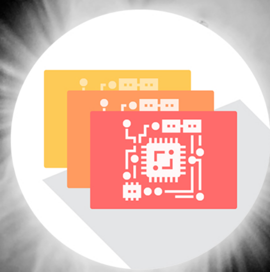


Interpolatory

Video Frame-Rate Interpolation IP Suite

Report & Documentation

INTEL VIDEO INTERPOLATION | IMPERIAL COLLEGE LONDON
NAIM GOVANI, OLLY LARKIN, LIN LEE, JIALONG YU, NAVID ZANDPOUR, ZHIYUAN ZHANG

GITHUB.COM/LHL2617/INTERPOLATORY

Interpolatory is supported by Dr. Kieron Turkington of Intel Corporation United Kingdom and Prof. George Constantinides of Imperial College London.

© 2020 Interpolatory. All Rights Reserved.

June 2020

Contents

1 Introduction . . . 5

1.1 Motivation 5

1.2 Objective 5

2 Design History . . . 7

2.1 Kick-off 8

2.1.1 Meetings . . . 8

2.2 Phase I - Research 8

2.2.1 Meetings . . . 8

2.2.2 Plans & Requirements . . . 10

2.2.3 Design Review . . . 11

2.3 Phase II - Development: Software Simulator 11

2.3.1 Meetings . . . 11

2.3.2 Plans & Requirements . . . 14

2.3.3 Outcome & Conclusion . . . 15

2.4 Phase III - Development: Hardware Estimation 15

2.4.1 Meetings . . . 15

2.4.2 Plans & Requirements . . . 16

3 Reports . . . 18

3.1 Sustainability Report 18

3.2 Ethical Consequences Report 18

3.3 Technical reports 19

3.3.1 Benchmarks & Performance Metrics . . . 19

3.3.2 Trivial Frame Interpolation methods . . . 20

3.3.3 MEMCI Frame Interpolation methods . . . 21

3.3.4 Machine Learning Interpolation methods 26
3.3.5 Hardware Estimations 27
3.3.6 Graphical User Interface 27
3.3.7 Numba 28
3.3.8 Results 28
3.3.9 Future Improvements 29

A Hardware Estimation Cache Schemes 32

A.1 Basic Algorithms 32
A.1.1 Frame Repetition 32
A.1.2 Oversampling 33
A.1.3 Linear 34

A.2 MEMCI 36
A.2.1 Full Search and Unidirectional 36
A.2.2 Three Step Search and Unidirectional 39
A.2.3 Hierarchical Block Matching Algorithm and Unidirectional 43
A.2.4 Full Search and Bidirectional 48
A.2.5 Three Step Search and Bidirectional 51
A.2.6 Hierarchical Block Matching Algorithm and Bidirectional 56
A.2.7 Full Search and Advanced Bidirectional 61
A.2.8 Three Step Search and Advanced Bidirectional 65
A.2.9 Hierarchical Block Matching Algorithm and Advanced Bidirectional 69

1. Introduction

This document serves as a record of design history and is meant to be read in conjunction with two other sources: the literature review (https://github.com/lhl2617/Interpolatory/files/4832591/LitReview.pdf) and the Github repository (https://github.com/lhl2617/Interpolatory). Please contact us if you do not have access to both resources.

When the resources are referenced, there will be a small icon, shown below, referencing the relevant section; for example, the README of the repository would be referenced as:

Refer to the repository:
Interpolatory/README.md
(https://github.com/lhl2617/Interpolatory/blob/master/README.md)

Before the design history is delineated, the subsequent two sections detail the motivation and objective of Interpolatory.

1.1 Motivation

A common requirement of video processing systems is the ability to convert from one frame rate at the input to a different frame rate at the output (for example, converting 24fps film footage to 60fps for display on typical televisions).

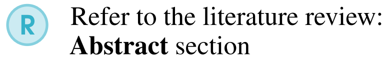
Trivial methods exist, most commonly, frame dropping or repeating is used; however, this can lead to undesirable judder in the output video which the human visual system is surprisingly good at detecting. Less naïve approaches involve interpolating frames between two original frames, essentially creating additional frames at points in time to increase the frame-rate. For instance, converting from 30fps to 60fps would require a new frame created in the halfway point in time between each pair of input frames.

Video frame interpolation is generally accepted to be a hard problem to solve, and it is easy to make the interpolated video look much worse than the results achieved through frame dropping and repeating. Moreover, video frame interpolation is extremely desirable as a real-time system, hence motivating the use of dedicated hardware, e.g. FPGAs.

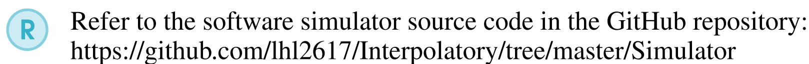
1.2 Objective

Interpolatory's main objective is to offer customers a IP FRUC suite, comprising of three sections.

The first section is a literature review intended to give an engineer with no prior experience in Frame-Rate Up-Conversion (FRUC) an introduction into the various terms, methods and benchmarking tools currently being researched in FRUC. A collection of relevant papers is summarised succinctly, and it is intended to get an engineer up-to-speed with a plethora of FRUC algorithms, from trivial to state-of-the-art.


Refer to the literature review:
Abstract section

The second section is a software simulation suite, more precisely a user-friendly graphical user interface that can (1) perform FRUC on any video using different FRUC algorithms, (2) benchmark interpolation algorithms using a industry-standard testbench, (3) interpolate the middle frame out of any given pair of frames and (4) estimate hardware resource usage of a specific FRUC algorithm. Moreover, the software suite would allow the engineer to fine-tune any of the algorithms, or implement new ones easily, to the desired level of quality and cost.


Refer to the software simulator source code in the GitHub repository:
https://github.com/lhl2617/Interpolatory/tree/master/Simulator

The third section features hardware resource estimation. One of the major limitations in hardware, especially when creating real-time solutions, is the data bandwidth available. Computer vision tasks are inherently data intensive and processing many frames in a second means that bandwidth requirements can sometimes become unfeasible. So, in order to determine which methods would be feasible in hardware it is required to estimate their bandwidth and cache data usage requirements. For a client this can be especially important because they will have limitations on how much bandwidth and cache space they can afford for the FRUC module. Hence, providing this information can save a lot of time for our clients when it comes to choosing which algorithm to use.

2. Design History

Echoing the three objectives of Interpolatory as detailed in Section 1.1, the project is split up into three phases, with each phase delivering its respective section. Interpolatory's work follows a 9-week weekly schedule from 27 April 2020 to 25 June 2020, and from this point onward, W1 (Week 1) shall refer to the week beginning 27 April 2020.

Figure 2.1 shows the actual timeline of our development, and was created post-completion of our project.

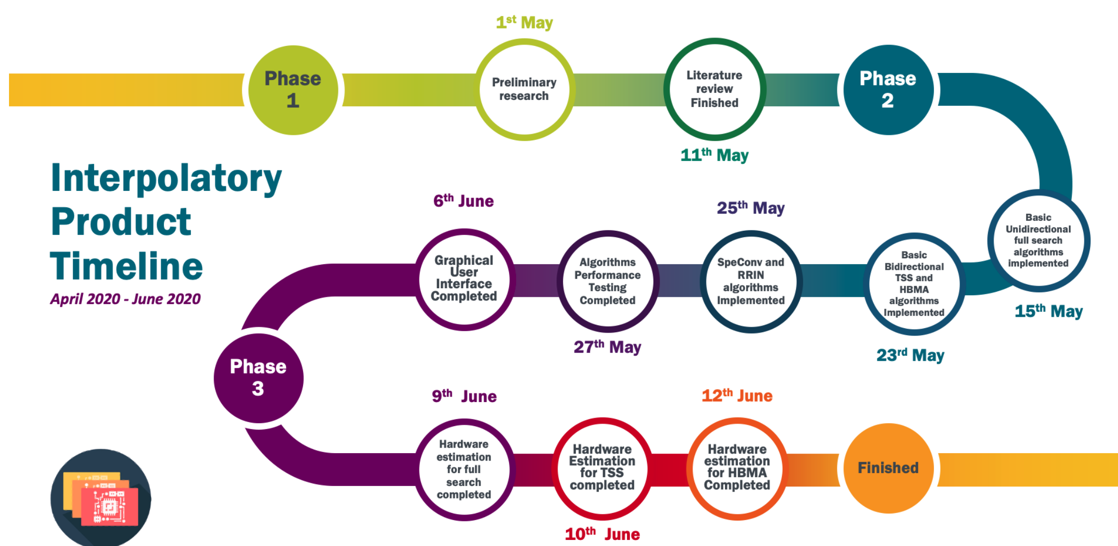


Figure 2.1: Timeline of Interpolatory product development

2.1 Kick-off

Period: W1.

Interpolatory's kick-off involves a kick-off meeting involving Kieron Turkington, advisor from Intel Corporation United Kingdom.

2.1.1 Meetings

Meeting 1 - W1 Monday (24 April 2020) 8:00AM to 8:30AM

Title

Project kick-off meeting

Attendees

All members of Interpolatory and Kieron Turkington, advisor from Intel Corporation United Kingdom.

Minutes

1. Delineation of high-level project plans and expectations from Kieron:
   - Explanation of Problem statement: Video Frame-rate Up-Conversion on FPGA where video is coming in raster scan order and streamed real time
   - Focus put on Frame-rate Up Conversion (FRUC) from 24 to 60 fps as down-conversion is trivial
   - A good literature review (Phase I) is sufficient
   - Software implementation (Phase II) of an algorithm and good testing would be a good end goal
   - Real-time IP (Phase III) is treated as a stretch goal and it is worth a lot of work
   - Software implementation of an algorithm and good testing would be a good end goal
   - Speed: should get results for a 5-minute long video within a couple of hours
2. Discussion of preliminary Phase I plans:
   - Literature review: look up research papers and gather as many papers as possible, filtering based on abstract and then look through for promising algorithms
   - Algorithm should not be too heavy on resources, consider DDR bandwidth with regard to hardware
   - To send three most interesting and feasible papers to Kieron for reading over the weekend

2.2 Phase I - Research

Period: W1-W2.

In the first two weeks, the main object of Interpolatory was to produce a literature review which is suitable for engineers without any prior experience in FRC. The intended purpose of the literature review was to give a review of FRC algorithms reviewed over the past few decades.

2.2.1 Meetings

Meeting 1 - W1 Monday (24 April 2020) 8:30AM to 9:30AM

Title

Preliminary Meeting

Attendees

All members of Interpolatory.

Minutes

1. Plans to kick-start Phase I:
   - Discussion of teamwork-related issues, including time-zones and best communication channels
   - Plan: each member review 5 papers based on abstract and relevant details located within to be presented in a sharing session in a catch-up meeting on Friday.
   - Brief introduction of skill sets held by each member to better split work in future work items

Meeting 2 - W1 Friday (1 May 2020) 11:00AM to 1:00PM

Title

Research Paper Sharing

Attendees

All members of Interpolatory.

Minutes

1. Presentation of research papers
   - Each member presents papers found and reviewed. Each member's presentation is followed by a brief discussion of its feasibility in Interpolatory's hardware solution context, as well its value to the project
2. Picking the most interesting papers
   - Further discussion to choose 3 most interesting papers to send to Kieron. The chosen papers are:
     (a) https://ieeexplore.ieee.org/document/5443548 : Motion-Compensated Frame Rate Up-Conversion—Part I: Fast Multi-Frame Motion Estimation This paper introduces a technique for motion estimation called MAP that outperforms BMA (block matching algorithm) techniques in quality and speed (for certain BMAs)
     (b) https://ieeexplore.ieee.org/document/5440975 : Motion-Compensated Frame Rate Up-Conversion—Part II: New Algorithms for Frame Interpolation Part 2 of the previous paper. Has good explanations for unidirectional and bidirectional motion vectors and proposes techniques to reduce hole and overlap issues
     (c) https://ieeexplore.ieee.org/document/7980587 : Motion compensation based multiple inter frame interpolation Performs slightly worse than other techniques but is much less computationally complex than other algorithms. May be more applicable to hardware. The related works section is a good introduction to MCFI methods.

Meeting 3 - W2 Monday (4 May 2020) 9:30AM to 10:00AM

Title

Literature review sync-up

Attendees

All members of Interpolatory and Kieron Turkington, advisor from Intel Corporation United Kingdom.

Minutes

1. Discussion of three research papers sent to Kieron:
   - Received feedback from Kieron about feasibility of algorithms presented in the papers in the hardware context
2. Planning a literature review:
   - Goals of literature review are set out:
     (a) Succinct review of papers from the last few decades on FRUC
     (b) Intended for engineers with little to no background in video processing to get up-to-speed with basic and advanced algorithms
     (c) Focus put in the context of hardware feasibility
3. Work distribution based on reviewed papers
   - Main outline of literature review put out: Introduction, Trivial methods, Basic method (MEMCI: Motion Estimation & Motion Compensated Interpolation), Advanced Methods
   - Based on research reviewed by each member, members are assigned to the four sections. The assignment is as follows:
     (a) Introduction: Naim
     (b) Trivial Methods: Jialong

(c) Basic method: Navid & Olly
(d) Advanced methods: Lin & Zhiyuan

MEETING 4 - W2 FRIDAY (8 MAY 2020) 11:00AM TO 11:30AM

Title
Literature review catch-up
Attendees
All members of Interpolatory.
Minutes

1. Discussion of work done so far on literature review
2. Cross-review of each others' sections
3. Discussion on how to better join parts written by each member to make it flow more like a story
4. Finalising details to be worked on over the weekend, including parts (i.e.
5. Sending of draft literature review to be reviewed by Kieron over the weekend

MEETING 5 - W3 MONDAY (11 MAY 2020) 2:00PM TO 2:30PM

Title
Literature review review & Kick off for Phase II
Attendees
All members of Interpolatory and Kieron Turkington
Minutes

1. Discussion of submitted literature review
    - Kieron satisfied with the quality and depth provided by the literature review
2. Closing off work in Phase I
3. Kicking off work in Phase II
    - Discussion of tools used: Python vs. Matlab
    - Description of what is realisable and expected to be performed in Phase II
        (a) a basic Motion Estimation & Motion Compensated Interpolation (MEMCI) method
        (b) able to process Full HD video of real videos

2.2.2 Plans & Requirements

Plans and requirements for the main deliverable –a literature review– was set out in MEETING 3. This is set out below:

- Requirements
    1. Succinct review of papers from the last few decades on FRUC
    2. Wide array of cited resources for further reading by reader
    3. Easy-to-read and suitable for engineers with little to no background in video processing to get up-to-speed on FRUC algorithms
    4. Focus is put on hardware feasibility, but advanced algorithms are included should hardware improve
- Plan
    1. The structure was split out into four parts, with members allocated to each part based on research reviewed in W1
        (a) Introduction: Naim
        (b) Trivial Methods: Jialong
        (c) Basic method: Navid & Olly
        (d) Advanced methods: Lin & Zhiyuan

2. Work on the literature review was planned to start on W2 Monday (Meeting 3: 4 May 2020)
3. The plan is to complete a draft by W2 Friday (Meeting 4: 8 May 2020) to be reviewed by Kieron on W3 Monday (Meeting 5: 11 May 2020).

There were no changes in plans and/or requirements throughout this phase of the project.

2.2.3 Design Review

Review of the literature review was done using two methods.

1. Cross-review amongst members
   - Members cross-reviewed work not assigned to them to check for flow, validity and soundness (Meeting 4)
   - Members each read through the review as a whole once before submission for review in Meeting 5
2. Review by Kieron from Intel
   - Read and reviewed by Kieron prior to Meeting 5
3. Review by George from Imperial College London
   - Received email and feedback on W6, suggest additions and improvements added

2.3 Phase II - Development: Software Simulator

Period: W3-W7 While Phase II was officially planned to start in W3 as kicked-off in Meeting 5 of Phase I, preliminary and experimental work were done in W1-2 to aid in both the literature review and to allow deciding the architecture used for Phase II.

2.3.1 Meetings

MEETING 1 - W3 TUESDAY (12 MAY 2020) 2:00PM TO 2:30PM

Title

Planning of Phase II

Attendees

All members of Interpolatory

Minutes

1. Discussion of preliminary work done in W1-2
   - Preliminary work was done by Lin in W1-2: this included a comparison of tools (Python vs Matlab), implementation of trivial FRUC algorithms (nearest, linear and oversample) and an interpolation testbench using the Middlebury dataset
   - Python was decided in agreement with Kieron in Meeting 5 of Phase I
   - Instructions were laid out on how to work on the existing Python code and extend the design to other algorithms
2. Planning & Splitting work
   - The project at this phase was split into four different sections, with clearly defined APIs:
     (a) ME (part of MEMCI)
       – Assigned to Naim and Olly
     (b) MCI (part of MEMCI)
       – Assigned to Navid, Jialong and Zhiyuan
     (c) Testbench, software architecture and dataset
       – Assigned to Lin
     (d) Advanced algorithms (ML) - Stretch goal
       – Assigned to Lin
3. Communication of plan to advisor
   - Discussion of timeline and expectations to be sent to Kieron after meeting

4. Discussion of marketing material
    - Lin to lead production of leaflet material (deadline: 5 June 2020) with information sourced from each member

Meeting 2 - W3 Wednesday (13 MAY 2020) 02:30PM TO 03:00PM

Title
MCI Sync up and Task Assignment
Attendees
Members assigned the MCI process: Navid, Jialong and Zhiyuan
Minutes

1. Clarification of the basic structure of MCI method.
2. Task arrangement was shown as follows:
    (a) Navid: implementing the unidirectional Interpolation and hole filling;
    (b) Jialong: testing of performance with combined MEMCI;
    (c) Zhiyuan: implementing the bidirectional method.

Meeting 3 - W5 Monday (25 May 2020) 11:00AM to 11:30AM

Title
Phase II Catch-up
Attendees
All members of Interpolatory
Minutes

1. Discussion of progress since start of Phase II
    - Lin
        (a) Implemented 2 ML algorithms with one not working well, could be cancelled
        (b) Implemented a GUI interface (unplanned)
        (c) Created a preliminary version of the leaflet to be discussed and improved upon
    - Olly & Naim
        (a) Completed three different methods of ME: FS, TSS and HBMA
        (b) Discussion of pros and cons of each method of ME
        (c) Implentation of Motion Vector Smoothing: Median, Mean, and Weighted-Mean
    - Navid
        (a) MCI: Ran benchmarks but still too slow to be tractable
        (b) Worked on filling holes
        (c) To use HBMA developed by Olly for more experiments
        (d) Worked on unidirectional MCI
    - Zhiyuan
        (a) Worked on bidirectional MCI
        (b) Results not as expected: worse than unidirectional
        (c) Will investigate the reason why
    - Jialong
        (a) Ran tests for unidirectional MEMCI with different combinations of parameters
        (b) Recorded metrics for different parameters
        (c) Found 5 best combinations based on Middlebury dataset
2. Discussion of further work
    - Lin
        (a) Update progress to advisors
        (b) Complete leaflet draft

(c) Finishing touches on GUI
(d) Experiment on final unsuccessful ML method

- Olly & Naim
  (a) Implement some chosen algorithms in C++ to speed up
  (b) Investigate further on HBMA method
  (c) Implement MAP which is a statistical approach used in hardware, investigate further
- Navid
  (a) Investigate further on MCI methods
  (b) Find out the feasibility of other datasets as testbench other than Middlebury, i.e. UCF101 and Vimeo90K
- Zhiyuan
  (a) Continue on investigation of bidirectional method
  (b) Write end-to-end testbench for MEMCI
- Jialong
  (a) Fine tune HBMA and investigate
  (b) Fine tune Weighted mean filter parameters
  (c) Work on MEMCI testbench

MEETING 4 - W6 MONDAY (1 JUNE 2020) 9:30AM TO 10:00AM

Title
Phase II Catch-up with Kieron
Attendees
All members of Interpolatory and Kieron Turkington
Minutes

1. Status update of MEMCI algorithms - results are O.K. and work as expected
2. Discussion on technical details of hardware implementation (Phase III)
3. Kick-off Phase 3

MEETING 5 - W6 MONDAY (1 JUNE 2020) 10:00AM TO 10:30AM

Title
Phase II Wrap up and Phase III kick off
Attendees
All members of Interpolatory
Minutes

1. Discussion of work distribution
   - Lin
     (a) Continue work on GUI and software interface
     (b) Begin work on video presentation deliverables
   - Olly & Naim
     (a) Read up on background of hardware estimations (Phase III), including DDR, cache schems, etc.
     (b) Discussion of deprecated MAP motion estimation method
   - Navid, Jialong & Zhiyuan
     (a) Fine-tune and wrap up MCI algorithms
     (b) Clean-up work on Phase II

MEETING 6 - W6 FRIDAY (5 JUNE 2020) 12:00PM TO 12:30PM

Title
Progress review with George

Attendees
All members of Interpolatory and George
Minutes

1. Leaflet discussion
    - Discussion of final leaflet draft prior to submission on same day
2. Literature review
    - Discussion of edits and updates done based on George's feedback
3. Progress report
    - Review of progress, and state of project in its three phases
    - Discussion of plans and predictions of project wrap-up statuses and deliverable work.

MEETING 7 - W6 FRIDAY (5 JUNE 2020) 12:30PM TO 1:00PM

Title
Phase II Wrap-Up
Attendees
All members of Interpolatory
Minutes

1. Discussion of work distribution
    - Everyone
        (a) Begin work on documentation by documenting findings, pitfalls and learnings from individual parts
    - Lin
        (a) Submit leaflet to college and George
        (b) Finish work on software GUI + document it
        (c) Start work on video
        (d) Obtain metrics from ML methods
    - Olly & Naim
        (a) Continue work on hardware estimates
        (b) Discussion on whether it is worth it to implement part of the algorithms in C++ to boost speed
    - Navid, Jialong & Zhiyuan
        (a) Record metrics and findings of MCI methods
        (b) Work on documentation

2.3.2 Plans & Requirements

As this Phase proved to be challenging and dynamic, the plans and requirements changed throughout the weeks. The original plans and requirements decided are laid out first. In subsequent sections when changes were required, they are documented.

Original Plans & Requirements

These items were decided in Meeting 1.

- Requirements
    1. Python Software framework allowing video FRUC from 24fps to 60fps using MEMCI and (stretch) advanced (ML) methods
    2. Implementation of best MEMCI variation based on MEMCI literature
    3. MEMCI post-processing pipeline to get rid of artifacts
    4. Production of leaflet marketing material
- Plan
    1. By end of W4:
        – Familiarise with existing Python framework and codebase

  - Set up type interfaces between ME and MCI to ease splitting work
  - Discuss further to choose best implementation of MEMCI
  - (Stretch) Set up minimal MEMCI pipeline
  - Draft leaflet created
2. By end of W5:
  - Achieved minimal MEMCI pipeline
  - Some ML algorithms implemented
  - (Stretch) MEMCI pipeline with multiple different ME/MCI algorithms
  - (Stretch) MEMCI postprocessing pipeline set up

In W4 and W6, additional requirements were added to improve the project.

W4 Plans & Requirements Changes

- Additional requirements
  1. GUI wrapper interface around the software framework to improve user experience
- Additional plans
  1. Lin to pick up GUI project and finish project by end of W4

W6 Plans & Requirements Changes

- Additional requirements
  1. Speed up MEMCI algorithms using `numba`, which JIT-compiles part of the inefficient Python code
  2. Review code
- Additional plans
  1. Lin to pick up usage of `numba` to speed up code by end of W6
  2. All to cross-review written code and test the software by end of W6

2.3.3 Outcome & Conclusion

Since a predominant amount of development was done in this phase, methodologies and results were compiled into numerous technical reports so as to organise our documentation in a more structured fashion. All of these reports can be found in Section 3.3 and each report documents a different aspect of our product.

2.4 Phase III - Development: Hardware Estimation

Period: W6-End Phase III started concurrently towards the end of Phase II, two members of the team, Naim and Olly, were put in charge of creating the hardware estimation for the algorithms we offered.

2.4.1 Meetings

MEETING 8 - W7 MONDAY (15 JUNE 2020) 11:00AM TO 11:30AM

Title
Phase II Wrap Up, Phase III Catch-up
Attendees
All members of Interpolatory
Minutes

1. Phase II Wrap-Up
   - Phase II is announced complete
   - GUI was built again with latest changes
   - Outstanding task of changing name from unidir2 to bidir2
2. Phase III Following estimations have been completed

  - Full-search w/ uni & bidir
  - HBMA w/ uni & bidir
  - Bidir might be unfeasible, need to look further
3. Presentation
   - Video Done
   - 4/5 slides complete
   - Need to add lit review in slide 2
   - Add PSNR and SSIM to slide 4
4. Lin
   - Build GUI
   - Add Slides
   - documentation GUI
5. Olly & Naim
   - Continue with hardware estimations
   - prepare speech for presentation
   - Add hardware estimations on slide 5
6. Navid
   - Documentation
   - outstanding task of changing name from unidir2 to bidir2
   - HBMA, ask olly/naim
7. Charles
   - Documentation sustainability report
   - Write email to george to ask about it
8. Bruce
   - Documentation - bidir & unidir
9. Documentation TODOs
   - Section 3
     - HBMA (navid)
     - MEMCI (bruce)
     - Clarify more on ML (LH)
     - GUI (LH)
   - Section 4
     - sustainability (Charles)

2.4.2 Plans & Requirements

The intention of this phase was to provide an initial estimate of the DRAM bandwidth and on-chip memory requirements for the algorithms provided in the simulation suite, so as to confirm that they were feasible in hardware.

- Requirements
  1. Provide cache schemes for each algorithm to reduce DRAM bandwidth
  2. Estimate resource footprint for all cache schemes to determine if implementations were hardware feasible
  3. Create scripts which could be integrated into the GUI in order to provide the information to the end user
- Plan
  1. Research current methods when creating cache schemes
  2. Start by estimating bandwidth and resource consumption for non-caches algorithms

3. Construct and record proposed cache schemes in a formal and structured manner including estimation for DRAM bandwidth and on-chip memory usage
4. Create estimations scripts and integrate into GUI

3. Reports

3.1 Sustainability Report

Environment
Climate change is one of the most, if not the most, pressing issues of our world today. Interpolatory recognises the urgency of climate change and the importance of sustainability. Interpolatory believes we are responsible, and are committed, to tackle this global challenge. Interpolatory is proud to state that research, development and production of the product does not contribute to climate change in any way.

Economics
Interpolatory targets the worldwide TV-makers as potential customers. Hence, the frame rate up conversion feature that Interpolatory provides may affect the purchase decision made by the worldwide 1.67 billion TV-owners and the potential TV-owners.

Social Impacts
Televisions which are equipped with Interpolatory FRUC features can play any videos at increased frames per second. This means that the TV-owners can watch videos which are smoother. Therefore, videos with educational purposes such as documentaries can provide the viewers with a more vivid watching experience. On the other hand, the higher quality provided by Interpolatory may encourage longer TV watching time, which was previously proven to associate with obesity, language delays and learning disabilities [1]. Hence, parental control is encouraged to complement the lack of self-control for young children in TV watching.

3.2 Ethical Consequences Report

Research & Development
Research and development done in this product is carefully audited and checked by our advisors at Imperial College London and Intel Corporation UK. Interpolatory's research & development, where applicable, follow the ethics code [2] set forth by Imperial College London.

Use of external sources
All external work, research, and intellectual property are cited and attributed exhaustively to their respective authors. Code snippets, algorithms and/or models from external sources in the GitHub repository are

checked for their licenses, and where required, Interpolatory obtained permission for use in this project. Where applicable, links to sources of code, included but not limited to links to GitHub repositories, are included.

External sources are also checked and confirm to have no ethical violation made. In addition to using preferred sources from reputable sources (e.g. ACM and IEEE), where applicable, sources are checked for ethical violations.

Collection of Data

No personal data collection was necessary and henceforth was not carried out in the project. Interpolatory follows guidance laid out by GDPR.

Usage

The usage of the Interpolatory IP suite does not cause any harm to the user. There is no direct nor indirect impact to the climate changed, as echoed by the statement in the **Sustainability Report**.

3.3 Technical reports

3.3.1 Benchmarks & Performance Metrics

The datasets used in benchmarking are the Middlebury [3] and UVG (Ultra Video Group) [4] datasets. The Middlebury dataset is used to obtain a quantitative measurement on how well the frame interpolation methods perform and the UVG dataset is used to compare the interpolation methods qualitatively.

Middlebury

Middlebury was first used as the evaluation dataset for optical flow [3]. In recent years, it is commonly used as a testbench for examining the performance of frame rate up conversion (FRUC) algorithms. This dataset contains sets of images of moving object. Twelve image sets are chosen by Interpolatory as the testbench dataset. The chosen image sets contains objects ranging from small item like tennis ball to larger object such as a skyscraper. Each image set consists of three consecutive frames, the first frame is used as reference frame, the last frame is the target frame and the middle frame is used as the ground truth frame. The interpolated frame is compared with the ground truth frame to produce PSNR and SSIM scores which are used to evaluate the performance of the interpolation algorithms.

Peak signal-to-noise ratio (PSNR) is calculated by:

$$\text{PSNR} = 10\log_{10}\left(\frac{\text{max}_i^2}{\text{MSE}}\right)$$

where max_i^2 is maximum range of the input data type and MSE is the mean square error between the images. This score is commonly used to examine how well the interpolated frame is reconstructed comparing to the ground truth frame.

The Structural Similarity Index (SSIM) is calculated by:

$$\text{SSIM}(x,y) = [l(x,y)]^{\alpha} \times [c(x,y)]^{\beta} \times [s(x,y)]^{\gamma}$$

where $l(x,y)$ is the luminance comparison between signal x and signal y, $c(x,y)$ is the contrast comparison and $s(x,y)$ stands for structural comparison. α, β and γ are the weights of each comparison, normally defaulted to 1. This metric is used to measure the similarity between two images. Please refer to the Literature review for a more detailed explanation.

R Refer to the literature review:
Definitions of IE, PSNR and SSIM section

UVG (Ultra Video Group)

The UVG dataset consists of 16 4K test videos which are captured at 50/120 fps [4]. This dataset is available under a non-commercial Creative Commons BY-NC license. Interpolatory chose 7 versatile test videos from the UVG dataset and downsampled them to 15/24 fps. The downsampled videos were upsampled to 60 fps by different FRUC methods and then compared to the original videos. We used this dataset to assess the FRUC methods qualitatively. After a large amount of testing on Middlebury and UVG datasets, Interpolatory's quality assurance team concluded that the FRUC methods which have high SSIM and PSNR scores from the Middlebury dataset also outputs upsampled video with better quality and show resemblance to the original video. This meant that there is a direct correlation between the PSNR and SSIM scores and the output video quality.

3.3.2 Trivial Frame Interpolation methods

R Refer to the literature review:
Basic Interpolation Methods section

Frame repetition

Frame repetition is one of the most trivial frame rate up conversion methods. The source frames from the original video str repeated according to the up conversion ratio. For instance, a 24-to-60 fps FRUC requires a 2.5 up conversion ratio. The implementation of this method simply returns the frame with frame_index = frame_index/up conversion ratio.

Oversampling

Oversampling is a improved FRUC method base on Frame repetition. Instead of repeating the source frames, the new frames are produced by blending the two adjacent source frames with different weights. The implementation of oversampling is shown in Algorithm 1.

Algorithm 1: Oversampling

Find the latest source frame key_frame
Find the offset from the target frame $offset$
Find the distance from the next blending frame $blending_position$
if $blending_position >= 1$ **then**
| return frame[key_frame / up conversion ratio + $source_frame_offset$]
else
| return blend frames according to $blending_position$
end

Linear interpolation

Linear Interpolation algorithm interpolates new frames by blending the corresponding adjacent source frames with linear weights. The implementation is shown in Algorithm 2.

Algorithm 2: Linear Interpolation

Frame_A_index = Floor(Frame_index / up conversion ratio)
weight_A = 1 - (Frame_index / up conversion ratio - Frame_index)
Frame_B_index = Frame_A_index + 1
weight_B = 1 - weight_A
return Blend Frame A and Frame B

The result shown in Table 3.1 suggests that the Linear Interpolation method performs the best on Middlebury dataset. The PSNR and the SSIM scores of Linear Interpolation is 16% and 6.6% higher than

Methods	PSNR	SSIM	Time (s)
Frame Repetition	24.8	0.753	1
Oversampling	24.8	0.753	1
Linear Interpolation	28.0	0.804	2

Table 3.1: Performance of trivial methods on the Middlebury dataset.

the other two methods respectively. However, the execution time of the Linear Interpolation is 1s slower than the other two methods.

3.3.3 MEMCI Frame Interpolation methods

Motion Estimation

The objective of the motion estimation algorithms is to estimate the motion of moving objects between consecutive frames. This is done by finding motion vectors that represent the spatial differences of moving objects between the frames, which are later used for the interpolation part. Due to their relatively low complexity and compatibility with hardware implementation, we decided to implement three different block matching algorithms (BMA) for motion estimation. BMA divides the current frame into $b \times b$ sized blocks and search for the $b \times b$ block in a reference frame that minimizes the matching error function. Sum of absolute differences (SAD) is used as matching error function in Interpolatory due to its low computational cost. The ME algorithms implemented in Interpolatory are Full Search, Three Step Search and Hierarchical Block Matching. It should also be mentioned that it is made easy to integrate additional ME algorithms in Interpolatory and combine them with the existing MCI methods.

Refer to the literature review:
Motion Estimation section

Full Search

Full search is a simple ME algorithm that compares the SAD between the source block and all $b \times b$ blocks within a $w \times w$ search region centered around the location of the source block in the target frame. This is demonstrated in Figure 3.1. The motion vectors of the source block is then obtained as the spatial difference between the source block and the block in the target frame with the lowest SAD. The full search algorithm therefore takes source frame, target frame, block size and size of the search region as input arguments. It then returns motion vectors for each pixel together with the corresponding SAD.

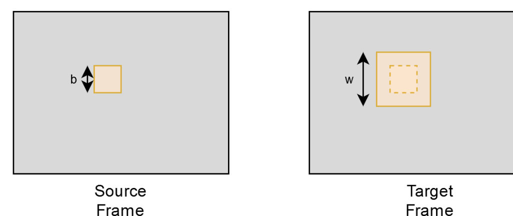


Figure 3.1: A source block (left) and the region it performs the full search in the target frame (right)

Three Step Search

The three step search (TSS) is an ME algorithm that requires significantly fewer block comparisons than the FS. TSS benefits from an efficient search scheme which is shown in Figure 3.2. In the first step, the matching error from 9 blocks in a 3 by 3 grid centered at the location of the source block is compared.

The block with the lowest matching error is then the center point for the second step where a smaller 3 by 3 grid is used. This procedure is repeated for a given number of steps and the obtained motion vector then points from the source block to the best matching block in the last step.

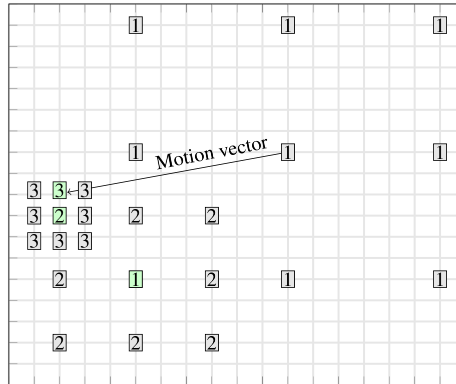


Figure 3.2: Search pattern of the three step search. The numbers corresponds to the current step number and green points indicates where the lowest matching error in each step is located.

Hierarchical Block Matching Algorithm

For the Hierarchical Block Matching Algorithm (HBMA) the blocks size has to be power of 2. The algorithm can be described in three main steps.

1. The HBMA first reduces the size of the source and target frame by convolving the frames with the 2D filter

$$w = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \quad ,$$

   using 2×2 strides. This is performed a given number of times s and reduces the height and width of the frames by a factor 2^s. Full search is then performed on the down scaled frames to obtain the initial motion vectors.
2. In this step we want to increase the motion vector density by doubling the size of the motion vector field. This step is repeated s times which will result in a motion vector field with the same size as the original frame. Because the image size is halved when down-scaled, given a consistent block size, a (“parent") block in the down-scaled frame encompasses 4 (“child") blocks in the up-scaled frame, see Figure 3.3. For each of the 4 child blocks of every parent block in the down-scaled image: Search areas in a (3×3) window around position pointed to by parent vector, and the 2 vectors associated with 2 blocks adjacent to the parent block. Select vectors corresponding to the smallest SAD. To account for the change in image size we also need to multiply the parent vector by 2 to find the right search area for the child block.
3. Now we want to further increase the motion vector density by reducing the block size. This is done by halving the block size, each previous ("parent") block holds 4 ("child") blocks, similar to what is shown in Figure 3.3, however the child blocks have a size of $b/2$ instead of b. For each of the 4 child blocks of every parent block in the original image: Search areas in a (3x3) window around position pointed to by parent vector, and the 2 vectors associated with 2 blocks adjacent to the parent block. Select vector corresponding to the smallest SAD. This step is repeated until the desired block size is obtained. The HBMA finally returns a block-wise motion vector field.

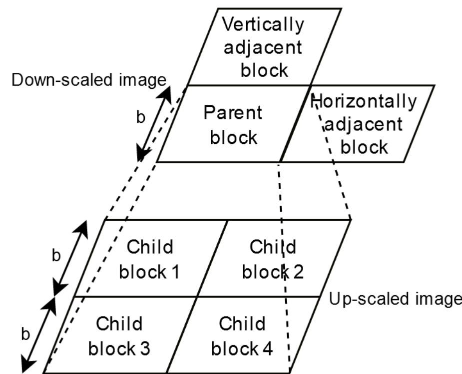
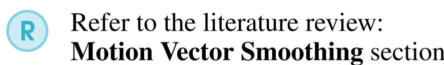


Figure 3.3: The parent block and it's corresponding child blocks in the up-scaled image

Motion vector smoothing

Motion vector smoothing is used to enhance the performance of block-matching algorithms, by reducing deformable motion and noise. Three trivial motion vector smoothing methods are implemented and shown below.


Refer to the literature review:
Motion Vector Smoothing section

Mean filter
The mean filter simply takes the average value of the surrounding eight motion vector blocks and the current motion vector block as the new smoothened motion vector block.
Median filter
The median filter first sorts the current motion vector block and the eight surrounding blocks. Then, the median block is chosen as the new current motion vector block.
Weighted mean filter
The weighted mean filter smoothen the motion vector block by averaging the 3×3 motion vector blocks with pre-tuned weights.

Smoothing filter	**ME**								
	HBMA			Full Search			Three Step Search		
	PSNR	SSIM	Time (s)	PSNR	SSIM	Time (s)	PSNR	SSIM	Time (s)
None	28.9	0.886	9	27.3	0.836	15	26.4	0.813	6
mean	29.1	0.894	9	27.2	0.833	15	26.2	0.809	7
median	29.2	0.894	9	27.5	0.843	15	26.7	0.827	6
weighted mean	28.6	0.885	9	27.0	0.830	15	26.1	0.807	6

Table 3.2: Performance of smoothing methods on the Middlebury dataset.

Table 3.2 shows the PSNR and SSIM and execution time of the unidirectional MCI method with different motion vector smoothing filters (these times are recorded on `corona50.doc.ic.ac.uk` (Intel Core i7-8700, 16GB RAM)) The result suggests that the median filter improves the PSNR and SSIM score regardless the motion estimation method used. However, the mean and weighted mean filter do not show consistent improvement on PSNR and SSIM scores. Hence, median filter is recommended as the motion vector smoothing filter when needed.

Motion Compensation Interpolation

Refer to the literature review:
Motion Compensation Interpolation section

Basic unidirectional

Unidirectional interpolation is the most trivial approach for MCI method. It uses one motion vector generated from only one direction to predict the motion of the pixel. The algorithm is show in Algorithm 3.

Firstly, by calling an ME function, the pixel-wise motion vectors for the whole image are obtained. The motion vectors are stored in a 2D-array, where the index corresponds to the coordinates of each pixel. The content of the motion vector includes the direction and magnitude of the predicted movement towards the target pixel and the SAD value.

Depending on the number of frames that need to be interpolated, the coordinates of the interpolated pixel can be calculated by adding a fraction of the motion vector to the coordinates of the current pixel. For example, if two frames needs to be interpolated between the source frame and the target frame, the coordinates of the first interpolated frame will be $source_frame + MotionVector \times \frac{1}{3}$ and the second frame will be $source_frame + MotionVector \times \frac{2}{3}$.

If multiple pixel blocks end up in the same interpolated pixel, an overlapping occurs. To resolve the overlapping issue, Sum of absolute differences a metric known as SAD is used. SAD measures the distance from the prediction to the ground truth. Hence, the pixel with smallest SAD would be the final choice. An three dimensional array containing the X,Y coordinate and SAD score of the pixel is used. The implemented algorithm selects the pixel with the lowest SAD score. The array Map_{SAD} stores the SAD score of each pixel. This algorithm has a computational complexity of $O(n)$ where n represents the resolution of the image.

Each pixel in the resultant frame is initialised with value equals to -1. After interpolation, holes (blank pixels) may be identified in the resultant image (pixels with value set to -1). A median filter is applied to fill these holes.

Algorithm 3: Unidirectional MCI

MotionField = ME($Frame_1$, $Frame_2$)
Initialise $Frame_{output}$ with all pixel defaulted to [-1, -1, -1]
Initialise Map_{SAD} with all pixel's SAD defaulted to infinite
for *each pixel* **do**
  (u_i, v_i) = (u, v) + ratio * MotionField[u,v].MotionVector
  if $MotionField[u,v].SAD < Map_{SAD}[u_i, v_i]$ **then**
    $Frame_{output}[u_i, v_i]$ = $Frame_1$[u,v]
    $Map_{SAD}[u_i, v_i]$ = MotionField[u,v].SAD
  end
end
for *each pixel* **do**
  if $pixel == [-1,-1,-1]$ **then**
    pixel = median(kernel)
  end
end
return($Frame_{output}$)

Basic bidirectional

Implementation

The Bidirectional method is a further improvement from unidirectional. It makes use of the unidirectional method, as per the pseucdocode shown in Algorithm 4. However, compared to the unidirectional method, the bidirectional method combines both forward and backward motion vectors. Similar to unidirectional method the forwards interpolation is first performed then, as a new step, the backwards interpolation is executed.

After both the forward and backward images have been interpolated holes must be filled. While this method does produce less holes than unidirectional there are still some left over which are filled using the

method proposed in the unidirectional implementation.

Algorithm 4: Bidirectional MCI

```
ForwardMotionField = ME(Frame_1, Frame_2)
BackwardMotionField = ME(Frame_2, Frame_1)
Initialise Frame_output with all pixel defaulted to [-1, -1, -1]
Initialise Map_SAD with all pixel's SAD defaulted to infinite
for each pixel do
    (u_i, v_i) = (u, v) + ratio * ForwardMotionField[u,v].MotionVector
    if ForwardMotionField[u,v].SAD < Map_SAD[u_i, v_i] then
        Frame_output[u_i, v_i] = Frame_1[u,v]
        Map_SAD[u_i, v_i] = ForwardMotionField[u,v].SAD
    end
    (u_i, v_i) = (u, v) + (1-ratio) * BackwardMotionField[u,v].MotionVector
    if BackwardMotionField[u,v].SAD < Map_SAD[u_i, v_i] then
        Frame_output[u_i, v_i] = Frame_2[u,v]
        Map_SAD[u_i, v_i] = BackwardMotionField[u,v].SAD
    end
end
for each pixel do
    if pixel == [-1, -1, -1] then
        pixel = median(kernel)
    end
end
return(Frame_output)
```

Improvements & attempts

The first improvement to interpolation is focused on what happens when two pixels overlap. Using the pixel with lower SAD would only take into account of only one pixel. However the motion vector shows that there could probably be multiple contribution from different blocks. Therefore a weighted average is proposed. Since the one with lower SAD should contribute more, the final choice is $result = pixel_{low} * SAD_{high}\% + pixel_{high} * SAD_{low}\%$. This new method shows that less artifacts in the interpolated frames.

The second improvement is the median filter at hole-filling. The initial method was to return the median in the region within the kernel size. However, even with this holes occur, the reason is the empty pixels are not excluded. For example, when a kernel with more than half of the pixels are empty, the result returned would remain empty. The first change proposed by Charles, was to fill the holes with the corresponding pixels in the original frame. This method firstly eliminates the holes and works well for relatively static region. Bruce proposed the second method, which was to perform the median filtering after filtered out the empty pixels. The second method, generates higher PSNR and SSIM scores, and works better for both static and dynamic regions. The drawback remains is that when an empty block is larger than the kernel size, the result is still empty.

The mean filter was also been tested. But the results show it is slightly worse than the median filter (around 0.07%).

The last improvement revolved around post-processing. There are a lot of artifacts even after the interpolation and hole-filling. Thus, some smoothing filters(average filter, Gaussian filter, median filter, etc.) are applied in order to get a better quality of image. But the result was not what was hope, the images become blurrier after the filter applied. Even though the artifacts are reduced a little, the cost of quality (measured by PSNR and SSIM) were unacceptable. The cause of the massive drop in performance was due to the low resolution of test sets. For high resolution images, the smoothing is decent, but the time cost is too large.

Advanced bidirectional

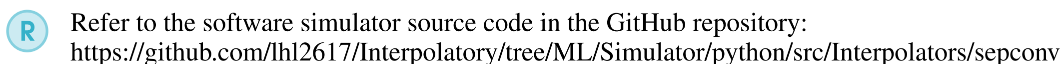
In an attempt to further improve the MCI methods, a decision was made to implement an advanced bidirectional algorithm. This algorithm is an extension to the algorithm described in section 3.3.3 and therefore requires both backward and forward motion estimation. The algorithm is based the one presented by Wang *et al.*[5] and uses expanded overlapped blocks to reduce blocking artifact in the interpolated frame. In the first step, two intermediate frames are produced, using the forward and backward motion field respectively. The two frames are then combined to one intermediate frame using an error adaptive scheme. Finally, the output frame is obtained by filling the holes in the intermediate frame using block-wise directional hole interpolation. For a more detailed description of the algorithm we refer the interested reader to [5]. Although this specific algorithm may not be suitable for hardware implementation, it serves as a demonstration of achievable performance of advanced MCI methods.

3.3.4 Machine Learning Interpolation methods

In the literature review, state-of-the-art methods were discussed, with focus given to the current best five methods. All methods incorporate some form of Machine Learning, and the details are covered in the literature review.

For Interpolatory, we decided to implement two selected methods that were interesting: **SepConv** and **RRIN**. These methods are picked for their novelty and ease of implementation. Whilst it is not currently reasonably feasible to implement these methods on hardware, it is useful to the user as an indication of how state-of-the-art methods perform.

SepConv

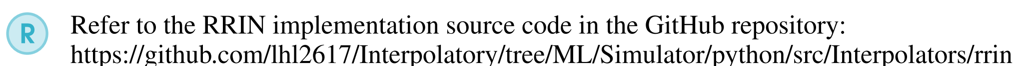


SepConv is a kernel-based interpolation method, in which frame interpolation is formualted as convolution operators over patches instead of relying on optical or motion flow. This rids the disadvantages of optical and motion flow, which includes requiring a post-processing step to remove image artifacts.

Proposed by Niklaus *et al.*[6], SepConv is the improved version of the AdaConv model [7]. SepConv uses less memory and produces better interpolation results. More details can be found in the literature review.

Interpolatory's implementation of SepConv is based on Niklaus *et al.*'s PyTorch implementation [6], and was able to reproduce similar metrics and results as reported by the original paper.

RRIN



RRIN (Residue Refinement video frame Interpolation Network) [8] exploits residue learning [9] and adaptive weight map for accurate video frame interpolation. The submodules are implemented using U-Net, which effectively reduces the model size and computation complexity. However, Interpolatory's

implementation which was based on Li *et al.*'s implementation [8] was not able to interpolate Full-HD (1920 × 1080) video frames due to insufficient CUDA memory on an 8GB NViDIA GTX-1080 (SepConv was able to interpolate Full-HD footage).

Refer to the literature review:
Other Methods section

3.3.5 Hardware Estimations

Motivation & Goals

Since the end goal of Interpolatory was to produce RTL code it was decided that we would begin the initial stage of implementing our current array of algorithms in hardware. Under advisement from our industry supervisor we decided to calculate DRAM bandwidth estimates and design cache structures to argue that it was possible to fit our algorithms on modern FPGAs. It should be noted that at the time of writing it is unfeasible to implement CNNs on FPGAs and thus while it is possible to calculate DRAM bandwidth for the more advanced algorithms the estimates would ultimately be useless and are therefore skipped.

The cache schemes are given as individual markdown documents in the appendix, including the equations to calculate the bandwidth and cache requirements along with an example calculation for the default parameters.

Basic Algorithms

Frame repetition - A.1.1
Oversampling - A.1.2
Linear - A.1.3

MEMCI

Full search and unidirectional - A.2.1
Three step search and unidirectional - A.2.2
Hierarchical block matching algorithm and unidirectional - A.2.3
Full search and bidirectional - A.2.4
Three step search and bidirectional - A.2.5
Hierarchical block matching algorithm and bidirectional - A.2.6
Full search and advanced bidirectional - A.2.7
Three step search and advanced bidirectional - A.2.8
Hierarchical block matching algorithm and advanced bidirectional - A.2.9

3.3.6 Graphical User Interface

To help our customers quickly explore and prototype different FRUC algorithms which Interpolatory provides, a highly-featured graphical user interface Interpolatory Simulator was developed. The tool has four core features covering most of the work done in Phase 2 and 3. Figure 3.4 shows a screenshot of the home page of the interface.

The first feature allows user to operate video frame conversion for any video. Users can choose the target framerate and the frame-rate conversion algorithm applied. The output video can be previewed, so that the users can quickly check the algorithm's quality.

The second feature is benchmarking in which users can run the industry standard Middlebury benchmark on the algorithms. This feature is designed to be simple to use, the user can simply select the testing parameters for the algorithms and run the benchmark. Upon completion, benchmark SSIM and PSNR scores are given, allowing users to compare algorithms. In addition, the interpolated frames are produced for visual inspection of the output.

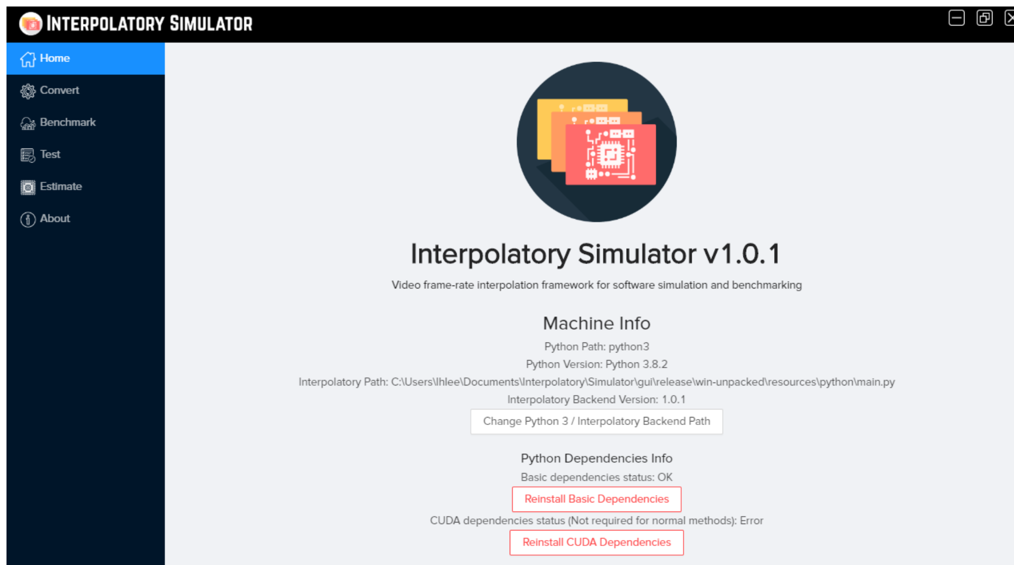


Figure 3.4: Home page of Interpolatory Simulator Graphical User Interface

The third feature allows the users to interpolate any pair of frames, effectively allowing the creation of custom benchmarks. Again, quantitative metrics SSIM and PSNR scores are shown, together with the actual output for quantitative analysis.

The fourth and final feature is hardware resource estimation. This feature allow engineers to quickly estimate how much hardware resource is used by a certain algorithm at a certain frame size.

The graphical user interface is available for Windows, macOS and Linux. A Windows installer is available in the releases section of the repository, whilst macOS and Linux users can build a copy based on the source code.

Refer to the software simulator GUI source code in the GitHub repository:
https://github.com/lhl2617/Interpolatory/tree/master/Simulator/gui

Users not wishing to use the graphical user interface can also opt to use the command line interface, which is similarly featured.

3.3.7 Numba

To speed up benchmarking and video conversion, a open source JIT compiler, `Numba` is used. `Numba` speeds up Python code by translating Python functions to optimized machine code at runtime using the industry-standard `LLVM` compiler library. The translated Python functions can approach the speed of C. Without `Numba`, a 5 second 24 fps video can take up to 2 hours to be up-converted to 60 fps. After all functions are translated into machine code by `Numba`, a 24 to 60 fps up-conversion only takes less than 10 minutes. The execution time of running a complete Middlebury benchmark for 24 to 60 fps up conversion is reduced from half an hour to less than a minute.

3.3.8 Results

As expected, the Advanced Bidirectional method obtains the highest PSNR and SSIM out of all MCI methods implemented. Table 3.3 shows that the Advanced Bidirectional method together with HBMA performs the most accurate interpolation. The results for the non-ML methods (i.e. all except SepConv and RRIN) are run on `corona50.doc.ic.ac.uk` (Intel Core i7-8700K, 16GB RAM), whereas the ML methods are run on `ray25.doc.ic.ac.uk` (Intel Core i7-7700K, 8GB RAM, NViDIA GTX 1080).

Because the ML methods use CUDA, they cannot be compared to the non-ML methods, as the non-ML methods use the CPU instead of the GPU.

Note that the runtime reported is just an weak indication of the complexity of the algorithm and should not be taken to gauge an algorithm's complexity. Because there are implementation and optimisation specifics, these reported runtimes should just be used for comparison between the algorithms. Moreover, as the ultimate target of these algorithms is hardware, these software simulation runtime results are not indicative of the algorithm's performance on hardware.

MCI	**ME**								
	HBMA			Full Search			Three Step Search		
	PSNR	SSIM	Time (s)	PSNR	SSIM	Time (s)	PSNR	SSIM	Time (s)
Unidirectional	28.9	0.886	12	27.3	0.835	18	22.5	0.683	17
Bidirectional	30.5	0.910	15	29.8	0.872	26	23.0	0.711	15
Advanced Bidirectional	33.5	0.945	19	30.8	0.907	29	28.6	0.906	11

Table 3.3: Performance by MEMCI methods on the Middlebury dataset.

FRUC method	PSNR	SSIM	Time (s)
SepConv	35.2	0.954	10
RRIN	34.2	0.954	5
Advanced Bidirectional	33.5	0.945	19
Bidiredtional	30.5	0.910	15
Unidirectional	28.9	0.886	12
Linear	28.0	0.804	5
Oversample	24.7	0.752	5
Nearest	24.7	0.752	5

Table 3.4: Performance by all FRUC methods on the Middlebury dataset.

3.3.9 Future Improvements

There were few interesting points when developing the MEMCI methods. But due to the time limits, these issues has not been investigated.

Object detection

As the results from MEMCI and RRIN shown Figure 3.5, the most obvious difference is that MEMCI has 4 blurry balls whereas RRIN has 2 solid balls. The reason is that the motion estimation of MEMCI only on block-wise, however the RRIN has done the object detection, knowing that the ball is an single object which all pixels of the ball object should move consistently.

Figure 3.5: Results using MEMCI and RRIN

A different paradigm of bidirectional interpolation

There are two different bidirectional interpolation methods are found during literature review. Both of them are referred as bidirectional interpolation in different papers. The reason not developing this one is that it requires a different way of motion estimation indicated in Figure 3.6. This requires the motion estimation to perform which takes two time flows into account.

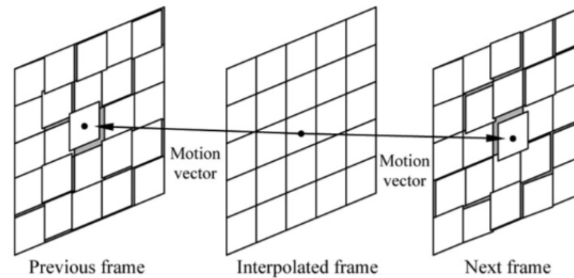


Figure 3.6: Different bidirectional interpolation

Image pyramid

The block search in motion estimation can also use various size of blocks. But smaller would increase the computational time exponentially, because for each of different size, the search would done to the whole image. And through the test, it is found that the the ME process takes much longer time than MCI process. Consider the currently results are still slow than expectation, this method could be a less interesting way of investigation.

Bibliography

[1] Urban Children Institute, "Infants, Toddlers and Television." `http://www.urbanchildinstitute.org/articles/policy-briefs/infants-toddlers-and-television`, 2016. Online; accessed 25 June 2020.

[2] Imperial College London, "An ethics code for imperial college london." `https://www.imperial.ac.uk/media/imperial-college/research-and-innovation/public/research-integrity/Ethics-Code.pdf`, 2013. Online; accessed 25 June 2020.

[3] S. Baker, S. Roth, D. Scharstein, M. J. Black, J. P. Lewis, and R. Szeliski, "A database and evaluation methodology for optical flow," in *2007 IEEE 11th International Conference on Computer Vision*, pp. 1–8, 2007.

[4] A. Mercat, M. Viitanen, and J. Vanne, "Uvg dataset: 50/120fps 4k sequences for video codec analysis and development," in *Proceedings of the 11th ACM Multimedia Systems Conference*, MMSys '20, (New York, NY, USA), p. 297–302, Association for Computing Machinery, 2020.

[5] D. Wang, A. Vincent, P. Blanchfield, and R. Klepko, "Motion-compensated frame rate up-conversion—part ii: New algorithms for frame interpolation," *IEEE Transactions on Broadcasting*, vol. 56, no. 2, pp. 142–149, 2010.

[6] S. Niklaus, L. Mai, and F. Liu, "Video frame interpolation via adaptive separable convolution," in *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 261–270, 2017.

[7] S. Niklaus, L. Mai, and F. Liu, "Video frame interpolation via adaptive convolution," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2270–2279, 2017.

[8] H. Li, Y. Yuan, and Q. Wang, "Video frame interpolation via residue refinement," in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2613–2617, 2020.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

A. Hardware Estimation Cache Schemes

A.1 Basic Algorithms

A.1.1 Frame Repetition

```
# Frame Repetition Interpolation :

## Parameters

- `r` = number of rows of pixels in frame
- `c` = number of columns of pixels in frame

## Frame Repetition Algorithm

- As frame is streamed in, store in DRAM and replace previous frame.
- When new frame is scheduled to be streamed out, stream out frame in DRAM

## Hardware Estimation

### DRAM Writing Bandwidth:

- For each incoming frame (1/24 s):
    - Entire frame is written to DRAM:
        - 3 * `r` * `c` bytes
- Total :
    - 72 * `r` * `c` bytes / s

### DRAM Reading Bandwidth:

- For each outgoing frame (1/60 s):
    - Entire frame is read from DRAM
        - 3 * `r` * `c` bytes
- Total :
```

```
    - 180 * `r` * `c` bytes / s

### Required Cache Size

- None
    - Since you do read the same frame from DRAM multiple times you could
        ↪ potentially cache the frame, however this may cause an unfeasible
        ↪ resource footprint on the board.


### Example Estimations:

For:
- `r` = 1080
- `c` = 1920

Results:
- DRAM write bandwidth = 149.3 MB/s
- DRAM read bandwidth = 373.2 MB/s
- Required cache size = 0 MB
```

A.1.2 Oversampling

```
# Oversampling Interpolation :

## Parameters

- `r` = number of rows of pixels in frame
- `c` = number of columns of pixels in frame

## Oversampling Algorithm

- Keep 2 frames in DRAM
- As frame is streamed in, if both frame slots in DRAM are filled replace oldest
    ↪  frame.
    - Also create all interpolated frames using the frame being streamed in and
        ↪ the newer frame in DRAM
        - If either of the weightings for the source and target frame are 0 then
            ↪ do not write-back the interpolated frame (The interpolated frame,
            ↪ in this case, is a copy of either the source or target frame)
- Stream out scheduled frame from DRAM

## Hardware Estimation

### DRAM Writing Bandwidth:

- For each incoming frame (1/24 s):
    - Entire frame is written to DRAM:
        - 3 * `r` * `c` bytes
```

```
    - Interpolated frames are written to DRAM
        - (3 * `r` * `c`) / 4 bytes
        - For this specific frame rate conversion (24-60) only one in every 4
          ↪ interpolated frames are written to DRAM (due to the conditional
          ↪ write-back specified above) so this bandwidth is divided by 4
- Total :
    - 90 * `r` * `c` bytes / s


### DRAM Reading Bandwidth:

- For each incoming frame (1/24 s):
    - Previous frame is read from DRAM
        - 3 * `r` * `c` bytes
- For each outgoing frame (1/60 s):
    - Entire frame is read from DRAM
        - 3 * `r` * `c` bytes
- Total :
    - 252 * `r` * `c` bytes / s

### Required Cache Size

- None

### Example Estimations:

For:
- `r` = 1080
- `c` = 1920

Results:
- DRAM write bandwidth = 186.6 MB/s
- DRAM read bandwidth = 522.5 MB/s
- Required cache size = 0 MB
```

A.1.3 Linear

```
# Linear Interpolation :

## Parameters

- `r` = number of rows of pixels in frame
- `c` = number of columns of pixels in frame

## Linear Algorithm

- Keep 2 frames in DRAM
- As frame is streamed in, if both frame slots in DRAM are filled replace oldest
    ↪  frame.
```

```
- Also create all interpolated frames using the frame being streamed in and the
  ↪ newer frame in DRAM
    - If either of the weightings for the source and target frame are 0 then do
      ↪ not write-back the interpolated frame (The interpolated frame, in this
      ↪  case, is a copy of either the source or target frame)
- Stream out scheduled frame from DRAM

## Hardware Estimation

### DRAM Writing Bandwidth:

- For each incoming frame (1/24 s):
    - Entire frame is written to DRAM:
        - 3 * `r` * `c` bytes
    - Interpolated frames are written to DRAM
        - (3 * `r` * `c`) * 2 bytes
        - For this specific frame rate conversion (24-60) every input frame
          ↪ produces 2 interpolated frames, all of which need to be written to
          ↪  DRAM
- Total :
    - 216 * `r` * `c` bytes / s


### DRAM Reading Bandwidth:

- For each incoming frame (1/24 s):
    - Previous frame is read from DRAM
        - 3 * `r` * `c` bytes
- For each outgoing frame (1/60 s):
    - Entire frame is read from DRAM
        - 3 * `r` * `c` bytes
- Total :
    - 252 * `r` * `c` bytes / s

### Required Cache Size

- None

### Example Estimations:

For:
- `r` = 1080
- `c` = 1920

Results:
- DRAM write bandwidth = 447.9 MB/s
- DRAM read bandwidth = 522.5 MB/s
- Required cache size = 0 MB
```

A.2 MEMCI

A.2.1 Full Search and Unidirectional

```
# Full Search ME and Unidirectional MCI:

## Parameters:

- `b` = size of single axis of block in pixels
- `r` = number of rows of pixels in frame
- `c` = number of columns of pixels in frame
- `w` = size of single axis of search window in pixels

## Full Search:

- For each block in first image (`source block`):
    - Calculate SAD of `source block` with blocks of matching size in a window
        ↪ in the second image
        - For each block in the window in the second image (`target block`):
            - Calculate SAD of `source block` and `target block` and note
                ↪ corresponding vector
    - Take the vector corresponding to the lowest SAD and record in output with
        ↪ SAD (for occlusion in MCI) (in cases where there are multiple lowest
        ↪ SADs, precedence should be given to the smallest vector)
- Return block-wise motion vector field

## Unidirectional Interpolation:

- Create interpolated frame
- Create `r`*`c` SAD table
- Create `r`*`c` hole table
- For each block in the source frame:
    - Find new coordinates of block in interpolated frame (by following vector)
    - Any pixels that are already written, compare SAD with table value and
        ↪ overwrite if lower
    - Fill all other pixels
    - Record them as being filled in the hole table
    - Record SAD in SAD table
- For each pixel in interpolated frame:
    - If hole table says pixel is a hole:
        - Apply median filter over hole

## Full Integrated System:

To save on resources, the Y channel of YCbCr images are used for the motion
    ↪ estimation, as it has only a small effect on performance.

When describing cache, sizes are given as (rows, columns).
```

```
The following stage is designed to store the incoming frame in a block wise
    ↪ format, as apposed to raster order.

- Create (2 * `b`, `c`) * 3 bytes cache
- As the first frame streams in, place the pixels into the cache
- When `b` rows have been streamed in, start writing the cache to DRAM in a
    ↪ block wise fashion
    - Blocks are flattened and then written in contiguously
    - Left to right
    - Stream frame into the other `b` rows while writing
    - Repeat until full image in DRAM and stored block wise

After first frame, the following stages happen in a loop with each incoming
    ↪ frame:

- Create (`w` + `b`, `c`) * 1 byte cache (called `win_cache`)
    - This cache will be used to search the target frame for the motion vectors
    - The height is `w`+`b` so that the first `w` rows can be processed while
        ↪ the next `b` rows is streamed in
- Create (`w` + 5, `c`) * 3 bytes cache (called `write_cache`)
    - This cache is to store the interpolated frame as it's being written to
        ↪ save bandwidth
    - As 2 frames are interpolated between each input frame, 2 of these caches
        ↪ are required
    - Set top 5 rows to 0
    - Start addressing after top 5 rows (top 5 rows are only used for hole
        ↪ filling)
- Create (`w`, `c`) * (1 bit + 15 bits) (called `w_h_s_cache`)
    - Used to store metadata regarding the interpolated frame
    - As there are 2 interpolated frames, 2 of this cache are needed
    - First bit - low if hole, high if filled
    - Following 15 bits - SAD score
    - Set all values to 0
- Create (`b`, n * 64) * 1 byte cache (called `block_cache`)
    - Used to store the source block from the previous frame
    - Width = n * 64, because the read from DRAM is 64 contiguous bytes (512
        ↪ bits), so if `b`^2 < 512, then n=1, but if it is larger, then a larger
        ↪  n is required to store all of the data
- Stream frame into DRAM following procedure outlined for first frame
- Also stream greyscale values from frame into `win_cache` in parallel
- When `w` rows have been read in to `win_cache`:
    - Continue streaming greyscale pixels into other `b` rows
    - Begin pulling out the blocks from the previous frame from DRAM (that
        ↪ correspond to the row of windows that have just been stored in cache)
        ↪ and convert to greyscale
        - Cache block in `block_cache`
    - Perform full search (detailed above) for given block in `block_cache` and
        ↪ search window from `win_cache`
        - Output should be single motion vector with corresponding SAD score
```

```
    - Calculate new position of block in ‘write_cache‘ (relative to block row)
      ↪ by following motion vector
      - As there are multiple interpolated frames, this is done for both (also
        ↪ true for all following steps)
    - For each pixel in the block, if the corresponding location in ‘w_h_s_cache‘
      ↪  says there is a hole, fill pixel in ‘write_cache‘ with pixel from
      ↪ block
    - For each pixel in the block, if the corresponding location in ‘w_h_s_cache‘
      ↪  says there is no hole, compare SAD in ‘w_h_s_cache‘ with found SAD
      ↪ and keep the lowest
    - Fill ‘w_h_s_cache‘ with a 1 for the hole and then appropriate SAD score
    - After each block in the row in the source frame has been processed:
      - Ignoring the top ‘b‘ rows of ‘write_cache‘, search the following ‘b‘
        ↪ rows for pixels that have a hole value of 0 in ‘w_h_s_cache‘ in
        ↪ the corresponding location
      - Apply median filter to any such pixel in ‘write_cache‘
      - Ignoring the top 5 rows of ‘write_cache‘, write the following ‘b‘ rows
        ↪ into DRAM (either in block wise format for consistency when
        ↪ streaming out, or in raster order)
      - Shift the addressing of the ‘write_cache‘ so that the start is ‘b‘ rows
        ↪  later
      - Set the first ‘b‘ rows to 0 in ‘w_h_s_cache‘ (the values corresponding
        ↪ to the ‘b‘ rows of ‘write_cache‘ that have just been written to
        ↪ DRAM) and then shift the addressing so that the start is ‘b‘ rows
        ↪ later
    - Repeat for the next ‘w‘ rows in ‘win_cache‘
- Repeat until entire frame has been processed

## Hardware Estimation:

### DRAM Writing Bandwidth:

- For each incoming frame (1/24 s):
    - Entire frame is written to DRAM:
      - 3 * ‘r‘ * ‘c‘ bytes
    - 2 interpolated frames are written to DRAM:
      - 6 * ‘r‘ * ‘c‘ bytes
    - Total:
      - 9 * ‘r‘ * ‘c‘ bytes
- Total:
    - 216 * ‘r‘ * ‘c‘ bytes / s

### DRAM Reading Bandwidth:

- For each incoming frame (1/24 s):
    - Entire previous frame is pulled out block by block (wasted data minimised)
      - 3 * ‘r‘ * ‘c‘ bytes
- Every 1/60 s:
    - Stream out the frame for this time stamp:
```

```
        - 3 * `r` * `c` bytes
- Total:
    - 252 * `r` * `c` bytes / s

### Required Cache Size:

- To store frame block wise:
    - 6 * `b`* `c` bytes
- `win_cache`:
    - (`w`+`b`) * `c` bytes
- `write_cache` * 2:
    - 6 * (`w` + 5) * `c` bytes
- `w_h_s_cache` * 2:
    - 4 * `w` * `c`
- Total:
    - `c` * (7 * `b` + 11 * `w` + 30) bytes

### Example Estimations:

For:
- `b` = 8
- `r` = 1080
- `c` = 1920
- `w` = 22

Results:
- DRAM write bandwidth = 427.1 MB/s
- DRAM read bandwidth = 498.3 MB/s
- Required cache size = 0.601 MB
```

A.2.2 Three Step Search and Unidirectional

```
# Three Step Search ME and Unidirectional MCI:

## Parameters:

- `b` = size of single axis of block in pixels
- `r` = number of rows of pixels in frame
- `c` = number of columns of pixels in frame
- `steps` = number of steps taken when searching

## Some required values:

- `w` = `b` + (2 * Sum (`i` = 0 -> `steps`-1) (2^`i`))
    - This is the size of a single axis of the search window

## Three Step Search:

- For each block in first image (`source block`):
```

```
    - `center` = `source block`
    - For `step` in (`steps`-1 -> 1) (default value of `steps` is 3):
        - `space` = 2^`step`
        - For each block in 3x3 around `center`, `space` pixels apart (`target
            ↪ block`):
            - Calculate SAD between `source block` and `target block` and note
                ↪ corresponding vector (no need for `center` block as it has been
                ↪  previously calculated)
        - `center` = `target block` with lowest SAD
    - `space` = 1
    - For each block in 3x3 around `center`, `space` pixels apart (`target block
        ↪ `):
        - Calculate SAD between `source block` and `target block` and note
            ↪ corresponding vector (no need for `center` block as it has been
            ↪ previously calculated)
    - Assign vector with lowest corresponding SAD to `source block`
- Return block-wise motion vector field

## Unidirectional Interpolation:

- Create interpolated frame
- Create `r`*`c` SAD table
- Create `r`*`c` hole table
- For each block in the source frame:
    - Find new coordinates of block in interpolated frame (by following vector)
    - Any pixels that are already written, compare SAD with table value and
        ↪ overwrite if lower
    - Fill all other pixels
    - Record them as being filled in the hole table
    - Record SAD in SAD table
- For each pixel in interpolated frame:
    - If hole table says pixel is a hole:
        - Apply median filter over hole

## Full Integrated System:

To save on resources, the Y channel of YCbCr images are used for the motion
    ↪ estimation, as it has only a small effect on performance.

When describing cache, sizes are given as (rows, columns).

The following stage is designed to store the incoming frame in a block wise
    ↪ format, as apposed to raster order.

- Create (2 * `b`, `c`) * 3 bytes cache
- As the first frame streams in, place the pixels into the cache
- When `b` rows have been streamed in, start writing the cache to DRAM in a
    ↪ block wise fashion
    - Blocks are flattened and then written in contiguously
```

```
    - Left to right
    - Stream frame into the other `b` rows while writing
    - Repeat until full image in DRAM and stored block wise

After first frame, the following stages happen in a loop with each incoming
    ↪ frame:

- Create (`w` + `b`, `c`) * 1 byte cache (called `win_cache`)
    - This cache will be used to search the target frame for the motion vectors
    - The height is `w`+`b` so that the first `w` rows can be processed while
        ↪ the next `b` rows is streamed in
- Create (`w` + 5, `c`) * 3 bytes cache (called `write_cache`)
    - This cache is to store the interpolated frame as it's being written to
        ↪ save bandwidth
    - As 2 frames are interpolated between each input frame, 2 of these caches
        ↪ are required
    - Set top 5 rows to 0
    - Start addressing after top 5 rows (top 5 rows are only used for hole
        ↪ filling)
- Create (`w`, `c`) * (1 bit + 15 bits) (called `w_h_s_cache`)
    - Used to store metadata regarding the interpolated frame
    - As there are 2 interpolated frames, 2 of this cache are needed
    - First bit - low if hole, high if filled
    - Following 15 bits - SAD score
    - Set all values to 0
- Create (`b`, n * 64) * 1 byte cache (called `block_cache`)
    - Used to store the source block from the previous frame
    - Width = n * 64, because the read from DRAM is 64 contiguous bytes (512
        ↪ bits), so if `b`^2 < 512, then n=1, but if it is larger, then a larger
        ↪  n is required to store all of the data
- Stream frame into DRAM following procedure outlined for first frame
- Also stream greyscale values from frame into `win_cache` in parallel
- When `w` rows have been read in to `win_cache`:
    - Continue streaming greyscale pixels into other `b` rows
    - Begin pulling out the blocks from the previous frame from DRAM (that
        ↪ correspond to the row of windows that have just been stored in cache)
        ↪ and convert to greyscale
        - Cache block in `block_cache`
    - Perform three step search (detailed above) for given block in `block_cache`
        ↪  and search window from `win_cache`
        - Output should be single motion vector with corresponding SAD score
    - Calculate new position of block in `write_cache` (relative to block row)
        ↪ by following motion vector
        - As there are multiple interpolated frames, this is done for both (also
            ↪ true for all following steps)
    - For each pixel in the block, if the corresponding location in `w_h_s_cache`
        ↪  says there is a hole, fill pixel in `write_cache` with pixel from
        ↪ block
```

```
    - For each pixel in the block, if the corresponding location in `w_h_s_cache`
        ↪  says there is no hole, compare SAD in `w_h_s_cache` with found SAD
        ↪ and keep the lowest
    - Fill `w_h_s_cache` with a 1 for the hole and then appropriate SAD score
    - After each block in the row in the source frame has been processed:
        - Ignoring the top `b` rows of `write_cache`, search the following `b`
            ↪ rows for pixels that have a hole value of 0 in `w_h_s_cache` in
            ↪ the corresponding location
        - Apply median filter to any such pixel in `write_cache`
        - Ignoring the top 5 rows of `write_cache`, write the following `b` rows
            ↪ into DRAM (either in block wise format for consistency when
            ↪ streaming out, or in raster order)
        - Shift the addressing of the `write_cache` so that the start is `b` rows
            ↪  later
        - Set the first `b` rows to 0 in `w_h_s_cache` (the values corresponding
            ↪ to the `b` rows of `write_cache` that have just been written to
            ↪ DRAM) and then shift the addressing so that the start is `b` rows
            ↪ later
    - Repeat for the next `w` rows in `win_cache`
- Repeat until entire frame has been processed

## Hardware Estimation:

### DRAM Writing Bandwidth:

- For each incoming frame (1/24 s):
    - Entire frame is written to DRAM:
        - 3 * `r` * `c` bytes
    - 2 interpolated frames are written to DRAM:
        - 6 * `r` * `c` bytes
    - Total:
        - 9 * `r` * `c` bytes
- Total:
    - 216 * `r` * `c` bytes / s

### DRAM Reading Bandwidth:

- For each incoming frame (1/24 s):
    - Entire previous frame is pulled out block by block (wasted data minimised)
        - 3 * `r` * `c` bytes
- Every 1/60 s:
    - Stream out the frame for this time stamp:
        - 3 * `r` * `c` bytes
- Total:
    - 252 * `r` * `c` bytes / s

### Required Cache Size:

- To store frame block wise:
```

```
    - 6 * `b`* `c` bytes
- `win_cache`:
    - (`w`+`b`) * `c` bytes
- `write_cache` * 2:
    - 6 * (`w` + 5) * `c` bytes
- `w_h_s_cache` * 2:
    - 4 * `w` * `c`
- Total:
    - `c` * (7 * `b` + 11 * `w` + 30) bytes

### Example Estimations:

For:
- `b` = 8
- `r` = 1080
- `c` = 1920
- `steps` = 3
- `w` = `b` + (2 * Sum (`i` = 0 -> `steps`-1) (2^`i`)) = 22

Results:
- DRAM write bandwidth = 427.1 MB/s
- DRAM read bandwidth = 498.3 MB/s
- Required cache size = 0.601 MB
```

A.2.3 Hierarchical Block Matching Algorithm and Unidirectional

```
# HBMA ME and Unidirectional MCI:

## Parameters:

- `b_max` = max size of single axis of block in pixels
- `b_min` = min size of single axis of block in pixels
- `r` = number of rows of pixels in frame
- `c` = number of columns of pixels in frame
- `w` = size of single axis of search window in pixels
- `s` = number of times the frames are downscaled

## HBMA:

- Block size must be power of 2
- Reduce first and second images `steps` times using linear filter, storing each
  ↪ image
    - Weightings for linear filter:
      | | | |
      |------|-----|------|
      |0.0625|0.125|0.0625|
      |0.125 |0.25 |0.125 |
      |0.0625|0.125|0.0625|
    - Convolve filter with images with stride of (2,2) to downscale
```

```
- Perform full search algorithm on smallest first and second frames
- For next smallest images -> original images:
    - Increase motion vector density
        - Because the image size is halved when downscaled, with a consistent
          ↪ block size, a ("parent") block in the downscaled image is 4 ("
          ↪ child") blocks in the original image
        - For each of the 4 child blocks for every block in the previous
          ↪ iteration:
            - Search areas in a (3x3) window around position pointed to by parent
              ↪  vector, and the 2 vectors associated with 2 blocks adjacent to
              ↪  the parent block (parent vectors need to be multiplied by 2 to
              ↪  account for change in image size)
            - Select vector corresponding to the smallest SAD
- For `b_max` -> `b_min` by halving:
    - Increase motion vector density
        - By halving the block size, each previous ("parent") block holds 4 ("
          ↪ child") blocks
        - For each of the 4 child blocks for every block in the previous
          ↪ iteration:
            - Search areas in a (3x3) window around position pointed to by parent
              ↪  vector, and the 2 vectors associated with 2 blocks adjacent to
              ↪  the parent block
            - Select vector corresponding to the smallest SAD
- Return block-wise motion vector field

## Unidirectional Interpolation:

- Create interpolated frame
- Create `r`*`c` SAD table
- Create `r`*`c` hole table
- For each block in the source frame:
    - Find new coordinates of block in interpolated frame (by following vector)
    - Any pixels that are already written, compare SAD with table value and
      ↪ overwrite if lower
    - Fill all other pixels
    - Record them as being filled in the hole table
    - Record SAD in SAD table
- For each pixel in interpolated frame:
    - If hole table says pixel is a hole:
        - Apply median filter over hole

## Full Integrated System:

To save on resources, the Y channel of YCbCr images are used for the motion
    ↪ estimation, as it has only a small effect on performance.

When describing cache, sizes are given as (rows, columns).

Some required formulas:
```

```
- `a(i)` = 2^(`s` - `i`) * `w` + Sum (`n` = 1 -> `s` - `i`) (2^`n`)
- `pad` = Sum (`n` = 1 -> log_2 (`b_max`/`b_min`)) (2^`n`)

For simplicity of explanation, `s`=1 has been used in the algorithm:

The following stage is designed to store the incoming frame in a block wise
    ↪ format, as apposed to raster order.

- Create (2 * `b_min`, `c`) * 3 bytes cache
    - For original frame so it can be written block wise
- Create (2 * `b_max`, `c`/2) * 1 byte cache
    - For downscaled frame so that it can be written block wise
- As the first frame streams in, place the pixels into the first cache
- When `b_max` rows have been streamed in, start writing the cache to DRAM in a
    ↪ block wise fashion
    - Blocks are flattened and then written in contiguously
    - Left to right
    - Channels stored separately
    - Stream frame into the other `b_max` rows while writing
    - Repeat until full image in DRAM and stored block wise
- Concurrently, when 3 rows have been streamed in, begin applying downscale
    ↪ filter to image and store in second cache
    - Once `b_max` rows of downscaled image have been created, save to DRAM in
        ↪ block wise format (as above)

After first frame, the following stages happen in a loop with each incoming
    ↪ frame (`target_frame`):

- Stream `target_frame` into DRAM following procedure outlined for first frame
- Create (`a(0)+pad`, `c`) * 1 byte cache (called `win_cache_0`)
    - This cache will be used to search the largest scale target frame for the
        ↪ motion vectors
    - Number of rows is `a(0)+pad`, as `a(0)` is the range of motion a block can
        ↪  take, and the `pad` is so that the cache can be reused for future
        ↪ density increases
    - 1 byte is needed because it is only the Y channel
- Create (`a(1)`, `c`/2) * 1 byte (called `win_cache_1`)
    - This cache will be used to search the downscaled target frame for the
        ↪ motion vectors
    - 1 byte needed as it is only the Y channel
- Create (`a(0)+pad+5`, `c`) * 3 bytes cache (called `write_cache`)
    - This cache is to store the interpolated frame as it's being written to
        ↪ save bandwidth
    - As 2 frames are interpolated between each input frame, 2 of these caches
        ↪ are required.
    - Set top 5 rows to 0
    - Start addressing after top 5 rows (top 5 rows are only used for hole
        ↪ filling)
```

```
- Create (‘a(0)+pad‘, ‘c‘) * (1 bit + 15 bits) (called ‘w_h_s_cache‘)
    - Used to store metadata regarding the interpolated frame
    - As there are 2 interpolated frames, 2 of this cache are needed
    - First bit - low if hole, high if filled
    - Following 15 bits - SAD score
    - Set all values to 0
- Create (2, ‘c‘/‘b_max‘) * 2 bytes cache (called ‘vec_cache_0‘)
    - Used when increasing vector density
    - Records previous row of motion vectors
    - 1 byte for x
    - 1 byte for y
    - Initialised to 0
- Create (2, ‘c‘/(2*‘b_max‘)) * 2 bytes cache (called ‘vec_cache_1‘)
    - Similar to above but for downscaled image
- Read into ‘win_cache_0‘ from the downscaled target frame in DRAM
- When ‘a(1)‘ rows have been read in:
    - Read blocks sequentially from downscaled source frame
    - For each block:
        - Calculate motion vector by performing a full search in ‘win_cache_0‘
        - Store motion vector in ‘vec_cache_0‘ for reference by its children and
            ↪ adjacent children
        - Begin reading into ‘win_cache_1‘ from level above target frame (in this
            ↪  case, the original image) (Y channel)
        - When ‘a(0)+pad‘ rows have been read in:
            - Read 4 children blocks from layer above source frame (in this case,
                ↪  the original image)
            - For each child block:
                - Perform search in 27 locations (as described in HBMA section)
                - Store motion vector in ‘vec_cache_1‘ for reference by its
                    ↪ children and adjacent children
                - Break block into 4 child blocks
                    - If not currently using original image, then children come
                        ↪ from the layer above
                    - If on the original image, halve the block size to get child
                        ↪ blocks
                    - For each child:
                        - Perform 27 searches as stated before
                        - As its the original image, follow motion vector and
                            ↪ write child block into ‘write_cache‘ and update ‘
                            ↪ w_h_s_cache‘ with corresponding metadata
                        - When ‘write_cache‘ is full, apply hole filling filter to
                            ↪  top ‘b_min‘ rows below the top 5 rows (they are
                            ↪ only to provide hole filling info) and then write
                            ↪ those rows to DRAM and shift the starting address by
                            ↪  ‘b_min‘ rows. set corresponding rows in ‘
                            ↪ w_h_s_cache‘ to 0 and shift the start address by the
                            ↪  same amount

## Hardware Estimation:
```

```
### DRAM Writing Bandwidth:

- For each incoming frame (1/24 s):
    - Frame written to DRAM:
        - 3 * ‘r‘ * ‘c‘ bytes
    - Downscaled frames written to DRAM:
        - Sum (‘i‘ = 1 -> ‘s‘) (‘r‘ * ‘c‘ / 2^‘i‘)
    - 2 interpolated frames written to DRAM:
        - 6 * ‘r‘ * ‘c‘ bytes
    - Total:
        - 9 * ‘r‘ * ‘c‘ + Sum (‘i‘ = 1 -> ‘s‘) (‘r‘ * ‘c‘ / 2^‘i‘) bytes
- Total:
    - (9 * ‘r‘ * ‘c‘ + Sum (‘i‘ = 1 -> ‘s‘) (‘r‘ * ‘c‘ / 2^‘i‘)) * 24 bytes / s

### DRAM Reading Bandwidth:

- For each incoming frame (1/24 s):
    - Every image saved in DRAM is read out (only Y channel):
        - 2 * Sum (‘i‘ = 0 -> ‘s‘) (‘r‘ * ‘c‘ / 2^‘i‘) bytes
    - 2 colour channels from original source frame:
        - 2 * ‘r‘ * ‘c‘ bytes
    - Total:
        - 2 * ‘r‘ * ‘c‘ + 2 * Sum (‘i‘ = 0 -> ‘s‘) (‘r‘ * ‘c‘ / 2^‘i‘) bytes
- Every 1/60 s:
    - Stream out a frame:
        - 3 * ‘r‘ * ‘c‘ bytes
- Total:
    228 * ‘r‘ * ‘c‘ + 48 * Sum (‘i‘ = 0 -> ‘s‘) (‘r‘ * ‘c‘ / 2^‘i‘) bytes / s

### Required Cache Size:

- To store frames:
    - 6 * ‘b_min‘ * ‘c‘ bytes
- To downscale and store:
    - Sum (‘i‘ = 1 -> ‘s‘) (2 * ‘b_max‘ * ‘c‘/2^‘i‘) bytes
- ‘win_cache‘:
    - Sum (‘i‘ = 0 -> ‘s‘) (‘a(i)‘ * ‘c‘/2^‘i‘) + ‘pad‘ * ‘c‘ bytes
- ‘write_cache‘ * 2:
    - 6 * (‘a(0)+pad+5‘) * ‘c‘ bytes
- ‘w_h_s_cache‘ * 2:
    - 4 * (‘a(0)+pad‘) * ‘c‘ bytes
- ‘vec_cache‘:
    - Sum (‘i‘ = 0 -> ‘s‘) (4 * ‘c‘/(2^‘i‘ * ‘b_max‘))

### Example estimation:

For:
- ‘b_max‘ = 8
```

```
- ‘b_min‘ = 4
- ‘r‘ = 1080
- ‘c‘ = 1920
- ‘w‘ = 22
- ‘s‘ = 2

Results:
- DRAM write bandwidth = 462.7 MB/s
- DRAM read bandwidth = 617.0 MB/s
- Required cache size = 2.12 MB
```

A.2.4 Full Search and Bidirectional

```
# Full Search ME and Bidirectional MCI:

## Parameters:

- ‘b‘ = size of single axis of block in pixels
- ‘r‘ = number of rows of pixels in frame
- ‘c‘ = number of columns of pixels in frame
- ‘w‘ = size of single axis of search window in pixels

## Full Search:

- For each block in first image (‘source block‘):
    - Calculate SAD of ‘source block‘ with blocks of matching size in a window
      ↪ in the second image
      - For each block in the window in the second image (‘target block‘):
          - Calculate SAD of ‘source block‘ and ‘target block‘ and note
            ↪ corresponding vector
    - Take the vector corresponding to the lowest SAD and record in output with
      ↪ SAD (for occlusion in MCI) (in cases where there are multiple lowest
      ↪ SADs, precedence should be given to the smallest vector)
- Return block-wise motion vector field

## Bidirectional Interpolation:

- Calculate forward and backward motion vectors using previous algorithm
    - Swap frames to get backward motion vectors
- Create interpolated frame
- Create ‘r‘ * ‘c‘ SAD table
- Create ‘r‘ * ‘c‘ hole table
- For each block in the source frame:
    - Find new coordinates of block in interpolated frame (by following forward
      ↪ vector)
    - Any pixels that are already written, compare SAD with table value and
      ↪ overwrite if lower
    - Fill all other pixels
    - Record them as being filled in the hole table
```

```
    - Record SAD in SAD table
- For each block in the target frame:
    - Find new coordinates of block in interpolated frame (by following backward vector)
    - Any pixels that are already written, compare SAD with table value and overwrite if lower
    - Fill all other pixels
    - Record them as being filled in the hole table
    - Record SAD in SAD table
- For each pixel in interpolated frame:
    - If hole table says pixel is a hole:
        - Apply median filter over hole

## Full Integrated System:

To save on resources, the Y channel of YCbCr images are used for the motion estimation, as it has only a small effect on performance.

When describing cache, sizes are given as (rows, columns).

Images are assumed to be YCbCr. If not, a conversion is applied as it is streamed in.

The following stage is designed to store the incoming frame in a block wise format, as apposed to raster order.

- Create (2 * `b`, `c`) * 3 bytes cache
- As the first frame streams in, place the pixels into the cache
- When `b` rows have been streamed in, start writing the cache to DRAM in a block wise fashion
    - All 3 channels are written to DRAM as separate 1 channel images
    - Blocks are flattened and then written in contiguously
    - Left to right
    - Stream frame into the other `b` rows while writing
    - Repeat until full image in DRAM and stored block wise

After first frame, the following stages happen in a loop with each incoming frame (target frame):

- Create (`w`, `c`) * 1 byte cache (called `source_win_cache`)
    - This cache will be used to search the source frame for the motion vectors
- Create (`w` + `b`, `c`) * 1 byte cache (called `target_win_cache`)
    - This cache will be used to search the target frame for the motion vectors
    - The height is `w`+`b` so that the first row of windows can be searched while the values needed for the next row are streamed in
- Create (`w` + 5, `c`) * 3 bytes cache (called `write_cache`)
    - This cache is to store the interpolated frame as it's being written to save bandwidth
```

```
    - As 2 frames are interpolated between each input frame, 2 of these caches
        ↪ are required
    - Set top 5 rows to 0
    - Start addressing after top 5 rows (top 5 rows are only used for hole
        ↪ filling)
- Create (‘w‘, ‘c‘) * (1 bit + 15 bits) (called ‘w_h_s_cache‘)
    - Used to store metadata regarding the interpolated frame
    - As there are 2 interpolated frames, 2 of this cache are needed
    - First bit - low if hole, high if filled
    - Following 15 bits - SAD score
    - Set all values to 0
- Stream frame into DRAM following procedure outlined for first frame
- Also stream greyscale values from frame into ‘target_win_cache‘ in parallel
- Also pull out greyscale values from the source frame into ‘source_win_cache‘
- When ‘w‘ rows have been read in to ‘target_win_cache‘ and ‘source_win_cache‘:
    - Continue streaming greyscale pixels into other ‘b‘ rows of ‘
        ↪ target_win_cache‘ from target frame
    - Taking a block from the ‘source_win_cache‘:
        - Search the corresponding search window in ‘target_win_cache‘
            - Returns the best motion vector and SAD score
        - Pull the other 2 channels for the block out of DRAM
        - Write the block to ‘write_cache‘ by following the motion vector
            - If the ‘w_h_s_cache‘ says the says there is a hole, write to the
                ↪ pixel
            - If there is no hole, only write if the SAD is lower for block being
                ↪  written
        - Update ‘w_h_s_cache‘ for every pixel written with the SAD score and
            ↪ setting the hole bit to 1
    - Do the same as the previous step in parallel but swap ‘source_win_cache‘
        ↪ and ‘target_win_cache‘
    - Repeat the last 2 stages for the whole row of blocks in both ‘
        ↪ source_win_cache‘ and ‘target_win_cache‘
    - Ignoring the top 5 rows in ‘write_cache‘, apply the median filter to any
        ↪ pixel that is a hole in the following ‘b‘ rows
    - Write the same ‘b‘ rows to DRAM (for the interpolated frame) and set all
        ↪ values to 0
    - Set the corresponding values in ‘w_h_s_cache‘ to 0
    - Shift the the start address of both the ‘write_cache‘ and ‘w_h_s_cache‘ by
        ↪  ‘b‘ rows
    - Repeat until whole image is read in

## Hardware Estimation:

### DRAM Writing Bandwidth:

- For each incoming frame (1/24 s):
    - Entire frame is written to DRAM:
        - 3 * ‘r‘ * ‘c‘ bytes
    - 2 interpolated frames are written to DRAM:
```

```
        - 6 * ‘r‘ * ‘c‘ bytes
    - Total:
        - 9 * ‘r‘ * ‘c‘ bytes
- Total:
    - 216 * ‘r‘ * ‘c‘ bytes / s

### DRAM Reading Bandwidth:

- For each incoming frame (1/24 s):
    - Entire previous frame is pulled out block by block (wasted data minimised)
        - 3 * ‘r‘ * ‘c‘ bytes
    - 2 channels read from the incoming frame
        - 2 * ‘r‘ * ‘c‘ bytes
- Every 1/60 s:
    - Stream out the frame for this time stamp:
        - 3 * ‘r‘ * ‘c‘ bytes
- Total:
    - 300 * ‘r‘ * ‘c‘ bytes / s

### Required Cache Size:

- To store frame block wise:
    - 6 * ‘b‘* ‘c‘ bytes
- ‘source_win_cache‘:
    - (‘w‘ + ‘b‘) * ‘c‘ bytes
- ‘target_win_cache‘:
    - (‘w‘ + ‘b‘) * ‘c‘ bytes
- ‘write_cache‘ * 2:
    - 6 * (‘w‘ + 5) * ‘c‘ bytes
- ‘w_h_s_cache‘ * 2:
    - 4 * ‘w‘ * ‘c‘
- Total:
    - 2 * ‘c‘ * (4 * ‘b‘ + 3 * (2 * ‘w‘ + 5)) bytes

### Example Estimations:

For:
- ‘b‘ = 8
- ‘r‘ = 1080
- ‘c‘ = 1920
- ‘w‘ = 22

Results:
- DRAM write bandwidth = 427.1 MB/s
- DRAM read bandwidth = 593.3 MB/s
- Required cache size = 0.66 MB
```

A.2.5 Three Step Search and Bidirectional

```
# Full Search ME and Bidirectional MCI:

## Parameters:

- `b` = size of single axis of block in pixels
- `r` = number of rows of pixels in frame
- `c` = number of columns of pixels in frame
- `steps` = number of steps taken when searching

## Some required values:

- `w` = `b` + (2 * Sum (`i` = 0 -> `steps`-1) (2^`i`))
    - This is the size of a single axis of the search window

## Three Step Search:

- For each block in first image (`source block`):
    - `center` = `source block`
    - For `step` in (`steps`-1 -> 1) (default value of `steps` is 3):
        - `space` = 2^`step`
        - For each block in 3x3 around `center`, `space` pixels apart (`target
            ↪ block`):
            - Calculate SAD between `source block` and `target block` and note
                ↪ corresponding vector (no need for `center` block as it has been
                ↪  previously calculated)
        - `center` = `target block` with lowest SAD
    - `space` = 1
    - For each block in 3x3 around `center`, `space` pixels apart (`target block
        ↪ `):
        - Calculate SAD between `source block` and `target block` and note
            ↪ corresponding vector (no need for `center` block as it has been
            ↪ previously calculated)
    - Assign vector with lowest corresponding SAD to `source block`
- Return block-wise motion vector field

## Bidirectional Interpolation:

- Calculate forward and backward motion vectors using previous algorithm
    - Swap frames to get backward motion vectors
- Create interpolated frame
- Create `r` * `c` SAD table
- Create `r` * `c` hole table
- For each block in the source frame:
    - Find new coordinates of block in interpolated frame (by following forward
        ↪ vector)
    - Any pixels that are already written, compare SAD with table value and
        ↪ overwrite if lower
    - Fill all other pixels
    - Record them as being filled in the hole table
```

```
    - Record SAD in SAD table
- For each block in the target frame:
    - Find new coordinates of block in interpolated frame (by following backward
      ↪  vector)
    - Any pixels that are already written, compare SAD with table value and
      ↪ overwrite if lower
    - Fill all other pixels
    - Record them as being filled in the hole table
    - Record SAD in SAD table
- For each pixel in interpolated frame:
    - If hole table says pixel is a hole:
        - Apply median filter over hole

## Full Integrated System:

To save on resources, the Y channel of YCbCr images are used for the motion
    ↪ estimation, as it has only a small effect on performance.

When describing cache, sizes are given as (rows, columns).

Images are assumed to be YCbCr. If not, a conversion is applied as it is
    ↪ streamed in.

The following stage is designed to store the incoming frame in a block wise
    ↪ format, as apposed to raster order.

- Create (2 * `b`, `c`) * 3 bytes cache
- As the first frame streams in, place the pixels into the cache
- When `b` rows have been streamed in, start writing the cache to DRAM in a
    ↪ block wise fashion
    - All 3 channels are written to DRAM as separate 1 channel images
    - Blocks are flattened and then written in contiguously
    - Left to right
    - Stream frame into the other `b` rows while writing
    - Repeat until full image in DRAM and stored block wise

After first frame, the following stages happen in a loop with each incoming
    ↪ frame (target frame):

- Create (`w`, `c`) * 1 byte cache (called `source_win_cache`)
    - This cache will be used to search the source frame for the motion vectors
- Create (`w` + `b`, `c`) * 1 byte cache (called `target_win_cache`)
    - This cache will be used to search the target frame for the motion vectors
    - The height is `w`+`b` so that the first row of windows can be searched
      ↪ while the values needed for the next row are streamed in
- Create (`w` + 5, `c`) * 3 bytes cache (called `write_cache`)
    - This cache is to store the interpolated frame as it's being written to
      ↪ save bandwidth
```

```
    - As 2 frames are interpolated between each input frame, 2 of these caches
      ↪ are required
    - Set top 5 rows to 0
    - Start addressing after top 5 rows (top 5 rows are only used for hole
      ↪ filling)
- Create ('w', 'c') * (1 bit + 15 bits) (called 'w_h_s_cache')
    - Used to store metadata regarding the interpolated frame
    - As there are 2 interpolated frames, 2 of this cache are needed
    - First bit - low if hole, high if filled
    - Following 15 bits - SAD score
    - Set all values to 0
- Stream frame into DRAM following procedure outlined for first frame
- Also stream greyscale values from frame into 'target_win_cache' in parallel
- Also pull out greyscale values from the source frame into 'source_win_cache'
- When 'w' rows have been read in to 'target_win_cache' and 'source_win_cache':
    - Continue streaming greyscale pixels into other 'b' rows of '
      ↪ target_win_cache' from target frame
    - Taking a block from the 'source_win_cache':
        - Search the corresponding search window in 'target_win_cache' following
          ↪ algorithm laid out above for three step search
            - Returns the best motion vector and SAD score
        - Pull the other 2 channels for the block out of DRAM
        - Write the block to 'write_cache' by following the motion vector
            - If the 'w_h_s_cache' says the says there is a hole, write to the
              ↪ pixel
            - If there is no hole, only write if the SAD is lower for block being
              ↪  written
        - Update 'w_h_s_cache' for every pixel written with the SAD score and
          ↪ setting the hole bit to 1
    - Do the same as the previous step in parallel but swap 'source_win_cache'
      ↪ and 'target_win_cache'
    - Repeat the last 2 stages for the whole row of blocks in both '
      ↪ source_win_cache' and 'target_win_cache'
    - Ignoring the top 5 rows in 'write_cache', apply the median filter to any
      ↪ pixel that is a hole in the following 'b' rows
    - Write the same 'b' rows to DRAM (for the interpolated frame) and set all
      ↪ values to 0
    - Set the corresponding values in 'w_h_s_cache' to 0
    - Shift the the start address of both the 'write_cache' and 'w_h_s_cache' by
      ↪  'b' rows
    - Repeat until whole image is read in

## Hardware Estimation:

### DRAM Writing Bandwidth:

- For each incoming frame (1/24 s):
    - Entire frame is written to DRAM:
        - 3 * 'r' * 'c' bytes
```

```
    - 2 interpolated frames are written to DRAM:
        - 6 * ‘r‘ * ‘c‘ bytes
    - Total:
        - 9 * ‘r‘ * ‘c‘ bytes
- Total:
    - 216 * ‘r‘ * ‘c‘ bytes / s

### DRAM Reading Bandwidth:

- For each incoming frame (1/24 s):
    - Entire previous frame is pulled out block by block (wasted data minimised)
        - 3 * ‘r‘ * ‘c‘ bytes
    - 2 channels read from the incoming frame
        - 2 * ‘r‘ * ‘c‘ bytes
- Every 1/60 s:
    - Stream out the frame for this time stamp:
        - 3 * ‘r‘ * ‘c‘ bytes
- Total:
    - 300 * ‘r‘ * ‘c‘ bytes / s

### Required Cache Size:

- To store frame block wise:
    - 6 * ‘b‘* ‘c‘ bytes
- ‘source_win_cache‘:
    - (‘w‘ + ‘b‘) * ‘c‘ bytes
- ‘target_win_cache‘:
    - (‘w‘ + ‘b‘) * ‘c‘ bytes
- ‘write_cache‘ * 2:
    - 6 * (‘w‘ + 5) * ‘c‘ bytes
- ‘w_h_s_cache‘ * 2:
    - 4 * ‘w‘ * ‘c‘
- Total:
    - 2 * ‘c‘ * (4 * ‘b‘ + 3 * (2 * ‘w‘ + 5)) bytes

### Example Estimations:

For:
- ‘b‘ = 8
- ‘r‘ = 1080
- ‘c‘ = 1920
- ‘w‘ = 22

Results:
- DRAM write bandwidth = 427.1 MB/s
- DRAM read bandwidth = 593.3 MB/s
- Required cache size = 0.66 MB
```

A.2.6 Hierarchical Block Matching Algorithm and Bidirectional

```
# HBMA ME and Bidirectional MCI:

## Parameters:

- `b_max` = max size of single axis of block in pixels
- `b_min` = min size of single axis of block in pixels
- `r` = number of rows of pixels in frame
- `c` = number of columns of pixels in frame
- `w` = size of single axis of search window in pixels
- `s` = number of times the frames are downscaled

## HBMA:

- Block size must be power of 2
- Reduce first and second images `steps` times using linear filter, storing each
    ↪ image
    - Weightings for linear filter:
      | | | |
      |------|-----|------|
      |0.0625|0.125|0.0625|
      |0.125 |0.25 |0.125 |
      |0.0625|0.125|0.0625|
    - Convolve filter with images with stride of (2,2) to downscale
- Perform full search algorithm on smallest first and second frames
- For next smallest images -> original images:
    - Increase motion vector density
        - Because the image size is halved when downscaled, with a consistent
          ↪ block size, a ("parent") block in the downscaled image is 4 ("
          ↪ child") blocks in the original image
        - For each of the 4 child blocks for every block in the previous
          ↪ iteration:
            - Search areas in a (3x3) window around position pointed to by parent
              ↪ vector, and the 2 vectors associated with 2 blocks adjacent to
              ↪ the parent block (parent vectors need to be multiplied by 2 to
              ↪ account for change in image size)
            - Select vector corresponding to the smallest SAD
- For `b_max` -> `b_min` by halving:
    - Increase motion vector density
        - By halving the block size, each previous ("parent") block holds 4 ("
          ↪ child") blocks
        - For each of the 4 child blocks for every block in the previous
          ↪ iteration:
            - Search areas in a (3x3) window around position pointed to by parent
              ↪ vector, and the 2 vectors associated with 2 blocks adjacent to
              ↪ the parent block
            - Select vector corresponding to the smallest SAD
- Return block-wise motion vector field
```

```
## Bidirectional Interpolation:

- Calculate forward and backward motion vectors using previous algorithm
    - Swap frames to get backward motion vectors
- Create interpolated frame
- Create `r` * `c` SAD table
- Create `r` * `c` hole table
- For each block in the source frame:
    - Find new coordinates of block in interpolated frame (by following forward
        ↪ vector)
    - Any pixels that are already written, compare SAD with table value and
        ↪ overwrite if lower
    - Fill all other pixels
    - Record them as being filled in the hole table
    - Record SAD in SAD table
- For each block in the target frame:
    - Find new coordinates of block in interpolated frame (by following backward
        ↪  vector)
    - Any pixels that are already written, compare SAD with table value and
        ↪ overwrite if lower
    - Fill all other pixels
    - Record them as being filled in the hole table
    - Record SAD in SAD table
- For each pixel in interpolated frame:
    - If hole table says pixel is a hole:
        - Apply median filter over hole

## Full Integrated System:

To save on resources, the Y channel of YCbCr images are used for the motion
    ↪ estimation, as it has only a small effect on performance.

When describing cache, sizes are given as (rows, columns).

Some required formulas:

- `a(i)` = 2^(`s` - `i`) * `w` + Sum (`n` = 1 -> `s` - `i`) (2^`n`)
- `pad` = Sum (`n` = 1 -> log_2 (`b_max`/`b_min`)) (2^`n`)

For simplicity of explanation, `s`=1 has been used in the algorithm:

The following stage is designed to store the incoming frame in a block wise
    ↪ format, as apposed to raster order.

- Create (2 * `b_min`, `c`) * 3 bytes cache
    - For original frame so it can be written block wise
- Create (2 * `b_max`, `c`/2) * 1 byte cache
    - For downscaled frame so that it can be written block wise
- As the first frame streams in, place the pixels into the first cache
```

```
- When ‘b_max‘ rows have been streamed in, start writing the cache to DRAM in a
    ↪ block wise fashion
    - Blocks are flattened and then written in contiguously
    - Left to right
    - Channels stored separately
    - Stream frame into the other ‘b_max‘ rows while writing
    - Repeat until full image in DRAM and stored block wise
- Concurrently, when 3 rows have been streamed in, begin applying downscale
    ↪ filter to image and store in second cache
    - Once ‘b_max‘ rows of downscaled image have been created, save to DRAM in
        ↪ block wise format (as above)

After first frame, the following stages happen in a loop with each incoming
    ↪ frame (‘target_frame‘):

- Stream ‘target_frame‘ into DRAM following procedure outlined for first frame
- Create (‘a(0)+pad‘, ‘c‘) * 1 byte cache (called ‘source_win_cache_0‘)
    - This cache will be used to search the largest scale source frame for the
        ↪ motion vectors
    - Number of rows is ‘a(0)+pad‘, as ‘a(0)‘ is the range of motion a block can
        ↪  take, and the ‘pad‘ is so that the cache can be reused for future
        ↪ density increases
    - 1 byte is needed because it is only the Y channel
- Create (‘a(1)‘, ‘c‘/2) * 1 byte (called ‘source_win_cache_1‘)
    - This cache will be used to search the downscaled source frame for the
        ↪ motion vectors
    - 1 byte needed as it is only the Y channel
- Create (‘a(0)+pad‘, ‘c‘) * 1 byte cache (called ‘target_win_cache_0‘)
    - This cache will be used to search the largest scale target frame for the
        ↪ motion vectors
    - Number of rows is ‘a(0)+pad‘, as ‘a(0)‘ is the range of motion a block can
        ↪  take, and the ‘pad‘ is so that the cache can be reused for future
        ↪ density increases
    - 1 byte is needed because it is only the Y channel
- Create (‘a(1)‘, ‘c‘/2) * 1 byte (called ‘target_win_cache_1‘)
    - This cache will be used to search the downscaled target frame for the
        ↪ motion vectors
    - 1 byte needed as it is only the Y channel
- Create (‘a(0)+pad+5‘, ‘c‘) * 3 bytes cache (called ‘write_cache‘)
    - This cache is to store the interpolated frame as it’s being written to
        ↪ save bandwidth
    - As 2 frames are interpolated between each input frame, 2 of these caches
        ↪ are required.
    - Set top 5 rows to 0
    - Start addressing after top 5 rows (top 5 rows are only used for hole
        ↪ filling)
- Create (‘a(0)+pad‘, ‘c‘) * (1 bit + 15 bits) (called ‘w_h_s_cache‘)
    - Used to store metadata regarding the interpolated frame
    - As there are 2 interpolated frames, 2 of this cache are needed
```

```
    - First bit - low if hole, high if filled
    - Following 15 bits - SAD score
    - Set all values to 0
- Create (2, ‘c‘/‘b_max‘) * 2 bytes cache (called ‘forward_vec_cache_0‘)
    - Used when increasing vector density for forward vectors
    - Records previous row of motion vectors
    - 1 byte for x
    - 1 byte for y
    - Initialised to 0
- Create (2, ‘c‘/(2*‘b_max‘)) * 2 bytes cache (called ‘forward_vec_cache_1‘)
    - Similar to above but for downscaled image
- Create (2, ‘c‘/‘b_max‘) * 2 bytes cache (called ‘backward_vec_cache_0‘)
    - Used when increasing vector density backward vectors
    - Records previous row of motion vectors
    - 1 byte for x
    - 1 byte for y
    - Initialised to 0
- Create (2, ‘c‘/(2*‘b_max‘)) * 2 bytes cache (called ‘backward_vec_cache_1‘)
    - Similar to above but for downscaled image
- Read into ‘source_win_cache_1‘ from the downscaled source frame in DRAM
- Read into ‘target_win_cache_1‘ from the downscaled target frame in DRAM
- Read into ‘source_win_cache_0‘ from the downscaled source frame in DRAM
- Read into ‘target_win_cache_0‘ from the downscaled target frame in DRAM
- When ‘a(1)‘ rows have been read in to both top level caches (the sub steps
    ↪ here are performed once in the forward direction and then in the backward
    ↪  direction. To perform backwards, swap ‘source_win_cache_n‘ with ‘
    ↪ target_win_cache_n‘ and ‘forward_vec_cache_n‘ with ‘backward_vec_cache_n‘)
    ↪ :
    - Read blocks sequentially from ‘source_win_cache_0‘
    - For each block:
        - Calculate motion vector by performing a full search in ‘
            ↪ target_win_cache_0‘
        - Store motion vector in ‘forward_vec_cache_0‘ for reference by its
            ↪ children and adjacent children
        - Read 4 children blocks from ‘source_win_cache_1‘
        - For each child block:
            - Perform search in 27 locations (as described in HBMA section)
            - Store motion vector in ‘forward_vec_cache_1‘ for reference by its
                ↪ children and adjacent children
            - Break block into 4 child blocks
                - If not currently using original image, then children come from
                    ↪ the layer above
                - If on the original image, halve the block size to get child
                    ↪ blocks
                - For each child:
                    - Perform 27 searches as stated before
                    - As its the original image, follow motion vector and write
                        ↪ child block into ‘write_cache‘ and update ‘w_h_s_cache‘
                        ↪  with corresponding metadata
```

```
                  - When `write_cache` is full, apply hole filling filter to top
                    ↪  `b_min` rows below the top 5 rows (they are only to
                    ↪ provide hole filling info) and then write those rows to
                    ↪  DRAM and shift the starting address by `b_min` rows.
                    ↪ set corresponding rows in `w_h_s_cache` to 0 and shift
                    ↪ the start address by the same amount

## Hardware Estimation:

### DRAM Writing Bandwidth:

- For each incoming frame (1/24 s):
    - Frame written to DRAM:
        - 3 * `r` * `c` bytes
    - Downscaled frames written to DRAM:
        - Sum (`i` = 1 -> `s`) (`r` * `c` / 2^`i`)
    - 2 interpolated frames written to DRAM:
        - 6 * `r` * `c` bytes
    - Total:
        - 9 * `r` * `c` + Sum (`i` = 1 -> `s`) (`r` * `c` / 2^`i`) bytes
- Total:
    - (9 * `r` * `c` + Sum (`i` = 1 -> `s`) (`r` * `c` / 2^`i`)) * 24 bytes / s

### DRAM Reading Bandwidth:

- For each incoming frame (1/24 s):
    - Every image saved in DRAM is read out (only Y channel):
        - 2 * Sum (`i` = 0 -> `s`) (`r` * `c` / 2^`i`) bytes
    - 2 colour channels from original source frame:
        - 2 * `r` * `c` bytes
    - Total:
        - 2 * `r` * `c` + 2 * Sum (`i` = 0 -> `s`) (`r` * `c` / 2^`i`) bytes
- Every 1/60 s:
    - Stream out a frame:
        - 3 * `r` * `c` bytes
- Total:
    228 * `r` * `c` + 48 * Sum (`i` = 0 -> `s`) (`r` * `c` / 2^`i`) bytes / s

### Required Cache Size:

- To store frames:
    - 6 * `b_min` * `c` bytes
- To downscale and store:
    - Sum (`i` = 1 -> `s`) (2 * `b_max` * `c`/2^`i`) bytes
- `win_cache` * 2:
    - 2 * (Sum (`i` = 0 -> `s`) (`a(i)` * `c`/2^`i`) + `pad` * `c`) bytes
- `write_cache` * 2:
    - 6 * (`a(0)+pad+5`) * `c` bytes
- `w_h_s_cache` * 2:
```

```
    - 4 * (`a(0)+pad`) * `c` bytes
- `vec_cache` * 2:
    - 2 * Sum (`i` = 0 -> `s`) (4 * `c`/(2^`i` * `b_max`))

### Example estimation:

For:
- `b_max` = 8
- `b_min` = 4
- `r` = 1080
- `c` = 1920
- `w` = 22
- `s` = 2

Results:
- DRAM write bandwidth = 462.7 MB/s
- DRAM read bandwidth = 617.0 MB/s
- Required cache size = 2.34 MB
```

A.2.7 Full Search and Advanced Bidirectional

```
# Full Search ME and Advanced Bidirectional MCI:

## Parameters:

- `b` = size of single axis of block in pixels
- `r` = number of rows of pixels in frame
- `c` = number of columns of pixels in frame
- `w` = size of single axis of search window in pixels

## Full Search:

- For each block in first image (`source block`):
    - Calculate SAD of `source block` with blocks of matching size in a window
        ↪ in the second image
        - For each block in the window in the second image (`target block`):
            - Calculate SAD of `source block` and `target block` and note
                ↪ corresponding vector
    - Take the vector corresponding to the lowest SAD and record in output with
        ↪ SAD (for occlusion in MCI) (in cases where there are multiple lowest
        ↪ SADs, precedence should be given to the smallest vector)
- Return block-wise motion vector field

## Advanced Bidirectional Interpolation:

Some of the details of this algorithm are quite complex so for a detailed
    ↪ explanation please refer to the paper:
"Motion-Compensated Frame Rate Up-ConversionPart II: New Algorithms for Frame
    ↪ Interpolation", 2010
```

```
by: Demin Wang, Senior Member, IEEE, Andr Vincent, Philip Blancheld, and Robert
    ↪ Klepko
https://ieeexplore.ieee.org/document/5440975
**Note:** As the motion vectors are generated with a non-fixed technique,
    ↪ occlusions are not noted at the ME stage. Disregard any steps that rely
    ↪ on this data in the above paper.

- Calculate forward and backward motion vectors using previous algorithm
    - Swap frames to get backward motion vectors
- Create `r` * `c` interpolated frame (to store pixel values)
- Create `r` * `c` SAD table (to store SADs (errors))
- Create `r` * `c` hole table (to tell if pixel is a hole)
- Create `r` * `c` weightings table (to record number of contributions to each
    ↪ pixel)
- Create a second version of the above tables for the image created using the
    ↪ reverse motion vectors
- For the forward motion vectors:
    - Perform IEWMC to generate interpolated frame
        - Expand the source block in question and apply weightings
        - Follow the vector to the location of the block in the interpolated
            ↪ frame
        - Accumulate values into the appropriate entries in the 4 tables listed
            ↪ above
- Repeat for the backward motion vectors
- Combine forward and backward images using error-adaptive combination
- Apply BDHI to fill holes
    - As this step can operate on cached data, we will not go into further
        ↪ details here, as it does not contribute to the bandwidth estimation or
        ↪  cache scheme used

## Full Integrated System:

To save on resources, the Y channel of YCbCr images are used for the motion
    ↪ estimation, as it has only a small effect on performance.

When describing cache, sizes are given as (rows, columns).

Images are assumed to be YCbCr. If not, a conversion is applied as it is
    ↪ streamed in.

The following stage is designed to store the incoming frame in a block wise
    ↪ format, as apposed to raster order.

- Create (2 * `b`, `c`) * 3 bytes cache
- As the first frame streams in, place the pixels into the cache
- When `b` rows have been streamed in, start writing the cache to DRAM in a
    ↪ block wise fashion
    - All 3 channels are written to DRAM as separate 1 channel images
    - Blocks are flattened and then written in contiguously
```

```
    - Left to right
    - Stream frame into the other ‘b‘ rows while writing
    - Repeat until full image in DRAM and stored block wise

After first frame, the following stages happen in a loop with each incoming
    ↪ frame (target frame):

- Create (‘w‘, ‘c‘) * 1 byte cache (called ‘source_win_cache‘)
    - This cache will be used to search the source frame for the motion vectors
- Create (‘w‘ + ‘b‘, ‘c‘) * 1 byte cache (called ‘target_win_cache‘)
    - This cache will be used to search the target frame for the motion vectors
    - The height is ‘w‘+‘b‘ so that the first row of windows can be searched
        ↪ while the values needed for the next row are streamed in
- Create (‘w‘ + 2, ‘c‘) * 3 bytes cache (called ‘write_cache‘)
    - This cache is to store the interpolated frame as it’s being written to
        ↪ save bandwidth
    - For each interpolated frame, create 2 caches
        - 1 for forward motion image
        - 1 for backward motion image
    - As 2 frames are interpolated between each input frame, 4 of these caches
        ↪ are required
    - Set top 2 rows to 0
    - Start addressing after top 2 rows (top 2 rows are only used for hole
        ↪ filling)
- Create (‘w‘, ‘c‘) * (1 bit + 15 bits + 16 bits) (called ‘w_h_s_cache‘)
    - Used to store metadata regarding the interpolated frame
    - For each interpolated frame, create 2 caches
        - 1 for forward motion image
        - 1 for backward motion image
    - As there are 2 interpolated frames, 4 of this cache are needed
    - First bit - low if hole, high if filled
    - Following 15 bits - Accumulated weightings
    - Following 16 bits - SAD score (error)
    - Set all values to 0
- Stream frame into DRAM following procedure outlined for first frame
- Also stream greyscale values from frame into ‘target_win_cache‘ in parallel
- Also pull out greyscale values from the source frame into ‘source_win_cache‘
- When ‘w‘ rows have been read in to ‘target_win_cache‘ and ‘source_win_cache‘:
    - For the forward motion image:
        - Continue streaming greyscale pixels into other ‘b‘ rows of ‘
            ↪ target_win_cache‘ from target frame
        - Taking a block from the ‘source_win_cache‘:
            - Search the corresponding search window in ‘target_win_cache‘
                - Returns the best motion vector and SAD score
            - Pull the other 2 channels for the block out of DRAM
            - Expand the block by doubling its size and apply the weighting
                ↪ filter
            - Write the expanded block to ‘write_cache‘ by following the motion
                ↪ vector
```

```
              - Change the `w_h_s_cache` for every modified pixel:
                  - to indicate that the pixel is not a hole
                  - Accumulate the SAD of the pixel
                  - Accumulate the weighting of the pixel
       - Do the same as the previous step in parallel but swap `source_win_cache
         ↪ ` and `target_win_cache`
       - Repeat the last 2 stages for the whole row of blocks in both `
         ↪ source_win_cache` and `target_win_cache`
   - Repeat for the backwards motion image
   - Ignoring the top 2 rows in `write_cache`, for the following `b` rows:
       - Normalise each pixel according to its accumulated weighting
       - For each corresponding pixel in the forward and backwards cache, apply
         ↪ error-adaptive combination algorithm
       - Apply BDHI to fill holes (can use the 2 top in calculations)
       - Write the same `b` rows to DRAM (for the interpolated frame) and set
         ↪ all values to 0
       - Set the corresponding values in `w_h_s_cache` to 0
   - Shift the the start address of both the `write_cache` and `w_h_s_cache` by
     ↪  `b` rows
   - Repeat until whole image is read in

## Hardware Estimation:

### DRAM Writing Bandwidth:

- For each incoming frame (1/24 s):
   - Entire frame is written to DRAM:
       - 3 * `r` * `c` bytes
   - 2 interpolated frames are written to DRAM:
       - 6 * `r` * `c` bytes
   - Total:
       - 9 * `r` * `c` bytes
- Total:
   - 216 * `r` * `c` bytes / s

### DRAM Reading Bandwidth:

- For each incoming frame (1/24 s):
   - Entire previous frame is pulled out block by block (wasted data minimised)
       - 3 * `r` * `c` bytes
   - 2 channels read from the incoming frame
       - 2 * `r` * `c` bytes
- Every 1/60 s:
   - Stream out the frame for this time stamp:
       - 3 * `r` * `c` bytes
- Total:
   - 300 * `r` * `c` bytes / s

### Required Cache Size:
```

```
- To store frame block wise:
    - 6 * `b`* `c` bytes
- `source_win_cache`:
    - (`w` + `b`) * `c` bytes
- `target_win_cache`:
    - (`w` + `b`) * `c` bytes
- `write_cache` * 4:
    - 12 * (`w` + 2) * `c` bytes
- `w_h_s_cache` * 4:
    - 16 * `w` * `c`
- Total:
    - 2 * `c` * (4 * `b` + 15 * `w` + 12) bytes

### Example Estimations:

For:
- `b` = 8
- `r` = 1080
- `c` = 1920
- `w` = 22

Results:
- DRAM write bandwidth = 427.1 MB/s
- DRAM read bandwidth = 593.3 MB/s
- Required cache size = 1.21 MB
```

A.2.8 Three Step Search and Advanced Bidirectional

```
# Three Step Search ME and Advanced Bidirectional MCI:

## Parameters:

- `b` = size of single axis of block in pixels
- `r` = number of rows of pixels in frame
- `c` = number of columns of pixels in frame
- `w` = size of single axis of search window in pixels

## Three Step Search:

- For each block in first image (`source block`):
    - `center` = `source block`
    - For `step` in (`steps`-1 -> 1) (default value of `steps` is 3):
        - `space` = 2^`step`
        - For each block in 3x3 around `center`, `space` pixels apart (`target
          ↪ block`):
            - Calculate SAD between `source block` and `target block` and note
              ↪ corresponding vector (no need for `center` block as it has been
              ↪  previously calculated)
```

```
        - `center` = `target block` with lowest SAD
    - `space` = 1
    - For each block in 3x3 around `center`, `space` pixels apart (`target block
        ↪ `):
        - Calculate SAD between `source block` and `target block` and note
            ↪ corresponding vector (no need for `center` block as it has been
            ↪ previously calculated)
    - Assign vector with lowest corresponding SAD to `source block`
- Return block-wise motion vector field

## Advanced Bidirectional Interpolation:

Some of the details of this algorithm are quite complex so for a detailed
    ↪ explanation please refer to the paper:
"Motion-Compensated Frame Rate Up-ConversionPart II: New Algorithms for Frame
    ↪ Interpolation", 2010
by: Demin Wang, Senior Member, IEEE, Andr Vincent, Philip Blancheld, and Robert
    ↪ Klepko
https://ieeexplore.ieee.org/document/5440975
**Note:** As the motion vectors are generated with a non-fixed technique,
    ↪ occlusions are not noted at the ME stage. Disregard any steps that rely
    ↪ on this data in the above paper.

- Calculate forward and backward motion vectors using previous algorithm
    - Swap frames to get backward motion vectors
- Create `r` * `c` interpolated frame (to store pixel values)
- Create `r` * `c` SAD table (to store SADs (errors))
- Create `r` * `c` hole table (to tell if pixel is a hole)
- Create `r` * `c` weightings table (to record number of contributions to each
    ↪ pixel)
- Create a second version of the above tables for the image created using the
    ↪ reverse motion vectors
- For the forward motion vectors:
    - Perform IEWMC to generate interpolated frame
        - Expand the source block in question and apply weightings
        - Follow the vector to the location of the block in the interpolated
            ↪ frame
        - Accumulate values into the appropriate entries in the 4 tables listed
            ↪ above
- Repeat for the backward motion vectors
- Combine forward and backward images using error-adaptive combination
- Apply BDHI to fill holes
    - As this step can operate on cached data, we will not go into further
        ↪ details here, as it does not contribute to the bandwidth estimation or
        ↪  cache scheme used

## Full Integrated System:
```

```
To save on resources, the Y channel of YCbCr images are used for the motion
    ↪ estimation, as it has only a small effect on performance.

When describing cache, sizes are given as (rows, columns).

Images are assumed to be YCbCr. If not, a conversion is applied as it is
    ↪ streamed in.

The following stage is designed to store the incoming frame in a block wise
    ↪ format, as apposed to raster order.

- Create (2 * `b`, `c`) * 3 bytes cache
- As the first frame streams in, place the pixels into the cache
- When `b` rows have been streamed in, start writing the cache to DRAM in a
    ↪ block wise fashion
    - All 3 channels are written to DRAM as separate 1 channel images
    - Blocks are flattened and then written in contiguously
    - Left to right
    - Stream frame into the other `b` rows while writing
    - Repeat until full image in DRAM and stored block wise

After first frame, the following stages happen in a loop with each incoming
    ↪ frame (target frame):

- Create (`w`, `c`) * 1 byte cache (called `source_win_cache`)
    - This cache will be used to search the source frame for the motion vectors
- Create (`w` + `b`, `c`) * 1 byte cache (called `target_win_cache`)
    - This cache will be used to search the target frame for the motion vectors
    - The height is `w`+`b` so that the first row of windows can be searched
        ↪ while the values needed for the next row are streamed in
- Create (`w` + 2, `c`) * 3 bytes cache (called `write_cache`)
    - This cache is to store the interpolated frame as it's being written to
        ↪ save bandwidth
    - For each interpolated frame, create 2 caches
        - 1 for forward motion image
        - 1 for backward motion image
    - As 2 frames are interpolated between each input frame, 4 of these caches
        ↪ are required
    - Set top 2 rows to 0
    - Start addressing after top 2 rows (top 2 rows are only used for hole
        ↪ filling)
- Create (`w`, `c`) * (1 bit + 15 bits + 16 bits) (called `w_h_s_cache`)
    - Used to store metadata regarding the interpolated frame
    - For each interpolated frame, create 2 caches
        - 1 for forward motion image
        - 1 for backward motion image
    - As there are 2 interpolated frames, 4 of this cache are needed
    - First bit - low if hole, high if filled
    - Following 15 bits - Accumulated weightings
```

```
    - Following 16 bits - SAD score (error)
    - Set all values to 0
- Stream frame into DRAM following procedure outlined for first frame
- Also stream greyscale values from frame into `target_win_cache` in parallel
- Also pull out greyscale values from the source frame into `source_win_cache`
- When `w` rows have been read in to `target_win_cache` and `source_win_cache`:
    - For the forward motion image:
        - Continue streaming greyscale pixels into other `b` rows of `
          ↪ target_win_cache` from target frame
        - Taking a block from the `source_win_cache`:
            - Search the corresponding search window in `target_win_cache`
              ↪ following three step search algorithm outlined above
              - Returns the best motion vector and SAD score
            - Pull the other 2 channels for the block out of DRAM
            - Expand the block by doubling its size and apply the weighting
              ↪ filter
            - Write the expanded block to `write_cache` by following the motion
              ↪ vector
              - Change the `w_h_s_cache` for every modified pixel:
                  - to indicate that the pixel is not a hole
                  - Accumulate the SAD of the pixel
                  - Accumulate the weighting of the pixel
        - Do the same as the previous step in parallel but swap `source_win_cache
          ↪ ` and `target_win_cache`
        - Repeat the last 2 stages for the whole row of blocks in both `
          ↪ source_win_cache` and `target_win_cache`
    - Repeat for the backwards motion image
    - Ignoring the top 2 rows in `write_cache`, for the following `b` rows:
        - Normalise each pixel according to its accumulated weighting
        - For each corresponding pixel in the forward and backwards cache, apply
          ↪ error-adaptive combination algorithm
        - Apply BDHI to fill holes (can use the 2 top in calculations)
        - Write the same `b` rows to DRAM (for the interpolated frame) and set
          ↪ all values to 0
        - Set the corresponding values in `w_h_s_cache` to 0
    - Shift the the start address of both the `write_cache` and `w_h_s_cache` by
      ↪  `b` rows
    - Repeat until whole image is read in

## Hardware Estimation:

### DRAM Writing Bandwidth:

- For each incoming frame (1/24 s):
    - Entire frame is written to DRAM:
        - 3 * `r` * `c` bytes
    - 2 interpolated frames are written to DRAM:
        - 6 * `r` * `c` bytes
    - Total:
```

```
        - 9 * `r` * `c` bytes
- Total:
    - 216 * `r` * `c` bytes / s

### DRAM Reading Bandwidth:

- For each incoming frame (1/24 s):
    - Entire previous frame is pulled out block by block (wasted data minimised)
        - 3 * `r` * `c` bytes
    - 2 channels read from the incoming frame
        - 2 * `r` * `c` bytes
- Every 1/60 s:
    - Stream out the frame for this time stamp:
        - 3 * `r` * `c` bytes
- Total:
    - 300 * `r` * `c` bytes / s

### Required Cache Size:

- To store frame block wise:
    - 6 * `b`* `c` bytes
- `source_win_cache`:
    - (`w` + `b`) * `c` bytes
- `target_win_cache`:
    - (`w` + `b`) * `c` bytes
- `write_cache` * 4:
    - 12 * (`w` + 2) * `c` bytes
- `w_h_s_cache` * 4:
    - 16 * `w` * `c`
- Total:
    - 2 * `c` * (4 * `b` + 15 * `w` + 12) bytes

### Example Estimations:

For:
- `b` = 8
- `r` = 1080
- `c` = 1920
- `w` = 22

Results:
- DRAM write bandwidth = 427.1 MB/s
- DRAM read bandwidth = 593.3 MB/s
- Required cache size = 1.21 MB
```

A.2.9 Hierarchical Block Matching Algorithm and Advanced Bidirectional

```
# HBMA ME and Advanced Bidirectional MCI:
```

```
## Parameters:

- ‘b_max‘ = max size of single axis of block in pixels
- ‘b_min‘ = min size of single axis of block in pixels
- ‘r‘ = number of rows of pixels in frame
- ‘c‘ = number of columns of pixels in frame
- ‘w‘ = size of single axis of search window in pixels
- ‘s‘ = number of times the frames are downscaled

## HBMA:

- Block size must be power of 2
- Reduce first and second images ‘steps‘ times using linear filter, storing each
    ↪  image
    - Weightings for linear filter:
      | | | |
      |------|-----|------|
      |0.0625|0.125|0.0625|
      |0.125 |0.25 |0.125 |
      |0.0625|0.125|0.0625|
    - Convolve filter with images with stride of (2,2) to downscale
- Perform full search algorithm on smallest first and second frames
- For next smallest images -> original images:
    - Increase motion vector density
        - Because the image size is halved when downscaled, with a consistent
          ↪ block size, a ("parent") block in the downscaled image is 4 ("
          ↪ child") blocks in the original image
        - For each of the 4 child blocks for every block in the previous
          ↪ iteration:
            - Search areas in a (3x3) window around position pointed to by parent
              ↪  vector, and the 2 vectors associated with 2 blocks adjacent to
              ↪  the parent block (parent vectors need to be multiplied by 2 to
              ↪  account for change in image size)
            - Select vector corresponding to the smallest SAD
- For ‘b_max‘ -> ‘b_min‘ by halving:
    - Increase motion vector density
        - By halving the block size, each previous ("parent") block holds 4 ("
          ↪ child") blocks
        - For each of the 4 child blocks for every block in the previous
          ↪ iteration:
            - Search areas in a (3x3) window around position pointed to by parent
              ↪  vector, and the 2 vectors associated with 2 blocks adjacent to
              ↪  the parent block
            - Select vector corresponding to the smallest SAD
- Return block-wise motion vector field

## Advanced Bidirectional Interpolation:
```

```
Some of the details of this algorithm are quite complex so for a detailed explanation please refer to the paper:
"Motion-Compensated Frame Rate Up-ConversionPart II: New Algorithms for Frame Interpolation", 2010
by: Demin Wang, Senior Member, IEEE, Andr Vincent, Philip Blancheld, and Robert Klepko
https://ieeexplore.ieee.org/document/5440975
**Note:** As the motion vectors are generated with a non-fixed technique, occlusions are not noted at the ME stage. Disregard any steps that rely on this data in the above paper.

- Calculate forward and backward motion vectors using previous algorithm
  - Swap frames to get backward motion vectors
- Create `r` * `c` interpolated frame (to store pixel values)
- Create `r` * `c` SAD table (to store SADs (errors))
- Create `r` * `c` hole table (to tell if pixel is a hole)
- Create `r` * `c` weightings table (to record number of contributions to each pixel)
- Create a second version of the above tables for the image created using the reverse motion vectors
- For the forward motion vectors:
  - Perform IEWMC to generate interpolated frame
    - Expand the source block in question and apply weightings
    - Follow the vector to the location of the block in the interpolated frame
    - Accumulate values into the appropriate entries in the 4 tables listed above
- Repeat for the backward motion vectors
- Combine forward and backward images using error-adaptive combination
- Apply BDHI to fill holes
  - As this step can operate on cached data, we will not go into further details here, as it does not contribute to the bandwidth estimation or  cache scheme used

## Full Integrated System:

To save on resources, the Y channel of YCbCr images are used for the motion estimation, as it has only a small effect on performance.

When describing cache, sizes are given as (rows, columns).

Some required formulas:

- `a(i)` = 2^(`s` - `i`) * `w` + Sum (`n` = 1 -> `s` - `i`) (2^`n`)
- `pad` = Sum (`n` = 1 -> log_2 (`b_max`/`b_min`)) (2^`n`)

For simplicity of explanation, `s`=1 has been used in the algorithm:
```

```
The following stage is designed to store the incoming frame in a block wise
    ↪ format, as apposed to raster order.

- Create (2 * ‘b_min‘, ‘c‘) * 3 bytes cache
    - For original frame so it can be written block wise
- Create (2 * ‘b_max‘, ‘c‘/2) * 1 byte cache
    - For downscaled frame so that it can be written block wise
- As the first frame streams in, place the pixels into the first cache
- When ‘b_max‘ rows have been streamed in, start writing the cache to DRAM in a
    ↪ block wise fashion
    - Blocks are flattened and then written in contiguously
    - Left to right
    - Channels stored separately
    - Stream frame into the other ‘b_max‘ rows while writing
    - Repeat until full image in DRAM and stored block wise
- Concurrently, when 3 rows have been streamed in, begin applying downscale
    ↪ filter to image and store in second cache
    - Once ‘b_max‘ rows of downscaled image have been created, save to DRAM in
        ↪ block wise format (as above)

After first frame, the following stages happen in a loop with each incoming
    ↪ frame (‘target_frame‘):

- Stream ‘target_frame‘ into DRAM following procedure outlined for first frame
- Create (‘a(0)+pad‘, ‘c‘) * 1 byte cache (called ‘source_win_cache_0‘)
    - This cache will be used to search the largest scale source frame for the
        ↪ motion vectors
    - Number of rows is ‘a(0)+pad‘, as ‘a(0)‘ is the range of motion a block can
        ↪  take, and the ‘pad‘ is so that the cache can be reused for future
        ↪ density increases
    - 1 byte is needed because it is only the Y channel
- Create (‘a(1)‘, ‘c‘/2) * 1 byte (called ‘source_win_cache_1‘)
    - This cache will be used to search the downscaled source frame for the
        ↪ motion vectors
    - 1 byte needed as it is only the Y channel
- Create (‘a(0)+pad‘, ‘c‘) * 1 byte cache (called ‘target_win_cache_0‘)
    - This cache will be used to search the largest scale target frame for the
        ↪ motion vectors
    - Number of rows is ‘a(0)+pad‘, as ‘a(0)‘ is the range of motion a block can
        ↪  take, and the ‘pad‘ is so that the cache can be reused for future
        ↪ density increases
    - 1 byte is needed because it is only the Y channel
- Create (‘a(1)‘, ‘c‘/2) * 1 byte (called ‘target_win_cache_1‘)
    - This cache will be used to search the downscaled target frame for the
        ↪ motion vectors
    - 1 byte needed as it is only the Y channel
- Create (‘a(0)+pad+2‘, ‘c‘) * 3 bytes cache (called ‘write_cache‘)
    - This cache is to store the interpolated frame as it’s being written to
        ↪ save bandwidth
```

```
    - For each interpolated frame, create 2 caches
        - 1 for forward motion image
        - 1 for backward motion image
    - As 2 frames are interpolated between each input frame, 4 of these caches
        ↪ are required.
    - Set top 5 rows to 0
    - Start addressing after top 5 rows (top 5 rows are only used for hole
        ↪ filling)
- Create (‘a(0)+pad‘, ‘c‘) * (1 bit + 15 bits + 16 bits) (called ‘w_h_s_cache‘)
    - Used to store metadata regarding the interpolated frame
    - For each interpolated frame, create 2 caches
        - 1 for forward motion image
        - 1 for backward motion image
    - As there are 2 interpolated frames, 4 of this cache are needed
    - First bit - low if hole, high if filled
    - Following 15 bits - Accumulated weightings
    - Following 16 bits - Accumulated SAD (error)
    - Set all values to 0
- Create (2, ‘c‘/‘b_max‘) * 2 bytes cache (called ‘forward_vec_cache_0‘)
    - Used when increasing vector density for forward vectors
    - Records previous row of motion vectors
    - 1 byte for x
    - 1 byte for y
    - Initialised to 0
- Create (2, ‘c‘/(2*‘b_max‘)) * 2 bytes cache (called ‘forward_vec_cache_1‘)
    - Similar to above but for downscaled image
- Create (2, ‘c‘/‘b_max‘) * 2 bytes cache (called ‘backward_vec_cache_0‘)
    - Used when increasing vector density backward vectors
    - Records previous row of motion vectors
    - 1 byte for x
    - 1 byte for y
    - Initialised to 0
- Create (2, ‘c‘/(2*‘b_max‘)) * 2 bytes cache (called ‘backward_vec_cache_1‘)
    - Similar to above but for downscaled image
- Read into ‘source_win_cache_1‘ from the downscaled source frame in DRAM
- Read into ‘target_win_cache_1‘ from the downscaled target frame in DRAM
- Read into ‘source_win_cache_0‘ from the downscaled source frame in DRAM
- Read into ‘target_win_cache_0‘ from the downscaled target frame in DRAM
- When ‘a(1)‘ rows have been read in to both top level caches (the sub steps
    ↪ here are performed once in the forward direction and then in the backward
    ↪  direction. To perform backwards, swap ‘source_win_cache_n‘ with ‘
    ↪ target_win_cache_n‘ and ‘forward_vec_cache_n‘ with ‘backward_vec_cache_n‘)
    ↪ :
    - Read blocks sequentially from ‘source_win_cache_0‘
    - For each block:
        - Calculate motion vector by performing a full search in ‘
            ↪ target_win_cache_0‘
        - Store motion vector in ‘forward_vec_cache_0‘ for reference by its
            ↪ children and adjacent children
```

```
        - Read 4 children blocks from `source_win_cache_1`
        - For each child block:
            - Perform search in 27 locations (as described in HBMA section)
            - Store motion vector in `forward_vec_cache_1` for reference by its
                ↪ children and adjacent children
            - Break block into 4 child blocks
                - If not currently using original image, then children come from
                    ↪ the layer above
                - If on the original image, halve the block size to get child
                    ↪ blocks
                - For each child:
                    - Perform 27 searches as stated before
                    - If working on the original image with the min block size,
                        ↪ follow motion vector and write child block into `
                        ↪ write_cache` and update `w_h_s_cache` with
                        ↪ corresponding metadata
                    - When A full row of blocks have been processed, move to the
                        ↪ next step
- Repeat for backwards motion (writing into appropriate caches) (this can be
    ↪ done in parallel with previous step)
- Ignoring the top 2 rows in `write_cache`, for the following `b_min` rows:
    - Normalise each pixel according to its accumulated weighting
    - For each corresponding pixel in the forward and backwards cache, apply
        ↪ error-adaptive combination algorithm
    - Apply BDHI to fill holes (can use the 2 top rows in calculations)
    - Write the same `b_min` rows to DRAM (for the interpolated frame) and set
        ↪ all values to 0
    - Set the corresponding values in `w_h_s_cache` to 0
- Shift the the start address of both the `write_cache` and `w_h_s_cache` by `
    ↪ b_min` rows
- Repeat the last 4 steps until all of the image has been processed

## Hardware Estimation:

### DRAM Writing Bandwidth:

- For each incoming frame (1/24 s):
    - Frame written to DRAM:
        - 3 * `r` * `c` bytes
    - Downscaled frames written to DRAM:
        - Sum (`i` = 1 -> `s`) (`r` * `c` / 2^`i`)
    - 2 interpolated frames written to DRAM:
        - 6 * `r` * `c` bytes
    - Total:
        - 9 * `r` * `c` + Sum (`i` = 1 -> `s`) (`r` * `c` / 2^`i`) bytes
- Total:
    - (9 * `r` * `c` + Sum (`i` = 1 -> `s`) (`r` * `c` / 2^`i`)) * 24 bytes / s

### DRAM Reading Bandwidth:
```

```
- For each incoming frame (1/24 s):
    - Every image saved in DRAM is read out (only Y channel):
        - 2 * Sum (‘i‘ = 0 -> ‘s‘) (‘r‘ * ‘c‘ / 2^‘i‘) bytes
    - 2 colour channels from original source frame:
        - 2 * ‘r‘ * ‘c‘ bytes
    - Total:
        - 2 * ‘r‘ * ‘c‘ + 2 * Sum (‘i‘ = 0 -> ‘s‘) (‘r‘ * ‘c‘ / 2^‘i‘) bytes
- Every 1/60 s:
    - Stream out a frame:
        - 3 * ‘r‘ * ‘c‘ bytes
- Total:
    228 * ‘r‘ * ‘c‘ + 48 * Sum (‘i‘ = 0 -> ‘s‘) (‘r‘ * ‘c‘ / 2^‘i‘) bytes / s

### Required Cache Size:

- To store frames:
    - 6 * ‘b_min‘ * ‘c‘ bytes
- To downscale and store:
    - Sum (‘i‘ = 1 -> ‘s‘) (2 * ‘b_max‘ * ‘c‘/2^‘i‘) bytes
- ‘win_cache‘ * 2:
    - 2 * (Sum (‘i‘ = 0 -> ‘s‘) (‘a(i)‘ * ‘c‘/2^‘i‘) + ‘pad‘ * ‘c‘) bytes
- ‘write_cache‘ * 4:
    - 12 * (‘a(0)+pad+2‘) * ‘c‘ bytes
- ‘w_h_s_cache‘ * 4:
    - 16 * (‘a(0)+pad‘) * ‘c‘ bytes
- ‘vec_cache‘ * 2:
    - 2 * Sum (‘i‘ = 0 -> ‘s‘) (4 * ‘c‘/(2^‘i‘ * ‘b_max‘))

### Example estimation:

For:
- ‘b_max‘ = 8
- ‘b_min‘ = 4
- ‘r‘ = 1080
- ‘c‘ = 1920
- ‘w‘ = 22
- ‘s‘ = 2

Results:
- DRAM write bandwidth = 462.7 MB/s
- DRAM read bandwidth = 617.0 MB/s
- Required cache size = 5.49 MB
```