Attention:

Chris Hecker
definition six, incorporated
checker@d6.com
http://www.d6.com/users/checker

PS. The email address at the end of the article is incorrect. Please use checker@d6.com for any correspondence.

Memory Miscellanea

Chris Hecker

Don't worry, perspective texture mapping is alive and well. But this month, Chris Hecker takes a break from his series on texture mapping to explore the nuances of memory bandwidth in game programming.

As you can see from the title, this article is not "Perspective Texture Mapping, Part IV," the continuation of our epic perspective texture mapping series. Don't panic, I'm not breaking my promise to deliver a wicked fast perspective texture mapper, but to break up the series a bit I thought I'd insert a non-texture mapping article here in the middle (although the topic is definitely applicable to texture mapping, as you'll see). We'll resume with Part IV next issue.

This time through, we're going to discuss memory bandwidth. Plainly stated, memory bandwidth is a measure of how much memory you can read and or write in a given amount of time.

From that description, it should be clear that memory bandwidth affects every kind of game on every platform, from scrolling platform games on an 8-bit Nintendo or Atari 2600 system to high-end military simulators that cost millions of dollars. Memory bandwidth governs how many sprites the hardware in the older consoles can move around, and how many polygons can be textured per second in hardware on the newest machines or in software on the PC.

In fact, an oft-cited goal of PC graphics programmers is to "get your texture mapper running at memory bandwidth," because there's not much more you can do to increase its speed after that. To get a tad flowery, memory bandwidth can be an open door or a brick wall. It all depends on how much of it you've got.

Lies, Damn Lies, and Bandwidth Numbers

On today's machines, we usually measure memory bandwidth in megabytes per second (MB/s), but you'll sometimes see bytes per second, dwords (a dword is four bytes in this article) per second, and so on. If you're looking at a bunch of memory bandwidth numbers, it's obviously important to know which measurement units they're in.

Like most statistics, the bandwidth numbers themselves aren't enough to tell the whole story, and you need to know exactly how the numbers were generated for a specific machine to give them meaning. For example, I could tell you the laptop on which I'm typing right now gets 42 MB/s, but you really aren't any more knowledgeable than before because you don't know if I mean read bandwidth, write bandwidth, copy bandwidth, sequential or random reads or writes, or any combination thereof. All these parameters can make a big difference inwhat a bandwidth number really means.

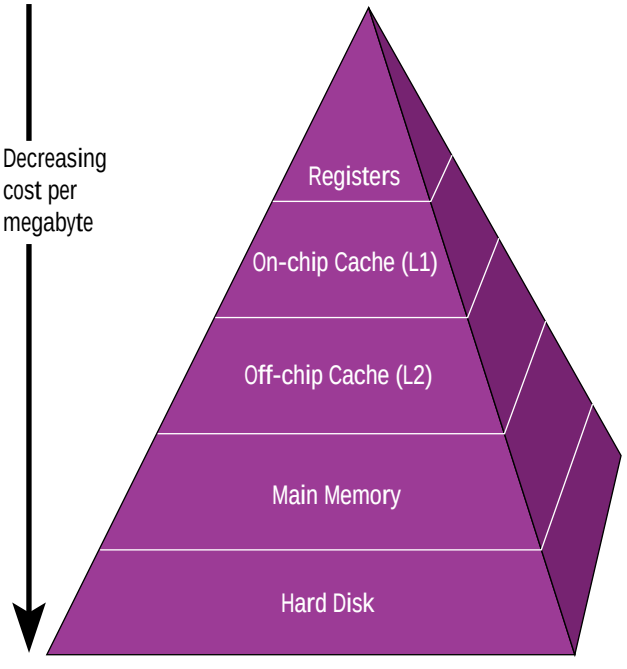
In fact, it's rare that any general bandwidth number will mean anything in the context of your specific game. I'm going to talk about various things that can affect your game's memory bandwidth, techniques for measuring that bandwidth, and pitfalls you'll encounter along the way.

Pyramid Power

First, we need a one-minute refresher on how modern CPUs and motherboards work. I'm certainly no hardware engineer, so we'll limit our discussion to how the software sees the hardware.

Throughout computer history, there's always been a pyramid diagram

Figure 1. Component Cost vs. Amount



that describes the speed of the component vs. how much of that component you're likely to have in your system (usually because faster components are more expensive). Figure 1 shows this diagram for memory components.

At the top of Figure 1 we have the CPU registers, which usually take a single cycle to access and are the most flexible of all types of memory, but are pretty scarce (Intel x86 processors have only eight general purpose registers, each of which holds four bytes, while most new processors like the PowerPC have around 32—all the parenthetical numbers in this paragraph are estimates and will vary in practice). Below the registers, we have the on-chip cache memory, sometimes called the level 1 cache. This is usually a small amount (8 to 32KB or so) of fast memory with access times slightly slower than registers. Cache memory is a little less flexible than registers, as well. Most CPU architectures don't let you add a number in the cache directly to another number in the cache without using the registers for temporary storage.

On the next rung down, we have the off-chip level 2 cache, which usually has more storage space (256KB to 1MB), but is much slower than the on-chip cache, generally on the order of five times slower or more.

Second to last in our diagram, we have main memory—of which there's usually a relatively large amount (4 to 32MB). As you'd expect, it's even slower than any of the memories above it. Finally, we end up with the hard disk, which has oodles of storage (well, okay, my hard disk sometimes has less space free than I have main memory, but it still has more raw storage!) if you're willing to pay for the access time and transfer rates. CD-ROMs and tape drives would be below hard disks if we put them in the diagrams, because they're cheaper (and slower) per megabyte.

The most interesting thing about Figure 1 is that it holds for almost every machine architecture, from Commodore 64s to Crays. On the lowest end, you might not have caches, and at the higher end you might have more layers, but the speed vs. cost ratios still stand.

With that refresher, let's get to the hints, tips, and techniques for determining the memory bandwidth for your game, so you can strive to achieve it.

What's Your Access Pattern?

As I mentioned, a single number doesn't tell the whole story about memory bandwidth. In fact, there are zillions of different kinds of memory bandwidth, each different because the access pattern used to generate the numbers is different. The access pattern is the way the application moves the memory around, and there are as many different types as there are programs. Three general categories that are important, but by no means form a complete list, are sequential copy bandwidth, sequential write bandwidth, and random read-sequential write bandwidth.

Sequential copy bandwidth is the number that applies when you're copying a block of memory from one place to another—to copy a new piece of digital audio into the play buffer, for example. Sequential write bandwidth is what you see when filling a rectangle or polygon, zeroing an array, or anything else where you're writing a single value or a value that's generated using instructions (as opposed to read from a source) to a destination. Finally, random read-sequential write bandwidth is what you see when texture mapping, where your source locations are fairly randomly distributed, but you're usually writing a scanline at a time to the destination.

From these descriptions, you can easily come up with other kinds of bandwidths and situations in which they'd arise. You might find multiple reads and a single write interesting if you're mixing digital audio or alpha blending sprites. Likewise, a single read and multiple writes might be your thing if you're stretching an image. The key is to figure out which type of bandwidth is most appropriate to your application and measure it.

It's clear that any useful and interesting application is going to do more than just copy bits all day, but memory bandwidth gives a good upper bound on your performance. In other words, even if you were the best optimizer on earth, you still wouldn't be able to get your code faster than memory bandwidth if you need to move those bits around. This may seem like a limitation, and it is, but you can also look at it as an opportunity. If you can figure out a way to reduce your memory bandwidth requirements by redesigning your algorithm or possibly by changing your

access pattern you can open up new possibilities for optimization.

An Accessible Example

For example, if you were writing a solid polygon rasterizer, you could measure sequential write bandwidth and compare it to the fill rate you get through your rasterizer. The difference is your overhead above memory bandwidth. Your goal is to minimize this overhead or, if possible, cheat somehow so you get the same effect on the screen but aren't bound by the same memory bandwidth limitations.

Let's say you have the world's fastest 90-degree bitmap rotator; you can take a bitmap and rotate it 90 degrees at memory bandwidth on your machine—you're very proud of this code. You know it works at memory bandwidth because you measured it without any instructions except the copies in the inner loop and got the same bandwidth number when you added your rotator code. Let's also say your code is "destination-centric," that is, it scans horizontally in the destination and therefore it scans vertically in the source to accomplish the rotation. Of course, you measured memory bandwidth doing the same thing, so let's call this access pattern vertical read-sequential write. Since we're running up against this memory bandwidth limitation, how can we restructure the algorithm to have a different limitation?

It's immediately apparent that you should measure sequential read-vertical write, which will accomplish the same rotation, but might be a different speed. Also, another possibility is to spend a little memory and pre-rotate your bitmaps, so your access pattern is sequential copy. Will either of these be faster? I don't know, and we can't say with certainty until we've timed it. My hunch is that sequential copy will be the fastest in terms of pure bandwidth because it's probably the access pattern for which the memory subsystem was optimized, but it's just that, a hunch. It's entirely possible the extra memory overhead from pre-rotated bitmaps would make the overall code slower because of paging.

The real solution, if this is a bottleneck in your game's run-time speed (and you shouldn't even be bothering to measure this stuff if it isn't a bottleneck), is to profile the various techniques at startup and have your game self-configure to use the fastest possible pattern for the given machine.

What's fastest on your machine might not be fastest on mine. The best we can do is have a list of things to look for when we're measuring bandwidth.

It may seem like I'm being wishy-washy by not just declaring a single access pattern the fastest, but we've got far too many variables to do so. The problem is compounded by the number of different hardware architectures out there, so what's fastest on your machine might not be fastest on mine and vice versa. The best we can do is have a list of things to look out for when we're measuring bandwidth. Cache effects would definitely be at the top of this list.

Understand the Cache

The processor cache is usually an object of great fear, wonder, and misunderstanding. A friend of mine named Terje Mathisen says, "All programming can be thought of as an exercise in caching." Although Terje isn't talking specifically about the processor cache, this is a rule to live by when you're trying to optimize on modern processors. If we apply this idea to the processor cache and memory bandwidth, it means, "Figure out how to put your important data in the cache and keep it there." This may seem obvious, but keeping your data in the cache is more difficult than you might think.

Before we bother getting into this, what difference does it make? Well, on my laptop, the speed difference between reading from the on-chip cache and reading from sequential uncached memory is tenfold—and this isn't even the whole story. I'm reading sequentially in this example, so at least some of the reads are cached for reasons I'll explain shortly. If I ensure that all the reads are uncached by reading pseudo-randomly, the program reads from the cache about 30 times faster than from main memory. You can do a lot in 30 cycles on a modern processor, so I'd rather not spend them waiting on memory.

I'm going to assume you know generally what a cache is and how it works, so the only high-level description I'll give is this: the cache stores frequently accessed data in fast on-chip memory, so when you reference it the chip doesn't have to go out to the memory bus to fetch your request.

Caches are broken up into cache lines, which are usually 16 or 32 bytes long, and the processor reads in an entire cache line from memory when a cache miss occurs. These cache lines are aligned on address boundaries that correspond to their length, so 16-byte cache lines are aligned on 16-byte boundaries, for example (addresses ending in 0 hexadecimal). This is why my previous sequential reads were partially cached. Assuming a 16-byte cache line, every fourth dword I read in my test brought in another cache line,

and the next three dwords were read from the cache's fast memory. The use of cache lines also means that if you're referencing two bytes at addresses that differ by more than a cache line (or if there's a cache line boundary between them) you'll be using two lines, even if you're only accessing those two bytes.

Life in the cache gets even worse when we delve deeper into its behavior. Most modern caches are N-way set associative for some small integer N, usually 2 or 4. A cache set is a group of N cache lines, so a two-way set associative cache has a bunch of sets, each containing two cache lines. You can tell how many sets there are by taking the size of the cache in bytes, dividing by the number of bytes per line to get the total number of lines, and then dividing by N for your cache to get the total number of sets. For example, the Pentium has 8KB of two-way set associative data cache with 32-byte cache lines, so 8KB / 32 bytes / 2 lines per set = 128 sets.

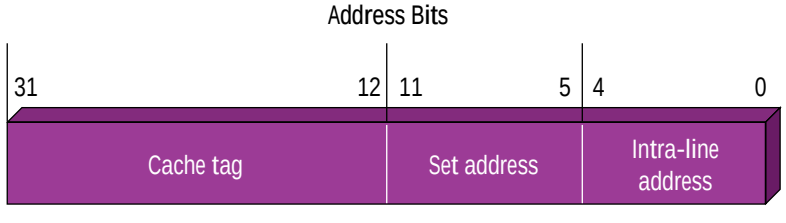
The cache translates a memory address into a cache line address by using the lowest bits for the intra-line address, the next few bits for the set address, and the remaining high bits for the cache tag. Figure 2 shows the breakdown for the Pentium. Because there are 32 bytes in a cache line, the lower five bits are used for the intra-cache line address, and the next seven bits give us the 128 sets we calculated previously. Which line a given address uses in its set is up to a replacement algorithm (usually least recently used or an algorithm close to it) based on the cache tag bits. In other words, every address that contains the same set address bits will map to the same set, and all those addresses must share the lines in that set.

This is where the replacement algorithm comes in. If all the lines in the set are currently full and none of the tags matches the requested address, then one of the currently cached lines needs to be replaced by the current requested line. For example, on the Pentium only two of the many possible cache lines (20 bits worth of lines because bits 12-31 make up the tag—that's 1,048,576 possible lines!) for a given set can be in the cache at the same time.

You can see why this works well in the general case because referenced addresses are likely to be near each other—a phenomenon called locality of reference—so they'll have different set addresses. The set architecture allows for some addresses to be not-so-near each other because it lets very different addresses with the same low bits map to N different cache lines (in contrast, a cache architecture called direct mapped has no sets, so each address with the same low bits shares a single cache line).

However, in certain cases this kind of cache architecture can really screw you up. For example, let's say you're reading vertical strips from a bitmap like the bitmap rotator we discussed previously—down one vertical scanline, then down the next, and so on. The width of your bitmap will dictate how much of the cache you end up using. If your bitmap is 256 bytes wide, a single increment vertically will step bit 8—and never any bits below bit 8—in your address. If you compare that with the Pentium's cache address layout in Figure 2, you'll notice that you're only using four bits of your possible seven bits of set address. This means that instead of using all 128 sets, you're only using 16 of them or only 1/8 of your total cache! The startling implication of this is the next horizontal byte from the first scanline will not be in the cache when you get back up there if you've gone farther than 32 scan lines (16 sets x 2 lines per set) because it's been pushed out of its set by another line.

Figure 2. Pentium Cache Addressing



Assume Nothing

What can you do about this sort of thing? Well, first you need to realize when it is happening in your code. To do this, you need to get good at profiling. Profiling at this level doesn't mean just running the code profiler that comes with your compiler and examining the results, it means figuring out exactly where your code is spending its time in the inner loop. You can definitely use the high level profiler to find the inner loop in the first place, but once you've found it, if you want to max it out and really pin down why it's taking the time it is, you're going to need to get down and dirty with a very accurate timer (personally, I use `timeGetTime` or `QueryPerformanceCounter` on Windows, but any accurate timer will work) and a knowledge of assembly language.

Before continuing I should stress that this kind of profiling and optimization takes a very long time, so you should make absolutely sure you're applying that time to the right part of your code. In a game, there are probably 10 lines of code in the entire project that might need this sort of attention, and if you're going to spend a week looking at them you had better make sure they're the right 10 lines.

Michael Abrash's phrase, "Assume nothing!" and its corollary, "Time everything," are words to live by in this neck of the woods. Abrash is the master of this sort of optimizing, so you should definitley read his book *Zen of Code Optimization* (Coriolis, 1994). As a bonus, the disk that accopanies the book comes with a very accurate timer designed specifically for this in-depth profiling.

While I was writing this column, the need for this very kind of profiling came up. I was gathering bandwidth

statistics for various access patterns on my 486 laptop and I was trying to time cached reads. I know from both experience and the 486 manual that a read from the cache is a single cycle, but I couldn't seem to convince my timing program of this fact. It kept returning around 1.5 cycles per read, and when you're timing at this level that's 50% off. My test program had an unrolled loop of a couple of hundred reads, and then I looped back to the top a bunch of times. I was very careful not to unroll my loop so much that it blew out the code cache, so I simply couldn't figure out what was going on. I timed other single cycle instructions with the same timing harness (using a millisecond timer and looping a lot), and they returned reasonable times, like 1.02 cycles, but my reads kept returning 1.5. If I stuck a nop in between the unrolled reads I got the expected two cycles, one for the read and one for the nop. I stared at the code, trying to find an address generation interlock, (AGI—Intel-speak for a type of pipeline stall), but there weren't any.

Finally it hit me. I remembered that if you're continuously reading from cached memory without allowing even a single free memory cycle for prefetching instructions, the 486 will stall your code to fill the prefetch queue. Eureka! I verified this was the culprit by changing the number of consecutive reads and got the expected one cycle per read. I also looked it up in the 486 Programmer's Reference Manual from Intel, and the stall was listed there among the others.

Time to Cache Out

As you can see, figuring out where every cycle is going in your inner loop, especially when there are strange effects brought on by your memory access pattern, is very difficult and time consuming. I highly recommend reading and rereading the manual for your processor before you try to do this. Also, always test your timing program with known inputs so you can verify that it works; Heisenberg is alive and well at this level.

I haven't covered video memory and its associated bandwidth weirdnesses at all. Nor have I discussed processor write buffers, write-back versus write-through caches, processors that don't write allocate cache lines (like the Pentium), new trends in memory that affect the bandwidth numbers (like EDO memory, RAMBUS, and SDRAM), groovy new cache/access pattern debugging instructions (like RDMSR on the Pentium), and much more. Hopefully this article gives you enough background and forewarning about the strangeness you'll encounter that when it's time to max out your inner loop, memory bandwidth won't be the mystery it can be for the unprepared. ■

Chris Hecker wonders why five-year-old workstations with incredibly slow CPUS still have better memory bandwidth than today's top-of-the-line PCs. You can commiserate with him at checker@bix.com.