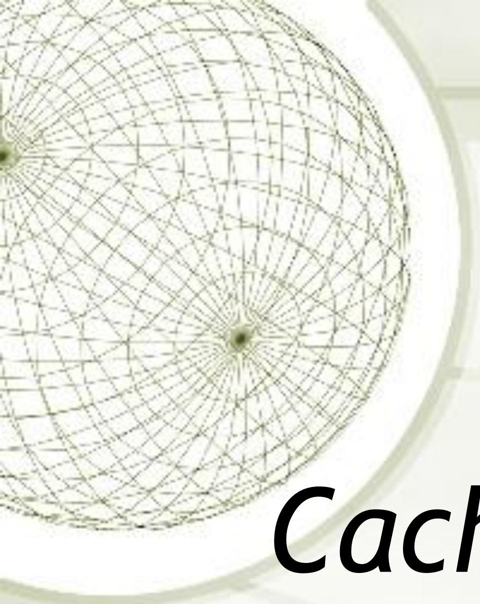


Caching Architectures and Graphics Processing

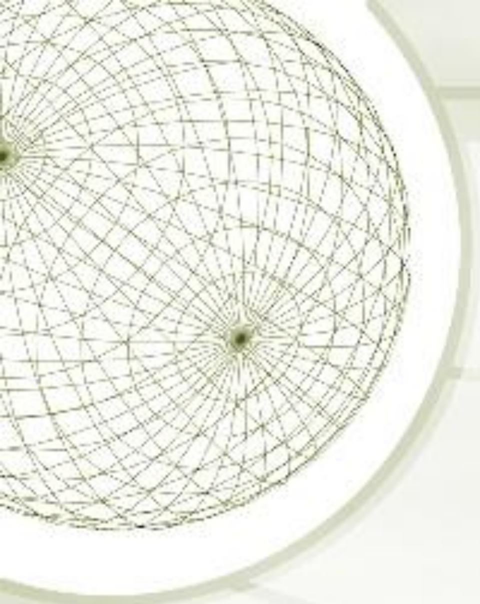
Todd Gamblin

1. Cache Crash Course
   - Quick review of the basics
2. Some traditional profile-based optimizations
   - Static: compile-time
   - Dynamic: runtime
3. How does this apply to the GPU?
   - Maybe it doesn’t: Matrix-matrix multiplication
   - GPU architectural assumptions
   - Optimizing the architecture for texture mapping

Why Cache?

- CPU/GPU Speed increasing at a much higher rate than memory (DRAM) speed
- DRAM is made of capacitors, requires electric refresh, which is slow
- Speed improves at a rate of 7% per year
- CPU speed doubles every 18 months
- GPU speed doubles every 6 months (Moore^3)

- Bottom Line: Memory is slow.

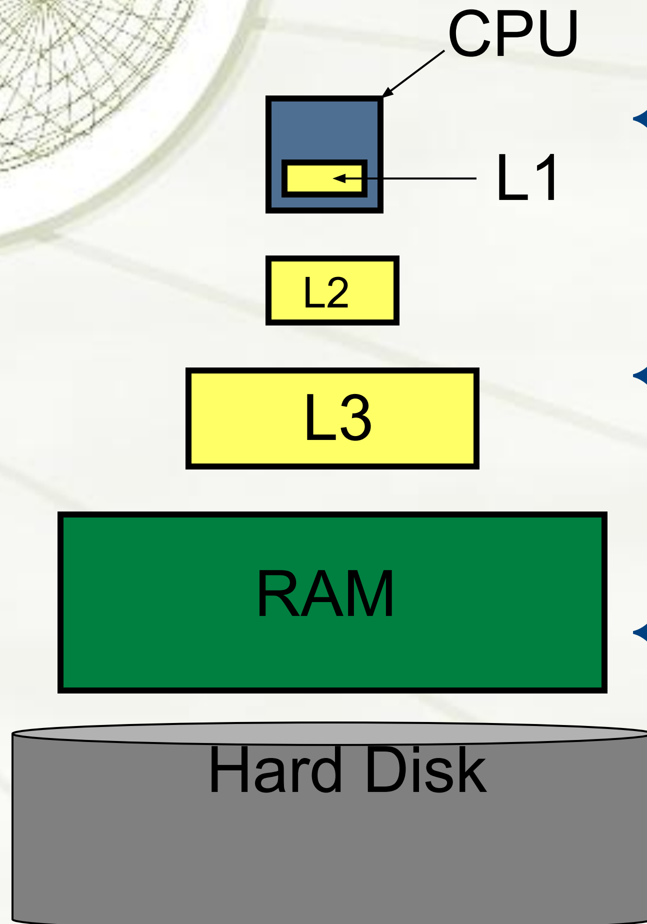


So what to do?

- DRAM not the only option
- Can use SRAM, which uses flip-flops for storage
- Takes 2 transistors for a flip-flop
- Fast, but expensive
- Can't afford SRAMs even close to the size of main memory

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

Use memory hierarchy



- Small, fast memory close to CPU (even on-die)
- Progressively slower, larger memories further away
- Disk can also be seen as a level of this (with VM system as the caching mechanism in RAM)

- How does this speed things up?
- Key observation: Most programs do not access all code or data uniformly
- Locality
  - **Temporal:**Programs tend to access data that has been accessed recently (e.g. instructions in a loop)
  - **Spatial**: Programs tend to access data with addresses similar to recently referenced data (e.g. a contiguously stored matrix)
- Point is that we don’t need all of memory close by all the time, only what we’re referencing right now.

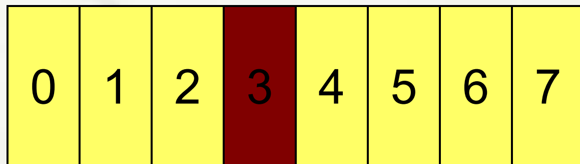
Working Set

- Set of data a program needs during a certain time to complete a certain task is called its *working set*
- If we can fit this in cache, we don't need to go to a lower level (which costs time)

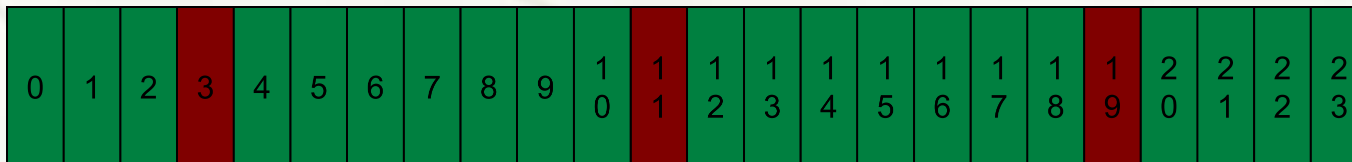
- Cache is transparent
- CPU still fetches with same addresses, can be completely unaware of cache and still operate correctly
- Need a function to map memory addresses to cache slots
- Data in cache is stored in *blocks* (also called lines)
  - This is the unit of replacement -- If a new block comes into the cache, we may need to evict an old one
  - Must decide on eviction policy
    - LRU tries to take advantage of temporal locality
- Along with data we store a *tag*
  - Tag is the part of the address needed for all blocks to be unique in cache
  - Typically the high lg(Mem size/cache size) bits of the address

Direct mapped cache

- Blocks of memory map to their address modulo cache size
- Evict on conflict
- Pros
  - simple to implement: just shift bits
  - fast access time
- Cons
  - Simple hash function => can get many conflicts



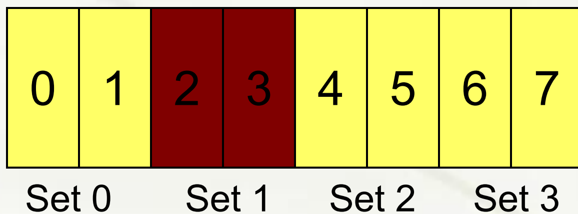
Direct Mapped Cache



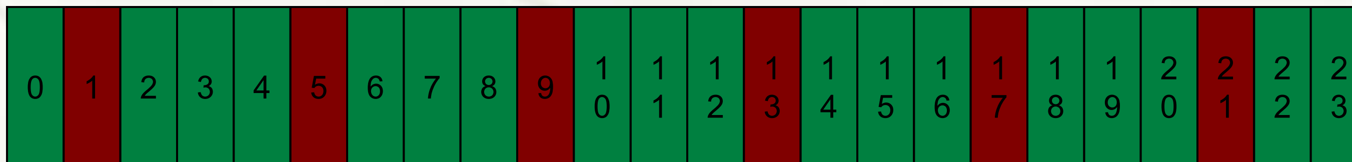
RAM

Associative Cache

- Now have sets of “associated” blocks in cache
- Blocks from memory can map to any block in a particular set
- Typically have 2-way, 4-way, 8-way, and fully associative caches
- Pros
  - A k-way cache can eliminate conflicts if no more than k blocks of memory map to the same block in cache concurrently (I.e. k blocks in the same working set)
- Cons
  - harder to implement, need a parallel comparison of tags at each block in cache
  - Results in slower access times, more expensive hardware



2-way associative Cache



RAM

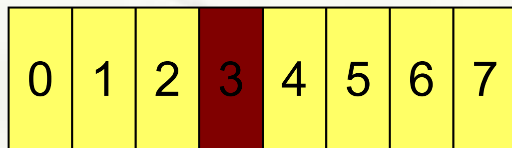
- Any block in memory can map to any block in cache.
- Most expensive to implement, requires the most hardware
- Completely eliminates conflicts

- Need some way to itemize why cache misses occur
- "Three C's" of cache misses:
  - Compulsory (or Cold)
  - Conflict
  - Capacity
- Sometimes coherence is listed as a fourth, but this is for distributed caches. We won't cover it.

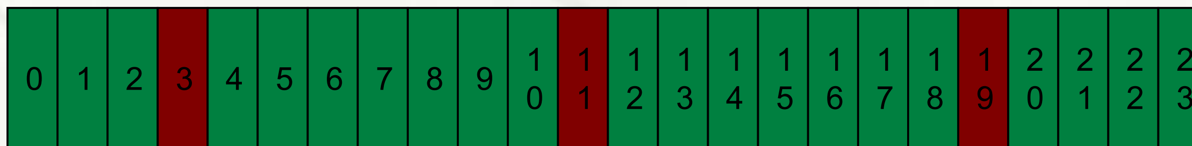
- Caused when data first comes into the cache
  - Can think of these as misses that occur in an infinite cache
- Not much you can do about these
- Can slightly alleviate by prefetching
  - Make sure the thing you need next is in the same block as what you're fetching now
  - Essentially this is the same thing as saying to avoid cache pollution
    - Make sure you're not fetching things you don't need

Conflict Misses

- Caused when data needs to be fetched again because it was evicted when another block mapped to the same cache line.
- Fully associative caches have no conflict misses
- Typically the biggest obstacle to reuse of data
  - Ideally blocks in the same working set will not conflict with each other
  - May need to move things around in memory in order to optimize for this
  - Can also add associativity
- Recall direct mapped cache:
  - If 11 and 19 are fetched in strict alternation, we can get worst case access time
  - Have to go to memory every time



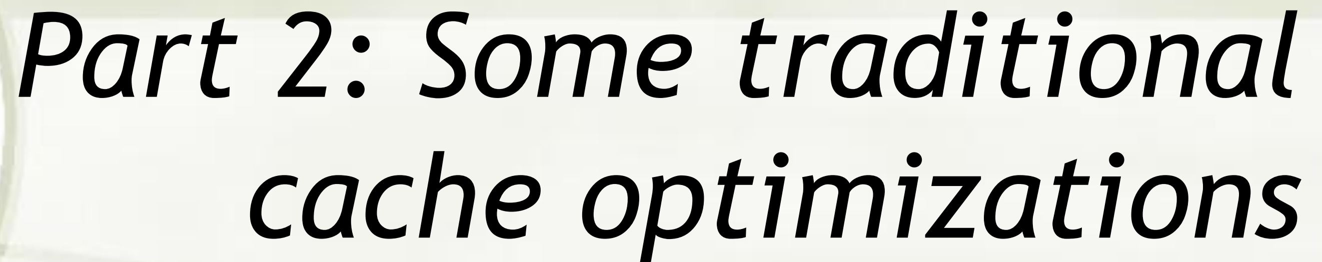
Direct Mapped Cache



RAM

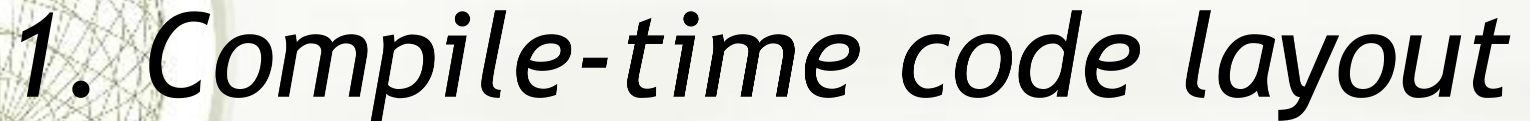
Capacity Misses

- If the cache cannot contain the whole working set, then *capacity misses* will occur when blocks are discarded for lack of space and fetched again later
  - Think of these as misses that would occur in a fully associative cache, discounting compulsory misses
- Can alleviate by making working set smaller
  - Smaller working set => everything fits into cache



Part 2: Some traditional cache optimizations

- Not graphics hardware related, but maybe these can give us some insight
- All of these are profile-based
  - Take memory traces and find out what the program’s reference patterns are
  - Find “Hot spots”: Frequently executed code or frequently accessed data
    - Reorganize code at compile time to reduce conflict misses in hot spots
    - Reduce working set size
  - Can do this at runtime, as well
    - Java profiles code as it runs: HotSpot JIT compiler
    - Garbage collector, VM system both move memory around
    - Can get some improvement by putting things in the right place



1. Compile-time code layout

- Want to optimize instruction cache performance
- In code with branches and loops, fetching is not done in strict sequential order
- Can get cache conflicts in the instruction cache if two procedures map to the same place
  - Particularly noticeable in a direct-mapped cache
- Pathological case: might have two procedures that alternate repeatedly, just as cache lines did in the earlier conflict miss example
- Working set is actually small, but you can't fit it in cache because each half of code evicts the other from cache

Map profile data to the code

- Pettis & Hansen investigated code layout based on profile info
- Profile naively compiled code, and annotate the call graph with frequency of calls
- Try to find most frequently executed call sequences and build up chains of these procedures
- Observe that a procedure may be called from many places, so it’s not entirely obvious which chain it should be in

QuickTime™ an
TIFF (LZW) deco
are needed to see t

QuickTime™ and
TIFF (LZW) decomp
are needed to see this

Lay out code based on chains

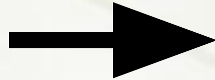
- Try to lay out chains contiguously, so they will not conflict in cache
  - Increases spatial locality of code that has obvious temporal locality
- Can go further and split entire procedures, to put unused code aside
  - keep unused error code out of critical path
  - Allows more useful code in working set
- Speedups from 2 to 10%, depending on cache size
- Interesting detail:
  - MS insiders claim this was key for codes like Office in the early 90’s

2. Smaller scale: Struct layout

- We saw instructions, now what about data?
- Most languages today use something like a struct (records, objects, etc.)
  - Fields within a struct may have different reference frequency
  - Directly related to likelihood of their being used
- In C, at least, structs are allocated contiguously
- But, unit of replacement in cache is a block
  - when we fetch a field we might get a lot of useless data along with the data we want.
  - Ideally the data we fetch would come with the data we want to fetch next

Split structs for better prefetching

- Chilimbi suggests breaking structs into pieces based on profile data:
- Profile code
- Find “hot” fields, and reorder them to be first
- Split struct into hot and cold sections
  - Trade off speed hit of indirection on infrequently referenced cold fields for benefit of less cache pollution on hot ones
- Reduced miss rates by 10-27%, got speedup of 6-18% for Java programs.



QuickTime™ and a
TIFF (LZW) decompressor
are needed to see this picture.

- Chilimbi suggests using runtime profiling to make garbage collectors smarter
  - Need a low-overhead profiling mechanism, with reasonable accuracy, for this to work
- Similar to code layout
  - Tries to reduce conflict misses
  - Deduce *affinity* between objects from profile data
    - Data equivalent of call graph parent-child relation
    - Indicates temporal locality

More garbage

- Garbage collector copies data when it runs:
  - Determines which objects are alive, which are dead
  - Copies live objects to new memory space
- Can use gathered information to co-locate objects with affinity when we copy
- Once again, temporal locality info used to construct spatial locality
- Chilimbi, et. al. claim reductions in execution time of 14-37%

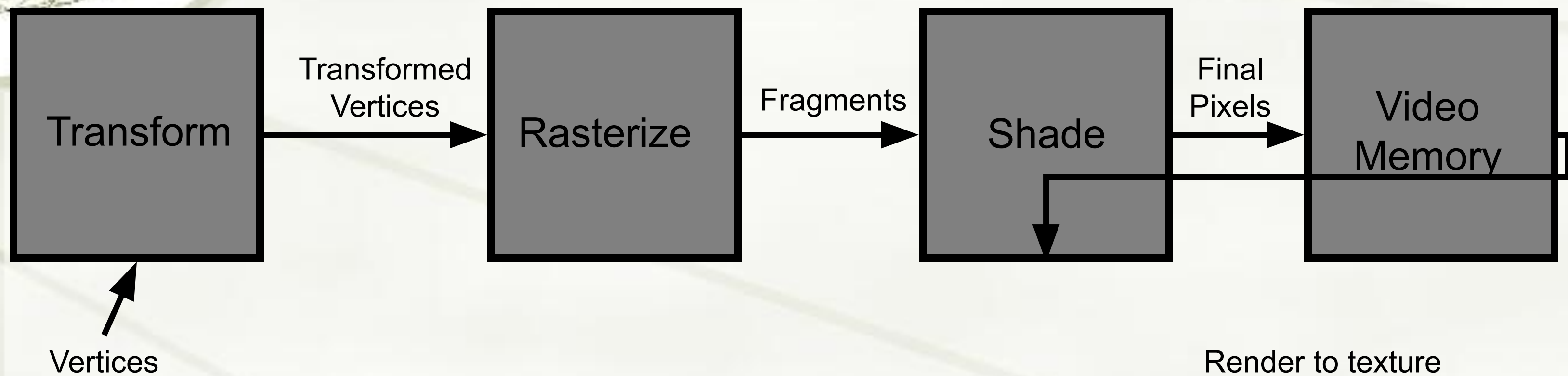
Other dynamic approaches

- Similar techniques suggested for VM system by Bershad, et. al.
  - Involves a table alongside the TLB, along with special software
  - Monitors hot pages, looks for opportunities to reallocate them cache-consciously
- Adaptive techniques not confined to systems domain
  - I could see this kind of technique being used in walkthrough
  - Dynamically restructure something like Sung-Eui's CHPM, based on profile information

- Things to think about when optimizing for cache:
  - How much data do I need (working set)
  - How much am I fetching, in total? (bandwidth)
  - How much of that is the same data? (conflict, capacity misses)
- Solution is almost always to move things around

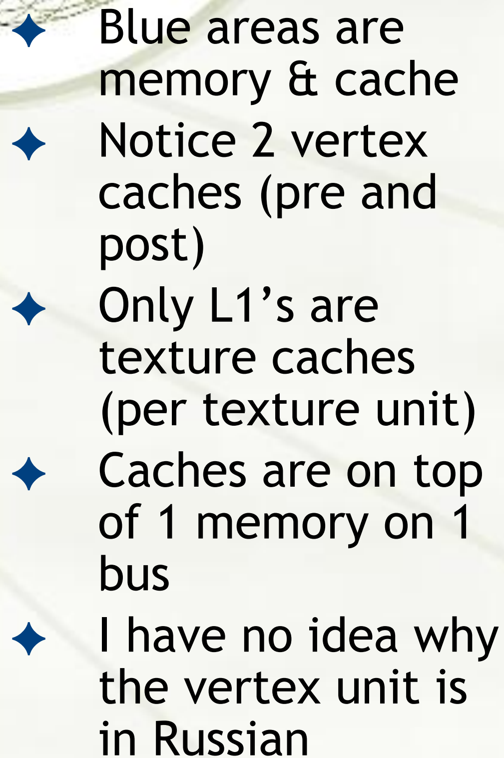
- Architectural Overview
- Optimization Example:
  - Texture cache architecture
- Matrix-matrix Multiplication
  - Why it’s so horrible
- Remedying the situation
  - What can be improved?

GPU Pipeline



- Recall GPU pipeline at high level (from Cg manual)
- Naga talked about vertex cache, texture cache
  - Sung-Eui is optimizing large model representations for vertex caches, trying to get more bandwidth
- Can easily imagine caches alongside these units, but let's look at this in-depth

NV40 architecture



- Blue areas are memory & cache
- Notice 2 vertex caches (pre and post)
- Only L1's are texture caches (per texture unit)
- Caches are on top of 1 memory on 1 bus
- I have no idea why the vertex unit is in Russian

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

Some points about the architecture

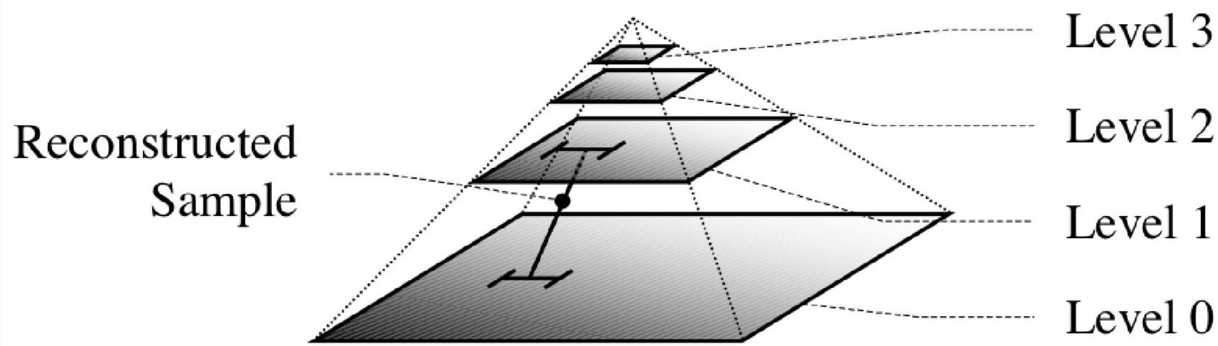
- Seems pretty ad-hoc
  - I feel like this will gradually merge together as programmability features increase
  - e.g.: Vertex shaders can reference fragments in texture cache, so these are slated to move together (per Mike Henson's info)
- Can tell optimizations are very specifically targeted
  - Lots of specialized caches
  - Only 2-level cache system is for textures
- Recent example of such an optimization
  - ATI 9800 Pro's Z-buffer touted to be optimized specifically to work better with stencil bufffer data
  - No specifics, but if architecture looks anything like this could make a guess as to why
    - Shared address space -> conflicts bt/w stencil and Z-buffer in cache
    - Esp. since you typically draw similar shapes in similar positions

GPU Optimmization example: Texture cache on the GPU

- We do not know exact specs for texture caches today, as they are not released.
- But, can guess based on papers on the subject.
- Igehy, et. al. present a texture cache architecture for mip-mapping and rasterizing.
- This texture cache is optimized heavily for one task: rendering
- Storage of textures on card could contribute to the lack of cache performance for GPGP applications
  - GPGP reference patterns different from those for rendering

MIP Mapping

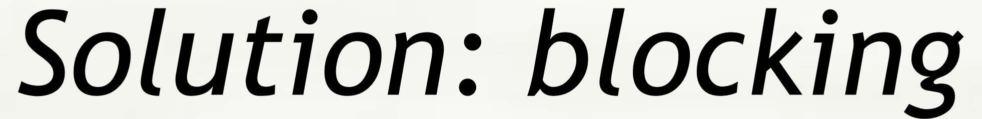
- Textures on card are stored in multiple levels of hierarchy
- Precompute small versions of texture, so that when it is rendered far away, we can save computation with no visual loss
- Compute MIP map level and interpolate between nearest maps
- MIP Maps have spatial locality built-in
  - Approximate 1-1 correspondence between MIP mapped pixels and screen pixels, which follows from the way they are used.



- Trilinear filtering used to interpolate pixels from MIP maps during rasterization references pixels in maps above and below the MIP level
- Difficult to avoid conflict misses between neighboring maps, because MIP maps are powers of 2 in size, just like caches.
- Texture data organization is key to avoiding these misses

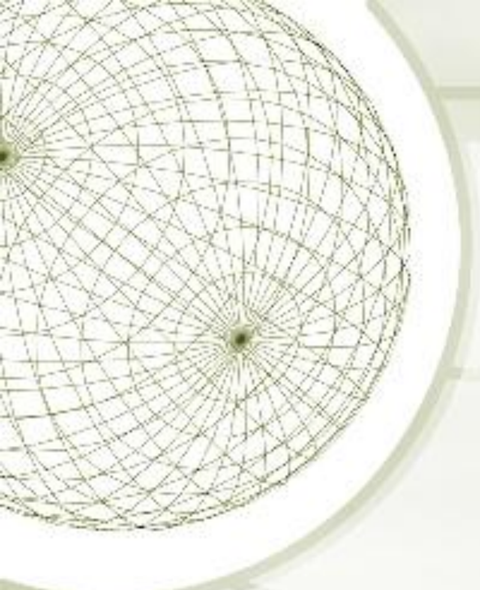
Rasterization

- Another pitfall for texture caches
- We saw in matrix multiplication how column-major memory accesses can be detrimental to a cache
- Same holds for textures, only we cannot be sure what their orientation is.
  - Depends on how they are oriented relative to the viewer at rendering time
- Rasterization typically moves left to right across screen pixels (with some tiling), regardless of the textures
  - Can be a disaster for cache if this direction ends up being orthogonal to the texture



Solution: blocking

- Igehy, et. al. use a blocked texture representation with special addressing to avoid these problems
  - Call it “6D blocking”
- Change order of texture pixels so that geometrically local pixels are also physically local in memory



Cache-Sized Superblock

Texture

Block

t_{super} t_{block} t_{offset}

s_{super} s_{block} s_{offset}

0	1	0	1
2	3	2	3
0	1	0	1
2	3	2	3

Bank Interleaving of Tags to Blocks in Superblock

0	1	0	1
2	3	2	3
0	1	0	1
2	3	2	3

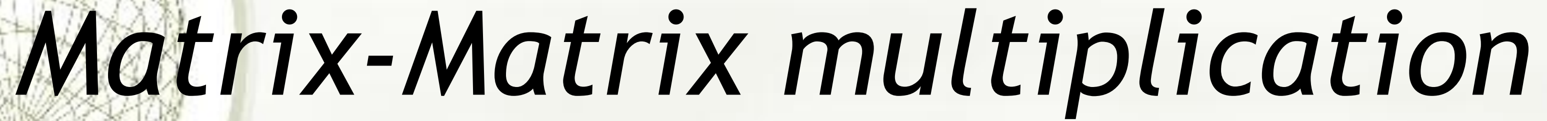
Bank Interleaving of Texels in Block

Locality in the texture representation

- First level of blocking keeps working set in cache.
  - Blocks are size of whole cache
- Second level of blocking makes sure nearby texels are prefetched
  - Sub-blocks are the size of cache blocks
  - Good for trilinear filter, as there’s a much higher likelihood that the needed pixels will be fetched.
- Texture accesses no longer depend on direction of rasterization for efficiency

Rasterization direction

- Igehy architecture uses 2 banks of memory, for alternating level MIP maps
- This avoids conflict misses from MIP mapping altogether
  - conflict misses occurred between levels during filtering
  - No adjacent levels can conflict

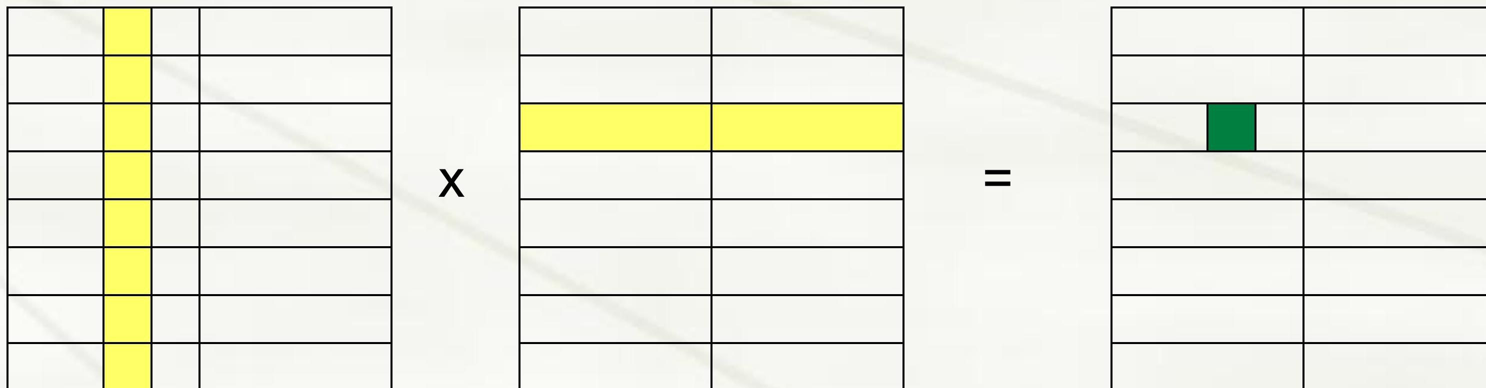


Matrix-Matrix multiplication

- GPU implementations so far:
  1. Larsen, et. al. - heard about this the other day
     - Performance equal to CPU’s, but on 8-bit data
  2. Hall, et. al.; Moravanszky
     - Both have improved algorithms
     - Moravanszky reports his is still beaten by optimized CPU code
- Not much on this, as results are dismal, as we’ll see
- First, let’s look at the typical approach to this problem

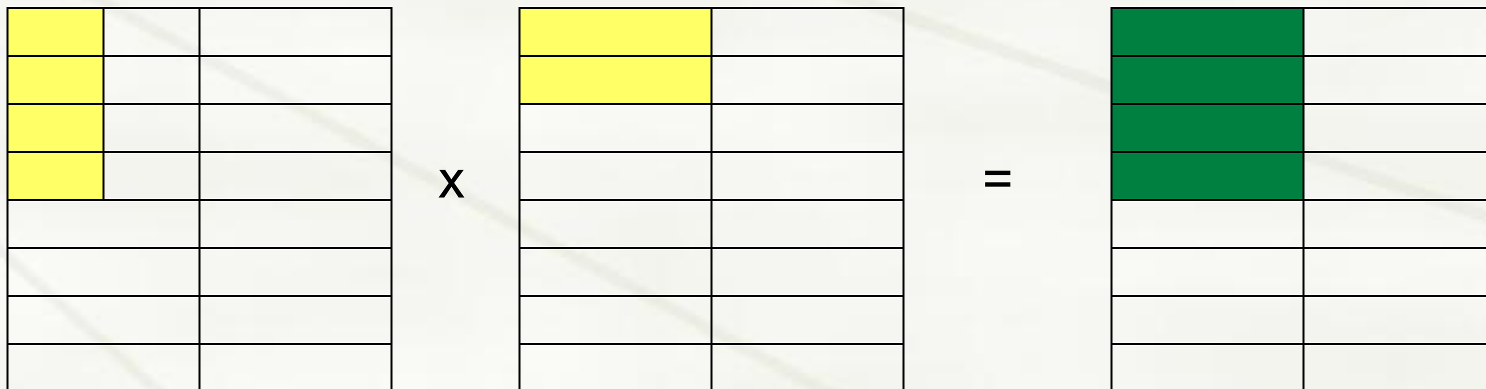
Cache pitfall in matrix-matrix multiply

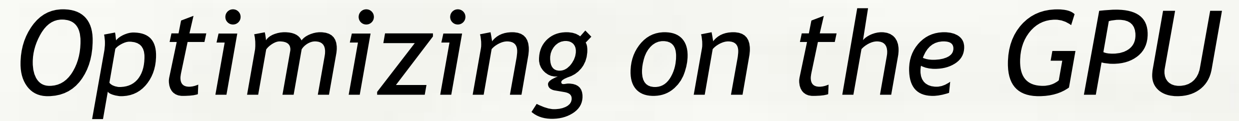
- Imagine each row in matrices below is 2 cache blocks
- To compute one element, need to read a column of one input matrix.
- For each element in the column read in, we fetch the entire contents of a block of which it is a part
- Strains bandwidth by requiring extra data
- Extra data in block is useless when fetched, and if the matrix is large it can be evicted from the cache before it is used.



Typical solution

- Use blocking to compute partial dot-products from submatrices
- Make sure that the total size of values processed in any of these “blocks” is no more than cache size
- Store partial sums in result
- Increases locality, as more data is used per block fetch
- Fewer data items need to be fetched twice now





- Fatahalian, et. al. tried:
  - blocked access to texture pixels
  - Unrolling loops
  - Single- and Multi-pass algorithms
    - Multipass references fewer rows/columns per pass
    - Expect higher hit rate within pass
  - Submatrix multiplication inside shaders (like blocking)
    - Hardware limitations on shader programs make this hard
- Unoptimized algorithms still yield best performance
- Hard to tell which optimizations to run, as cache parameters aren't public
  - Something like texture architecture we saw might lessen the effects of these optimizations

Performance

- ATLAS profiles a CPU and compiles itself based on cache parameters
  - Fully optimized to cache
  - Only ATIX800XT slightly outperforms ATLAS
- GPU measures do not count time for texture packing and transfer to GPU
  - ATLAS's full running time is measured
  - Tests conservatively favor GPU, so even worse than they look
- Why so bad?

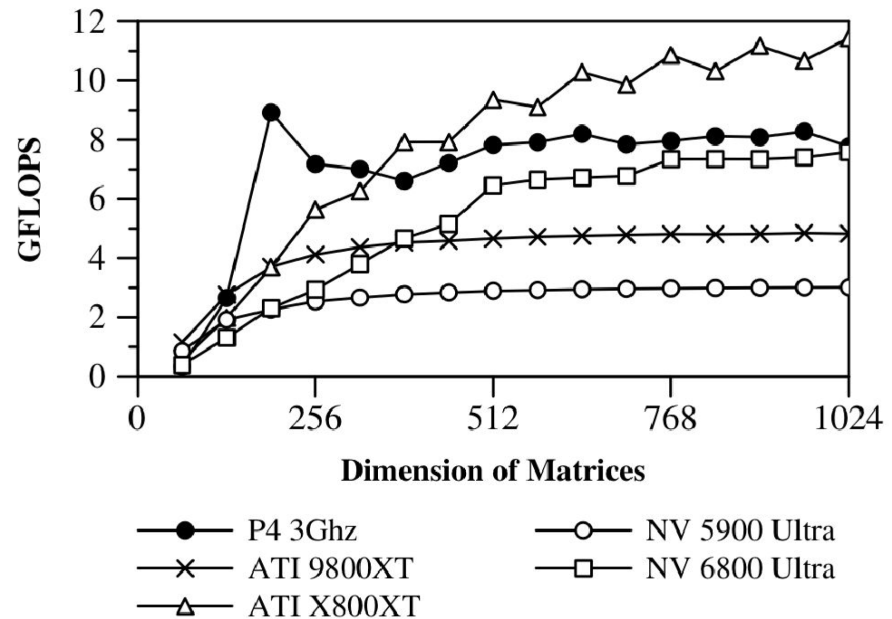


Figure 1: *Performance of multiplying square matrices on a 3 GHz Pentium 4, NVIDIA GeForceFX 5900 Ultra, prerelease GeForce 6800XT, ATI Radeon 9800XT, and prerelease Radeon X800XT.*

Bandwidth

- Cards aren't operating too far from peak bandwidth
  - ATI Multi is above 95%

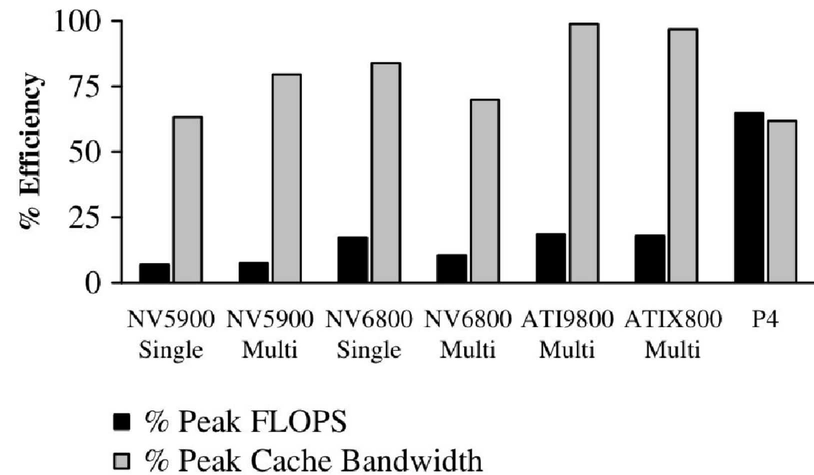


Figure 2: *Percentage computational and bandwidth efficiency when multiplying 1024x1024 matrices.*

- GPU’s get no better than 17-19% utilization of ALU’s for matrix multiplication
  - Implies we’re still not shipping enough useful data to the processor
- Available floating point bandwidth from closest cache on GPU is up to several times slower than CPU to L1 cache.
  - This will only get worse unless it’s specifically addressed
  - GPU computational speed is increasing faster than that of CPU (more cycles per cache access)

- Paper tried blocking within shaders
- Shaders have few registers available
  - For multiplying, can only manage two 6x6 matrices
  - Also, shaders do not allow many outputs
  - We can’t output the results so 6x6 is also out of reach
- Better shaders would allow us to do more computation on each item fetched
  - Compute to fetch ratio increases
  - Utilization of GPU resources increases
  - Currently have to fetch items more times than necessary due to these limitations

How to increase bandwidth

- Igehy, et. al. suggest:
  - Improve the cache
    - Wider bus to cache
    - Closer cache to the GPU
      - Naga mentioned in earlier lecture that texture cache is exclusive texture storage, but doesn’t run faster than memory.
  - Improve shaders
    - Make them capable of processing more data

Another alternative: Stream processing

- Dally suggests using stream processing for computation
  - Calls his architecture Imagine
- Eliminate load on caches by streaming needed data from unit to unit
  - GPU doesn’t do this: memory accesses go to common buffers
  - Dally proposes harnessing producer-consumer locality
    - Passing data between pipeline phases in stream processor
- Dally also points out, though, that GPU’s are *not* stream processors
  - Architecture is different in some fundamental ways: do we really want (or need) to change this?

- Mark Harris has the following to offer on Dally’s proposal:
  1. He's right that GPUs are not stream processors.
     - To the programmer, maybe, but not architecturally
  2. He oversimplifies GPUs in the interest of stream processors.
     - Understandable -- stream processors are his thing and GPU architectures are secret.
  3. Stream processors are a subset of data-parallel processors. GPUs are a different subset.
  4. GPU architecture is rapidly changing. Very rapidly. But they aren't exactly changing into stream processors like Imagine.
- Industry doesn’t seem to be heading in the streamed direction

Conclusions

- Bandwidth is the big problem right now
  - Not enough data to compute on per cycle
  - GPU ends up starved and waiting for cache
  - Need to change existing architecture or develop new one
- Knowing cache parameters and texture layout might also help
  - Typical matrix multiply doesn't optimize for something like Igehy's 6D blocking
- Will have to wait for hardware to change before we see fast numerical libraries on GPU.
  - Mark Harris at nVidia says he can't comment on specifics, but "expects things to improve"

1. B. Bershad, D. Lee. T. Romer, and B. Che. **Avoiding Conflict Misses Dynamically in Large Direct Mapped Caches**. Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, 1994.
2. T. Chilimbi, B. Davidson, and J. Larus. **Cache-conscious Structure Definition**. Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation
3. T. Chilimbi, J. Larus. **Using Generational Garbage Collection To Implement Cache-Conscious Data Placement**. International Symposium on Memory Management, 1998.
4. K. Fatahalian, J. Sugerman, and P. Hanrahan. **Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication**, Graphics Hardware 2004.
5. Z. S. Hakura and A. Gupta. **The Design and Analysis of a Cache Architecture for Texture Mapping**. 24th International Symposium on Computer Architecture, 1997.
6. Hennessy, J. and Patterson, D. **Computer Architecture: A Quantitative Approach**. Boston: Morgan Kaufman, 2003.
7. H. Igehy, M. Eldridge, and K. Proudfoot. **Prefetching in a Texture Cache Architecture**. EUROGRAPH, 1998.
8. K. Pettis & R. C. Hansen. **Profile Guided Code Positioning**. PLDI 90, SIGPLAN Notices 25(6), pages 16–27.
9. S. Yoon, B. Salomon, R. Gayle, and D. Manocha. **Quick-VDR: Interactive View-Dependent Rendering of Massive Models**, 2004.
10. NV40 architecture features, at http://www.digit-life.com/articles2/gffx/nv40-part1-a.html
11. Thanks to Mark Harris for additional input