November 7, 2017

DVA336 Parallel Systems

Assignment 3

GPU Programming using CUDA

In this assignment, you will gain some experience in GPU programming. Modern GPUs are very powerful and highly parallel compute engines that may provide substantial speedups when it comes to numerical computations in e.g. computer graphics, visualization, artificial intelligence, and various computational sciences. In particular, computations with a high degree of data-parallelism are good candidates for GPU acceleration.

To get access to the computational power of the GPU we will use CUDA, which is a parallel computing platform and programming model. You need a GPU from Nvidia to be able to run CUDA programs. See the CUDA QUICK START GUIDE for information on how to install the required libraries and get started with CUDA programming projects. The programming problems you need to solve as parts of this assignment are given below. You can work either in pairs or individually.

3.1 The saxpy kernel

In the standard Basic Linear Algebra Subroutines (BLAS) library, there is a function called saxpy (which stands for single-precision a x plus y). As the name indicate, this function computes ax + y, where x and y are vectors with n elements each, and a is a scalar. Note that this computation is embarrassingly parallel, since each element of the result vector (the output) can be calculated independently.

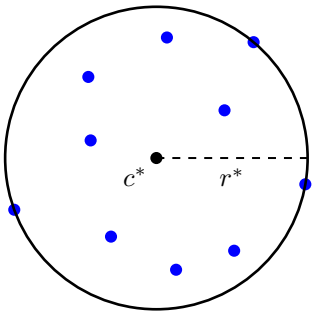
A. An example program that includes a CUDA version of this function is available in the files `saxpy.h`, `saxpy.cpp`, and `saxpy.cu`. Start by studying this program in detail. It is important that you understand all parts of it. Then run some tests with different input sizes. What is the actual execution time of the kernel? What is the data transfer time? How many times faster is the CUDA version compared to the provided sequential CPU alternative?

B. Vary the kernel launch parameters, and rerun the program. Clearly, these parameters can have a significant effect on the performance. Set up an experiment to study the performance behaviour more carefully. Finally, create a table that gives a nice presentation of your results, and add it to your lab report.

C. When execution times are reported, it is of course interesting to know what kind of GPU device that was used. Extend the program so that it also outputs the most relevant properties of the device. Note that there are built-in functions in the CUDA library that you can use to make queries about specific properties. Check the CUDA API for more details. List the most relevant device properties in your lab report.

3.2 Finding minimum enclosing balls

The minimum enclosing ball (MEB) problem is about finding the ball of minimum radius r^* that cover all points in a given point set $P = \{p_0, p_1, \ldots, p_{n-1}\} \subset \mathbb{R}^d$. This is an optimization problem, where we search for the "perfect" center $c^* \in \mathbb{R}^d$: the one that minimize the distance to its farthest point in P. The figure below gives an example of point set (shown in blue) and the smallest possible circle that encloses them.



Instead of solving the MEB problem exactly, we can settle for high quality approximations. A simple $(1+\epsilon)$-approximation algorithm has been proposed by Bădoiu-Clarkson. It has been shown that after $\lceil 1/\epsilon^2 \rceil$ iterations, it returns a center c such that the radius $R := \max \|c - p_i\|$ satisfies

$$r^* \leq R \leq (1+\epsilon)r^*,$$

where r^* is the radius of the exact solution to the MEB problem. To see how the algorithm works, study the pseudocode below.

```
1: procedure MinBall(P, ε > 0)
2:   c ← p_0                          ▷ any p_i ∈ P can be used to init c
3:   for j = 1 to ⌈1/ε²⌉ do
4:     q ← FarthestPnt(c, P)
5:     c ← c + (q − c)/(j + 1)
6:   end for
7:   return c, R ← max ‖p_i − c‖
8: end procedure
```

First, an initial center is selected (Line 2). Then, the algorithm enters the main loop (Lines 3–6). In each iteration, the point q that maximizes the distance to the current center c is found by calling the subroutine FarthestPnt (Line

4). Then the center point is updated by moving it in the direction $q - c$ (Line 5). How far it is moved depends on the iteration counter j. When the main loop finishes after $\lceil 1/\epsilon^2 \rceil$ iterations, the center point is accurate enough. Before the algorithm ends, the radius R, which is simply given by the maximum distance to a point in P, is found as well (Line 7). Clearly, the overall time complexity of this algorithm is $O(\frac{dn}{\epsilon^2})$.

A. Start by implementing a sequential version of the presented algorithm using standard C/C++. Although the given algorithm works in any dimension, it is here enough that you solve the problem specifically for the two-dimensional case. Let the input points be represented by a flat array with the points in row major order, i.e., the coordinates are stored as $x_0 y_0 x_1 y_1 x_2 y_2 \ldots$ in the array. Create suitable test cases and make sure that your C/C++ version of the algorithm works in all cases.

B. Clearly, the bottleneck in the algorithm is the execution of the subroutine that is called on Line 4, since this part of the algorithm has time complexity $O(dn)$ and it is called in every iteration. Fortunately, there is plenty of data-parallelism available in this computation. To exploit this, it seems appropriate to move this part of the algorithm to an accelerator chip, such as a GPU. Then the CPU acts as the host that uses a GPU device to execute this subroutine. Implement a parallel version of your sequential algorithm that follows this strategy by using CUDA.

   When everything works fine, set up an experiment to study the performance of your solutions. Consider at least three values of ϵ, and test several input sizes, say $n = 10^4, 10^5, 10^6, 10^7, \ldots$, for each one of them. How much faster is your CUDA version? How does the data transfer time influence the speedups you get? Before you give your final answers, remember to carefully examine what kernel launch parameters that give the best performance on your particular GPU system. Describe the details of your experimental setup and your experimental results in your lab report.

Examination

Send in your source code with your solutions together with a lab report that contains your experimental results and your answers to the questions above in a compressed archive file by e-mail to the lab assistant. Use either the zip or gzip archive format. Furthermore, be prepared to answer—individually—any questions that may arise concerning your solutions.