# Practical Real-Time Video Rendering with Modern OpenGL and GStreamer

Heinrich Fink (R&D Engineer)

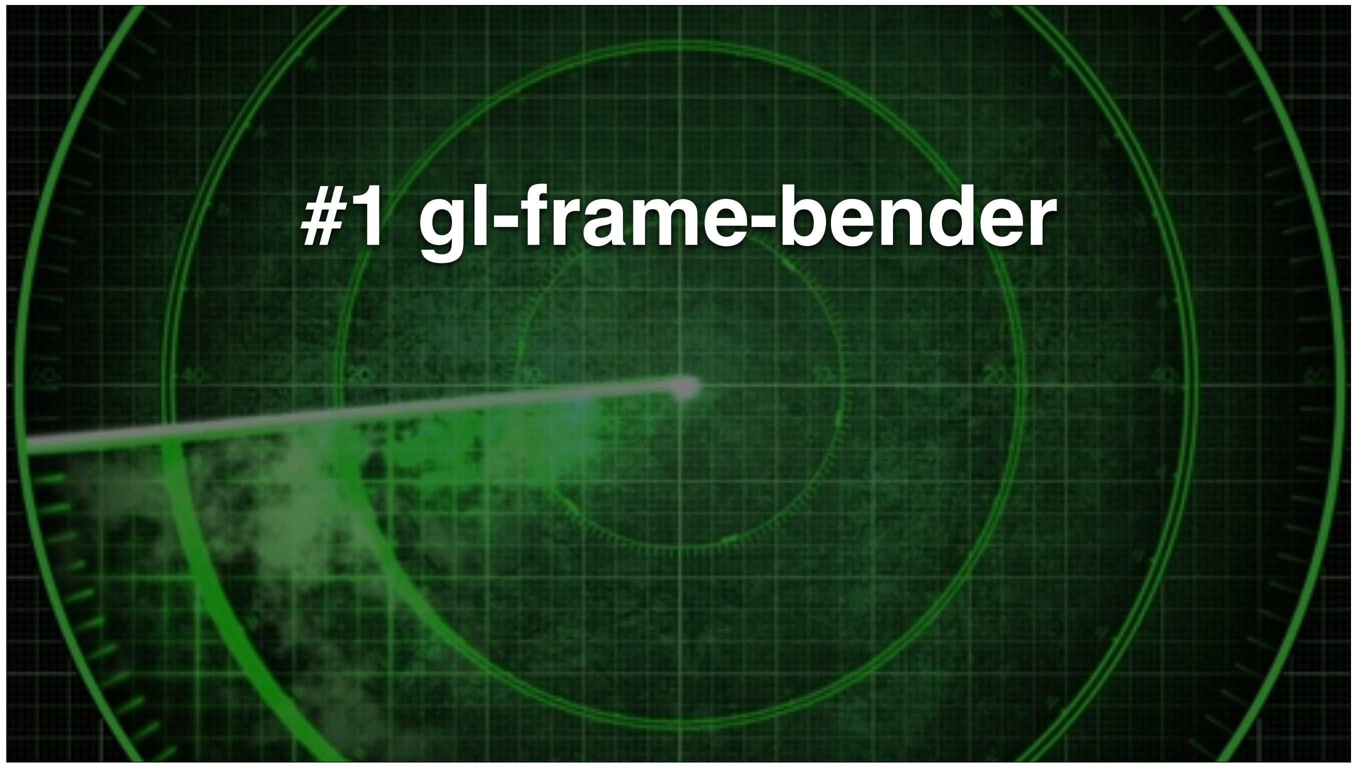
tools ON air

tools ON air

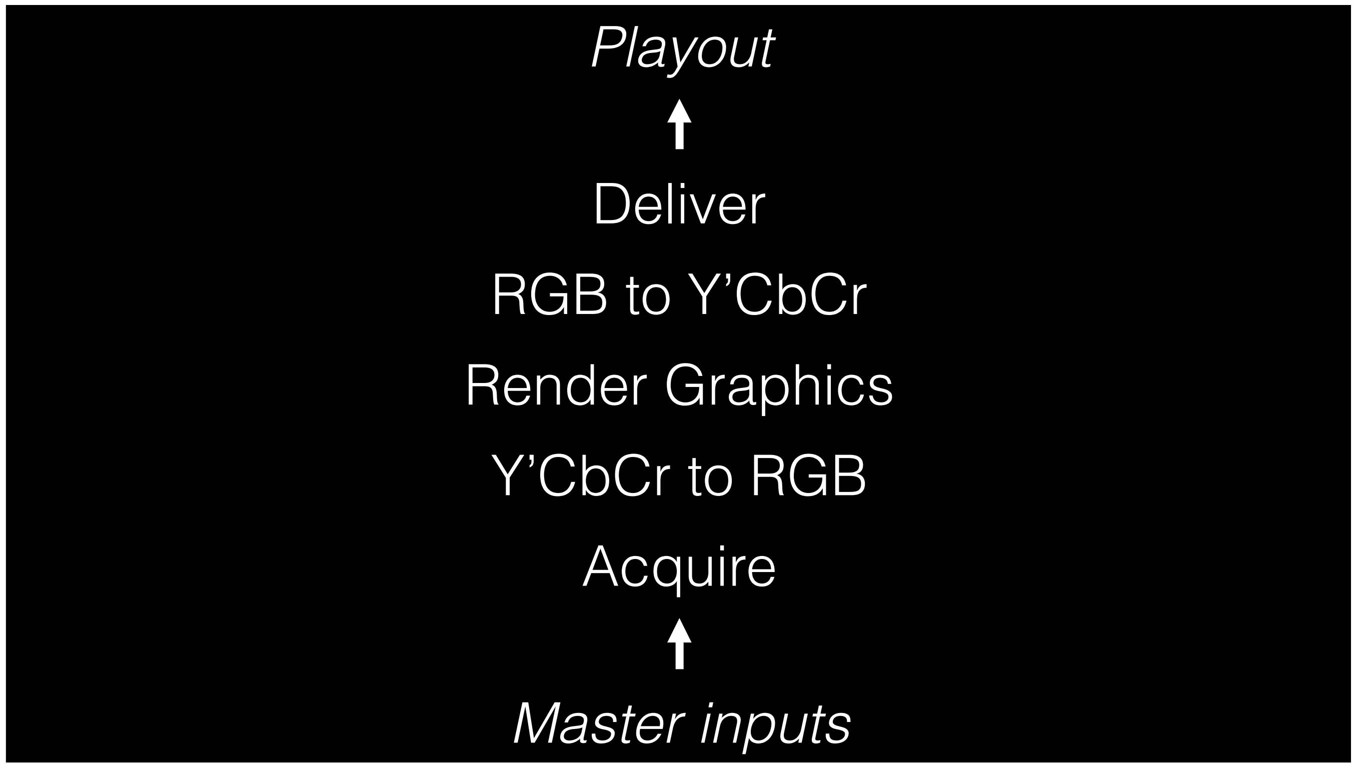
http://www.toolsonair.com

- ToolsOnAir is known for its Mac-based software solutions for broadcasting.
  - Capture, playout with real-time graphics
- Check out our webpage for more details

- Today I'd like to talk about two R&D projects that we have been working on in the past year



- The first one is a benchmarking framework for GPU-based video processing using OpenGL

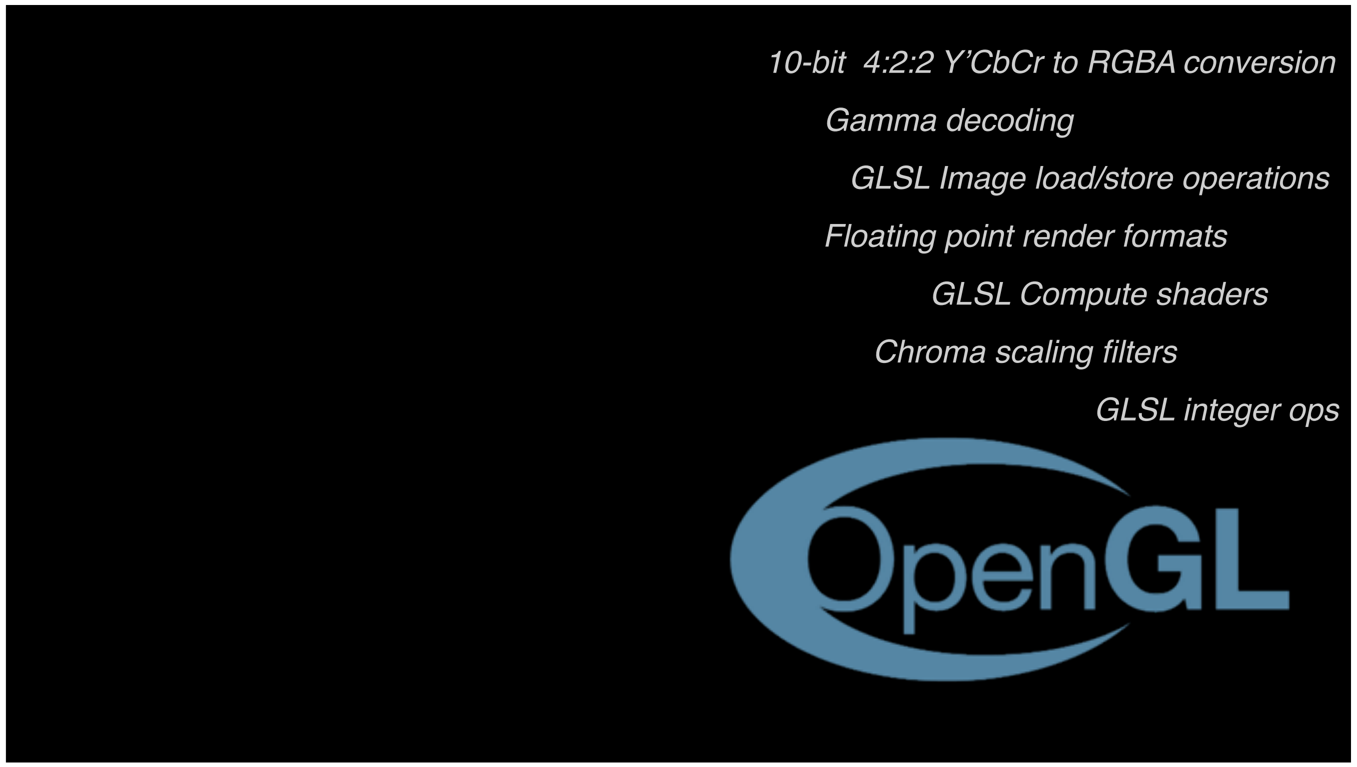


- The use-case is a simple playout pipeline…
  - takes input master streams
  - converts them from luma/chroma coding to RGB
  - renders graphics
  - then encode back to luma/chroma
  - and delivers the mixed output to the playout medium

- OpenGL is a good choice for implementing this, especially when graphics is going to be a focus

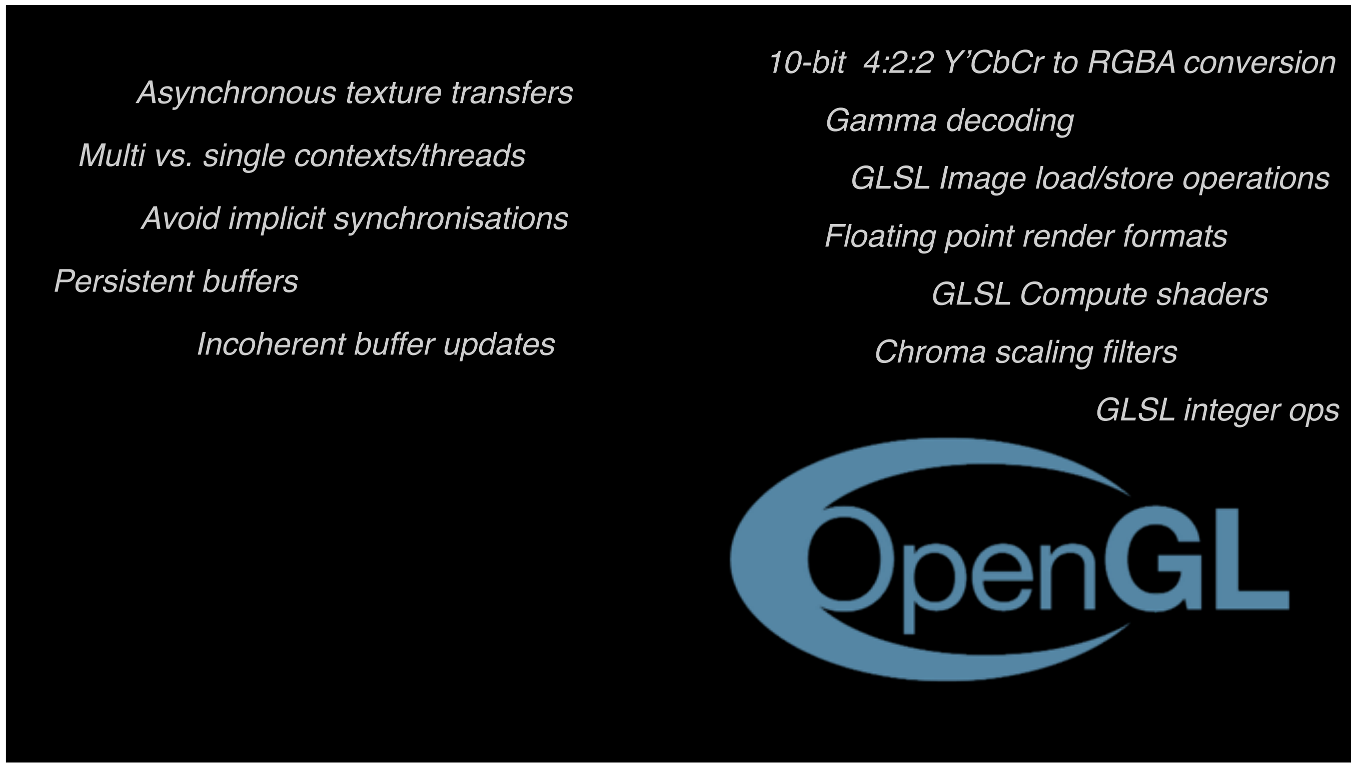
But OpenGL has become a very large set of APIs, even for our simple use case, there many different code paths for

- Video image processing algorithms
- Optimising frame transfer
- Application infrastructure & tools


10-bit 4:2:2 Y'CbCr to RGBA conversion
Gamma decoding
GLSL Image load/store operations
Floating point render formats
GLSL Compute shaders
Chroma scaling filters
GLSL integer ops
OpenGL

But OpenGL has become a very large set of APIs, even for our simple use case, there many different code paths for

- Video image processing algorithms
- Optimising frame transfer
- Application infrastructure & tools



*Asynchronous texture transfers*

*Multi vs. single contexts/threads*

*Avoid implicit synchronisations*

*Persistent buffers*

*Incoherent buffer updates*

*10-bit 4:2:2 Y'CbCr to RGBA conversion*

*Gamma decoding*

*GLSL Image load/store operations*

*Floating point render formats*

*GLSL Compute shaders*

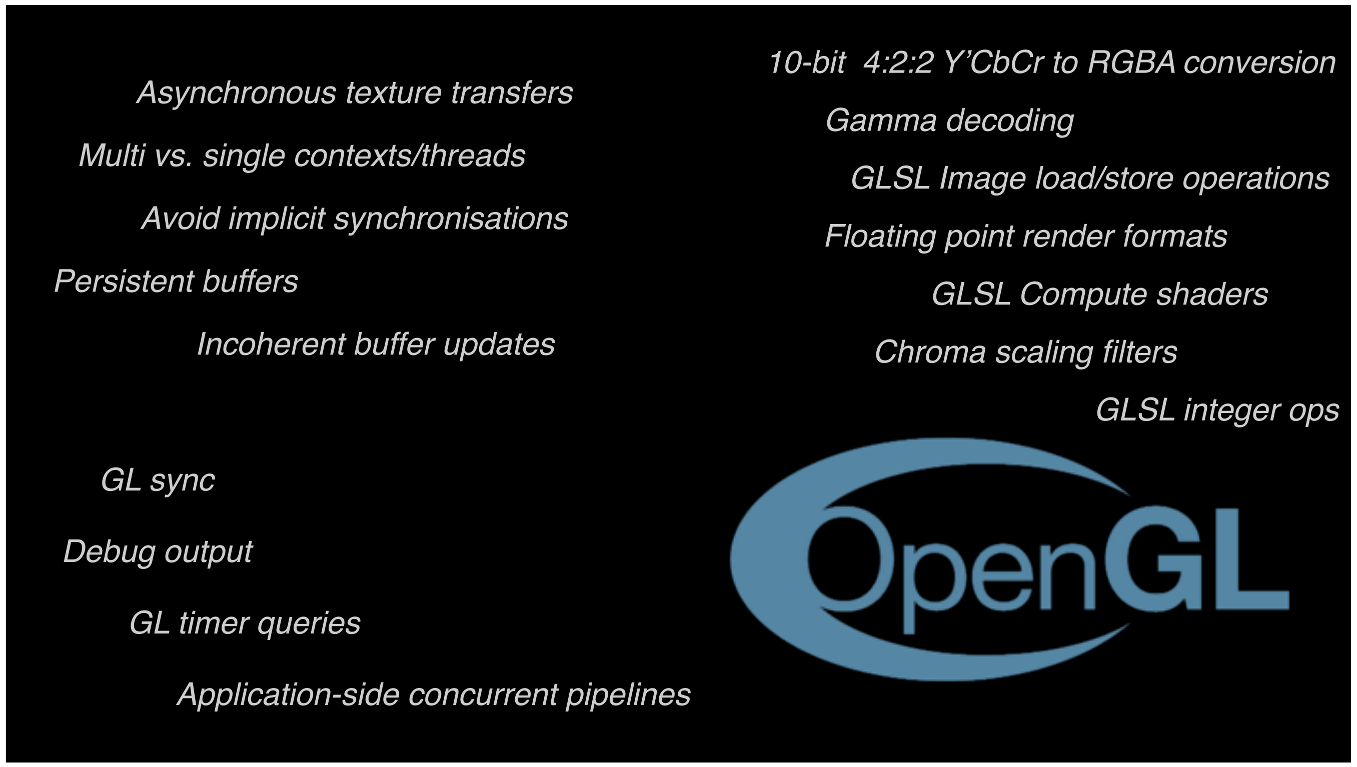
*Chroma scaling filters*

*GLSL integer ops*

OpenGL

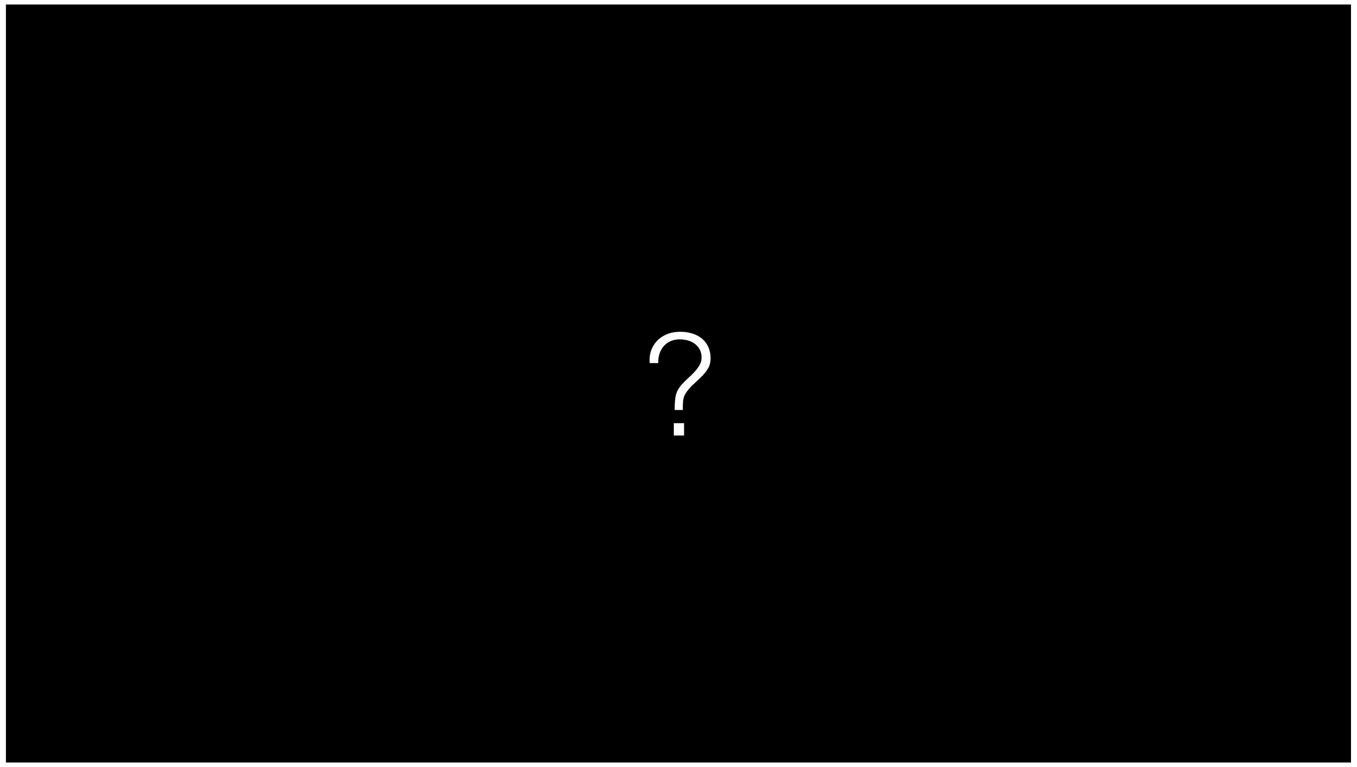
But OpenGL has become a very large set of APIs, even for our simple use case, there many different code paths for

- Video image processing algorithms
- Optimising frame transfer
- Application infrastructure & tools


Asynchronous texture transfers
Multi vs. single contexts/threads
Avoid implicit synchronisations
Persistent buffers
Incoherent buffer updates
GL sync
Debug output
GL timer queries
Application-side concurrent pipelines
10-bit 4:2:2 Y'CbCr to RGBA conversion
Gamma decoding
GLSL Image load/store operations
Floating point render formats
GLSL Compute shaders
Chroma scaling filters
GLSL integer ops
OpenGL

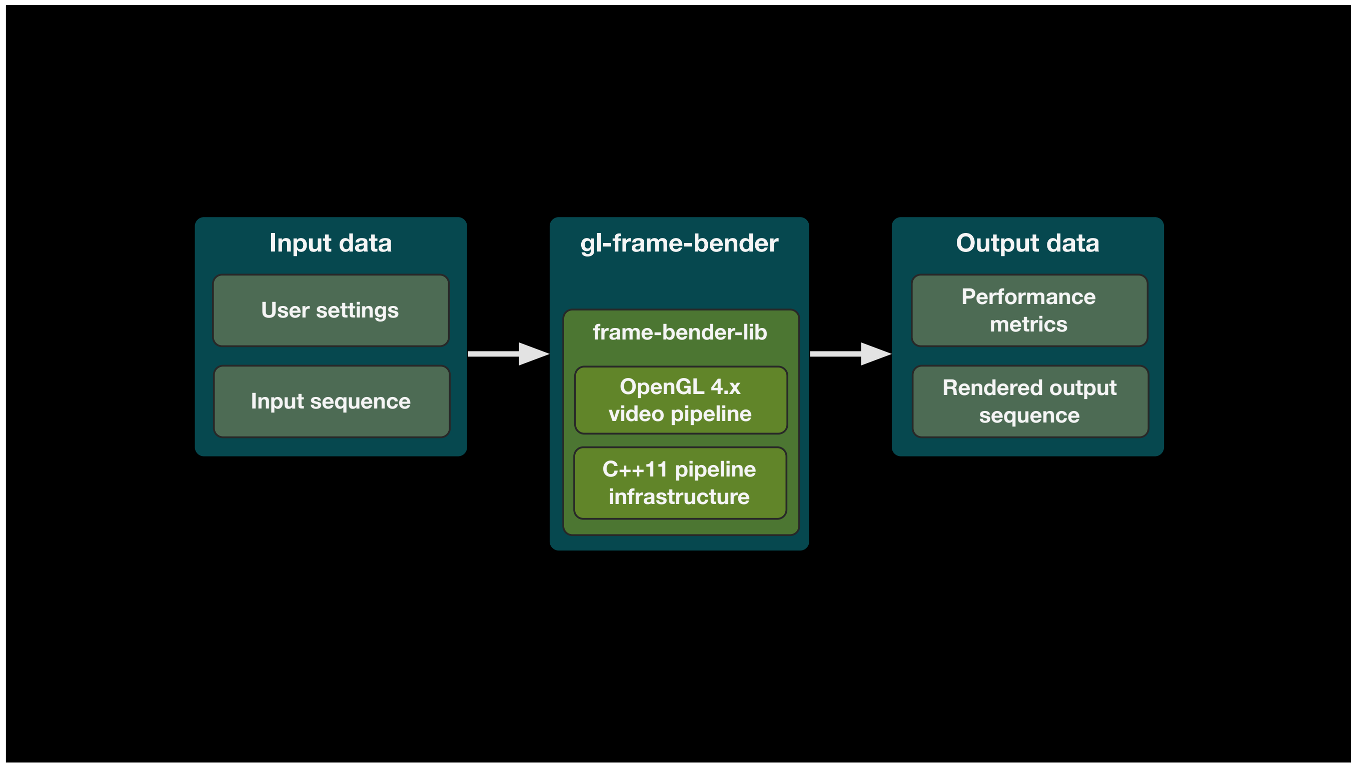
But OpenGL has become a very large set of APIs, even for our simple use case, there many different code paths for

- Video image processing algorithms
- Optimising frame transfer
- Application infrastructure & tools

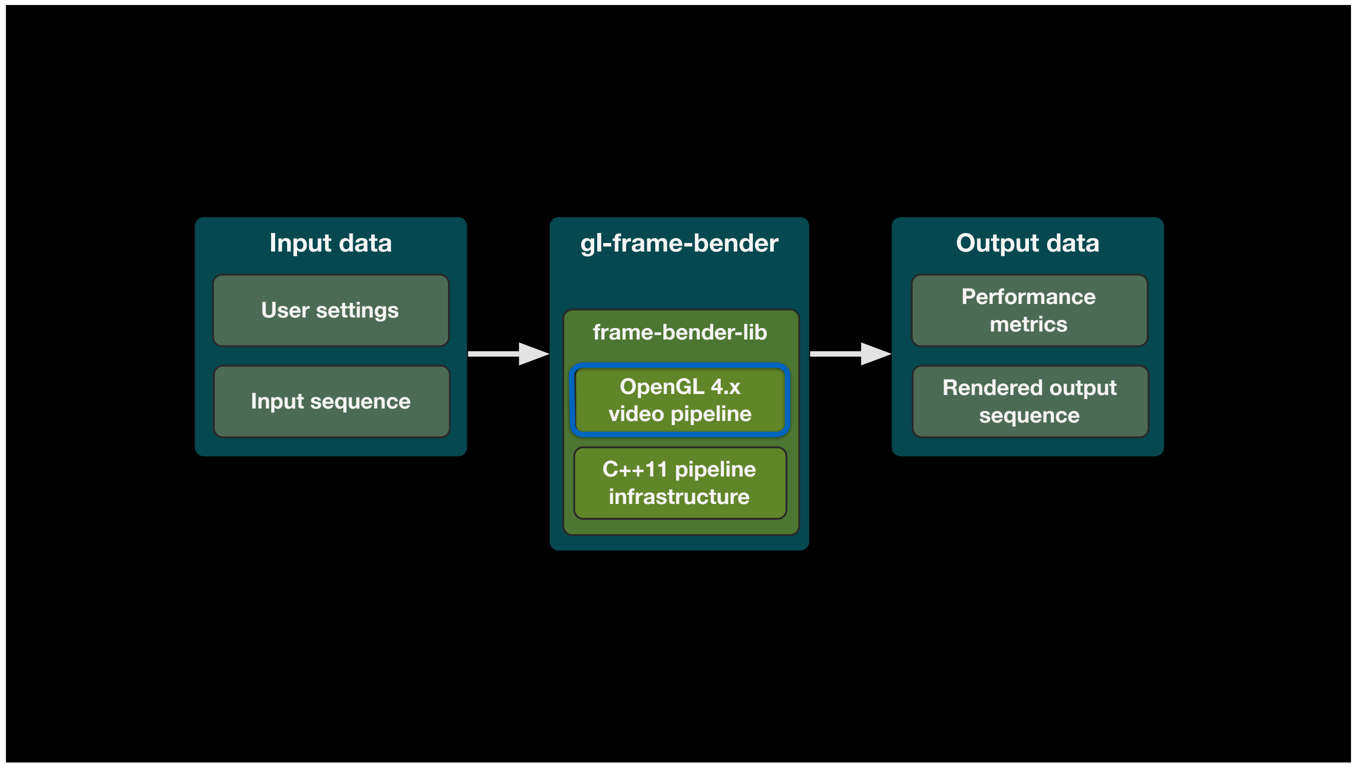


?

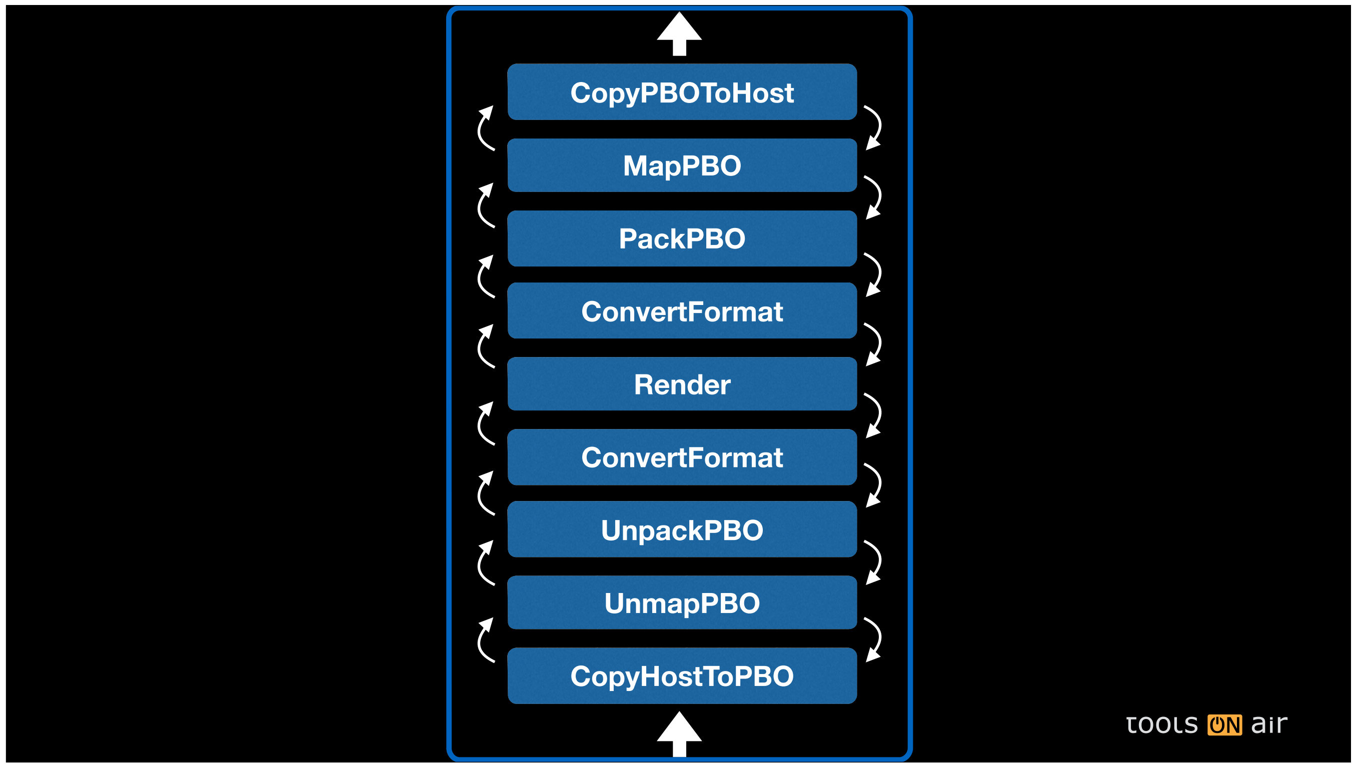
Well, how do we know which works best?



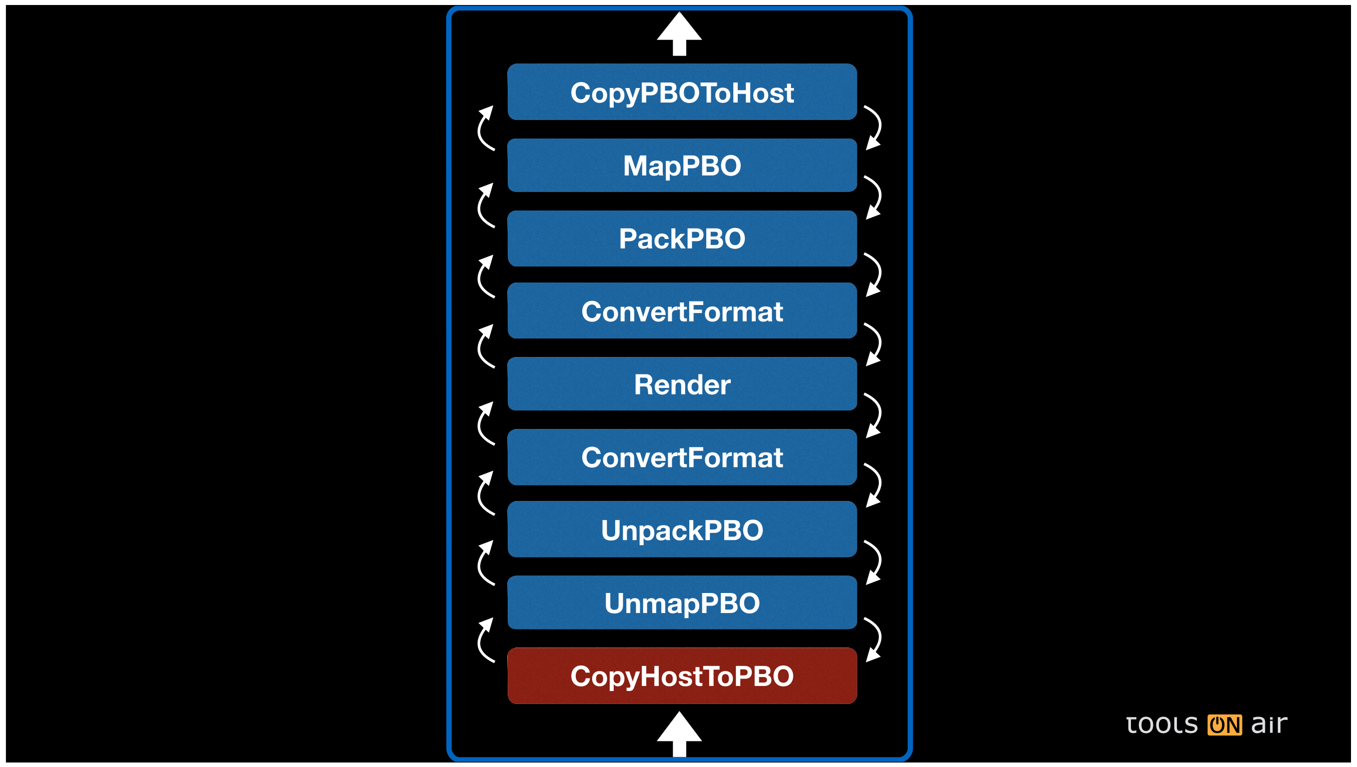
- To answer this, we've created a tool called "gl-frame-bender"
- It's a throughput-oriented benchmarking framework that runs a simple playout-like video processing pipeline.
- It takes a set of standard 10-bit test sequences as input
- Allows the user to choose between various implementation variants and records performance metrics into a file.



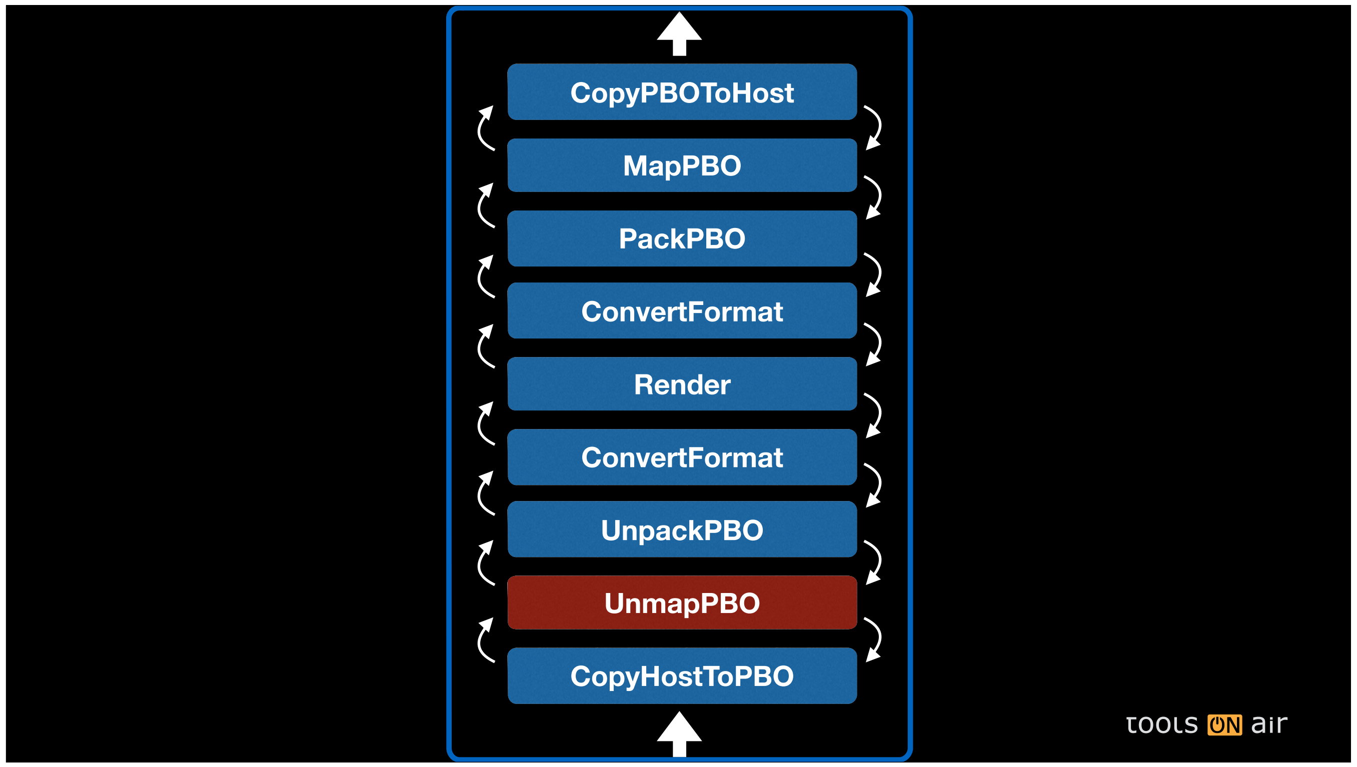
- To answer this, we’ve created a tool called “gl-frame-bender”
- It’s a throughput-oriented benchmarking framework that runs a simple playout-like video processing pipeline.
- It takes a set of standard 10-bit test sequences as input
- Allows the user to choose between various implementation variants and records performance metrics into a file.



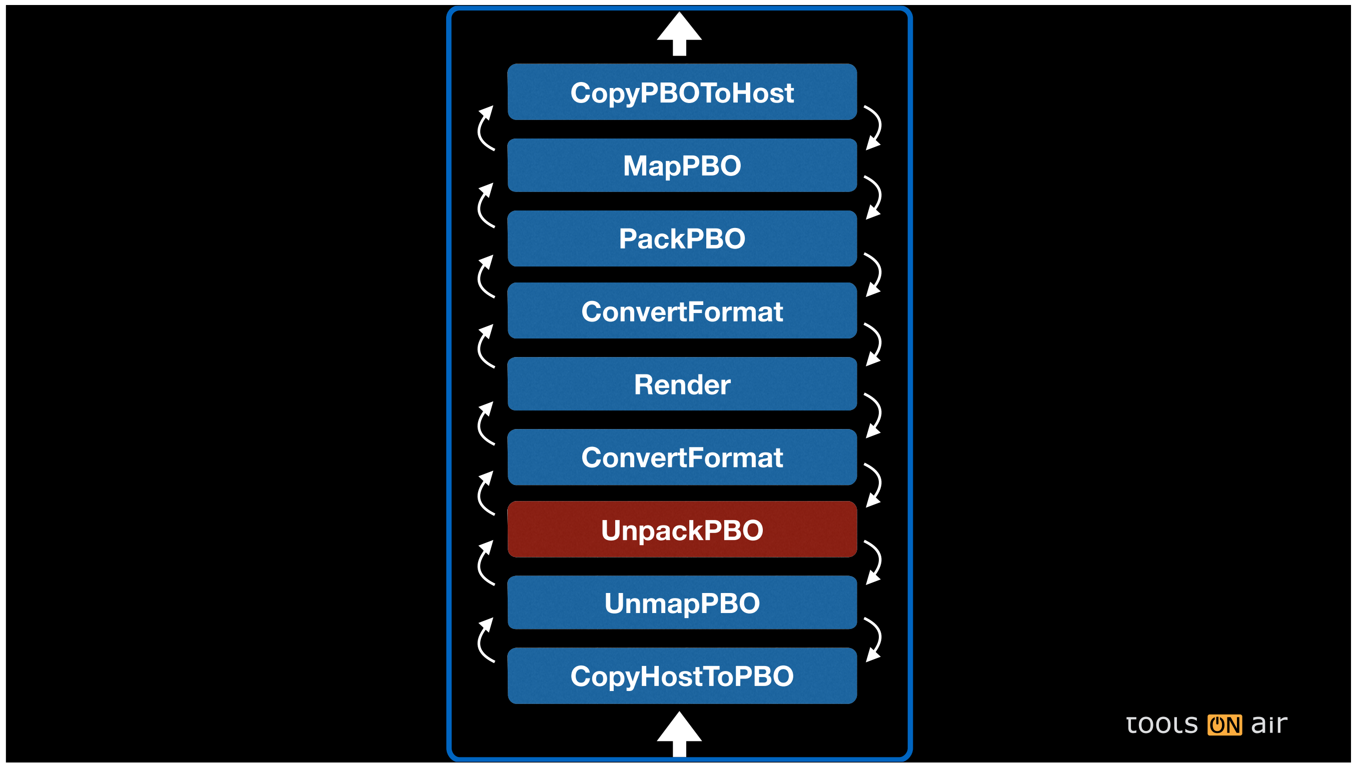
- The framework uses a software pipeline pattern. Individual "stages" transport data down- and upstream via thread-safe queues
- Potentially, each stage could be executed on its own thread
- Speaking in OpenGL terms, the first stage copies host-memory into memory mapped by a pixel buffer object
- The next stage unmaps the PBO, which is then "unpacked" (uploaded) to a GL texture.
- This texture, still in video luma/chroma color space is then converted into **linear RGB**, where simple graphics are then rendered.
- Then we reverse everything
- The size of the queues and their threading are are freely configurable by the user
- Each stage is able to record their execution start and end times using CPU and GPU clocks.
- These records can be written into a compressed file after the execution of the test program
- Now let's look at some optimisations we have implemented for running this pipeline



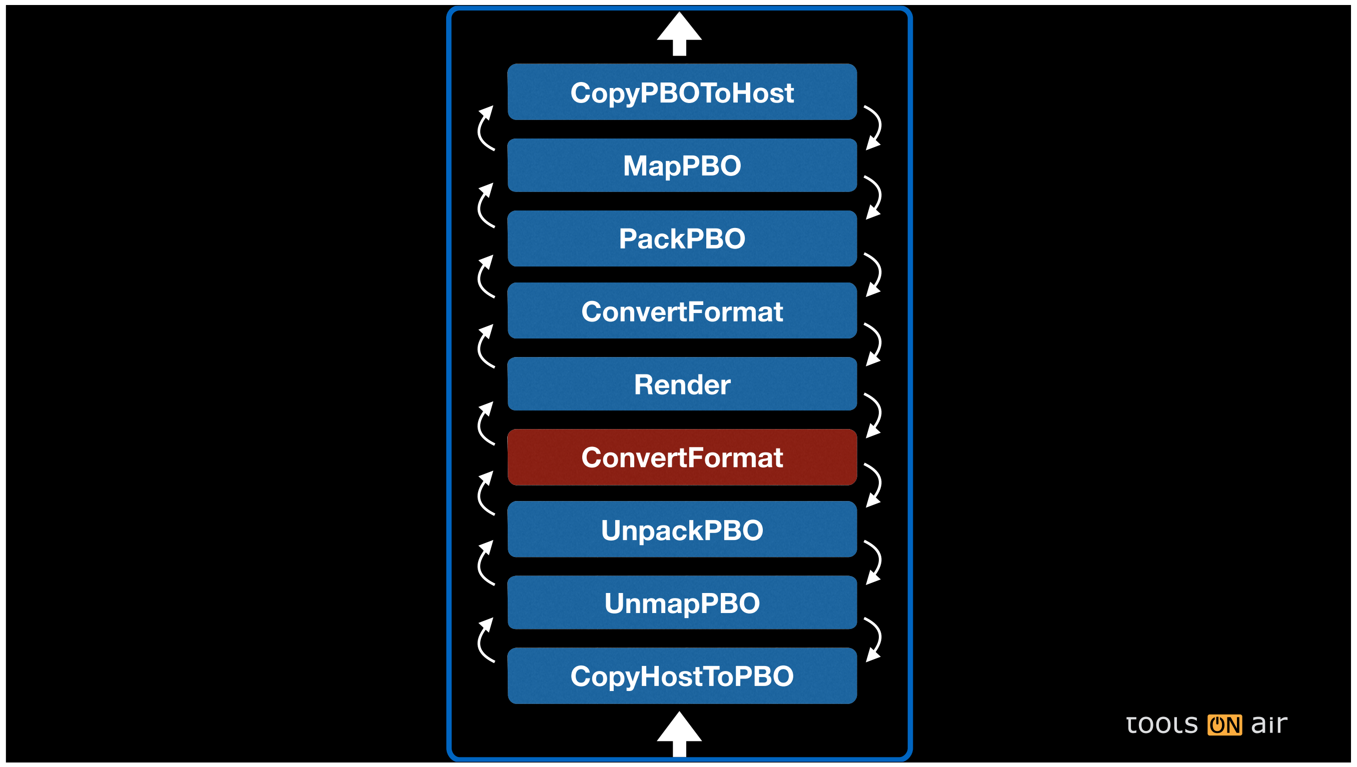
- The framework uses a software pipeline pattern. Individual "stages" transport data down- and upstream via thread-safe queues
- Potentially, each stage could be executed on its own thread
- Speaking in OpenGL terms, the first stage copies host-memory into memory mapped by a pixel buffer object
- The next stage unmaps the PBO, which is then "unpacked" (uploaded) to a GL texture.
- This texture, still in video luma/chroma color space is then converted into **linear RGB**, where simple graphics are then rendered.
- Then we reverse everything
- The size of the queues and their threading are are freely configurable by the user
- Each stage is able to record their execution start and end times using CPU and GPU clocks.
- These records can be written into a compressed file after the execution of the test program
- Now let's look at some optimisations we have implemented for running this pipeline



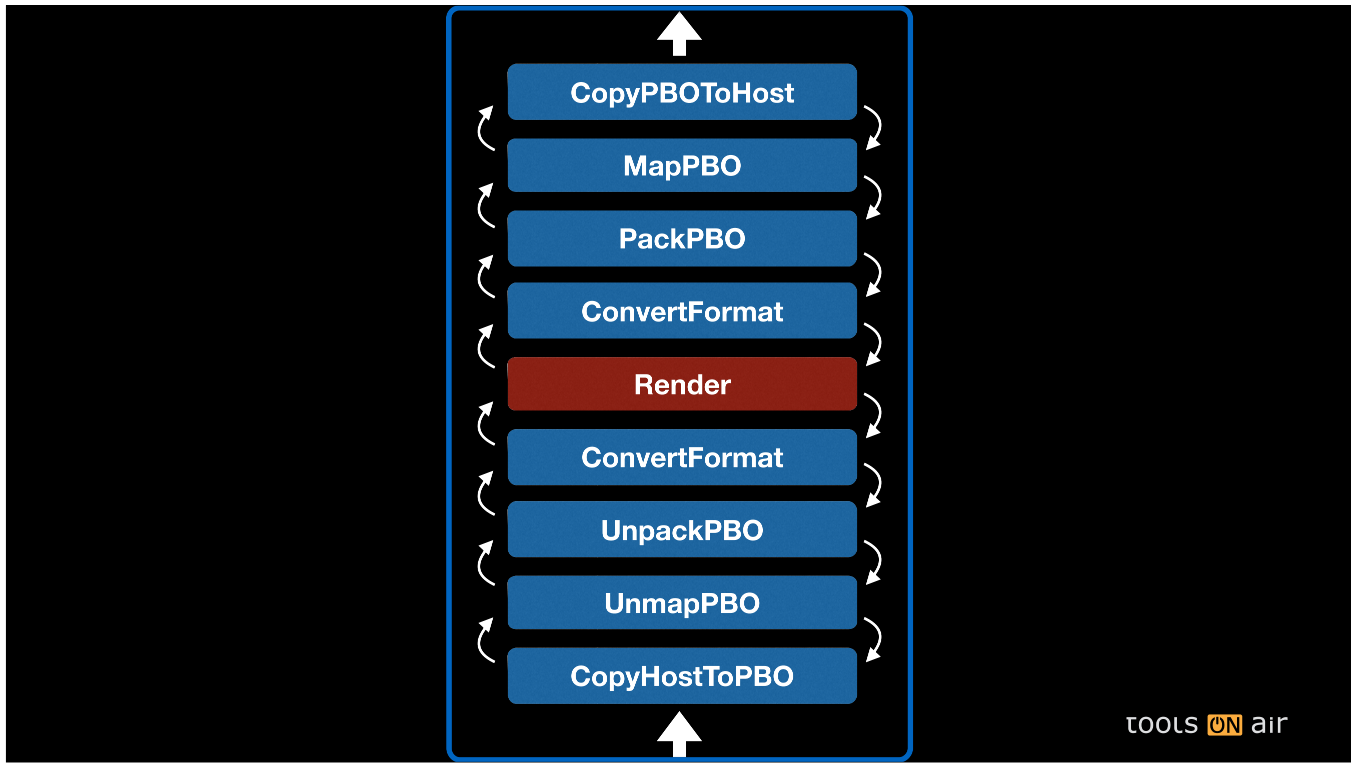
- The framework uses a software pipeline pattern. Individual "stages" transport data down- and upstream via thread-safe queues
- Potentially, each stage could be executed on its own thread
- Speaking in OpenGL terms, the first stage copies host-memory into memory mapped by a pixel buffer object
- The next stage unmaps the PBO, which is then "unpacked" (uploaded) to a GL texture.
- This texture, still in video luma/chroma color space is then converted into **linear RGB**, where simple graphics are then rendered.
- Then we reverse everything
- The size of the queues and their threading are are freely configurable by the user
- Each stage is able to record their execution start and end times using CPU and GPU clocks.
- These records can be written into a compressed file after the execution of the test program
- Now let's look at some optimisations we have implemented for running this pipeline



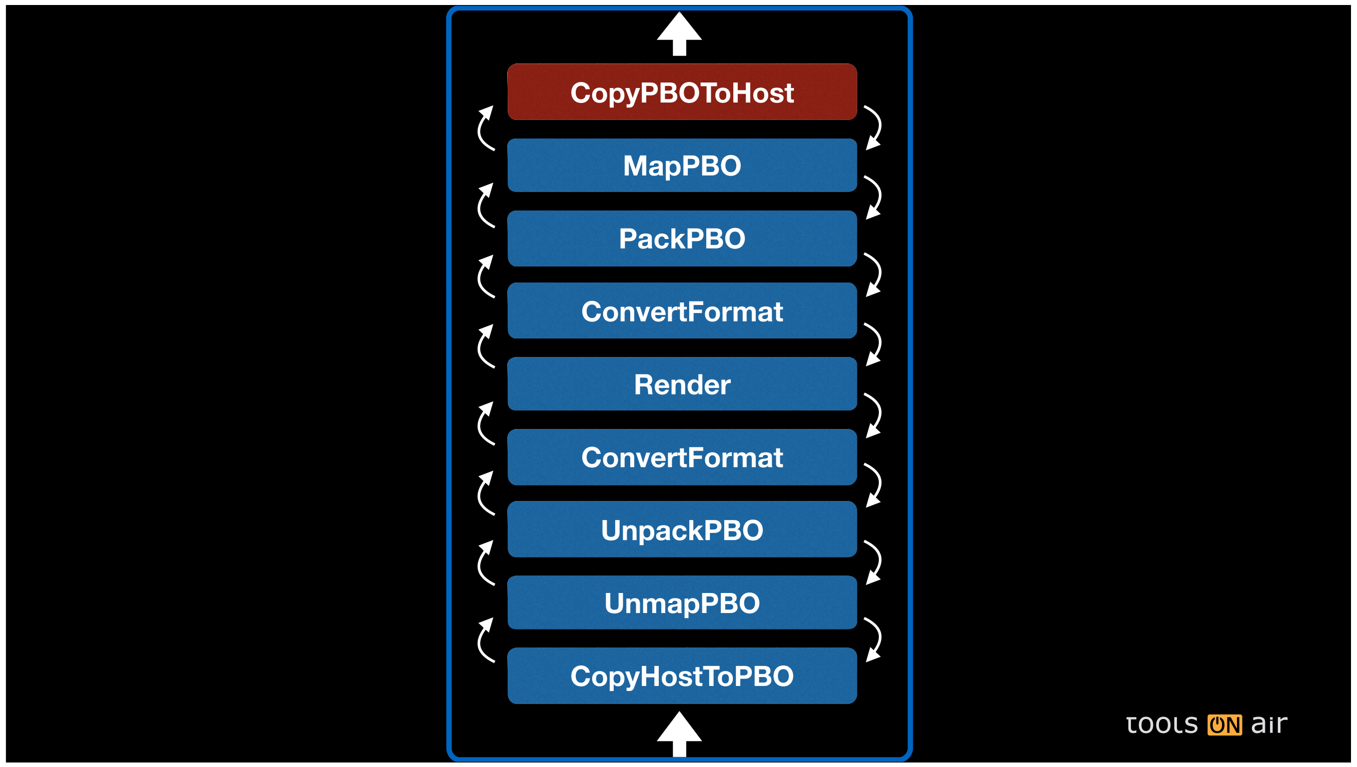
- The framework uses a software pipeline pattern. Individual “stages” transport data down- and upstream via thread-safe queues
- Potentially, each stage could be executed on its own thread
- Speaking in OpenGL terms, the first stage copies host-memory into memory mapped by a pixel buffer object
- The next stage unmaps the PBO, which is then “unpacked” (uploaded) to a GL texture.
- This texture, still in video luma/chroma color space is then converted into **linear RGB**, where simple graphics are then rendered.
- Then we reverse everything
- The size of the queues and their threading are are freely configurable by the user
- Each stage is able to record their execution start and end times using CPU and GPU clocks.
- These records can be written into a compressed file after the execution of the test program
- Now let’s look at some optimisations we have implemented for running this pipeline



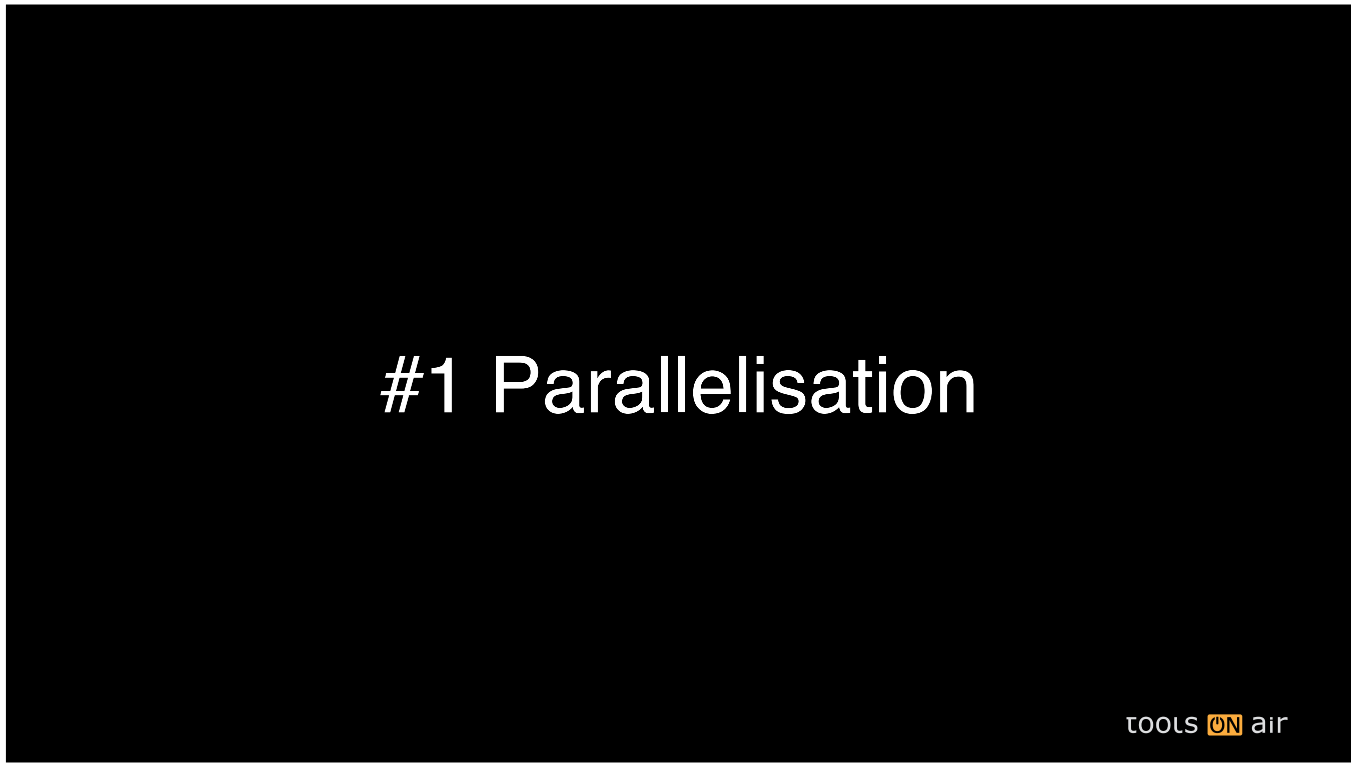
- The framework uses a software pipeline pattern. Individual "stages" transport data down- and upstream via thread-safe queues
- Potentially, each stage could be executed on its own thread
- Speaking in OpenGL terms, the first stage copies host-memory into memory mapped by a pixel buffer object
- The next stage unmaps the PBO, which is then "unpacked" (uploaded) to a GL texture.
- This texture, still in video luma/chroma color space is then converted into **linear RGB**, where simple graphics are then rendered.
- Then we reverse everything
- The size of the queues and their threading are are freely configurable by the user
- Each stage is able to record their execution start and end times using CPU and GPU clocks.
- These records can be written into a compressed file after the execution of the test program
- Now let's look at some optimisations we have implemented for running this pipeline



- The framework uses a software pipeline pattern. Individual “stages” transport data down- and upstream via thread-safe queues
- Potentially, each stage could be executed on its own thread
- Speaking in OpenGL terms, the first stage copies host-memory into memory mapped by a pixel buffer object
- The next stage unmaps the PBO, which is then “unpacked” (uploaded) to a GL texture.
- This texture, still in video luma/chroma color space is then converted into **linear RGB**, where simple graphics are then rendered.
- Then we reverse everything
- The size of the queues and their threading are are freely configurable by the user
- Each stage is able to record their execution start and end times using CPU and GPU clocks.
- These records can be written into a compressed file after the execution of the test program
- Now let’s look at some optimisations we have implemented for running this pipeline



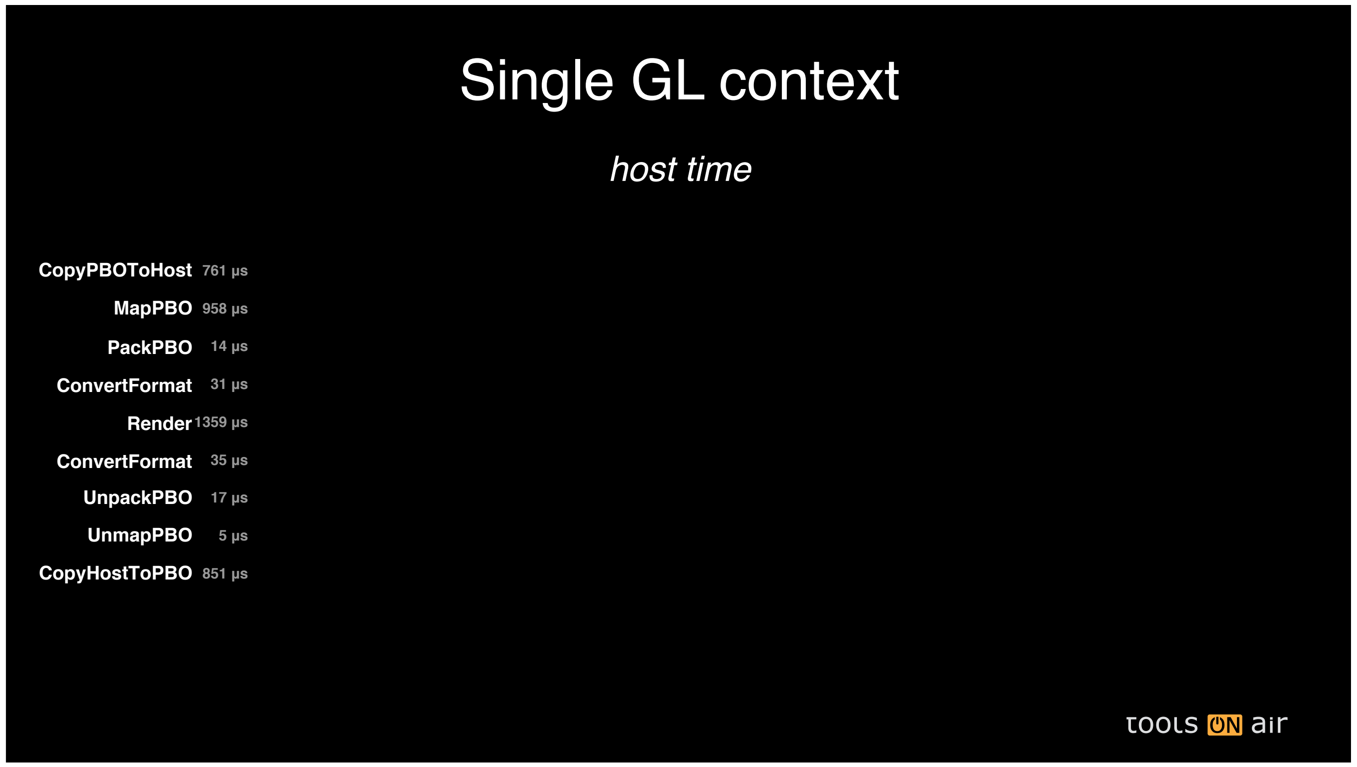
- The framework uses a software pipeline pattern. Individual "stages" transport data down- and upstream via thread-safe queues
- Potentially, each stage could be executed on its own thread
- Speaking in OpenGL terms, the first stage copies host-memory into memory mapped by a pixel buffer object
- The next stage unmaps the PBO, which is then "unpacked" (uploaded) to a GL texture.
- This texture, still in video luma/chroma color space is then converted into **linear RGB**, where simple graphics are then rendered.
- Then we reverse everything
- The size of the queues and their threading are are freely configurable by the user
- Each stage is able to record their execution start and end times using CPU and GPU clocks.
- These records can be written into a compressed file after the execution of the test program
- Now let's look at some optimisations we have implemented for running this pipeline



## #1 Parallelisation

tools ON air

- The number one optimisation that we would like achieve is to parallelise as much as possible of the pipeline



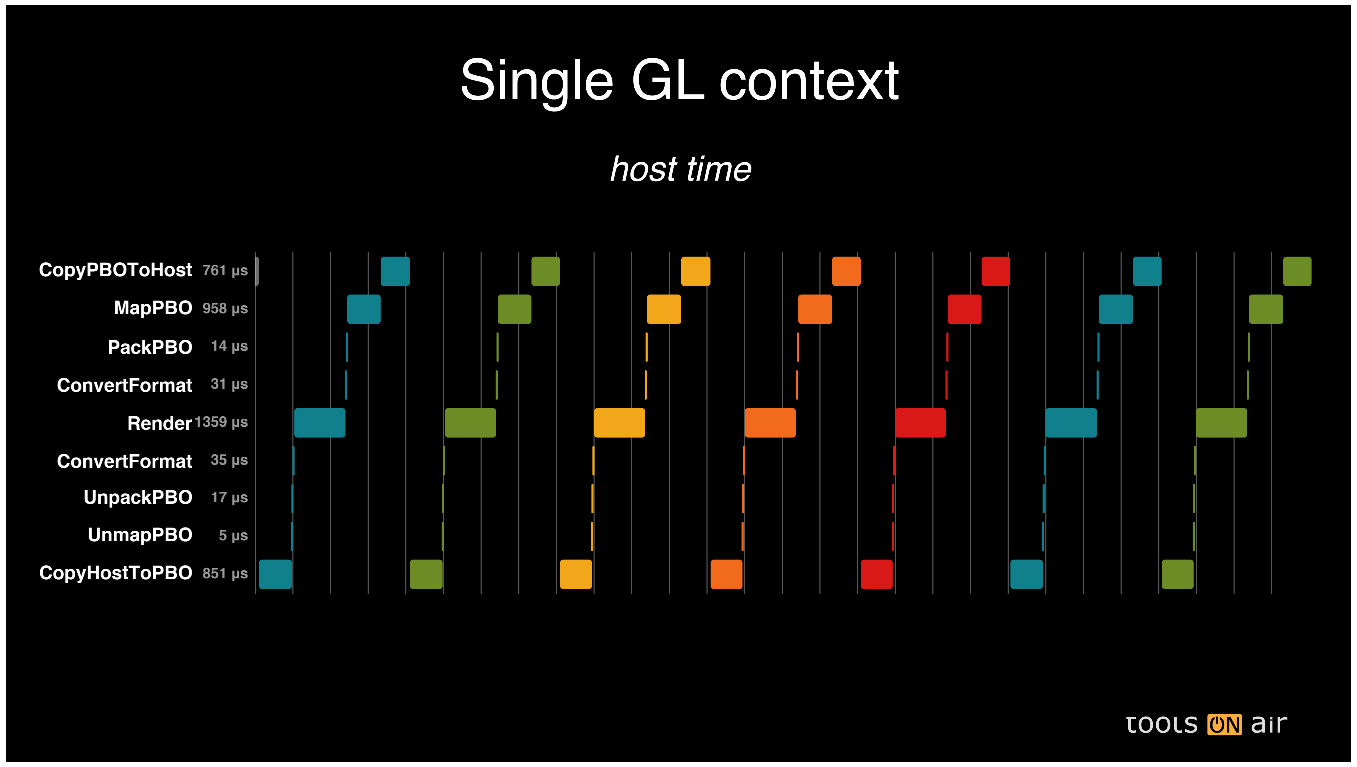
## Single GL context

*host time*

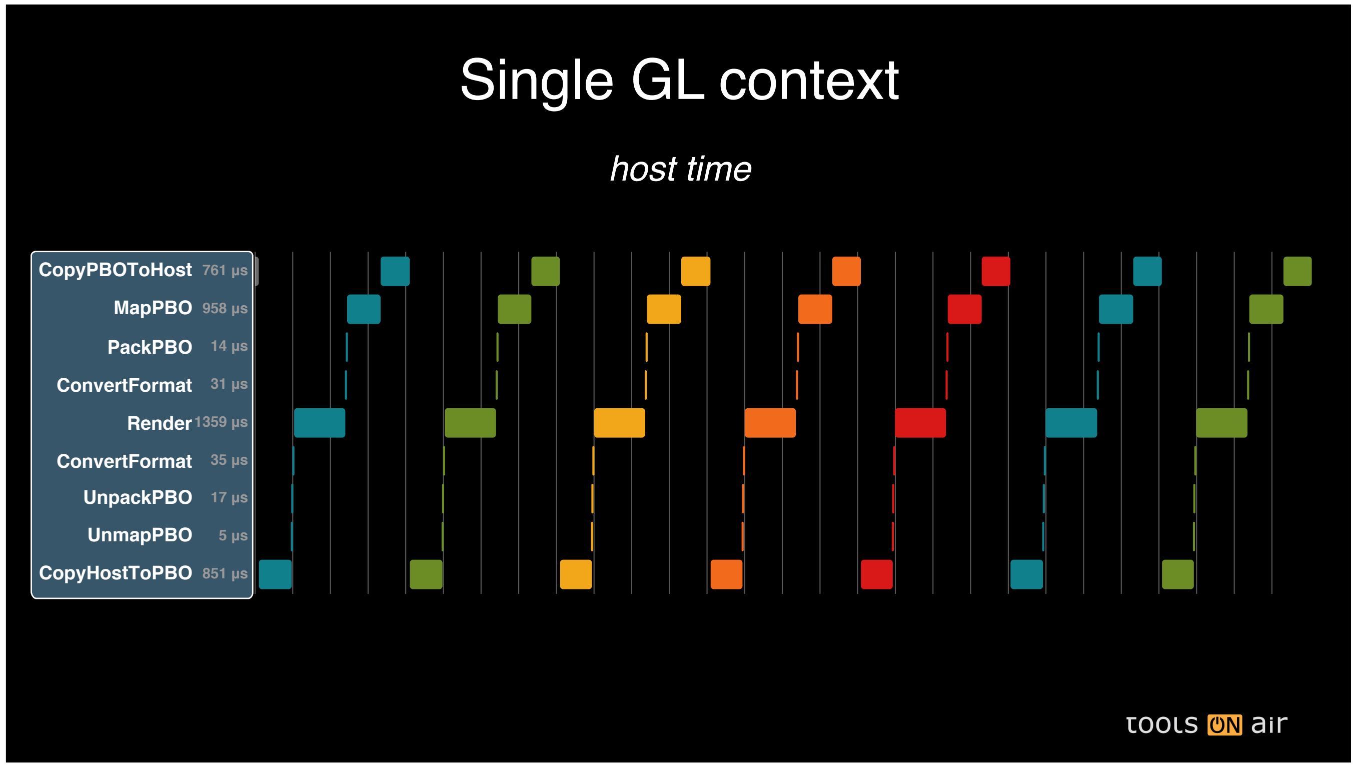
| | |
|---|---|
| CopyPBOToHost | 761 µs |
| MapPBO | 958 µs |
| PackPBO | 14 µs |
| ConvertFormat | 31 µs |
| Render | 1359 µs |
| ConvertFormat | 35 µs |
| UnpackPBO | 17 µs |
| UnmapPBO | 5 µs |
| CopyHostToPBO | 851 µs |

tools ON air

- We will now look at an execution graph of the CPU-side execution graph for several frames. Each frame has a different color.
- In this first case, a single thread and GL context was used to execute all stages
- We can see that there is no overlap in the execution



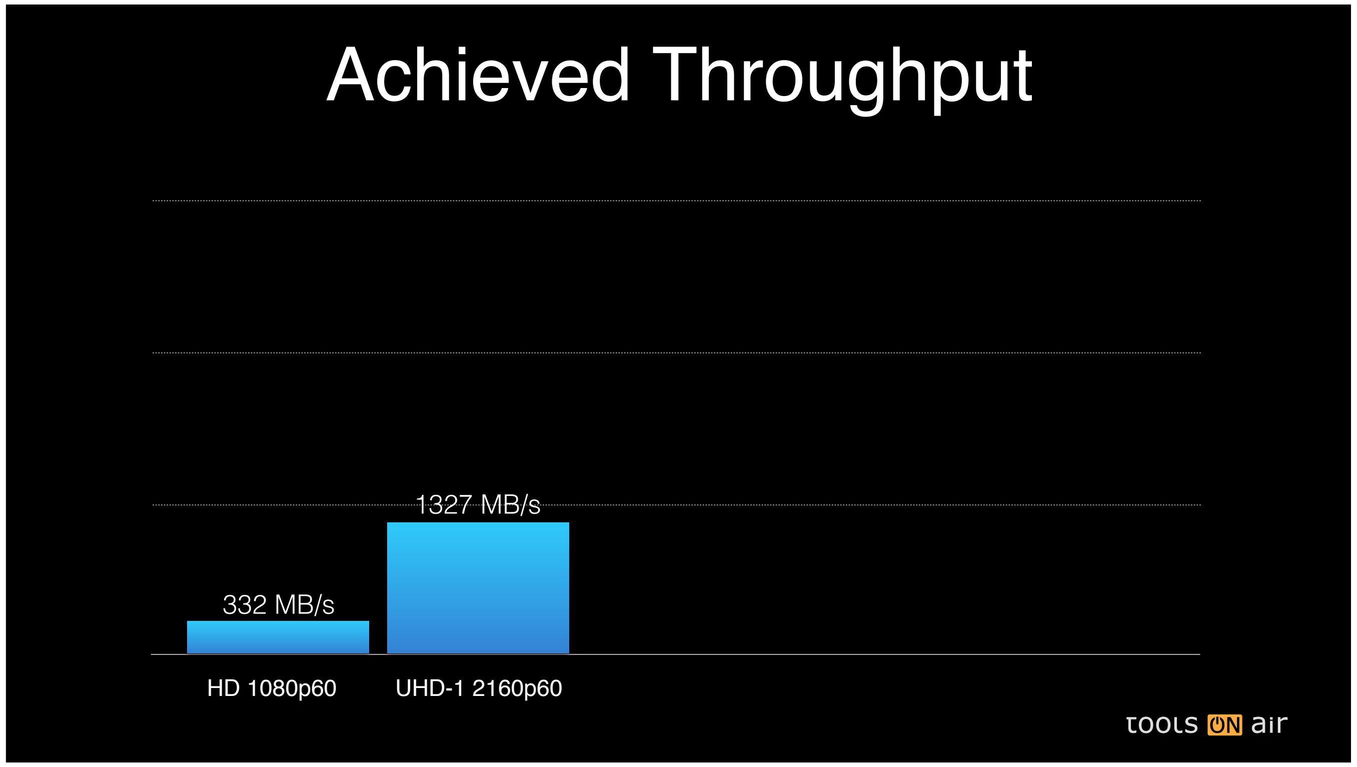
- We will now look at an execution graph of the CPU-side execution graph for several frames. Each frame has a different color.
- In this first case, a single thread and GL context was used to execute all stages
- We can see that there is no overlap in the execution



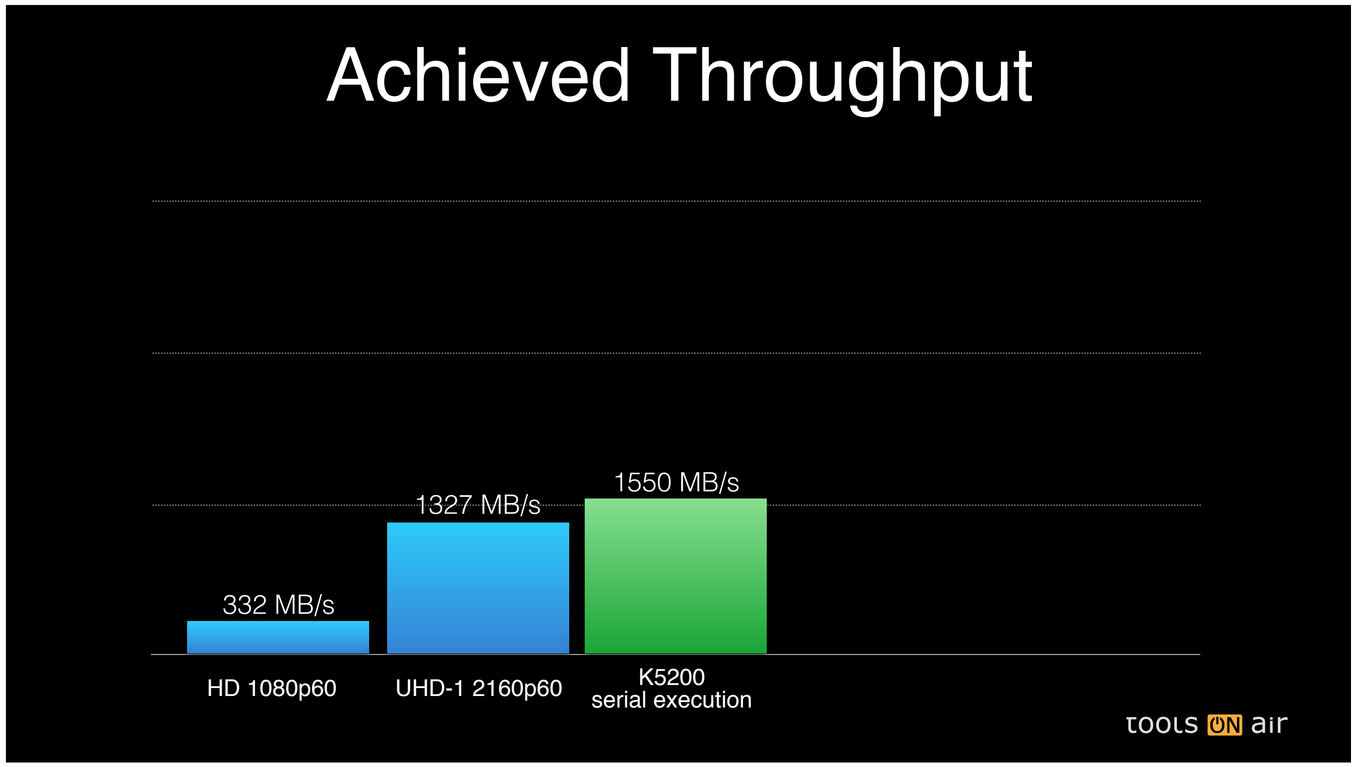
- We will now look at an execution graph of the CPU-side execution graph for several frames. Each frame has a different color.
- In this first case, a single thread and GL context was used to execute all stages
- We can see that there is no overlap in the execution

## Achieved Throughput

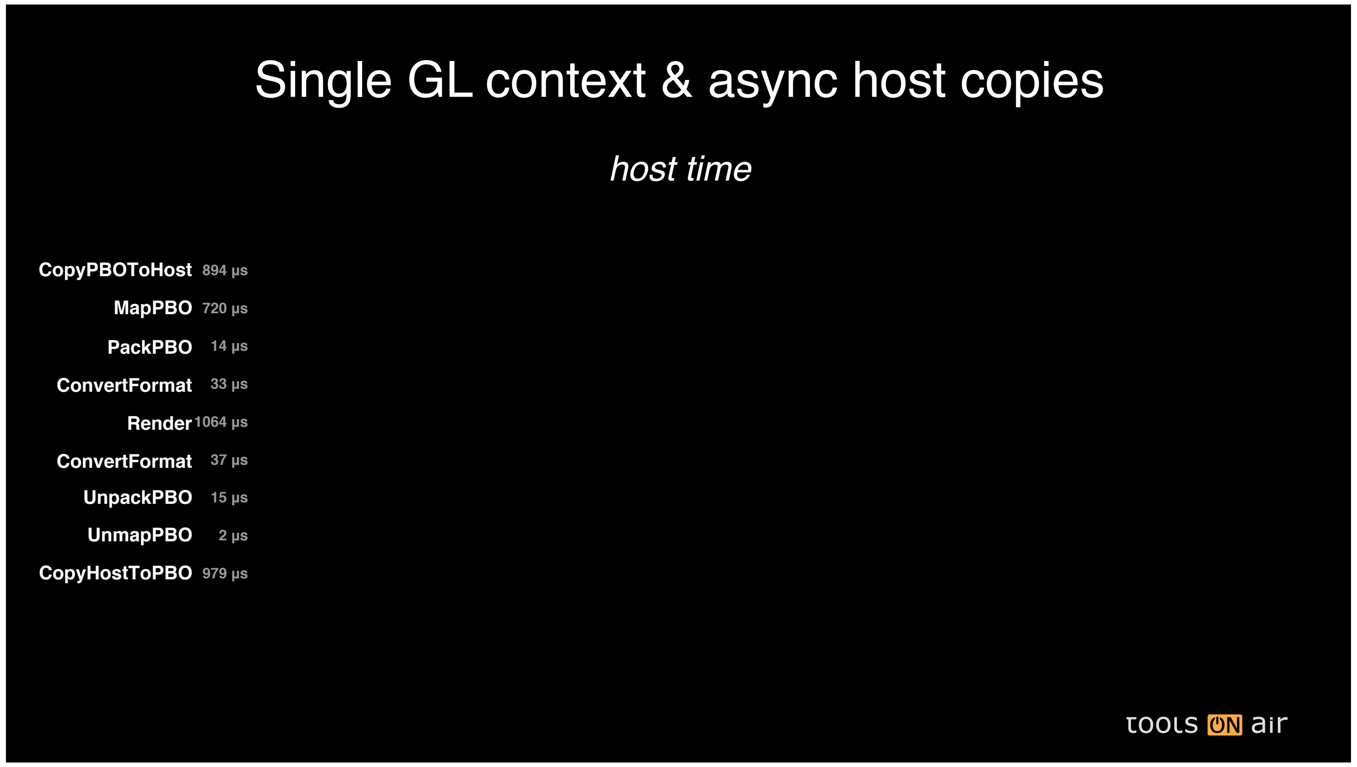
- Let’s look at the real throughput with all our rendering operations and up/downloads in place we were able to achieve.



- Let’s look at the real throughput with all our rendering operations and up/downloads in place we were able to achieve.



- Let’s look at the real throughput with all our rendering operations and up/downloads in place we were able to achieve.

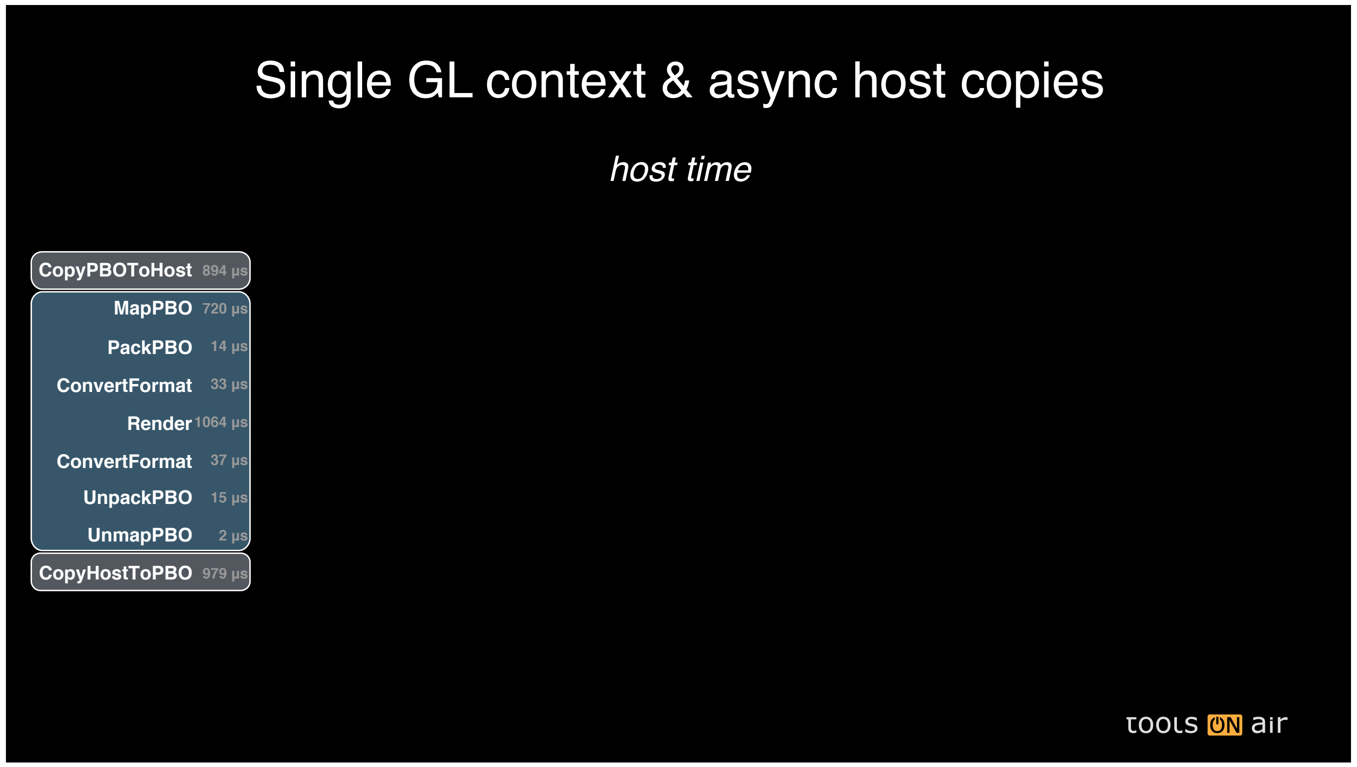


## Single GL context & async host copies

*host time*

| | |
|---|---|
| CopyPBOToHost | 894 µs |
| MapPBO | 720 µs |
| PackPBO | 14 µs |
| ConvertFormat | 33 µs |
| Render | 1064 µs |
| ConvertFormat | 37 µs |
| UnpackPBO | 15 µs |
| UnmapPBO | 2 µs |
| CopyHostToPBO | 979 µs |

- As a next step, we offload the copying from/into host-side frames to a separate thread, which already doubles the performance as some CPU-side execution is now done concurrently
- By using only CPU timers, this graph only tells us where the GL API is blocking us, but it doesn't show the time actually spend by the GPU



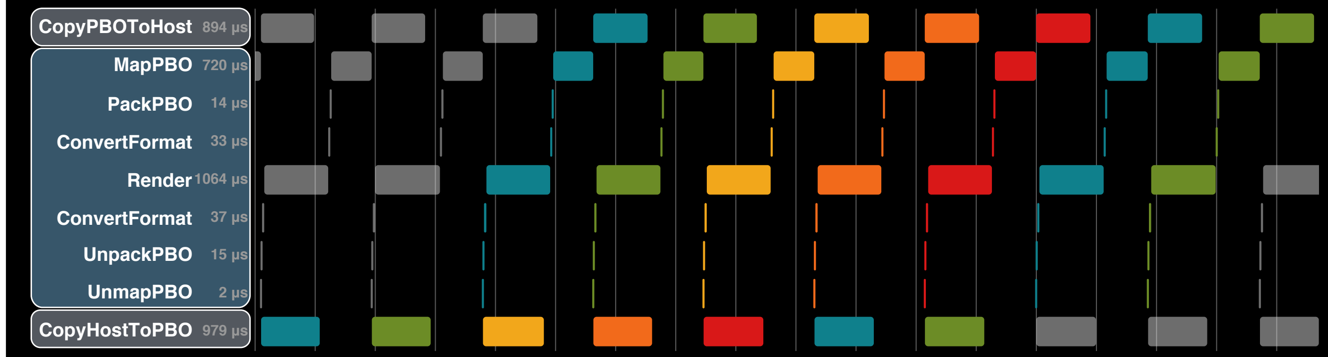
## Single GL context & async host copies

*host time*

- As a next step, we offload the copying from/into host-side frames to a separate thread, which already doubles the performance as some CPU-side execution is now done concurrently
- By using only CPU timers, this graph only tells us where the GL API is blocking us, but it doesn't show the time actually spend by the GPU

## Single GL context & async host copies

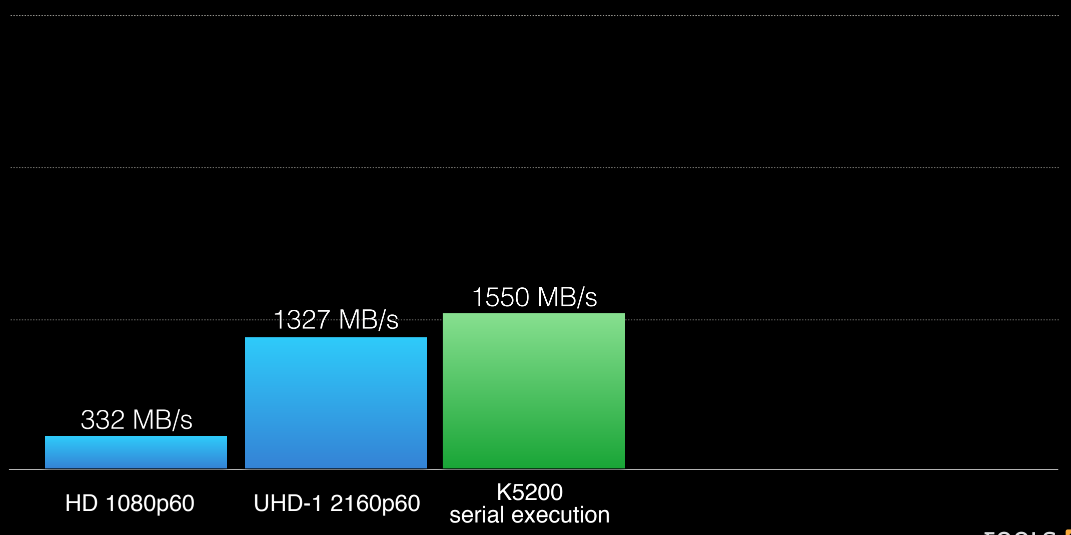
*host time*



tools ON air

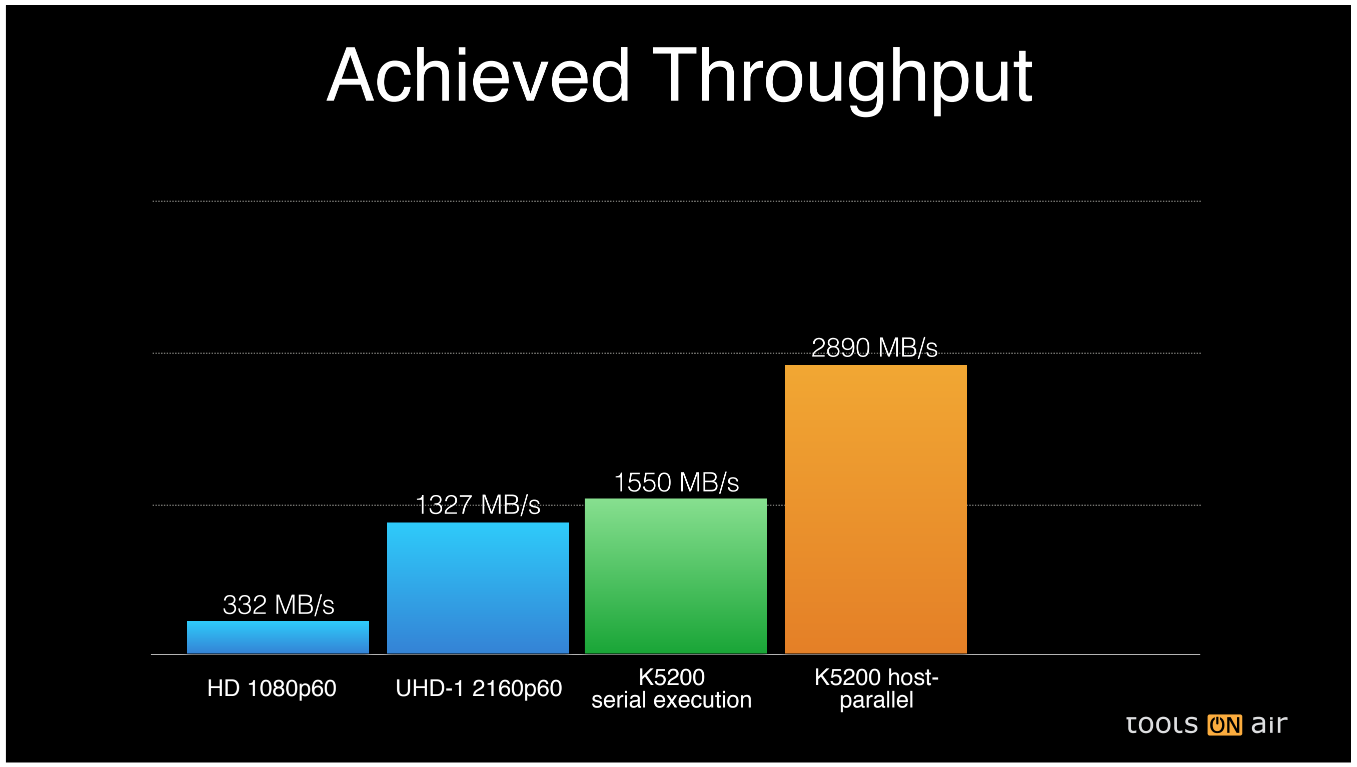
- As a next step, we offload the copying from/into host-side frames to a separate thread, which already doubles the performance as some CPU-side execution is now done concurrently
- By using only CPU timers, this graph only tells us where the GL API is blocking us, but it doesn't show the time actually spend by the GPU

## Achieved Throughput

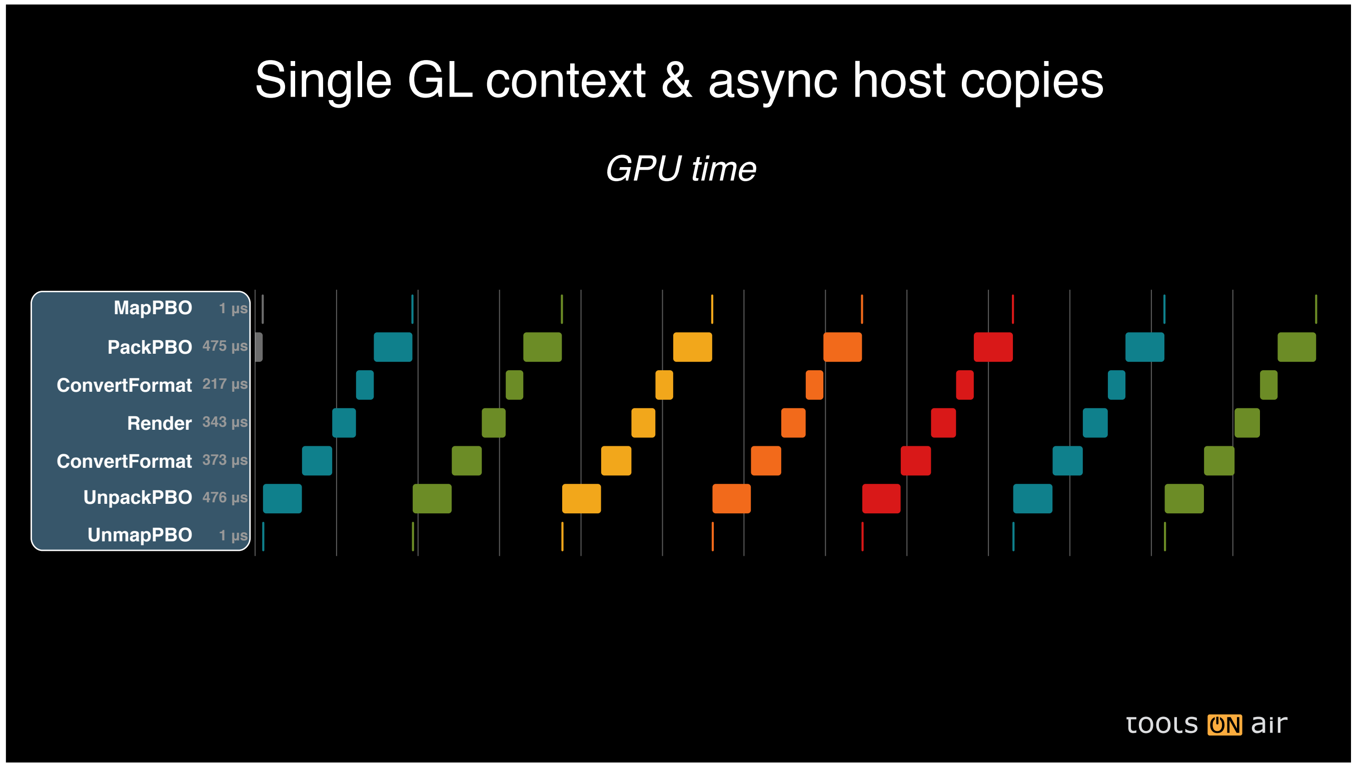


tools ON air

- Let’s look at the real throughput with all our rendering operations and up/downloads in place we were able to achieve.



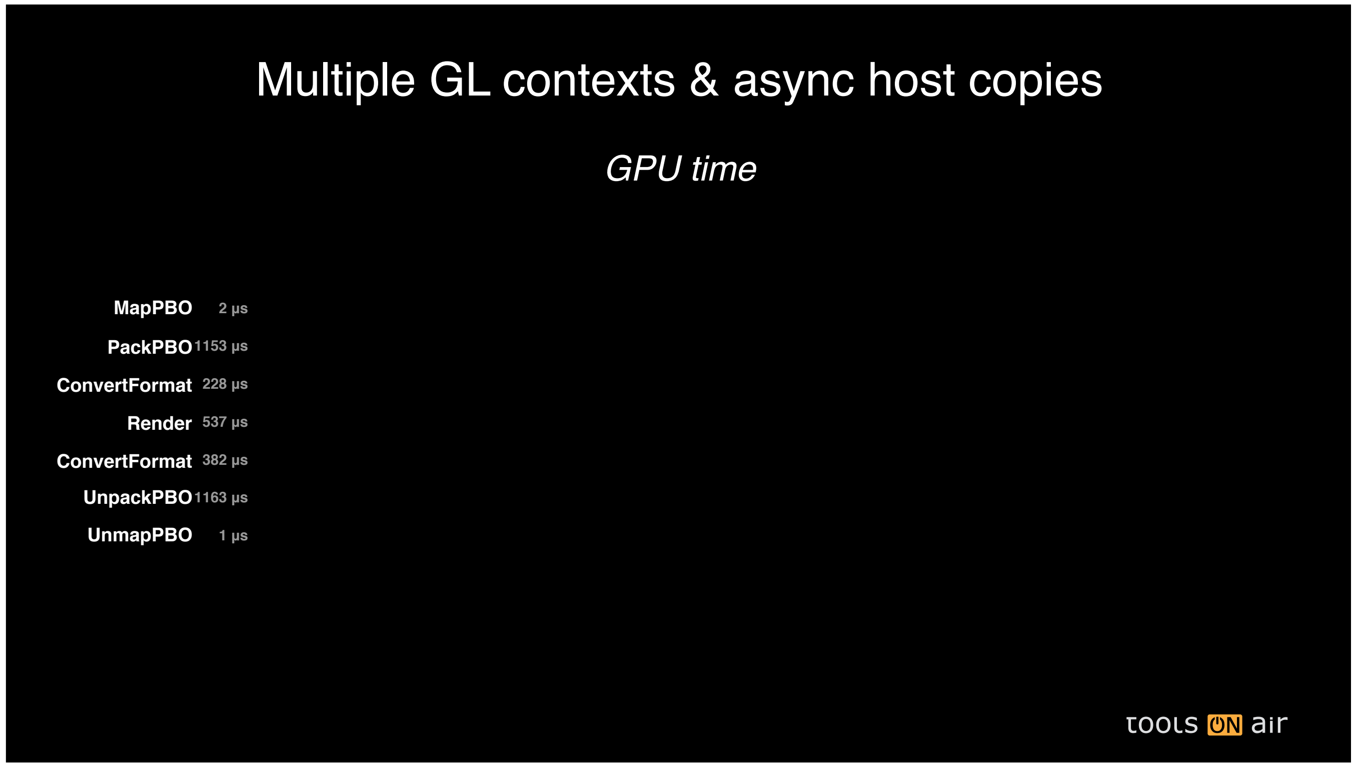
- Let’s look at the real throughput with all our rendering operations and up/downloads in place we were able to achieve.



## Single GL context & async host copies

*GPU time*

- If we look at the trace of the GPU timers (which where recorded using GL timer queries), we can see much better where the GPU is actually spending time.
- And most importantly we can see that on the GPU everything is still running sequentially, even though we are already using async PBO transfers

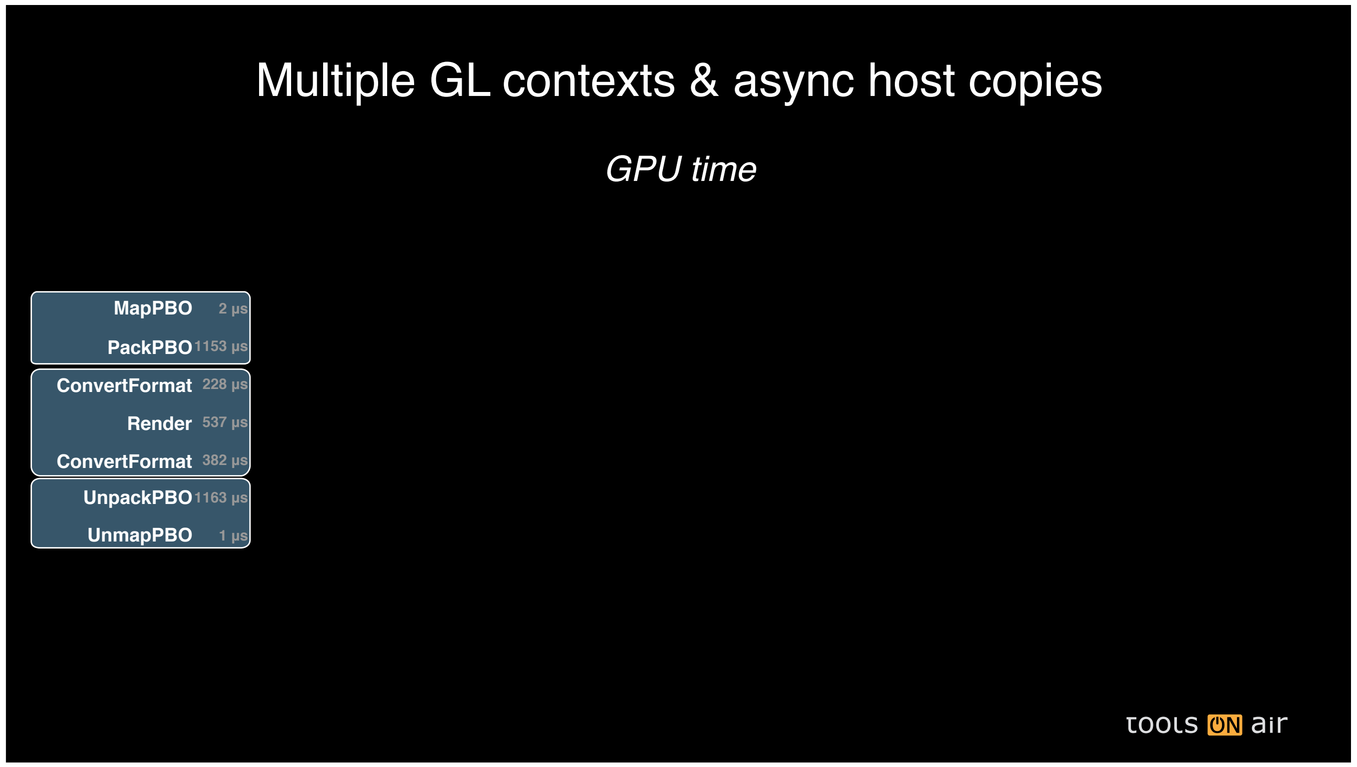


## Multiple GL contexts & async host copies

*GPU time*

| | |
|---|---|
| MapPBO | 2 µs |
| PackPBO | 1153 µs |
| ConvertFormat | 228 µs |
| Render | 537 µs |
| ConvertFormat | 382 µs |
| UnpackPBO | 1163 µs |
| UnmapPBO | 1 µs |

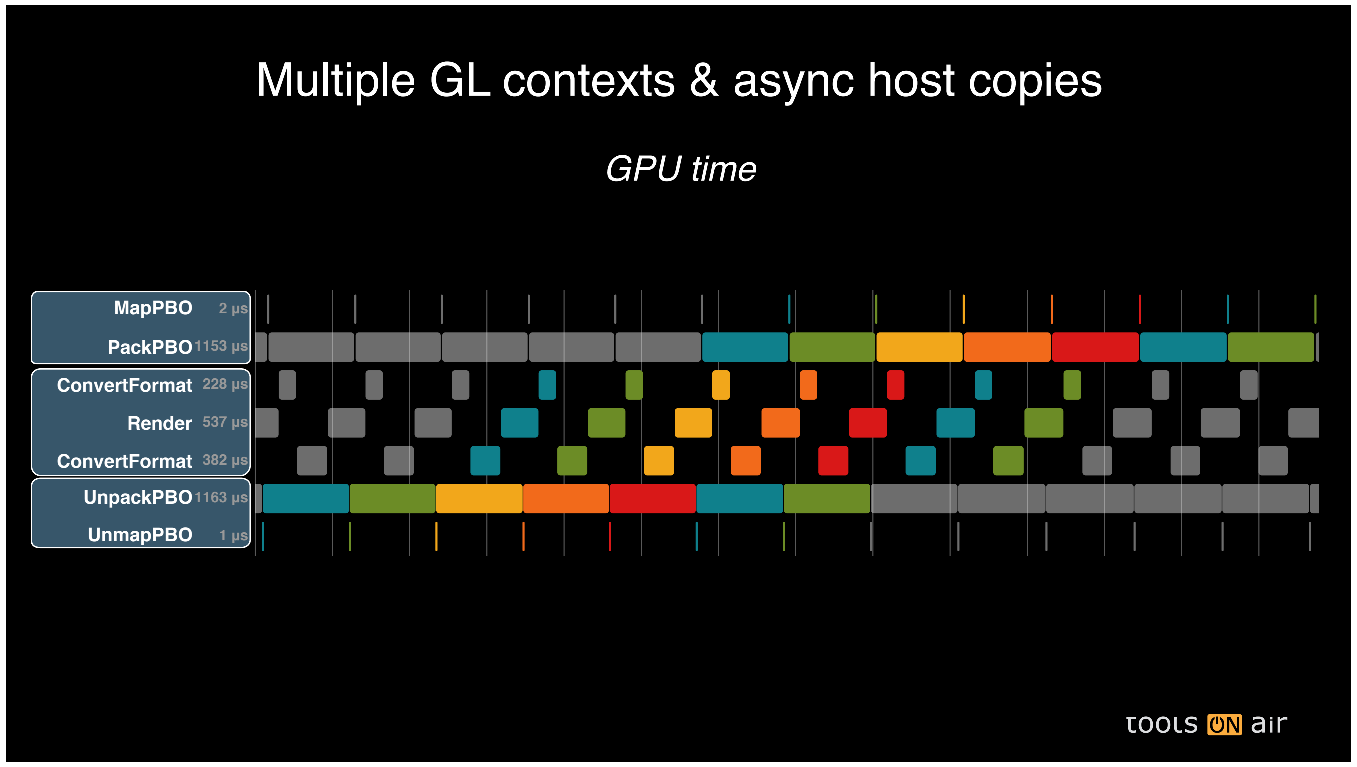
- If we look at the GPU trace this time, we'll see confirmation that the GPU now concurrently performs upload, render and download
- While this is still standard OpenGL, this is achieved by the NVIDIA drivers running on a Quadro with two DMA engines (aka dual-copy engines)



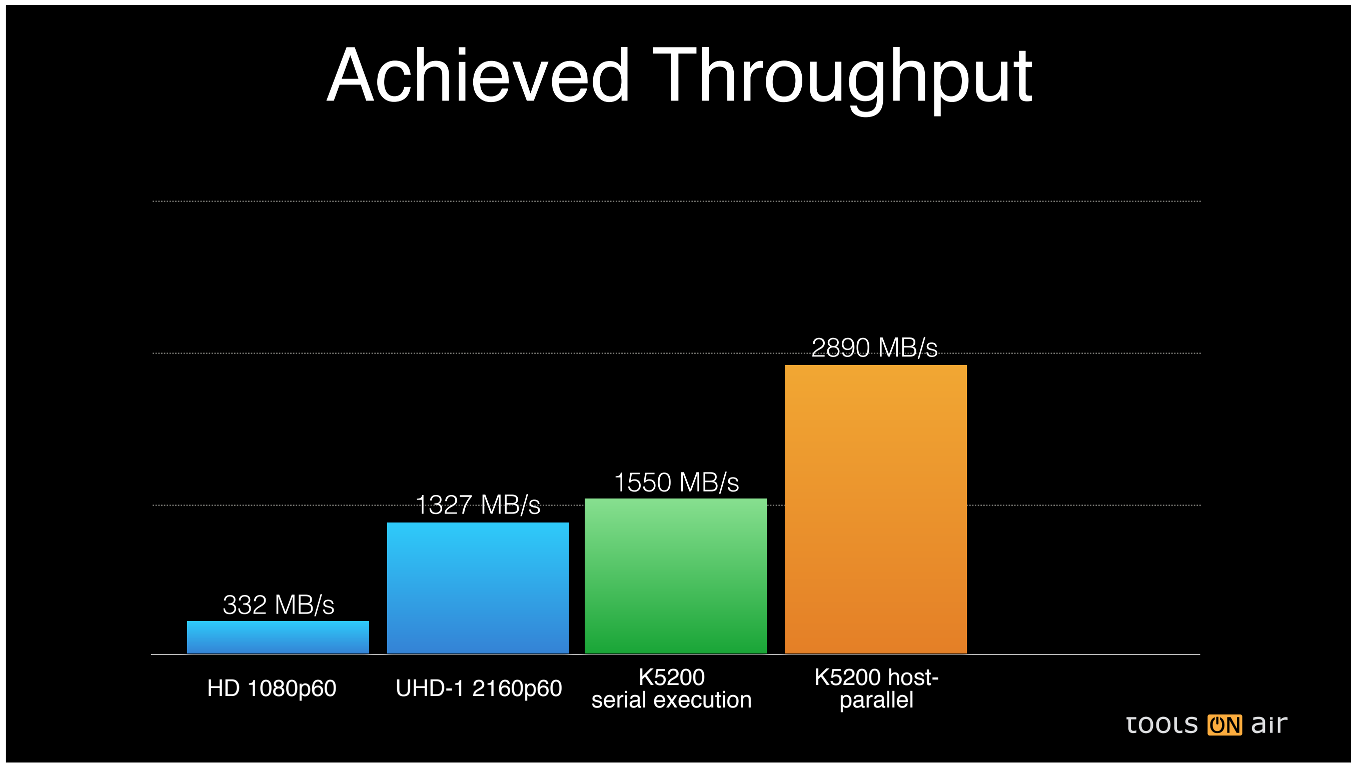
## Multiple GL contexts & async host copies

*GPU time*

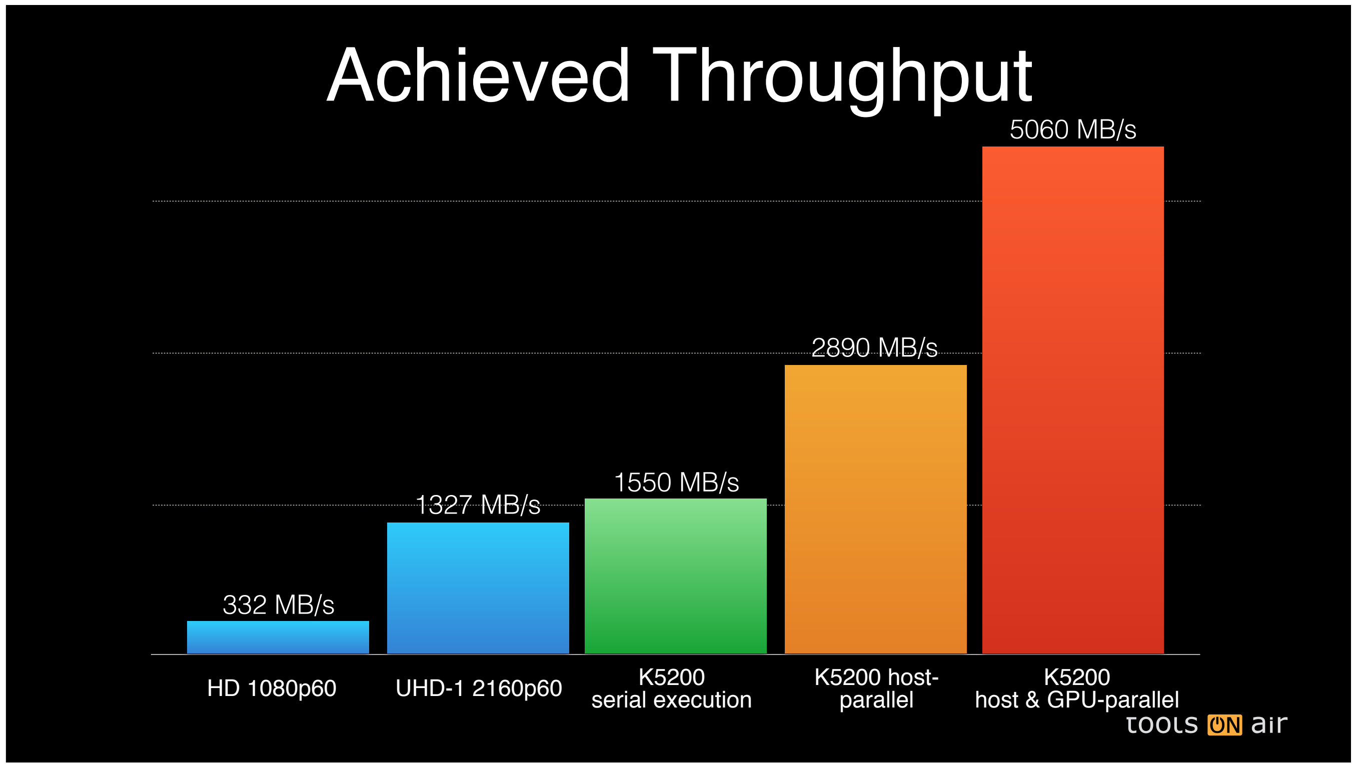
- If we look at the GPU trace this time, we'll see confirmation that the GPU now concurrently performs upload, render and download
- While this is still standard OpenGL, this is achieved by the NVIDIA drivers running on a Quadro with two DMA engines (aka dual-copy engines)



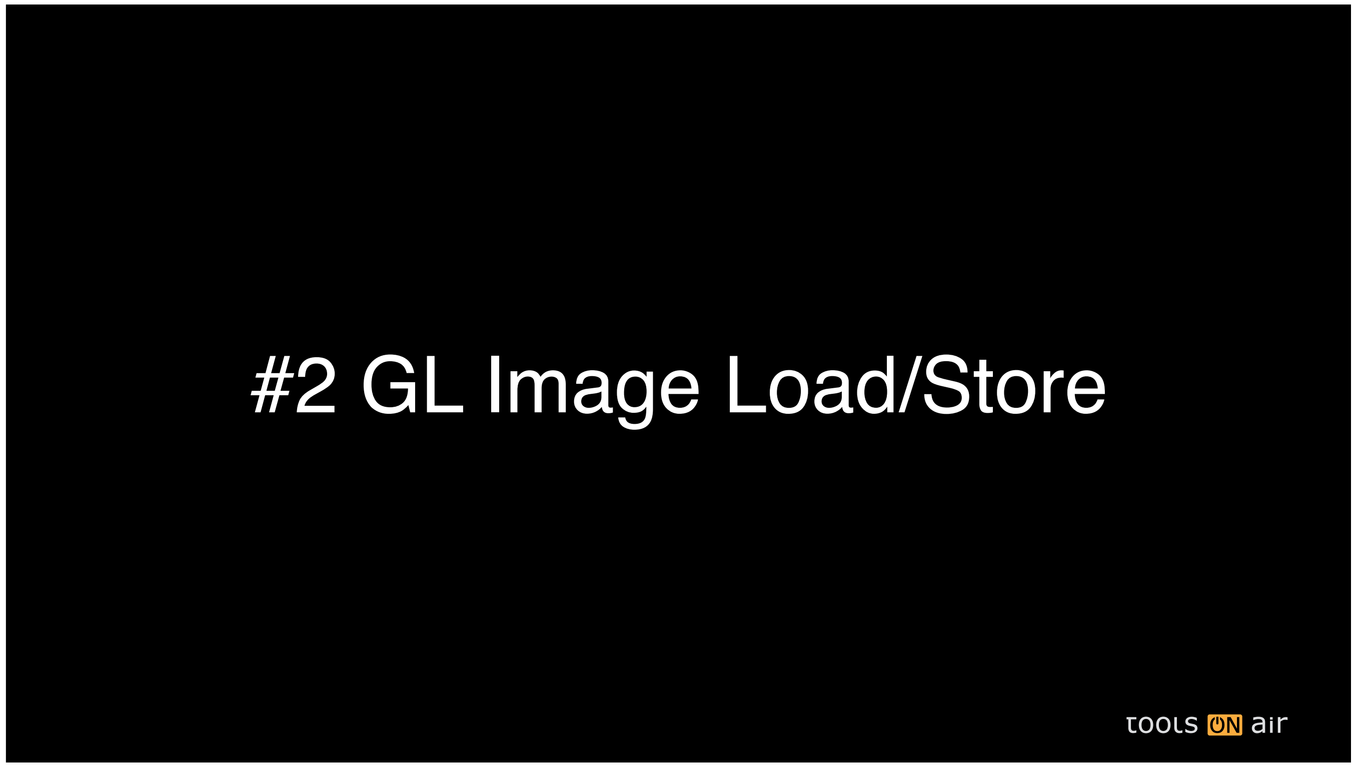
- If we look at the GPU trace this time, we'll see confirmation that the GPU now concurrently performs upload, render and download
- While this is still standard OpenGL, this is achieved by the NVIDIA drivers running on a Quadro with two DMA engines (aka dual-copy engines)



- That again almost doubles the performance. This throughput would easily allow rendering of three UHD streams in real time



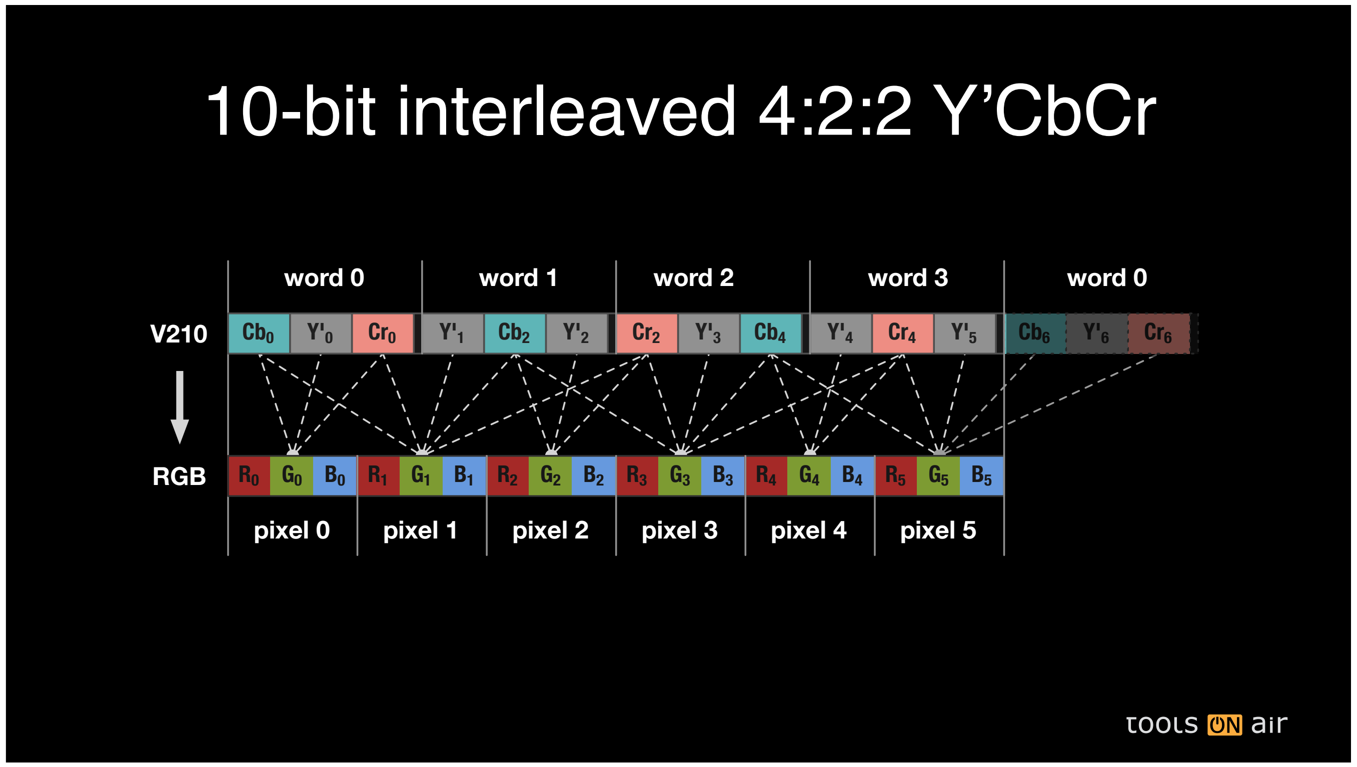
- That again almost doubles the performance. This throughput would easily allow rendering of three UHD streams in real time



## #2 GL Image Load/Store

tools ON air

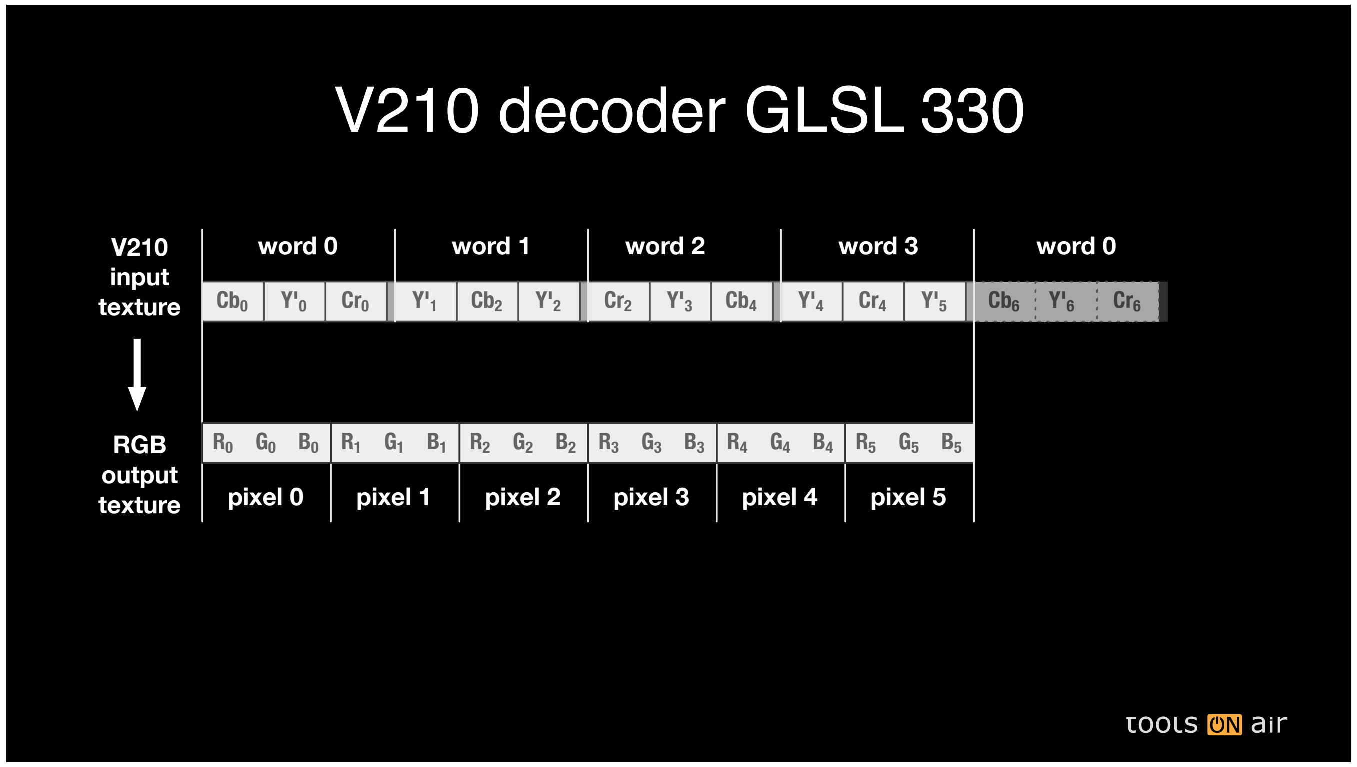
- The second most important optimisation technique that we found, is to take advantage of GLSL image/load operations in shaders for format conversions



## 10-bit interleaved 4:2:2 Y’CbCr

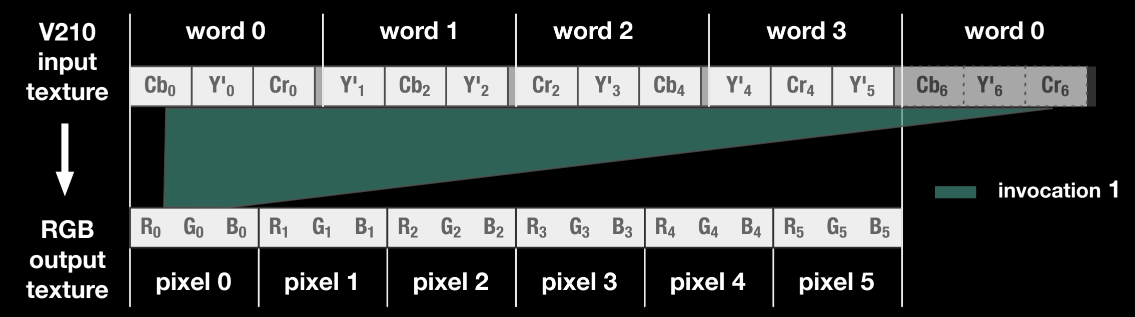
word 0
word 1
word 2
word 3
word 0
V210
$Cb_0$ $Y'_0$ $Cr_0$ $Y'_1$ $Cb_2$ $Y'_2$ $Cr_2$ $Y'_3$ $Cb_4$ $Y'_4$ $Cr_4$ $Y'_5$ $Cb_6$ $Y'_6$ $Cr_6$
RGB
$R_0$ $G_0$ $B_0$ $R_1$ $G_1$ $B_1$ $R_2$ $G_2$ $B_2$ $R_3$ $G_3$ $B_3$ $R_4$ $G_4$ $B_4$ $R_5$ $G_5$ $B_5$
pixel 0
pixel 1
pixel 2
pixel 3
pixel 4
pixel 5
tools ON air

- In our benchmarking use case, we took the challenge of using a very nasty pixel format called V210
- It’s an interleaved 4:2:2-subsampled luma/chroma format using 10-bit components
- Three 10-bit luma/chroma components are packed into a single 32-bit word and zero-padded with 2 bits.
- Because of the interleaved luma/chroma pattern, you would reconstruct 6 RGB pixels from a group of four 32-bit words
- That’s usually very difficult to work with



- We've first implemented the conversion using GLSL 330 shaders
- Here you have to output a single pixel for each invocation of the shader
- But because of the interleaved storing, you would do a lot of redundant work for each invocation

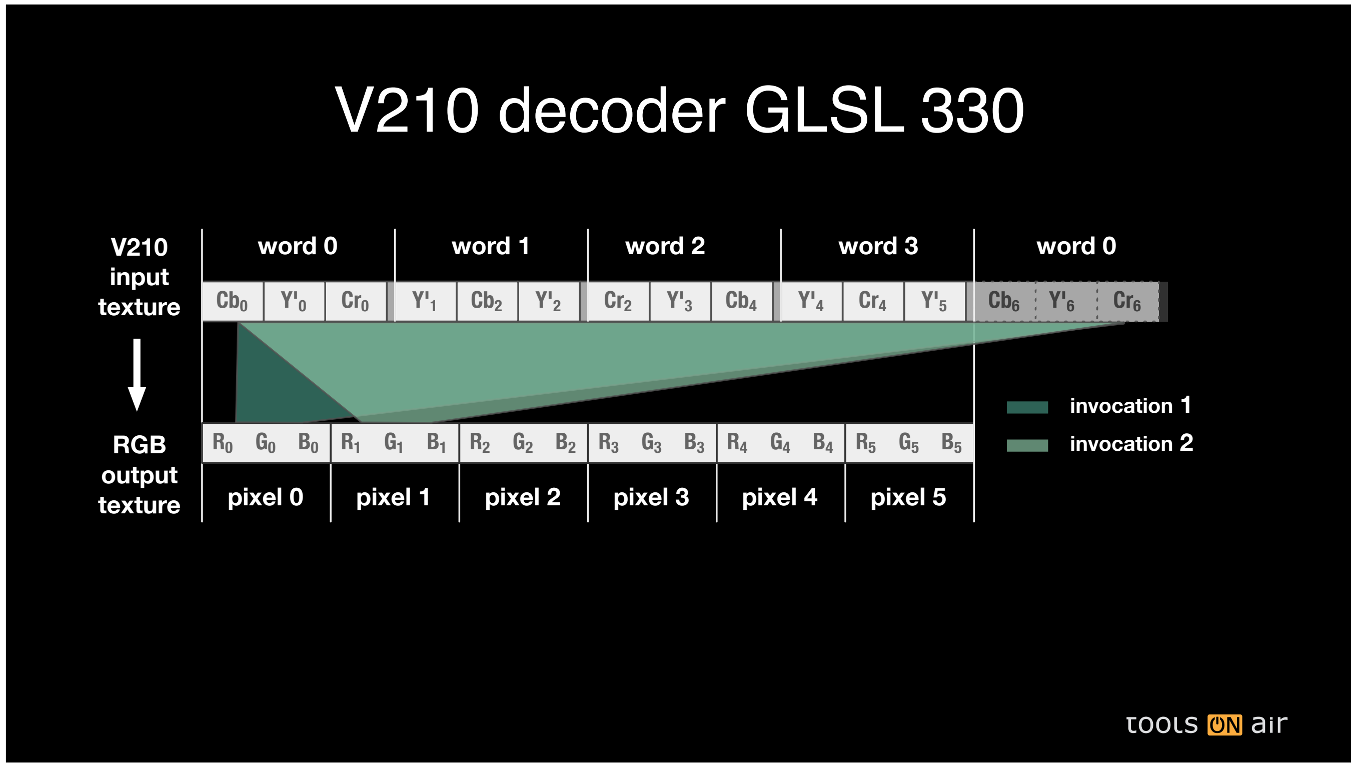
## V210 decoder GLSL 330



V210 input texture
word 0
word 1
word 2
word 3
word 0
$Cb_0$ $Y'_0$ $Cr_0$ $Y'_1$ $Cb_2$ $Y'_2$ $Cr_2$ $Y'_3$ $Cb_4$ $Y'_4$ $Cr_4$ $Y'_5$ $Cb_6$ $Y'_6$ $Cr_6$
invocation 1
RGB output texture
$R_0$ $G_0$ $B_0$ $R_1$ $G_1$ $B_1$ $R_2$ $G_2$ $B_2$ $R_3$ $G_3$ $B_3$ $R_4$ $G_4$ $B_4$ $R_5$ $G_5$ $B_5$
pixel 0
pixel 1
pixel 2
pixel 3
pixel 4
pixel 5

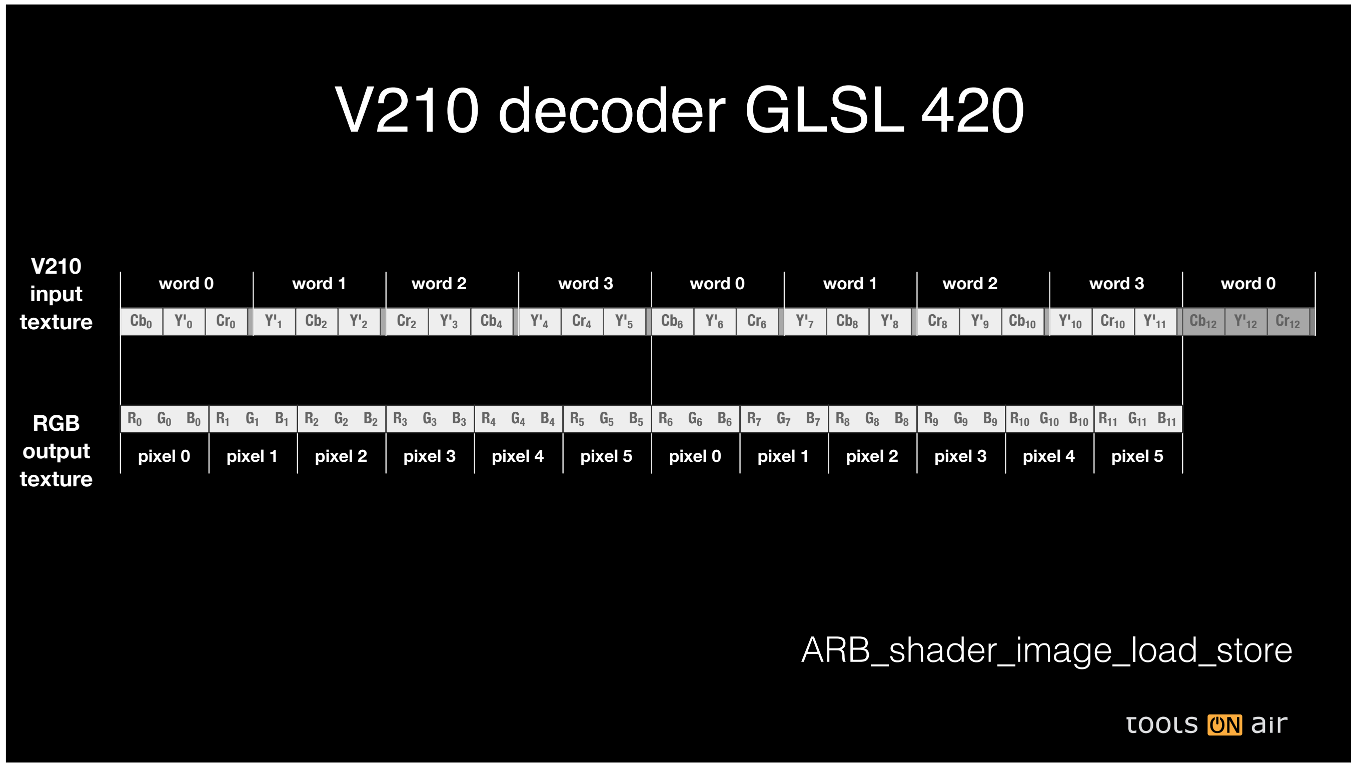
tools ON air

- We’ve first implemented the conversion using GLSL 330 shaders
- Here you have to output a single pixel for each invocation of the shader
- But because of the interleaved storing, you would do a lot of redundant work for each invocation



## V210 decoder GLSL 330

- We've first implemented the conversion using GLSL 330 shaders
- Here you have to output a single pixel for each invocation of the shader
- But because of the interleaved storing, you would do a lot of redundant work for each invocation



## V210 decoder GLSL 420

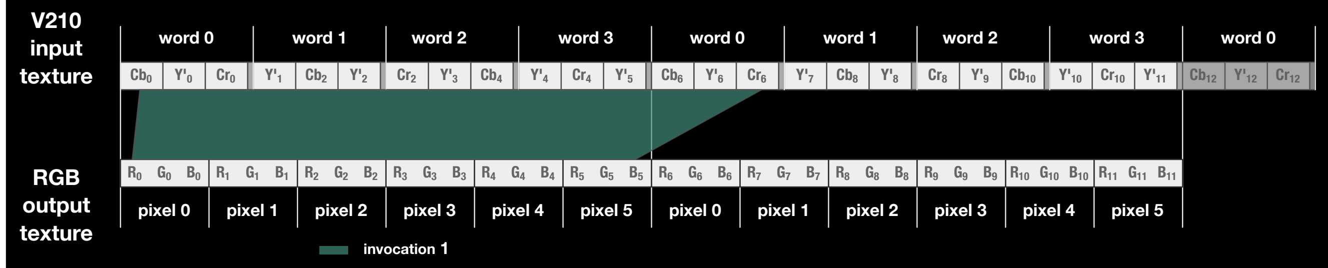
V210 input texture
word 0
word 1
word 2
word 3
word 0
word 1
word 2
word 3
word 0
Cb0 Y'0 Cr0 Y'1 Cb2 Y'2 Cr2 Y'3 Cb4 Y'4 Cr4 Y'5 Cb6 Y'6 Cr6 Y'7 Cb8 Y'8 Cr8 Y'9 Cb10 Y'10 Cr10 Y'11 Cb12 Y'12 Cr12
RGB output texture
R0 G0 B0 R1 G1 B1 R2 G2 B2 R3 G3 B3 R4 G4 B4 R5 G5 B5 R6 G6 B6 R7 G7 B7 R8 G8 B8 R9 G9 B9 R10 G10 B10 R11 G11 B11
pixel 0
pixel 1
pixel 2
pixel 3
pixel 4
pixel 5
pixel 0
pixel 1
pixel 2
pixel 3
pixel 4
pixel 5

ARB_shader_image_load_store

tools ON air

- Using GLSL 420, we can take advantage of image load/store operations, where you can perform random writes into texture images.
- That way we can writes all 6 pixels for a 32-bit word in each shader invocation
- In some cases, this is about 8 times faster than the GLSL 330 implementation
- By the way, we have also implemented this using compute shaders, which was just a bit slower than using this approach

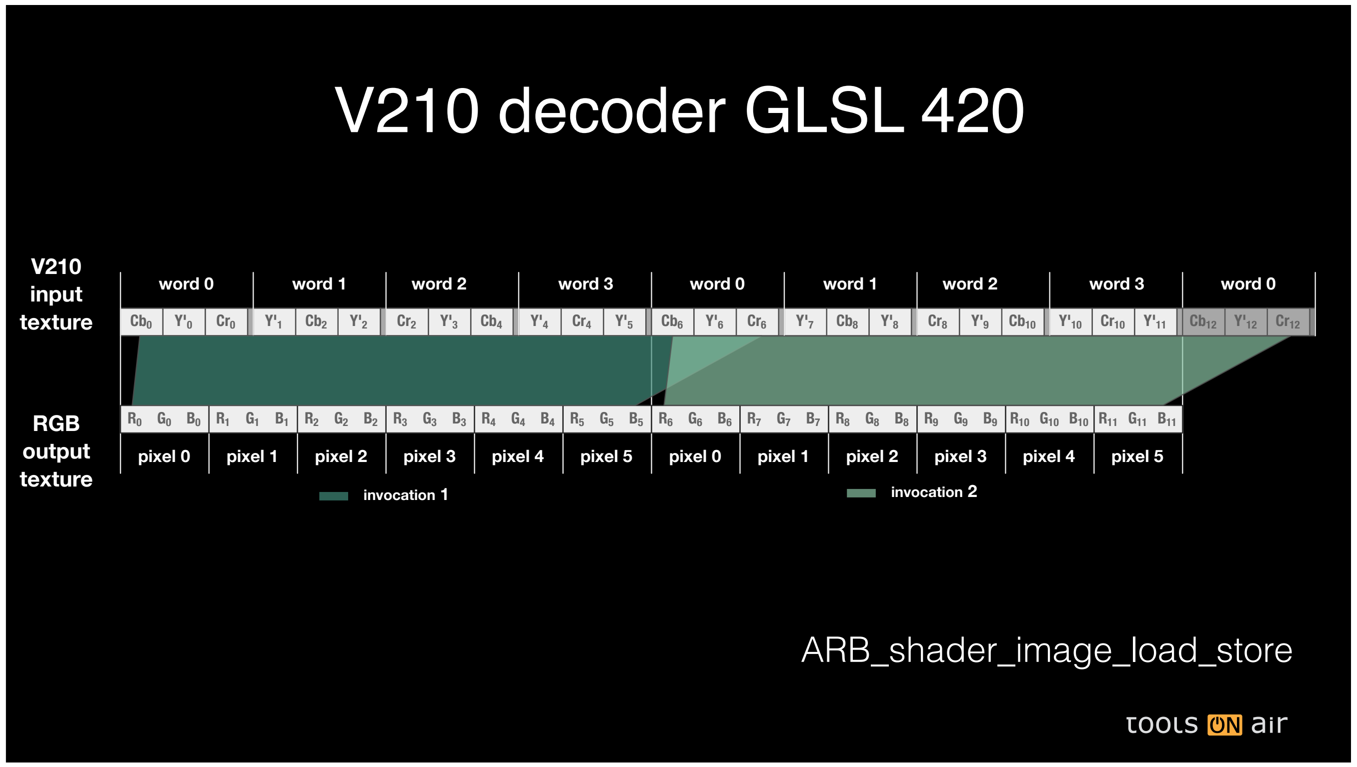
## V210 decoder GLSL 420



ARB_shader_image_load_store

tools ON air

- Using GLSL 420, we can take advantage of image load/store operations, where you can perform random writes into texture images.
- That way we can writes all 6 pixels for a 32-bit word in each shader invocation
- In some cases, this is about 8 times faster than the GLSL 330 implementation
- By the way, we have also implemented this using compute shaders, which was just a bit slower than using this approach



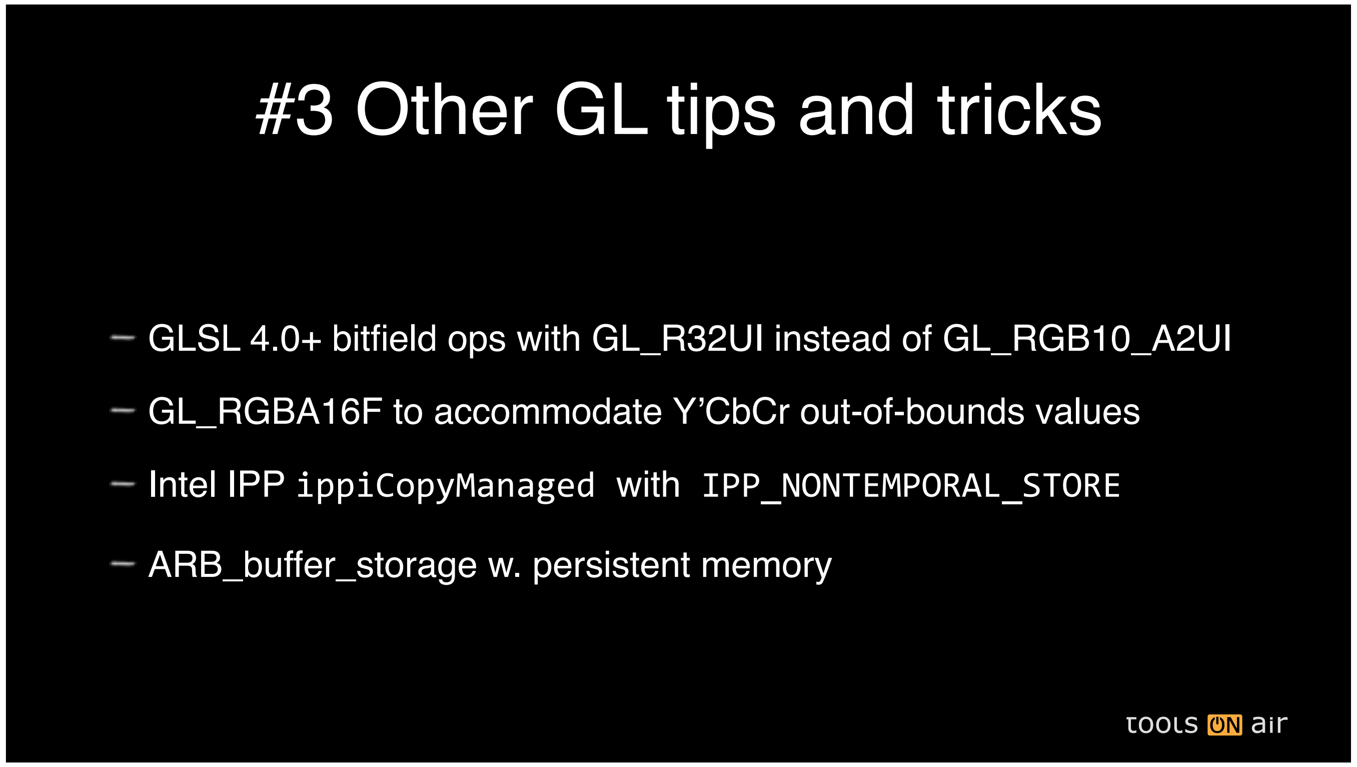
## V210 decoder GLSL 420

V210 input texture
word 0
word 1
word 2
word 3
word 0
word 1
word 2
word 3
word 0
Cb0 Y'0 Cr0 Y'1 Cb2 Y'2 Cr2 Y'3 Cb4 Y'4 Cr4 Y'5 Cb6 Y'6 Cr6 Y'7 Cb8 Y'8 Cr8 Y'9 Cb10 Y'10 Cr10 Y'11 Cb12 Y'12 Cr12
RGB output texture
R0 G0 B0 R1 G1 B1 R2 G2 B2 R3 G3 B3 R4 G4 B4 R5 G5 B5 R6 G6 B6 R7 G7 B7 R8 G8 B8 R9 G9 B9 R10 G10 B10 R11 G11 B11
pixel 0
pixel 1
pixel 2
pixel 3
pixel 4
pixel 5
pixel 0
pixel 1
pixel 2
pixel 3
pixel 4
pixel 5
invocation 1
invocation 2

ARB_shader_image_load_store

tools ON air

- Using GLSL 420, we can take advantage of image load/store operations, where you can perform random writes into texture images.
- That way we can writes all 6 pixels for a 32-bit word in each shader invocation
- In some cases, this is about 8 times faster than the GLSL 330 implementation
- By the way, we have also implemented this using compute shaders, which was just a bit slower than using this approach



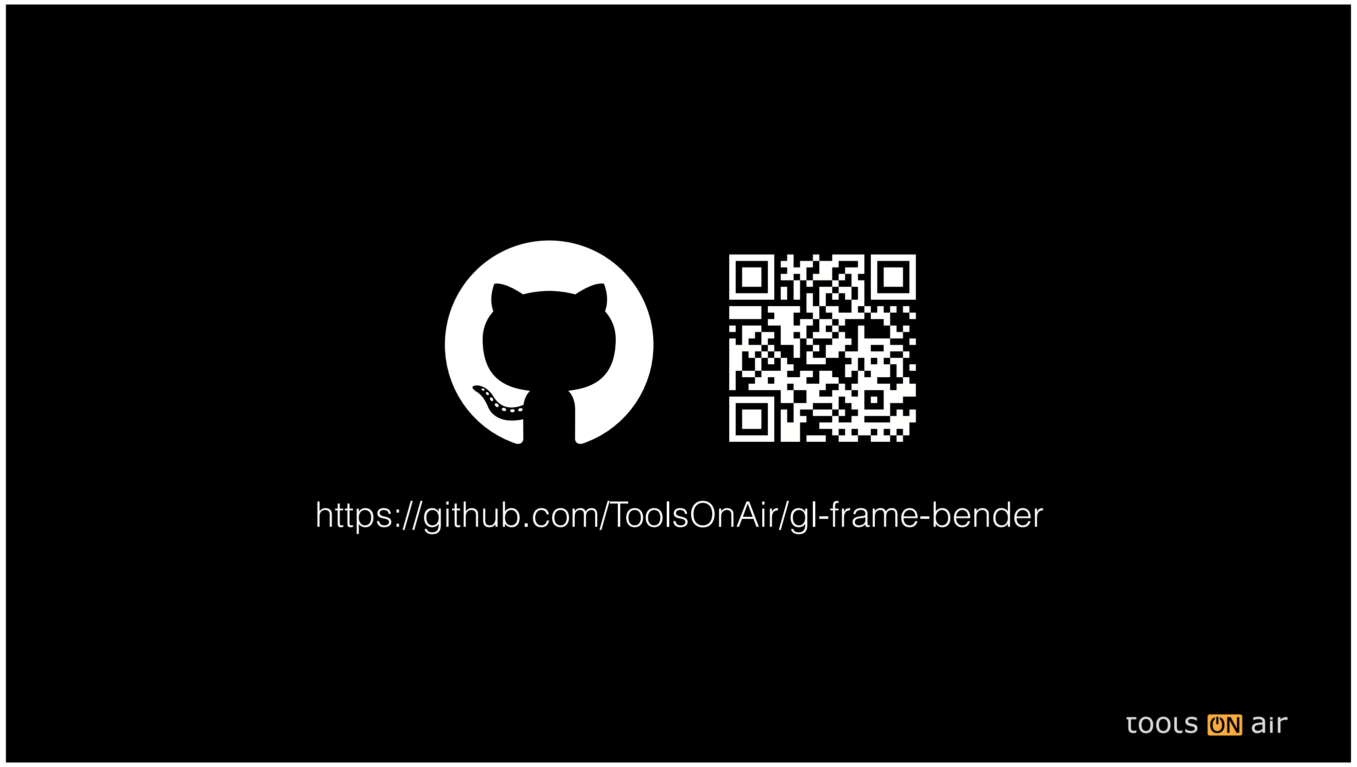
## #3 Other GL tips and tricks

- GLSL 4.0+ bitfield ops with GL_R32UI instead of GL_RGB10_A2UI
- GL_RGBA16F to accommodate Y'CbCr out-of-bounds values
- Intel IPP `ippiCopyManaged` with `IPP_NONTEMPORAL_STORE`
- ARB_buffer_storage w. persistent memory

tools ON air

- In order to unpack the 10/10/10/2 pattern, it's much better to upload the frames using GL_R32UI pixel format and unpacking the words with GLSL 4 bitfield operations instead of using the native pixel format RGB10_A2UI. The difference here is about 1.6 GB/sec.
- The BT.709 luma/chroma color space is larger than the RGB color range, as such we might end up with negative RGB values which would be clipped when using normalised RGB texture formats. Using floating-point render formats allows us to keep those negative values and reduces the precision loss converting back and forth between luma/chroma and RGB.
- Since we perform the format conversion on the GPU, we can do all intermittent steps in floating-point RGB texture.
- Intel's Performance Primitive library has a handy function that copies data using non-temporal store, which avoid thrashing your caches. This improves throughput by about 1 GB/sec
- Finally, we have tested ARB_buffer_storage. While it doesn't improve performance in our case, it also doesn't hurt. And it might be interesting in situations where you can take advantage of persistently mapped PBO memory (e.g. keeping PBO-mapped pointers in memory pools for decoder, etc).

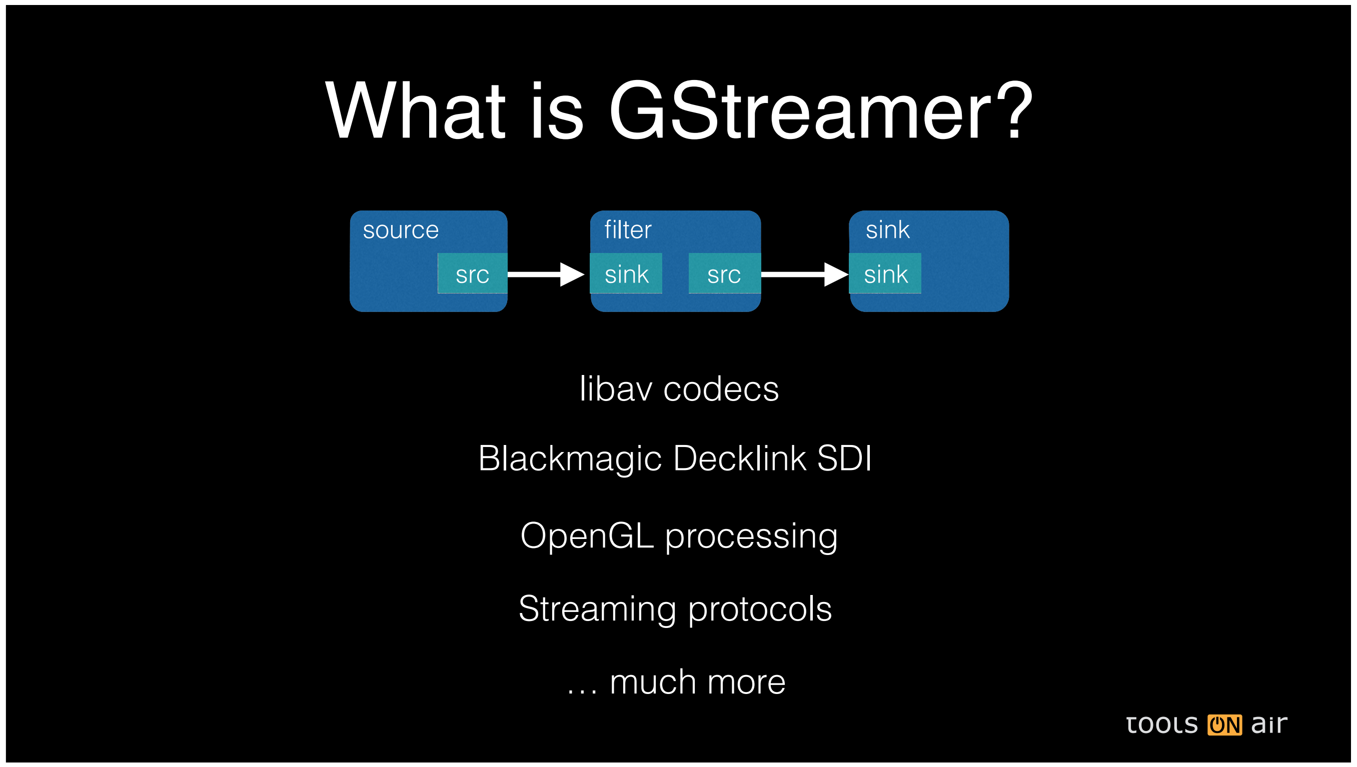
- At this time it’s my pleasure to announce that the complete benchmarking framework, including unit tests, scripts for visualisation and some other handy tools, is now online on Github as of this afternoon.
- It compiles and runs equally well on Window and Linux
- Please let me know if you have any issues building/running it.


https://github.com/ToolsOnAir/gl-frame-bender

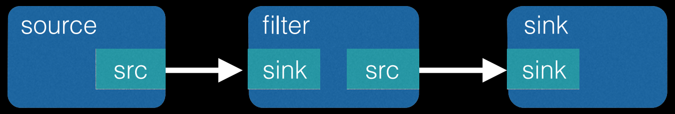
TOOLS ON air

- At this time it's my pleasure to announce that the complete benchmarking framework, including unit tests, scripts for visualisation and some other handy tools, is now online on Github as of this afternoon.
- It compiles and runs equally well on Window and Linux
- Please let me know if you have any issues building/running it.

- The benchmark tool provided some good insights into what is possible using modern OpenGL for video processing
- But building a real-world application needs a much broader infrastructure.
- And that’s where our second, more recent R&D project comes into play: A live mixing engine build on top of GStreamer



## What is GStreamer?



libav codecs

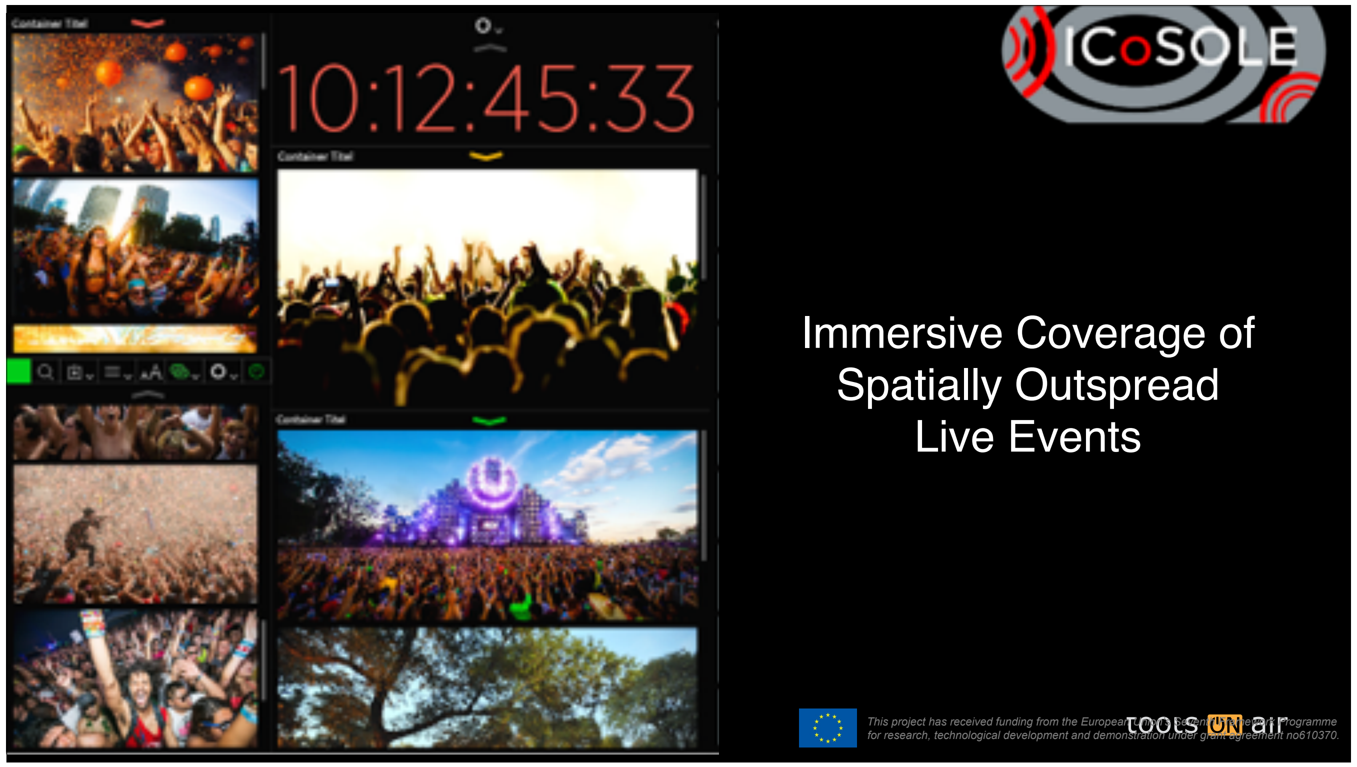
Blackmagic Decklink SDI

OpenGL processing

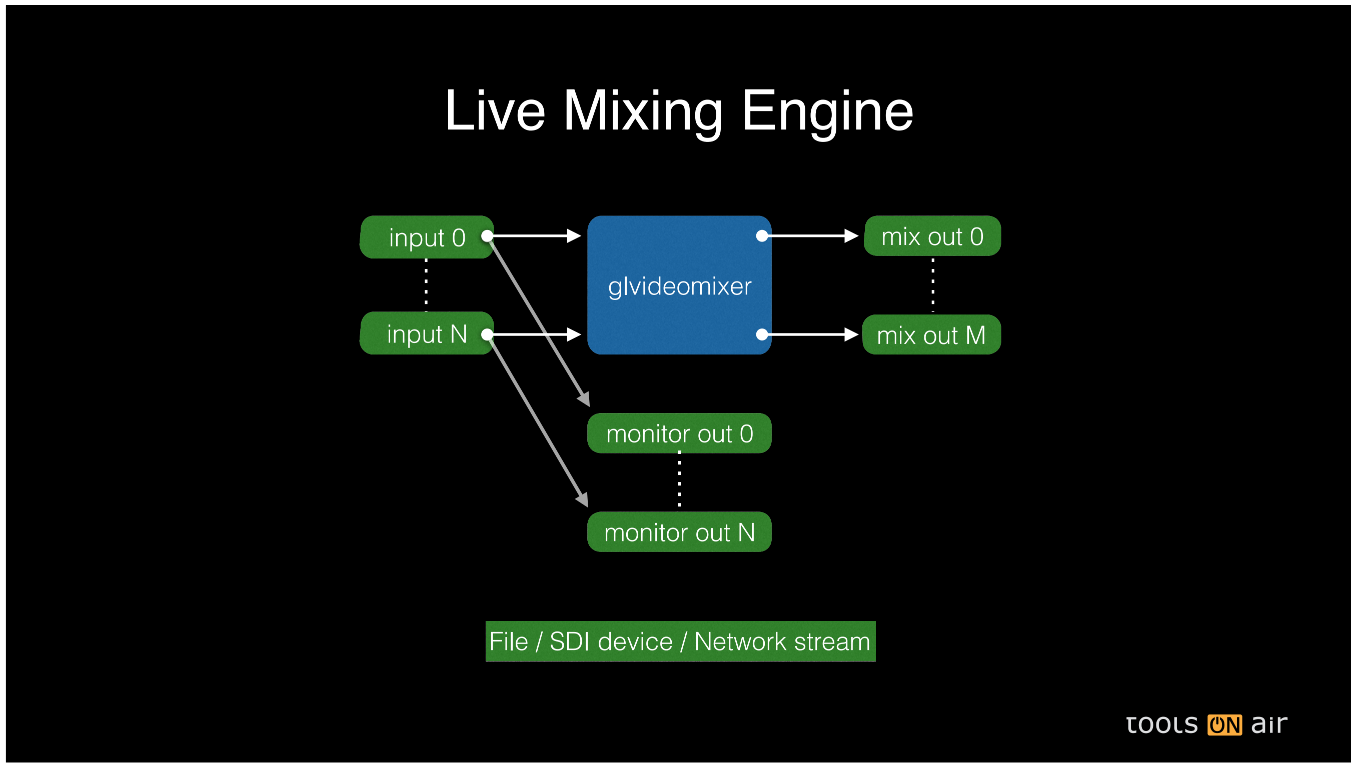
Streaming protocols

… much more

- Pipeline-based multimedia framework
- Elements are interconnected to pipelines
- Plugins provide elements
- LGPL, open-source and cross-platform
- ABI-stable object-oriented C API
- http://gstreamer.freedesktop.org

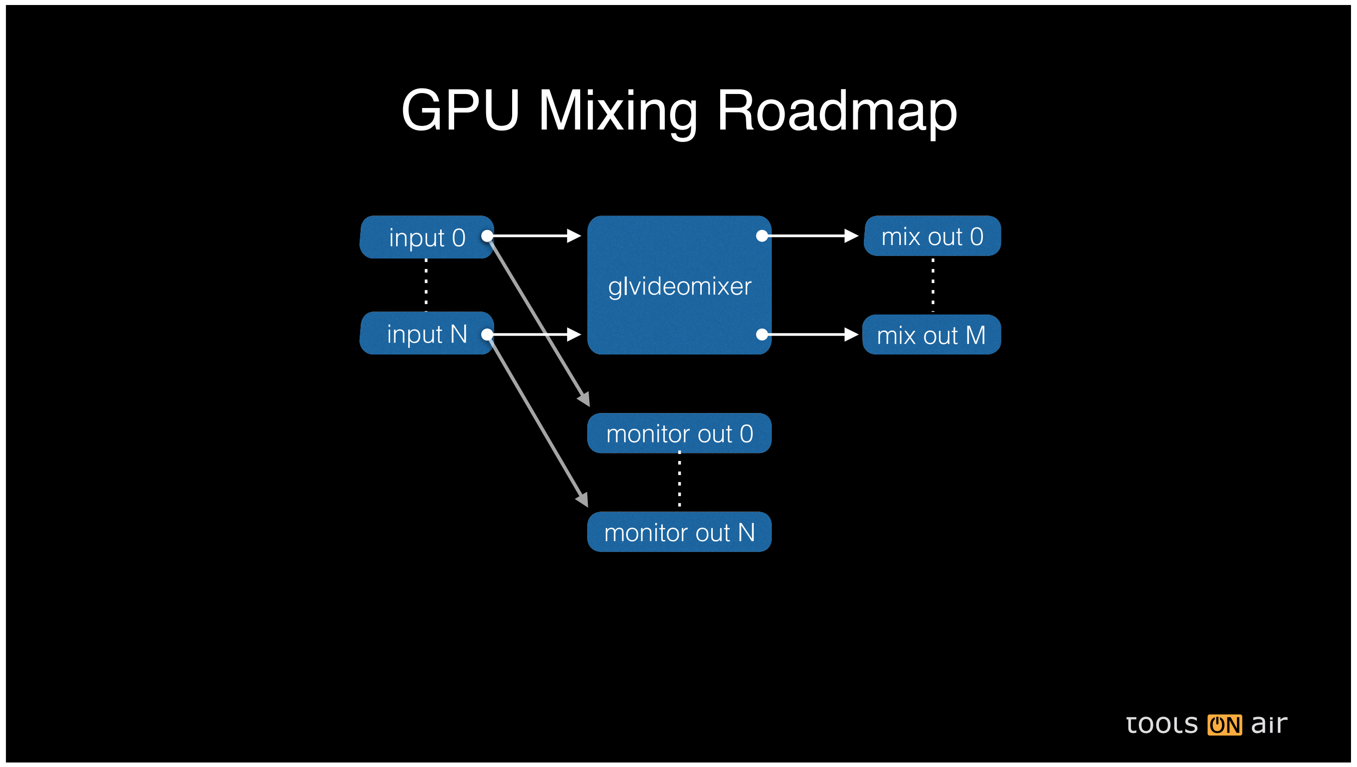

10:12:45:33
ICoSOLE
Immersive Coverage of Spatially Outspread Live Events
tools ON air
This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no610370.

- Toolsonair is part of a larger EU project called ICoSOLE
- In this project, we develop a live production tool and, most importantly, a live mixing engine that is used in a large-scale outdoor live production



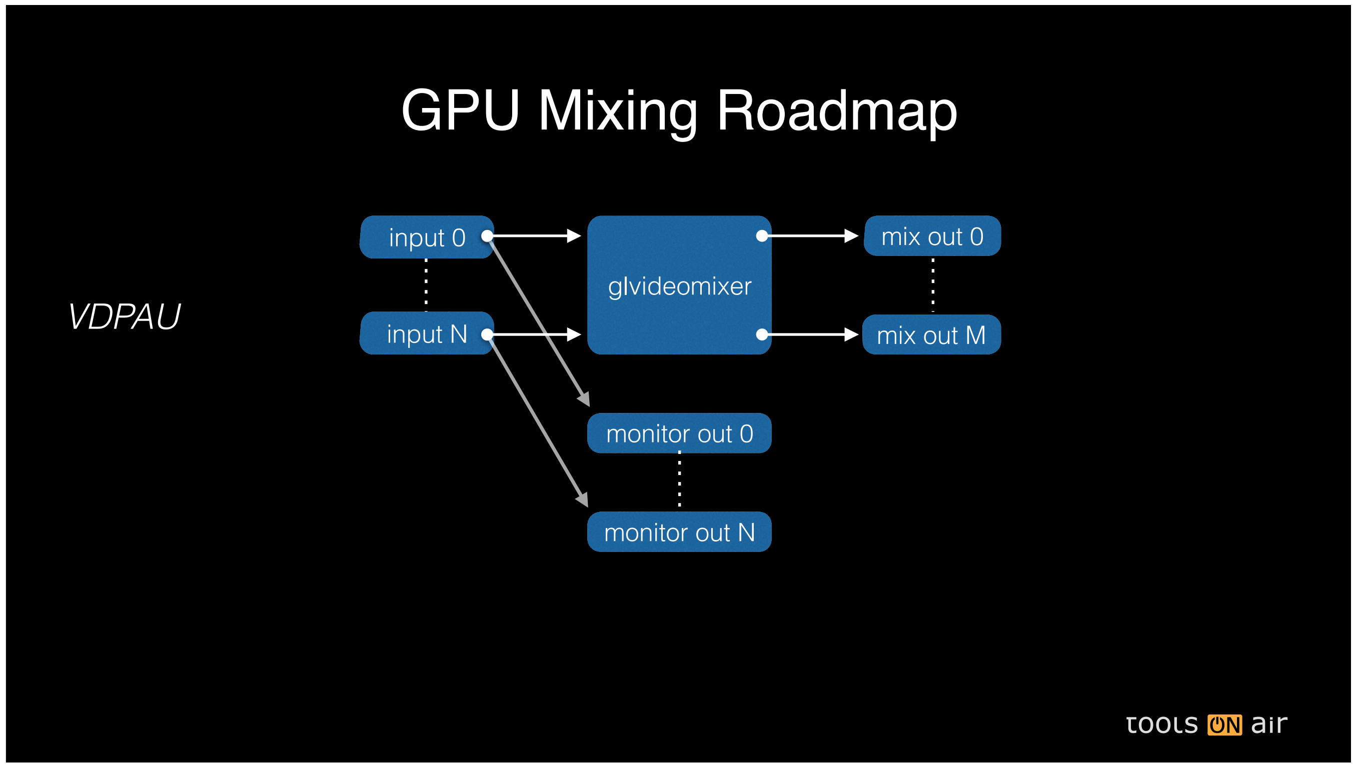
## Live Mixing Engine

- From a very high level, the engine mixes any number of input stream to any number of output streams
- Inputs and outputs can be SDI devices, network streams and file-based videos
- Each input can be monitored individually
- As of today, GStreamer 1.4 and up supports GL 3.2 / ES2 across all platforms
- We use “glvideomixer” to mix videos on the GPU which works really well and GIT master has much improved support for live mixing pipelines
- but texture transfers can quickly become the bottleneck and except video mixing we don’t do much on the GPU



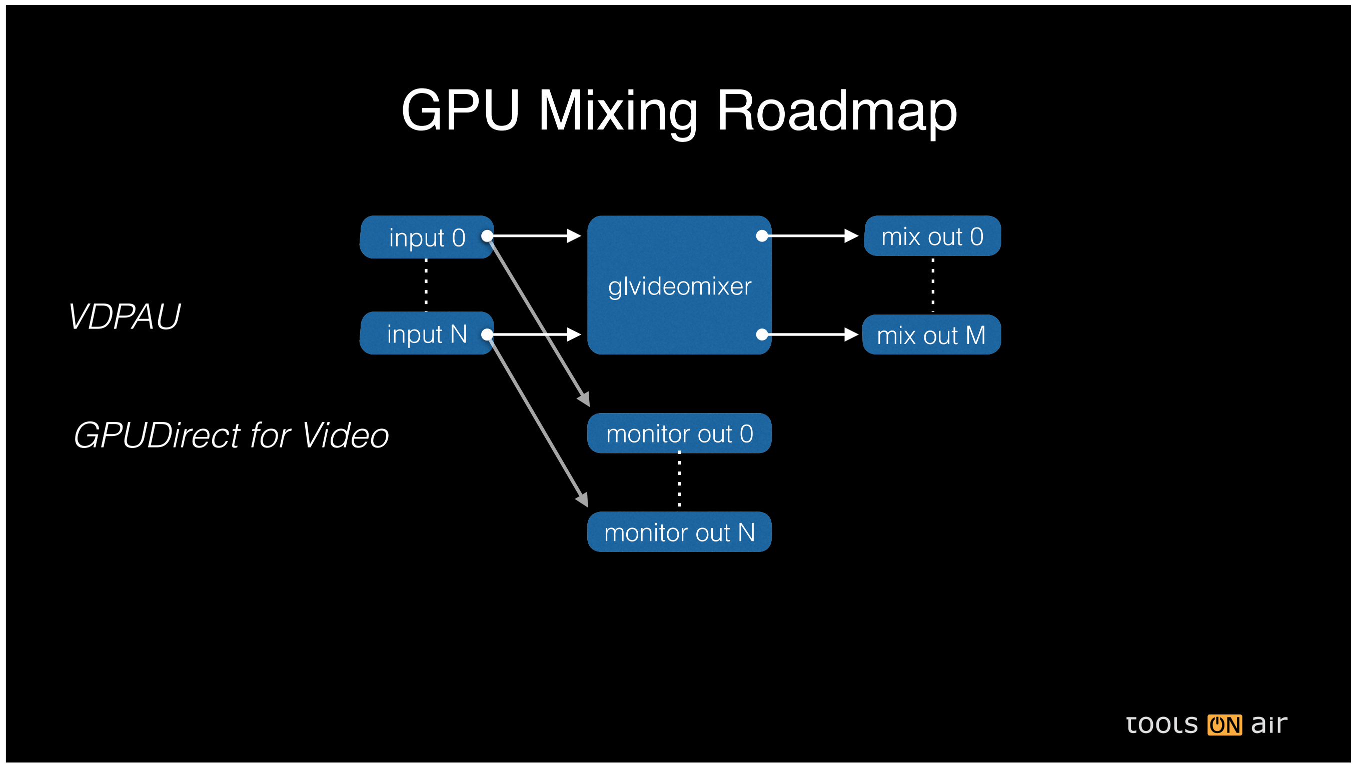
## GPU Mixing Roadmap

- VDPAU = Video Decode and Presentation API for Unix
- We work together closely with Centricular, a company founded by several GStreamer core developers
- Together with them, we'd like to add these features to GIT master upstream in the coming year
- If you have any questions about this development, and would like to help us speed up the process, please contact Sebastian Dröge of Centricular for these matters

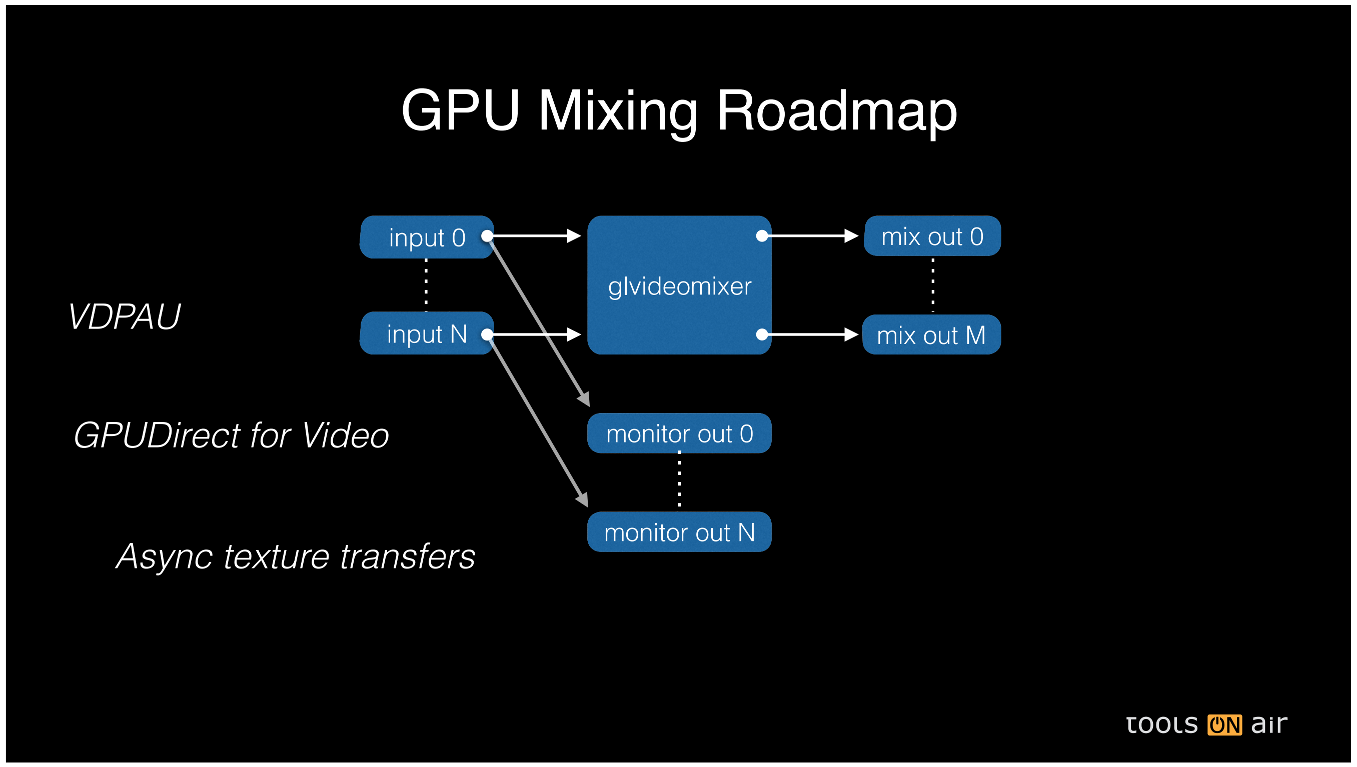


## GPU Mixing Roadmap

- VDPAU = Video Decode and Presentation API for Unix
- We work together closely with Centricular, a company founded by several GStreamer core developers
- Together with them, we'd like to add these features to GIT master upstream in the coming year
- If you have any questions about this development, and would like to help us speed up the process, please contact Sebastian Dröge of Centricular for these matters



- VDPAU = Video Decode and Presentation API for Unix
- We work together closely with Centricular, a company founded by several GStreamer core developers
- Together with them, we'd like to add these features to GIT master upstream in the coming year
- If you have any questions about this development, and would like to help us speed up the process, please contact Sebastian Dröge of Centricular for these matters



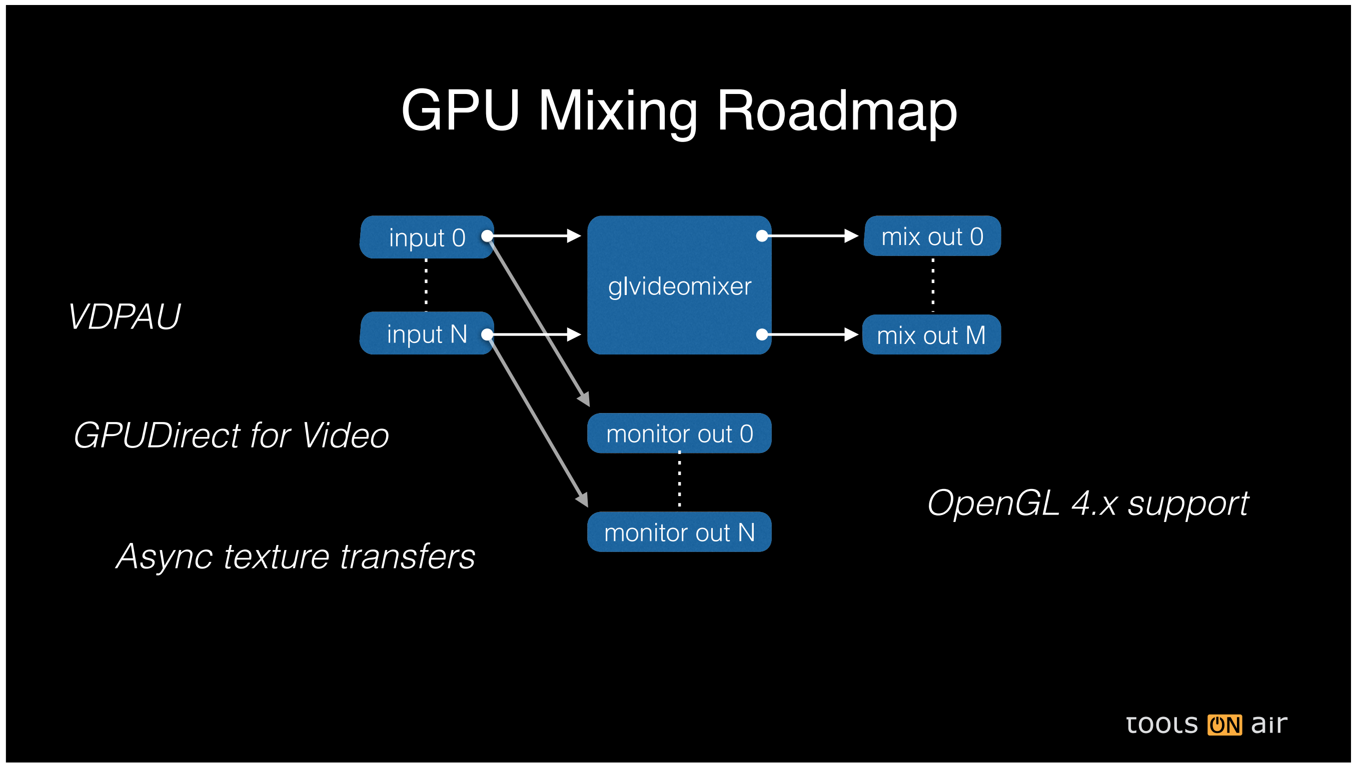
## GPU Mixing Roadmap

*VDPAU*

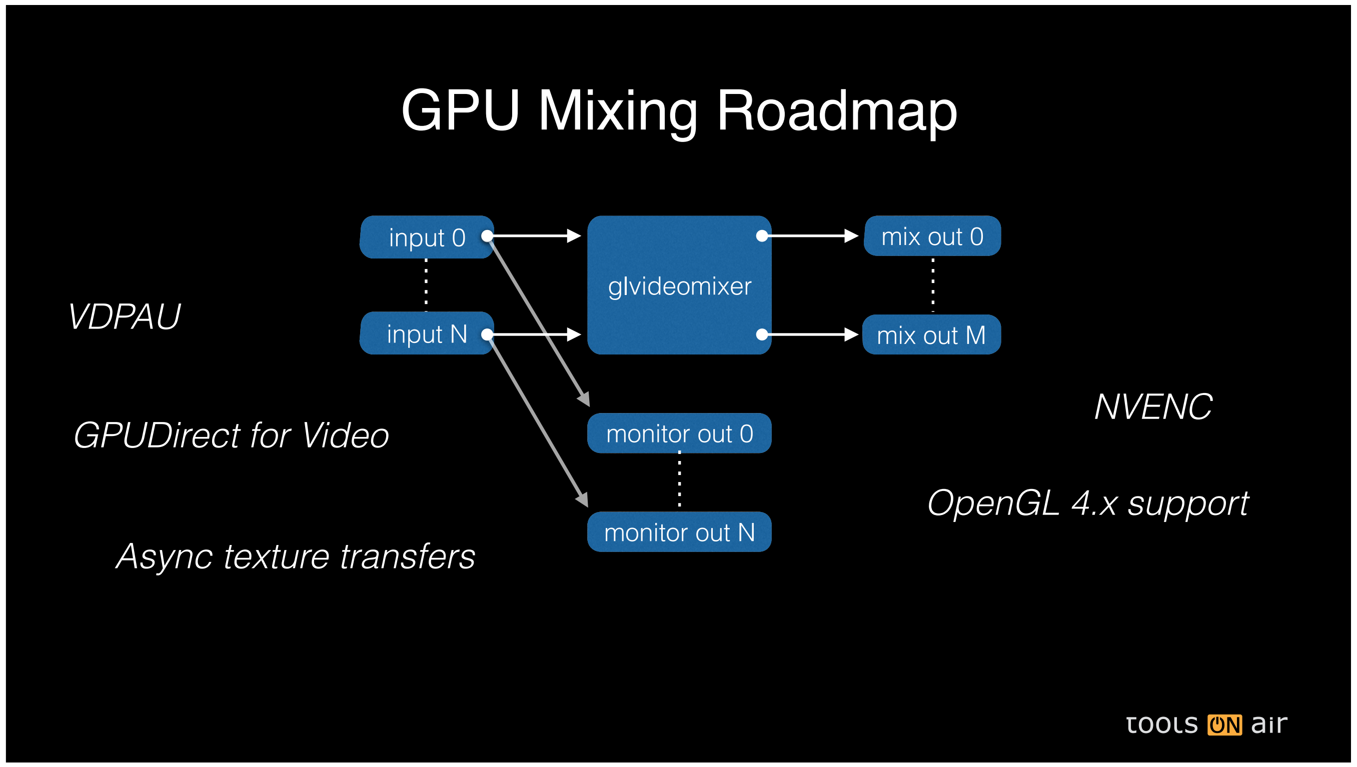
*GPUDirect for Video*

*Async texture transfers*

- VDPAU = Video Decode and Presentation API for Unix
- We work together closely with Centricular, a company founded by several GStreamer core developers
- Together with them, we'd like to add these features to GIT master upstream in the coming year
- If you have any questions about this development, and would like to help us speed up the process, please contact Sebastian Dröge of Centricular for these matters

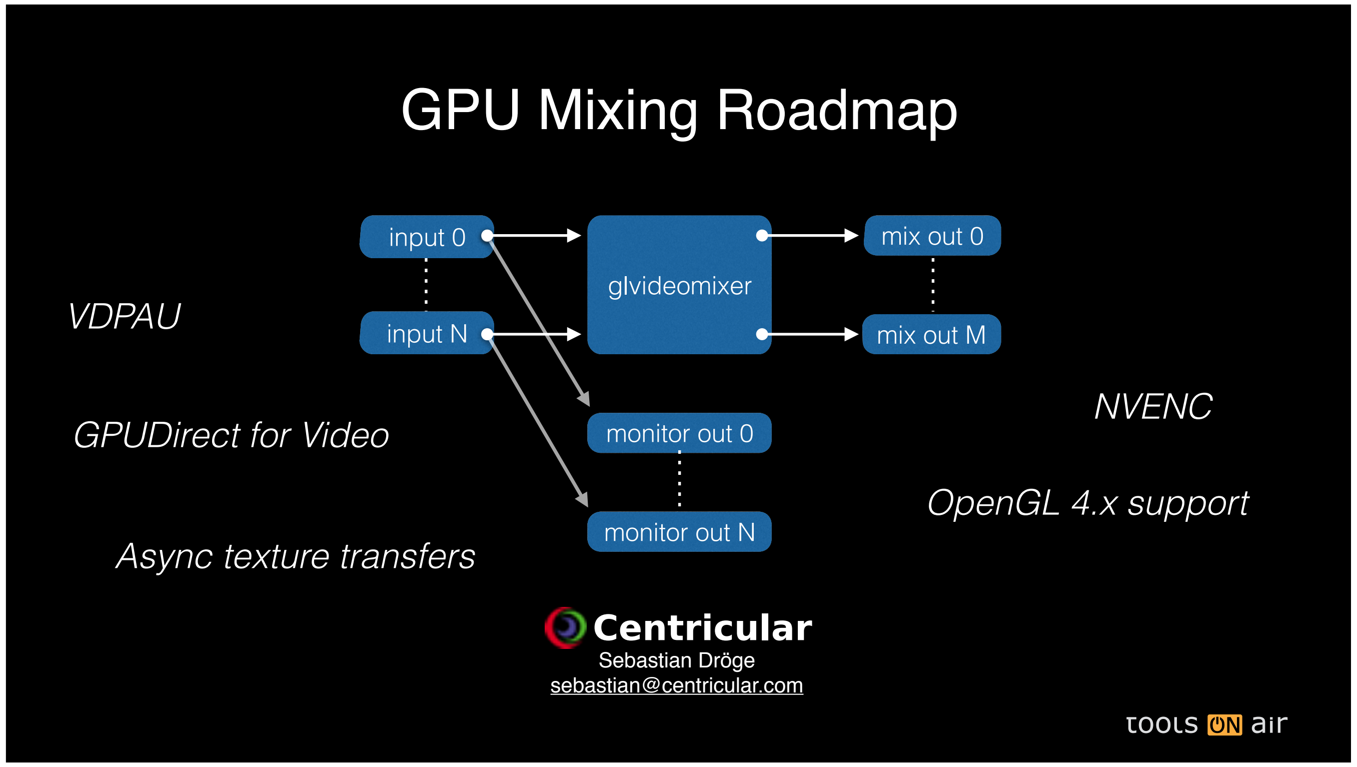

## GPU Mixing Roadmap

- VDPAU = Video Decode and Presentation API for Unix
- We work together closely with Centricular, a company founded by several GStreamer core developers
- Together with them, we'd like to add these features to GIT master upstream in the coming year
- If you have any questions about this development, and would like to help us speed up the process, please contact Sebastian Dröge of Centricular for these matters



## GPU Mixing Roadmap

- VDPAU = Video Decode and Presentation API for Unix
- We work together closely with Centricular, a company founded by several GStreamer core developers
- Together with them, we'd like to add these features to GIT master upstream in the coming year
- If you have any questions about this development, and would like to help us speed up the process, please contact Sebastian Dröge of Centricular for these matters



## GPU Mixing Roadmap

Centricular
Sebastian Dröge
sebastian@centricular.com

tools ON air

- VDPAU = Video Decode and Presentation API for Unix
- We work together closely with Centricular, a company founded by several GStreamer core developers
- Together with them, we'd like to add these features to GIT master upstream in the coming year
- If you have any questions about this development, and would like to help us speed up the process, please contact Sebastian Dröge of Centricular for these matters

## Summary

#1 gl-frame-bender

OpenGL 4.x recipes for high-performance real-time processing of professional video

#2 GStreamer live mixing

Build powerful real-world live mixing using state-of-the-art open-source tools

hfink@toolsonair.com

Twitter: @heinrichfink

tools ON air

ICoSOLE

http://www.icosole.eu

*This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no610370.*