Future PPGTT [part 4] (Dynamic page table allocations, 64 bit address space, GPU "mirroring", and yeah, something about relocs too)

Author : Ben Widawsky

Contents

Preface

GPU mirroring provides a mechanism to have the CPU and the GPU use the same virtual address for the same physical (or IOMMU) page. An immediate result of this is that relocations can be eliminated. There are a few derivative benefits from the removal of the relocation mechanism, but it really all boils down to that. Other people call it other things, but I chose this name before I had heard other names. SVM would probably have been a better name had I read the OCL spec sooner. This is not an exclusive feature restricted to OpenCL. Any GPU client will hopefully eventually have this capability provided to them.

If you're going to read any single PPGTT post of this series, I think it should **not** be this one. I was not sure I'd write this post when I started documenting the PPGTT (part 1, part2, part3). I had hoped that any of the following things would have solidified the decision by the time I completed part3.

1. CODE: The code is not not merged, not reviewed, and not tested (by anyone but me). There's no indication about the "upstreamability". What this means is that if you read

my blog to understand how the i915 driver currently works, you'll be taking a crap-shoot on this one.
2. DOCS: The Broadwell public Programmer Reference Manuals are not available. I can't refer to them directly, I can only refer to the code.
3. PRODUCT: Broadwell has not yet shipped. My ulterior motive had always been to rally the masses to test the code. Without product, that isn't possible.

Concomitant with these facts, my memory of the code and interesting parts of the hardware it utilizes continues to degrade. Ultimately, I decided to write down what I can while it's still fresh (for some very warped definition of "fresh").

Goal

GPU mirroring is the goal. Dynamic page table allocations are very valuable by itself. Using dynamic page table allocations can dramatically conserve system memory when running with multiple address spaces (part 3 if you forgot), which is something which should become pretty common shortly. Consider for a moment a Broadwell legacy 32b system (more details later). TYou would require about 8MB for page tables to map one page of system memory. With the dynamic page table allocations, this would be reduced to 8K. Dynamic page table allocations are also an indirect requirement for implementing a 64b virtual address space. Having a 64b virtual address space is a pretty unremarkable feature by itself. On current workloads [that I am aware of] it provides no real benefit. Supporting 64b did require cleaning up the infrastructure code quite a bit though and should anything from the series get merged, and I believe the result is a huge improvement in code readability.

Current Status

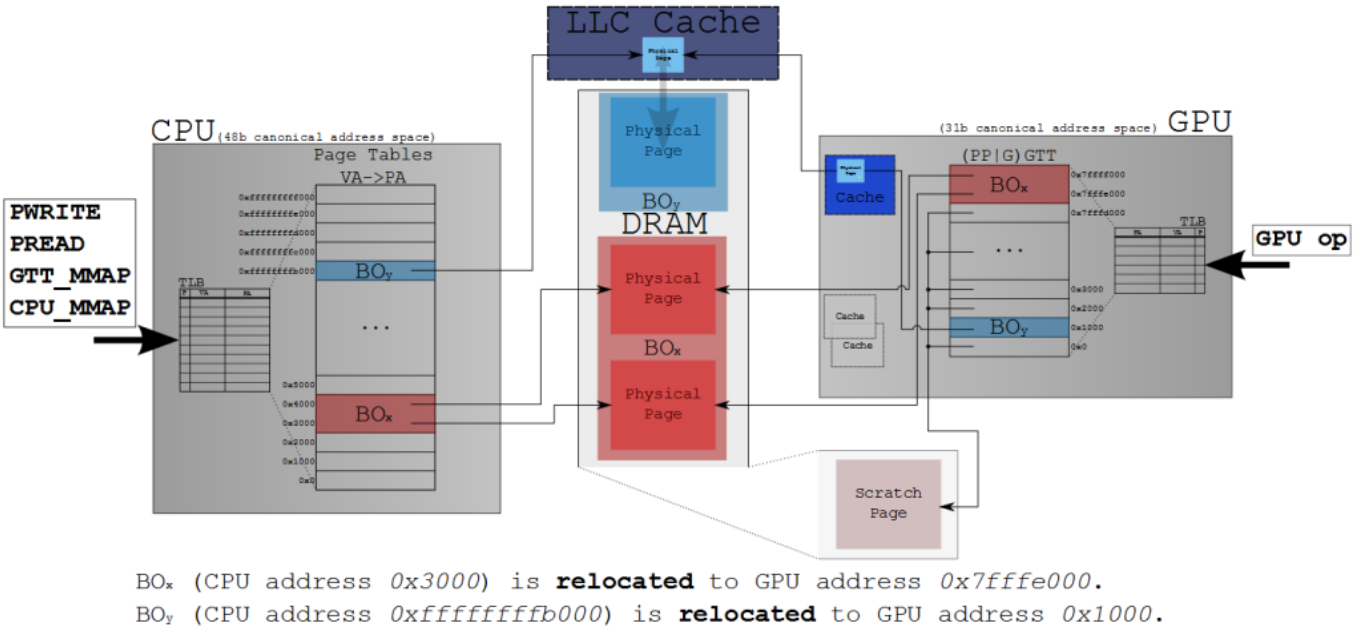
I briefly mentioned dogfooding these several months ago. At that time I only had the dynamic page table allocations on GEN7 working. The fallout wasn't nearly as bad as I was expecting, but things were far from stable. There was a second posting which is much more stable and contains support of everything through Broadwell. To summarize:

Feature	Status	TODO
Dynamic page tables	Implemented	Test and fix bugs
64b Address space	Implemented	Test and fix bugs
GPU mirroring	Proof of Concept	Decide on interface; Implement interface.[1]

Testing has been limited to just one machine, mine, when I don't have a million other things to do. With that caveat, on top of my last PPGTT stabilization patches things look pretty stable.

Present: Relocations

Throughout many of my previous blog posts I've gone out of the way to avoid explaining relocations. My reluctance was because explaining the mechanics is quite tedious, not because it is a difficult concept. It's impossible [and extremely unfortunate for my weekend] to make the case for why these new PPGTT features are cool without touching on relocations at least a little bit. The following picture exemplifies both the CPU and GPU mapping the same pages with the current relocation mechanism.



Current PPGTT support

To get to the above state, something like the following would happen.

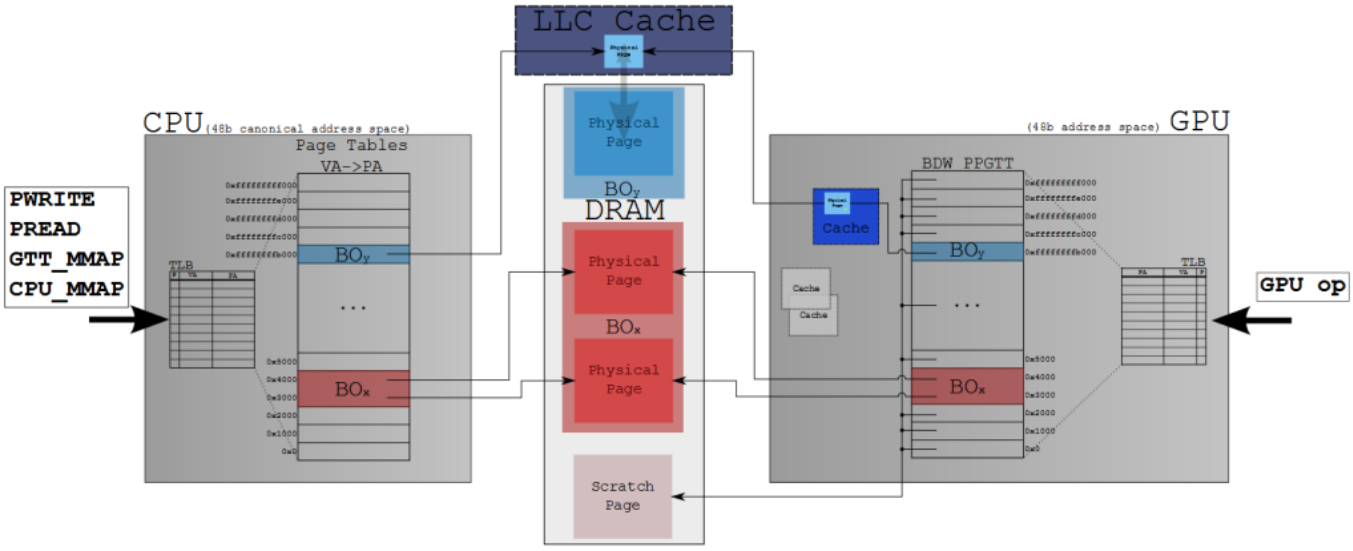
1. Create BOx
2. Create BOy
3. Request BOx be uncached via (IOCTL DRM_IOCTL_I915_GEM_SET_CACHING).
4. Do one of aforementioned operations on BOx and BOy
5. Perform execbuf2.

Accesses to the BO from the CPU require having a CPU virtual address that eventually points to the pages representing the BO[2]. The GPU has no notion of CPU virtual addresses (unless you have a bug in your code). Inevitably, all the GPU really cares about is physical pages; which ones. On the other hand, userspace needs to build up a set of GPU commands which sometimes need to be aware of the absolute graphics address.

Several commands do not need an absolute address. 3DSTATE_VS for instance **does not** need to know anything about where Scratch Space Base Offset
is actually located. It needs to provide an offset to the General State Base Address. The General State Base Address **does** need to be known by userspace:
STATE_BASE_ADDRESS

Using the relocation mechanism gives userspace a way to inform the i915 driver about the BOs which needs an absolute address. The handles plus some information about the GPU commands that need absolute graphics addresses are submitted at execbuf time. The kernel will make a GPU mapping for all the pages that constitute the BO, process the list of GPU commands needing update, and finally submit the work to the GPU.

Future: No relocations



GPU Mirroring

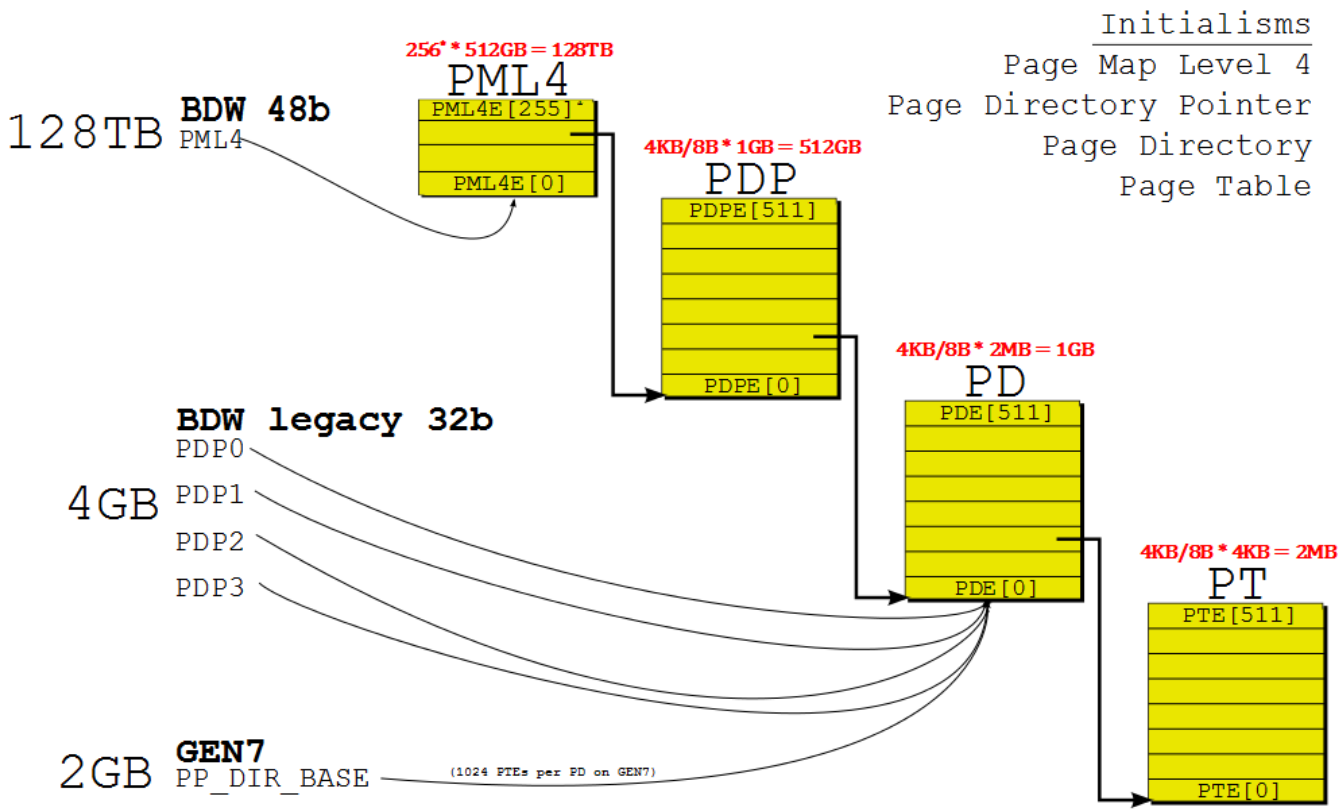
The diagram above demonstrates the goal. Symmetric mappings to a BO on both the GPU and the CPU. There are benefits for ditching relocations. One of the nice side effects of getting rid of relocations is it allows us to drop the use of the DRM memory manager and simply rely on malloc as the address space allocator. The DRM memory allocator does not get the same amount of attention with regard to performance as malloc does. Even if it did perform as ideally as possible, it's still a superfluous CPU workload. Other people can probably explain the CPU overhead in better detail. Oh, and OpenCL 2.0 requires it.

```
"OpenCL 2.0 adds support for shared virtual memory (a.k.a. SVM). SVM a
llows the host and  kernels executing on devices to directly share co
mplex, pointer-containing data structures such  as trees and linked l
ists. It also eliminates the need to marshal data between the host and
 devices.  As a result, SVM substantially simplifies OpenCL programmi
ng and may improve performance."
```

Makin’ it Happen

64b

As I’ve already mentioned, the most obvious requirement is expanding the GPU address space to match the CPU.



Page Table Hierarchy

If you have taken any sort of Operating Systems class, or read up on Linux MM within the last 10 years or so, the above drawing should be incredibly unremarkable. If you have not, you’re probably left with a big ‘WTF’ face. I probably can’t help you if you’re in the latter group, but I do sympathize. For the other camp: Broadwell brought 4 level page tables that work exactly how you’d expect them to. Instead of the x86 CPU’s CR3, GEN GPUs have PML4. When operating in legacy 32b mode, there are 4 PDP registers that each point to a page directory and therefore map 4GB of address space[3]. The register is just a simple logical address pointing to a page directory. The actual changes in hardware interactions are trivial on top of all the existing PPGTT work.

The keen observer will notice that there are only 256 PML4 entries. Th is has to do with the way in which we've come about 64b addressing in x86. This wikipedia article explains it pretty well, and has links.

“This will take one week. I can just allocate everything up front.” (Dynamic Page Table Allocation)

Funny story. I was asked to estimate how long it would take me to get this GPU mirror stuff in shape for a very rough proof of concept. “One week. I can just allocate everything up front.” If what I have is, “done” then I was off by 10x.

Where I went wrong in my estimate was math. If you consider the above, you quickly see why allocating everything up front is a terrible idea and flat out impossible on some systems.

```
Page for the PML4  512 PDP pages per PML4 (512, ok we actually use 256
)  512 PD pages per PDP (256 * 512 pages for PDs)  512 PT pages per PD
 (256 * 512 * 512 pages fo
r PTs)  (256 * 512^2 + 256 * 512 + 256 + 1) * PAGE_SIZE = ~256G = oops
```

Dissimilarities to x86

First and foremost, there are no GPU page faults to speak of. We cannot demand allocate anything in the traditional sense. I was naive though, and one of the first thoughts I had was: the Linux kernel [heck, just about everything that calls itself an OS] manages 4 level pages tables on multiple architectures. The page table format on Broadwell is remarkably similar to x86 page tables. If I can’t use the code directly, surely I can copy. Wrong.

Here is some code from the Linux kernel which demonstrates how you can get a PTE for a given address in Linux.

```
  typedef unsigned long   pteval_t;  typedef struct { pteval_t pte; }
pte_t;    static inline pteval_t native_pte_val(pte_t pte)  {
 return pte.pte;  }    static inline pteval_t pte_flags(pte_t pte)  {
         return native_pte_val(pte) & PTE_FLAGS_MASK;  }    static inl
ine int pte_present(pte_t a)  {          return pte_flags(a) & (_PAGE_
PRESENT | _PAGE_PROTNONE |                                    _PAGE_NUMA)
;  }  static inline pte_t *pte_offset_kernel(pmd_t *pmd, unsigned long
 address)  {          return (pte_t *)pmd_page_vaddr(*pmd) + pte_index
(address);  }  #define pte_offset_map(dir, address) pte_offset_kernel(
(dir), (address))    #define pgd_offset(mm, address) ( (mm)->pgd + pgd
_index((address)))  static inline pud_t *pud_offset(pgd_t *pgd, unsign
ed long address)  {          return (pud_t *)pgd_page_vaddr(*pgd) + pu
d_index(address);  }  static inline pmd_t *pmd_offset(pud_t *pud, unsi
gned long address)  {          return (pmd_t *)pud_page_vaddr(*pud) +
pmd_index(address);  }    /* My completely fabricated example of findi
ng page presence */  pgd_t *pgd;  pud_t *pud;  pmd_t *pmd;  pte_t *pte
p;  struct mm_struct *mm = current->mm;  unsigned long address = 0xdef
```

```
eca7e;    pgd = pgd_offset(mm, address);  pud = pud_offset(pgd, addres
s);  pmd = pmd_offset(pud, address);  ptep = pte_offset_map(pmd, addre
ss);  printk("Page is present: %s\n", pte_present(*ptep) ? "yes" : "no
");
```

X86 page table code has a two very distinct property that does not exist here (warning, this is slightly hand-wavy).

1. The kernel knows exactly where in physical memory the page tables reside[4]. On x86, it need only read CR3. We don't know where our pages tables reside in physical memory because of the IOMMU. When VT-d is enabled, the i915 driver only knows the DMA address of the page tables.
2. There is a strong correlation between a CPU process and an mm (set of page tables). Keeping mappings around of the page tables is easy to do if you don't want to take the hit to map them every time you need to look at a PTE.

If the Linux kernel needs to find if a page is present or not without taking a fault, it need only look to one of those two options. After about of week of making the IOMMU driver do things it shouldn't do, and trying to push the square block through the round hole, I gave up on reusing the x86 code.

Why Do We Actually Need Page Table Tracking?

The IOMMU interfaces were not designed to pull a physical address from a DMA address. Pre-allocation is right out. It's difficult to try to get the instantaneous state of the page tables…

Another thought I had very early on was that tracking could be avoided if we just never tore down page tables. I knew this wasn't a good solution, but at that time I just wanted to get the thing working and didn't really care if things blew up spectacularly after running for a few minutes. There is actually a really easy set of operations that show why this won't work. For the following, think of the four level page tables as arrays. ie.

- PML4[0-255], each point to a PDP
- PDP[0-255][0-511], each point to a PD
- PD[0-255][0-511][0-511], each point to a PT
- PT[0-255][0-511][0-511][0-511] (where PT[0][0][0][0][0] is the 0th PTE in the system)

1. [mesa] Create a 2M sized BO. Write to it. Submit it via execbuffer
2. [i915] See new BO in the execbuffer list. Allocate page tables for it…
    1. [DRM]Find that address 0 is free.
    2. [i915]Allocate PDP for PML4[0]
    3. [i915]Allocate PD for PDP[0][0]
    4. [i915]Allocate PT for PD[0][0][0]/li>

        5. [i915](condensed)Set pointers from PML4->PDP->PD->PT
        6. [i915]Set the 512 PTEs PT[0][0][0][0][511-0] to point to the BO’s backing page.
3. [i915] Dispatch work to the GPU on behalf of mesa.
4. [i915] Observe the hardware has completed
5. [mesa] Create a 4k sized BO. Write to it. Submit both BOs via execbuffer.
6. [i915] See new BO in the execbuffer list. Allocate page tables for it…
        1. [DRM]Find that address 0×200000 is free.
        2. [i915]Allocate PDP[0][0], PD[0][0][0], PT[0][0][0][1].
        3. Set pointers… Wait. Is PDP[0][0] allocated already? Did we already set pointers? I have no freaking idea!
        4. Abort.

Page Tables Tracking with Bitmaps

Okay. I could have used a sentinel for empty entries. It is possible to achieve this same thing by using a sentinel value (point the page table entry to the scratch page). To implement this involves reading back potentially large amounts of data from the page tables which will be slow. It *should* work though. I didn’t try it.

After I had determined I couldn’t reuse x86 code, and that I need some way to track which page table elements were allocated, I was pretty set on using bitmaps for tracking usage. The idea of a hash table came and went – none of the upsides of a hash table are useful here, but all of the downsides are present(space). Bitmaps was sort of the default case. Unfortunately though, I did some math at this point, notice the LaTex!.

That’s 4GB simply to track every page. There’s some more overhead because page [tables, directories, directory pointers] are also tracked.

I can’t remember whether I had planned to statically pre-allocate the bitmaps, or I was so caught up in the details and couldn’t see the big picture. I remember thinking, 4GB just for the bitmaps, that will never fly. I probably spent a week trying to figure out a better solution. When we invent time travel, I will go back and talk to my former self: 4GB of bitmap tracking if you’re using 128TB of memory is inconsequential. That is 0.3% of the memory used by the GPU. Hopefully you didn’t fall into that trap, and I just wasted your time, but there it is anyway.

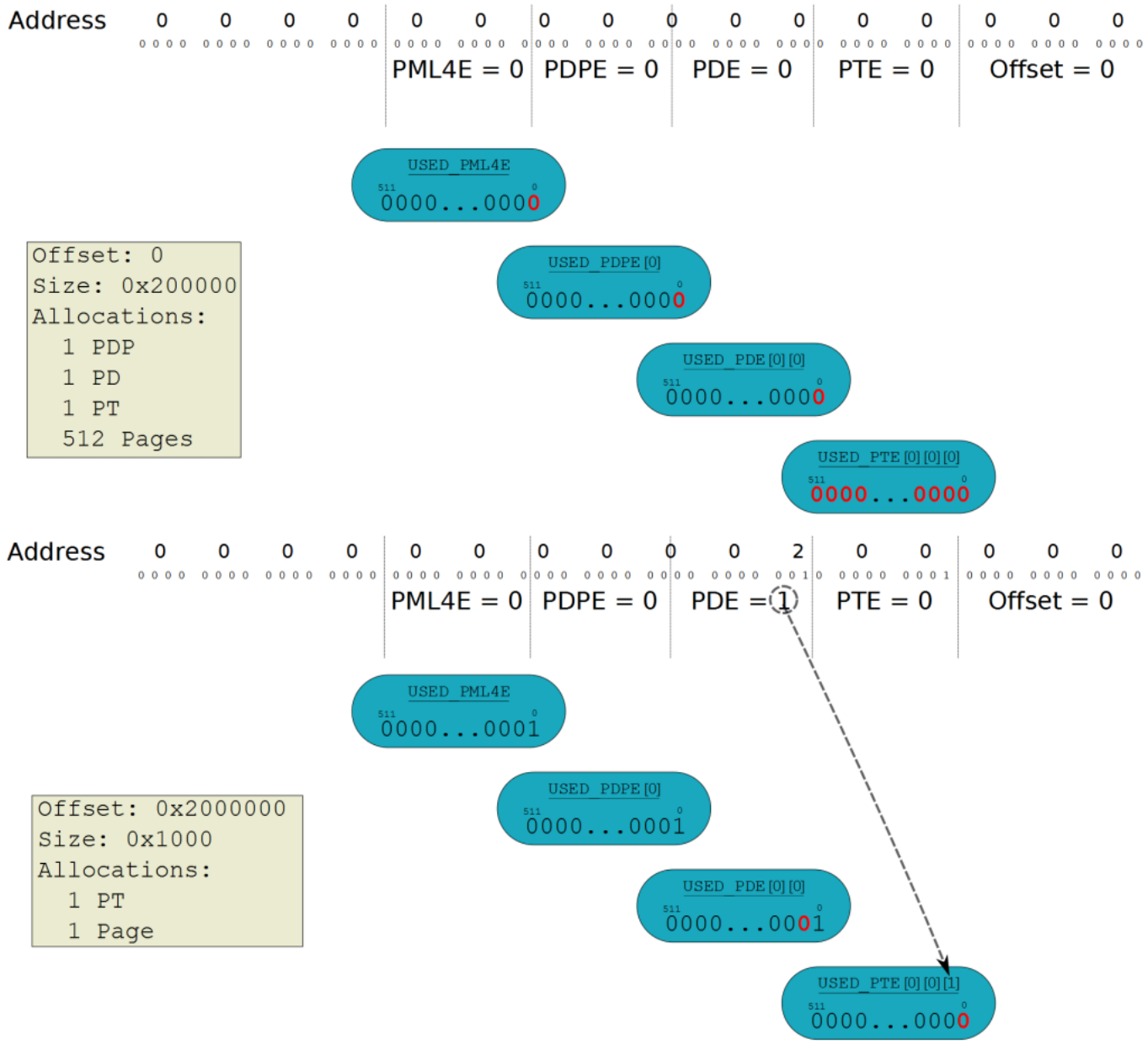
Sample code to walk the page tables

This code does not actually exist, but it is very similar to the real code. The following shows how one would “walk” to a specific address allocating the necessary page tables and setting the

bitmaps along the way. Teardown is a bit harder, but it is similar.

```
  static struct i915_pagedirpo *  alloc_one_pdp(struct i915_pml4 *pml4
, int entry)  {   ...  }    static struct i915_pagedir *  alloc_one_pd
(struct i915_pagedirpo *pdp, int entry)  {   ...  }    static struct i
915_tab *  alloc_one_pt(struct i915_pagedir *pd, int entry)  {   ...
}    /**   * alloc_page_tables - Allocate all page tables for the give
n virtual address.   *   * This will allocate all the necessary page t
ables to map exactly one page at   * @address. The page tables will no
t be connected, and the PTE will not point   * to a page.   *   * @ppg
tt: The PPGTT structure encapsulating the virtual address space.   * @
address: The virtual address for which we want page tables.   *   */
static void  alloc_page_tables(ppgtt, unsigned long address)  {   stru
ct i915_pagetab *pt;   struct i915_pagedir *pd;   struct i915_pagedirp
o *pdp;   struct i915_pml4 *pml4 = &ppgtt->pml4; /* Always there */
  int pml4e = (address >> GEN8_PML4E_SHIFT) & GEN8_PML4E_MASK;   int p
dpe = (address >> GEN8_PDPE_SHIFT) & GEN8_PDPE_MASK;   int pde = (addr
ess >> GEN8_PDE_SHIFT) & I915_PDE_MASK;   int pte = (address & I915_PD
ES_PER_PD);     if (!test_bit(pml4e, pml4->used_pml4es))    goto alloc
;     pdp = pml4->pagedirpo[pml4e];   if (!test_bit(pdpe, pdp->used_pd
pes;))    goto alloc;     pd = pdp->pagedirs[pdpe];   if (!test_bit(pd
e, pd->used_pdes)    goto alloc;     pt = pd->page_tables[pde];   if (
test_bit(pte, pt->used_ptes))    return;    alloc_pdp:   pdp = alloc_o
ne_pdp(pml4, pml4e);   set_bit(pml4e, pml4->used_pml4es);  alloc_pd:
 pd = alloc_one_pd(pdp, pdpe);   set_bit(pdpe, pdp->used_pdpes);  allo
c_pt:   pt = alloc_one_pt(pd, pde);   set_bit(pde, pd->used_pdes);  }
```

Here is a picture which shows the bitmaps for the 2 allocation example above.



Bitmaps tracking page tables

The GPU mirroring interface

I really don't want to spend too much time here. In other words, no more pictures. As I've already mentioned, the interface was designed for a proof of concept which already had code using userptr. The shortest path was to simply reuse the interface.

In the patches I've submitted, 2 changes were made to the existing userptr interface (which wasn't then, but is now, merged upstream). I added a context ID, and the flag to specify you want mirroring.

```
 struct drm_i915_gem_userptr { __u64 user_ptr; __u64 user_size; __u32
ctx_id; __u32 flags; #define I915_USERPTR_READ_ONLY (1
```