EVALUATION SEGMENTATION ON GPUS

ABDUL DAKKAK

ABSTRACT

Image segmentation is an important and well studied problem in computer vision. The task is to label pixels, grouping them into the same category. In medical imaging, for example, one needs to classify a liver from the rest of the image, or discern the shape of cells to detect any diseases. Sadly, segmentation is slow — taking on the order of minutes. Speeding up segmentation would not only improve existing applications, but also open the possibility of new ones.

In this project we evaluate image segmentation on the GPU. We look at Cellular Automaton (CA) based approaches for binary segmentation (classifying pixels as either background or foreground). We try different optimizations on both CPUs and GPU and perform a comparison.

1 CELLULAR AUTOMATON (GROWCUT) BASED SEGMENTATION

GrowCut [1] is an algorithm that uses ideas from cellular automaton to perform image segmentation. GrowCut flood neighbors based on an initial seed until it reaches a barrier. One can think of each pixel as a bacteria with an energy function, if a bacteria has higher energy than one of its neighbors, then it will devour them — otherwise it is the victim. The algorithm is iterated until either a fixed point is reached or the maximum number of iterations are reached. A pseudocode of the algorithm is shown in listing 2. For our experiments, we set `MAX_ITERATIONS` to 2048 and use a penalty function $g(x, y) = 1 - \|x - y\|_2$.

Listing 1: Pseudo Code for GrowCut Segmentation

```
Input: Image, Approximate Label
Output: Label Sgementation
-----------------------------
changed = true
do_wile changed && iterations < MAX_ITERATIONS:
  changed = false
  for ii from 0 to height:
    for jj from 0 to width:
      cp = image[ii][jj]
      nl = lp = label[ii][jj]
      ns = sp = strength[ii][jj]
      for ni from -1 to 1:
        for nj from -1 to 1:
```

* dakkak@illinois.edu

```
          cq = image[ii+ni][jj+nj]
          lq = label[ii+ni][jj+nj]
          sq = strength[ii+ni][jj+nj]
          gc = g(cp, cq)
          if gc * sq > sp:
            nl = lq
            ns = sq * gc
            changed = true
          end
        end
      end
      nextLabel[ii][jj] = nl
      nextStrength[ii][jj] = ns
    end
  end
  iterations++
  label = nextLabel
  strength = nextStrength
end
```

1.1 Markov Random Field (GraphCut) Based Segmentation

MRF have been used in GPU computing with good success. The problem is that they do not map nicely to the hardware — use irregular data structures, require atomic updates, etc... Furthermore, the method is patented which limits its usefulness in real world applications. While we initially had an MRF implementation, we abandoned it in favor of a much more interesting algorithm that maps nicely to GPUs.

1.2 Optimizations

In this section we evaluate different optimizations on both CPUs and GPUs.

TASK BASED PARALLELIZATION We first parallelized the code to run on different CPU cores. This was done using a task based approach (using Thread Building Blocks [2]) which simplifies some of the programming.

BASE CUDA IMPLEMENTATION Once the threaded multi-core version was working, we ported the code to use CUDA. Each thread in CUDA processes a pixel, access its elements, and updates its state. A global synchronization is required after each iteration of the algorithm since boundary edges are not known after a thread group completes.

CUDA IMPLEMENTATION USING SHARED MEMORY Shared memory are memory residing on each core of the GPU. Placing reused memory there, instead of always accessing it of chip, is a common optimization. In place of explicitly computing the 2-norm, we use the `hypotf` function in this step as an added optimization. We also give compile hints so constant memory are to be placed in a constant memory space (which is a separate part of the hardware dedicated to read-only memory).

APPROXIMATING THE 2-NORM The 2-norm computation is the only complicated arithmetic operation in our code, we therefore substituting it by using the fact that $\|x - y\|_2$ can be approximated with $\alpha Max(x, y) + \beta Min(x, y)$, the choice of α and β depend on the error you can tolerate. We chose $\alpha = 1$ and $\beta = 1/2$ which gives a maximum error of 11.8% and a mean error of 8.86 on average.

(a) True Solution (b) Approximate Solution (c) Error

Figure 1: This shows the effects of approximating the algorithm by not having a global synchronization after each step. We rescale our differences so that even small errors are visable.

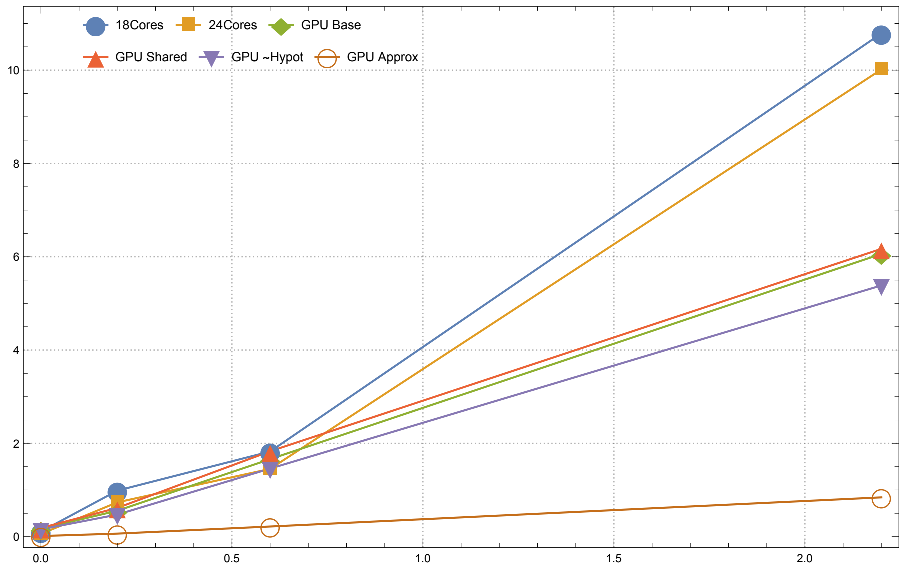

Figure 2: This shows the timing of GrowCut on a 24 and 32 core system. As expected, no difference is noticed between the two configurations due to hitting I/O bandwidth limits. The x axis is the number of megapixels in the image and the y axis is the execution time in seconds. It also shows the effects of different GPU optimizations.

APPROXIMATING THE ALGORITHM A synchronization is required between iterations different iterations of the algorithm. Since CUDA has no global synchronization capabilities, we essentially do a local synchronization (within the block). To not propagate error too much, we perform 32 iterations before doing a global synchronization.

Since the value of boundary pixels is not known within a local synchronization, we make the assumption that boundary pixels "look" like their neighbors or their previous state. If both the past history and the neighbors match, then we keep the value unchanged. Otherwise, we have a 50% probability to be correct and take the value of the neighborhood pixel. Figure 1 shows the effects of this approximation.

2 EVALUATION

In Figure 2 we see that having 24 core (Intel Xeon X5660) vs 32 core (Intel Xeon X5675) does not make a difference for streaming, since one hits the memory and I/O bandwidth limits (and essentially encounters Amdahl's law). More advanced configurations, such as RAID, would allow us to scale better as the number of cores become large.

Figure 2 shows the effects of different GPU optimizations on a C2070 GPU. The GPU base version performs the computation using an 8×8 work group. The GPU shared version places the `label`, `strength`, and `image` data into shared memory before performing the computation. Since error can be tolerated, the GPU ~Hypot version approximates the 2-norm using the "α-max plus β-min algorithm" with $\alpha = 1$ and $\beta = 1/2$. The final version, GPU Approx, introduces more error by performing multiple iterations without doing a global synchronization. This shows that beyond program tuning for hardware, one can exploit the error tolerance for this class of algorithms to achieve great speedup.

3 FUTURE WORK

One key observation in segmentation is that the labeling is decided in the first few iterations of the algorithm. A simple optimization to reduce our error while approximating the algorithm is to start the algorithm with a global synchronization after each iterations (computing the true solution). At some time t we can more safely perform the approximation by having global synchronization after n iterations of the algorithm. At a later time $t + s$, we can perform the synchronization after $2n$ iterations.

REFERENCES

[1] Vladimir Vezhnevets and Vadim Konouchine. Growcut: Interactive multi-label nd image segmentation by cellular automata. In *Proc. of Graphicon*, pages 150–156, 2005.

[2] James Reinders. *Intel threaded building blocks*. O'Reilly, 2007.

4 APPENDIX

Listing 2: Final optimized and approximated code

```

#include <limits.h>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>
#include <fstream>

#include "assert.h"
#include "dataset.h"
#include "mfi.h"
#include "timer.h"

#include "math.h"
#include "cuda.h"
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

using namespace std;

#define RADIUS 1
#define MAX_ITERATIONS 2048 / 32
#define BLOCKDIM_X 8
#define BLOCKDIM_Y 8

template <typename T0, typename T1> static T0 zCeil(const T0 &n, const T1 &d) {
  return (T0) ceil(static_cast<double>(n) / static_cast<double>(d));
}

#define cudaCheck(stmt) _cudaCheck(__LINE__, stmt)
#define _cudaCheck(line, stmt)                                              \
  do {                                                                      \
    cudaError_t p_err = stmt;                                               \
    assert(p_err == cudaSuccess);                                           \
    if (p_err != cudaSuccess) {                                             \
      printf("ERROR on line %d (%s): %s -- %d\n", line, #stmt,              \
             cudaGetErrorString(p_err), p_err);                             \
    }                                                                       \
  } while (0)

#define _max(x, y) (((x) > (y)) ? x : y)
#define _min(x, y) (((x) < (y)) ? x : y)
#define g(x, y) (1 - _max(x, y) - _min(x, y) / 2)

__global__ void growcut(unsigned short const *__restrict__ img, char *label,
                        float *strength, int height, int width) {

  __shared__ unsigned short
      imgShared[BLOCKDIM_Y + 2 * RADIUS][BLOCKDIM_X + 2 * RADIUS];
  __shared__ char lShared[BLOCKDIM_Y + 2 * RADIUS][BLOCKDIM_X + 2 * RADIUS];
  __shared__ float sShared[BLOCKDIM_Y + 2 * RADIUS][BLOCKDIM_X + 2 * RADIUS];
  __shared__ char ltShared[BLOCKDIM_Y][BLOCKDIM_X];
  __shared__ float stShared[BLOCKDIM_Y][BLOCKDIM_X];

  int tidX = threadIdx.x;
  int tidY = threadIdx.y;
  int tx = tidX;
  int ty = tidY;
  int jj = tidX + BLOCKDIM_X * blockIdx.x;
  int ii = tidY + BLOCKDIM_Y * blockIdx.y;

#define idx(arry, y, x)                                                     \
  ((y) >= 0 && (y) < height && (x) >= 0 && (x) < width)                    \
      ? arry[(y) * width + (x)]                                             \
      : 0

  tidY += RADIUS;
  tidX += RADIUS;

  imgShared[tidY][tidX] = idx(img, ii, jj);
  sShared[tidY][tidX] = idx(strength, ii, jj);
  lShared[tidY][tidX] = idx(label, ii, jj);

  if (tx < RADIUS) {
```

```
    imgShared[tidY][tidX - RADIUS] = idx(img, ii, jj - RADIUS);
    sShared[tidY][tidX - RADIUS] = idx(strength, ii, jj - RADIUS);
    lShared[tidY][tidX - RADIUS] = idx(label, ii, jj - RADIUS);

    imgShared[tidY][tidX + BLOCKDIM_X] = idx(img, ii, jj + BLOCKDIM_X);
    sShared[tidY][tidX + BLOCKDIM_X] = idx(strength, ii, jj + BLOCKDIM_X);
    lShared[tidY][tidX + BLOCKDIM_X] = idx(label, ii, jj + BLOCKDIM_X);
  }

  if (ty < RADIUS) {
    imgShared[tidY - RADIUS][tidX] = idx(img, ii - RADIUS, jj);
    sShared[tidY - RADIUS][tidX] = idx(strength, ii - RADIUS, jj);
    lShared[tidY - RADIUS][tidX] = idx(label, ii - RADIUS, jj);

    imgShared[tidY + BLOCKDIM_Y][tidX] = idx(img, ii + BLOCKDIM_Y, jj);
    sShared[tidY + BLOCKDIM_Y][tidX] = idx(strength, ii + BLOCKDIM_Y, jj);
    lShared[tidY + BLOCKDIM_Y][tidX] = idx(label, ii + BLOCKDIM_Y, jj);
  }

  for (int kk = 0; kk < 16; kk++) {
    __syncthreads();
    char nl = 0;
    float ns = 0;
    if (jj < width && ii < height) {
      char lq;
      float sq, gc;
      unsigned short cq;

      unsigned short cp = imgShared[tidY][tidX];
      char lp = lShared[tidY][tidX];
      float sp = sShared[tidY][tidX];
      nl = lp;
      ns = sp;

      cq = imgShared[tidY - 1][tidX];
      lq = lShared[tidY - 1][tidX];
      sq = sShared[tidY - 1][tidX];
      gc = g(cp, cq) * sq;
      if (gc > sp) {
        nl = lq;
        ns = gc;
      }

      cq = imgShared[tidY + 1][tidX];
      lq = lShared[tidY + 1][tidX];
      sq = sShared[tidY + 1][tidX];
      gc = g(cp, cq) * sq;
      if (gc > sp) {
        nl = lq;
        ns = gc;
      }

      cq = imgShared[tidY][tidX - 1];
      lq = lShared[tidY][tidX - 1];
      sq = sShared[tidY][tidX - 1];
      gc = g(cp, cq) * sq;
      if (gc > sp) {
        nl = lq;
        ns = gc;
      }

      cq = imgShared[tidY][tidX + 1];
      lq = lShared[tidY][tidX + 1];
      sq = sShared[tidY][tidX + 1];
      gc = g(cp, cq) * sq;
      if (gc > sp) {
        nl = lq;
        ns = gc;
      }

    }

    ltShared[ty][tx] = nl;
    stShared[ty][tx] = ns;

    __syncthreads();

    lShared[tidY][tidX] = ltShared[ty][tx];
    sShared[tidY][tidX] = stShared[ty][tx];

```

```
    if (tx < RADIUS) {
      sShared[tidY][tidX - RADIUS] =
          (sShared[tidY][tidX - RADIUS] + stShared[ty][tx]) / 2;
      lShared[tidY][tidX - RADIUS] =
          (lShared[tidY][tidX - RADIUS] + ltShared[ty][tx]) / 2;

      sShared[tidY][tidX + BLOCKDIM_X] =
          (sShared[tidY][tidX + BLOCKDIM_X] +
           stShared[ty][BLOCKDIM_X - tx - RADIUS]) / 2;
      lShared[tidY][tidX + BLOCKDIM_X] =
          (lShared[tidY][tidX + BLOCKDIM_X] +
           ltShared[ty][BLOCKDIM_X - tx - RADIUS]) / 2;
    }

    if (ty < RADIUS) {
      sShared[tidY - RADIUS][tidX] =
          (sShared[tidY - RADIUS][tidX] + stShared[ty][tx]) / 2;
      lShared[tidY - RADIUS][tidX] =
          (lShared[tidY - RADIUS][tidX] + ltShared[ty][tx]) / 2;

      sShared[tidY + BLOCKDIM_Y][tidX] =
          (sShared[tidY + BLOCKDIM_Y][tidX] +
           stShared[BLOCKDIM_Y - ty - RADIUS][tx]) / 2;
      lShared[tidY + BLOCKDIM_Y][tidX] =
          (lShared[tidY + BLOCKDIM_Y][tidX] +
           ltShared[BLOCKDIM_Y - ty - RADIUS][tx]) / 2;
    }

  }
  if (jj < width && ii < height) {
    label[ii * width + jj] = ltShared[ty][tx];
    strength[ii * width + jj] = stShared[ty][tx];
  }
  return;
}

int runGrowCut(MFI *mfi, char *label, int *iterations0) {
  int iterations = 0;
  int width = mfi->width;
  int height = mfi->height;
  char *nextLabel = (char *)malloc(sizeof(char) * width * height);
  float *strength = (float *)malloc(sizeof(float) * width * height);
  float *nextStrength = (float *)malloc(sizeof(float) * width * height);
  unsigned short *cap_source = (unsigned short *)mfi->cap_source;
  unsigned short *cap_sink = (unsigned short *)mfi->cap_sink;
  int len = width * height;
  for (int ii = 0; ii < len; ii++) {
    float s = label[ii] != 0;
    strength[ii] = s;
  }

  unsigned short *dcapsource;
  float *dStrength;
  char *dLabel;

  cudaCheck(cudaMalloc(&dcapsource, sizeof(unsigned short) * len));
  cudaCheck(cudaMalloc(&dStrength, sizeof(float) * len));
  cudaCheck(cudaMalloc(&dLabel, sizeof(char) * len));

  cudaCheck(cudaMemcpy(dcapsource, mfi->cap_source,
                       sizeof(unsigned short) * len, cudaMemcpyHostToDevice));
  cudaCheck(
      cudaMemcpy(dLabel, label, sizeof(char) * len, cudaMemcpyHostToDevice));
  cudaCheck(cudaMemcpy(dStrength, strength, sizeof(float) * len,
                       cudaMemcpyHostToDevice));

  dim3 blockDim(BLOCKDIM_X, BLOCKDIM_Y);
  dim3 gridDim(zCeil(width, blockDim.x), zCeil(height, blockDim.x));

  while (iterations++ < MAX_ITERATIONS) {
    growcut << <gridDim, blockDim>>>
        (dcapsource, dLabel, dStrength, height, width);
    cudaCheck(cudaThreadSynchronize());
  }

  cudaCheck(
      cudaMemcpy(label, dLabel, sizeof(char) * len, cudaMemcpyDeviceToHost));

  cudaFree(dcapsource);
  cudaFree(dStrength);
```

```
  cudaFree(dLabel);

  free(nextLabel);
  free(strength);
  free(nextStrength);

  *iterations0 = iterations;
  return -1;
}

int main(int argc, char **argv) {

  const char *dataset_path =
      argc == 2 ? argv[1] : "C:\\Users\\abduld\\Documents\\visual␣studio␣"
                            "2012\\Projects\\growcut\\x64\\Debug\\dataset";

  int num_instances = (sizeof(instances) / sizeof(Instance));

  ofstream timesFile;
  string timeFileName = string(dataset_path);
  timeFileName.append("\\..\\times_cuda_opt_c2070_alpha_max_itermore.data");
  timesFile.open(timeFileName, ios::out);
  timesFile << "instance,num,width,height,changes,iterations,time\n";

  for (int i = 0; i < num_instances; i++) {

    for (int j = 0; j < instances[i].count; j++) {
      char fileName[1024];

      ofstream myfile;
      string sfileName;
      int iterations;
      unsigned short *cap_source;
      unsigned short *cap_sink;
      char *label;
      int changes;
      uint64_t tic, toc;
      double compute_time;

      sprintf(fileName, instances[i].filename, dataset_path, j);
      MFI *mfi = mfi_read(fileName);

      if (!mfi) {
        //printf("FAILED to read instance %s\n",fileName);
        goto skip;
      }

      if (mfi->connectivity != 4 || mfi->dimension != 2 ||
          mfi->type_terminal_cap != MFI::TYPE_UINT16 ||
          mfi->type_neighbor_cap != MFI::TYPE_UINT16) {
        goto skip;
      }

      cap_source = (unsigned short *)mfi->cap_source;
      cap_sink = (unsigned short *)mfi->cap_sink;
      label = (char *)calloc(mfi->width * mfi->height, sizeof(char));
      for (int ii = 0; ii < mfi->width * mfi->height; ii++) {
        if (cap_source[ii] > 15000) {
          label[ii] = 1;
        }
        if (cap_sink[ii] > 15000) {
          label[ii] = -1;
        }
      }

      tic = _hrtime();
      changes = runGrowCut(mfi, label, &iterations);
      toc = _hrtime();

      compute_time = (toc - tic) / 1000000000.0f;

      sfileName = string(fileName);
      sfileName.append(".dat");

      myfile.open(sfileName, ios::out);

      for (int ii = 0; ii < mfi->height; ii++) {
        for (int jj = 0; jj < mfi->width; jj++) {
          myfile << ((int) label[ii * mfi->width + jj] + 2) / 2 << "␣";
          //myfile << ((int) cap_sink[ii*mfi->width + jj]) << " ";
```

```
        }
        myfile << "\n";
      }
      myfile.close();

      printf("%s,%d,%d,%d,%d,%d,%0.9f␣\n", instances[i].name, j, mfi->width,
             mfi->height, changes, iterations, compute_time);
      timesFile << instances[i].name << "," << j << "," << mfi->width << ","
                << mfi->height << "," << changes << "," << iterations << ","
                << compute_time << "\n";
      timesFile.flush();
      free(label);
    skip:
      if (mfi != NULL)
        mfi_free(mfi);

    }
  }
  timesFile.close();
  return 0;
}
```