drm/komeda Arm display driver

The drm/komeda driver supports the Arm display processor D71 and later products, this document gives a brief overview of driver design: how it works and why design it like that.

Overview of D71 like display IPs

From D71, Arm display IP begins to adopt a flexible and modularized architecture. A display pipeline is made up of multiple individual and functional pipeline stages called components, and every component has some specific capabilities that can give the flowed pipeline pixel data a particular processing.

Typical D71 components:

Layer

Layer is the first pipeline stage, which prepares the pixel data for the next stage. It fetches the pixel from memory, decodes it if it's AFBC, rotates the source image, unpacks or converts YUV pixels to the device internal RGB pixels, then adjusts the color_space of pixels if needed.

Scaler

As its name suggests, scaler takes responsibility for scaling, and D71 also supports image enhancements by scaler. The usage of scaler is very flexible and can be connected to layer output for layer scaling, or connected to compositor and scale the whole display frame and then feed the output data into wb_layer which will then write it into memory.

Compositor (compiz)

Compositor blends multiple layers or pixel data flows into one single display frame. its output frame can be fed into post image processor for showing it on the monitor or fed into wb_layer and written to memory at the same time. user can also insert a scaler between compositor and wb_layer to down scale the display frame first and then write to memory.

Writeback Layer (wb_layer)

Writeback layer does the opposite things of Layer, which connects to compiz and writes the composition result to memory.

Post image processor (improc)

Post image processor adjusts frame data like gamma and color space to fit the requirements of the monitor.

Timing controller (timing_ctrlr)

Final stage of display pipeline, Timing controller is not for the pixel handling, but only for controlling the display timing.

Merger

D71 scaler mostly only has the half horizontal input/output capabilities compared with Layer, like if Layer supports 4K input size, the scaler only can support 2K input/output in the same time. To achieve the ful frame scaling, D71 introduces Layer Split, which splits the whole image to two half parts and feeds them to two Layers A and B, and does the scaling independently. After scaling the result need to be fed to merger to merge two part images together, and then output merged result to compiz.

Splitter

Similar to Layer Split, but Splitter is used for writeback, which splits the compiz result to two parts and then feed them to two scalers.

Possible D71 Pipeline usage

Benefitting from the modularized architecture, D71 pipelines can be easily adjusted to fit different usages. And D71 has two pipelines, which support two types of working mode:

- Dual display mode Two pipelines work independently and separately to drive two display outputs.
- Single display mode Two pipelines work together to drive only one display output.

  On this mode, pipeline_B doesn't work indenpendently, but outputs its composition result into pipeline_A, and its pixel timing also derived from pipeline_A.timing_ctrlr. The pipeline_B works just like a "slave" of pipeline_A(master)

Single pipeline data flow

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst, line 97)

Unknown directive type "kernel-render".

```
.. kernel-render:: DOT
   :alt: Single pipeline digraph
   :caption: Single pipeline data flow

   digraph single_ppl {
      rankdir=LR;

      subgraph {
         "Memory";
         "Monitor";
      }

      subgraph cluster_pipeline {
          style=dashed
          node [shape=box]
          {
              node [bgcolor=grey style=dashed]
              "Scaler-0";
              "Scaler-1";
              "Scaler-0/1"
          }

         node [bgcolor=grey style=filled]
         "Layer-0" -> "Scaler-0"
         "Layer-1" -> "Scaler-0"
         "Layer-2" -> "Scaler-1"
         "Layer-3" -> "Scaler-1"

         "Layer-0" -> "Compiz"
         "Layer-1" -> "Compiz"
         "Layer-2" -> "Compiz"
         "Layer-3" -> "Compiz"
         "Scaler-0" -> "Compiz"
         "Scaler-1" -> "Compiz"

         "Compiz" -> "Scaler-0/1" -> "Wb_layer"
         "Compiz" -> "Improc" -> "Timing Controller"
      }

      "Wb_layer" -> "Memory"
      "Timing Controller" -> "Monitor"
   }
```

Dual pipeline with Slave enabled

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst, line 143)

Unknown directive type "kernel-render".

```
.. kernel-render:: DOT
   :alt: Slave pipeline digraph
   :caption: Slave pipeline enabled data flow

   digraph slave_ppl {
      rankdir=LR;

      subgraph {
         "Memory";
         "Monitor";
      }
      node [shape=box]
      subgraph cluster_pipeline_slave {
          style=dashed
          label="Slave Pipeline_B"
          node [shape=box]
          {
              node [bgcolor=grey style=dashed]
              "Slave.Scaler-0";
              "Slave.Scaler-1";
          }

         node [bgcolor=grey style=filled]
```

```
            "Slave.Layer-0" -> "Slave.Scaler-0"
            "Slave.Layer-1" -> "Slave.Scaler-0"
            "Slave.Layer-2" -> "Slave.Scaler-1"
            "Slave.Layer-3" -> "Slave.Scaler-1"

            "Slave.Layer-0" -> "Slave.Compiz"
            "Slave.Layer-1" -> "Slave.Compiz"
            "Slave.Layer-2" -> "Slave.Compiz"
            "Slave.Layer-3" -> "Slave.Compiz"
            "Slave.Scaler-0" -> "Slave.Compiz"
            "Slave.Scaler-1" -> "Slave.Compiz"
        }

        subgraph cluster_pipeline_master {
            style=dashed
            label="Master Pipeline_A"
            node [shape=box]
            {
                node [bgcolor=grey style=dashed]
                "Scaler-0";
                "Scaler-1";
                "Scaler-0/1"
            }

           node [bgcolor=grey style=filled]
           "Layer-0" -> "Scaler-0"
           "Layer-1" -> "Scaler-0"
           "Layer-2" -> "Scaler-1"
           "Layer-3" -> "Scaler-1"

           "Slave.Compiz" -> "Compiz"
           "Layer-0" -> "Compiz"
           "Layer-1" -> "Compiz"
           "Layer-2" -> "Compiz"
           "Layer-3" -> "Compiz"
           "Scaler-0" -> "Compiz"
           "Scaler-1" -> "Compiz"

           "Compiz" -> "Scaler-0/1" -> "Wb_layer"
           "Compiz" -> "Improc" -> "Timing Controller"
        }

        "Wb_layer" -> "Memory"
        "Timing Controller" -> "Monitor"
    }
```

Sub-pipelines for input and output

A complete display pipeline can be easily divided into three sub-pipelines according to the in/out usage.

Layer(input) pipeline

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst, line 221)

Unknown directive type "kernel-render".

```
.. kernel-render:: DOT
   :alt: Layer data digraph
   :caption: Layer (input) data flow

   digraph layer_data_flow {
      rankdir=LR;
      node [shape=box]

      {
         node [bgcolor=grey style=dashed]
           "Scaler-n";
      }

      "Layer-n" -> "Scaler-n" -> "Compiz"
   }
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst, line 237)

Unknown directive type "kernel-render".

```
.. kernel-render:: DOT
   :alt: Layer Split digraph
   :caption: Layer Split pipeline

   digraph layer_data_flow {
      rankdir=LR;
      node [shape=box]

      "Layer-0/1" -> "Scaler-0" -> "Merger"
      "Layer-2/3" -> "Scaler-1" -> "Merger"
      "Merger" -> "Compiz"
   }
```

Writeback(output) pipeline

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst, line 252)

Unknown directive type "kernel-render".

```
.. kernel-render:: DOT
   :alt: writeback digraph
   :caption: Writeback(output) data flow

   digraph writeback_data_flow {
      rankdir=LR;
      node [shape=box]

      {
         node [bgcolor=grey style=dashed]
           "Scaler-n";
      }

      "Compiz" -> "Scaler-n" -> "Wb_layer"
   }
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst, line 268)

Unknown directive type "kernel-render".

```
.. kernel-render:: DOT
   :alt: split writeback digraph
   :caption: Writeback(output) Split data flow

   digraph writeback_data_flow {
      rankdir=LR;
      node [shape=box]

      "Compiz" -> "Splitter"
      "Splitter" -> "Scaler-0" -> "Merger"
      "Splitter" -> "Scaler-1" -> "Merger"
      "Merger" -> "Wb_layer"
   }
```

Display output pipeline

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst, line 284)

Unknown directive type "kernel-render".

```
.. kernel-render:: DOT
   :alt: display digraph
   :caption: display output data flow

   digraph single_ppl {
      rankdir=LR;
      node [shape=box]

      "Compiz" -> "Improc" -> "Timing Controller"
   }
```

In the following section we'll see these three sub-pipelines will be handled by KMS-plane/wb_conn/crtc respectively.

Komeda Resource abstraction

struct komeda_pipeline/component

To fully utilize and easily access/configure the HW, the driver side also uses a similar architecture: Pipeline/Component to describe the HW features and capabilities, and a specific component includes two parts:

- Data flow controlling.
- Specific component capabilities and features.

So the driver defines a common header struct komeda_component to describe the data flow control and all specific components are a subclass of this base structure.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst, line 315)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/arm/display/komeda/komeda_pipeline.h
   :internal:
```

Resource discovery and initialization

Pipeline and component are used to describe how to handle the pixel data. We still need a @struct komeda_dev to describe the whole view of the device, and the control-abilites of device.

We have &komeda_dev, &komeda_pipeline, &komeda_component. Now fill devices with pipelines. Since komeda is not for D71 only but also intended for later products, of course weâ€™d better share as much as possible between different products. To achieve this, split the komeda device into two layers: CORE and CHIP.

- CORE: for common features and capabilities handling.
- CHIP: for register programing and HW specific feature (limitation) handling.

CORE can access CHIP by three chip function structures:

- struct komeda_dev_funcs
- struct komeda_pipeline_funcs
- struct komeda_component_funcs

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst, line 339)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/arm/display/komeda/komeda_dev.h
   :internal:
```

Format handling

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst, line 345)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/arm/display/komeda/komeda_format_caps.h
   :internal:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst, line 347)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/arm/display/komeda/komeda_framebuffer.h
   :internal:
```

Attach komeda_dev to DRM-KMS

Komeda abstracts resources by pipeline/component, but DRM-KMS uses crtc/plane/connector. One KMS-obj cannot represent only one single component, since the requirements of a single KMS object cannot simply be achieved by a single component, usually that needs multiple components to fit the requirement. Like set mode, gamma, ctm for KMS all target on CRTC-obj, but komeda needs compiz, improc and timing_ctrlr to work together to fit these requirements. And a KMS-Plane may require multiple komeda resources: layer/scaler/compiz.

So, one KMS-Obj represents a sub-pipeline of komeda resources.

- Plane: Layer(input) pipeline
- Wb_connector: Writeback(output) pipeline
- Crtc: Display output pipeline

So, for komeda, we treat KMS crtc/plane/connector as users of pipeline and component, and at any one time a pipeline/component only can be used by one user. And pipeline/component will be treated as private object of DRM-KMS; the state will be managed by drm_atomic_state as well.

How to map plane to Layer(input) pipeline

Komeda has multiple Layer input pipelines, see: - Single pipeline data flow - Dual pipeline with Slave enabled

The easiest way is binding a plane to a fixed Layer pipeline, but consider the komeda capabilities:

- Layer Split, See Layer(input) pipeline

  Layer_Split is quite complicated feature, which splits a big image into two parts and handles it by two layers and two scalers individually. But it imports an edge problem or effect in the middle of the image after the split. To avoid such a problem, it needs a complicated Split calculation and some special configurations to the layer and scaler. We'd better hide such HW related complexity to user mode.

- Slave pipeline, See Dual pipeline with Slave enabled

  Since the compiz component doesn't output alpha value, the slave pipeline only can be used for bottom layers composition. The komeda driver wants to hide this limitation to the user. The way to do this is to pick a suitable Layer according to plane_state->zpos.

So for komeda, the KMS-plane doesn't represent a fixed komeda layer pipeline, but multiple Layers with same capabilities. Komeda will select one or more Layers to fit the requirement of one KMS-plane.

Make component/pipeline to be drm_private_obj

Add :c:type:`drm_private_obj` to :c:type:`komeda_component`, :c:type:`komeda_pipeline`

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst`, line 405); *backlink*

Unknown interpreted text role "c:type".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst`, line 405); *backlink*

Unknown interpreted text role "c:type".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst`, line 405); *backlink*

Unknown interpreted text role "c:type".

```
struct komeda_component {
    struct drm_private_obj obj;
    ...
}

struct komeda_pipeline {
    struct drm_private_obj obj;
    ...
}
```

Tracking component_state/pipeline_state by drm_atomic_state

Add :c:type:`drm_private_state` and user to :c:type:`komeda_component_state`, :c:type:`komeda_pipeline_state`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst, line 422); *backlink*

Unknown interpreted text role "c:type".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst, line 422); *backlink*

Unknown interpreted text role "c:type".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst, line 422); *backlink*

Unknown interpreted text role "c:type".

```
struct komeda_component_state {
    struct drm_private_state obj;
    void *binding_user;
    ...
}

struct komeda_pipeline_state {
    struct drm_private_state obj;
    struct drm_crtc *crtc;
    ...
}
```

komeda component validation

Komeda has multiple types of components, but the process of validation are similar, usually including the following steps:

```
int komeda_xxxx_validate(struct komeda_component_xxx xxx_comp,
            struct komeda_component_output *input_dflow,
            struct drm_plane/crtc/connector *user,
            struct drm_plane/crtc/connector_state, *user_state)
{
     setup 1: check if component is needed, like the scaler is optional depending
              on the user_state; if unneeded, just return, and the caller will
              put the data flow into next stage.
     Setup 2: check user_state with component features and capabilities to see
              if requirements can be met; if not, return fail.
     Setup 3: get component_state from drm_atomic_state, and try set to set
              user to component; fail if component has been assigned to another
              user already.
     Setup 3: configure the component_state, like set its input component,
              convert user_state to component specific state.
     Setup 4: adjust the input_dflow and prepare it for the next stage.
}
```

komeda_kms Abstraction

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst, line 468)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/arm/display/komeda/komeda_kms.h
   :internal:
```

komde_kms Functions

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst, line 473)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/arm/display/komeda/komeda_crtc.c
   :internal:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\gpu\[linux-master][Documentation][gpu]komeda-kms.rst, line 475)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: drivers/gpu/drm/arm/display/komeda/komeda_plane.c
   :internal:
```

Build komeda to be a Linux module driver

Now we have two level devices:

- komeda_dev: describes the real display hardware.
- komeda_kms_dev: attachs or connects komeda_dev to DRM-KMS.

All komeda operations are supplied or operated by komeda_dev or komeda_kms_dev, the module driver is only a simple wrapper to pass the Linux command (probe/remove/pm) into komeda_dev or komeda_kms_dev.