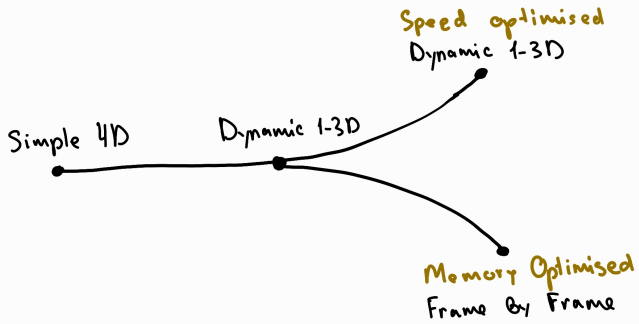
Binary Video Encoder

Kyryl Lebedin

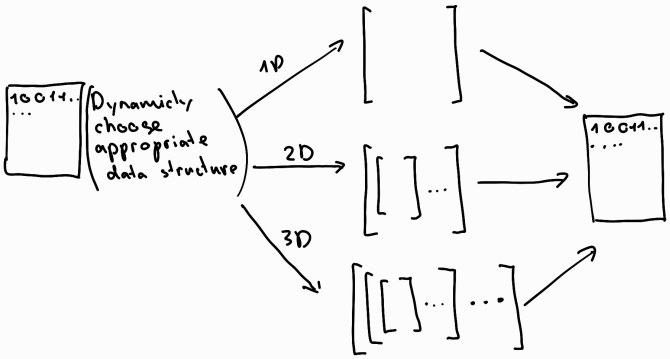
Binary read/write Evolution



Dynamic 1-3D and Simple 4D

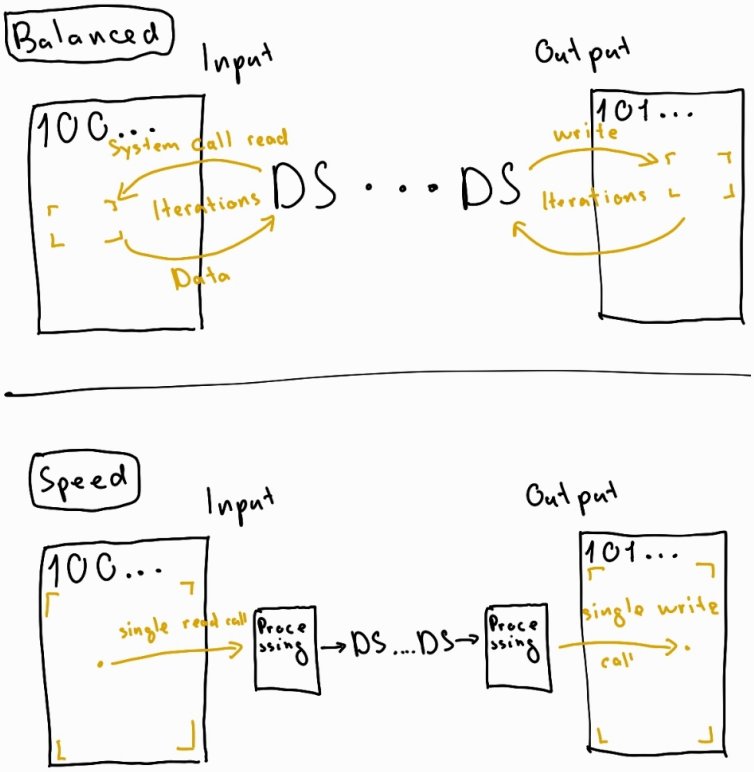
Iintial implementation - store the each pixel, channel, frame as separate element in 4D array, quickly proved to be slow and memory inefficient.

New implementation was Struct that could hold either 1D, 2D or 3D array dynamically. Quickly perform operations on channels - 3D, frames - 2D. This implementation is used in all Balanced (-B) versions and its modified version in every -S.



Speed Optimized Dynamic 1-3D

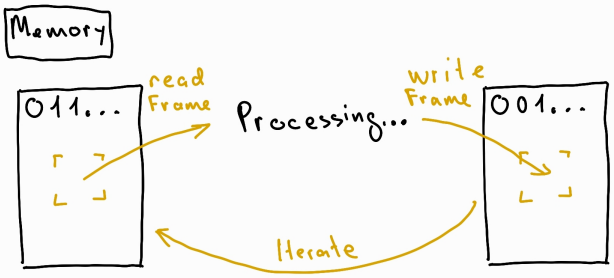
The main difference between Balanced and Speed implementations is that in -S read and write are done in one call on the whole content of binary file, to optimize speed for slow read/write conditions, such that flash drive or slow HDD, by minimizing number of read/write calls.
As program reads the whole file, it might need additional implementation to handle very big files.



Balanced - uses multiple read/write, to build ds on the way. Speed - one read, one write. The dominance of -S in slow IO is shown later.

-M IO Frame by Frame

Every -M reads one frame processes it accordingly and writes processed frame to output.
This allows to lower the peak space usage independent of number of frames n in video: O(1); while in other implementations it is O(n).



Final comparison

	Pros	Cons
Speed	• Fast no matter how slow IO system operations are • Signifficantly outperforms other implementations in slow IO conditions	• Has overhead to process IO data • Uses more memory
Memory	• Space complexity O(1) relative to number of frames, uses much less memory • If speed of system IO is fast, tend to run faster than -S and -B	• Heavily dependent on speed of IO operations
Balanced	• Less dependent on IO speed than -M • Doesn't have data processing overhead like -M	• Not as fast as -S if system IO is slow • Not as fast and memory efficient as -M

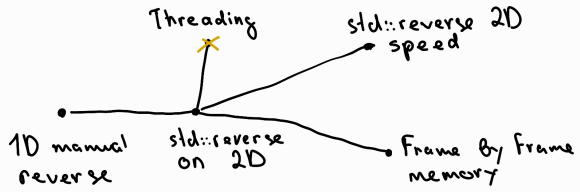
Function	Allocations/Frees	Peak Memory Usage (KB)
Reverse -B	107 / 107	5,008
-S	111 / 111	14,839
-M	7 / 7	140
Swap Channel -B	1,406 / 1,406	5,020
-S	1,409 / 1,409	9,935
-M	1,010 / 1,010	160
Clip -B	1,911 / 1,911	5,022
-S	1,914 / 1,914	9,938
-M	1,511 / 1,511	147
Scale -B	1,913 / 1,913	5,024
-S	1,916 / 1,916	9,939
-M	1,513 / 1,513	148

Basic Functions Implementation

In general: same algorithms used to process data in -B and -S, but different IO approaches, as said above.
-M: read-process-write for every chunk.

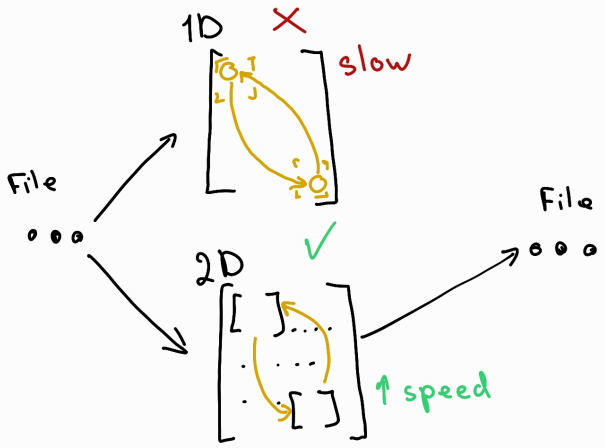
Memory usage: `valgrind ./runme ...` on provided test.bin.

Reverse



reverse evolution

1D manual vs 2d std::reverse



The first implementation of reverse: use flattened 1D vector, where pixels are swapped for each frame.

There was attempt to implement threads into this functionality which proved to be inefficient due to threads create/delete overhead.

Better 2D implementation: pointers to frames rather than actual pixels. Gives increase in speed by 41%, because data was structured out of the box, overhead to calculate flattened indexes was removed.

-S always uses more memory to minimize system read/write calls, shown in table.

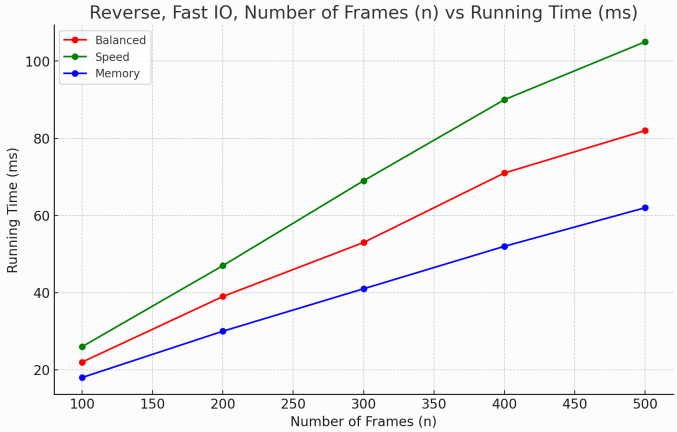
Frame by frame reverse

-M: read from end, write into start.

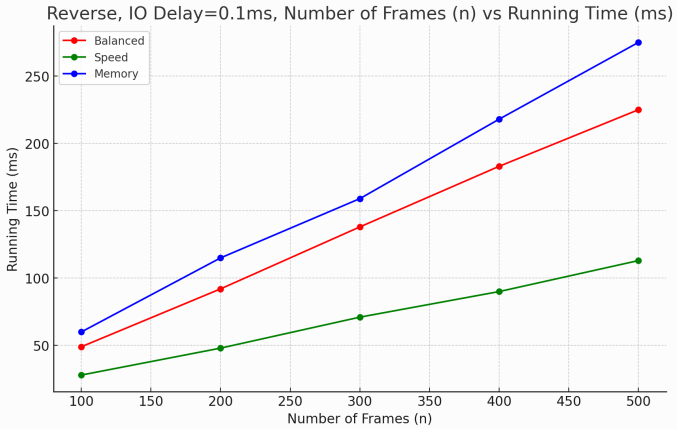
Speed

I want to demonstrate the two variations of running speed. One with running it on Mira with fast by default read/write operations. Another with artificially introduced delay 0.1ms (0.4% of average running speed for test.bin). The delay happens after every read/write operation with the help of custom implemented class.

Mean Time is measured with <chrono> for the full function runtime 100 times.

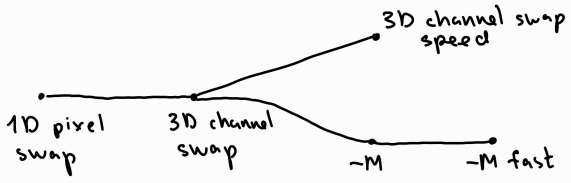


Speed: -S < -B < -M



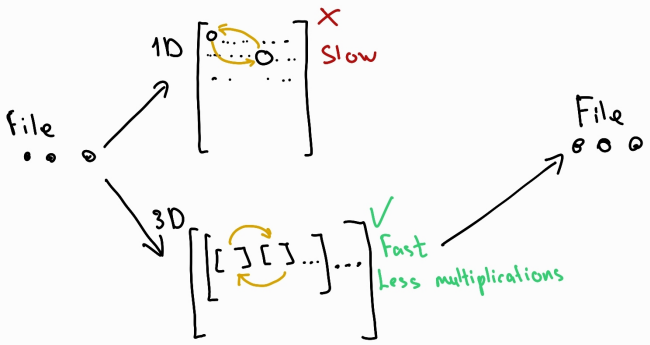
Speed: -M < -B < -S. As delay is introduced, Speed now always runs significantly faster, as it is relies less on system IO speed.
This pattern is found in all the further speed evaluations.

Swap Channel



swap_channel evolution

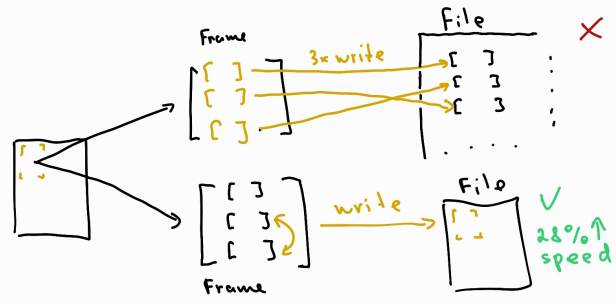
1D pixel swap vs 3D channel swap



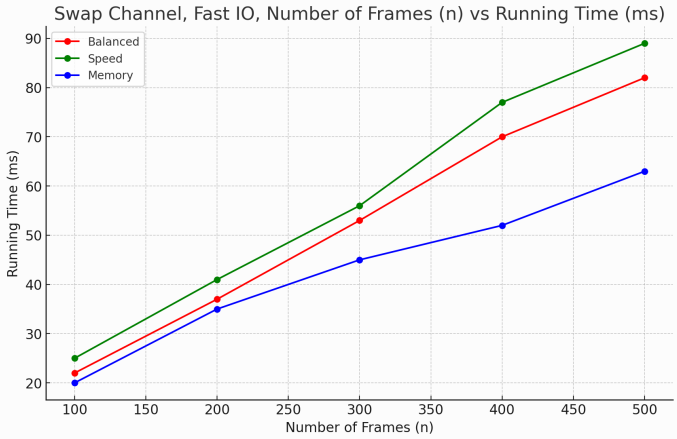
Initial -S/-B: swap every pixel in channels calculated in flat array.
Improved: use 3D array instead, with same benefits as for reverse.

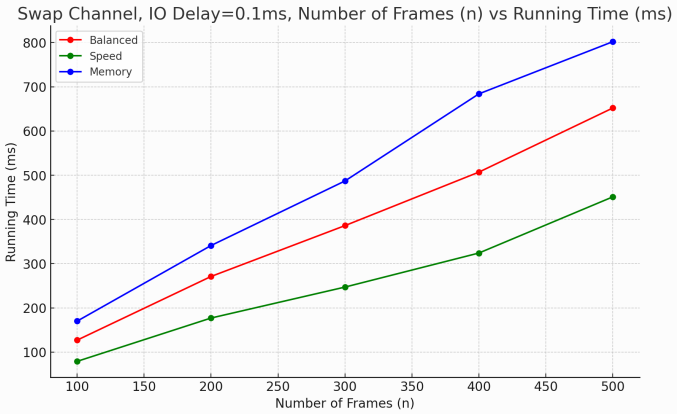
1D vs 2D -M implementation

Use 2D for faster swaps.



Speed

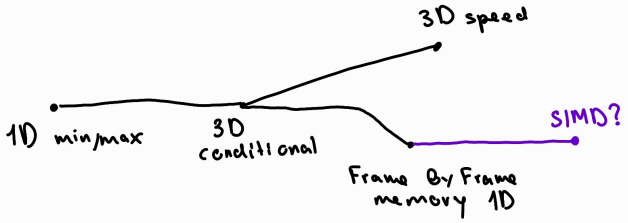




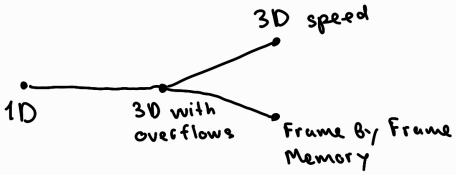
Clip and Scale Channel

As algorithms are similar sections are combined.

Using SIMD, to process data simultaneously on memory level can significantly increase speed, but implementing of SIMD requires accurate work directly with architecture, that program was built to run on.

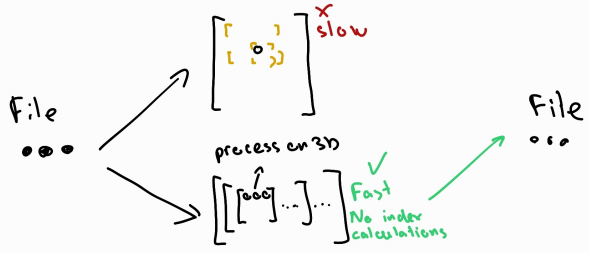


clip_channel



scale_channel

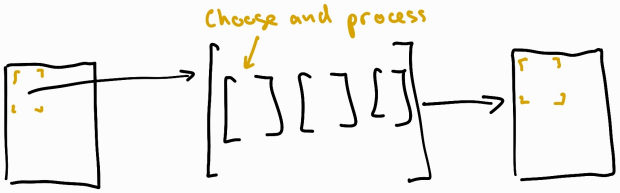
1D vs 3D



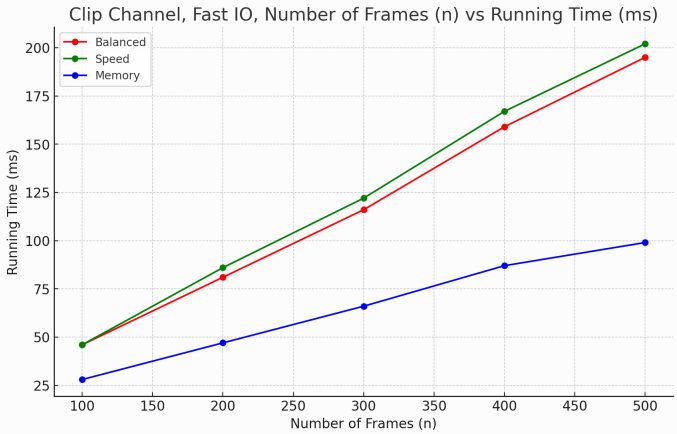
For -S and -B found out that using 3D vector for finding channels is faster than my initial implementation with flat vector.

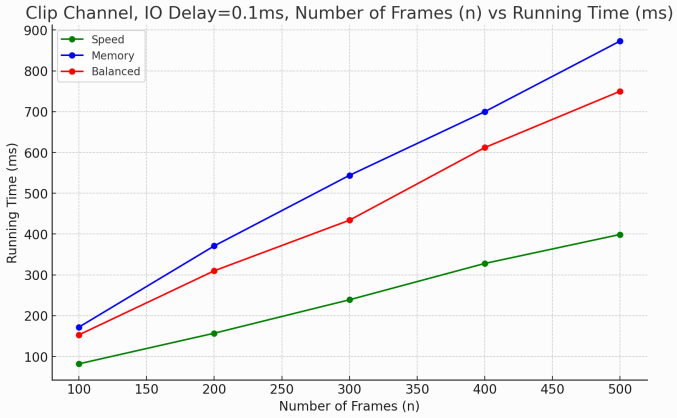
Memory 2D implementation

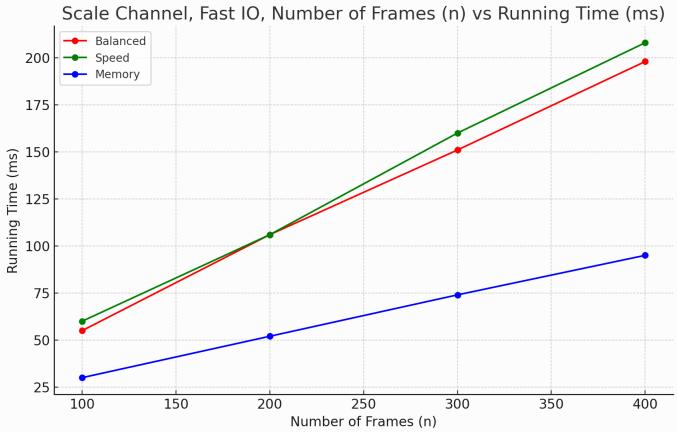
By trying different implementation found out that using 2D to access channels pixels is the fastest way.

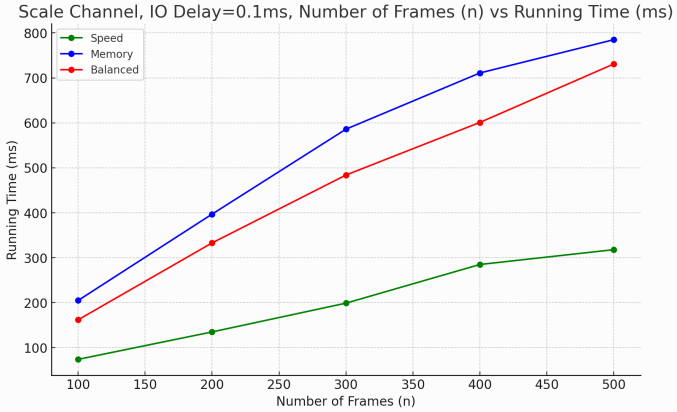


Speed optimisation and evaluation








Scale Channel, IO Delay=0.1ms, Number of Frames (n) vs Running Time (ms)
Speed
Memory
Balanced
Running Time (ms)
Number of Frames (n)
100
150
200
250
300
350
400
450
500
800
700
600
500
400
300
200
100