CUDA Implementation of Chan-Vese Algorithm for Image Segmentation

Prateeth Vishwanath Nayak, Darshan Mallapur Vishwanath, Sanjay Yellambalase Ravikumar, Gautham Sureshkumar, Abhishek Sutar, Xiangyi Chen, Yuejun Ma

ABSTRACT

Image segmentation is the process of extraction of a region of interest for an object. It is a major preprocessing step for object identification and image analysis. Usually, image segmentation applies to large images which required immense computation. The computations as of now are primarily done over the CPU cores which are more time-consuming. Some algorithms require to iterative computing of the pixel information such as intensity, gradient, texture, etc. For large images, this is a very cumbersome job. Hence, it has a huge potential for exploiting parallelism. Using the GPUs, we can address each pixel concurrently. In this project, we focused on the Chan-Vese algorithm for image segmentation and implemented it using CUDA.

Keywords

Image segmentation; Chan-Vese algorithm; CUDA; Active contour; Region of interest; Object detection;

1. INTRODUCTION

We are considering the image segmentation algorithm called level set methods which is widely used in medical diagnostic imaging. Level set methods, also called active-contours, are an efficient way to isolate any particular ROI (region of interest). The method involves initializing a mask over the image and then using the pixel information for narrowing down to the ROI.

One well-known algorithm for image segmentation is Chan-Vese [1]. The basic idea of the algorithm is to separate the foreground and background based on our region of interest(ROI). The algorithm will evolve a curve, subject to constraints from a given image in order to detect objects in that image. At the beginning, the curve can be initialized at a random position in the image, gradually moving towards the interior normal of the object to be detected, thereby stopping at the boundary of the ROI. While there are some work trying to improve the performance of Chan-Vese from algorithmic side [3], we try to accelerated the algorithm by exploiting parallelism in the algorithm.

We define the curve as a sign-distance-function (SDF), where the regions within the curve are negative, and the regions exterior to the curve are positive. At the boundary of the curve, the SDF is zero. After every iteration, the curve evolves based on the pixel information near to the boundary. We can rely on the region-based evolution, where the pixel means intensities interior and exterior of the curve mapped image is computed for each iteration. The curve stops at the boundary of the ROI where there is an immense contrast in the pixel intensity. Hence, this algorithm is used in images with intensity inhomogeneity.

Based on the Chan-Vese algorithm in [1], we need to calculate the mean intensities for pixels in the level set and out of the level set, the gradient of each pixel, the force pushing the curve towards the boundary of the objects, and the curvature of the objects. If we can do the calculation concurrently, we will get a decent speedup for the program. We started the CUDA implementation by first converting these functions into the CUDA kernel functions.

2. DESIGN OVERVIEW

The overview of our design is shown in Fig. 1 which follows from Chan-Vese algorithm, the blocks in the figure need to be serialized, we can only exploit parallelism within each block of the algorithm. We write a CUDA kernel functions computation in the while loop, leading to 6 kernels overall. All the computation are done in GPU within loop to avoid intensive memory copies between device and host.

3. IMPLEMENTATION

3.1 Kernel Functions

3.1.1 Mean Kernel

We use the mean kernel to calculate the sum of pixel values inside and outside the contour. We also use it to calculate the number of pixels inside and outside contour. In this algorithm, the distance between the contour and a pixel inside the contour is considered negative, and the distance between the contour and a pixel outside the contour is considered positive. We use privatization here to implement the mean kernel. Each block is given a private copy of sum and count variables on shared memory. The sum and increment operations are performed atomically. The private sum and count values are added to the global sum and count variables atomically. The computation complexity of this kernel is $O(n^2)$. The bottleneck of kernel is that atomicAdd

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

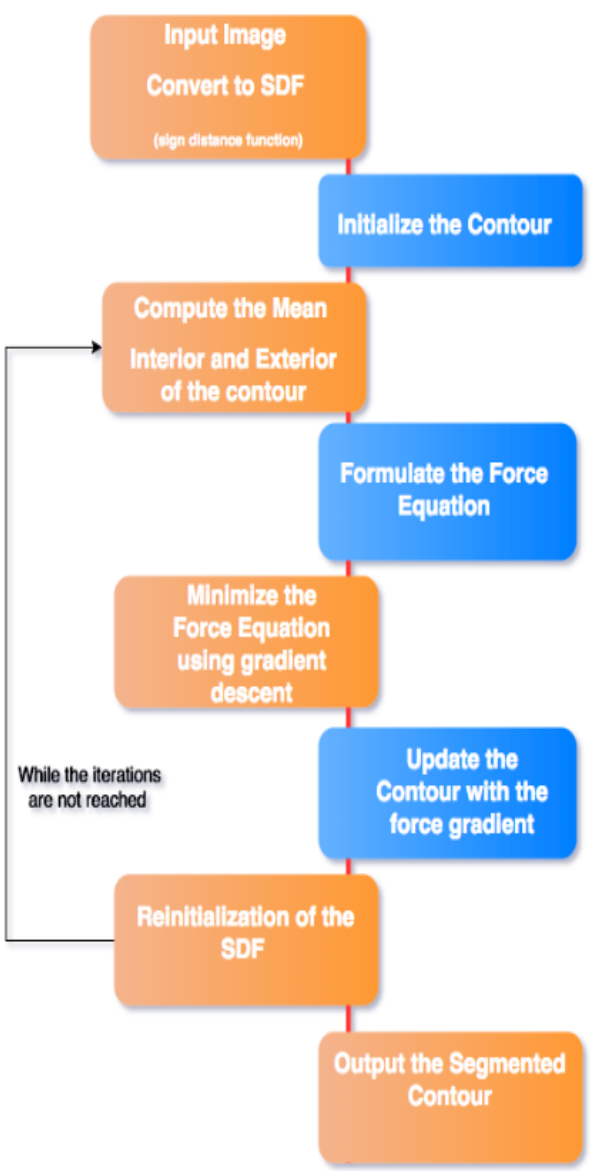


Figure 1: Design overview

within a thread block will be serialized. The code for mean kernel is shown in Fig. 2.

3.1.2 *Force Kernel*

The force kernel computes force values of each pixel from image information and mean values from mean kernel by taking difference of squared difference of pixel data from positive and negative means. It keeps track of the maximum of absolute value of force values which will be used in gradient descent equation. Since force computation for each pixel is independent, we made each thread compute one output. For computing maximum, we made use of shared memory and atomicMax function. Each block maintains a copy of shared max value which will be updated by all the threads in a block using atomicMax. And then we update max across blocks. The time complexity of this kernel is $O(n^2)$. The bottleneck of this kernel is that atomicMax within a thread block will be serialized. The code of the force kernel is shown in Fig. 3.

```
__global__ void mean_kernel(float* phid, unsigned int* datain_d,
                unsigned int height, unsigned int width,
                unsigned int* sum_neg_d, unsigned int* sum_pos_d,
                int* c_neg_d, int* c_pos_d)
{
    __shared__ unsigned int s_sum_neg[32];
    __shared__ unsigned int s_sum_pos[32];
    __shared__ int s_c_neg[32];
    __shared__ int s_c_pos[32];

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int tx = threadIdx.x;
    // private copies of sum and counters
    if (tx < 32)
    {
        s_sum_neg[tx] = 0;
        s_sum_pos[tx] = 0;
        s_c_neg[tx] = 0;
        s_c_pos[tx] = 0;
    }
    __syncthreads();
    // calculate sum and count
    if (tid < height*width)
    {
        if (phid[tid] <= 0)
        {
            atomicAdd(&s_sum_neg[tx%32], datain_d[tid]);
            atomicAdd(&s_c_neg[tx%32], 1);
        }
        else
        {
            atomicAdd(&s_sum_pos[tx%32], datain_d[tid]);
            atomicAdd(&s_c_pos[tx%32], 1);
        }

        __syncthreads();
        if(tx<32)
        {
            atomicAdd(&s_sum_neg[0], s_sum_neg[tx]);
            atomicAdd(&s_sum_pos[0], s_sum_pos[tx]);
            atomicAdd(&s_c_neg[0], s_c_neg[tx]);
            atomicAdd(&s_c_pos[0], s_c_pos[tx]);
        }
        if (tx == 0)
        {
            // accumulate global sum and counter
            atomicAdd(sum_neg_d, s_sum_neg[0]);
            atomicAdd(c_neg_d, s_c_neg[0]);
            atomicAdd(sum_pos_d, s_sum_pos[0]);
            atomicAdd(c_pos_d, s_c_pos[0]);
        }
    }
}
```

Figure 2: Mean Kernel Code

3.1.3 *Gradient Kernel*

The gradient kernel computes the gradient values for each pixel to direct the contours toward the boundaries of desired objects. The gradient computation for each pixel is done using the curvature and the force values computed for each pixel by other kernels. It keeps track of the maximum gradient value among the pixels which is used by the Contour Evolution Kernel. The gradient computation for each pixel is independent and thus each thread computes gradi-

```
__global__ void force_kernel(float* phid, unsigned int* datain_d, float* F_d,
unsigned int height, unsigned int width, double* maxF_d, int* stop_d,
unsigned int* mean_neg_d, unsigned int* mean_pos_d, int* c_neg_d, int* c_pos_d)
{
    __shared__ double s_maxF[32];
    __shared__ float mean_neg;
    __shared__ float mean_pos;
    //__shared__ int s_stop;

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int tx = threadIdx.x;
    // mean of inside and outside of 0 level set
    if (tx == 0)
    {
        mean_neg = *mean_neg_d / ((float)(*c_neg_d) + 0.00001);
        mean_pos = *mean_pos_d / ((float)(*c_pos_d) + 0.00001);
    }
    if (tx < 32)
    {
        s_maxF[tx] = 0;
    }
    __syncthreads();
    float phiVal = phid[tid];
    float dataVal = datain_d[tid];
    if (tid < height*width)
    {
        // calculate the force pushing the boundary
        float val = 0.0f;
        if (phiVal < 1.2 && phiVal > -1.2)
        {
            val = (dataVal - mean_neg) * (dataVal - mean_neg)
                - (dataVal - mean_pos) * (dataVal - mean_pos);
            atomicMaxDouble(&s_maxF[tx%32], fabs(val));
            F_d[tid] = val;

        }
        __syncthreads();
        if (tx < 32)
        {
            atomicMaxDouble(&s_maxF[0], s_maxF[tx]);
        }
        if (tx == 0)
        {
            atomicMaxDouble(maxF_d, fabs(s_maxF[0]));
        }
    }
}
```

Figure 3: Force Kernel Code

ent for each pixel. For computing maximum, we made use of shared memory and atomicMax function. Each block maintains a copy of shared maximum gradient value which will be updated by all the threads in a block using atomicMax and then we update maximum gradient value across blocks. The computation complexity of this kernel is $O(n^2)$. The bottleneck of this kernel is that atomicMax within a thread block will be serialized. The code of this kernel is shown in Fig. 4.

3.1.4 Contour Evolution Kernel

This kernel does the evolution of the contour. The phi values for the pixels are computed using the old phi values and the gradient values of the pixels given by the Gradient Kernel. It keeps track of the maximum gradient value among the pixels which is used by the CFL Kernel. The gradient computation for each pixel is independent and thus each thread computes gradient for each pixel. For computing maximum, we made use of shared memory and atomicMax function. Each block maintains a copy of shared max value which will be updated by all the threads in a block using atomicMax. And then we update max across blocks. The time complexity of this kernel is $O(n^2)$. The bottleneck of this kernel is that atomicMax within a thread block will be

```
__global__ void gradient_kernel(float* phid, float* curvature_d,
        float* F_d, float* dphidt_d, double* max_dphidt_d, double alpha,
        double* maxF_d, unsigned int height, unsigned int width)
{
    __shared__ double max_dphidt_s;

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int tx = threadIdx.x;

    if (tx == 0)
        max_dphidt_s = 0;

    __syncthreads();


    if (tid < height*width)
    {
        // calculate gradient only near boundary of 0 level set
        if (phid[tid] < 1.2 && phid[tid] > -1.2)
        {
            dphidt_d[tid] = (F_d[tid] / *maxF_d) + alpha * curvature_d[tid];
            atomicMaxDouble(&max_dphidt_s, (double)dphidt_d[tid]);
        }

        __syncthreads();

        // max of gradient for normalization
        if (tx == 0)
        {
            atomicMaxDouble(max_dphidt_d, max_dphidt_s);
        }
    }

}
```

Figure 4: Gradient Kernel Code

serialized. The code of this kernel is shown in Fig. 5.

3.1.5 Curvature Kernel

The curvature penalty kernel is used for smoothing the evolving curve and maintaining the regularization. The kernel computes the curvature along SDF. Curvature is computed using central derivatives of SDF at x, y. The central derivatives at x, y are computed using phi values of 8 surrounding pixels. Each pixel makes 8 global memory access for phi values of 8 surrounding pixels. Since curvature computation for each pixel is independent, we made each thread compute one output. Curvature value along with 'alpha' is used to compute gradient descent values for each pixel. Alpha determines the smoothness of the curve. Higher the value smoother the curve. The time complexity of this kernel is $O(n^2)$. The bottleneck of this kernel is the global memory access.

3.1.6 Re-Initialization of SDF Kernel

This kernel is also called sussman kernel [2]. This kernel reinitialize SDF so that it remains to be a signed distance function. This is done by solving a PDE using a iterative algorithm. The re-initialization involve pixel-wise computation which can be done in parallel. Each pixel update depend on its direct neighbor pixels, a new copy of SDF is needed. To avoid memory copy, contour evolution kernel write to a new copy of SDF and re-initialization kernel write back to the original copy. Each pixel can be reused at most 4 times, a version of kernel using shared memory for SDF is tried (worse performance, more computation needed for computing indices, more registers needed, bank conflict, original kernel may access L2 cache). The computation complexity

```
__global__ void CFL_kernel(float* phidCFL, float* phid,
    float* dphidt_d, double* max_dphidt_d,
    unsigned int height, unsigned int width)
{
    __shared__ float dt;
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    // calculate step size
    if (threadIdx.x == 0)
    {
        dt = 0.45 / (*max_dphidt_d + 0.00001);
    }

    __syncthreads();
    if (tid < height*width)
    {
        // update only pixels near boundary of 0 level set
        if (phid[tid] < 1.2 && phid[tid] > -1.2)
        {
            phidCFL[tid] = phid[tid] + dt * dphidt_d[tid];
        }
        else
        {
            phidCFL[tid] = phid[tid];
        }

    }
}
```

Figure 5: Contour Evolution Kernel Code

of this kernel is $O(n^2)$. The code of this kernel is shown in Fig. 7.

```
__global__ void curvature_kernel_shared(float* curvature_d,
          float* phid, unsigned int height, unsigned int width)
{
    float phi_x, phi_y, phi_xx, phi_yy, phi_xy;
    __shared__ float phi_s[BLOCK_SIZE][BLOCK_SIZE];
    int tid_y = threadIdx.x + blockIdx.x * (BLOCK_SIZE-2);
    int tid_x = threadIdx.y + blockIdx.y * (BLOCK_SIZE-2);
    int iny = threadIdx.x + blockIdx.x * (BLOCK_SIZE-2) -1;
    int inx = threadIdx.x + blockIdx.x * (BLOCK_SIZE-2) -1;

    inx = inx > 0 ? inx : 0;
    inx = inx < width ? inx : width - 1;
    iny = iny > 0 ? iny : 0;
    iny = iny < height ? iny : height - 1;

    phi_s[threadIdx.x][threadIdx.y] = phid[inx*height + iny];
    __syncthreads();
    if (tid_x<width&&tid_y<height)
    {
        int l_x = threadIdx.x;
        int r_x = threadIdx.x+2;
        int m_x = threadIdx.x+1;
        int u_y = threadIdx.y;
        int d_y = threadIdx.y+2;
        int m_y = threadIdx.y+1;
        if (phi_s[m_x][m_y] < 1.2 && phi_s[m_x][m_y] > -1.2)
        {
            // differences between neighbors
            phi_x = -phi_s[l_x][m_y] + phi_s[r_x][m_y];
            phi_y = -phi_s[m_x][u_y] + phi_s[m_x][d_y];
            phi_xx = phi_s[l_x][m_y] + phi_s[r_x][m_y] - 2 * phi_s[m_x][m_y];
            phi_yy = phi_s[m_x][u_y] + phi_s[m_x][d_y] - 2 * phi_s[m_x][m_y];
            phi_xy = 0.25*(-phi_s[l_x][u_y] - phi_s[r_x][d_y]
                           + phi_s[r_x][u_y] + phi_s[l_x][d_y]);
            // curvature calculation
            curvature_d[tid_x*height + tid_y] = phi_x*phi_x * phi_yy
                                                + phi_y*phi_y * phi_xx
                                                - 2 * phi_x * phi_y * phi_xy;
            curvature_d[tid_x*height + tid_y] = curvature_d[tid_x*height + tid_y]
                                                / (phi_x*phi_x + phi_y*phi_y + 0.00001);
        }
        else
        {
            curvature_d[tid_x*height + tid_y] = 0;
        }
    }
}
```

Figure 6: Curvature Kernel Code

4. VERIFICATION

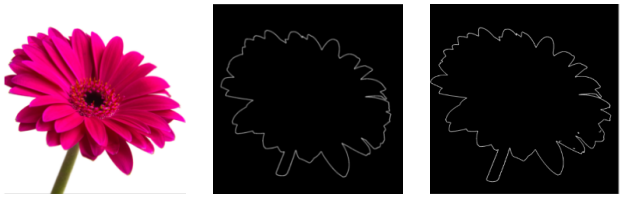
We ran the sequential and CUDA code on the CPU and GPU, and transferred the output files for sequential and parallel and displayed the results using an ImageViewer application.The CPU specifications include Intel Xenon 1.6Ghz quad-core and the GPU specifications include NVIDIA GeForce GTX 1080 8GB GDDR5X 1.73GHz. To verify the correctness of our implementation, we compared the converted image from CPU and GPU. If the two images we compare are quite similar, we can verify that our design is good in precision and robustness. The comparison results we have for different images are shown in Fig. 8, Fig. 9, Fig. 10, and Fig. 11:

Original CPU GPU

Figure 9: Apple, size 400 × 456

Original CPU GPU

Figure 8: Tomato, size 200 × 200



Original CPU GPU

Figure 10: Flower, size 1280 × 1280

```cuda
__global__ void sussman_kernel(float* phidd, float* phid,
                               unsigned int height, unsigned int width)
{
    int r_x, l_x, u_y, d_y;
    float d_phid;
    float a, b, c, d;

    int pixly = threadIdx.x + blockIdx.x * blockDim.x;
    int pixlx = threadIdx.y + blockIdx.y * blockDim.y;
    float plxly = phid[pixlx*height + pixly];
    if (pixlx < width && pixly < height)
    {
        // neighbor indices
        l_x = pixlx - 1 > 0 ? pixlx - 1 : width - 1;
        r_x = pixlx + 1 < width ? pixlx + 1 : 0;
        u_y = pixly - 1 > 0 ? pixly - 1 : height - 1;
        d_y = pixly + 1 < height ? pixly + 1 : 0;

        // sign of the value
        float sussman_sign = plxly / sqrtf(plxly * plxly + 1);
        // calculate difference between neighbor pixels (with thresholding)
        if (plxly > 0)
        {
            a = fmax((float)(plxly - phid[l_x*height + pixly]), (float)0);
            b = fmin((float)(phid[r_x*height + pixly] - plxly), (float)0);
            c = fmax((float)(plxly - phid[pixlx*height + d_y]), (float)0);
            d = fmin((float)(phid[pixlx*height + u_y] - plxly), (float)0);
            d_phid = sqrtf(fmax(a*a, b*b) + fmax(c*c, d*d)) - 1;
        }
        else if (plxly < 0)
        {
            a = fmin((float)(plxly - phid[l_x*height + pixly]), (float)0);
            b = fmax((float)(phid[r_x*height + pixly] - plxly), (float)0);
            c = fmin((float)(plxly - phid[pixlx*height + d_y]), (float)0);
            d = fmax((float)(phid[pixlx*height + u_y] - plxly), (float)0);
            d_phid = sqrtf(fmax(a*a, b*b) + fmax(c*c, d*d)) - 1;
        }
        else
        {
            d_phid = 0;
        }
        // update phi
        phidd[pixlx*height + pixly] = plxly - 0.5 * sussman_sign * d_phid;
    }
}
```

Figure 7: Sussman Kernel Code

Original CPU GPU

Figure 11: Nvidia, size 2256 × 2160

5. PERFORMANCE

To show the performance of our implementation, we measured the running time of both CPU and GPU for different image sizes. Since the CPU time is much larger than the GPU time, we use the logarithmic scale for values. The comparison between the CPU and GPU time is shown in the analysis graph. To have a better view of the performance, we calculated the speedup for different image sizes. As we can see from the speedup graph, the performance increases drastically with the increasing of image size. The best performance in our implementation can reach 500 when the image size is about 2000x2000.

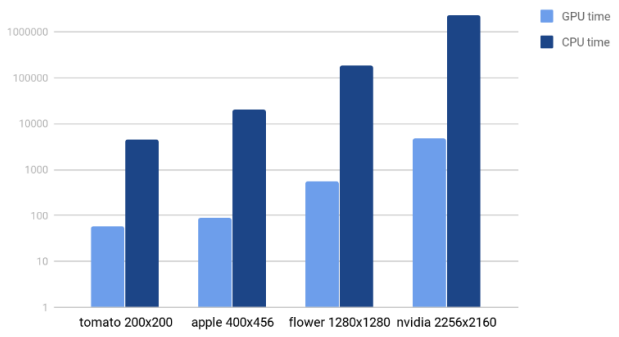


Figure 12: Performance Analysis

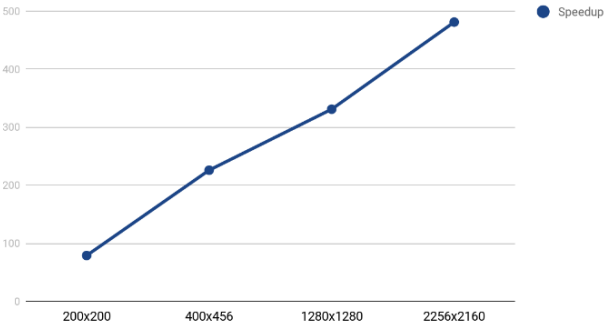


Figure 13: Speedup

5.1 Limitations

There are some limitations in our project. The first is that there is not much reuse of data for image segmentation. What we did was calculating the related values for each pixel concurrently. Due to this limitation, we cannot fully exploit the use of shared memory.

The second limitation is that there are some data dependence between each kernel function. The results from mean kernel will be used in other kernel functions. We used cudaMemcpy in the first version of our implementation. We found that using cudaMemcpy is not efficient. The reason is that there will be multiple memory copy between the host and the device. To eliminate the use of cudaMemcpy, we put it in the kernel functions. Although we put the cudaMemcpy in the kernel functions, we still have data dependence.

The third limitation is that we used a lot of atomic operations in mean, force, and gradient kernel function. We can try to use scan to replace the atomic operations. This could result in a better performance.

The last limitation is that there is always divergence in the sussman and mean kernel function. In the mean kernel, when it compares the mean values with 0, there is a divergence. For now, we haven't found a way to eliminate it. There may be some further optimization for this limitation.

Function Name	Duration (µs)	Occupancy	Registers per Thread	Static Shared Memory per Block	Allocated Warps Per Block	Allocated Registers Per Block	Allocated Shared Memory	Max Block Limit	Max Block Limit	Block Limit Reason	Achieved FLOPS: Double GFLOPS	Achieved FLOPS: Single GFLOPS
set	4.544	0.5	17	0	1	768	0	64	84	Blocks	0	0
mean_kernel	152.768	1	25	16	8	8192	256	8	8	Warps, Registers	0	0
force_kernel	279.584	0.75	36	16	8	10240	256	8	6	Registers	0.014600263	0.030109019
curvature_kernel	105.408	0.75	36	0	8	10240	0	8	6	Registers	0.141848816	0.263433515
gradient_kernel	189.92	0.75	36	8	8	10240	256	8	6	Registers	0.078727885	0.016870261
CFL_kernel	83.456	0.75	36	4	8	10240	256	8	6	Registers	0.024456294	0.031238392
sussman_kernel	294.208	0.75	37	0	8	10240	0	8	6	Registers	0	6.253338636

Figure 14: Statistics

6. CONCLUSION

In this project, we proposed a CUDA implementation of the well-known Chan-Vese algorithm for image segmentation. We implemented six kernel functions and optimized the project several times. The best performance we can achieve is about 500 times faster than CPU. The future goal of this project is to improve the performance using further optimization.

7. ACKNOWLEDGEMENT

We would like to thank Professor John Sartori for his guidance and suggestions to further improve the performance of our program by making use of shared memory in curvature penalty kernel and multiple copies of 'max' variables in the mean and force kernel to reduce serialization imposed by the atomic operations. All these measures resulted in overall improvement in the performance.

8. REFERENCES

[1] Chan, T. F., and Vese, L. A. Active contours without edges. *IEEE Transactions on image processing 10*, 2 (2001), 266–277.

[2] Sussman, M., Smereka, P., and Osher, S. A level set approach for computing solutions to incompressible two-phase flow. *Journal of Computational physics 114*, 1 (1994), 146–159.

[3] Wang, X.-F., Huang, D.-S., and Xu, H. An efficient local chan–vese model for image segmentation. *Pattern Recognition 43*, 3 (2010), 603–618.