AES ENCRYPTION AND DECRYPTION USING DIRECT3D 10 API

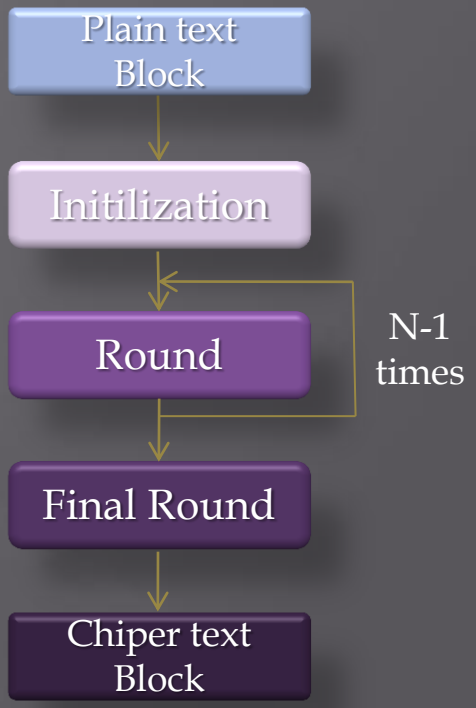
Chiuță Adrian Marius

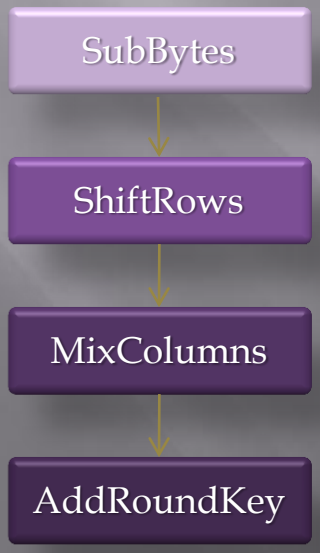
Agenda

- Advanced Encryption Standard
- AES optimizations
- Direct3D API
- AES Implementation using Direct3D 10 API
- Demo
- Conclusions

Advanced Encryption Standard

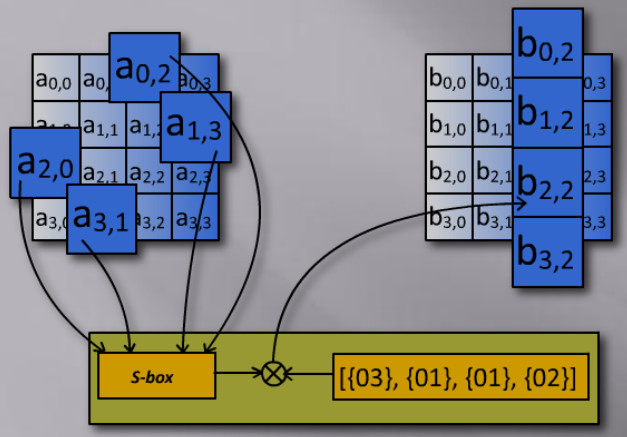
• It's an symmetric-key encryption standard comprised of block ciphers that operates on 128 bits of data and uses keys of 128, 192 and 256 bits.
• The encryption consists of a number of rounds, dependent of the size of the key.
• Before the encryption is started, the key is extended ,using a specific algorithm, in such a way that 128 bits from the extended key will correspond to initialization stage and to each round





• The initialization stage consists of initialization of state with the plain text adding the first 128 bits of the key to this state.
• One round consists of four transformations that applies to the state: ***SubBytes***, ***ShiftRows***, ***MixColumns*** and ***AddRoundKey***.
• The final round is special becaus it does not contain the ***MixColumns*** transformation.

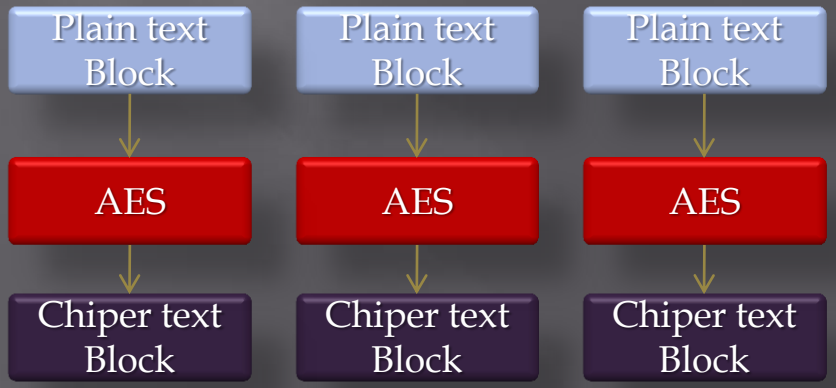
AES Optimizations



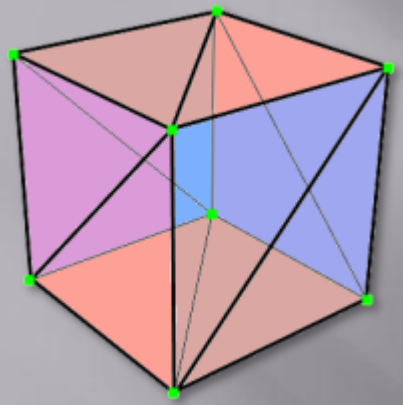
- One of the first optimizations that can be applied to AES algorithm is to combine the transformations ***SubBytes()***, ***ShiftRows()*** and ***MixColumns()*** into only one transformation. This can be acomplished by using Look Up Tables that contains the possible results for ***MixColumns()*** transformed by ***SubBytes().***
- Another optimization that can reduce the number of accesses to LUTs is to use 32 bits elements for the tables.
- These two simple optimizations will allow to transform 32 bits of the state using just four acceses to LUTs :

$$b_{3210} = LUT_0[a_0] \wedge LUT_1[a_1] \wedge LUT_2[a_2] \wedge LUT_3[a_3]$$

- For cases where more than one processing units are available, we can take advantage of this and encrypt/decrypt blocks in parallel.
- Parallel processing is suitable only for modes that don't have dependency between the blocks, for example: ECB, CTR and CBC (for CBC only decryption can be parallelized)



Direct3D API



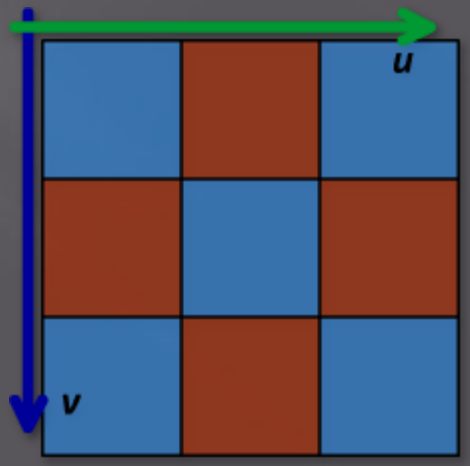
• Direct3D is an Application Programming Interface on Windows OS, that permits easy access to video card (GPU) resources .

• Starting with version 8, Direct3D includes support for ***shaders*** (small programs that run on GPU). Version 10 includes support for operations with integers and table look-ups using integer index.

To be able to use Direct3D 10, we need at least four types of resources that instructs the GPU what needs to be draw on the screen:

• ***Vertex buffers*** – vertex is a data structure that contains for a point at least its position in space (it can also contain other informations, like color or texture coordinate)

• ***Textures*** – a texture represent the pixels (an image) that can be mapped onto a surface. Every pixel can be accessed using the texture coordinates.



• ***Shaders*** – small programs that can be executed by the GPU for every vertex or every pixel that is draw (so we have vertex shaders and pixel shaders)

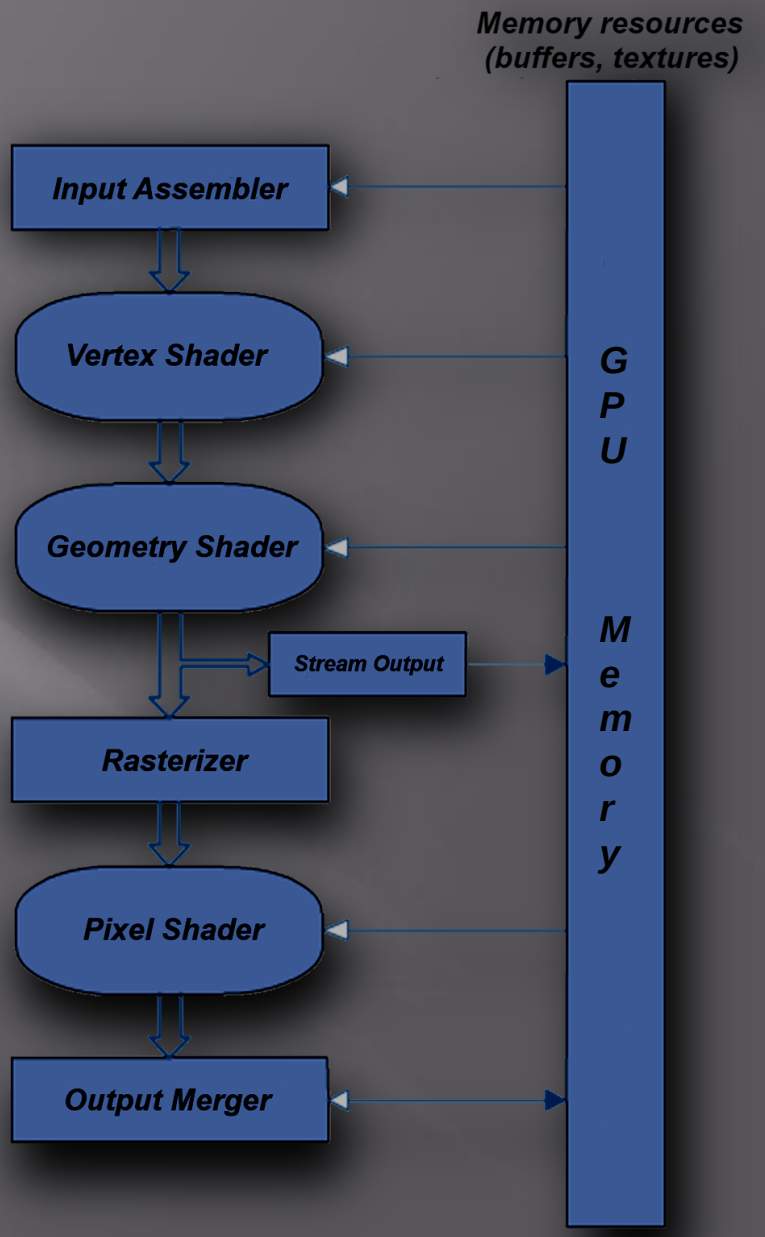
• ***Constant buffers*** – tables that cand be accessed by the shaders, but they can't be modified.

Direct3D API

Once the resources needed for drawing are created, they can be attached to the drawing pipeline and send drawing command to the GPU. Drawing can be seen as beeing done by a machine with a six stage pipeline (every stage of the pipeline can run in parallel with other stages, or even another instance of the same stage):

• *Input Assembler* – reads and interprets the vertices from the input vertex buffer and send each vertex to a vertex shader processing unit.

• *Vertex Shader* – for every vertex, executes a function that takes as input parameter the vertex and must return a data structure that must contain at least the position of the vertex (projected in the unit cube). Other information can also be returned, like the color or texture coordinate associated with the vertex.

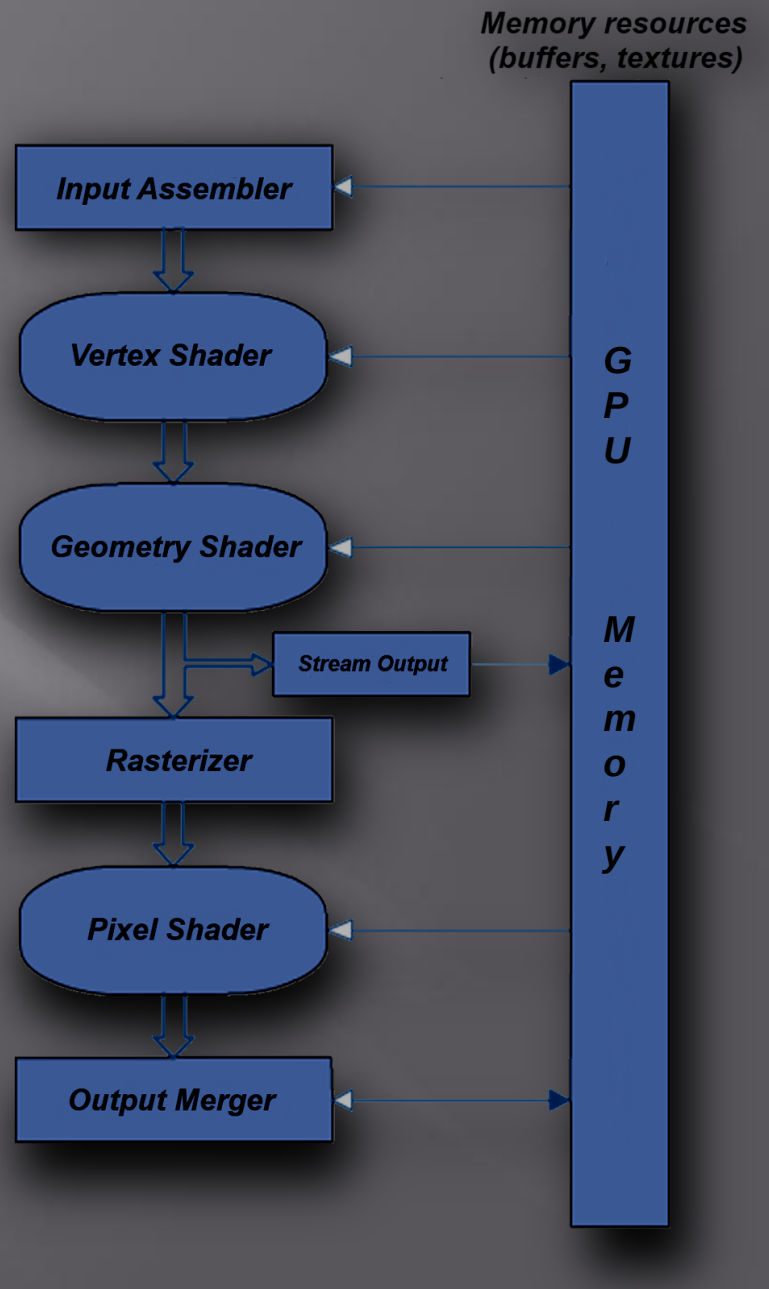


Direct3D API

•*Rasterization* - using the information returned by the vertex shader for three vertices that form a triangle, this stage determine the pixels that forms the triangle and interpolates the information returned by the vertex shader for each pixel of the triangle, and then sends there interpolated values to a pixel shader processing unit.

• *Pixel Shader* – for every pixel of the drawn triangle, executes a function that takes as input the interpolated data from the rasterizer. The pixel shader must return only one value: the color of the pixel.

• *Output merger* – receive the output of the pixel shader and also the position of the pixel from the rasterization stage and saves the value of the pixel in the output render target (texture).



AES Implementation using Direct3D 10 API

Because the GPU has a highly parallel architecture, but it does not offer special instructions that helps to speed up the AES algorithm, we will have to encrypt/decrypt blocks in parallel.

We will implement the encryption on the GPU equivalent to the following C code:

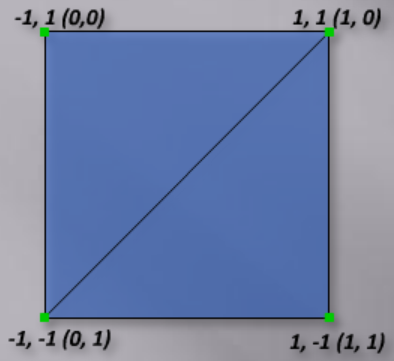
```
#pragma omp parallel for
for(uint blockID = 0; blockID < nbBlocks; blockID++)
        EncryptBlock(dst + blockID * BlockSize,  src + blockID * BlockSize);
```

To acomplish this, we need to setup the rendering pipeline in such a way that for each block that needs to be encrypted, we will be able to execute one shader.

The ideal shader for this task is the pixel shader because it only needs as input a texture coordinate associated for the pixel, and the input texture that contains the data that needs to be encrypted/decrypted.

The texture coordinate needed by the pixel shader can be generated using the vertex shader and the rasterizer (by interpolation).

AES Implementation using Direct3D 10 API



The resources needed to implement this on Direct3D 10, are:

• Two textures of equal sizes that will be used as input and output for AES algorithm. The format for the textures is DXGI_FORMAT_R32G32B32A32_UINT (a pixel represent four 32 bits unsigned integers), so an entire block that needs to be encrypted/decrypted is represented by exactly one pixel.

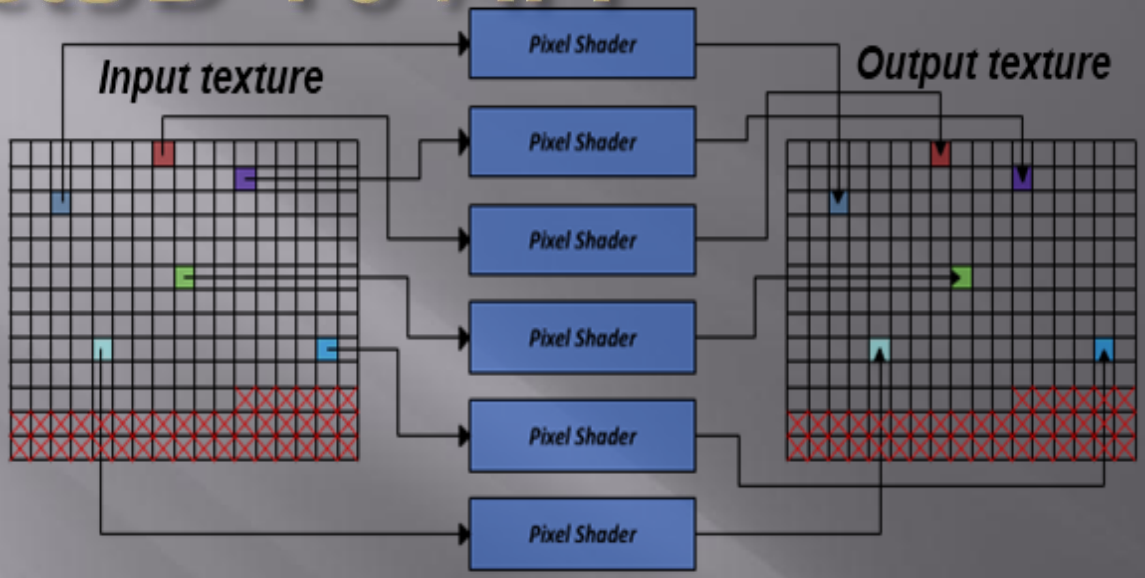
• A vertex buffer that describe a screen aligned quad that will cover the entire output surface. The vertices are already projected in the unit cube and each one contains the extreme texture coordinates.

• Constant buffers that contains the encryption key, and other LUTs needed by the algorithm.

• A vertex shader that only transmits to the rasterizer the position of the vertex and its associated texture coordinates.

• A pixel shader that executes for each pixel from the input texture and it computes the AES encryption/decryption on a single pixel/block and then sends the results to the output merger stage of the pipeline (this will write the resulted pixel in the output texture at the same position as the input pixel).

AES Implementation using Direct3D 10 API



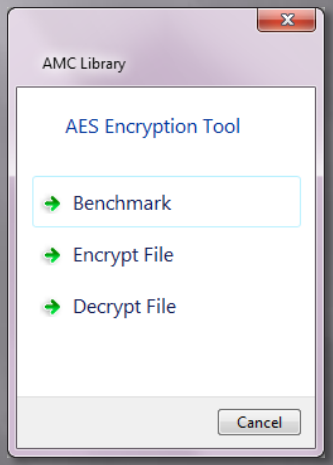
Once all the resources has been created and attached to the rendering pipeline, we only need to fill the input texture with the data that needs to be encrypted/decrypted, draw the screen aligned quad and read the results from the output texture.

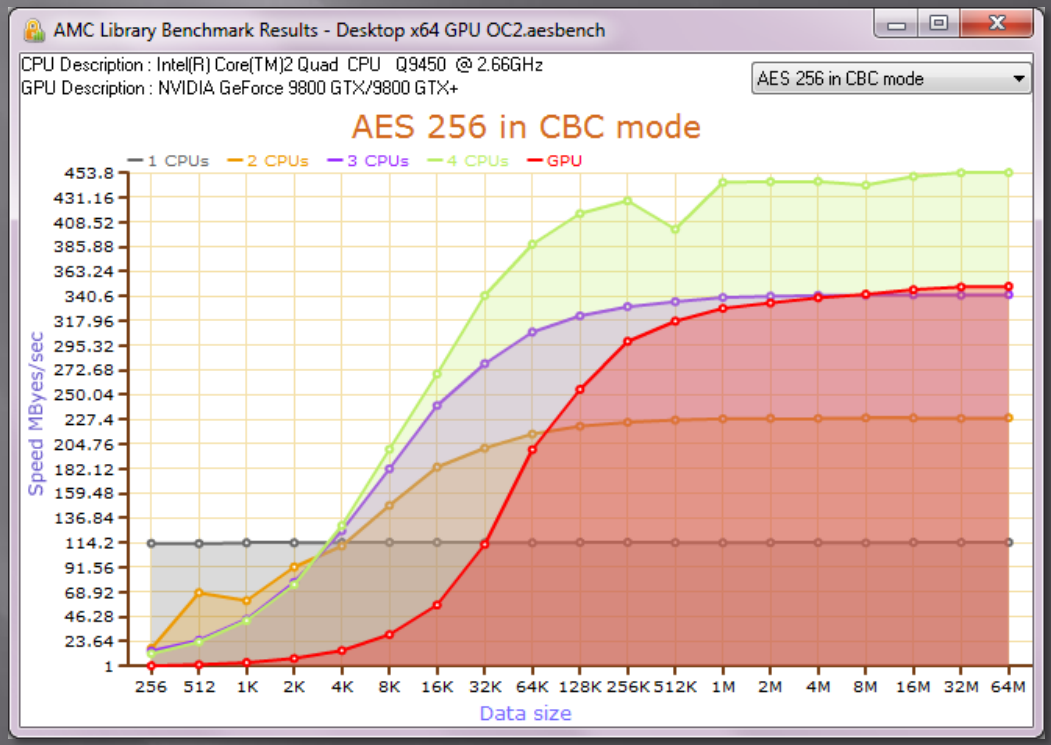
To create the shader that does the encryption/decryption, we only need to port the C code that does the encryption on a single block, taking into account the following:

- Reads from the input texture should be done using the ***Load*** method of the texture.
- The ID of the block that is encrypted/decrypted can be derived from the pixel shader input texture coordinates.
- Operations on vectors should be used where possible (are faster on GPUs that contains SIMD units).
- Random access to constant buffers can be very costly on some GPUs (is recommended to use texture in those cases).

AMC Tool

Demo





Conclusions

• Using Direct3D 10 API for implementing AES on GPUs, ensures that the algorithm will work on a large range of GPUs that implement the specifications for Direct3D 10.

• A large portion of the encryption/decryption time, is consumed by the upload and download of the data to/from the memory addressable by the GPU. This limitation is mainly caused by the speed of the PCI Express bus, all the data needed by the GPU and its commands are transferred using this bus.

• Encryption/decryption on the GPU is not recommended for a small number of blocks because of the overhead imposed by the data upload/download to the GPU memory and also GPU commands.

• The GPU can be successfully used in cases where large data sets need to be encrypted/decrypted with the same key and in the same time the CPU needs to be used for other tasks. One examples that meets these requirements is the case of back-up: the CPU does the compression of the data and the GPU does the encryption of the compressed data.