The Global GTT [Part 1]

Author : Ben Widawsky

Contents

Global Graphics Translation Tables

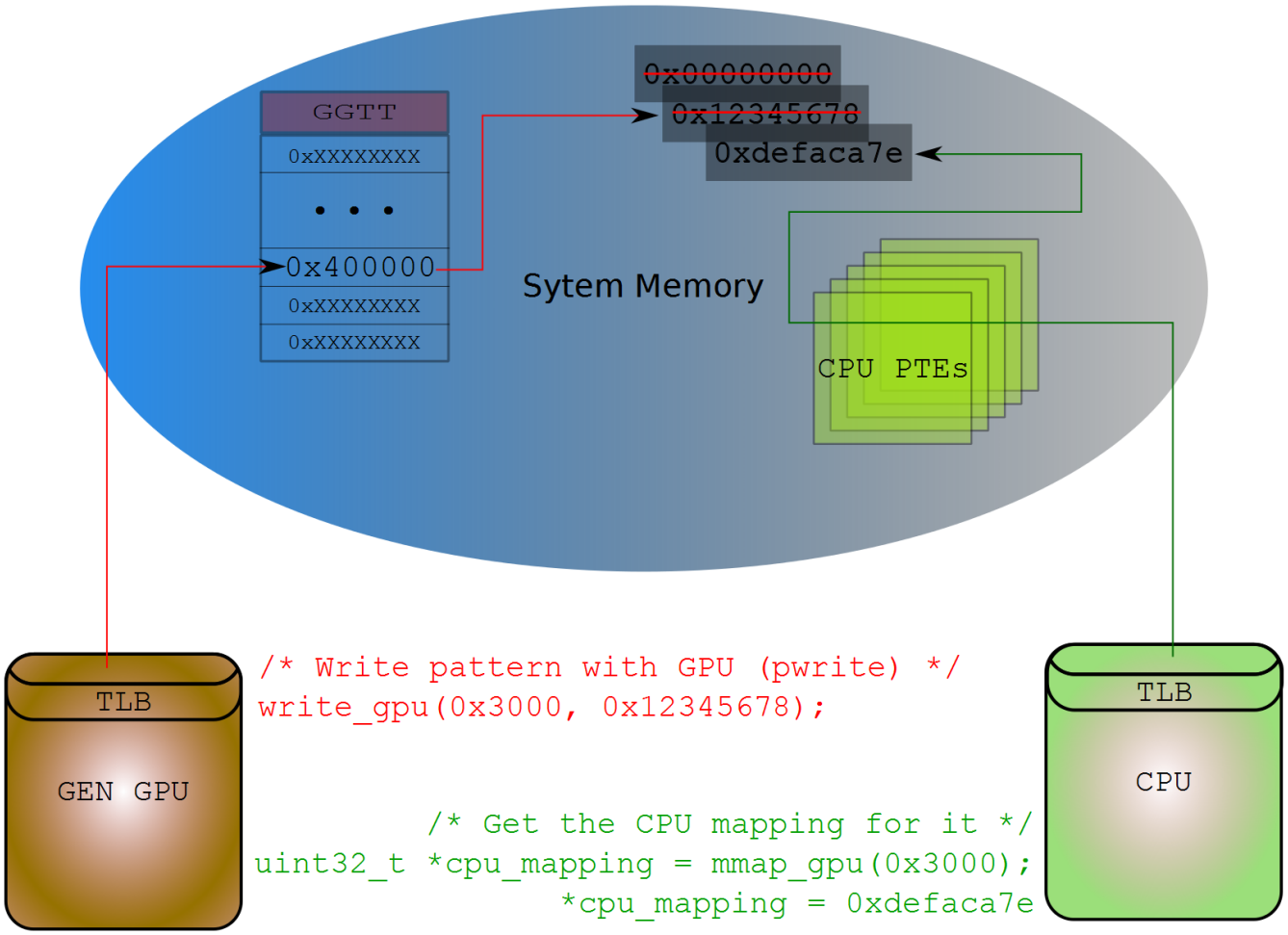
Here goes the basics of how the GEN GPU interacts with memory. It will be focused on the lowest levels of the i915 driver, and the hardware interaction. My hope is that by going through this in excruciating detail, I might be able to take more liberties in the future posts.

What are the Global Graphics Translation Table

The graphics translation tables provide the address mapping from the GPU's virtual address space to a physical address[1]. The GTT is somewhat a relic of the AGP days (GART) with the distinction being that the GTT as it pertains to Intel GEN GPUs has logic that is contained within the GPU, and does not act as a platform IOMMU. I believe (and wikipedia seems to agree) that GTT and GART were used interchangeably in the AGP days.

GGTT architecture

Each element within the GTT is an entry, and the initialism for each entry is a, "PTE" or page table entry. Much of the required initialization is handled by the boot firmware. The i915 driver will get any required information from the initialization process via PCI config space, or MMIO.



Example illustrating Intel/GEN memory organization:

Location

The table is located within system memory, and is allocated for us by the BIOS or boot firmware. To clarify the docs a bit, GSM is the portion of stolen memory for the GTT, DSM is the rest of stolen memory used for misc things. DSM is the stolen memory referred to by the current i915 code as “stolen memory.” In theory we can get the location of the GTT from MMIO MPGFXTRK_CR_MBGSM_0_2_0_GTTMMADR (0×108100, 31:20), but we do not do that. The register space, and the GTT entries are both accessible within BAR0 (GTTMMADR).

All the information can be found in Volume 12, p.129: UNCORE_CR_GTTMMADR_0_2_0_PCI. Quoting directly from the HSW spec, “The range requires 4 MB combined for MMIO and Global GTT aperture, with 2MB of that used by MMIO and 2MB used by GTT. GTTADR will begin at GTTMMADR 2 MB while the MMIO base address will be the same as GTTMMADR.”

In the below code you can see we take the address in the PCI BAR and add half the length to the base. For all modern GENs, this is how things are split in the BAR.

```
  /* For Modern GENs the PTEs and register space are split in the BAR
*/  gtt_phys_addr = pci_resource_start(dev->pdev, 0) +   (pci_resource
_len(dev->pdev, 0) / 2);    dev_priv->gtt.gsm = ioremap_wc(gtt_phys_ad
dr, gtt_size);
```

One important thing to notice above is that the PTEs are mapped in a write-combined fashion. Write combining makes sequential updates (something which is very common when mapping objects) significantly faster. Also, the observant reader might ask, ‘why go through the BAR to update the PTEs if we have the actual physical memory location.’ This is the only way we have to make sure the GPUs TLBs get synchronized properly on PTE updates. If this weren’t required, a nice optimization might be to update all the entries as once with the CPU, and then go tell the GPU to invalidate the TLBs.

Size

Size is a bit more straight forward. We just read the relevant PCI offset. In the docs: p.151 GSA_CR_MGGC0_0_2_0_PCI offset 0×50, bits 9:8

And the code is even more straightforward.

```
 static inline unsigned int gen6_get_total_gtt_size(u16 snb_gmch_ctl)
{         snb_gmch_ctl >>= SNB_GMCH_GGMS_SHIFT;         snb_gmch_ctl
&= SNB_GMCH_GGMS_MASK;         return snb_gmch_ctl pdev,
SNB_GMCH_CTRL, &snb_gmch_ctl); gtt_size =
gen6_get_total_gtt_size(snb_gmch_ctl); gtt_total = (gtt_size /
sizeof(gen6_gtt_pte_t)) = (gtt_total / 4); /* We can only use 38
address bits */ BUG_ON(address >= (1
```