Abstract

Single processor technology has been evolving across last decades, but due to physical limitations of chip manufacturing process, the industry is pursuing alternatives to sustain computational power growth, including the creation of multi-core systems. Parallel computing targets problems that are scalable and possibly distributed, dividing the problem into smaller pieces. This approach may be explored to satisfy real time constraints required by augmented reality algorithms. Its use may vary from realistic generation of virtual scenes to 3D reconstructions and image processing to perform optimized tracking. In this paper, the labeling algorithm, commonly used in the augmented reality pipeline, is implemented using a massively parallel paradigm. The implementation is able to provide satisfactory speed up improvements using CUDA, NVIDIA’s architecture for GPU programming. The aim of this paper is not to present a new technology, but to show the great improvements that can be obtained by applying it in computer vision and augmented reality applications.

1. Introduction

Single processor technology has been evolving across last decades. For years, programmers have created their applications based on the premise that an enhancement in performance meant a higher clock frequency of operation. Unfortunately, because of physical and architectural bounds, there are limitations on the computational power that can be achieved with a single processor system. Due to physical limitations of chip manufacturing process, 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 the GPU (Graphics Processing Unit) industry. Since nowadays GPUs are seen as high performance units, this capability made parallel processing paradigm stronger. It demands another type of development, and not every application can take advantage of it.

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 (through OpenMP [1], or some third part library); use of a distributed solution (using 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.

An alternative for making 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 with image generation and rendering, named GPGPU (General Purpose computing on Graphics Processing Unit), is commonly seen for algorithms in the image processing area, computer vision, and several research areas that can make benefit of massively parallel processing. GPUs usually process millions of pixels per second through its several pixel shading pipelines (a common high definition 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
attempt of making GPGPU available for the average programmer, and a step into the High Performance Computing (HPC) market. G80 hardware is accessed through a low-level parallel thread execution virtual machine, and a virtual instruction set architecture called PTX. 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) [4], that has a very well defined low level interface, NVIDIA preferred to focus on a C-based language, although it still requires a great knowledge of how the GPU works, and how it is organized.

The GeForce 8800, the first G80-based card, has 16 groups of 8 scalar processors each, totaling 128 processors, which are configured to run at 675 MHz. Since the execution unit uses double pumped frequencies, the resulting clock is 1350 MHz. 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, and are used by the processor for scheduling, 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 into grids of blocks are defined by the programmer, as seen in Figure 1.

The massive parallel computing approach may be explored to satisfy real time constraints required by augmented reality (AR) applications. Its use may vary from realistic generation of 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 paper, the labeling algorithm (commonly used in the traditional AR pipeline) is reimplemented using the new massively parallel computing paradigm, delivered by NVIDIA CUDA [3].

The G80 processor, NVIDIA’s first implementation of Compute Unified Device Architecture (CUDA), is an 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.

The massive parallel computing approach may be explored to satisfy real time constraints required by augmented reality (AR) applications. Its use may vary from realistic generation of 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 paper, the labeling algorithm (commonly used in the traditional AR pipeline) is reimplemented using the new massively parallel computing paradigm, delivered by NVIDIA CUDA [3].

Despite the fact that this architecture represents a brand new technology, the implementation is able to provide very satisfactory speed up improvements (the transition between sequential and parallel paradigms will be discussed in Section 4). The authors highlight the possibility of working with images from High Definition (HD) videos (about 1080p). Before the massively processing technology, the idea of processing images with resolutions above 800x600 pixels was nearly unachievable, due to the use of sequential or even ordinary parallel approaches. The aim of this paper is not to present a new technology, but to show the great improvements that can be obtained by applying it in computer vision and AR applications.

The next sections are divided as follows. Section 2 explains both hardware and software infrastructure provided by CUDA. Section 3 presents related works that also use GPGPU for performing tasks done traditionally in a sequential way. Section 4 details the case study chosen, in which the authors’ labeling algorithm implementation is described. The basic sequential algorithm is presented and then its correspondence and modifications on the GPGPU side. Section 5 describes our test scenarios and lists the results obtained. At last, Section 6 gives some conclusions about the possibilities of using the GPU as a coprocessor for enhancing AR functions performance, and points some future works in this area.

2. CUDA

The G80 processor, NVIDIA’s first implementation of Compute Unified Device Architecture (CUDA), is an attempt of making GPGPU available for the average programmer, and a step into the High Performance Computing (HPC) market. G80 hardware is accessed through a low-level parallel thread execution virtual machine, and a virtual instruction set architecture called PTX. 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) [4], that has a very well defined low level interface, NVIDIA preferred to focus on a C-based language, although it still requires a great knowledge of how the GPU works, and how it is organized.

The GeForce 8800, the first G80-based card, has 16 groups of 8 scalar processors each, totaling 128 processors, which are configured to run at 675 MHz. Since the execution unit uses double pumped frequencies, the resulting clock is 1350 MHz. 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, and are used by the processor for scheduling, 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 into grids of blocks are defined by the programmer, as seen in Figure 1.

Figure 1. The G80 execution model [3].
Each multiprocessor contains 8 generalized processors (or Scalar Processors - 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. The main explanation is that speed, rather than precision, was a major concern when rendering 3D scenes. An integer multiplication, for example, is processed by SFU and requires 8 cycles, while the lower precision mode of this instruction (24 instead of 32 bits) can be executed by the standard processors in only 2 cycles.

The memory model implemented on the G80, shown in Figure 2, defines the following scope for read/write operations [3]: per-thread for registers, per-thread for local memory, per-block for shared memory, and per-grid for global memory. For constant memory and texture memory, per-grid read-only operations are enabled, and only reads from texture and constant memory are cached. Some of the memory is located on chip, like registers and shared memory, while texture, constant, local, and global memory are implemented in device memory.

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

Another issue when developing using CUDA is to appropriately feed the G80 with enough threads for execution, as the processor can schedule millions of threads. This way, 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 they can improve CUDA applications.

NVIDIA constructed a powerful piece of hardware, accompanied by useful software and even a line of products for high demand customers. As CUDA is a hardware platform, it can be expected even more processing power on years to come, for less money; and as the competitors wish to enter this market, it possibly will be seen improvements on ATI’s side as well.

3. Related work
GPGPU computation has been widely adopted in several research areas, such as audio and signal processing [5], medicine and biological science [6], image processing, among others.

Focusing in the image processing area, which is directly linked with AR systems, many algorithms have been ported to the GPU platform using fragment programs to have access to the hardware as a co-processing unit. Some examples of image processing techniques
implemented in GPU are found in GpuCV [7], an implementation of the OpenCV [8] library using acceleration in GPU.

Also, there is the GPU_KLT [9], a well known feature tracker based in the original KLT algorithm [10], which is used by most 3D reconstruction systems. This work is also implemented using fragment programs, which suffers from hardware optimization.

Another library focused in parallel algorithms, this time using CUDA, is the CUDDP [11] (CUDA Data Parallel Primitives). CUDDP's objective is to group data parallel algorithm primitives in order to execute tasks like sorting, stream compaction, building data structures such as trees and summed-area tables (using a scan algorithm). CUDDP was created initially to wrap up the implementation part of the algorithm described in [12].

All these projects aim to speed up an algorithm or technique, minimizing the contribution of the time spent by the procedure in the entire process. It is important to take advantage of the time that was initially used by the algorithm to do more tasks and achieve better results, spending the same time.

4. Case study

Labeling of a binary image refers to the act of assigning an unique value to pixels belonging to the same connected region [13]. Many vision problems can be posed as labeling problems in which the solution to a problem is a set of labels assigned to image pixels or features. This algorithm is often exploited in marker based AR applications [14], and because of that fact it was chosen as the case study for this work.

In formal mathematical terms, the labeling problem is to assign a label from the label set \( \mathcal{L} \) to each of the sites in \( S \). Edge detection of an image, for example, is to assign a label \( f_i \) from the set \( \mathcal{L} = \{\text{edge, non-edge}\} \) to site \( i \in S \) where elements in \( S \) index the image pixels. The set \( f=\{f_1, \ldots, f_m\} \) is called a labeling of the sites in \( S \) in terms of the labels in \( \mathcal{L} \). When each site is assigned to an unique label, \( f=f(i) \) can be regarded as a function with domain \( S \) and image \( \mathcal{L} \). Because the support of the function is the whole domain \( S \), it is a mapping from \( S \) to \( \mathcal{L} \), that is, \( f: S \rightarrow \mathcal{L} \). A labeling is also called a coloring in mathematical programming.

Since the labeling problem relies on image's global information for associating the connected regions, the processing unit must have access to all pixels inside its search range. In order to have the labeling algorithm implemented in parallel, one should be tempted to divide the problem in two consecutive steps: local labeling processing and further merge of the obtained labels. There is a tradeoff between these two steps: the easier and more parallel the local labeling step is, the more complex the merge becomes. In the approach used for performing this case study, the authors decided to perform a simple local labeling and at the same time create a hybrid (sequential/parallel) merge implementation.

The parallel labeling implementation is generic enough to divide an input image of \( m \times n \) pixels in \( m_x \times n_y \) blocks, each one able of being processed independently from the others. The variables \( m_x \) and \( n_y \) represent the division factor along the \( x \) and \( y \) axes, respectively. As a consequence, the size of each image block should be \( \frac{m}{m_x} \times \frac{n}{n_y} \) pixels, as shown in Figure 3.

![Figure 3. Input image divided in blocks](image)

The local labeling algorithm implemented was the one described in [13]. The method scans a binary image from top to bottom and from left to right, per each line. Conceptually, the operations can be divided into four major steps that are illustrated in Figure 4a to Figure 4d.

In Figure 4a, when an external contour point, say \( A \), is encountered the first time (it does not have any label yet), a complete trace of the contour is made until returning to the first point. A label is then assigned to \( A \) and to all points of that contour.

In Figure 4b, when a labeled external contour point \( A' \) is encountered, the scan line to find all subsequent black pixels (if they exist) is followed and assigned with
the same label as \( A' \).

In Figure 4c, when an internal contour point, say \( B \), is encountered the first time, \( B \) is assigned to the same label as the external contour of the same component. Then the internal contour containing \( B \) is traced and the same label as \( B \) is also assigned to all contour points.

In Figure 4d, when a labeled internal contour point, say \( B' \), is encountered, the scan line to find all subsequent black pixels (if they exist) are followed and the same label as \( B' \) is assigned to them.

The next step, the merge process, must wait until all processing blocks conclude their operations. In other words, the processing time of the local labeling phase is limited by the slowest processing block.

The hybrid merge is described as follows. At first, all blocks perform the boundary check at the same time (as shown in Figure 5), and then each one generates a segment of the list of label equivalences.

After that, a sequential loop iterates along the generated list of equivalences, and using a recursive approach the label values are grouped.

### 5. Results

The algorithm presented in Section 4 was implemented using the sequential approach on the PC platform, and using the massive parallel paradigm on the GPU. The computer used was an AMD Athlon X2 4800 processor with 1GB of RAM, running Windows© XP SP2. The device used as a coprocessor in the parallel approach was a NVIDIA GeForce 8800 GTX, with 768MB of GDDR3 memory. The tests were conducted in a non-biased environment.

The authors took for test an image composed by interleaved concentrical 1-pixel rectangles, which is considered a worst case scenario, because the trace function will walk through every rectangle in the image. The picture used has a resolution of 960x720 pixels, and a zoomed part can be seen in Figure 6.

For the sequential implementation, the average running time was 28.465 milliseconds, measured through 100 iterations, discarding outliers (about 3%). Using the GPU, the average running time was 4.858599 milliseconds, measured on the same terms of the sequential way, which gave us a speed-up of 5.858x, from 35.13 to 205.79 frames per second. The output of the algorithm is shown in Figure 7.

Another test scenario was used in order to obtain a performance comparison for the average case. The
adopted input image was the one shown in . It is composed by 3 different square patterns, and resulted from the binarization process. The sequential algorithm was executed in 9.649 milliseconds, while using the CUDA parallel approach it lasted the same amount of time corresponding to the worst case scenario. This represents a speed up of 1.9862x, and the parallel duration can be justified by the fact that since there are less black pixels in the image to work on, there are threads not being used. This way, the processing time of the algorithm in parallel is almost constant.

The use of CUDA or any other massively parallel programming approach is a key to apply to the real time constraints in computer vision and AR offline results. Physics simulation, photorealistic rendering, 3D reconstruction for interaction between real and virtual worlds, and richer inputs and outputs to AR algorithms (High Definition Augmented Reality) are a good starting point to apply that new paradigm.

At this time, the use of a GPU as C coprocessor is only possible with last generation video cards, which have an average cost of US$500.00 (NVIDIA GeForce 8800 GTX). The cost-benefit relation is very good, considering the additional 500 GFLOPS (Giga Floating-point Operations per Second) provided by the video card model used to produce the results presented in this paper.

6. Conclusions and future work

The labeling algorithm taken as reference in this work is not the best example to explore the device's computational power, since it was created to deal with sequential approaches. However, by simply applying a divide-and-conquer technique, keeping the entire logical structure and adding a posterior merge phase, the authors just got the same algorithm running around 205 frames per second, almost 6 times faster than the sequential implementation on CPU.

Other algorithms included in the AR pipeline, even the simpler ones, like threshold, are better to transcript to the massive parallel approach due to its intrinsic independence of their executing parts. Taking threshold as an example, the data independence is in pixel level, since each pixel is compared to a fixed reference value, which means that the number of threads can be up to the number of pixels in the image.

Future work relative to the labeling algorithm consists basically in optimizing the code to remove the expensive instruction calls. It will increase even more the frame rate achieved.

The speed-up could be improved by the addition of a warm-up phase, which consists of a kernel code that accesses the texture and constant data from the device in the CUDA implementation, in order to intensify the use of their respective caches, minimizing the memory access time. Another optimization that could also be done was to convert the expensive operations (especially multiplication) to cheaper ones (mul24 instructions or shift operations, which take less clock cycles to be executed).

7. References


