GRAPHICS PROCESSING UNITS: MORE THAN THE PATHWAY TO REALISTIC VIDEO-GAMES

UNIDADES DE PROCESAMIENTO GRÁFICO: MÁS QUE LA RUTA HACIA JUEGOS DE VIDEO CON REALIDAD VISUAL Y DE MOVIMIENTO

CARLOS TRUJILLO
Facultad de Minas, Universidad Nacional de Colombia, Medellín-Colombia, 050034, catrujila@unal.edu.co

JORGE GARCIA-SUCERQUIA
Universidad Nacional de Colombia, Medellín-Colombia, 050034, jigarcia@unal.edu.co

Received for review Mach 20th, 2011; accepted May 19th, 2011; final version May, 31th, 2011

ABSTRACT: The huge video games market has propelled the development of hardware and software focused on making the game environment more realistic. Among such developments are graphics processing units (GPUs). These devices are intended to alleviate the central processing unit (CPU) of the host computer from the computation that creates “life” for video games. The GPUs reach this goal with the use of multiple computation cores operating on a parallel architecture. Such features have made the GPUs attractive for more than the development of video games. In this paper, a brief description of the features of compute unified device architecture (CUDA™), a GPU parallel computing architecture, is presented. The application of GPUs on the numerical reconstruction of holograms from a digital in-line holographic microscope is shown. Upon the completion of this experiment, we reached an 11-fold acceleration with respect to the same calculation done on a typical CPU.

KEYWORDS: graphics processing units, parallel programming, CUDA™, single instruction multiple data (SIMD), processing thread, diffraction integrals, numerical hologram reconstruction

RESUMEN: El amplio mercado de los juegos de video ha impulsado un acelerado progreso del hardware y software orientado a lograr ambientes de juego de mayor realidad. Entre estos desarrollos se cuentan las unidades de procesamiento gráfico (GPU), cuyo objetivo es liberar la unidad de procesamiento principal (CPU) de los elaborados cómputos que proporcionan “vida” a los juegos de video. Para lograrlo, las GPUs son equipadas con múltiples núcleos de procesamiento operando en paralelo, lo cual permite utilizarlas en tareas mucho más diversas que el desarrollo de juegos de video. En este artículo se presenta una breve descripción de las características de compute unified device architecture (CUDA™), una arquitectura de cómputo paralelo en GPUs. Se presenta una aplicación de esta arquitectura en la reconstrucción numérica de hologramas, para la cual se reporta una aceleración de 11X con respecto al desempeño alcanzado en una CPU.

PALABRAS CLAVE: unidades de procesamiento gráfico, programación paralela, CUDA™, single instruction multiple data (SIMD), hilos de procesamiento, integrales de difracción, reconstrucción numérica de hologramas.

1. INTRODUCTION

The demand for powerful computer systems is rising every day.This situation is evident in scientific, industrial, commercial, and even home environments. Traditionally, the capability of computing relies on the processor. For the latter the power of computing is proportional to the number of transistor per unit area into the chip. This is why hardware architectures try to increase the density of transistors, but since 2006 approximately the number of these semiconductors per unit area was not able be increased any more due to physical limitations such as, for instance, diffraction-limited photolithography [1]. As a consequence, the use of alternative processing paradigms has come to have particular importance.

It is expected that the demand for faster, cheaper, lighter, and more portable computing systems will increase in the future. Just to mention one example, video game consoles with the features mentioned above will be earnestly sought out by video game lovers. Proposed answers to these daunting demands can be classified into two main types: software and hardware. The answers that are software oriented consist of *algorithm optimization.* Such optimization is designed with the purpose of diminishing the number of operations to be executed. The hardware-based solutions rely on computing technology improvements that result in a faster execution of instructions

An alternative solution, compatible with the mentioned above is *parallel computing*. This computation paradigm splits the set of data to be computed into equal sub-sets of data that are concurrently processed by a specialized hardware. A fairly new approach to increase the processing power by means of parallel computing is the one that runs on graphics processing units (GPUs).

A GPU is a specialized processor that offloads most of the graphics tasks from the central processor unit (CPU). Intense 3D rendering applications, like what is often used on video-games, are the kind of tasks that are offloaded from the CPU. Realistic video-games are based on rendering process.Hence, the current demands for more realistic video games require exhaustive rendering processing, which can only be provided by GPUs. As a direct consequence, the video-game market has propelled the hardware developers to build even more powerful GPUs in very compact designs. These features make GPUs attractive for more than just graphics calculations. Modern GPUs are very efficient with manipulating floating point operations, and their multi-core structure make them more effective than general-purpose CPUs for a range of complex computation duties [2].

Computing is evolving from central processing carried out entirely by the CPU, to co-processing between the CPU and the GPU. The CPU takes care of data-flow control, the fetching and the decoding of the instructions that have to be performed. The GPU executes them efficiently thanks to the highly parallel nature of its architecture.

General-purpose computing on a GPU [2], namely GPGPU, is a GPU- oriented concept which uses the multithread architecture of GPUs for parallel computing by means of single-instruction multiple-data (SIMD). Its performance is based on executing the same set of instructions on multiple data by individual threads; *stream processing* [3] is another name for this computer programming paradigm. Many applications that process large data sets can use stream processing to accelerate their calculations. For instance, in 3D rendering, large sets of pixels and vertices are mapped to parallel threads for their execution. Similarly, applications for image processing and general media processing such as image post-processing, video encoding and decoding, image scaling, 3D vision and pattern recognition, can map pixels to parallel processing threads [4]. In fact, not only image processing algorithms are accelerated by parallel data processing: from the general processing of signals for simulating physical phenomena [5], to financial engineering [6], and computational biology [7] we find areas which are counted among those highly influenced by the current trend of GPGPU application.

In this paper, an introduction to a general purpose parallel computing architecture named compute unified device architecture (CUDA™) made by NVIDIA® is presented.This architecture has a new parallel programming model and its own set of instructions to control the parallel computing engine from NVIDIA® [4]. As a particular application, an implementation in CUDA™ which accelerates the calculation of diffraction integrals operating over up to four million pixels is shown. The implemented calculation on the GPU is 11 times faster than that which runs on a typical CPU.

2. DIFFERENCES BETWEEN A CPU AND A GPU

A GPU has more transistors devoted to data processing than a CPU [4]. This fact means that for massive floating point operations, like graphics rendering, a GPU expends less transistors on flow control and cache memory than a CPU. This feature makes GPUs the appropriated tool to run highly parallelized algorithms that handle very efficiently thousands of complex floating point operations. Figure 1 illustrates the difference in the way how the silicon area is distributed on a CPU and a GPU;

notice the larger area devoted to arithmetic and logic units (ALU) in a GPU than in a CPU.

NVIDIA® chips are built based on multiprocessors. These devices have about ten cores, hundreds of ALUs, several thousand registers and tens of kilobytes of shared memory. Also, the GPUs contain global memory, which can be used by all multiprocessors, local memory in each multiprocessor, and a special memory area for constants.

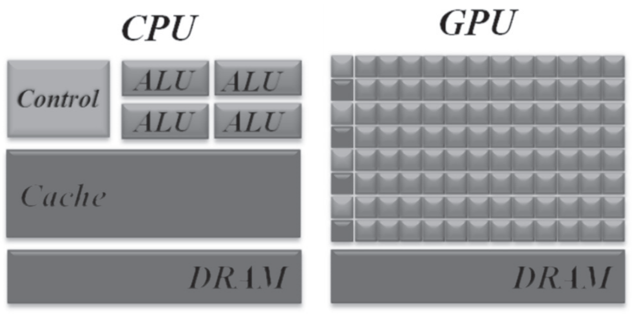


Figure 1. Silicon area distribution for the CPU and GPU.

As product of the hardware differences, the cores of a CPU -up most four for the quad processors- are designed to execute a single thread per core but a very high speed. A GPU, on the other hand, is designed for fast execution of many parallel threads, running on hundreds of ALUs. Memory operations are distinct in GPUs compared with that in CPUs. For example, normally a CPU does not contain memory controllers, and in fact, all GPUs have several of them. Additionally to use faster memory, a GPU has larger memory bandwidth than a CPU, which is important at the moment of processing large data sets in parallel.

CPUs use a large number of their transistors for data control and large amount of on-chip cache memory. The cache memory is needed for accelerating the execution of a few instructions streams. In contrast, GPUs mainly use theirs transistors for multiprocessors, which are composed of lots of ALUs, shared memory and memory controllers. In Figure 1, the above differences are shown in terms of the proportion of transistors devoted to each task. The GPUs´ features mentioned above do not accelerate the execution of individual processing threads, but enable to process concurrently thousands of threads by taking advantage of the high memory bandwidth.

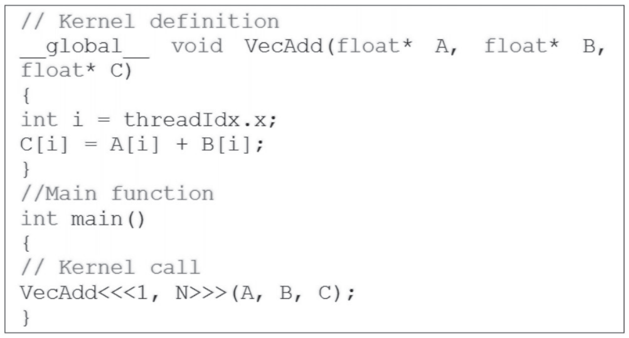
The extended use of the cache memory in the CPU is oriented to increase their performance due to low the latencies associate with this kind of memory. The use of the shared memory, namely the GPU cache, in a graphics card is to increase memory bandwidth. In a CPU, the larger the cache, the shorter latencies are. In a GPU, the memory access latencies are hidden by the simultaneous execution of many threads.

In graphics processing, the whole GPU hardware and design features described above, make it capable of receiving a group of primitives, performing all the needed operations, and then rendering output pixels very efficiently. The primitives are processed independently, concurrently and separately from each other. GPGPU goes a little further: the primitives are considered as processing threads that can be used for much more than graphics processing. This feature constitutes the key to extend the GPUs to other environments rather than the development of realistic video-games.

3. CUDA PROGRAMMING MODEL WITH BASIC EXAMPLES.

In 2007 NVIDIA® launched a parallel computing architecture for its GPUs named CUDA™ [4]. CUDA™ uses a software environment that enables developers to use C as the high level programming language. CUDA C is the actual programming language that enables the programmer to use CUDA^{TM}. CUDA C extends C by adding a fundamental concept, the *kernel*, along with a group of functions that make CUDA™ be similar to a huge library.

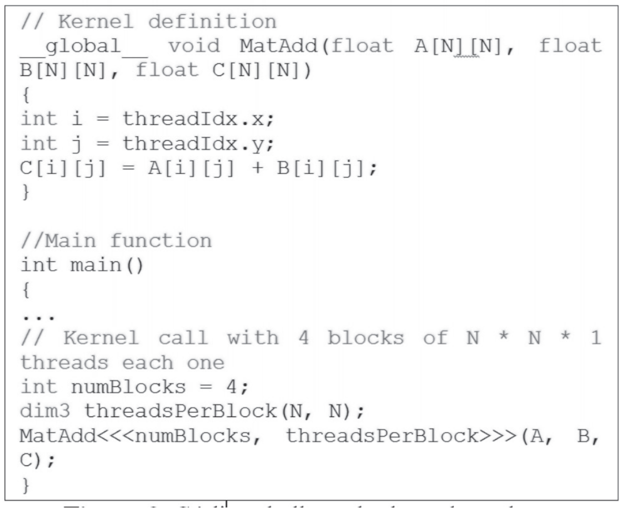
When a kernel is called, it is executed *N* times in parallel by *N* different CUDA™ threads [4]. A kernel is defined by declaring *__global__* before its name, followed by the used variables. Calling a kernel follows the same procedure of calling a normal C function, but adds a new syntax: <<<...>>>. Within these brackets is written the number of parallel threads that execute the kernel, an example of this calling is the code shown in Figure 2. A CUDA™ thread or parallel thread is an abstract concept that represents the task that is executed on each one of the CUDA™ cores; these cores are inside the streaming multiprocessors of a GPU [3]. The number of task that can be executed in parallel is given by the features of the GPU; currently, an average GPU has around one hundred of CUDA™ cores.



```
// Kernel definition
__global__ void VecAdd(float* A, float* B,
float* C)
{
int i = threadIdx.x;
C[i] = A[i] + B[i];
}
//Main function
int main()
{
// Kernel call
VecAdd<<<1, N>>>(A, B, C);
}
```

Figure 2. Basic Code for a kernel call.

In Figure 2 is shown the typical way of using CUDA™. That is, in linear fashion, because *VecAdd()* function is executed *N* times only. The first argument within the brackets denotes the dimensions in which the thread blocks are organized. If it is set to 1, a row of threads are called to execute the kernel.

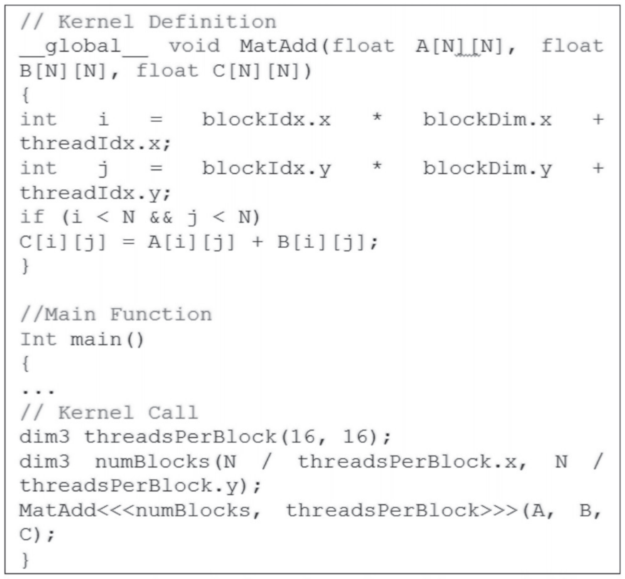


```
// Kernel definition
__global__ void MatAdd(float A[N][N], float
B[N][N], float C[N][N])
{
int i = threadIdx.x;
int j = threadIdx.y;
C[i][j] = A[i][j] + B[i][j];
}

//Main function
int main()
{
...
// Kernel call with 4 blocks of N * N * 1
threads each one
int numBlocks = 4;
dim3 threadsPerBlock(N, N);
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B,
C);
}
```

Figure 3. Code for a kernel call with two-dimensional thread blocks.

The code in Figure 3 shows the kernel call structure with two-dimensional thread blocks.This code shows that a thread matrix can be called as input for running a kernel. The blocks can also be arranged in one or two dimensions. In the code, a row of 4 blocks is called, that is defined in the variable named *numblocks*. This structure is useful when working with large data arrays, since the size of each block is limited: In a typical GPU only 512 threads per block can be defined. Therefore, if the data set is larger than 512 positions, several blocks must be used.



```
// Kernel Definition
__global__ void MatAdd(float A[N][N], float
B[N][N], float C[N][N])
{
int i = blockIdx.x * blockDim.x +
threadIdx.x;
int j = blockIdx.y * blockDim.y +
threadIdx.y;
if (i < N && j < N)
C[i][j] = A[i][j] + B[i][j];
}

//Main Function
Int main()
{
...
// Kernel Call
dim3 threadsPerBlock(16, 16);
dim3 numBlocks(N / threadsPerBlock.x, N /
threadsPerBlock.y);
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B,
C);
}
```

Figure 4. Kernel call with two-dimensional thread blocks.

Code in Figure 4 uses a grid of 16x16 blocks, in which each block has *N*/16 threads along each dimension. Following these definitions all entries of the matrix are accounted in the event of a huge matrix. Also is shown in Figure 4, the use of some new built-in variables: *blockIdx.x*, *blockDim.x* and *threadIdx.x*. These variables allow the access to each thread. They are used to define the position of the data structure that each thread runs. The variable *blockIdx.x* defines the block, *blockDim.x* defines the column or row where each block is, and *threadIdx.x* is the identifier of each thread within the block. A diagram illustrating the previous concepts is shown in Figure 5.

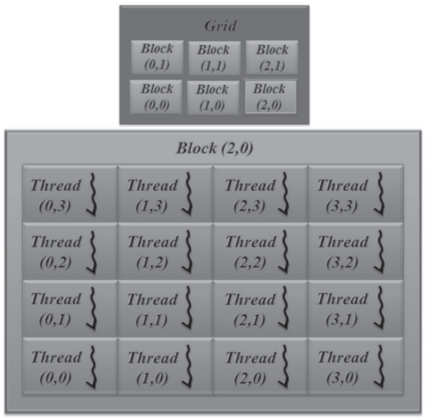


Figure 5. Threads, blocks and grids in a GPU.

3.1. Memory hierarchy and data transfer.

CUDA™ threads have access to different types of memory during execution, as shown in Figure 6. Each thread has a private local memory and registers, and each thread block has a shared memory visible and accessible to all threads within the same block. The lifetime of the shared memory is the same as the block. There is another memory, named global memory, which all threads can access. Two additional spaces for accessible read-only memory are available for the threads: the constant memory and the texture memory. These two memory spaces and the shared memory space have optimized benefits in certain applications. Generally, the use of shared memory is much more efficient than all others due to its fast access, since with this on-chip memory, there is no need for sending data over the system memory bus [4].

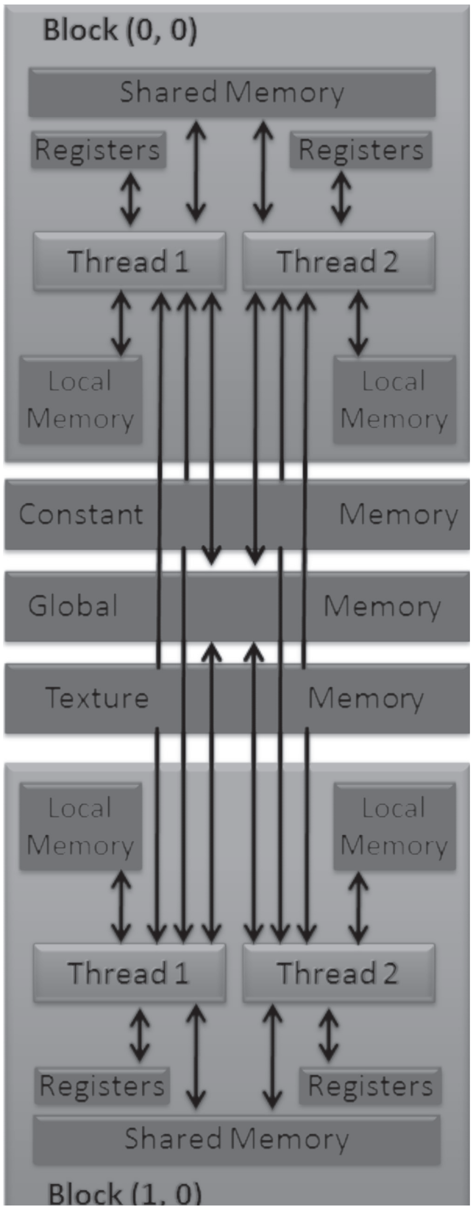


Figure 6. Memory Hierarchy.

CUDA™ allows heterogeneous programming as shown in Figure 7. The model assumes that kernels run on a physically separate *device* which operates as a co-processor for the *host* which is running the program sequentially. The CUDA™ programming model also assumes that both the *host* and the *device* maintain their separate memory spaces in the DRAM; these memory spaces are named host memory and device memory, respectively [4].

Before calling a kernel and running it concurrently in a GPU, the data to be processed have to be defined. To define the data, variables of any type can be used: int, float, float2, char, or others types default of CUDA™ as uchar1 or Complex. The variables must have one characteristic: they must be allocated in the memory of the GPU for being usable by CUDA™. Such allocation process is done by means of functions that allow the data transfer between the CPU and the GPU memory spaces. The functions that CUDA™ architecture has built-in for this purpose are: *cudaMalloc(),* a function that places a variable in the memory space of the GPU, *cudaFree(),* a function that frees the memory of the GPU of a variable and *cudaMemcopy()* a function that passes data from a variable in the CPU memory to a variable in the GPU memory or vice versa. In the code of Figure 8 is shown the use of each one of these functions.

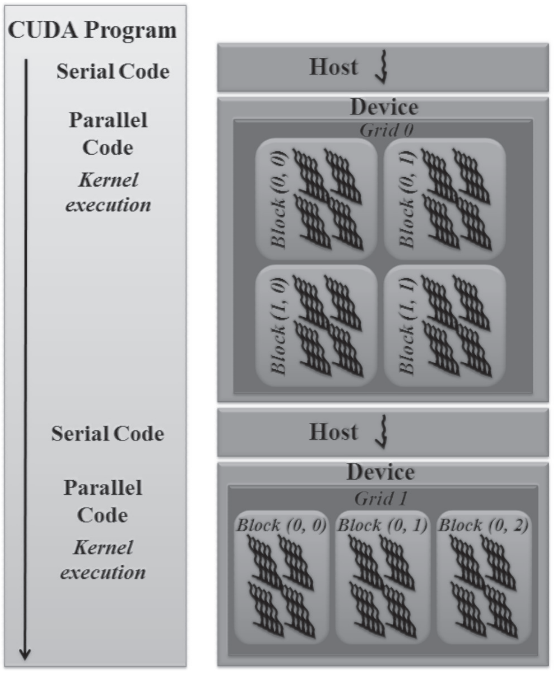


Figure 7. CUDA™ Program Execution.

3.2. CUDA Libraries and Graphic Interoperability.

CUDA™ allows for more than generic arithmetic operations on data arrays. With CUDA™ is possible to carry out specific operations widely used by the scientific and engineering community by means of specialized libraries. Just to present two examples, CUBLAS and CUFFT, are the two most celebrated NVIDIA® GPU-accelerated math libraries. CUBLAS is an implementation of BLAS, (Basic Linear Algebra Subprograms). The basic model by which an application uses the CUBLAS library, is by creating matrix and vector objects in the GPU memory space. Once the objects are created they are filled with data to call a sequence of CUBLAS functions. The final step is to upload the results from GPU memory space back to the host CPU memory. To accomplish this set of task, CUBLAS provides helper functions for creating and destroying objects in the GPU space. Additional functions are available for writing data to- and retrieving data from-the created objects.

```
// Device Code
__global__ void VecAdd(float* A, float* B,
float* C, int N) {
int i = blockDim.x * blockIdx.x +
threadIdx.x;
if (i < N) C[i] = A[i] + B[i];
}

// Host Code
int main() {
int N = ...;
size_t size = N * sizeof(float);
// Allocate variables in host memory space
float* h_A = (float*)malloc(size);
float* h_B = (float*)malloc(size);
// Allocate variables in device memory space
float* d_A;
cudaMalloc(&d_A, size);
float* d_B;
cudaMalloc(&d_B, size);
float* d_C;
cudaMalloc(&d_C, size);
// Transfer data from host to device
cudaMemcpy(d_A, h_A, size,
cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size,
cudaMemcpyHostToDevice);
// Kernel call
int threadsPerBlock = 256;
int blocksPerGrid =
(N + threadsPerBlock - 1) / threadsPerBlock;
VecAdd<<<blocksPerGrid,
threadsPerBlock>>>(d_A, d_B, d_C, N);
// Copy the result from the GPU to the host
cudaMemcpy(h_C,d_C,size,
cudaMemcpyDeviceToHost);
// Free device memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
}
```

Figure 8. Code for data transfer in CUDA™.

CUFFT is the NVIDIA® CUDA™ library for the fast Fourier transform. The FFT is a very efficient algorithm for calculating the discrete Fourier transform of real or complex data sets. The CUFFT lets, through few instructions, to compute in parallel the FFT of a data set in the GPU. CUFFT takes advantage of all the computational power of the GPU without having to write the full implementation from scratch [8]. CUFFT is modeled after the most efficient implementation of the fast Fourier algorithm in the market, the FFTW [9]. The parameters used in its implementation are similar to those used in FFTW. In Figure 9 is illustrated, by an example, how to perform a 2D Fourier transform from complex to complex in CUDA™ in device code. Initially the dimensions of the transform are defined. After, the plan (FFT configuration), the input variable, and output variable are also defined. For the illustrated case, the variables are of the type complex, that is, a special CUFFT float2 variable; right after comes the dynamic allocation of these variables with the *cudaMalloc()* function. The function *cufftplan2d* generates the FFT plan. The next step is to run the plan with *cufftExecC2C()*, in which is determined if the transform is direct or inverse. In *odata* is allocated the output data. Finally, the instructions *cufftdestroy()* and *cudaFree()* free the variables used by CUDA™.

```
#define NX 256
#define NY 128

cufftHandle plan;
cufftComplex *idata, *odata;
// Device memory allocation
cudaMalloc((void**)&idata,
sizeof(cufftComplex)*NX*NY);
cudaMalloc((void**)&odata,
sizeof(cufftComplex)*NX*NY);
// Plan declaration
cufftPlan2d(&plan, NX, NY, CUFFT_C2C);
// FFT execution
cufftExecC2C(plan,       idata,       odata,
CUFFT_FORWARD);
// Inverse FFT execution
cufftExecC2C(plan,       odata,       odata,
CUFFT_INVERSE);
cufftDestroy(plan);
cudaFree(idata); cudaFree(odata);
```

Figure 9. CUDA™ Program for Two-dimensional computation of Fourier transforms on complex data.

Other useful feature of CUDA™ is the graphics interoperability with DirectX and OpenGL [4]. OpenGL (Open Graphics Library) is a 2D and 3D graphics library made for C [10]. It consists of a series of functions that allow for handling different features of images, as textures, shades, among others. Together with a set of auxiliary libraries, OpenGL provides the necessary tools to manage windows and events happening over the windows in a very simplified way. The system itself is an event handler, this means, the library cyclically checks every event, and if one is activated, it executes the function designated for that particular event.

OpenGL provides a graphic user interface for CUDA™ through a full interoperability between them. Some resources of OPENGL can be mapped to the address space of CUDA™; either to enable CUDA™ to read data processed by OpenGL or to enable CUDA™ to transfer already processed data to be consumed by OPENGL [4]. This capability eliminates the need of transferring data from device memory to host memory, every time is needed the visualization of the results of an operation performed by a kernel, for example, over an image. The fewer the data transfers the faster the algorithms are, especially for large amount of processed data.

4. APPLICATION OF CUDA™ TO ACCELERATED COMPUTATION OF THE DIFFRACTION INTEGRALS FOR DIGITAL IN-LINE HOLOGRAPHIC MICROSCOPY.

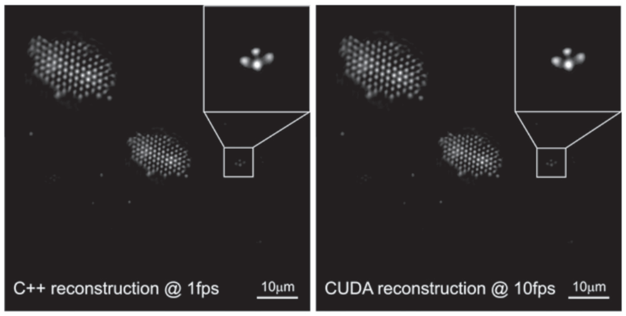


Figure 10. DIHM hologram reconstruction by scalable convolution reconstruction. The input and output images are 2048x2048pixles.The image at left is obtained by C++ @ 1 frame per second (fps) and at right by CUDA™ @ 10 fps.

In this section the power of computation provided by CUDA™ is applied to evaluate numerically diffraction integrals. In particularly, CUDA™ is employed for numerical reconstruction of digitally recorded holograms in digital in-line holographic microscopy (DIHM). Once the hologram is recorded by a digital camera, it is transferred to a PC for its further processing. The hologram reconstruction can be described as the diffraction that a converging spherical wavefront undergoes as it illuminates the recorded hologram. In the Fraunhofer domain, the described above diffraction process is represented precisely by [11, 12]:

$$K(\hat{\mathbf{\imath}}) = \int_{\sigma} \tilde{I}(\mathbf{r}) \exp\left[ik\left(\hat{\mathbf{\imath}} \cdot \mathbf{r}/|\mathbf{r}|\right)\right] d^2r \, . \qquad (1)$$

In equation (1) the integration extends over the surface of the screen σ, a charge-coupled device (CCD) or complementary metal-oxide semiconductor (CMOS) camera, with coordinates $\mathbf{r} = (x, y, L)$, L the distance from the illumination source to the center of the screen; $k = 2\pi/\lambda$ is the propagation number; $\tilde{I}(\mathbf{r})$ is the contrast in-line hologram obtained by subtracting the images with and without the sample present; and $\hat{\mathbf{\imath}} = (\xi_x, \xi_y, \xi_z)$ are the coordinates at the reconstruction plane. $K(\hat{\mathbf{\imath}})$ is a complex quantity that can be calculated on a number of planes at various distances z_r from the illumination source (pinhole) in order to recover the three-dimensional information of the sample, from a single two-dimensional DIHM hologram. The reconstruction operation is entirely done numerically over a single two-dimensional DIHM hologram with MxN pixels. The computational implementation of equation (1) has a complexity of $\mathcal{O}$ (MxN) which is extremely time consuming.. With the aim of reducing the computation time of equation (1) without compromising the very demanding requirements of DIHM, Kreuzer [12] patented a procedure. This method casts equation (1) into a scalable convolution that allows for using FFT and in this way for accelerating the numerical reconstruction of DIHM holograms. This scalable convolution reduces the complexity of the computation to $\mathcal{O}$ (Mxlog$N$), still a large number if, for instance, M=N=2048. Fortunately, once the reconstruction process is represented in terms of FFTs, CUFFT can be used for the parallelized reconstruction of the holograms, with the results that are shown below.

For comparison, the scalable convolution algorithm has been implemented in C++ and CUDA™ [13]. With both algorithms there has been reconstructed a 2048x2048 pixels hologram into a 2048x2048 pixels image. The results are shown in Figure 10. From this Figure is clear that there is no difference between the recovered information from the hologram, meaning that the scalable convolution algorithm is fully parallelizable. For clearer comparison the area in the white square has been enlarged and its contrast optimized. A cluster of four beads is very well resolved with both algorithms.

The reconstruction algorithms have been implemented in a personal computer powered with an Intel® Core™ 2 Quad running at 2.33GHz and 4GB of RAM memory. The computer hosts a Geforce 9800GT graphics card with 112 stream cores, 512MB of local memory and running at 600MHz. Table 1 summarizes the results obtained as holograms of different sizes are reconstructed. It is apparent the important reduction of the reconstruction time, which for the largest hologram is of the order of 11 times.

Table 1. Scalable convolution reconstruction time for different sizes of holograms in CUDA™ and C++. The output images have the same number of pixels that the input holograms.

Size of the hologram	**Time CUDA™ (ms)**	**Time C++ (ms)**
512x512	9.06	63
1024x1024	26.40	312
2048x2048	97.82	1061

5.CONCLUSIONS

Parallel computing is a technique in which many tasks run simultaneously. Traditionally it has been developed in multi-processor systems (multiples CPUs) called clusters, with the disadvantage that these cluster are bulky and quite expensive.. The GPUs offer an alternative approach to parallel computing, which allow for using the developed power for video games in different environments. Many are the advantages of parallel computing on GPUs. Mainly the cost and size are two reasons why GPUs can be chosen for implementing parallel computing as a very viable option to accelerate floating point numerical computing.

CUDA™ is a parallel computing architecture for embedded devices created by NVIDIA®. It allows, thanks to its rich software development kit (SDK), the implementation of algorithms for concurrent execution and thus reducing the processing time. Many of these algorithms are designed to achieve real-time execution. This new opportunity provided by NVIDIA®, shows another side of the GPUs, that traditionally have been used exclusively for graphics processing on the video-game world.

Almost any algorithm that is implemented in a CPU can be implemented in a GPU. It should be noted that in both cases there will not be the same computational efficiency; only those tasks that have a high degree of parallelism, and especially a high floating point arithmetic demand, are the greatest benefited when implemented on a GPU. This is the case of holographic reconstruction via the computation of diffraction integrals.

Digital in-line holographic microscopy is perhaps the simplest methodology to obtain three-dimensional information from the micrometer world. Particularly this technique requires very large and complex calculations. One limitation at the moment of reconstructing large-size holograms is the elapsed time for calculation. This has prevented, until now, the possibility of having portable systems for real-time holographic reconstruction. In this paper it has been shown that using CUDA™ is possible to extend the use of DIHM for real time application. CUDA™ can be employed for accelerating diverse tasks as: object recognition, many bodies problems computation, Monte Carlo simulation among many others.

REFERENCES

[1] HARRIOTT, L. Limits of Lithography. Proceedings of the IEEE, Vol. 89, No. 3, March 2001.

[2] OWENS, J., HOUSTON, M., LUEBKE, D. AND GREEN, S. GPU Computing: Graphics Processing Units - powerful, programmable, and highly parallel - are increasingly targeting general-purpose computing applications. Proceedings of the IEEE, Vol. 96, No. 5, May 2008.

[3] AHRENBERG, L., PAGE A., HENNELLY, B., MCDONALD, J., AND NAUGHTON, T. Using Commodity Graphics Hardware for Real-Time Digital Hologram View-Reconstruction, J. Disp. Tech., 5, 4, 2009.

[4] NVIDIA, CUDA™, "NVIDIA CUDA C Programming Guide," 3.1.1, 2010.

[5] HARRIS, C., HAINES, K. AND STAVELEY -SMITH L. GPU Accelerated Radio Astronomy Signal Convolution, Exper. Astro., 22, 129–141, 2008.

[6] GAIKWAD, A. AND TOKE I. GPU based Sparse Grid Technique for Solving Multidimensional Options Pricing PDEs. Proceeding WHPCF '09, 2009.

[7] STONE, J., PHILLIPS, J., FREDDOLINO, P., HARDY, D., TRABUCO, L. AND SCHULTEN, K. Accelerating molecular modeling applications with graphics processors. J. Comp. Chem., 28, 2618-2640, 2007.

[8] NVIDIA, CUDA™, " CUDA CUFFT Library," 3.1, 2010.

[9] FFTW home page. Available: http://www.fftw.org/ [cited March 8 2011].

[10] OPENGL HOME PAGE. Available: http://www.opengl.org/ [cited March 8 2011].

[11] GARCIA-SUCERQUIA, J., XU, W., JERICHO, S., KLAGES, P., JERICHO, M. AND KREUZER, H. Digital In-line Holography Microscopy, Appl. Opt. 45, 836-850, 2006.

[12] US Patent 6.411.406 B1, KREUZER H.J. Holographic Microscope and Method of Hologram Reconstruction, 2002.

[13] KREUZER H.J. AND KLAGES P. A software package for the reconstruction of digital in-line and other holograms DIHM-software, Helix Science Applications, Halifax, N.S., Canada, 2006.