GigaVoxels/GigaSpace Device Function Pointers Benchmark

Pascal Guehl , Fabrice Neyret

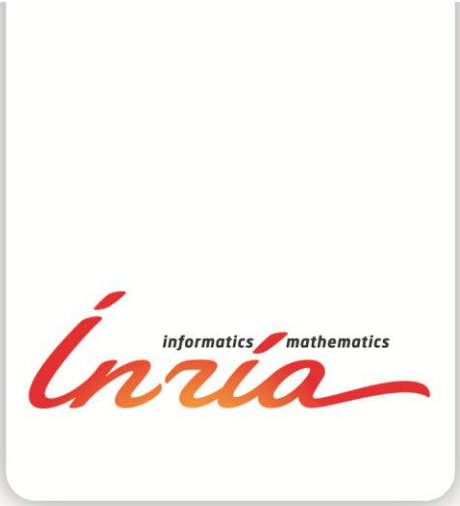
TECHNICAL REPORT

N° xxxx

13/01/2014

Project-Team Maverick

ISSN 0249-6399



GigaVoxels/GigaSpace

Pascal Guehl[1] , Fabrice Neyret [2]
Project-Teams Maverick

Technical Report N° xxxx — 13/01/2002014 —14 pages.

Abstract: This the work done during the RTIGE ANR.

Key-words: insérez ici les mots-clés en anglais

[1] Pascal Guehl affiliation – pascal.guehl@inria.fr

[2] Fabrice Neyret affiliation – fabrice.neyret@inria.fr

RESEARCH CENTRE
GRENOBLE - RHÔNE-ALPES

Inovallée
655 avenue de l'Europe - Montbonnot
38334 Saint Ismier Cedex France

GigaVoxels/GigaSpace

Résumé : Benchmark report and analysis of using device function pointers in CUDA kernels to be able to customize the GigaVoxels/GigaSpace API.

Mots clés : insérez ici les mots-clés en français

1. Introduction

This is a benchmark report on the use of function pointers of device functions in CUDA kernels. Function pointers are well suited to customize code of library in a dynamically way. This report analyses the CUDA API way to use device function pointers and the impact on performances.

2. Using function pointers

The CUDA programming does not explain something about function pointers, except that device function pointers requires SM 2.0 compute capability. This is the minimum requirements of GigaVoxels, the same one for using writable textures (i.e. "surfaces"). The CUDA GPU Computing SDK Samples show one example in which user can dynamically modify filters for image processing (sobel, etc...).

Device function pointers can be declared as :

```
__device__ unsigned char ( *f )( unsigned int pSize, const unsigned char* pInput );
```

Or with a "typedef" :

```
typedef unsigned char ( *FunctionPointerType )( unsigned int pSize, const unsigned char* pInput );
__device__ FunctionPointerType _d_algorithmFunction = NULL;
```

It is possible to create function pointers table with an array.
User can modify function pointers dynamically from host (i.e. CPU).

3. Implementation

a. Source code

Here is the source code of the kernel with one device function call "`algorithmFunction()`" (either a pointer or not).

```
/********************************************************************************
 * Standard kernel call with one function pointer
 *
 * @param pSize number of elements to process
 * @param pInput input array
 * @param pOutput output array
 ********************************************************************************/
__global__
void Kernel_StandardAlgorithm( const unsigned int pSize, const unsigned char* pInput, unsigned char* pOutput
)
{
    // Retrieve global data index
    const unsigned int index = threadIdx.x + blockIdx.x * blockDim.x + blockIdx.y * blockDim.x * gridDim.x;

    // Check bound
    if ( index < pSize )
    {
        // Compute value
          const unsigned char value = algorithmFunction( index, pInput );

          // Write value
          pOutput[ index ] = value;
    }
}
```

Here is the modification of code for the kernel with two device functions calls.

```
...

// Compute value
const unsigned char value = algorithmFunction( index, pInput );

// Compute value
const unsigned char value_2 = algorithmFunction_2( index, pInput );

// Write value
pOutput[ index ] = value + value_2;

...
```

Now, here is source of the two device functions called by previous kernels.

```
/********************************************************************************
 * Algorithm function
 *
 * @param pIndex index of the element to process
 * @param pInput input array
 ********************************************************************************/
__device__
unsigned char algorithmFunction( const unsigned int pIndex, const unsigned char* pInput )
{
    unsigned char value = (uchar)( ( sinf( pInput[ pIndex ] * 3.141592f / 180.f ) * 0.5f + 0.5f ) * 255.f );
    return value;
}
__device__
unsigned char algorithmFunction_2( const unsigned int pIndex, const unsigned char* pInput )
{
    unsigned char value = (uchar)( ( cosf( pInput[ pIndex ] * 3.141592f / 180.f ) * 0.5f + 0.5f ) * 255.f );
    return value;
}
```

b. Device function declaration

For the moment, device functions are initialized at compilation. It could be interesting to benchmark when user modify functions from host. To do that, user has to modify the address of the CUDA symbol associated to the function pointer with a call to "`cudaMemcpyToSymbol()`".

```
/**
 * Algorithm function
 */
__device__ unsigned char algorithmFunction( const unsigned int pIndex, const unsigned char* pInput );
__device__ unsigned char algorithmFunction_2( const unsigned int pIndex, const unsigned char* pInput );

/**
 * Define a function pointer type that will be used on host and device code
 */
typedef unsigned char (*FunctionPointerType)( const unsigned int pIndex, const unsigned char* pInput );

/**
 * Device-side function pointer
 */
//__device__ FunctionPointerType _d_algorithmFunction = NULL;
__device__ FunctionPointerType _d_algorithmFunction = algorithmFunction;
__device__ FunctionPointerType _d_algorithmFunction_2 = algorithmFunction_2;
```

c. Kernel execution parameters

In this test, each kernel is processing 1.000.000 elements distributed over a 1D grid of 3906 blocks of 256 threads. Each 1D block is further divided into 8 warps of 32 threads.

```
// Setup kernel execution parameters
const unsigned int cNbElements = 1000000;
dim3 gridSize( cNbElements / 256 + 1, 1, 1 );
dim3 blockSize( 256, 1, 1 );
```

4. Benchmarks

Benchmarks are expressed in milliseconds (ms). Each kernel is launched 25 times and time is average.

a. Using one function pointer

Hardware	Nb multi processors	Compute capability	Simple kernel	Kernel with function pointer	Overhead
GeForce GTX 670 (2GB)	7	3.0	0.105978 ms	0.124259 ms	17.25 %
GeForce GTX 570 (1.25 GB)	15	2.0	0.119132 ms	0.145857 ms	22.43 %
GeForce GT 540M (1 GB) - laptop	2	2.1	0.775214 ms	0.960614 ms	23.92 %

b. Using two function pointers

Hardware	Nb multi processors	Compute capability	Simple kernel	Kernel with function pointers	Overhead
GeForce GTX 670 (2GB)	7	3.0	0.128996 ms	0.220588 ms	71.00 %
GeForce GTX 570 (1.25 GB)	15	2.0	0.170374 ms	0.26454 ms	55.27 %
GeForce GT 540M (1 GB) - laptop	2	2.1	1.09315 ms	1.80992 ms	65.57 %

c. Other

Note : "strange behavior" - A ne pas prendre en compte pour l' instant...

When compiling the source code with only one function pointer, code seems to be faster ? I need to investigate that, and redo the test. Don' t take that into account for the moment. Here was the result :

Hardware	Simple kernel	Kernel with function pointer	Overhead
GeForce 670 / 2GB	0.31... ms	0.33... ms	
Geforce GTX 570			
GeForce GT 540M / 1 GB (laptop)	1.80934 ms	1.94048 ms	7.25 %

5. Comparison : using function pointers in GLSL

Note : This part is based on the OpenGL 4.0 core features.

Ca peut être intéressant de voir le coût chez GLSL pour comparer. Par exemple si c' est plus rapide en GLSL, ça veut peut être dire que CUDA à de la marge pour rattrapper ça par la suite.

a. Source code

In GLSL, function pointers are declared with "subroutine" keyword.

Say we have two functions that take current pixel position and generate the associated ray from camera, one for classical mode and one for "fish eye" appearence. We can declare a common function pointer that user could change on-the-fly.

```
// Example of prototype of a user function pointer
subroutine vec3 getRayDirectionSubRoutineType( vec2 uv );

// GLSL requires to declare a uniform of that type to be able to modify the function pointer index on the fly
with a call to glUniform
subroutine uniform getRayDirectionSubRoutineType getRayDirection;
```

On the OpenGL client side, the API enables user to modify pointer on-the-fly with the help of the "glGetSubroutineIndex" and "glUniformSubroutinesuiv" functions.

```
// Retrieve the index of a subroutine uniform of a given shader stage within a program
GLuint fishEyeRayDirectionSubroutineIndex = glGetSubroutineIndex( _shaderProgram, GL_FRAGMENT_SHADER,
"getFishEyeRayDirection" );

// load active subroutine uniforms
glUniformSubroutinesuiv( GL_FRAGMENT_SHADER, 1, &fishEyeRayDirectionSubroutineIndex );
```

Main "fragment shader" function :

```
void main()
{
    // Retrieve sample point in NDC space [-1;1]
    vec2 uv = ( gl_FragCoord.xy - uImageResolution * 0.5 + vec2( 0.5 ) );
    uv = uv / ( uImageResolution * 0.5 );

    // Generate ray direction
    vec3 rayDirection = getRayDirection( uv );

    // Write output data
    oData = vec4( rayDirection, 0.0 );
}
```

Method to compute ray direction :

```
////////////////////////////////////////////////////////////////////////////////
// Generate ray direction
////////////////////////////////////////////////////////////////////////////////
subroutine( getRayDirectionSubRoutineType )
vec3 getClassicalRayDirection( vec2 uv )
{
    return ( uv.x * uCameraXAxis + uv.y * uCameraYAxis - /*uCameraViewPlaneDistance*/2.0 * uCameraZAxis );
}
```

Method to compute "fish eye" ray direction :

```
////////////////////////////////////////////////////////////////////////////////
// Generate ray direction with Fish Eye nonlinear projection
////////////////////////////////////////////////////////////////////////////////
subroutine( getRayDirectionSubRoutineType )
vec3 getFishEyeRayDirection( vec2 uv )
{
    vec3 rayDirection = vec3( 0.0 );

    float r_squared = uv * uv;
    if ( r_squared <= 1.0 )
    {
        // psi( r ) = r * psiMax
        float r = sqrt( r_squared );
        float psi = r * radians( uFishEyePsiMaxAngle );

        float sin_psi = sin( psi );
        float cos_psi = cos( psi );

        float sin_alpha = uv.y / r;
        float cos_alpha = uv.x / r;

        rayDirection = sin_psi * cos_alpha * uCameraXAxis + sin_psi * sin_alpha * uCameraYAxis - cos_psi *
uCameraZAxis;
    }

    return rayDirection;
}
```

b. Benchmarks

Partie à completer...

I. Using one function pointer

Hardware	Nb multi processors	Compute capability	Simple kernel	Kernel with function pointer	Overhead
GeForce GTX 670 (2GB)					
GeForce GTX 570 (1.25 GB)					
GeForce GT 540M (1 GB) - laptop					

II. Using two function pointers

Hardware	Nb multi processors	Compute capability	Simple kernel	Kernel with function pointer	Overhead
GeForce GTX 670 (2GB)					
GeForce GTX 570					

(1.25 GB)					
GeForce GT 540M (1 GB) - laptop					

Conclusion

Based on this first benchmark, it' s seems that device function pointers in CUDA generate not negligible overhead.

The difference of using 1 or 2 pointers is really huge, what is really surprising ?

But it should be interesting to test them in real GigaVoxels two situations, where their costs might be amortized :

- for the producer,
- for the renderer.

For the producer it could be OK and very fast to test (ex : modify spatial oracle, etc...). For instance, it' s easy to test a producer with device function pointers for the oracle, and spatialTest like isInSphere(), isInCube(), etc...

For the rendering stage, I' m not sure on the impact. This is actually the main bottleneck of GigaVoxels.

Bibliography

[1] NVIDIA – CUDA C Programming Guide – *doc*

[2] NVIDIA – NVIDIA CUDA Samples Browser v5.0 – "Function Pointers" example

RESEARCH CENTRE
GRENOBLE - RHÔNE-ALPES

Inovallée
655 avenue de l'Europe - Montbonnot
38334 Saint Ismier Cedex France

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr
ISSN 0249-6399