B. Tech. Project

Second Stage Report on

GPU Based Active Contours

Submitted by

Sumit Shekhar

(05007028)

Under the guidance of

Prof Subhasis Chaudhuri

Table of Contents

1. Introduction

1.1 Graphic Processor Units

A graphic processor unit is simply a processor attached to the graphics card used in video games, play stations and computers. The way they are different from the CPUs, the central processing units, are that they are *massively threaded* and *parallel* in their operations. This is because of the nature of their work – they are used for fast rendering; same operation is carried out for each of the pixels in the image. Thus, they have *more* transistors devoted to *data processing* rather than flow control and data caching.

Today graphic processor units have outdated their primary purpose. They are being used and promoted for scientific calculations all over the world by the name of GPGPUs or general purpose GPUs; engineers are achieving several times speed-up by running their programs on GPUs. Applications fields are many: image processing, general signal processing, physics simulations, computational biology, etc, etc.

1.2 Problem Statement and Motivation

Graphic Processor Units can speed-up the computation manifolds over the traditional CPU implementation. Image processing, being inherently parallel, can be implemented quite effectively on a GPU. Thus, many applications which are otherwise run slowly can be fastened up, and can be put to useful real-time applications. This was the motivation behind my final year project.

NVIDIA *CUDA* is a parallel programming model and software, which has developed specifically to address the problems of efficiently program the GPU as well as be compatible with a wide variety of GPU cores available in the market. Further, being an extension to the standard C language, it presents a low-learning curve for the programmers, as well giving them flexibility to put in their creativity in the parallel codes they write.

The problem statement was to implement active contour algorithms on GPU. Active contour algorithms are solved using two approaches based on the way of representing the segmenting boundary: parametric methods and level set. The evolution of these curves can be a slow process for big images, as the number of points of contour goes up. I explored various aspects of these algorithms and how to parallelize them to gain the most time advantage on GPU. To achieve the same, I also explored the CUBLAS library which provides several optimized linear algebra functions, implemented in CUDA.

2. GPU Programming Model

2.1 Execution

The GPU programming model is very different from a CPU programming model as it inherently supports parallel programming. A usual template followed to program a GPU is as below:

- Transfer the data to the GPU memory.
- Launch the GPU kernel from the host program.
- Wait till GPU finishes off the calculations.
- Transfer the results back to CPU memory.

Thus, a CUDA program consists of two parts in its code: a) **host program** which runs on CPU and b) **kernel** which does the calculations on GPU. A brief description of the various programming features is as below:

- The execution of the kernels is done in *threads* and *blocks*. The structure of the grid of threads and blocks is passed in the host code while calling the kernel using a unique <<<…>>> syntax. A typical kernel call looks like:

  ```
  funcAdd<<<Grid, Block, 1>>>(A, B, C)
  ```

- These threads and blocks are executed on the various multiprocessors of the GPU. The threads of each block are executed concurrently in one multi-processor; each of them may process some maximum number of blocks depending on the resources used by each thread. As the blocks get executed, new blocks are launched on the vacated processors.
- The threads are processed in groups of 32 called *warps*. The splitting into warps is done according to increasing thread IDs.
- Each thread and block has a unique ID, which is defined by a 3 component vector namely **`threadIdx`** and **`blockIdx`** respectively. The thread IDs are with respect to their position in the block. A typical grid of threads and blocks is shown in figure 1.
- Threads within a block can synchronize using a barrier synchronization function **`__syncthreads()`**. This function stops the execution of threads until all the threads have reached that point.
- The threads are executed in parallel unless their paths diverge or they read/write from the same memory location. In the former case, the processor executes each path serially, disabling threads on other paths until the paths converge again. In the latter case too, the access is serialized, but in the case of non-atomic write, the outcome of the write is not guaranteed. Also the order in which access occurs is undefined.

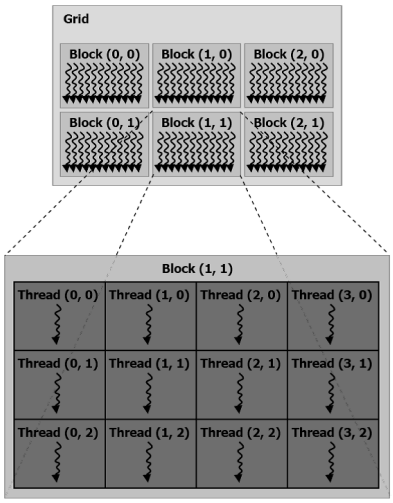

Figure 1: Grid and Thread Blocks

2.2 Memory Hierarchy

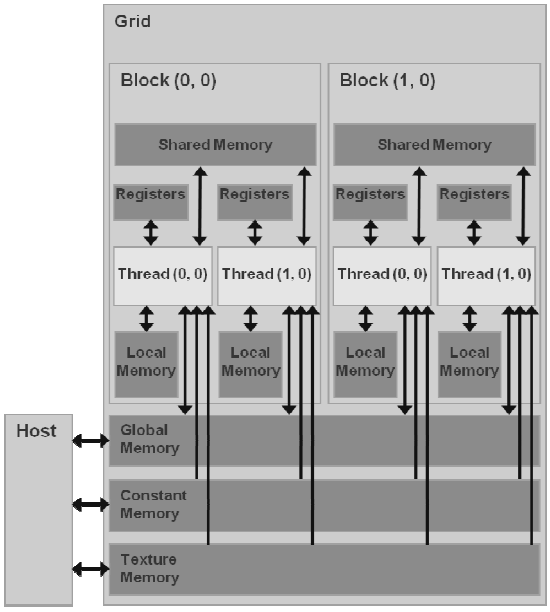

Figure 2: GPU Memory Model

GPU has different levels of memory – global memory, shared memory and local memory and registers.

- **Registers and local memory**

  Each multiprocessor has certain number of 32-bit registers. They are allocated per thread and only that thread can access the contents of the register. These are the fastest accessible memories, and are allocated at the compile time depending on the requirement of a thread. If the threads in a block need more registers than provided in a multiprocessor, the kernel fails to launch. *Local Memory* is also available per thread and the compiler automatically allocates certain variables on the local memory, like a big array. Local memory is not cached and hence its access is as costly as global memory.

- **Shared Memory**

  Shared memory is an on-chip memory and hence, it is faster than local or global memory. Every multiprocessor has 16kB of shared memory, and is accessible by all the threads in a block. The shared memory of one block cannot be accessed by another block even if they are running on the same multiprocessor. To achieve high bandwidth, the shared memory is divided into *memory banks,* which can be accessed simultaneously. If threads in the same half-warp access the same bank address, then there is a *bank conflict* and the access is serialized, thus reducing the effective bandwidth.

- **Global Memory**

  It is an off-chip memory and is accessible to all threads. The data transferred from CPU is stored in the global memory initially. The memory is connected to all the processors through a wide bus, which can be up to 512 bits wide. Since the memory is not cached, access to the global memory is expensive and should be avoided as far as possible. The memory bandwidth is effectively utilized when the threads in a half-warp access the memory in a sequence, resulting in *coalescing* of the memory transactions.

- **Texture Memory**

  CUDA also supports a part of the texturing hardware which is used for graphics processing. This allows the texture memory to be read using device functions called *texture fetches.* As the texture memory is cached, it is much faster than global memory access. It also supports various attributes like *texture coordinates*, which can be used to access the texture elements, interpolation options, normalization, etc.

2.3 Hardware Implementation

GPU derives its computing from an array of Streaming Multiprocessors (SMs). Each multiprocessor consists of eight Scalar Processor cores and a multithreaded instruction unit. The multiprocessor handles multiples threads using the architecture called SIMT, which maps each thread to a scalar processor. Further each multiprocessor has on-chip memory of different types: a set of 32-bit *registers,* an on-chip *shared memory* shared by all the scalar processors, a *constant cache* to speed up reading from constant memory and a *texture cache* which speeds up read from the texture memory space. A schematic of the GPU hardware is shown in figure 3.

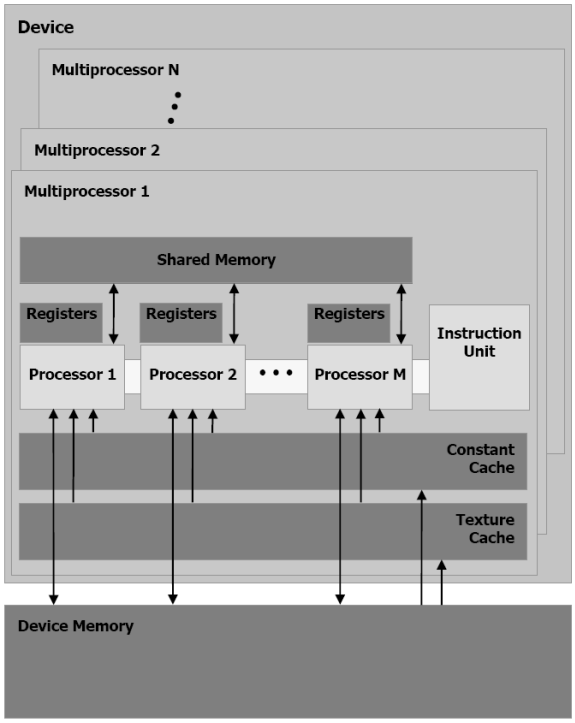

Figure 3: Multiprocessors on GPU

3. Active Contours

3.1 Introduction

Active contours or "snakes" are a very popular model in computer vision and has been used for segmentation of object of interest in an image, shape modelling, edge detection, etc. This is based on the evolution of an initial curve to minimize the integral of energy functional over the curve. The energy functional can be defined either in terms of image gradient or region based model. These approaches differ in their region of capture or in the accuracy of detection of object edges. Using *Euler – Lagrange's Equation*, this problem can be converted to time-domain evolution of the curve with a force function acting on the curve, causing the curve to be drawn to the object boundary.

3.2 Parametric Snake Model

A typical snake can be thought of as a curve **x**(s) = [x(s), y(s)], s ε [0, 1] which minimizes the energy functional, E [1]:

$$E = \int_0^1 \frac{1}{2}(\alpha|\boldsymbol{x}'(s)|^2 + \beta|\boldsymbol{x}''(s)|^2) + E_{ext}(\boldsymbol{x}(s))ds$$

For gradient-based approach, the energy term can be taken as, for example:

$$E = -|\nabla(G_\sigma(x,y) * I(x,y)|^2$$

Where the image I(x, y) is convolved with different Gaussian kernels to blur the object boundary and hence increase the region of capture of the contour. Thus using the Euler equation, we get the minimization equation as:

$$\alpha\boldsymbol{x}''(s) + \beta\boldsymbol{x}'''(\boldsymbol{s}) + \nabla E_{ext} = 0$$

This terms can be viewed as a combination of internal and external forces $\mathbf{F} = \mathbf{F_{ext}} + \mathbf{F_{int}}$, the first term drawing the curve to the edges and the second term keeping the curve smooth. Thus the curve evolution can be seen as:

$$\boldsymbol{x}'(s,t) = \boldsymbol{F}(s,t)$$

Hence, the curve will stabilize at the position where **F** = 0, reaching the steady state.

3.3 Curve Representation

B-splines are often used to model the curve used for evolution. This method represents the curve in terms of a basis function and control points [2]. Thus a curve **P**(t) is represented as:

$$\boldsymbol{P}(t) = \sum_{i=1}^{n+1} N_{i,k}(t)\boldsymbol{P}_i \quad t_{min} \leq t \leq t_{max}$$

Here, $\mathbf{P}_i$ are the control points, *i = 1, 2, 3... (N + 1)* control points and $N_{i,k}(t)$ are the basis functions of order $k >= 2$. To define the basis function, a knot vector which is a non-decreasing sequence $(t_1, t_2, \ldots, t_{(n+k+1)})$ is specified which determines the points at which *t* the pieces of curve join together. The basis function is defined recursively in terms of *k* and the knot vector as:

$$N_{i,1}(t) = \begin{cases} 1 & t_i \leq t < t_{i+1} \\ 0 & otherwise \end{cases}$$

$$N_{i,k} = \frac{t - t_i}{t_{i+k-1} - t_i} N_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} N_{i+1,k-1}(t)$$

For the purpose of curve evolution here, a closed curve is assumed and the knot vector is taken as uniform.

3.4 Algorithm for tracking

The parametric active contour tracking can thus be done using the following procedure:

- **Inputting the initial curve**

  The user gives the initial input as a set of points close to the desired object of interest. Given this set of points, the parameters of the B-spline are estimated based on least mean square approximation. This is done assuming a closed curve and uniform knot vector.

- **Calculating the force term**

  The $\mathbf{F}_{ext}$ depends only on the input image and needs to be calculated once. Assuming a gradient based energy term, the force function can be calculated. This can be done in one step, if taken as mentioned above or iteratively if, for example, GVF (gradient vector field) is calculated.

- **Evolving the curve**

  Having estimated the initial curve and the force function, the curve can now be updated. The curve evolution can be done at a large number of points on the curve. Neglecting the internal force terms for now, we can write the updating equation as:

  $$\boldsymbol{x}'(s,\tau) = \boldsymbol{F}(s)$$

For each point on the curve, ***x(s, τ)*** can be expressed in terms of the basis function. Hence we have:

$$\sum_{i=1}^{n+1} N_{i,k}(t)\Delta \boldsymbol{P}_i \;=\; \boldsymbol{F}(t)d\tau$$

Where **ΔP** is the change in the co-ordinates of the control point and $d\tau$ is the time-step. Thus, this can be expressed as a matrix equation of the form:

$$C_{N\ x\ N_b}\Delta P_{N_b x\ 2} = F_{N\ x\ 2}d\tau$$

Where, N_b = number of control points, N = number of points where curve is evolved, C = matrix of the values of basis function at the points of evolving curve. The two columns of F and ΔP matrices are for x and y coordinates respectively.

The equation can be solved by taking the pseudo-inverse of the matrix C. The control points can now be updated by adding the **ΔP** to the initial co-ordinates. The process will be repeated until the change in the co-ordinates is very small.

3.5 GPU Implementation

For optimizing the code on GPU, following guidelines were used:

- There should be minimum memory transfers between CPU and GPU as it is a slow process and can considerably bring down the performance.
- Access to global memory should be kept low as the global memory is not cached; hence its access is slow. For images, texture memory can be used for faster access of pixels.
- Enough thread blocks should be issued so that all the multi-processors are activated. The number of blocks should at least be the number of multiprocessors on the device. This is useful because large number of blocks means large amount of global memory transfers. Thus, the scheduler will need to time-slice the memory access, and this can help hiding the memory access latency.
- Launching multiple kernels can be efficiently used for *global synchronization*, as the kernel launch overhead is very less of the order of 10-15 μs. Moreover, the CUDA kernel launches are synchronized by default.
- CUBLAS library functions can be used to efficiently for operations like matrix multiplication, vector products, etc. However, there is no in-built function for solving Ax = b equation for general A. Hence, other methods need to be looked into.

The algorithm for tracking was implemented as follows:

- The ***initial estimation*** of the B-spline curve can be computed on CPU itself as the algorithm involves calculation of control points. As the number of control points are limited, say to 20 – 30 points, the GPU calculation would be not very much efficient than CPU. Moreover, this efficiency may be reduced to extra overheads of transferring data between CPU and GPU.

- ***Calculation of F(x, y)*** using the gradient method involves pixel wise calculations. Hence, the image was stored in texture memory for faster access. Further, $\mathbf{F} = \nabla E_{ext} = -\nabla|\nabla(G_\sigma(x,y) * I(x,y)|^2$. So, for efficiency, the Gaussian kernel was pre-computed in CPU and passed onto the kernel. Also to minimize global memory accesses, both the gradient values were calculated in a single kernel, temporary variables being used to store intermediate values. Further, the facility in texture memory of clamping coordinate values was used to get rid of the boundary problem.

- Curve evolution involves two steps:

  a) ***Calculating the matrix C:*** this involves calculating the value of basis functions $N_{i,k}$ at every point. The calculation of $N_{i,k}$ is recursive, hence a single kernel cannot be used, as the GPU functions cannot be recursive. To solve this, multiple kernels were launched with increasing values of k. This is an efficient operation as mentioned before.

  b) ***Solving the equation $C\Delta P = Fdt$:*** The equation can be solved by taking the pseudo-inverse of the matrix C. To achieve this, the equation is first multiplied by C^T to get a square matrix in LHS:

  $$(C^T C)\Delta P = C^T F dt$$

  But as CUBLAS does not support solving general linear equation, other methods need to be looked into. Here as $C^T C$ is symmetric and positive semi-definite for any C, hence *Conjugate Gradient Method* can be applied. This method can be implemented using CUBLAS functions.

3.6 Results

The calculations were done using NVIDIA GeForce 8600 GPU and the CPU was Intel Quad 4 core processor.

1. Calculation of Force function, F_{ext}

Image Size	GPU (8 x 8)	GPU (16 x 16)	CPU	Speed-up
48 x 96	145 ms	130 ms	1137 ms	6.6
512 x 512	220 ms	206 ms	1470 ms	8.5
576 x 768	255 ms	245 ms	3390 ms	10.8

2. Calculation of matrix, C

Time was measured for the execution of the kernel without data transfer part in both GPU and CPU cases. This is to measure the time taken per iteration in this step.

N	**K**	**CPU (ms)**	**GPU (ms)**
200	10	0.81	0.88
400	10	1.60	1.64
400	5	0.50	0.74

Hence, we can see that both GPU and CPU take negligible time, and no speed advantage can be drawn.

4. Future Works

- The conjugate-gradient method can be implemented and tested for speed-up for different instances of matrix C.
- The parametric contour problem was not fully solved. It can be implemented completely and the performance can be compared with CPU version.
- The algorithm can be tested on different kinds of images under different forcing functions to be made more robust.

5. References

[1] Chenyeng Xu and Prince: Gradient Vector Flow: A new external source for Snakes, CVPR, 1997.

[2] B splines: http://www.cl.cam.ac.uk/teaching/2000/AGraphHCI/SMEG/

[3] NVIDIA CUDA Programming Guide.