Dark-box Remember

Project Footprint Document

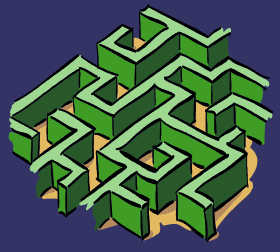
Graphic Processing Unit (GPU)
Memory Management System (MMS)

Jason S. Hardman

Dr. Kalpathi R. Subramanian

The University of N. Carolina at Charlotte

Department of Computer Science



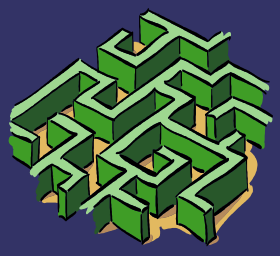
Current Industry Practices

GPGPU tradition holds that a data set is mapped to a 2D square, transformed by a monolithic fragment shader program, and then collected off the frame buffer.

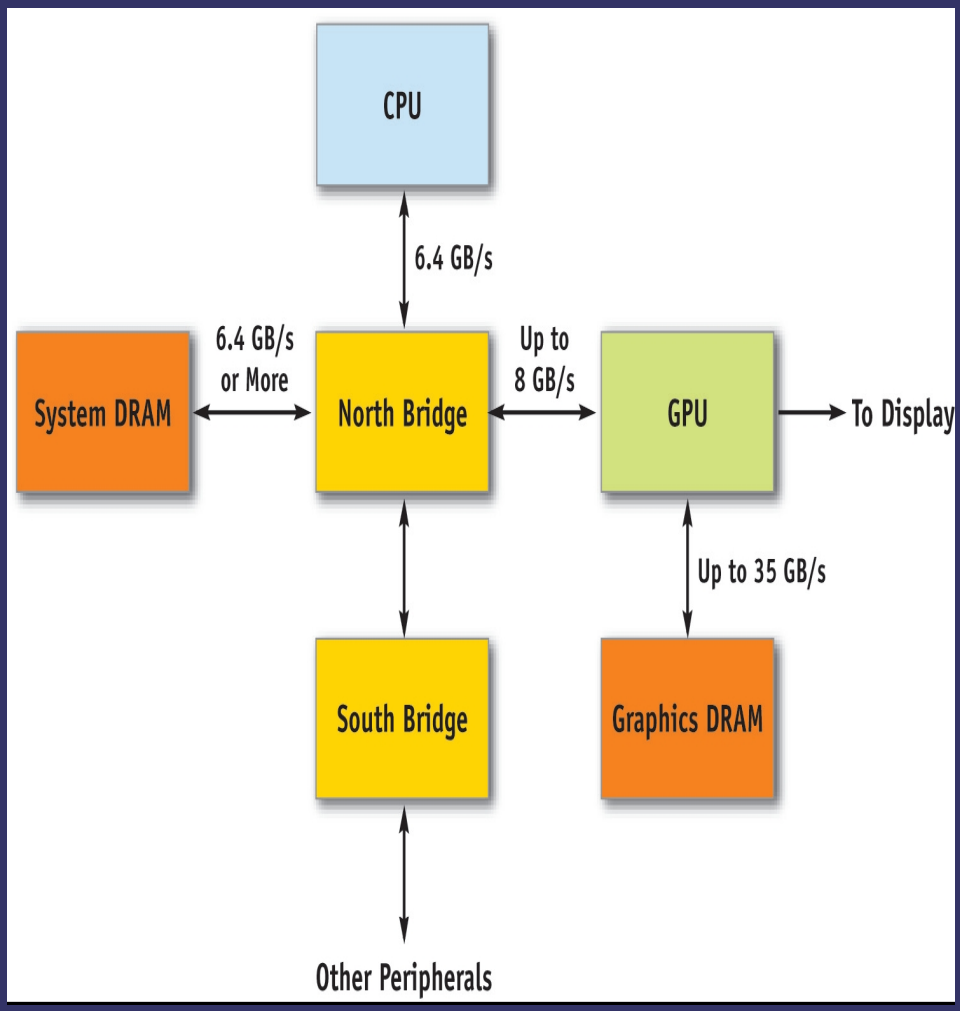
This presents problems for general data structures.

The GPU wants sequentially flowing data, or will loose most of its speed due to bridge thrashing and randomly accessing memory.

If we want to use the GPU for general purposes, we need to find a way to manage this limitation by eliminating the monolithic gpu application.

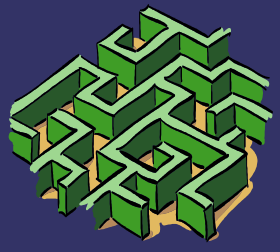


Justification for GPU-MMS



Data transfers from System DRAM (CPU memory) are bottle-necked at 6.4-8 GB maximum.

Internal (texture) memory organizations can utilize the full 35 GB/s rate of the GPU



Justification for GPU-MMS

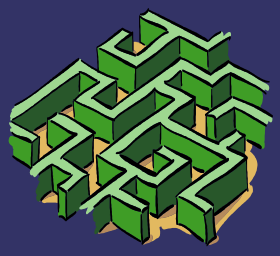
Available Memory Bandwidth in

Different Parts of the Computer System

Component	Bandwidth
GPU Memory Interface =	35 GB/sec
PCI Express Bus (×16) =	8 GB/sec
CPU Memory Interface =	6.4 GB/sec

(800 MHz Front-Side Bus)

Performance is maximized if we don't have to transfer our data back and forth across even the ultra-fast PCI Express bridge.



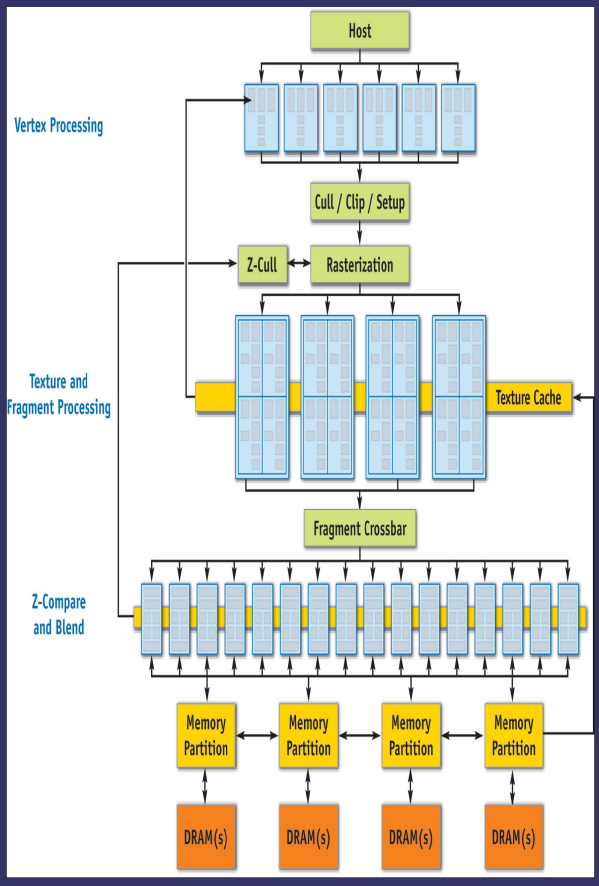
GPU-MMS Data Paths

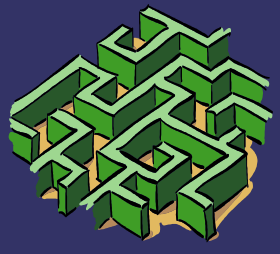
For the first time (nVidia 6 Series) we can read Texture memoryrender our Memory Partition data (output) back to Texture.

This allows us to completely avoid CPU-GPU data transfers except for:

Initially loading the GPU memory with data.

Recollecting that data for CPU purposes.





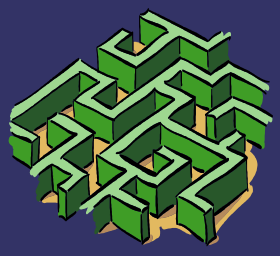
Notes

The CPU should be able to remotely control the GPU-MMS. Single control versus having two control models.

We can use the programmable vertex shader to reorganize our data by passing in signals.

We can control the vertex shader from the CPU by keeping an index of the GPU memory structure on the CPU.

The GPU can reorganize its own data by executing row/column folding and/or other reconciliatory instruction within the vertex shader.

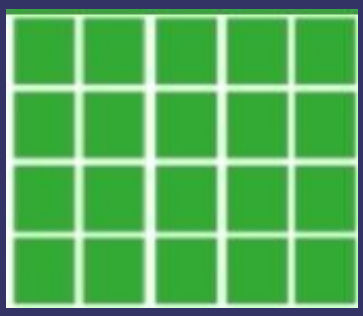


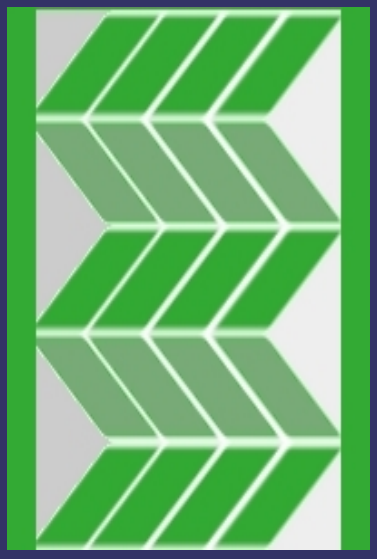
Algorithm Theory

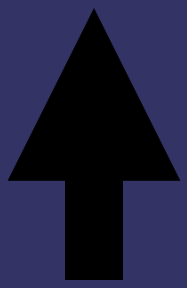
The CPU, by using a memory index, could construct a wave that would represent any combination of rows or columns. This wave could be sent to the vertex processor, which would warp the vertecies of our data-texture.

This warping would allow only the requested data-set to be eliminated by the rasterization process.

This operation is almost free, because values can still be passed on to the fragment processor for manipulation on the same pass.

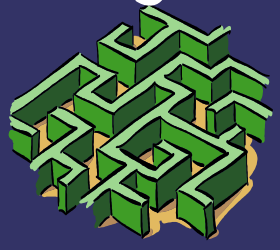






View Angle

View Angle



Render-To-Texture

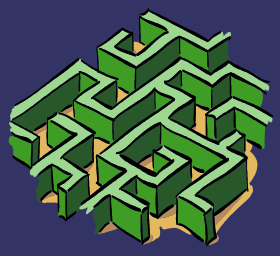
http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt

With the advancement of the new “Render-To-Texture” technology, we no longer have to endure the expense of reading the pBuffers.

Render-To-Texture allows us to render off-screen to framebuffer-attachable images, or in specific, images in texture-memory.

This creates a much faster, and more visually pleasing, method of collecting the post-processed data because it avoids the screen render and the expensive data-copy call, CopyTexSubImage. Uses a new object, renderbuffer.

Use **BindFramebufferEXT to bind.**



Example

We have a square object in 3D space with a data-set texture-mapped to it.

Pixel rows represent individual objects of a certain schema
Pixel columns represent individual properties of the schema.

We want to run some operation on every other row (object).
We define a saw-tooth wave and apply it to the vertecies.
We transform the data set 45 degrees along the x-axis.
We shrink our viewPort to fit the object to the screen

This leaves us looking orthographically at a set of steps, which is the exact data set that we wanted... Every other row.