.]
    },
    "sample10": {
        "file": "stencils/simulator10.json",
        "res": [1., 6., 11., 16., 21., 26.]
    },
    "sample11": {
        "file": "stencils/simulator11.json",
        "res": [4., 2., 3., 10., 5., 6., 16., 8., 9.]
    },
    "sample12": {
        "file": "stencils/simulator12.json",
        "res": [20.25, 20.25, 19.25, 20.25, 20.25, 19.25, 16.25, 16.25, 16.25, 20.25, 20.25, 19.25, 20.25,
                20.25,
                19.25, 16.25, 16.25, 16.25, 20.25, 20.25, 19.25, 20.25, 20.25, 19.25, 16.25, 16.25, 16.25]
    }
}
# run all samples
for sample in samples:
    chain = KernelChainGraph(path=samples[sample]['file'], plot_graph=False)
    sim = Simulator(input_nodes=chain.input_nodes,
                    input_config=chain.inputs,
                    kernel_nodes=chain.kernel_nodes,
                    output_nodes=chain.output_nodes,
                    dimensions=chain.dimensions,
                    input_config_name="test",
                    write_output=False,
                    verbose=False)
    sim.simulate()
    # check if result matches
    self.assertTrue(helper.arrays_are_equal(np.array(samples[sample]['res']), sim.get_result()['res'], 0.01))
```

```
if __name__ == '__main__':
    """
        Run all unit tests.
    """
    unittest.main()
```

dycore_estimate.py

```
import functools
import operator
from functools import reduce
from typing import List

import helper
from kernel_chain_graph import KernelChainGraph

"""
    Intro:
        This is a buffer size and bandwidth estimate for the whole dynamical core of the COSMO weather model. With only
        a few actual kernel chains ported and having the shape of the whole (production) dynamical core, we try to
        estimate if it the buffer space and bandwidth requirements are in the order of resources we have available on a
        single or an cluster of at most 32 Intel Stratix 10 FPGAs.

    Assumptions:
        - data type: float (32bit IEEE 754)
        - iteration over the smaller two dimensions possible (e.g. 1024x1024x64 -> iterate over 64x1024)
        - estimate for critical path length (#cycles):
            Stencil Chains (automatic output):
                fastwaves:     [3, 1, 54]
                diffusion_min: [3, 0, 24]
                advection_min: [7, 4, 26]
                -> use the mean of the three: [4, 2, 35]
        - clock frequency: 200Mhz
        - fast memory (Stratix 10): 25MB
        - bandwidth to slow memory (Stratix 10): 86.4GB/s
        - we assume that as soon as the pipeline is saturated, we can produce one result per cycle

"""
# datatype: for the moment, we assume float (32bit) are precise enough
_SIZEOF_DATATYPE: int = 4  # in bytes

# we assume that we can iterate over the smaller dimension by transposition of all data arrays, if not, change this to
# [64, 1024, 1024] (would lead to an additional factor 16 of buffer space growth)
_DIMENSIONS: List[int] = [1024, 1024, 64]  # longitude x latitude x altitude

#  FPGA clock frequency: 200Mhz
_FPGA_CLOCK_FREQUENCY: int = int(2e8)

# available FGPA fast memory (Stratix 10): 25MB
_FPGA_FAST_MEMORY_SIZE: int = int(25e6)

# available bandwidth to slow memory (Stratix 10): 86.4GB/s
_FPGA_BANDWIDTH: int = int(86.4e9)

# we assume that as soon as the pipeline is saturated, we can produce one result per cycle
_CYCLES_PER_OUTPUT: int = 1
```

dycore_estimate.py

```
def do_estimate():
    """
    This function is meant to programmatically go through the current 'best-estimate' calculation of our model.
    :return: None
    """

    # estimate for critical path length (#cycles): mean of the tree stencils: fastwaves, diffusion, advection
    critical_paths: List[List[int]] = list(list())
    # instantiate fastwaves and add critical path
    critical_paths.append(KernelChainGraph(path="input/fastwaves.json",
                                           plot_graph=False,
                                           verbose=False).compute_critical_path_dim())
    # instantiate diffusion and add critical path
    critical_paths.append(KernelChainGraph(path="input/diffusion.json",
                                           plot_graph=False,
                                           verbose=False).compute_critical_path_dim())
    # instantiate advection and add critical path
    critical_paths.append(KernelChainGraph(path="input/advection.json",
                                           plot_graph=False,
                                           verbose=False).compute_critical_path_dim())
    # calculate mean of the three
    critical_path_sum = functools.reduce(lambda x, y: helper.list_add_cwise(x, y), critical_paths, [0] * 3)
    mean: List[int] = list(map(lambda x: x / len(critical_paths), critical_path_sum))
    _MEAN_CRITICAL_PATH_KERNEL: List[int] = mean
    print("Mean critical path length of the three stencils is: {}".format(mean))
    # print header
    print("############################################################")
    print("COSMO dynamical core buffer size estimate report:\n")
    print("############################################################")
    # instantiate the dummy-dycore (modified to fixed latency per kernel of 4*latency(addition) to get full analysis
    chain = KernelChainGraph("input/dycore_upper_half.json")
    # assumption: since we implemented ~1/2 of the dycore in the dummy input file, we assume the critical path is 2x
    # longer
    _DYCORE_CRITICAL_PATH_LENGTH = 2 * chain.compute_critical_path()
    # compute total critical path
    critical_path_dim = [x * _DYCORE_CRITICAL_PATH_LENGTH for x in _MEAN_CRITICAL_PATH_KERNEL]
    print("total critical path length (dimensionless) = _MEAN_CRITICAL_PATH_KERNEL * _DYCORE_CRITICAL_PATH_LENGTH = "
          "{} * {} = {}".format(_MEAN_CRITICAL_PATH_KERNEL, _DYCORE_CRITICAL_PATH_LENGTH, critical_path_dim))
    critical_path_cyc = helper.dim_to_abs_val(critical_path_dim, _DIMENSIONS)
    print("total critical path length (cycles) = {} cycles\n".format(critical_path_cyc))
    # compute maximum possible communication volume
    run_time_cyc = critical_path_cyc + reduce(operator.mul, _DIMENSIONS)
    print("total run time (cycles) = latency + dimX*dimY*dimZ = {}".format(run_time_cyc))
    run_time_sec = run_time_cyc / _FPGA_CLOCK_FREQUENCY
    print("total run time (seconds) = total run time (cycles) / _FPGA_CLOCK_FREQUENCY = {}".format(run_time_sec))
    comm_vol = _FPGA_BANDWIDTH * run_time_sec
    print("maximum available communication volume (to slow memory) = _FPGA_BANDWIDTH * total run time (seconds) = {}"
          .format(comm_vol))
```

dycore_estimate.py

```
if __name__ == "__main__":
    do_estimate()
```

binary_input_generator.py

```
#!/usr/bin/env python3
import argparse

import numpy as np

# example call:
# python3 binary_input_generator.py data/random_0_10_16x16_fp32 16*16 random csv float32 -rand_lo 0.0 -rand_hi 10.0
parser = argparse.ArgumentParser()
parser.add_argument("filename")
parser.add_argument("N")
parser.add_argument("generator", choices=["const", "random", "incr"], default="random")
parser.add_argument("fileextension", choices=["csv", "bin", "dat"], default="dat")
parser.add_argument("datatype", choices=["float32", "float64"], default="float64")
parser.add_argument("-rand_lo")
parser.add_argument("-rand_hi")
parser.add_argument("-const_val")
args = parser.parse_args()

# general parameters
_N = int(eval(args.N))
_GENERATOR = args.generator  # {const, random, incr}
_FILENAME = args.filename
_FILE_EXTENSION = args.fileextension  # {csv, bin, dat}
_DATA_TYPE = np.dtype(args.datatype)
# random
_LOWER_BOUND = float(args.rand_lo) if args.rand_lo is not None else 0.0
_UPPER_BOUND = float(args.rand_hi) if args.rand_hi is not None else 0.0
# constant
_CONSTANT = float(args.const_val) if args.const_val is not None else 0.0

# generate data
if _GENERATOR == "random":
    # random data
    data = np.array(np.random.uniform(low=_LOWER_BOUND, high=_UPPER_BOUND, size=_N), dtype=_DATA_TYPE)
elif _GENERATOR == "const":
    # constant value
    data = np.array([_CONSTANT]*_N, dtype=_DATA_TYPE)
elif _GENERATOR == "incr":
    # increasing value
    data = np.arange(0.0, _N, dtype=_DATA_TYPE)
else:
    print("Generator {} not supported. Exit.".format(_GENERATOR))
    exit()

# write data to file
filename = _FILENAME + "." + _FILE_EXTENSION

if _FILE_EXTENSION == "csv":
    np.savetxt(filename, [data], delimiter=",")
```

```
elif _FILE_EXTENSION == "bin" or _FILE_EXTENSION == "dat":
    data.astype(_DATA_TYPE).tofile(filename)
```

slurm_hardware_synthesis.py

```
import argparse
import os
from subprocess import call

# example call: python3 slurm_hardware_synthesis.py jacobi3d

# grab all arguments
parser = argparse.ArgumentParser()
parser.add_argument("name")  # name of the design we want to synthesis
parser.add_argument("-N", default=1)  # nodes, default: 1
parser.add_argument("-n", default=8)  # task, default: 8
parser.add_argument("-c", default=8)  # cores, default: 8
parser.add_argument("--mem", default=131072)  # memory, default: 131072MB=128GB
parser.add_argument("-o", default="outfile")  # stdout, default: saved to outfile_NAME
parser.add_argument("-e", default="errfile")  # stderr, default: saved to errfile_NAME
parser.add_argument("-t", default="24:00:00")  # time requested hh:mm:ss, default: 24h
parser.add_argument("--partition", default="long")  # cluster queue, default: long
args = parser.parse_args()

if args.name is None:
    raise Exception("No design name specified, exit.")

header = [("-N", args.N), ("-n", args.n), ("-c", args.c), ("--mem", args.mem),
          ("-o", "{}_{}".format(args.o, args.name)), ("-e", "{}_{}".format(args.e, args.name)), ("-t", args.t),
          ("--partition", args.partition)]
home_dir = os.path.expanduser("~")

_SDK_PATH = "source /apps/ault/intelFPGA_pro/19.1/hld/init_stratix.sh\n"
_PYTHON_PATH = "module load python/3.7.2\n"
_CMAKE_PATH = "module load cmake/3.14.0\n"
_GCC_PATH = "module load gcc/8.3.0\n"

# generate slurm batch job file
with open("{}/{}.sh".format(home_dir, args.name), "w") as f:
    f.write("#!/bin/sh\n")
    for item in header:
        f.write("#SBATCH {} {}\n".format(item[0], item[1]))
    f.write(_SDK_PATH)
    f.write(_PYTHON_PATH)
    f.write(_CMAKE_PATH)
    f.write(_GCC_PATH)
    f.write("cd {}/.dacecache/{}/build/\n".format(os.getcwd(), args.name))
    f.write("make intelfpga_compile_{}_hardware\n".format(args.name))

call(["sbatch", "{}/{}.sh".format(home_dir, args.name)])
```

slurm_hardware_run.py

```
import argparse
import os
from subprocess import call

# example call: python3 slurm_hardware_run.py jacobi3d

# grab all arguments
parser = argparse.ArgumentParser()
parser.add_argument("name")  # name of the design we want to synthesis
parser.add_argument("-N", default=1)  # nodes, default: 1
parser.add_argument("-n", default=1)  # task, default: 1
parser.add_argument("-c", default=1)  # cores, default: 1
parser.add_argument("--mem", default=32768)  # memory, default: 131072MB=128GB
parser.add_argument("-o", default="outfile")  # stdout, default: saved to outfile_NAME
parser.add_argument("-e", default="errfile")  # stderr, default: saved to errfile_NAME
parser.add_argument("-t", default="01:00:00")  # time requested hh:mm:ss, default: 24h
parser.add_argument("--partition", default="fpga")  # cluster queue, default: long
args = parser.parse_args()

if args.name is None:
    raise Exception("No design name specified, exit.")

header = [("-N", args.N), ("-n", args.n), ("-c", args.c), ("--mem", args.mem),
          ("-o", "{}_{}_run".format(args.o, args.name)), ("-e", "{}_{}_run".format(args.e, args.name)), ("-t", args.t),
          ("--partition", args.partition)]
home_dir = os.path.expanduser("~")

_SDK_PATH = "source /apps/ault/intelFPGA_pro/19.1/hld/init_stratix.sh\n"
_PYTHON_PATH = "module load python/3.7.2\n"
_CMAKE_PATH = "module load cmake/3.14.0\n"
_GCC_PATH = "module load gcc/8.3.0\n"

# generate slurm batch job file
with open("{}/{}_run.sh".format(home_dir, args.name), "w") as f:
    f.write("#!/bin/sh\n")
    for item in header:
        f.write("#SBATCH {} {}\n".format(item[0], item[1]))
    f.write(_SDK_PATH)
    f.write(_PYTHON_PATH)
    f.write(_CMAKE_PATH)
    f.write(_GCC_PATH)
    f.write("python3 code/run_program.py stencils/{}.json hardware -log-level 0\n".format(args.name))

call(["sbatch", "{}/{}_run.sh".format(home_dir, args.name)])
```

Efficient Implementation of a High-Performance In-production Weather Model on FPGA

Task description of the Bachelor thesis, ETH Zurich, supervised by Johannes de Fine Licht, Professor Torsten Hoefler

Andreas Kuster

Introduction

Accurate and reliable weather forecast is of vital importance for a broad field of industries, as well as the general public. Highly regular and statically analyzable stencil operators [1] on structured grids are used to numerically solve the partial differential equations of such weather prediction models. This allows optimizations for data re-use while minimizing the high demand of memory bandwidth [2, 3] on the FPGA (field-programmable gate array) platform. Our collaboration with MeteoSwiss [4] enables us to apply our theoretical optimization findings to the numerical weather prediction and regional climate model COSMO [5, 6]. By cooperating closely with the University of Paderborn [7], we gain access to a clustered heterogeneous supercomputer containing 32 interconnected Stratix 10 FPGAs [8] whereby we intend to figure out if FPGAs [9] are the optimal choice for future high-performance weather prediction simulations.

Research questions to answer (Q) / Development tasks to perform (D)

1. Manual analysis and formalization of the problem. (Q)
2. Defining a suitable input representation. (Q)
3. Automatic analysis of the input including computation of important characteristics such as latency, buffer requirements, etc. (D)
4. Feasibility estimate. (Q)
5. Formulation of optimization goals and constraints. (Q)

6. Automatic optimization of the input according to the goals/constraints using a suitable solver. (D)

7. Formalization of a FPGA simulation model and implementation of the software simulation for testing, debugging and performance metric measures. (Q,D)

8. Manual implementation of the stencil chains on (single/multiple) FPGAs including performance optimizations in HLS [10]. (Q,D)

9. Automatic code generation. (D)

10. [Optional] Porting and testing of the complete dynamical core of COSMO to the FPGA platform. (D)

References

[1] Saturo Yamamoto Wang Luzhou, Kentaro Sano. Domain-Specific Language and Compiler for Stencil Computation on FPGA-based Systolic Computational-Memory Array. *ARC'12*, 2012.

[2] Oliver Fuhrer Mauro Bianco Thomas C. Schulthess Tobias Gysi, Carlos Osuna. STELLA: A Domain-specific Tool for Structured Grid Methods in Weather and Climate Models. *SC 15*, 2015.

[3] Tomofumi Yuki Steven Derrien Sanjay Rajopadhye Gaël Deest, Nicolas Estibals. Towards Scalable and Efficient FPGA Stencil Accelerators. *IMPACT 2016*, 2016.

[4] Federal Office of Meteorology and Climatology MeteoSwiss. www.meteosuisse.admin.ch.

[5] Doris Folini. Climate, weather, space weather: model development in an operational context. *J. Space Weather Space Clim. 2018*, 2018.

[6] Consortium for Small-scale Modeling. www.cosmo-model.org.

[7] Paderborn university. www.uni-paderborn.de/en/university.

[8] Noctua Supercomputer. pc2.uni-paderborn.de/hpc-services/available-systems/noctua.

[9] Torsten Hoefler Johannes de Fine Licht, Michaela Blott. Designing scalable FGPA architectures using high-level synthesis. *PPoPP18*, 2018.

[10] Torsten Hoefler Johannes de Fine Licht, Simon Meierhans. Transformations of High-Level Synthesis Codes for High-Performance Computing. *arXiv.org*, 2018.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

StencilFlow: Stencil Dataflow on Reconfigurable Hardware

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Kuster	Andreas

Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigge" beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Zürich, 20.08.2019	A. Kuster

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.