Project -- Tour d'Algorithms: Cuda Sorting Championship

Overview:

In this assignment, you will explore massively parallel computing through the Cuda programming model. This assignment will give you experience programming for GPU style accelerators. The concepts used in this assignment are applicable to Cude, OpenMP, and Xeon Phi models of computing.

If you have no experience with Cuda, you should check out:

1. The Cuda C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
2. The *Thrust* documentation. http://docs.nvidia.com/cuda/thrust/

GPU style computing is set take advantage of increasing computing power with doubling of transistor density much more than desktop and server superscalar processors.

Description:

Code:

You will investigate **three** sorting algorithms.

1. The sort algorithm provided by the Cuda Thrust Library. This implementation is already done for you! Check out the Thrust documentation for instructions on how to use it. This will simply give you a baseline to see the performance the GPU is capable of.
2. A single threaded sorting algorithm that will run on the GPU. This implementation is meant for some basic practice with Cuda Syntax and memory models. You may implement any sorting algorithm you wish.
3. A massively parallel sorting algorithm on the GPU. Again, you may implement any sort you wish, but it **MUST** be massivley parallel. Given an input of 1, 000, 000 numbers, your algorithm should use at least 1, 024 at once at some point in the algorithm.

The easiest massively parallel algorithm is probably some form of bucket sort, but it does have some complexities. A forward recursive radix sort might actually be the easiest to implement (and will give you practice with bitwise operators), and is probably the fasted algorithm you could implement.

For **EACH** algorithm you will create 1 solution, totaling in **3 solutions**. A **separate** program that compiles to an individual executable is required for **EACH SOLUTION**.

Each solution must take command line arguments in the following form:

`[executable name] [number of random integers to generate] [seed value for random number generation]`

The executable names should be the following:

- `thrust`
- `singlethread`
- `multithread`

Report:

You must turn in a report with graphs that depict the running times of both of your algorithms for varying sizes of arrays. You should use enough numbers in the arrays to generate times long enough to make sense out of. You will have 3 graphs, one for `thrust`, one for `singlethread` and another for `multithread`. Therefore, your report will have at least 3 sections, one for each graph and its associated explanation.

For each section, you must describe in a manner in which a reasonable computer science student will understand, your algorithm, its theoretic time complexity, and its implementation details (i.e., what are the interesting implementation aspects: for example, how did you deal with arrays that you did not know their maximum size at compile time). Therefore, each section **must have $ subsections** titled: Algorithm Description, Time Complexity, Implementation Details, and Performance (the part that includes you graph and the performance discussion). Your graph should reinforce the statements made in those subsections.

In Subsection 4 of each Section, your report MUST explain why the graphs are behaving as they are. Your explanation must be based in computer architecture (i.e., memory access/behavior, CPU architectural issues, OS overhead, etc).

Deliverables:

A single ZIP file containing:

1. A README file in **plain text** containing your group information and anything I need to know about running your program.
2. A Makefile with a default target that will build ALL of your executables.
3. The source code for your 3 solutions. Your executables should print the running time at the end!
4. A **PDF** containing your report, with at least 3 sections (1 for each graph and associated discussion) each containing the required 4 subsections.

Skeleton File

You must use the following skeleton file. If you do not use it, be sure your code executes (with command line args) **EXACTLY** as this does. Note that the safe call and error checking functions can be found easily by searching the Internet. It is possible that updated functions exist.

```
#include <iostream>
#include <sstream>
#include <fstream>
#include <string>
using namespace std;


/*********************************************************
 * *******************************************************
 *                  error  checking stufff
 *******************************************************
 *******************************************************/
// Enable this for error checking
#define CUDA_CHECK_ERROR

#define CudaSafeCall( err )      __cudaSafeCall( err,___FILE__,___LINE   )
#define CudaCheckError()         __cudaCheckError(____FILE__,___LINE   )

inline void___cudaSafeCall( cudaError err,
        const char *file, const int line )
{
    #ifdef CUDA_CHECK_ERROR

    #pragma warning( push )
    #pragma warning( disable: 4127 ) // Prevent warning on do-while(0);

    do
    {
            if ( cudaSuccess != err )
            {
                    fprintf( stderr,
                            "cudaSafeCall() failed at %s:%i : %s\n",
                             file, line, cudaGetErrorString( err ) );
                    exit( -1 );
            }
    } while ( 0 );

    #pragma warning( pop )

    #endif  // CUDA_CHECK_ERROR

    return;
```

```
}

inline void __cudaCheckError( const char *file, const int line )
{
    #ifdef CUDA_CHECK_ERROR

    #pragma warning( push )
    #pragma warning( disable: 4127 ) // Prevent warning on do-while(0);

    do
    {
            cudaError_t err = cudaGetLastError();
            if ( cudaSuccess != err )
            {
                    fprintf( stderr,
                            "cudaCheckError() failed at %s:%i : %s.\n",
                             file, line, cudaGetErrorString( err ) );
                    exit( -1 );
            }

            // More careful checking. However, this will affect performance.
            // Comment if not needed.
            err = cudaThreadSynchronize();
            if( cudaSuccess != err )
            {
                    fprintf( stderr,
                            "cudaCheckError() with sync failed at %s:%i : %s.\n",
                            file, line, cudaGetErrorString( err ) );
                    exit( -1 );
            }
    } while ( 0 );

    #pragma warning( pop )

    #endif // CUDA_CHECK_ERROR

    return;
}

/*************************************************************
 * ***********************************************************
 *                  end of error checking stuff
 ***********************************************************
 ***********************************************************/



// function takes an array pointer, and the number of rows and cols in the array, and
// allocates and intializes the array to a bunch of random numbers
// Note that this function creates a 1D array that is a flattened 2D array
// to access data item data[i][j], you must can use data[(i*rows) + j]
int * makeRandArray( const int size, const int seed ) {

    srand( seed );
    int * array = new int[ size ];
    for( int i = 0; i < size; i ++ ) {
        array[i] = std::rand() % 1000000;
    }
    return array;
}
```

```
//*******************************//
// your kernel here!!!!!!!!!!!!!!!!
//*******************************//
__global   void matavgKernel( ...  )
{

}


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

    int * array; // the poitner to the array of rands
    int size, seed; // values for the size of the array
    bool printSorted = false;
    // and the seed for generating
    // random numbers

    // check the command line args
    if( argc < 4 ){
        std::cerr << "usage: "
            << argv[0]
            << " [amount of random nums to generate] [seed value for rand]"
            << " [1 to print sorted array, 0 otherwise]"
            << std::endl;
        exit( -1 );
    }

    // convert cstrings to ints
    {
        std::stringstream ss1( argv[1] );
        ss1 >> size;
    }
    {
        std::stringstream ss1( argv[2] );
        ss1 >> seed;
    }
    {
        int sortPrint;
        std::stringstream ss1( argv[2] );
        ss1 >> sortPrint;
        if( sortPrint == 1 )
            printSorted = true;
    }
    // get the random numbers
    array = randNumArray( size, seed );



    /**********************************
     *       create a cuda timer to time execution
     **********************************/
    cudaEvent_t startTotal, stopTotal;
    float timeTotal;
    cudaEventCreate(&startTotal);
    cudaEventCreate(&stopTotal);
    cudaEventRecord( startTotal, 0 );
    /**********************************
     *       end of cuda timer creation
     **********************************/
```

```
    ////////////////////////////////////////////////////////////////////
    ////////////////////////  YOUR CODE HERE       /////////////////////
    ////////////////////////////////////////////////////////////////////
    /*
     *       You need to implement your kernel as a function at the top of this file.
     *       Here you must
     *       1) allocate device memory
     *       2) set up the grid and block sizes
     *       3) call your kenrnel
     *       4) get the result back from the GPU
     *
     *
     *   to use the error checking code, wrap any  cudamalloc functions as follows:
     *      CudaSafeCall( cudaMalloc( &pointer_to_a_device_pointer,
     *          length_of_array * sizeof( int ) ) );
     *   Also, place the following function call immediately after you call your kernel
     *   ( or after any other cuda call that you think might be causing an error )
     *      CudaCheckError();
     */




    /*********************************
     *       Stop and destroy the cuda timer
     *********************************/
    cudaEventRecord( stopTotal, 0 );
    cudaEventSynchronize( stopTotal );
    cudaEventElapsedTime( &timeTotal, startTotal, stopTotal );
    cudaEventDestroy( startTotal );
    cudaEventDestroy( stopTotal );
    /*********************************
     *       end of cuda timer destruction
     *********************************/

    std::cerr << "Total time in seconds: "
                << timeTotal / 1000.0 << std::endl;
    if( printSorted ){

        ////////////////////////////////////////////////
        /// Your code to print the sorted array here //
        ////////////////////////////////////////////////

    }
}
```