Parallel and SIMD Programming – Assignment #4 (SIMD Point Classification)

Due on 09/17/2008

Create SIMD optimized to classify points in a 2D coordinate space. The program will consist of the following routines:

- `generate_points` - Routine to generate a random set of points. This routine is passed *at least* the following parameters:
  - `count` - Number of points to generate
  - `seed` - Random number generator seed
  - `output` - Data location to store the point data
  - `min_x` and `max_x` - Valid range of values in the X dimension
  - `min_y` and `max_y` - Valid range of values in the Y dimension

  Two versions of this routine will ultimately need to be created. One will be called `generate_points_AoS` and will generate point data in an array-of-structures format. The other will be called `generate_points_SoA` and will generate points in a structure-of-arrays format. The SoA version should take separate pointers to the X and Y data locations.

- `find_extreme_points` - Find the minimal and maximal points along the X and Y axes. At most one point should be selected for each axis. It is possible for there to be either 3 or 4 extreme points. If one of the extreme points is at a "corner", it will be the extreme point for 2 axes. This routine should return the number of extreme points.

  The extreme points should be stored in an output array. They can be stored either as indexes to the original data or as "raw" vertex data. Either SoA or AoS format can be used. Whichever choice is made, provide documentation *in comments in the code* defending your choice. Points should be stored in clockwise order (i.e., maximal Y, maximal X, minimal Y, then minimal X).

- `classify_points` - The N extreme points will define N planes. These planes define a polygon. Points are either inside the polygon or outside one of the planes (the proof that each point can be outside only one plane is left as an exercise for the reader). `classify_points` will process each non-extreme point and classify it as either inside or outside a particular plane. This routine should take the original data, the extreme points, and the number of extreme points as inputs. It should output 4 (or 3) new sets of data. Each set represents data outside a particular extreme point plane. This data should be formatted in a way suitable to be passed back to `find_extreme_points`.

You will need to come up with a way to *test* these routines. I suggest creating a scalar, AoS version of `find_extreme_points` and `classify_points`. These should be fairly trivial to test with simple input sets (i.e., generate 5 points and print all the data out at the end). Once you are satisfied that the scalar versions work, convert the routines, one at a time, to SIMD. The initial SIMD versions will likely read AoS data in and convert it, using shuffle instructions, to SoA. The final step is to convert everything to AoS.

Keep the scalar, AoS code around.

Once everything is fully vectorized done, compare the performance of the original scalar, AoS code to the final SIMD, SoA code on a large (i.e., 100,000 points) data set. Be sure to build both versions with compiler optimizations enabled. Without that the performance comparisons will not be meaningful.

Extra credit: In addition to the SIMD, SoA optimizations, *design* a parallel version of the algorithm. Be sure to go through the parallel design steps that we have studied this term and document the reasoning for your design decisions. *You do not need to implement the parallel version to get credit.*

Criteria	Excellent	Good	Satisfactory	Unacceptable
Completion	Program correctly implements all required elements in a manner that is readily apparent when the program is executed. User interface is complete and responsive to input. Program documents user interface functionality.	Program implements all required elements, but some elements may not function correctly. User interface is complete and responsive to input.	Program implements most required elements. Some of the implemented elements may not function correctly. User interface is complete and responsive to input.	Many required elements are missing. User interface is incomplete or is not responsive to input.
Correctness	Program executes without errors. Program handles all special cases. Program contains error checking code.	Program executes without errors. Program handles most special cases.	Program executes without errors. Program handles some special cases.	Program does not execute due to errors. Little or no error checking code included.
Efficiency	Program uses solution that is easy to understand and maintain. Programmer has analysed many alternate solutions and has chosen the most efficient. Programmer has included the reasons for the solution chosen.	Program uses an efficient and easy to follow solution (i.e., no confusing tricks). Programmer has considered alternate solution and has chosen the most efficient.	Program uses a logical solution that is easy to follow, but it is not the most efficient. Programmer has considered alternate solutions.	Program uses a difficult and inefficient solution. Programmer has not considered alternate solutions.
Presentation & Organization	Program code is formatted in a consistent manner. Variables, functions, and data structures are named in a logical, consistent manner. Use of white space improves code readability.	Program code is formatted in mostly consistent with occasional inconsistencies. Variables, functions, and data structures are named in a logical, mostly consistent manner. Use of white space neither helps or hurts code reability.	Program code is formatted with multiple styles. Variables, functions, and data structures are named in a logical but inconsistent manner. Use of white space neither helps or hurts code reability.	Program code is formatted in an inconsistent manner. Variables, functions, and data structures are poorly named. Use of white space hurts code reability.
Documentation	Code clearly and effectively documented including descriptions of all global variables and all non-obvious local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted, as are the input requirements and output results.	Code documented including descriptions of most global variables and most non-obvious local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted, as are the input requirements and output results.	Code documented including descriptions of the most important global variables and the most important local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted.	No useful documentation exists.

This rubric is based loosely on the "Rubric for the Assessment of Computer Programming" used by Queens University (http://educ.queensu.ca/ compsci/assessment/Bauman.html).