Colton Wedell
CPSC 445-01
Final Project

12/15/2021

Introduction

This report discusses a program to calculate the area of a shape defined by an implicit function. To do this, it approximates the calculus method of differentiation, by taking small slices in y across the entire width of the shape, calculating the area of each, and then summing to arrive at the total area. This program can calculate the area of any circle or ellipse defined on the cartesian plane by an implicit function.

Description

This program utilizes multithreading in CUDA to achieve performance – specifically, the `solver()` function is run on each CUDA device to calculate the area of an individual slice. For description, however, let's begin with the `main()` function. In this description, we will assume a sample implicit function of $2x^2 + 3y^2 - 5 = 0$.

First, a number of variables are declared. The `dim` variable is first, representing the width of each slice. Currently, it is set to 0.001, but it can be changed to increase or decrease precision. Next, the x and y coefficients are declared, along with the constant. We know that the degree will always be 2, since higher-order implicit functions do not produce closed, differentiable shapes. Each of these variables is declared as a double. The `area` and `xstart` variables, also doubles, represent a placeholder for the total area, and the leftmost differentiable x value, respectively. `xstart`, in particular, is defined as the negative square root of the absolute value of the constant divided by the x-coefficient, representing a solution where $y = 0$. In the example function, the `xstart` variable is equivalent to the negative solution of $2x^2 - 5 = 0$, or $x = -\sqrt{\frac{5}{2}}$. The final variable declaration is the integer `width`, which represents the total number of slices required, defined as `floor(abs(xstart)*2/dim)`. For our example, the width comes out to be 3162.

Following this, two arrays of pointers to doubles are created, `placeholder` and `outputs`. `placeholder` will remain on the host, and `outputs` will be passed to each device. Memory is allocated for `placeholder` at one index per slice. Next, CUDA memory is allocated for `outputs`, and copied to each device. Then, the `solver()` function is run, with 1024 threads per block and the correct number of blocks to fit each slice (the ceiling of the number of slices divided by 1024). In our case, this means 4 blocks of 1024 threads each, for 4096 total threads, of which 934 will remain unused.

The `solver()` function behaves as follows. It takes the following arguments:

```
__global__ void solver(double* outputs, double dim, double xcoeff, double ycoeff,
double constant, double xstart) { … }
```

After calculating its rank and its starting x – defined by which slice it has been assigned – it calculates the area of its slice. It does this by solving for y in the following manner, using the example function:

Colton Wedell 12/15/2021
CPSC 445-01
Final Project

$$2x^2 + 3y^2 - 5 = 0$$
$$2x^2 + 3y^2 = 5$$
$$3y^2 = 5 - 2x^2$$
$$y^2 = \frac{5 - 2x^2}{3}$$
$$y = \sqrt{\frac{5 - 2x^2}{3}}$$

This final equation is equivalent to the assignment to the `y1` variable, with the addition of the absolute value function to ensure the input to the square root is positive:

```
double y1 = sqrt(abs((-constant - xcoeff*pow(x,2))/ycoeff));
```

`y2` is simply the negation of `y1`, and the area is `dim*(y1-y2)`. This area is then written to the `outputs` array at the index `myrank`.

Returning to the `main()` function, we can now calculate the total area. The `outputs` array is copied back to `placeholder`, and then iterated over to determine the final sum. This sum is printed to the console, with the original implicit function for reference, memory is freed, and the program ends.

Parallelization

This program is parallelized across threads and blocks, with a slice of the area calculation being assigned to each thread. The computation required will grow linearly as the size of the shape grows, because the dimension is always a fixed size.

Limitations

This program can only handle functions defined in the second degree, i.e. no exponent larger than 2 for x or y. This is because an implicit function in a higher order will define something that is no longer a closed shape – a hyperbola, perhaps, or some other complex conic section. To calculate an area, we must have a closed shape to differentiate over, necessitating a second-degree function which can only create circles or ellipses.

This program does not currently accept input as arguments or through the terminal, but could easily be rewritten to do so. It currently works with example input.

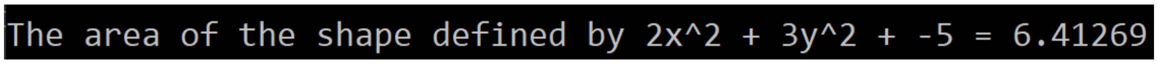
Functionality

To exhibit the functionality of this program, let's run it with our original example equation, $2x^2 + 3y^2 - 5 = 0$. We can thus define our variables:

```
double xcoeff = 2;
double ycoeff = 3;
double constant = -5;
```

Our output can be seen here:

Colton Wedell 12/15/2021
CPSC 445-01
Final Project



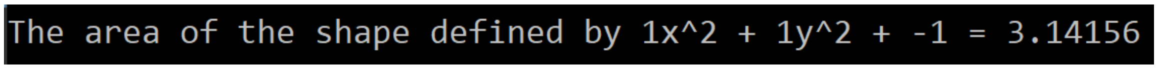
```
The area of the shape defined by 2x^2 + 3y^2 + -5 = 6.41269
```

Using calculus, we would expect an area of $\frac{5\pi}{\sqrt{6}} \approx 6.41275$. Our program is less than the true area by approximately 0.00006.

For another example, take the unit circle, $x^2 + y^2 - 1 = 0$, with variables as follows:

```
double xcoeff = 1;
double ycoeff = 1;
double constant = -1;
```

We would expect an area equal to $\pi \approx 3.14159\ldots$, this being the unit circle. Our program outputs the following:



```
The area of the shape defined by 1x^2 + 1y^2 + -1 = 3.14156
```

This is less than the true area by about 0.00003.

Improvement

Our error stems from the fact that we must create the `width` variable as an integer, since we cannot have fractional threads. The true width of the shape will never divide exactly into `dim`, so we must use `floor()` or `ceil()` to adjust it. This program uses `floor()`, which underestimates, but `ceil()` would similarly overestimate. We could improve this error by decreasing the value of `dim`; however, this comes with a performance penalty, requiring more threads and blocks. At a certain point, no CUDA device would have the capacity to handle more slices – and, in fact, we could never achieve the exact area by this method, since calculus by definition requires infinite slices of infinitesimal width.