The use of physical memory by the accelerated GLX module

John Carmack
id Software

johnc@idsoftware.com

Note: This document is mostly specific the the mga (matrox graphics) driver. The riva driver doesn't use dma and the ati rage driver isn't really functional at this point.

Definitions

Local memory is memory on the graphics card, usually 8/16/32 megs.

The primary consumer of local memory is the main graphics display. Unlike windows and macOS, when the resolution is changed lower during a session in X, the memory is not reclaimed. This means that if you are running an 8 mb G200 at 1600*1200*32 bit, you will never have any local memory left for 3D buffers, even if the application changes the resolution down to 640*480. (this is due to change in XF86 4.0)

You can still do accelerated 3D in that situation if you have system memory configured properly, but you are best off setting your full desktop resolution such that you can open a full screen 3D window without running out of local memory. That means no more than 1280*1024*16 bit or 1024*768*32 bit for an 8 meg card, and 1280*1024*32 bit or 1600*1200*16 bit for a 16 meg card. A 32 meg card has no practical restrictions.

Also note that high resolutions, color depths, and refresh rates can take a lot of bandwidth away from the 3D renderer, in some cases causing over a 2x performance difference on windowed apps like gears.

The X server also uses a small amount of local memory for pixmap caches and the mouse cursor image.

The rest of local memory is available for GLX to use for back buffers, depth buffers and textures.

System memory is the memory on the motherboard, which can be accessed from the card with either AGP or PCI transfers. You should have at least 64 megs of memory to be doing 3D work. An inefficiency of note with the current system is that mesa always has full 32 bit versions of all textures in virtual memory, even if they are also present (possibly in 16 bit form) in memory used by the card.

The matrox G200/G400 chips are very flexible in how they use memory. Almost all operations can be performed to local memory, to PCI addresses, or to AGP addresses. The only exception is, unfortunately, command buffers, which cannot reside in local memory. Attempting to put command buffers in card memory results in a hard system crash after a couple frames.

We can operate without command buffers in programmed IO (PIO) mode, but performance is drastically lower, so for high performance applications we must have some system memory.

The 3D card doesn't have a MMU like the CPU, so the command buffers need to be large and physically contiguous, which is difficult to achieve under linux (and most other OS's).

Configuring the GLX module's memory usage

The two options we use are the "LILO mem hack" and the AGP GART kernel module.

The LILO mem hack involves having LILO (the x86 Linux boot loader) tell the kernel that there is less memory in your machine than it actually has, and letting GLX use the memory that the kernel doesn't even know about. This gets you arbitrarily large, physically contiguous memory, which you can't get through any other service.

You need to add a line in `/etc/lilo.conf` that passes a kernel parameter with the amount of memory you want to use for normal system operations. If you have 64 megs and you want to leave 4 megs dedicated to graphics, you would have

```
image=/boot/bzImage             # or whatever your normal image is
append="mem=60m"                # leave some memory for glx
```

Then run `/sbin/lilo` to grab the new parameters, and reboot. Note that if you forget the "m" after the number, linux will crash on startup, because it tried to run in *60 bytes* of memory.

You will then need to tell glx what memory to use by adding or changing three lines in the `glx.conf` file (which is hopefully in `/etc/X11`, but may be elsewhere) to something like this:

```
mga_dma=3               # enable system memory access
mga_dmaadr=60           # the start of memory left for glx
mga_dmasize=4           # total system memory - mga_dmaadr
```

When you start X up, if everything goes properly, GLX will build command buffers in this memory space and the 3D chip will use PCI direct memory access to read from them. If you have "mga_boxes=1" in `glx.conf`, you should see a white box in the upper left corner of 3D windows. If you have a grey box, glx is still using PIO for command transfers, so check the logfile for errors.

The primary disadvantage to this method is that if you type a number wrong in `lilo.conf` or `glx.conf`, GLX or the 3D chip can write all over other people's memory, trashing everything in the system, including the kernel. This is Bad. Be careful.

The other disadvantage is that you can't add or remove memory from the system or change the amount dedicated to graphics without reconfiguring both files. Forgetting something can again cause Bad Things to happen.

Intel designed AGP to nicely solve the resizable physically contiguous memory problem, among other goals.

In an AGP system, the chipset acts like another memory manager for both the 3D card and the CPU, allowing scattered pages of memory to look physically contiguous. This allows an operating system to move, allocate, and de-allocate pages of memory and still have it (the AGP "aperature") look like a single large block of memory to an AGP card.

It also does this mapping for the CPU, which seems a little silly (the cpu could do it with its own memory page tables), except for a quirk of the pentium II architecture that requires physically contiguous memory with special alignment for the write combining range registers. Write combining is important for high performance graphics, so intel added this capability to the chipset, rather than wait for the next cpu. Later pentium II and III chips have the ability to set write combining on individual pages, but the aperature is still commonly used for the convenience of a single set of mappings.

AGP transfers have a few additional benefits beyond the configuration wins.

AGP 2x has twice the bandwidth of 66MHz PCI transfers from an AGP slot.

PCI reads cause snoop operations to the processor, which is always some degree of overhead, and on some AMD+VIA systems, may completely freeze the cpu while the transfer is underway.

AGP can pipeline addresses in the sideband, allowing the entire data bandwidth to (theoretically) be used, while PCI must spend data clocks to change addresses.

In practice, you want to use AGP unless:

You have a PCI G200 card. Some x86 G200 systems are PCI based, and all but the newest alpha systems are strictly PCI. We also may get accelerated support going on PPC systems at some point.

The GART kernel module doesn't work on your system, or you don't want to bother compiling/installing it.

There are two technical reasons why you theoretically might want to use PCI transfers instead of AGP, but it is doubtful they will ever come up:

If you need more texture memory than you can get with the AGP GART (theoretically up to 256 megs, but in practice usually 64 or 128 megs due to kernel allocation issues), you could use PCI accesses and have up to a couple gigs of texture space. Someone could port the SGI IR world texturing demos if they wanted to.

Some driver architectures might be able to take advantage of PCI cache snooping for vertex buffers, but I have heard (unsubstantiated) doubts from hardware people that all AGP chipsets actually implement that correctly.

The agpgart module

FIXME: insert good info on getting, building, installing and configuring gart – I don't feel knowledgeable enough about all the linux issues.

How much system memory should you allocate?

AGP should be able to grow and shrink as needed, but we have not implemented this yet. This is not difficult, and should be a fairly high priority. For now, you must explicitly size it like pci memory.

For optimal performance, you need large enough command buffers to hold two complete frames worth of commands: one for the card to be executing, and one for the driver to be building. For quake1/2/3 games, four megs (the default) is sufficient in almost all cases. Hardware utilization and overlap drops if the buffers are forced to overflow. If you are sending many more triangles, you may want to increase this size by adding a "mga_cmdsize = 8" to `glx.conf`. Asking for a larger command buffer than can be allocated will result in a fallback to slow programmed I/O.

We might want to look into automatically increasing the command buffer size if it overflows and memory is available, but distinguishing occasional texture uploads from steady performance might be tricky.

Textures can be stored in system memory. For a G200, performance is usually faster with textures in system memory, because the 64 bit local memory bus gets saturated handling the back and depth buffers. For a G400, texturing will be faster in local memory. If your textures don't all fit in local memory, you will be a lot better off using system memory textures instead of having the driver swap textures into local memory.

We might want to look into allocating textures first in local memory, then falling back to system memory when the allocation fails. There are tradeoffs with that, because it would be unfortunate to force a back or depth buffer into system memory due to texture allocation.

Back and depth buffers can also be allocated in system memory, but performance is significantly worse than local memory even for the G200. Buffers are only allocated there if the allocation in local memory fails. It's still a heck of a lot faster than the software rendering fallback, which is what you hit if you open up a half dozen large 3D windows (say, all the glx screensaver hacks) with only local memory.

We might want to look into buffer migration options. If you open one giant window, then open another that gets forced to system memory, then close the first one, the remaining window will be stuck in slow system memory. I’m not sure it is a common enough case to justify the added complexity.

So, the short answer for how much memory is either 4 megs for command buffers, or 16+ megs if you want it to also be used for textures and window buffer overflows. One of my systems is set up with 128 megs of system memory, but that is clearly overkill.

CPU memory performance.

In practice, the largest difference in performance of the different memory options is how fast the CPU can write to the buffers, not how fast the card can read from them.

Normal uncached memory can only be written to at about 30 MB/s on intel systems, and somewhat higher on amd systems.

Writeback cached memory usually sees about 100 MB/s on intel systems, and somewhat lower on amd systems. This would not be legal for AGP transfers, and we can’t usually set it for the LILO hack, so it is sort of irrelevant.

Write combined memory usually sees 300 MB/s to 400 MB/s on intel systems, and 100+ on amd K6 systems. K7 systems should see even higher numbers than intel, but I haven’t tested one yet.

Write combining is usually turned on with memory type range registers (MTRR). New systems can do it with page table bits, but there isn’t a linux interface for it. MTRR have picky alignment restrictions. The base must be a multiple of the size, which must be a power of two (I think amd systems are a bit more relaxed than this). MTRR can’t usually overlap. Intel systems must have all memory ranges covered by an MTRR or performance goes down to only a couple megs a second.

This means that you can’t put an MTRR on the top 4mb of a 64 meg system, because you can’t cover the other 60 megs with a single MTRR.

I had a system in place at one point that covered the remainder with as many MTRR as needed, but when the X server exited, all but the first got deleted, putting the system into the basically unusable few-meg-a-second memory mode.

Currently, you only get write combining if you use either AGP (NOTE: are we going to move this from the gart module to glx?) or you have mga_dmaadr set to a power of two. This restriction usually means that if you aren’t using AGP and want write combining, you need to configure your system with half the memory dedicated to graphics. That is not a viable option for most people. (NOTE: we could use a different check on amd systems)

Running without write combining is still much faster than PIO, so don’t worry too much about it.

Document history

Originally written 1999 Nov 16 – John Carmack <`johnc@idsoftware.com`>

Converted to DocBook xml with minor editing 1999 Nov 16 – Ralph Giles <`giles@ashlu.bc.ca`>

Last modified: $Id: memory-usage.sgml,v 1.2 1999/11/19 03:47:19 stephenc Exp $