Fast DB Operations using GPU

By: Vaibhav Kaul

- **The Need for GPU’s in DB**
  - **Rapid growth of data volume**
    
    The rapid growth of data volume for the past decades has intensified the need for high-speed database management systems.
  
  - **Data warehousing and Data mining applications**
    
    Many database queries and, more recently, data warehousing and data mining applications, are very data and computation intensive and therefore demand high processing power.
  
  - **Main-memory databases**
    
    As the current trend of database architecture moves from disk-based system towards main-memory databases, applications have become increasingly computation- and memory-bound. Recent work investigating the processor and memory behaviors of current DBMSs has demonstrated the significant increase of query execution time due to memory stalls (due to data miss and instruction miss), branch mispredictions, and resource stalls (due to instruction dependencies and hardware specific characteristics).

- **About GPU’s**
  
  - **Stream Computing**
    
    In theory, GPUs are capable of performing any computation that can be mapped to the stream-computing model. The Stream computing model basically talks about SIMD instructions where the same set of instructions is performed on large number of data items simultaneously.
  
  - **Limitation**
    
    The programming model of GPUs is somewhat limited, mainly due to the lack of random access writes. This limitation makes it more difficult to implement many data structures and common algorithms such as sorting.
Frame Processors

The high throughput of the fragment processors and direct access to texture memory makes the fragment engines a powerful computation engine for certain numerical algorithms, including dense matrix-matrix multiplication, general purpose vector processing, visual simulation based on coupled-map lattices, linear algebra operations, sparse matrix solvers for conjugate gradient and multigrid, a multigrid solver for boundary value problems, etc.

Components of the Buffer

A GPU is designed to rapidly transform the geometric description of a scene into the pixels on the screen that constitute a final image. Pixels are stored on the graphics card in a frame-buffer.

The Frame buffer is further divided into the following parts:

- **Color Buffer**: Stores the color components of each pixel in the frame-buffer. Color is typically divided into red, green, and blue channels with an alpha channel that is used for blending effects.

- **Depth Buffer**: Stores a depth value associated with each pixel. The depth is used to determine surface visibility.

- **Stencil Buffer**: Stores a stencil value for each pixel. It is called the stencil buffer because it is typically used to enable/disable writing to portions of the frame-buffer.

The GPU Pipeline

![GPU Pipeline Diagram]
- **Vertex Processing Engine:** This unit receives vertices as input and transforms them to points on the screen.

- **Setup Engine:** Transformed vertex data is streamed to the setup engine which generates slope and initial value information for color, depth, and other parameters associated with the primitive vertices. This information is used during rasterization to construct *fragments* at each pixel location covered by the primitive.

- **Pixel Processing Engines:** Before the fragments are written as pixels to the frame buffer, they pass through the pixel processing engines or *fragment processors*. A series of tests can be used to discard a fragment before it is written to the frame buffer. Each test performs a comparison using a user specified relational operator and discards the fragment if the test fails.
  - **Alpha test:** Compares a fragment’s alpha value to a user-specified reference value.
  - **Stencil test:** Compares the stencil value of a fragment’s corresponding pixel with a user-specified reference value.
  - **Depth test:** Compares the depth value of a fragment to the depth value of the corresponding pixel in the frame buffer.

- **Visibility and Occlusion Queries**

  When a primitive is rasterized it is converted to fragments. Some of these fragments may or may not be written to pixels in the frame buffer depending on whether they pass the alpha, stencil and depth tests. An occlusion query returns the *pixel pass count*, the number of fragments that pass the different tests. They can be used to perform aggregation computations.

- **Stencil Operations**

  Graphics processors use stencil tests to restrict computations to a portion of the frame-buffer based on the value in the stencil buffer. Abstractly, we can consider the stencil buffer as a mask on the screen. Each fragment that enters the pixel processing engine corresponds to a pixel in the frame-buffer. The stencil test compares the stencil value of a fragment’s corresponding pixel against a reference value. Fragments that fail the comparison operation are rejected from the rasterization pipeline.
Stencil operations can modify on the stencil value of a fragment’s corresponding pixel. Examples of such stencil operations are:

- **KEEP**: Keep the stencil value in stencil buffer. We use this operation if we do not want to modify the stencil value.
- **INCR**: Increment the stencil value by one.
- **DECR**: Decrement the stencil value by one.
- **ZERO**: Set the stencil value to zero.
- **REPLACE**: Set the stencil value to the reference value.
- **INVERT**: Bitwise invert the stencil value.

```c
STENCILOP(Op1, Op2, Op3)
if (stencil test) /* perform stencil test */
   /* fragment passed stencil test */
   if (depth test) /* perform depth test */
      /* fragment passed stencil and depth test */
      perform Op3 on stencil value
   else
      /* fragment passed stencil test */
      /* but failed depth test */
      perform Op2 on stencil value
   end if
else
   /* fragment failed stencil test */
   perform Op1 on stencil value
end if
```
DB Operations

- **Comparison Between an Attribute and a Constant.**

```plaintext
COMPARE( tex, op, d )
1 CopyToDepth( tex )
2 set depth test function to op
3 RENDERQUAD( d )

CopyToDepth( tex )
1 set up fragment program
2 RenderTexturedQuad( tex )
```

COMPARE compares the attribute values stored in texture tex against d using the comparison function op. CopyToDepth called on line 1 copies the attribute values in tex into the depth buffer. CopyToDepth uses a simple fragment program on each pixel of the screen to perform the copy operation. On line 2, the depth test is configured to use the comparison operator op. The function RENDERQUAD(d) called on line 3 generates a fragment at a specified depth d for each pixel on the screen. Rasterization hardware compares the fragment depth d against the attribute values in depth buffer using the operation op.

- **Comparison Between 2 Attributes.**

The comparison between two attributes, ai op aj, can be transformed into a special semi-linear query (ai − aj op 0), which can be performed very efficiently using the vector processors on GPUs.
**Kth Largest Number**

KthLargest(tex, k)
1. b_max = maximum number of bits in the values in tex
2. x = 0
3. for i = b_max-1 downto 0
4. count = Compare(tex, $\geq$, x + $2^i$)
5. if count > k - 1
6. x = x + $2^i$
7. return x

KTHLARGEST computes the k-th largest attribute value in texture tex. It uses b_max passes starting from the MSB to compute the k-th largest number. During a pass i, it determines the i-th bit of the k-th largest number. At the end of b_max passes, it computes the k-th largest number in x.

- **Performance Measures**

![Graph showing copy time on GPU vs number of records]