Simon Vergara
Vergars
001146596

4F03 Assignment 3

The following table includes the information for the following parameters:
Population Size: 100
Number of Generations = 1000

./genimg imagein.ppm imageout.ppm 1000 100
./genimg_omp num_threads imagein.ppm imageout.ppm 1000 100
./genimg_acc imagein.ppm imageout.ppm 1000 100

Server Name	Number of Cores	CPU Time	Speedup
	1	138s	--
	2	35s	3.94
	4	25s	5.52
	8	18s	7.67
Quadro FX4800		20m+	-∞
Quadro K2000		20m+	-∞
Quadro K5000		20m+	-∞
K40		20m+	-∞

(Speedups above are relative to running OMP with 1 core)

OMP Discussion

We can see that this genetic algorithm seems to be embarrassingly parallelizable. What this means is that when we use multiple cores, we will achieve a speedup close to the number of cores we use. Above, we can actually see that for 2 and 4 cores, we can achieve superlinear speedups. Speedup is larger than the number of cores.

OpenACC Discussion

For OpenAcc, I was not able to achieve any speedups. I will explain my process below:
First, I have a deep copyin of the population and all their images/fitness. Next, I generate an array of random integers. The number of integers depends on the problem size. It's calculated so that there will be as many random integers as are needed in the algorithm. I first tried to implement curand in order to generate random numbers from the GPU, but I was unable to do so.
Next, the generations loop begins. This loop is inside the GPU, as it has a #pragma acc kernels above it. I copyin any data that I may need from the GPU.
Since in OpenACC, function calls aren't permitted, I moved all the functions as if they were inline'd.
Starting with the mate loop, I parallelized it on the GPU by using a #pragma acc parallel gang independent. There aren't any loop dependencies here, so it's safe to parallelize.

The mutate loop follows. This loop uses the random numbers that were pregenerated and copied to the GPU.
Next is the fitness loop. This loop has an inner loop which can be parallelized and reduced. This is done as a worker.
After the fitness loop, we have to sort the population by fitness level. Since we can't use quicksort in the generations loop, I wrote a sorting algorithm.
After the generations loop has finished running all these other loops, the GPU section is finished. Now, we have to perform a copyout and updateself in order to update all the information that was changed inside the GPU.

Something in my code caused it to run extremely slowly. Due to this, I created a second chart below with numbers that are reasonable for my implementation.

Population Size: 8
Number of Generations = 100

Running these numbers without OpenMP or OpenACC yielded a runtime of 0.57s

Server Name (OpenACC)	Number of Cores (OpenMP)	CPU Time	Speedup
	1	19.5s	--
	2	4.7s	4.14
	4	4.7s	4.14
	8	4.9s	3.98
Quadro FX4800		--	--
Quadro K2000		12s	1.65
Quadro K5000		16s	1.21
K40		400s	0.048

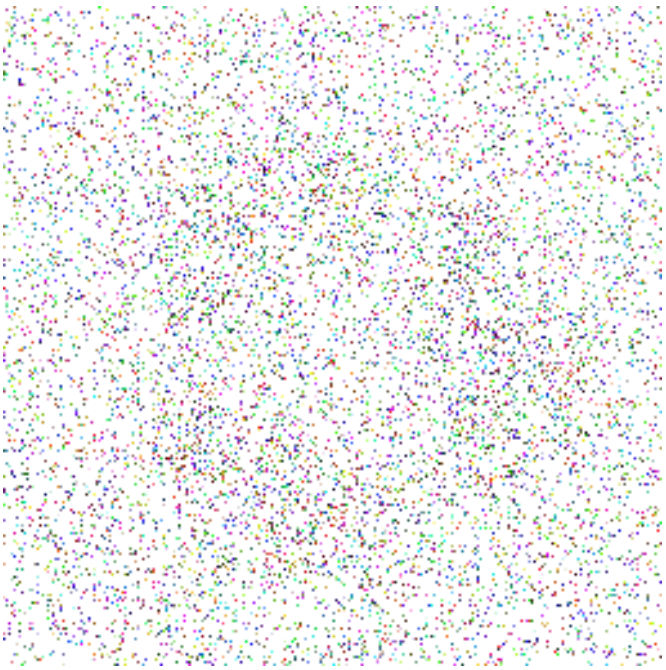
(Speedups above are relative to running OMP with 1 core)

Readings for K40 above are inaccurate, as the server was very congested at the time it was reported.

Different timings across GPUs can be explained by the GPUs' specs. For example, the K40 has twice as much global memory and three times as much cache size as the K5000. All of this can be found by looking at the pgaccelinfo for each GPU. In general, the K40 has the best specifications of all of the GPUs available. This would lead to a much better performance.

The best performance I achieved with OpenACC was the image below. Followed by what it's supposed to look like.

Simon Vergara
Vergars
001146596



Even though it’s not good, you can see that the image started to converge to what it’s supposed to look like.

Simon Vergara
Vergars
001146596

BONUS

As part of the bonus, I created an animated gif and included it in the folder called “images”

Simon Vergara
Vergars
001146596

compimage.c

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>

#include "a3.h"

static int fitnessCompare (const void *a, const void *b)
{
return ((*(Individual*)a).fitness - (*(Individual*)b).fitness);
}



void compImage( RGB *desired_image, int no_threads, int width, int
height, int max,
int num_generations, int population_size,
RGB *found_image, const char *output_file)
{

    int i;
    // Allocate an array/population of individuals.
    Individual *population = (Individual*)
    malloc(population_size * sizeof(Individual));
    assert(population);


    int imgnum = 0;
    int g;
    double prev_fitness, current_fitness;


    /* A. Create an initial population with random images.
    ****************************************************
    */


    // Initialize this population with random images

    #pragma omp parallel for num_threads(no_threads)
    for (i = 0; i < population_size; i++)
      population[i].image = randomImage(width, height, max);

    // Compute the fitness for each individual
    #pragma omp parallel for num_threads(no_threads)
```

```
    for (i = 0; i < population_size; i++)
      compFitness(desired_image, population+i, width, height);

    // Sort the individuals/images in non-decreasing value of fitness
    qsort(population, population_size, sizeof(Individual),
fitnessCompare);

    /* B. Now we can evolve the population over num_generations.
    ************************************************************
    */
    int mutation_start =  population_size/4;

    //#pragma omp parallel num_threads(8) private(g)
shared(population)



    for (g = 0; g < num_generations; g++)
    {
      prev_fitness = population[0].fitness;

      // The first half mate and replace the second half with children.
      #pragma omp parallel for
      for (i = 0; i < population_size/2; i += 2)
      {
            mate(population+i, population+i+1,
            population+population_size/2+i,
            population+population_size/2+i+1,
            width, height);
      }

      // Afterer the first 1/4 individuals, each individual can
      // mutate.
      for (i = mutation_start; i < population_size; i++)
      {
            mutate(population+i, width, height, max);
      }


      //  Recompute fitness
      #pragma omp parallel for num_threads(no_threads)
      for (i = 0; i < population_size; i++)
      {
            compFitness(desired_image, population+i, width, height);
      }

      // Sort in non-decreasing fitness
      qsort(population, population_size, sizeof(Individual),
```

```
fitnessCompare);

      current_fitness = population[0].fitness;

      double change = -(current_fitness-prev_fitness)/current_fitness*
100;

      #ifdef MONITOR
      // If compiled with flag -DMONITOR, update the output file every
      // 300 iterations and the fitness of the closest image.
      // This is useful for monitoring progress.

      char filen[20];

      if ( g % 300 == 0){
            sprintf(filen,"%s_%d","output_file",imgnum);
            writePPM(filen, width, height, max, population[0].image);
            imgnum ++;
      }

      printf(" generation % 5d fitness %e  change from prev %.2e%c \n",
      g, current_fitness, change, 37);
      #endif

    }

    // Return the image that is found
    memmove(found_image, population[0].image,
width*height*sizeof(RGB));

    // release memory
    for (i = 0; i < population_size; i++){
      free(population[i].image);
    }
    free(population);
}
```

Simon Vergara
Vergars
001146596

fitness.c

```
#include <math.h>
#include "a3.h"

    inline int sqr(int x)
    {
      return x*x;
    }


    inline double pixelDistance (const RGB *a, const RGB *b)
    {
      const RGB *locala = a;
      const RGB *localb = b;

      double rd,gd,bd;

      rd = locala->r - localb->r;
      gd = locala->g - localb->g;
      bd = locala->b - localb->b;

      return (sqr(rd) + sqr(gd) + sqr(bd));
    }

    void compFitness (const RGB *A, Individual *B, int width, int
height)
    {
      int i;
      double f = 0;

      const RGB *localA;
      Individual *localB;

      localA = A;
      localB = B;

      int end = width * height;

      #pragma omp parallel for reduction(+:f)
      for (i = 0; i < end; i++) {
            f += pixelDistance(&localA[i], &(localB->image[i]));
            i++;
            f += pixelDistance(&localA[i], &(localB->image[i]));
            i++;

            f += pixelDistance(&localA[i], &(localB->image[i]));
            i++;
```

```
        f += pixelDistance(&localA[i], &(localB->image[i]));

    }

    B->fitness = f;
}
```

Simon Vergara
Vergars
001146596

compimage_acc.c

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include "a3.h"

#include <curand.h>


static int fitnessCompare (const void *a, const void *b)
{
  return ((*(Individual*)a).fitness - (*(Individual*)b).fitness);
}



void compImage( RGB *desired_image, int num_threads, int width, int
height, int max,
int num_generations, const int population_size,
RGB *found_image, const char *output_file)
{

  int i;
  // Allocate an array/population of individuals.
  Individual *population = (Individual*)
  malloc(population_size*sizeof(Individual));
  assert(population);


  int imgnum = 0;
  int g;
  double prev_fitness, current_fitness;

  int mutation_start =  population_size/4;
  int imgsz = width * height ;


  /* A. Create an initial population with random images.
  ******************************************************
  */


  // Initialize this population with random images
  for (i = 0; i < population_size; i++)
    population[i].image = randomImage(width, height, max);
```

```
  // Compute the fitness for each individual
  for (i = 0; i < population_size; i++)
  {
    double fit;
    compFitness(desired_image, population+i, width, height);
  }

  // Sort the individuals/images in non-decreasing value of fitness
  qsort(population, population_size, sizeof(Individual),
fitnessCompare);


  /* Copy in the population to the GPU. Access from the GPU  as
gpupop*/
  Individual* gpupop;
  gpupop = (Individual*)acc_pcopyin( population, sizeof(Individual) *
population_size );

  RGB* dA;

  //perform the deep copyin
  for ( i=0; i < population_size; i++ ) {
    dA = (RGB *)acc_pcopyin( population[i].image, imgsz *
sizeof(RGB));     //device address in dA
    acc_memcpy_to_device( &gpupop[i].image, &dA, sizeof(RGB*));
  }




  /*PREGENERATE RADOM NUMBERS!*/
  long int numrand = (num_generations)*((4 *
(imgsz/500))*(population_size - population_size/4) +
population_size/2);
  int *myrandom = (int *)malloc(numrand * sizeof(int));
  int r;

  //create seed for random numbers
  srand ( time(NULL) );

  //create array of random numbers
  for (r=0;r<numrand;r++){
    myrandom[r] = rand();
  }

  // printf("random[%ld] = %d\n",0,myrandom[0]);
```

```
  // printf("random[%ld] = %d\n",1,myrandom[1]);
  // printf("random[%ld] = %d\n",2,myrandom[2]);



  /*
  unsigned int* rand_buffer = NULL;
  cudaMalloc((void **) &rand_buffer, 1*sizeof(unsigned int));
  curandGenerator_t gen;
  curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
  curandSetPseudoRandomGeneratorSeed(gen, 1234ULL);
  curandGenerate(gen, rand_buffer, 1);
  printf("sdsdsdsdsdssdsfsdfjhsfkjshfjkshfkjsdhfkjsdhfsdkjfh\n");
  printf("sssss%u\n", *(rand_buffer));



  printf("making rng\n");
  unsigned int  ll ;

  ll = randPlz();
  printf("generated num\n");
  printf("random=%u\n", (ll));
  */


  writePPM("out_before.ppm", width, height, max, population[0].image);

  int rptr=0;

  /* B. Now we can evolve the population over num_generations.
  ************************************************************
  */
  //#pragma acc data pcopyin(desired_image[0:imgsz], width, height,
population_size)
  #pragma acc kernels pcopyin(desired_image[0:imgsz], population_size,
imgsz, mutation_start,numrand, myrandom[0:numrand]) copy(rptr)
  for (g = 0; g < num_generations; g++)
  {
    prev_fitness = gpupop->fitness;


    // The first half mate and replace the second half with children.*
    #pragma acc loop gang independent
    for (i = 0; i < population_size/2; i += 2)
    {
      // mate(population+i, population+i+1,
      //   population+population_size/2+i,
```

```
      //   population+population_size/2+i+1,
      //   width, height);

      int crossover = myrandom[rptr] % (imgsz);//RANDOM(width*height-
1);
      rptr ++; rptr = (rptr % (numrand-1));

      int j;

      Individual * parent1 = gpupop+i;
      Individual * parent2 = gpupop+i+1;
      Individual * child1 = gpupop+population_size/2+i;
      Individual * child2 = gpupop+population_size/2+i+1;

      for (j = 0; j < crossover; j++)
      {
        child1->image[j] = parent1->image[j];
        child2->image[j] = parent2->image[j];
      }
      for (j = crossover; j < imgsz; j++)
      {
        child1->image[j] = parent2->image[j];
        child2->image[j] = parent1->image[j];
      }

    }


    // Afterer the first 1/4 individuals, each individual can
    // mutate.
    for (i = mutation_start; i < population_size; i++)
    {
      // mutate(population+i, width, height, max);

      // Set how many pixels to mutate. The constant 500 is somewhat
      // random. You can experiment with different constants.
      int rate = imgsz/500;

      RGB *img = (gpupop+i)->image;
      int k,j;

      for(k=0; k < rate; k++)
      {
        // Pick a pixel at random.
        j = myrandom[rptr] % (imgsz);
        rptr ++; rptr = (rptr % (numrand-1));

        // and modify it
```

```
    RGB * mimg = img + j;
    mimg->r = myrandom[rptr] % (max+1);
    rptr ++; rptr = (rptr % (numrand-1));

    mimg->g = myrandom[rptr] % (max+1);
    rptr ++; rptr = (rptr % (numrand-1));

    mimg->b = myrandom[rptr] % (max+1);
    rptr ++; rptr = (rptr % (numrand-1));
  }
}
```

```
/* Just checking how to copy data in.
#pragma acc kernels
{
int p;
int m;
//#pragma acc kernels loop
for (p=0;p<population_size;p++){
Individual *mypop = gpupop + p;

for (m=0;m<imgsz;m++){
RGB *myimg = mypop->image+m;

myimg->r=255;
myimg->g=255;
myimg->b=255;
}

}
}*/


// recompute the fitness for each individual
for (i = 0; i < population_size; i++) {

  int j;
  double f = 0;

  Individual *mypop;
  mypop = gpupop+i;
```

```
      #pragma acc loop worker reduction(+:f)
      for (j = 0; j < imgsz; j++)
      {
        RGB *popimg = mypop->image;
        popimg = popimg + j;

        double rd,gd,bd;

        RGB * desimg = desired_image + j;

        rd = desimg->r - popimg->r;
        gd = desimg->g - popimg->g;
        bd = desimg->b - popimg->b;

        f+= rd*rd + gd*gd + bd*bd;
      }

      mypop->fitness = f;
    }


    // Sort in non-decreasing fitness
    //qsort(population, population_size, sizeof(Individual),
fitnessCompare);
    Individual temp;
    int j;
    for(i=1; i< population_size; i++)
    {
      for(j = 0; j<(population_size)-i; j++)
      {
        if((gpupop+j)->fitness > (gpupop+j+1)->fitness)
        {
          temp = *(gpupop+j);
          *(gpupop+j) = *(gpupop+j+1);
          *(gpupop+j+1) = temp;
        }

      }

    }




    current_fitness = gpupop->fitness;

    //    double change = -(current_fitness-
```

```
prev_fitness)/current_fitness* 100;

    /*#ifdef MONITOR
    // If compiled with flag -DMONITOR, update the output file every
    // 300 iterations and the fitness of the closest image.
    // This is useful for monitoring progress.

    char filen[20];

    if ( g % 300 == 0){
      sprintf(filen,"%s_%d","output_file",imgnum);
      writePPM(filen, width, height, max, population[0].image);
      imgnum ++;
    }

    printf(" generation % 5d fitness %e  change from prev %.2e%c \n",
    g, current_fitness, change, 37);
    #endif*/
  }

  /*get stuff back from the GPU*/
  //perform a copyout from the kernel
  for (i=0; i < population_size; i++ ) {
    acc_update_self(&population[i].fitness, sizeof(double));
    acc_copyout(population[i].image, imgsz * sizeof(RGB));
  }

  printf("If you're here, you are a \nW\n I\n  N\n   N\n    E\n
R\n");

  // Return the image that is found
  memmove(found_image, population[0].image, width*height*sizeof(RGB));

  // release memory
  for (i = 0; i < population_size; i++){
    free(population[i].image);
  }
  free(population);
}




/*unsigned int myrand(){
unsigned int* rand_buffer = NULL;
cudaMalloc((void **) &rand_buffer, 1*sizeof(unsigned int));
curandGenerator_t gen;
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
```

```
curandSetPseudoRandomGeneratorSeed(gen, 1234ULL);
curandGenerate(gen, rand_buffer, 1);

return rand_buffer;
}*/
```