High Performance Computing: CUDA as a Supporting Technology for Next Generation Augmented Reality Applications

Thiago S. M. C. Farias 1

João Marcelo N. X. Teixeira 1

Pedro J. S. Leite 1

Gabriel F. Almeida 1

Mozart W. S. Almeida 1

Veronica Teichrieb 1

Judith Kelner 1

Abstract: The main purpose of this survey is presenting the potential of GPGPU technology for real time markerless augmented reality related processing. CUDA is a GPGPU technology developed by NVIDIA that allows programmers to use the C programming language to code algorithms for execution on the GPU. Applications that require mathematically intensive computation of large amounts of data are ideal targets for GPU computing. In this survey, CUDA architecture will be depicted, together with an optimized programming model for obtaining better results using the parallel approach. A case study, mainly related to tracking algorithms, will also be shown in order to demonstrate the performance improvement in comparison to sequential approaches.

1 Centro de Informática, UFPE, Caixa Postal 50670-901
{tsmcf, jmxnt, pjsl, gfa, mwsa, vt, jk@cin.ufpe.br}

1 Introduction

Single processor technology has been evolving across last decades, and for years programmers have created their applications based on the premise that an enhancement in performance must be supported by a higher clock frequency of the CPU. Indeed, because of physical and architectural bounds, there are limitations on the computational power that can be achieved with a single processor system, so chipmakers currently pursue alternatives to sustain computational power growth, including creation of multi-core systems embedded on a chip. One of these implementations can be found in GPU (Graphics Processing Unit) industry. Since nowadays GPUs are seen as high performance units, this capability made parallel processing paradigm [1] stronger. On the other hand, such paradigm demands another type of development that not every application can take advantage of.

Parallel computing targets problems that are scalable and possibly distributed, dividing the original problem into smaller pieces that will be solved in different places. Several parallel approaches may be used to speed up the solution of a given problem, such as: effective use of multiple cores of a processor; use of a distributed solution (using a middleware, or even a network grid).

In case of taking advantage of processor multiple cores, or using a machine with more than one processor, the solution will scale up to the number of cores in the CPU, or the (number of CPUs) x (cores in each CPU). In spite of this solution being not rare, it is not highly scalable, since the newest powerful mainstream processor is a Quad Core [2], which provides a maximum speed up of 4x, if no synchronization pass is needed in the process.

Scalability is a key point of the network grid solution. The achieved result is optimal if the problem can be split into independent parts. Although, in case there is any communication between the cells of a grid, the network latency will slow down the entire process.

Another alternative for building a parallel version of an algorithm is to use a massive parallel computing device as a coprocessor. The GPU can be considered a representative example. Its use for solving problems not related to image generation and rendering, named GPGPU (General-Purpose computation on GPUs), is commonly seen for algorithms in the image processing and computer vision areas, as well as other research domains that can take benefit of massively parallel processing. GPUs usually process millions of pixels per second through its several pixel shading pipelines (a common High Definition (HD) video in 1080p resolution has about 2,073,600 pixels at a frame rate of 30 Hz, which represents 62,208,000 pixels per second).

Although this amazing processing power may be used for general purpose computations, a GPU is not optimized for this function. There are problems intercommunicating the processing parts (implemented in fragment programs). Data can be shared and optimized for reading (through data transformed into textures), but the result of

each processing unit computation is only visible outside the scope of the GPU, since it is written in a texture (using a render to texture technique), as well. The time spent with memory transfers between host and GPU is also an issue and is certainly one of the problems to be solved by next generation GPUs.

Regardless, the massive parallel computing approach may be adequately explored to satisfy real time constraints required by Augmented Reality (AR) applications [3]. Its use may vary from generation of visually realistic virtual scenes to physics simulation. It could also aggregate data from 3D reconstructions and image processing to perform optimized tracking, providing ambient occlusion with virtual objects. In this survey, the massively parallel computing paradigm, delivered by NVIDIA CUDA (Compute Unified Device Architecture) [4] will be presented, and its potentiality will be exemplified through the possibility of working with images from HD videos for AR. Before the massive processing technology, the idea of real time processing of images with resolutions above 800x600 pixels was nearly unachievable, due to the use of sequential or even ordinary parallel approaches. The aim of this survey is not only to present a new technology, but also to show the great improvements that can be obtained by applying it in computer vision and AR applications. An important aspect to be considered is that using GPGPU will free CPU resources for computing other steps of the AR pipeline, as well as user defined specific application tasks.

Problems that arise from AR applications using markerless registration and 3D reconstruction integration can be tackled with massive parallel approaches, but there are still bottlenecks in the Markerless Augmented Reality (MAR) field. The solution to that seems to rely on the use of actual generic massively parallel programming, without worrying about rendering workarounds natively inherited from shaders programming. Examples will show that even with the mandatory knowledge about the GPU hardware architecture, CUDA programming model is very attractive and easy to tune when following some guidelines. This survey introduces guidelines defined by the authors for performing CUDA optimizations that are mandatory to be followed in order to achieve significant speedups.

2 NVIDIA CUDA

NVIDIA's G80 processor is the first CUDA-compliant hardware available, and will be the basis for this survey. It can be accessed through a low-level parallel thread execution virtual machine, and a virtual instruction set architecture called PTX (Parallel Thread Execution) [4]. When the application is transferred to the target hardware, PTX code is translated to the device instruction set. Contrary to ATI's interface to their hardware, named Close To Metal (CTM) [5], that has a very well defined low-level interface, NVIDIA focused on a C-based language, although it still requires a great knowledge of how the GPU works, as well as about its hardware.

The GeForce 8800 GTX, one of the first G80-based cards, has 16 groups of 8 Scalar Processors (SPs) each, totalizing 128 processors. This architecture enables the GPU to use

these groups, called multiprocessors, to process blocks of 64 to 512 threads. These blocks are divided into groups of 32, called warps, the lowest scheduling unit used by the multiprocessor, since they do not work on a thread-level context switch. Kernel is a program, executed as blocks of warps, but the arrangement of threads in blocks and blocks in grids is defined by the programmer, as seen in Figure 1. More information related to threads, blocks, grids and kernels can be found in Section 4.

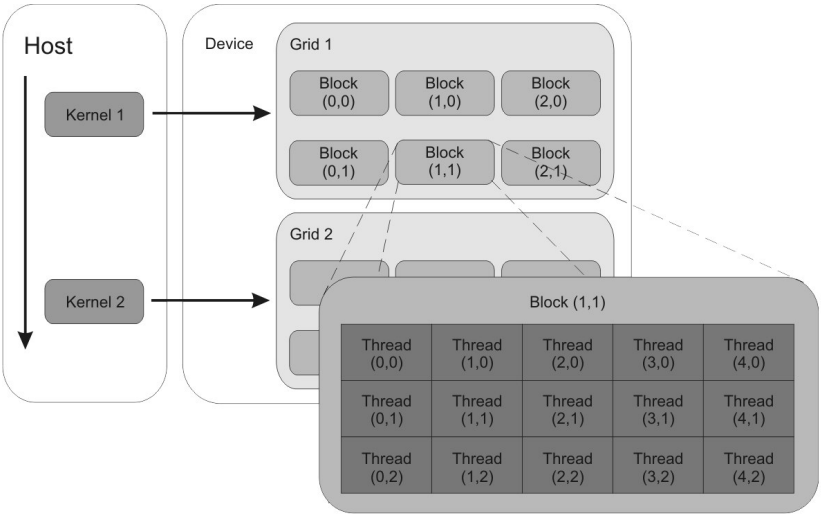

Figure 1. G80 execution model

Each multiprocessor contains 8 SPs and 2 Special Function Units (SFUs). The SFUs are used to calculate more complex instructions like sine, cosine and logarithm. Special instructions are several times slower because of the precision required, but it is possible to execute most instructions faster, in a less precise mode. An integer multiplication, for example, is processed by a SFU and requires 8 cycles, while the lower precision mode of this instruction (24 instead of 32 bits) can be executed by the SPs in only 2 cycles. Concluding, speed, rather than precision, is a major concern when rendering 3D scenes, the main purpose of GPUs.

The memory model implemented on the G80 is shown in Figure 2. It defines the scope for memory access operations: threads can individually read/write registers and local memory; threads in the same block can read/write in the same segment of shared memory; and all the threads in the grid can communicate through reads/writes in global memory. Per-grid read-only operations are enabled for constant memory and texture memory. As for memory organization, some of it is located on chip, like registers and shared memory, while texture, constant, local and global memory are implemented in device memory. Only reads from texture and constant memory can be cached. This organization confirms that the GPU is

designed for highly parallel computation of high arithmetic intensive problems; it is not recommended to use GPUs for any tasks that are not massively parallel. The G80 processor can be compared to a performance-optimized calculator, so the scope of applications that can benefit from GPU execution is not as big as if there was used a massive multi-core CPU.

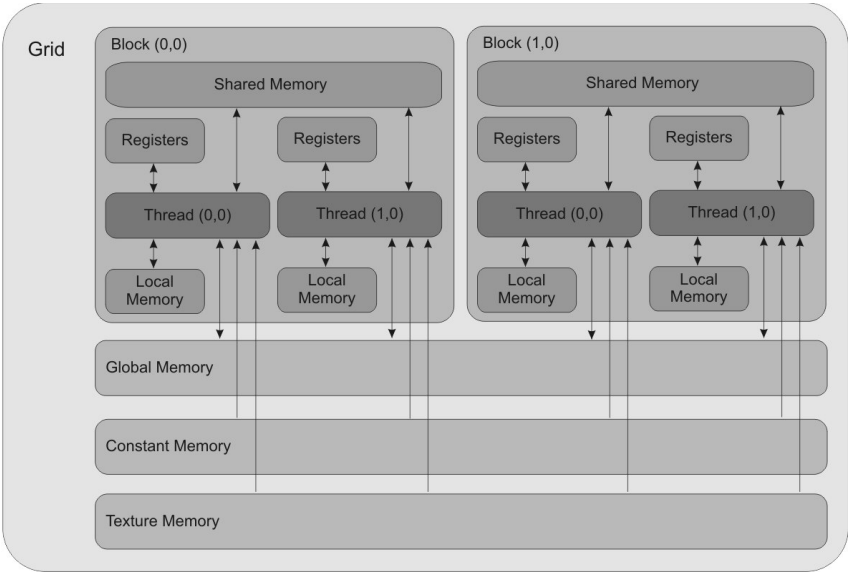

Figure 2. GPU memory model

To take advantage of the G80 capabilities, there are some guidelines that should be considered when developing applications, including hardware limitations that the programmer must be aware of. The maximum number of threads per multiprocessor is 768, or 24 warps; these threads must be organized in a maximum number of 8 blocks per multiprocessor, and 512 threads per block. Each multiprocessor contains 8,192 32-bit registers, 16 KB of shared memory, 8 KB of cached constants and 8 KB of cached 1D textures.

Memory latency is a significant matter, since the cost of memory access depends on its location. Contrary to what can be seen on hierarchical memory architectures, local memory is not faster than shared memory; indeed, it is several times slower, and cannot be cached. That is because local memory is a partition of the device memory, so it is important to use faster, on-chip, shared memory and registers.

Another issue when developing for CUDA is to appropriately feed the G80 with enough threads for execution, as the processor can schedule millions of threads. In order to help developers design efficient applications (in terms of memory and processor usage), NVIDIA released a CUDA Occupancy Calculator; using just a few parameters, as threads

per block, registers per thread and shared memory per block, the programmer can know how much he/she can improve CUDA applications. The CUDA profiler can give the same occupancy information, and also kernel execution times in both GPU and CPU. The time spent with memory transfers is also monitored. More specific programming guidelines will be discussed later.

3 System Configuration

CUDA Software Development Kit (SDK) is freely available for anyone to download it. This section explains how to download, install and consult the documentation, and gives some information about its license.

3.1 CUDA Software Download and License

CUDA is composed of three software pieces: CUDA SDK, CUDA Toolkit and CUDA graphics driver, which enables the use of compatible hardware. The download can be done through NVIDIA's CUDA Site (http://www.nvidia.com/object/cuda_home.html), clicking one of the "get CUDA" links at the top right corner of the page. Users can choose between Windows, Linux and Mac OS versions.

The CUDA Developer SDK provides examples with source code, utilities, and white papers to help writing software with CUDA. The NVIDIA CUDA Toolkit contains the compiler, profiler, and additional libraries, such as CUBLAS (CUDA port of Basic Linear Algebra Subprograms) and CUFFT (CUDA implementation of Fast Fourier Transform). It's Instruction Set Architecture is called PTX (Parallel Thread Execution). As noticed, CUDA Toolkit is required to run and compile CUDA based code.

CUDA is released free of charge for use in derivative works, whether academic, commercial, or personal. Basically, it is prohibited to disassemble, decompile or reverse engineer the object (compiled) code provided. It is also determined that all NVIDIA copyright notices and trademarks should be used properly and acknowledged on derivative works, using the following statement: "This software contains source code provided by NVIDIA Corporation".

3.2 Installation

System requirements for CUDA based development are a computer running Microsoft Windows, Mac OS X (10.5.2) or Linux (Fedora, Redhat, SUSE, OpenSUSE, Ubuntu distributions are compatible). Minimum hardware specifications are not defined by NVIDIA, but if the use of specific hardware is planned, 1 GB of system memory and at least 1 GB of free hard disk space would fit. To use a CUDA compatible video card, it is

necessary a vacant PCI-Express slot and an additional specific power connector, depending on the model .

First, the NVIDIA display driver has to be installed allowing hardware execution of CUDA based applications. After that, the Toolkit and the SDK can be installed, in this order. Then, the sample projects included can be run, which are compatible with Microsoft Visual Studio 2003 (7.0) and 2005 (8.0).

3.3 Documentation and Example Applications

There are lots of documents available to help the developer of CUDA based applications. After installation of the CUDA Toolkit, the documentation will be available inside the Toolkit's installation directory, in the folder called "doc". It contains CUBLAS, CUFFT, PTX ISA and CUDA Compiler (nvcc) documentations. Sample and training programs will be available inside the CUDA SDK's installation directory.

4 CUDA Programming Approach

CUDA is a new programming API that allows general purpose GPU access to the programmer. Part of the CUDA programming interface relies only on the CUDA files, which have the ".cu" extension. CU files enable some extensions to the C language that are necessary to deal with the GPU program (known as kernel) configuration, new intrinsic types and GPU program invocation.

Besides that, host code written in CU files can be also called from C or CPP files, since functions have the standard C calling convention and can be linked from elsewhere by adding the adequate function prototype before using the chosen function. Sometimes, to prevent linker errors, the *extern "C"* must be added before the imported function prototype.

This way, CU, C and CPP files can be mixed and linked together to better encapsulate the code, supporting any architecture that allows combining these files. Only host code is addressable by the user in extern C or CPP files. The device code (kernel) is only addressable inside the CU file.

The programming model reflects the hardware architecture described in Section 2. Some concepts that need to be introduced are threads, blocks, grids and kernels. Threads are the smallest units that will execute in parallel. Threads are grouped in blocks that may be logically divided in one, two or three dimensions. Threads belonging to the same block can synchronize among them and share a small block of fast memory named shared memory. The use of the shared memory can speed up the entire algorithm, since this memory is much faster than global memory. A grid is a group of blocks that can be also distributed in one, two or three dimensions. The code that will run inside the threads is called kernel. Each call to a kernel must specify a kernel configuration, which contains the number of blocks in each

grid dimension (up to three), the number of threads in each block dimension (up to three) and optionally the amount of shared memory to be allocated.

Some keywords were added by the CUDA compiler to aid code generation and make the programming interface easier. A set of these keywords represent the scope of a function or variable: `__host__`, `__device__` and `__global__`. The `__global__` keyword is only used in kernel declarations, because a kernel is a function that will run inside the device but is called from the host side. The `__host__` and `__device__` modifiers can be applied to both variables and functions, specifying where they will be located.

Another modification to the language is the kernel invocation. The kernel name is followed by its configuration between "<<<" and ">>>", as in:

```
kernel_name<<<gridDim, blockDim>>>(params);
```

The __shared__ keyword was introduced to indicate that a variable will be allocated inside shared memory space, and will be accessed only by threads belonging to the same block.

New types were also introduced with CUDA extension to deal with GPU internal types and make memory transfer faster. The dimensions used in kernel configuration are declared as *dim3*. The *dim3* type is a built-in vector type based on *uint3*, which has 3 unsigned integers, one for each dimension (x, y, z). The default value for all *dim3* coordinates is "1".

All conventional types (except *double*) have their built-in vector types, like *int2*, *float3*, etc. The maximum vector size is 4. The number in the type name indicates the number of elements inside the vector. These elements are accessed as coordinates (x, y, z, w). Examples of construction and use can be seen in the following code:

```
float3 temp;
temp.x = 0.1f;
temp.y = 2.0f;
temp.z = 4.9f;
float4 f = make_float4(1, 2, 3, 4);
```

Despite the fact that CUDA is an extension to the C language, the C++ template feature is also enabled inside CU modules. Data types and functions may receive types in compilation-time to be written generically. Some CUDA API types and functions use templates to behave polymorphically, such as textures and functions.

Textures are an important feature in CUDA. Reading data from textures enables the texture cache, which speeds up memory access inside the kernel. Any region of linear memory can be used as an one-dimensional texture. To use bi-dimensional textures, texture filtering, normalized coordinates or alternative addressing modes, CUDA Arrays must be used instead of the raw linear memory pointer. Using CUDA Arrays is very helpful, but it can be ignored in order to get direct access to the texture.

Before using textures, a texture reference must be declared. The texture reference is a template type that has parameters such as *type* (the type of the elements) and *dimension* (when using linear memory, the mandatory value is "1"). An example of texture reference declaration is given as follows:

```
texture<float, 1> texture1;

texture<int, 1> texture2;
```

After declaring the texture references, user must bind the linear memory to a reference. The CUDA API function responsible for this operation is *cudaBindTexture*. In sequence, the texture reference can be accessed inside the kernel by using the *tex1Dfetch* function.

Memory used for textures storage must be in the device memory. This memory must be allocated and filled with external content. CUDA provides a set of functions to handle memory allocation and copy. The function used for allocation is *cudaMalloc*, and it takes as parameters the output pointer to put the base address and the size of the contiguous memory region. To copy host data to the device memory or vice versa, there is the *cudaMemcpy* function, which is a simple memory copy operation that needs additionally the copy direction information (*cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*).

5 CUDA Programming Guidelines

This section presents some programming guidelines, based on CUDA documentation and experiences on CUDA optimization realized by the authors. Some topics discussed in this section include: finding an optimal number of threads for a specific application, thread arrangement, non-sequential memory access, shared memory usage, sequential writing, loop unrolling, pinned memory, type bandwidth and the necessity to avoid non-optimal type conversions. Such recommendations are made based on some simulations that will be described as follows and they can help developers to improve the performance of CUDA programs executed on GeForce 8 Series.

As an introduction to the results acquired, some remarks must be made regarding kernel launch configurations. The following data types are considered in simulations: *uchar4*, *int*, *int2*, *int4*, *float*, *float2* and *float4*, whose sizes are 32-bit (*uchar4*, *int* and *float*), 64-bit (*int2* and *float2*) and 128-bit (*int4* and *float4*). Such choice is made because the 8800

GTX provides a single load instruction for words with each of those sizes, improving the overall kernel performance.

Since developers will deal with streaming processors, higher bandwidths are achieved the greater is the amount of data to be processed. As a consequence, each kernel is configured to process 4,456,448 $\left(2^{22}+2^{18}\right)$ elements of those types mentioned earlier.

Finally, all kernel launch configurations consist of one-dimensional blocks with 128 threads. Furthermore, kernels meant to be executed with 1D configuration are distributed over a grid with 34,816 rows of blocks, while their 2D counterparts are distributed over a grid of 16 rows per 2,176 columns of blocks.

5.1 Thread Arrangement

Kernel calls must be done alongside with a launch configuration, which specifies how many thread blocks are supposed to be logically executed in parallel. A grid consists of thread blocks and will be distributed among the 8800 GTX stream processors. Threads are physically executed in parallel by the stream processors, as seen in Section 2. Grids and blocks may be distributed up to three dimensions and, because of this, grid and block configurations must be carefully chosen so that the execution reaches maximum performance.

The nature and amount of data to be processed will influence the launch configuration. 2D textures can be easily distributed over a 2D configuration of blocks, simplifying memory address calculation. However, our simulations showed that kernels with 2D or 3D configurations can result in a poor performance, regarding both bandwidth and processing time.

The code excerpt in Figure 3 describes two kernels that read a single element from an 1D texture (*read_only_tex_1D*) or a 2D one (*read_only_tex_2D*) into the shared memory. All code samples used for benchmarks were developed using macros. In some cases, it is necessary to replace *##type* and *type* with the type used, for example, for *uchar4* the kernel name will be *read_only_tex_1D_uchar4* or *read_only_tex_2D_uchar4*.

```
__global__ void read_only_tex_1D_##type() {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    __shared__ type shared[BLOCK_SIZE];
    shared[threadIdx.x] = tex1Dfetch(tex_##type, idx);
}

__global__ void read_only_tex_2D_##type() {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    const unsigned int idy = threadIdx.y + __mul24(blockIdx.y, blockDim.y);
    const unsigned int index = threadIdx.x + __mul24(threadIdx.y,
        BLOCK_SIZE);

    __shared__ type shared[BLOCK_SIZE];
    shared[index] = tex2D(tex_2D_##type, idx, idy);
}
```

Figure 3. 1D and 2D texture memory read-only kernels

The code shown in Figure 3 was executed and results of bandwidth and processing time can be seen in both charts in Figure 4.

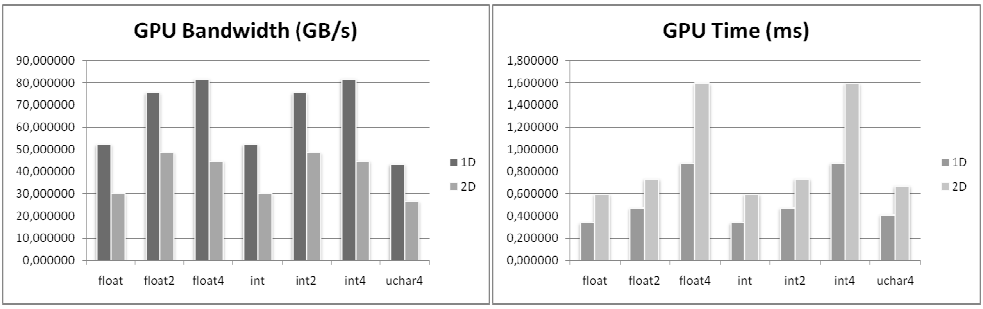

Figure 4. GPU bandwidth and processing time for different thread arrangements

These charts show that texture reads are slightly better with 128-bit data types, such as *int4* or *float4*. In addition, the best launch configurations were the unidimensional ones, since *tex1Dfetch* function is faster than *tex2D* one [4].

The code sample in Figure 5 is similar to the one in Figure 3, but its kernels store the value read into the global memory.

```
__global__ void copy_tex_1D_##type(type* g_odata) {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    g_odata[idx] = tex1Dfetch(tex_##type, idx);
}

__global__ void copy_tex_2D_##type(type* g_odata) {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    const unsigned int idy = threadIdx.y + __mul24(blockIdx.y, blockDim.y);
    g_odata[idx + __mul24(idy, __mul24(blockDim.x, gridDim.x))] =
        tex2D(tex_2D_##type, idx, idy);
}
```

Figure 5. 1D and 2D texture memory copy to global memory kernels

Complementing the results for the first benchmark realized, the current code sample was executed under the same conditions and the results obtained are presented in Figure 6.

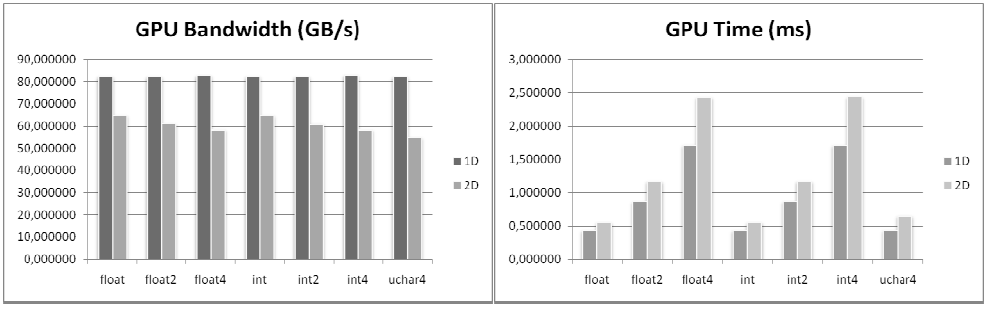

Figure 6. GPU bandwidth and processing time for different texture accesses

As well as previous executions, global memory writes have a better performance with types of 128-bit. Finally, the best launch configurations are the unidimensional ones. In the following simulations, all kernels were executed only with 1D launch configuration, since it demonstrated to provide better results than its 2D counterpart.

5.2 Sequential Memory Access

Among kernel performance issues, memory access has the greatest impact. As mentioned in Section 2, the 8800 GTX has four types of memory: shared, texture, constant and global memories, ranging from lower to higher access latency, respectively. This means that the more accesses are made to global memory, the higher will be the kernel processing time and the lower will be the bandwidth transfer. As a matter of fact, a single memory load of a word in global memory could take hundreds of cycles on that hardware. The same applies for memory storage, being more critical.

The CUDA Programming Guide [4] recommends that reads and writes should be coalesced whenever possible. Based on that, simulations were done to evaluate the 8800 GTX capabilities. The code sample in Figure 7 describes two kernel templates, the former storing a single value on global memory with coalesced writes, the later storing a single value with non-sequential writes. Both kernels were executed under the same conditions, with regard to launch configuration and number of elements to proces.

```
template <class T> __global__ void write_only(T* g_odata, T c) {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    g_odata[idx] = c;
}

template <class T> __global__ void write_only_t(T* g_odata, T c) {
    const unsigned int idxt = blockIdx.x + __mul24(threadIdx.x, gridDim.x);
    g_odata[idxt] = c;
}
```

Figure 7. Global memory write-only kernels

The results of bandwidth and processing time can be compared in the charts shown in Figure 8.

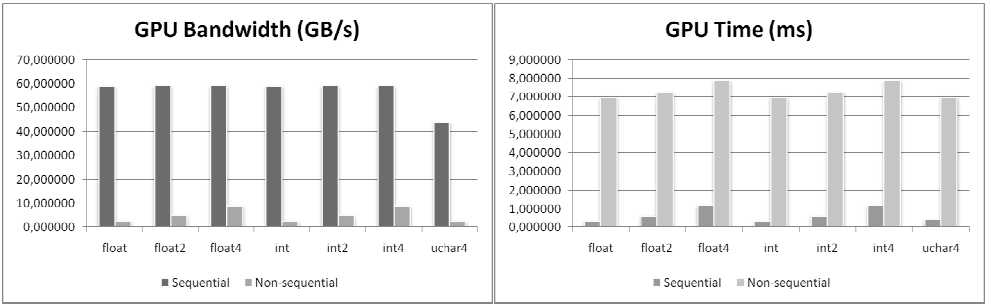

Figure 8. GPU bandwidth and processing time for sequential and non-sequential memory accesses

Regarding GPU processing time, the chart on the right of Figure 8 shows a great discrepancy between the two kernels results. Non-sequential writes were approximately 22 times worse than sequential ones for 32-bit types, 11 times worse for 64-bit types and 5 times worse for 128-bit types. Furthermore, the GPU times of the *write_only* kernel are practically linear with coalesced writes, i.e. the time elapsed for processing an amount of *int2* type elements is almost twice the time elapsed for processing the same amount of *int* type elements.

Focusing on the *write_only* kernel, in both charts can be noticed that *uchar4* type has a negative impact in performance. It was expected that its bandwidth and execution time were the same as with *int* or *float* types, since they have the same size. However, Figure 8 suggests that the 8800 GTX supports *uchar4* differently, imposing a severe limitation. In fact, what really occurs is that four loads are being made (seen in the PTX generated code), since *uchar4* is a vector type.

For global memory reads, it is also recommended that they be coalesced. The code sample in Figure 9 describes two kernel templates that test the 8800 GTX graphics card capabilities performing sequential (*read_only_gmem*) and non-sequential (*read_only_gmem_t*) reads. They make reads from global memory and store single values on shared memory.

```
template <class T> __global__ void read_only_gmem(T* g_idata, T c) {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    __shared__ T shared[BLOCK_SIZE];
    shared[threadIdx.x] = g_idata[idx];
}

template <class T> __global__ void read_only_gmem_t(T* g_idata, T c) {
    const unsigned int idxt = blockIdx.x + __mul24(threadIdx.x, gridDim.x);
    __shared__ T shared[BLOCK_SIZE];
    shared[threadIdx.x] = g_idata[idxt];
}
```

Figure 9. Global memory read-only kernels

The code sample was executed under the same conditions as the previous one. Results on bandwidth and execution time can be compared in both charts of Figure 10, respectively.

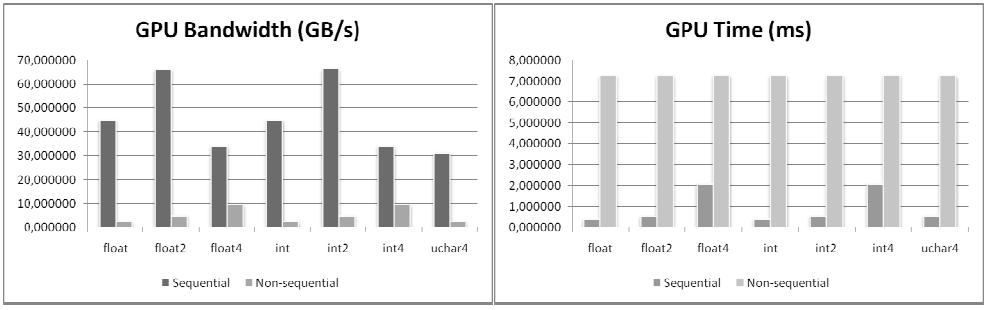

Figure 10. GPU bandwidth and processing time for different memory reads

It was expected that non-sequential reads performed worse than sequential ones, but an additional observation is that for all simulations, non-sequential results for GPU time were practically constant.

On these executions, *int*, *int2*, *float* and *float2* types indicated to be the best for sequential reads. Surprisingly, linear results were not obtained for *int4* and *float4* types, even though the generated PTX code is practically the same as those for *int2* and *float2* types. On the other hand, *uchar4* results were as expected: more processing time is necessary due to three additional store instructions.

5.3 Non-sequential Reading

It is a fact that reads show better performance results when done sequentially. However, some algorithms cannot always perform sequential reads, for example, convolution filters. As written in [4], reading device memory through texture fetching can be an advantageous alternative to reading device memory from global or constant one. Whenever possible, the developer should substitute global memory array accesses by the use

of textures. If textures are used by a kernel, the application must ensure that a bind operation is performed before the kernel is called, in order to inform the device about the memory region that encloses the texture. After kernel execution, the application must free texture resources by calling an unbind operation. This should be done so that further kernels that will be used are not harmed by unused texture space that was previously allocated.

In a non-sequential access context, some simulations were done comparing device memory reading through texture fetches against reading device global memory directly. The code sample in Figure 11 describes a vertical-only convolution kernel, which has a neighborhood radius of 5. It was implemented using non-sequential reads from an 1D texture (*convolve_V5_##type*) and from global memory (*convolve_V5*). Before running this benchmark, vector operators needed to be defined, to make possible the execution of sums and multiplications for all tested data types. A remark should be made: texture fetching is advantageous over global memory whenever more than one element access per thread occurs.

```
__global__ void convolve_V5_##type(type* g_idata, type* g_odata, type c) {
    const unsigned int loadPos = (blockIdx.x << 7) + threadIdx.x;\
    const unsigned int y = (loadPos >> 7);\

    type sum = c;\
    if((y >= 2) && (y < (gridDim.x - 2))) {\
        sum = sum + (tex1Dfetch(tex_##type, loadPos - 2*128)
            + tex1Dfetch(tex_##type, loadPos + 2*128))*0.0096200556f;
        sum = sum + (tex1Dfetch(tex_##type, loadPos - 1*128)
            + tex1Dfetch(tex_##type, loadPos + 1*128))*0.20542368f;
        sum = sum + tex1Dfetch(tex_##type, loadPos)*0.56991249f;
    }
    g_odata[loadPos] = sum;
}

template <class T>
__global__ void convolve_V5(T* g_idata, T* g_odata, T c) {
    const unsigned int loadPos = (blockIdx.x << 7) + threadIdx.x;
    const unsigned int y = (loadPos >> 7);

    T sum = c;
    if((y >= 2) && (y < (gridDim.x - 2))) {
        sum = sum + (g_idata[loadPos - 2*128]
            + g_idata[loadPos + 2*128])*0.0096200556f;
        sum = sum + (g_idata[loadPos - 1*128]
            + g_idata[loadPos + 1*128])*0.20542368f;
        sum = sum + g_idata[loadPos]*0.56991249f;
    }
    g_odata[loadPos] = sum;
}
```

Figure 11. Global memory and texture non-sequential read-only kernels

The kernels in Figure 11 were executed and the obtained results for bandwidth and execution time can be compared in charts shown in Figure 12.

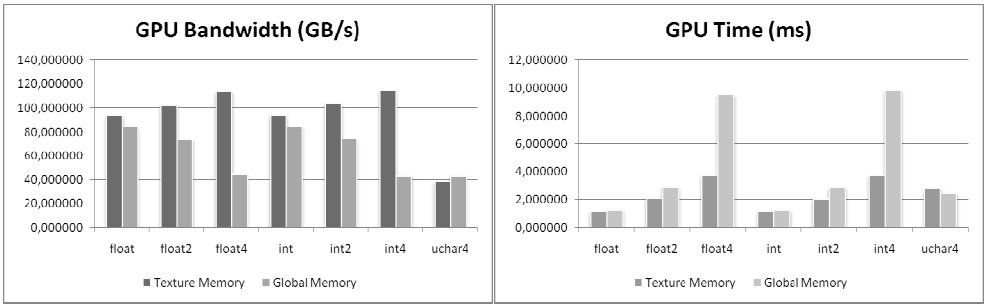

Figure 12. GPU bandwidth and processing time for non-sequential reads

Surprisingly, memory access through textures presents a great impact on performance. Their use gives a false impression that it has surpassed the theoretical 80GB/s bandwidth limit of the 8800 GTX graphics card. In fact, this is not true because texture memory is cached, and cannot be compared to that bandwidth limit. Apart from that, those results show that texture usage on non-sequential memory access is mandatory in real time constrained algorithms to achieve a higher performance.

5.4 Shared Memory Usage

Using shared memory improves the processing speed [4] when the previous global data is accessed more than once. It is common at the initial kernel stage to copy data from global memory to the region of memory shared by processing blocks. When utilizing shared memory, basically it is needed to pay attention to both size and synchronization. The first one is the size limit (only 16 KB per block), which restricts the amount of data that can be copied and shared by threads inside the same block. The second one, synchronization, is needed every time a thread in the same executing kernel requires data that comes as a result of another thread process. Because of that, there is a special function (the *__syncthread* one) which must be called to perform such synchronization, ensuring that different threads on the same block wait for the others to complete their task, guaranteeing data consistency.

There is no synchronization between different blocks, in such way that each block must access enough data to execute in an isolated way. The only way to force block synchronization is calling a kernel many times. At the end of each one, we are certain that all blocks have executed their tasks.

5.5 Page-locked Memory

Developers commonly allocate memory through the *malloc* function. Since such memory is paged, the device accesses it through CPU instructions and will have to poll the CPU to know when the memory transfer has ended. In CUDA, page-locked memory is

allocated through *cudaMallocHost* function. The advantage of using page-locked memory against paged one is that the device can access it directly, since it tracks the virtual memory ranges allocated with this function [4]. This avoids polling, thus increasing memory bandwidth transfers between host and device.

Even page-locked memory offering high bandwidth transfers, their usage should be moderated. The more page-locked memory is allocated, the fewer paged one is available, resulting in system performance degradation.

5.6 Loop Unrolling

Loop unrolling consists in taking some loop code and repeating it, instead of instructing the compiler to do so. This can be done by manually copying the code or using *pragma* directives, being this last option not recommended due to some limitations. Not all types of loops will improve performance using this technique, and, as with most CUDA optimizations, it is highly dependent on the algorithm being implemented and experimentation is very important.

5.7 Avoid Floating Point Conversion

A remark that should keep programmer's attention is the use of fractional numbers. Despite the fact that the 8800 GTX comes with a great number of floating point units, its 1.0 model does not support double precision manipulation directly, and it has to convert numbers of this type to a more suitable format (64-bit *float*).

The understanding of PTX assembly code by a high level C programmer is important, even if he/she is not going to write PTX code directly. For example, by the observation of the PTX code produced during one of our tests, we were able to notice that the compiler was generating instructions for manipulating 64-bit precision numbers, even if we were not expecting this behavior. Analyzing both PTX and C code, we found that every fractional constant that was used without being followed by the “f” letter was treated by the compiler as a double precision number. This fact made the compiler generating code to deal with 64-bit numbers, and as a consequence there was an unwanted lack of performance. By adding the “f” character after those constants, the compiler diminished the code, building an optimized version.

6 Case Study: KLT Tracker

Every AR system has a tracking stage that precedes the virtual information superposition algorithm. Some algorithms perform tracking based on edge detection, template matching, SIFT (Scale-Invariant Feature Transform) features, among others. The KLT (Kanade-Lucas-Tomasi) tracking technique [6] was chosen because it is generic enough

in a way that any registration can be performed by computer vision algorithms upon the features detected and tracked. An example is the use of tracked features to feed 3D reconstruction algorithms [7]. These algorithms are capable of discovering the 3D position of the features and based on it a segmentation algorithm can find specific geometries, such as planes, edges or even complex meshes by approximation.

No feature-based vision system can work unless good features are identified and tracked from frame to frame. In [6] an optimal feature selection criterion is proposed together with a method to track the selected features. The algorithm explores temporal information, which is extracted from the relative movement of the environment. The feature selection algorithm called Good Features to Track (GFTT) is based on the Harris corner detector, changing only the Harris cornerness expression to the Shi-Tomasi cornerness one. This expression represents the minimum eigenvalue of the 2x2 matrix that contains the gradient information of the search window [6] lowerbounded by a minimum acceptable eigenvalue (fixed threshold). Before the enforcement of the cornerness function, the Harris algorithm computes the summation of the squared gradients and the summation of the x and y relative gradient product against each other [8]. The KLT tracking stage uses a pyramidal approach from coarser to fine, searching for the features into a fixed window while looking for the minimum eigenvalue. This calculation is similar to the selection stage, using the same approach and solving linear equations to get the x and y displacements at each frame.

Implementation. All the guidelines described in Section 5 were applied to our case study implementation in order to perform source code optimization, increasing the kernel's data throughput. In this section, code excerpts will be used to illustrate where and why such optimizations were adopted. Kernel launch configuration is the entry point of our optimizations. As shown in Subsection 5.1, the optimal configuration found during tests performed is the one composed by unidimensional blocks and grids, with about 128 threads per block. The number of threads must be fixed near the next multiple of 16, necessary to perform the kernel execution. Another optimization was using textures as kernel input instead of accessing global memory directly. This choice is justified by the fact that the KLT algorithm needs to access pixels in different x and y coordinates, most of the time inside a search window. When the y coordinate changes, memory access becomes non-sequential. A code sample illustrating the optimizations given before can be found in Figure 13.

The *cudaBindTexture* function provides cacheable access to device global memory pointed by *d_img*. Inside the kernel, the *tex1Dfetch* function is used to address this region of memory, since in this case *d_img* corresponds to an unidimensional array of floats. We use textures whenever possible instead of global memory, according to the test results discussed in Subsection 5.3. The *convolveH5* kernel launch configuration (the expression between "<<<" and ">>>") has a block dimension of 144 threads, but the actual number of threads needed is 134. The remaining threads will belong to the block, despite the fact they are set to idle inside the kernel. The thread number padding corresponds to one of the optimization techniques described in Subsection 5.1: the block is completely divided into half warps, in a way the stream processors are better utilized. The same techniques used in the *convolveH5*

kernel may also be applied to the *convolveV5* kernel, as shown in Figure 13. The code sample in Figure 14 illustrates the use of shared memory, avoiding more than one texture memory access per thread, which might be costly.

```
cudaBindTexture(0, convolve_tex, d_img, SIZE_BYTES);
convolveH5<<< 1 << 13, 144 >>>(d_tmp_img1);
cudaThreadSynchronize();
cudaUnbindTexture(convolve_tex);

cudaBindTexture(0, convolve_tex, d_tmp_img1, SIZE_BYTES);
convolveV5<<< 1 << 13, 128 >>>(d_tmp_img2);
cudaThreadSynchronize();
cudaUnbindTexture(convolve_tex);
```

Figure 13. Kernel launch configuration (number of threads equals 128 or is near this value) and texture use (bind and unbind operations) code samples

```
__global__ void convolve7both1s(float* d_gradx, float* d_grady) {
  __shared__ float tmp[3 + 128 + 3 + 10];
  const unsigned int loadPos = (blockIdx.x << POT_THREADS) + threadIdx.x;
  const int x = (loadPos & (WIDTH - 1)) - 3;

  tmp[threadIdx.x] = tex1Dfetch(convolve_tex, loadPos - 3);
  __syncthreads();

  if(threadIdx.x < 128) {
    float sum1 = 0.0f;
    float sum2 = 0.0f;
    if((x >= 0) && (x < (WIDTH - 3 - 3))) {
      sum1 += (tmp[threadIdx.x+6]-tmp[threadIdx.x])*0.013353735f;
      sum1 += (tmp[threadIdx.x+5]-tmp[threadIdx.x+1])*0.10845453f;
      sum1 += (tmp[threadIdx.x+4]-tmp[threadIdx.x+2])*0.24302973f;

      sum2 += (tmp[threadIdx.x] + tmp[threadIdx.x+6])*0.0044330480f;
      sum2 += (tmp[threadIdx.x+1] + tmp[threadIdx.x+5])*0.054005578f;
      sum2 += (tmp[threadIdx.x+2] + tmp[threadIdx.x+4])*0.24203622f;
      sum2 += tmp[threadIdx.x+3]*0.39905027f;
    }
    d_gradx[loadPos] = sum1;
    d_grady[loadPos] = sum2;
  }
}
```

Figure 14. Code block for gradient calculation

The *convolve7both1s* kernel can be split into three distinct phases: loading, processing and storing. During the load phase, a single memory access per thread is performed. The processing phase makes use of the shared memory loaded to compute the gradient, taking

advantage of its fast access speed (compared to global memory). Finally, in the storing phase, each thread writes in a coalescent address, which speeds up the realized operations. The loop unroll optimization can also be noticed in the computation of sum1 and sum2 (shown in Figure 14), instead of a traditional loop code in which the float constants and memory accesses would be in an array. Another code sample that uses loop unroll optimization can be found in the enforce kernel, shown in Figure 15.

```
#define IFMAX2TEMP if(temp.x >= max2.x) max2 = temp;
#define __unroll_loop_enforce(i) \
  temp = sdata[threadIdx.x + (i)*(3+128+3 + 10)]; IFMAX2TEMP \
  temp = sdata[threadIdx.x + (i)*(3+128+3 + 10) + 1]; IFMAX2TEMP \
  temp = sdata[threadIdx.x + (i)*(3+128+3 + 10) + 2]; IFMAX2TEMP \
  temp = sdata[threadIdx.x + (i)*(3+128+3 + 10) + 3]; IFMAX2TEMP \
  temp = sdata[threadIdx.x + (i)*(3+128+3 + 10) + 4]; IFMAX2TEMP \
  temp = sdata[threadIdx.x + (i)*(3+128+3 + 10) + 5]; IFMAX2TEMP \
  temp = sdata[threadIdx.x + (i)*(3+128+3 + 10) + 6]; IFMAX2TEMP

...

float2 temp;
float2 max2 = make_float2(0.0f, 0.0f);
__unroll_loop_enforce(0)
__unroll_loop_enforce(1)
__unroll_loop_enforce(2)
__unroll_loop_enforce(3)
__unroll_loop_enforce(4)
__unroll_loop_enforce(5)
__unroll_loop_enforce(6)

if(p.x != max2.x) {
  features[pos].x = 0.0f;
} else {
  if(max2.y > p.y) {
    features[pos].x = 0.0f;
  } else {
    features[pos] = p;
  }
}
```

Figure 15. Code sample of the kernel that enforces a minimum distance between pixels feature selection and tracking stages

All memory allocation at host illustrates the pinned memory usage, by invoking the *cudaMallocHost* function. This type of allocation speeds up the memory transfer between host and device, as detailed in Subsection 5.5. The C language data types used to process and to exchange data between host and device were chosen according to Section 5. The *selectGoodFeatures* function makes use of a *float2* array to store the current features, besides other primitive data types.

Results. This section will present the results obtained with our GFTT and tracking algorithms, applied to four distinct scenarios. A sequence of 25 images with 1,024 rows per 1,024 columns (resolution) was used as input for each scenario. The maximum number of features to be tracked was fixed in 1,000 features. All scenarios were executed using a computer composed by an AMD Athlon 64 3200+ processor, 1 GB RAM, a NVIDIA GeForce 7900 GTX and a NVIDIA 8800 GTX graphics cards. Since in the starting phase of KLT algorithm there are no features selected, the application must perform the GFTT task in order to find the best features in the first frame of the image sequence. Figure 16.a shows the first input image of Scenario 2 and Figure 16.b shows its selected features, as result of the GFTT algorithm. Scenario 1 consists of a synthetic scene of a moving teapot (see Figure 17.1). In this scenario all 1,000 features can be tracked, but only a fraction of those are moving. The remaining scenarios (2, 3 and 4) were captured using a Microsoft VXCAM3000 webcam at a frame rate of 15fps and resolution of 1,280 rows per 1,024 columns. Once captured, the videos were cropped to fit in a 1,024 rows per 1,024 columns resolution. Scenario 2 comprises a real bunny doll wearing sun glasses standing over a table, while the camera is moving (see Figure 17.2). The goal of this scenario is to deal with fewer features, since only a small fraction of the detected ones is tracked. Scenario 3 represents an OpenGL book box turning and approaching the camera (see Figure 17.3). This scenario shows a less textured object with few trackable features. The last scenario comprehends some toys and the camera focusing at one of them (see Figure 17.4). In this scenario, 1,000 features can also be tracked, and all of those are moving. The expected results related with those input scenarios are smaller processing times in the ones that have fewer strong features, and higher processing times for the moving features cases. All visual results will be presented as follows.

For each frame in image sequence, some features may be lost. Figure 16.c, related to Scenario 1, shows a frame with less than 1,000 features selected, due to the loss of features during time. In order to replace the “dead features”, a reselection operation must be performed. This task consists of a tracking operation followed by a selection of new features capable of preserving the old valid ones. The reselected features for the last frame can be seen as dots in Figure 16.d. Reselecting new features, although being a requirement, is a time demanding operation, so it was only executed every 5 frames as a reselection policy.

Finally, the tracking algorithm will try to track the selected features across every remaining frame. Figure 17 shows the features tracked since the first frame, for each scenario. All tracked trajectories are represented by lines. Each stage of the KLT algorithm was analyzed, and for each scenario the processing times were gathered. The results show significant improvements in all stages of the KLT algorithm and a detailed analysis of those times will be further discussed.

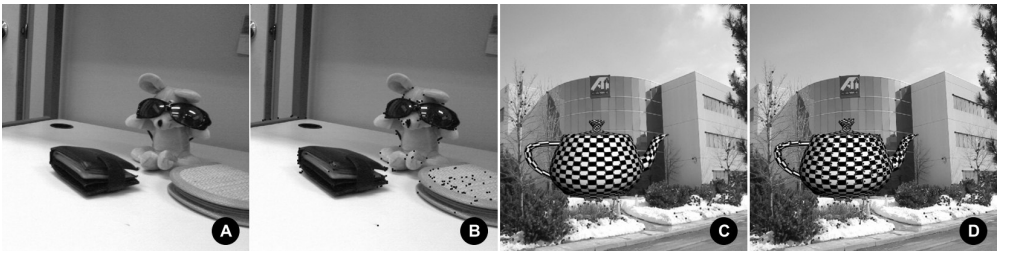

Figure 16. GFTT and feature reselection: a) Scenario 2 input image; b) Scenario 2 features; c) Scenario 1 input image; d) Scenario 1 features

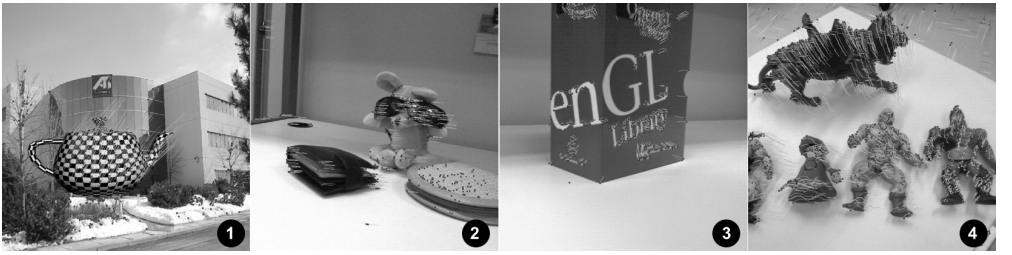

Figure 17. Tracked features on each scenario

Performance comparison. In this section, we will compare the execution of our GFTT algorithm implementation against the GPU KLT implementation, which is the KLT reference implementation for GPU [10]. This implementation also uses a massively parallel approach, but applying shaders as a GPGPU aide. In our implementation, each KLT algorithm stage (GFTT, feature tracking and feature reselection) was subdivided into steps, which were analyzed regarding processing time. Before presenting the acquired results, we will show a memory transfer comparison between OpenGL and pure CUDA functions. This test was performed using as input a 1,024x1,024 32-bit image (only 1 monochromatic channel) for the CUDA KLT and a 1,024x1,024 8-bit image for the GPU KLT. The goal was measuring the average transfer time in both directions, host to device and device to host. The functions invoked for uploading the image to the GPU device memory were *glTexImage2D* and *cudaMemcpy*, in OpenGL and CUDA, respectively. The data bandwidth results can be seen in Table 1.

Table 1. Memory transfer comparison results: time (milliseconds) and bandwidth (mega binary bytes)

	GPU_KLT		CUDA_KLT	
Algorithm step	Execution time	Bandwidth	Execution time	Bandwidth
Host to device transfer	8.27	120.9344	2.65	1,509.4784
Device to host transfer	19.78	50.5856	5.42	737.9968

As shown in Table 1, the bandwidth achieved in CUDA is much higher than the one in OpenGL with even lower timing values. This occurs because the transfer time and bandwidth in CUDA are optimized (also by using page-locked memory). This optimization presents significant disparities when dealing with HD content, which is a primary feature of next generation AR applications. The bandwidth improvement reaches 12.48 times in the upload case, and 14.59 times in the download case. Both implementations had their algorithms processed by the 8800 GTX, except that ours used the 7900 GTX as display device. Since the GPU KLT implementation uses shaders, it needs to do computation and display on the same graphics card. An important observation is that it was not possible to compare our feature tracking implementation with the GPU KLT feature tracking stage implementation on the 8800 GTX, since it did not work in this graphics card. Each scenario described earlier was given as input to both implementations and was analyzed with regard to processing time.

The parameters supplied to the applications were the same default parameters of the reference CPU implementation [9], which are pyramid levels (fixed at 2 levels) and tracker's search window (fixed at a 7x7 window). After setting these parameters, the applications were run under the same CPU conditions (memory usage and running process) and compilation settings (release optimization flags). All comparisons were made against the same scenarios described before. The results for our CUDA KLT algorithm implementation can be seen in Table 2, where each application step execution time was also measured.

Regarding Table 2, results did not present large variations for all different scenarios. Their execution times were similar, depending mainly on the number of features tracked. The GPU sorting, for example, is used to sort 220 elements, so its time is almost uniform across every scenario. The GFTT algorithm presents almost the same value in each scenario, since it is basically composed by filtering functions, whose performance is not influenced by the input image content. Furthermore, since all scenarios are distinct, the number of tracked features varies. This has a direct impact on feature tracking execution time, so the acquired times were not as uniform as the one with GPU sort, for example. In addition, the same can be said about feature reselection, primarily for scenarios 2 and 3, since they present a significant lost of features, which diminishes the algorithm execution time, when compared against scenarios 1 and 4. The acquired GFTT results shown in Table 2 and results gathered from the GPU KLT execution were compared, as can be seen in the left chart of Figure 18.

Table 2. CUDA KLT timing results (milliseconds)

Algorithm Step	Scenario 1	Scenario 2	Scenario 3	Scenario 4
Good Features to Track	46.470989	45.776768	45.898571	46.246941
GPU Sort	29.849045	29.690645	29.707129	29.757133
Feature Tracking (first time)	23.731787	21.052397	18.066822	25.064079
Build Pyramid 1	2.992280	2.975797	2.959035	4.738871
Build Pyramid 2	7.051176	5.490363	5.600712	7.357918
Track Features	13.684979	12.582604	9.504002	12.963938
# of Features Tracked	984	995	1,000	1,000
Feature Tracking (remaining times)	18.947102	17.056915	13.687773	19.904764
Build Pyramid 2	5.537855	5.543442	5.537016	5.562997
Track Features	13.192459	11.510122	8.147125	14.338135
# of Features Tracked	971	249	367	999
Feature Reselection	27.538696	16.118250	18.405691	22.419331

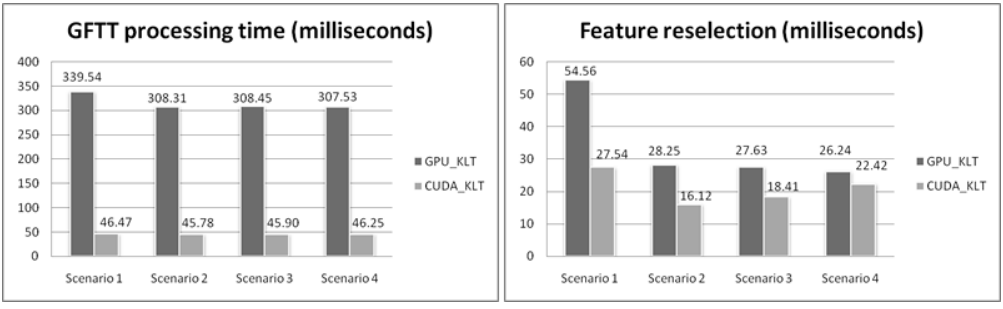

Figure 18. GFTT and feature reselection execution times comparison

A similar comparison was made between the feature reselection execution times and results are shown in the right chart of Figure 18. Our implementation has shown a speed up of about 630% in the GFTT selection, and between 17% and 98% in the reselect algorithm, with 1,000 moving features an with a few moving features, respectively, compared to the GPU KLT implementation. Furthermore, it can achieve frame rates of 50fps, enabling its use for real time MAR applications.

7 Final Considerations

GPGPU technology applies to general MAR related problems as well as to application domain specific ones. One of the most important steps of MAR systems' pipeline is real time tracking. Since multiple features should be tracked in large images, this process is unfeasible to be performed using a CPU approach. Therefore, there are already some important contributions related to interest point based techniques and tracking of corners and edges, implemented using this technology.

Massive data processing applications have for a long time demanded expensive dedicated hardware to run. This has been a major drawback for the deployment of real time applications based on AR, MAR, and computer vision techniques. The technology presented in this survey may be capitalized on to help breaking this frontier. This new approach should bring image processing of HD videos to the desktop arena. Using this approach, we can unify the CPU and GPU programming, and maintain time costly algorithms running concurrently with a sophisticated HD MAR pipeline.

The CUDA programming model is very attractive and easy to tune when following some guidelines and definitely worth the learning effort needed by the architecture. Although this survey is an initial step, it is important to share some design guidelines including thread arrangement, sequential memory access, non-sequential reading, shared memory usage, loop unrolling, and page-locked memory while avoiding the use of floating point conversion. These programming guidelines can help developers improve the performance of their CUDA programs executed on top of the GeForce 8 Series.

References

[1] Grama, A., Gupta, A., Karypis, G. and Kumar, V., Introduction to Parallel Computing (Second Edition), Addison-Wesley, 2003.

[2] Intel Quad Core. Available: Intel site. URL: http://www.intel.com/technology/quad-core/index.htm, visited on 30/11/2007.

[3] Azuma, R., Baillot, Y., Behringer, R., Feiner, S., Julier, S. and MacIntyre, B. (2001) “Recent advances in augmented reality”, IEEE Computer Graphics and Applications.

[4] NVIDIA CUDA Programming Guide. Available: NVIDIA site. URL: http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf, visited on 04/11/2008.

[5] ATI Close To Metal. Available: Sourceforge site. URL: http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf, visited on 04/11/2008.

[6] J. Shi, C. Tomasi, Good features to track, in: IEEE Conference on Computer Vision and Pattern Recognition (CVPR'94), Seattle, 1994. URL citeseer.ist.psu.edu/shi94good.html

[7] M. Pollefeys, Self-calibration and metric 3d reconstruction from uncalibrated image sequences, Ph.D. thesis, ESAT-PSI, K.U.Leuven (1999).

[8] C. Harris, M. Stephens, A combined corner and edge detector (1988).

[9] S. Birchfield, Klt reference implementation. URL http://www.ces.clemson.edu/ stb/klt/

[10] S. N. Sinha, J.-M. Frahm, M. Pollefeys and Y. Genc, GPU-Based Video Feature Tracking and Matching, EDGE 2006, workshop on Edge Computing Using New Commodity Architectures, Chapel Hill, May 2006.