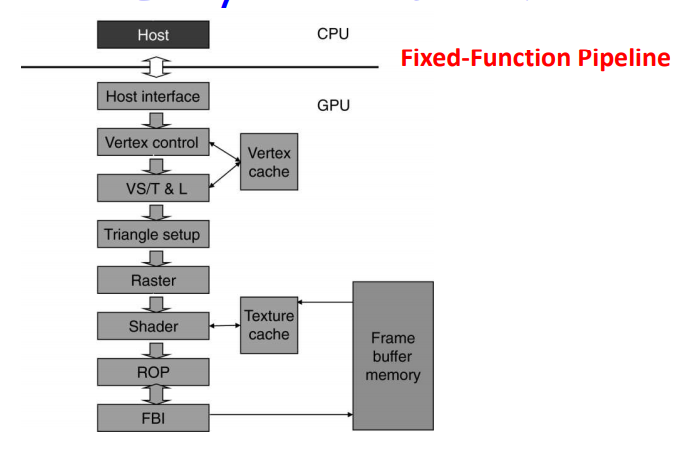
**GPU Background:**

Before GPUs introduce computer works on vertices to pixels that transforms done on CPU. Compute each pixel “by Hand”, in slow.



**Figure # 1:** Early 80s to late 90s GPUs

**Host interface:** Receives graphics commands and data from CPU.

**Vertex control:** Receives triangle data, converts them into a form that hardware understands and store the prepared data in vertex cache.

**VS/T & L:** Vertex shading transform and lighting, assigns per-vertex value.

**Triangle setup:** Create edge equations to interpolate colors across pixels touched by the triangle.

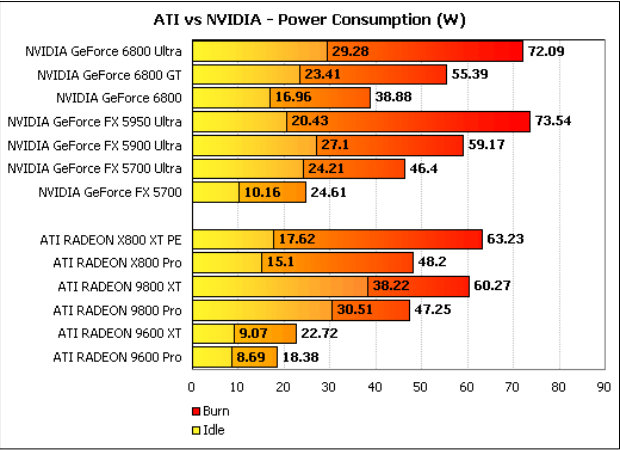
**Raster:** Determines which pixel falls in which triangle and for each pixel, interpolate per-pixel values.

**Shader:** Determines the final color of each pixel.

**Raster Operation Perform:** This performs color raster operations that blend the color of overlapping objects for transparency and antialiasing.

**Frame Buffer Interface:** This manages memory reads/ writes.

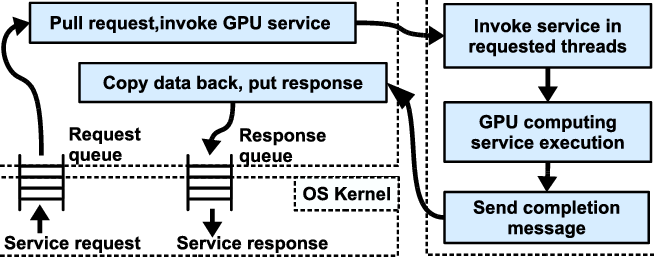
**Problem with GPUs Power:**



they need enough parallelism, under utilization and bandwidth to CPU. Data parallelism, programmability and much less restrictive instruction set includes in the art of GPUs.

Today graphics processing units are not only for graphics. they have found their way into fields as diverse as machine learning, oil exploration, scientific image processing, statistics, linear algebra, 3D reconstruction, Medical research and even stock options pricing determination. The GPU technology tends to add even more programmability and parallelism to a core architecture that is ever evolving towards a general-purpose CPU-like core.

**KGPU Framework Architecture:**



GPU is divided into three parts:

1. A module in the OS Kernel.
2. A user-space helper process.
3. NSK running on GPU.

GPU functions on the OS Kernel follows the steps:

1. The pinned-memory buffers and fills with there input that also requests a buffer for the result.
2. It builds a service request. Services are CUDA programs that have been per-load into NSK. For launch minimize time and that include a completion call back.
3. By the service request into request queue.
4. It will wait for the request to complete or either blocking until completion callback is called or busy-waiting on the response queue.

By the helper KGPU view the request queue in memory shared with the OS Kernel. A new service request comes the DMAs the input data buffer to the GPU using CUDA APIs. DMA completes the helper sends service request message to NSK using the message passing mechanism. The NSK sends a completion message to the CPU side and resumes polling for new request message. The OS kernel through their shared response queue that avoid copy between the kernel module and the user- space helper, the pinned data buffers allocated by CUDA driver are shared between two. The data of buffers locked in physical memory for manage it carefully.

On the CPU side buffers used for different purposes:

1. Preparing for a future service call by accepting data from a caller in the OS kernel.
2. To DMA input data from main memory to the GPU for the next service call.
3. To DMA results from the last service call from GPU memory to main memory.
4. Finishing a previous service call by returning data to the caller in the OS kernel.

Each performance will be done concurrently so along with the service currently running on the GPU the total depth of the service call pipeline is five stages. In the current KGPU prototype, we statically allocate four buffers, and each changes its purpose over time.

References:

1. <https://www.researchgate.net/figure/KGPU-framework-architecture_fig1_236834562>
2. <https://cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/lecture2.pdf>
3. <https://slideplayer.com/slide/3923513/>
4. <https://medium.com/altumea/a-brief-history-of-gpu-47d98d6a0f8a>