Accélération matérielle des flux de pixels avec Linux

Paul Kocialkowski
paulk@sys-base.io

[sys-base]

Embedded Linux Meetup
2025-11-04

Éléments de contexte

Composants matériels

Pixels :

- **Affichage** : écrans, projecteurs
- **Capture** : capteurs (caméras)
- **Interfaces** : affichage, capture, transcodeurs
  - Contrôleur (logique)
  - PHY
- **Rendu** : GPU, dessin vectoriel
- **Codec** : VPU (encodeurs/décodeurs)
- **Traitement** : opérations 2D, ISP

Système :

- **Contrôle** : CPU
- **Mémoire** : DRAM, interconnect

Flux de pixels

Pipelines :

- Connexion des composants : source (tx) à sink (rx)
- Cadence d'image (imposée ou non), pipelining
- Temps de parcours (latence)

Transfert des pixels :

- Volumes de données très conséquents
- Transfert direct par FIFO/interface entre composants
  - Timings, temps morts, `vblank`/`vsync`
- Transfert indirect par la mémoire (DMA)
  - Partage des buffers entre composants, adressabilité
  - Impact négatif des copies

Mémoire

Situations :

- Mémoire dédiée ou mémoire partagée du système
- Mémoire contigue ou scatter-gather (MMU/IOMMU)
- Mémoire cohérente ou incohérente (cache)
- Interconnect, arbitre et QoS pour l'accès

Buffers :

- Format de pixel, espace de couleur
- Stride/pitch, padding et taille totale
- Tiling, compression

Synchronisation :

- Accès concurrent : r/w, w/w
  - Affichage : tearing, double-buffering (`vsync`)
- Maintenance du cache : flush, invalidate

Accélération matérielle avec Linux

Accélération matérielle avec Linux

Linux

Intégration système

Description :

- Device-tree
- ACPI
- Fwnode graph : `port/endpoint`, bidirectionnel

```
imx219: camera@10 {
  compatible = "sony,imx219";
  ...
  port {
    camera_to_bridge: endpoint {
      data-lanes = <1 2>;
      link-frequencies = /bits/ 64 <456000000>;
      remote-endpoint = <&bridge_from_camera>;
    };
  };
};
```

```
mipi_csi2: csi@1cb1000 {
  compatible = "allwinner,sun8i-v3s-mipi-csi2";
  ...
  ports {
    ...
    port@0 {
      reg = <0>;
      bridge_from_camera: endpoint {
        data-lanes = <1 2>;
        remote-endpoint = <&camera_to_bridge>;
      };
    };
  };
};
```

Intégration système

Ressources :

- Driver et device par bus : `platform`, `pci`, `usb`, `i2c`, `spi`...
- Structure de données : `struct device`
- Ressources explicites : interruptions, horloges, régulateurs, reset, dma engine
- Ressources implicites : pinmux, power domain, iommu
- Accès aux registres avec helpers par bus

Gestion de l'énergie :

- Suspend/resume : Gestion d'état
- Runtime suspend/resume, autosuspend
- Devfreq (DVFS) : gouverneur, points d'opéaration (OPP)

Gestion de la mémoire

Implémentation :

- Pages et listes de pages scatter-gather (MMU)
- Allocateur de pages contigues : `__get_free_pages`
- Adresse DMA (physique ou virtuelle) : `dma_addr_t`
- Allocations DMA Coherent : `dma_alloc_coherent`
- Mappings DMA Streaming : `dma_map_single`
- Capacité d'adressage : `dma_set_mask_and_coherent`
- Mémoire virtuelle pour le DMA (IOMMU)

Gestion de la mémoire : CMA

Implémentation :

- Prise en charge de la réservation de mémoire contigue
- Backend d'allocation pour DMA coherent
- Réservation de pools de taille fixe au démarrage
- Pool par défaut pour tous les devices :
  - Configuration par Kconfig : `CONFIG_CMA_SIZE_MBYTES`
  - Configuration par cmdline : `cma`
- Pool spécifique :
  - Déclaration par device-tree : `shared-dma-pool`
  - Association au device par device-tree : `memory-region`
- Alignement des allocations : `CONFIG_CMA_ALIGNMENT`

```
/ {
  ...
  reserved-memory {
    #address-cells = <1>;
    #size-cells = <1>;
    ranges;

    gfx_memory: framebuffer {
      compatible = "shared-dma-pool";
      size = <0x01000000>;
      alignment = <0x01000000>;
      reusable;
    };
  };
  ...
  gfx: display@1e6e6000 {
    compatible = "aspeed,ast2600-gfx";
    ...
    memory-region = <&gfx_memory>;
  };
};
```

Gestion de la mémoire : dma-buf

Implémentation :

- Prise en charge du partage de mémoire entre devices et uAPIs
- Représentation par file descriptor
- Opérations `ioctl` pour la maintenance du cache
- Mapping avec `mmap` des buffers en espace utilisateur
- uAPI d'allocation : `dma-heap`
- Synchronisation (trigger) : `dma-fence`

Accélération matérielle avec Linux

DRM

Sous-système DRM

- **1998** *(v2.1.107)* : FBdev
- **1999** *(v2.3.18)* : DRM

Implémentation :

- Gestion de l'affichage, du rendu et de l'accélération (NPU)
  - Initialement pour les cartes graphiques (GPU)
  - Gestion de mémoire vidéo dédiée
  - Configuration et exécution de tâches de rendu
- Sources dans `drivers/gpu/drm`
- Multiples évolutions et rétro-compatibilité

Mémoire avec DRM

- **2008** *(v2.6.28)* : GEM
- **2009** *(v2.6.31)* : TTM
- **2011** *(v2.6.39)* : Dumb buffer
- **2012** *(v3.4)* : DRM Prime

Implémentation :

- Mémoire dédiée : `TTM`, wrapper pour `GEM`
- Mémoire intégrée : `GEM`, backend DMA
- uAPI : dumb buffer ou spécifique au driver, handle `GEM`
- Prise en charge `dma-buf` : Prime
- Prise en charge `dma-fence` : Syncobj

DRM KMS

- **2008** *(v2.6.29)* : DRM KMS
- **2010** *(v2.6.33)* : Page flip
- **2012** *(v3.5)* : Objets et propriétés

Implémentation :

- Interface générique pour l'affichage, indépendant du rendu
- Représentation des composants internes à l'interface d'affichage (ressources)
- Topologie : `framebuffer` → `crtc` → `encoder` → `connector`
  - `Framebuffer` : association buffer et méta-données
  - `CRTC` : Génération des timings et du flux de pixels
  - `Encoder` : Conversion en signaux physiques
  - `Connecteur` : Connexion, liste des modes

DRM KMS

Mode opératoire (uAPI) :

- Character device exposé à userspace : `/dev/dri/card0`
- Opérations `ioctl` pour la configuration : `drm/drm_mode.h`
- Création de buffers avec l'API dumb buffer
- Mapping avec `mmap` des buffers en espace utilisateur
- Création de framebuffers avec GEM handle
- Formats de pixel (fourcc) : `drm/drm_fourcc.h`
  - `DRM_FORMAT_XRGB8888`
  - `DRM_FORMAT_NV12`
- Association des composants
- Configuration du mode et framebuffer du CRTC
- Cycle d'affichage :
  - Rendu du framebuffer (clipping)
  - Page flip du framebuffer (vsync)
  - Attente d'interruption (synchrone) ou non (asynchrone)
  - Retour du framebuffer précédent

DRM KMS

Mode opératoire (uAPI) :

- Propriétés pour indications supplémentaires :
  - Types : `range`, `enum`, `blob`, `bitmask`
  - Exemples : liste des modes (`crtc`), formats pris en charge (`planes`)
- Gestion du curseur :
  - Configuration de la mémoire du curseur (format unique)
  - Déplacement du curseur (x/y)
- Demande d'évènement (page flip, vblank) et attente avec `poll`
- Évènement uevent de connexion : `HOTPLUG`
- Accès exclusif pour configuration : `master`

Espace utilisateur :

- `libdrm`
- `weston`, `wlroots`, `Qt`, `SDL`

Planes DRM KMS

- **2012** *(v3.3)* : Planes
- **2014** *(v3.15)* : Universal planes

Implémentation :

- Prise en charge de la composition de plusieurs sources
- Topologie : `framebuffer` → `plane` → `crtc` → `encoder` → `connector`
  `framebuffer` → `plane` ↗
- Opération `ioctl` dédiée pour la configuration d'overlay
- Pas de synchronisation (vsync)
- Universal planes avec plusieurs types : `overlay`, `primary`, `cursor`
- Configuration spécifique avec propriétés dédiées

DRM KMS atomic

- **2015** *(v4.0)* : DRM KMS atomic

Implémentation :

- Prise en charge de la mise à jour synchronisée des objets
  - Pas d'états intermédiaires
- Liste de propriétés à appliquer aux objets : `plane`, `crtc`, `connector`
  - Configuration initiale
  - Page flip
- Opération `ioctl` unique
  - Synchonisée au vsync ou asynchrone
  - Bloquante ou non-bloquante
  - Essai de configuration
- State interne maintenue par le core

Driver DRM KMS

- Structure de données : `struct drm_driver`
- Fonctionnalités : `DRIVER_MODESET`, `DRIVER_ATOMIC`, `DRIVER_GEM`
- Backend GEM : `DRM_GEM_DMA_DRIVER_OPS` ou spécifique
- Structure de données : `struct drm_device`
- Création du device DRM : `drm_dev_register`
- Callbacks de configuration atomique pour les objets
- Lecture de l'EDID pour la liste des modes
- Interruption `vblank`
- Wrapper simplifié : `DRM simple/tiny DRM` (écrans sur interface série)

DRM panel

- **2014** *(v3.14)* : DRM panel

Implémentation :

- Prise en charge de panels sans EDID
- Driver et device dédiés, transparent pour l'espace utilisateur
- Lien avec le device DRM (fwnode graph) : `drm_of_find_panel_or_bridge`
- Liste de modes codée en dur
- Configuration initiale du panel
- Association du rétro-éclairage
- Exemples : `panel-simple`, `panel-lvds`, `panel-edp`

DRM bridge

- **2013** *(v3.12)* : DRM bridge

Implémentation :

- Prise en charge des transcodeurs (visibles) après l'encodeur
- Driver et device dédiés, transparent pour l'espace utilisateur
- Lien avec le device DRM (fwnode graph) : `drm_of_find_panel_or_bridge`
- Enregistrement du connecteur final
- Détection de la connexion, EDID
- Configuration des interfaces et du mode
- Mise en chaîne de plusieurs bridges
- Wrapper pour `panel` (simplification) : `devm_drm_of_get_bridge`
- Exemples : `simple-bridge`, `lvds-codec`, `display-connector`

DRM render

- **2013** *(v3.12)*: Render node
- **2018** *(v4.16)*: Scheduler

Implémentation :

- Prise en charge des GPUs
- Render node : `/dev/dri/render128`
- Opérations `ioctl` spécifiques aux drivers
- Validation du command stream
- Configuration, trigger et interruption
- Scheduling entre applications : scheduler DRM

Espace utilisateur :

- `libdrm`
- `mesa`

Accélération matérielle avec Linux

V4L2

Sous-système media

- **1997** *(v2.1.67)*: V4L
- **2002** *(v2.5.7)*: Videobuf
- **2002** *v2.5.45*: DVB
- **2002** *(v2.5.46)*: V4L2
- **2008** *v2.6.28*: DVB v5 (S2API)
- **2010** *(v2.6.39)*: Videobuf2

Implémentation :

- Sous-système `media` : frameworks `v4l2`, `dvb`, `lirc`, `cec`
- Prise en charge des fonctionnalités multimedia (hors affichage et rendu)
  - Initialement capture et envoi de flux vidéo
  - Configuration et mise à disposition des données
- Sources dans `drivers/media`
- Multiples évolutions et rétro-compatibilité

Video device

Mode opératoire (uAPI) :

- Character device exposé à userspace : `/dev/video0`
- Opérations `ioctl` pour la configuration : `linux/videodev2.h`
- Queue de buffers avec type :
  - `V4L2_BUF_TYPE_VIDEO_CAPTURE(_MPLANE)`
  - `V4L2_BUF_TYPE_VIDEO_OUTPUT(_MPLANE)`
- Configuration du format de la queue : `struct v4l2_pix_format(_mplane)`
- Formats de pixel (fourcc) :
  - `V4L2_PIX_FMT_RGB24`
  - `V4L2_PIX_FMT_NV12(M)`
  - `V4L2_PIX_FMT_YUV420(M)`
- Configuration des contrôles avec type:
  - `V4L2_CTRL_TYPE_INTEGER`
  - `V4L2_CTRL_TYPE_MENU`
  - `V4L2_CTRL_TYPE_BUTTON`

Video device

Mode opératoire (uAPI) :

- Creation des buffers avec type :
  - `V4L2_MEMORY_MMAP`
  - `V4L2_MEMORY_USERPTR`
  - `V4L2_MEMORY_DMABUF`
- Mapping avec `mmap` des buffers en espace utilisateur
- Démarrage et arrêt du flux par queue
- Cycle de capture du flux :
  - Mise en queue des buffers inutilisés
  - Attente de l'interruption avec `poll`
  - Retour d'un buffer (rempli) de la queue

Espace utilisateur :

- `GStreamer`, `FFmpeg`
- Nombreuses applications vidéo

Drivers video device

- Structure de données : `struct video_device`
- Création du video device : `video_register_device`
- Callbacks de configuration pour le syscall `ioctl` : `struct v4l2_ioctl_ops`
- Callbacks pour les contrôles : `struct v4l2_ctrl_ops`
- Callbacks pour la queue : `struct vb2_ops`
- Sélection de l'implémentation `videobuf2` :
  - `vb2_dma_contig_memops` : DMA coherent et contigu
  - `vb2_dma_sg_memops` : DMA straming (IOMMU)
  - `vb2_vmalloc_memops` : pas de DMA
- Réception des buffers, DMA et interruption
- Nombre minimum de buffers
- Informations sur le format de pixel : `v4l2_format_info`

V4L2 subdev et device

- **2009** *(v2.6.29)* : V4L2 subdev et V4L2 device

Implémentation :

- Prise en charge de plusieurs composants chaînés
- V4L2 subdev : composant sans DMA (capteurs, interfaces, ISPs)
- V4L2 device : parent des composants
- Subdev et video device associés à un V4L2 device
- Création et notification asynchrone entre drivers (fwnode graph)
- Appel chaîné des opérations

V4L2 subdev et device

Mode opératoire (uAPI) :

- Character device exposé à userspace: `/dev/v4l-subdev0`
- Opérations `ioctl` pour la configuration : `linux/v4l2-subdev.h`
- Configuration dédiée des contrôles (mêmes operations `ioctl`)
- Pas de queue/buffer et gestion du flux

Espace utilisateur :

- `v4l-utils`: `v4l2-ctl --help-subdev`
  `media-ctl --set-v4l2`

Drivers V4L2 subdev et device

V4L2 subdev :

- Structure de données : `struct v4l2_subdev`
- Création directe : `v4l2_device_register_subdev`
- Création asynchrone : `v4l2_async_register_subdev`
- Callbacks de configuration des pads : `struct v4l2_subdev_pad_ops`
  - Format, taille, fréquence, EDID, timings DV, routage interne, streams
- Callbacks de gestion du flux video : `struct v4l2_subdev_video_ops`
  - Standard, source, stream
- Callbacks pour les contrôles : `struct v4l2_ctrl_ops`
- Formats de bus: `MEDIA_BUS_FMT_SBGGR8_1X8`

V4L2 device :

- Structure de données : `struct v4l2_device`
- Création du V4L2 device : `v4l2_device_register`

Intégration V4L2 subdev

- Topologie des drivers :
  - Parent : V4L2 device (et video device)
  - Enfant : V4L2 subdev et/ou video device
- Notification asynchrone des subdevs :
  - Identification des composants (fwnode graph) : `v4l2_async_nf_add_fwnode_remote`
  - Initialisation avec V4L2 device : `v4l2_async_nf_init`
  - Initialisation avec V4L2 subdev : `v4l2_async_subdev_nf_init`
  - Création : `v4l2_async_nf_register`
  - Callbacks d'association : `v4l2_async_notifier_operations`
  - Device V4L2 parent associé au subdev par le core
- Création des noeuds subdev pour userspace : `v4l2_device_register_subdev_nodes`
- Appel chaîné des opérations : `v4l2_subdev_call`
  - Démarrage et arrêt du flux par le video device

Media Controller

- **2010** *(v2.6.39)* : Media Controller

Implémentation :

- Représentation et configuration des liens entre composants (et drivers)
- Topologie : `entity`, `pad`, `link`, `interface`
- Media device : parent de la topologie
- Activation des liens entre les pads des composants
- Vérification des liens au démarrage du flux
- Entités : video device, V4L2 subdev
- Device V4L2 associé à un media device

Media Controller

Mode opératoire (uAPI) :

- Character device exposé à userspace: `/dev/media0`
- Opérations `ioctl` pour la topologie et l'activation des liens : `linux/media.h`
- Fonctions des composants : `MEDIA_ENT_F_CAM_SENSOR`
- Types des interfaces : `MEDIA_INTF_T_V4L_VIDEO`

Espace utilisateur :

- `v4l-utils`: `media-ctl -p`
  `media-ctl -l`

Drivers Media Controller

Media device :

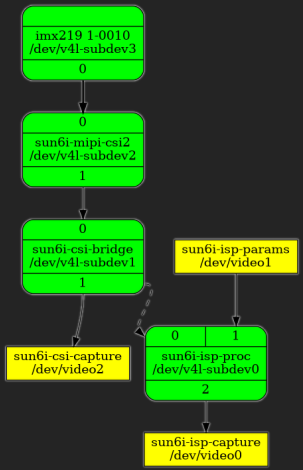
- Structure de données : `struct media_device`
- Création du media device : `media_device_register`
- Callback (`link_validate`) : `struct media_entity_operations`
- Démarrage de la pipeline : `media_pipeline_alloc_start`

Subdev / video device :

- Créátion des entités : `media_device_register_entity`
- Création des interfaces : `media_devnode_create`
- Créátion des liens : `media_create_pad_link`, `media_create_intf_link`

Example

- `imx219` : capteur MIPI CSI-2
  - Subdev `/dev/v4l-subdev3`
- `sun6i-mipi-csi2` : interface MIPI CSI-2
  - Subdev `/dev/v4l-subdev2`
- `sun6i-csi` : interface de capture
  - Video device `/dev/video2`
  - Subdev `/dev/v4l-subdev1`
- `sun6i-isp` : ISP
  - V4L2 et media device
  - Video device `/dev/video0`
  - Video device `/dev/video1`
  - Subdev `/dev/v4l-subdev0`



ISP avec V4L2

Mode opératoire (uAPI) :

- Subdev avec fonction : `MEDIA_ENT_F_PROC_VIDEO_ISP`
- Video devices distincts pour paramètres et statistiques
- Types de queues spécifiques :
  - `V4L2_BUF_TYPE_META_CAPTURE`
  - `V4L2_BUF_TYPE_META_OUTPUT`
- Formats de méta-données spécifiques aux drivers :
  - `V4L2_META_FMT_RK_ISP1_PARAMS`
  - `V4L2_META_FMT_RK_ISP1_EXT_PARAMS`
  - `V4L2_META_FMT_RK_ISP1_STAT_3A`

Espace utilisateur :

- `libcamera`

V4L2 M2M

- **2010** *(v2.6.35)* : V4L2 mem2mem

Implémentation :

- Prise en charge d'opérations en mémoire (opérations 2D, codecs)
- Video device unique avec deux queues (capture et output)
- Device V4L2 M2M global
- Contexte spécifique à chaque utilisateur
- Exécution de jobs avec contexte et queues

Drivers :

- Structure de données : `struct v4l2_m2m_dev`
- Création : `v4l2_m2m_init`
- Création du contexte : `v4l2_m2m_ctx_init`
- Callbacks pour l'exécution (`device_run`) : `struct v4l2_m2m_ops`
- Finalisation du job (irq) : `v4l2_m2m_job_finish`

Codecs stateful avec V4L2

Implémentation :

- Prise en charge du décodage/encodage vidéo avec microcontrolleur
- Device V4L2 M2M et video device
- Formats de pixel spécifiques :
  - `V4L2_PIX_FMT_H264`
  - `V4L2_PIX_FMT_HEVC`
- Méta-données brutes avec les données codées
- Contrôles pour les options d'encodage

Espace utilisateur :

- `GStreamer`, `FFmpeg`

Codecs stateless avec V4L2

Implémentation :

- Prise en charge du décodage vidéo sans microcontrolleur
- Device V4L2 M2M et video device
- Formats de pixel spécifiques :
  - `V4L2_PIX_FMT_H264_SLICE`
  - `V4L2_PIX_FMT_HEVC_SLICE`
- Contrôles pour les méta-données décodées et structurées
- Association des contrôles et buffers avec une media request
- Pas (encore) de prise en charge de l'encodage

Espace utilisateur :

- `GStreamer`

Merci pour votre attention !