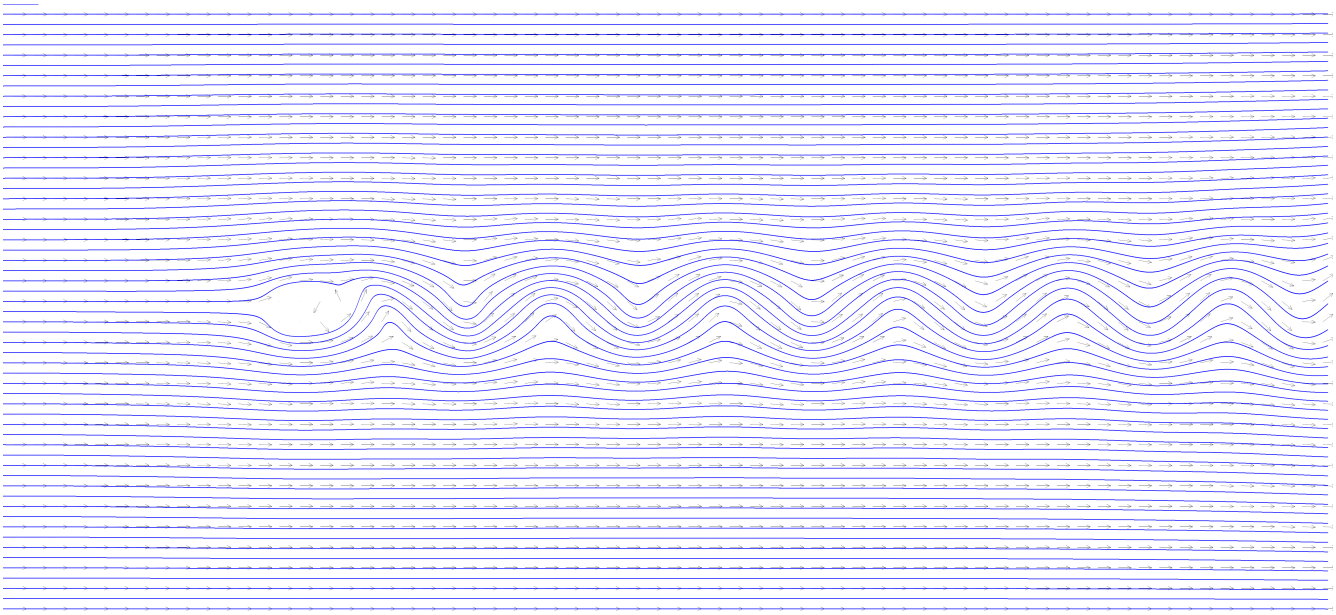
Gone With the Wind - Streamline Calculation

Philip Nelson

9th December 2021



Example output produced by the programs

Streamline Calculation

For this project I implemented a number of libraries to support calculating streamlines in a vector field and visualizing them. The image above is an example of the output produced by these programs. The first supporting library is a Math library to describe points, vectors, and to perform the Runge Kutta and Euler's method. The Vector Field library is build on top of Math and is used to model a vector field. It supports operations such as reading from a raw file, performing bilinear interpolation, and calculating streamlines. To perform error handling, the library Expected supports monadic error handling through the `Expected<T>`. This type represents a value that is expected to exist, however; if an error occurs, such as attempting to access a value outside the vector field, the expected value becomes invalid. This invalid state can continue to be operated on and the error state propagates through the program until the value must be checked, at which point the error can be handled. This performs a similar job to stack unrolling through throwing exceptions, however; exceptions can not be thrown in CUDA so this is usable on a GPU.

To visualize the vector fields and streamlines, I wrote a Graphics library to draw lines, arrows, and curves to a pixel array. The image can be written in the `ppm` file format and converted to a compressed format such as `png` or `jpeg` with the ImageMagick commandline utility. This allowed me to produce graphical representations of my calculations. More examples of vector fields generated from mathematical functions can be seen in Appendix A.

Shared Memory

The shared memory application, found in `driver_thread/main.cpp` and built with the target `run_thread`, uses `std::thread`s to calculate the streamlines in parallel. First, the main thread generates a list of starting points.

Then the points are handed off in blocks to each worker thread to calculate streamlines from the starting points. Finally, each thread locks a mutex onto a global list of streamlines and adds theirs to the list. The main thread then draws the streamlines onto the image.

Distributed Memory

The distributed memory application, found in `driver_mpi/main.cpp` and built with the target `run_mpi`, uses MPI to calculate the streamlines in parallel. The processes are first split into two communicators. One process is left behind to draw streamlines onto the image, and the rest are put into a worker communicator. Worker rank 0 then generates the list of starting points and scatters them to the rest of the workers. They work in parallel to calculate the streamlines and as each one is finished, it is sent to the world rank 0 process to draw to the image.

In the submitted application, each process reads the vector field, however I was working on an implementation where each process would read a block of the field. I was able to construct a communicator with a cartesian topology and have each process read a block of the field based on their two dimensional coordinate, however; I did not have time to finish this implementation. The plan was to have processes begin calculation streamlines in their blocks, and when a line reached the boundary of one region, to pass it to the adjacent process to continue, until the line left the field or the maximum line length was reached.

GPU

The GPU implementation, found in `driver_cuda/main.cu` and built with the target `run_cuda`, uses CUDA to calculate the streamlines in parallel. The input is copied to the GPU and each thread constructs a Vector Field object with the input array and dimensions. The vector field is instructed to not take ownership of the input array and act only as a view on the data. This gives access to the functions to calculate streamlines without deleting the input array when the object is destroyed. Each thread picks a starting position based on its global thread id and calculates the streamline. It writes its results to a global output array. The output is not able to change size, so enough space is allocated to fit each curve if it were the max length. Then the first point in the curve is used to store its length. After the output is copied back to the host the curves are drawn. The fist point is read to determine the length of the curve, then that many successive points are read to construct the image.

Building

The project includes a CMakeLists.txt. The following commands can be used to build and run the projects on the center for high performance computing (CHPC).

```
$ module load cmake/3.21 gcc/6 openmpi/4 cuda/10
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Release ..
$ make -j 4
$ cd ..
```

Running

Each target provides its usage with the `-h` option. An image will be written to `out.ppm` and converted to `out.png` via the commandline utility ImageMagick. Each implementation expects to find `cyl2d_1300x600_float32.raw` in the current working directory.

```
$ ./build/driver_thread/run_thread
$ mpiexec -n 4 ./build/driver_mpi/run_mpi
$ ./build/driver_cuda/run_cuda
```

If you want to use a different file, it can be specified with the following commandline arguments.

Shared Memory - Threads

```
$ ./build/driver_thread/run_thread -h
Compute streamlines in a vector field with multiple threads
Usage:
  ./build/driver_thread/run_thread [OPTION...]

  -h, --help                    Show usage
  -n, --num_threads arg         Number of threads (default: 4)
  -s, --num_streamlines arg     Number of streamlines (default: 100)
  -m, --max_line_length arg     Maximum number of line segments for each
                                streamline (default: 10000)
  -i, --image_scale_factor arg  Number of times larger to make the output
                                image than the input field (default: 2)
      --height arg              input file height (default: 600)
      --width arg               input file width (default: 1300)
  -f, --file arg                input file (default:
                                cyl2d_1300x600_float32.raw)
      --no_image                skip image generation
```

Distributed Memory - MPI

```
$ mpiexec -n 1 ./build/driver_mpi/run_mpi -h
Compute streamlines in a vector field with multiple processes
Usage:
  ./driver_mpi/run_mpi [OPTION...]

  -h, --help                    Show usage
  -s, --num_streamlines arg     Number of streamlines (default: 100)
  -m, --max_line_length arg     Maximum number of line segments for each
                                streamline (default: 10000)
  -i, --image_scale_factor arg  Number of times larger to make the output
                                image than the input field (default: 2)
      --height arg              input file height (default: 600)
      --width arg               input file width (default: 1300)
  -f, --file arg                input file (default:
                                cyl2d_1300x600_float32.raw)
      --no_image                skip image generation
```

GPU - CUDA

```
$ ./build/driver_cuda/run_cuda -h
Compute streamlines in a vector field on a GPU
Usage:
  ./driver_cuda/run_cuda [OPTION...]

  -h, --help                    Show usage
  -n, --num_threads_per_block arg
                                Number of CUDA threads per block (default:
                                1024)
  -s, --num_streamlines arg     Number of streamlines (default: 100)
  -m, --max_line_length arg     Maximum number of line segments for each
                                streamline (default: 10000)
  -i, --image_scale_factor arg  Number of times larger to make the output
                                image than the input field (default: 2)
```

```
    --height arg              input file height (default: 600)
    --width arg               input file width (default: 1300)
-f, --file arg                input file (default:
                              cyl2d_1300x600_float32.raw)
    --no_image                skip image generation
```

Results

Trials were performed with 64, 128, 256, 512, and 1024 streamlines with a 10 trial average for each configuration. The total execution time and streamline calculation time were measured. Streamline calculation includes generating the starting points, distributing them to threads or processes, and gathering the result. The total program time additionally included parsing input from the commandline and reading the input vector field.

Shared Memory - Threads

The trials were run on one node of kingspeak at the CHPC. The file `bench_thread.sh` contains the complete configuration used to submit the job to `SLURM` and `bench_thread.py` ran the trials. The nodes have 16 cores and I ran trials on 2 - 32 threads. The results of these trials can be seen in Figure **??**. The speedup and efficiency show that after 16 threads, performance increase slows which is to be expected when exceeding the hardware capability. However, up to 16 threads scales linearly.

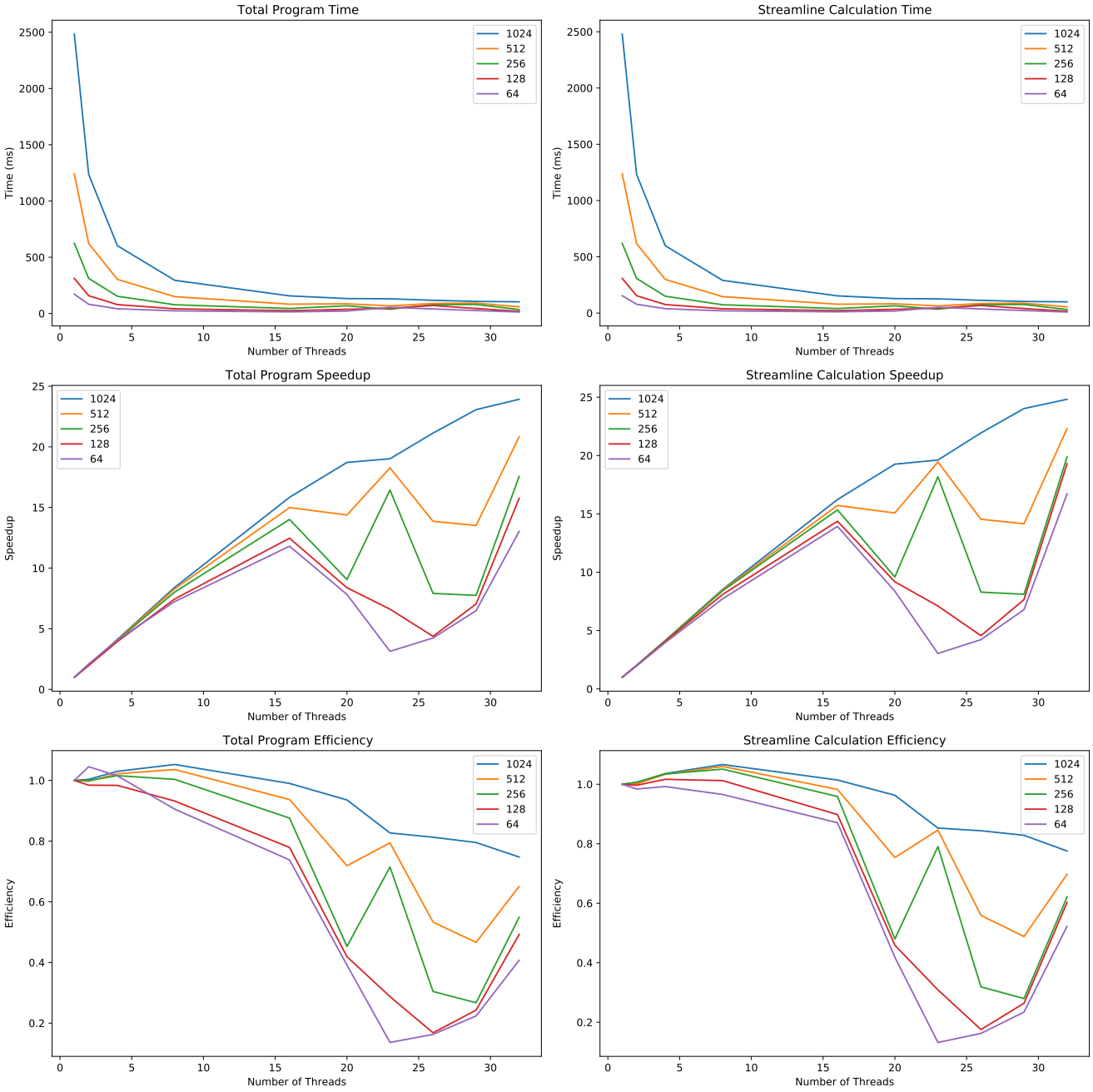


Figure 1: Timing, Speedup, and Efficiency for Shared Memory Implementation

Distributed Memory - MPI

The trials were run on four nodes of kingspeak at the CHPC. The file `bench_mpi.sh` contains the complete configuration used to submit the job to `SLURM` and `bench_mpi.py` ran the trials. With 16 cores per node, I ran trials up to 64 processes. The results of these trials can be seen in Figure **??**. Up to 16 threads scales linearly, however; after this, the performance decreases. This must be the point where the communication overhead of MPI takes over and performance suffers.

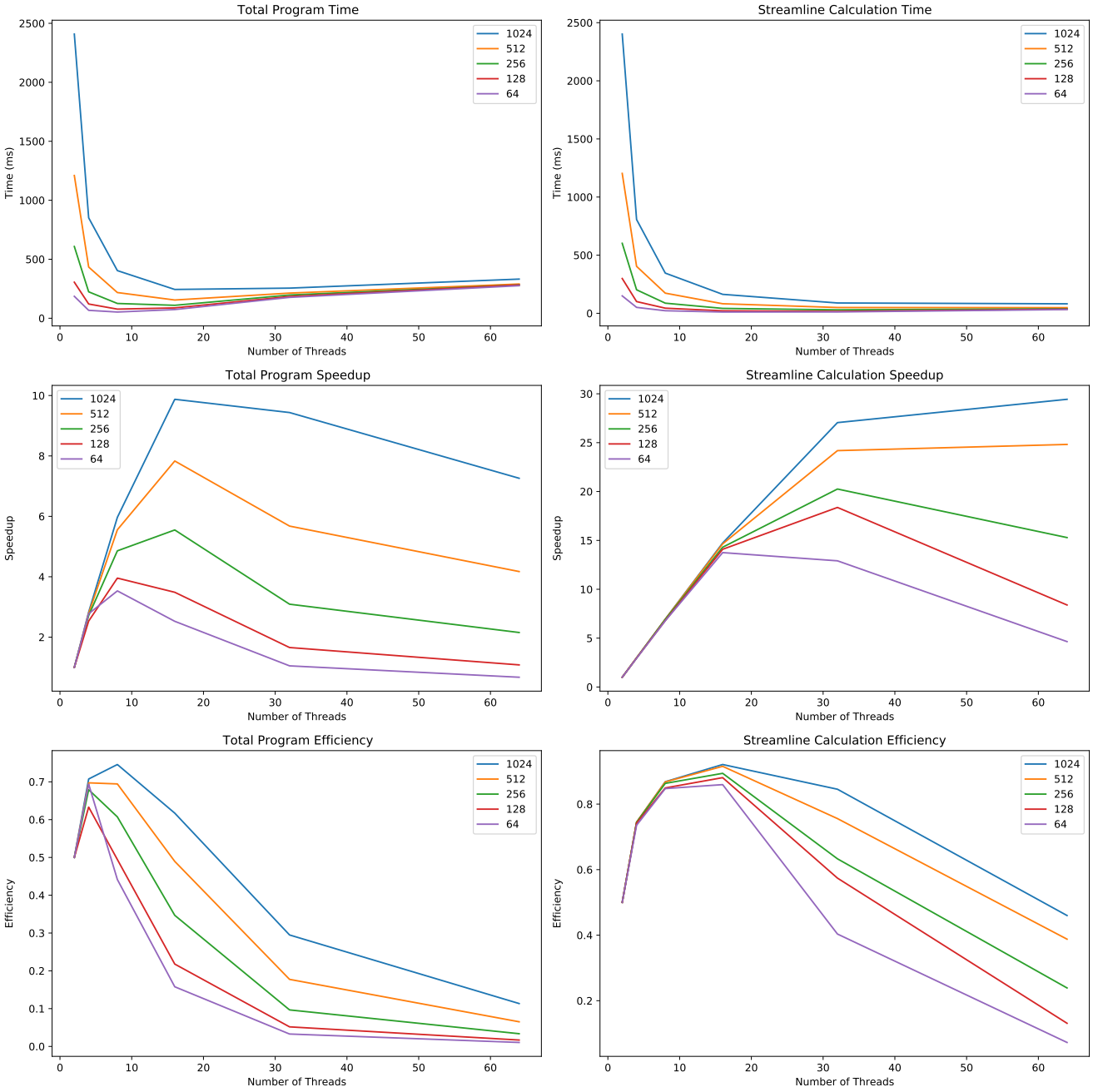


Figure 2: Timing, Speedup, and Efficiency for Distributed Memory Implementation

GPU - CUDA

The trials were run on a node of notchpeak on a NVIDIA Tesla K80 at the CHPC. The file `bench_cuda.sh` contains the complete configuration used to submit the job to `SLURM` and `bench_cuda.py` ran the trials. In the GPU trials, I tried varying the number of threads per block which had a dramatic impact on performance. The plots reveal that between 8 and 16 threads per block produced the best performance. The results of these trials can be seen in Figure **??**. The top plots provide a zoomed in view of the bottom plots. This may have to do with the warp size and the capacity of the scheduling multi processor.

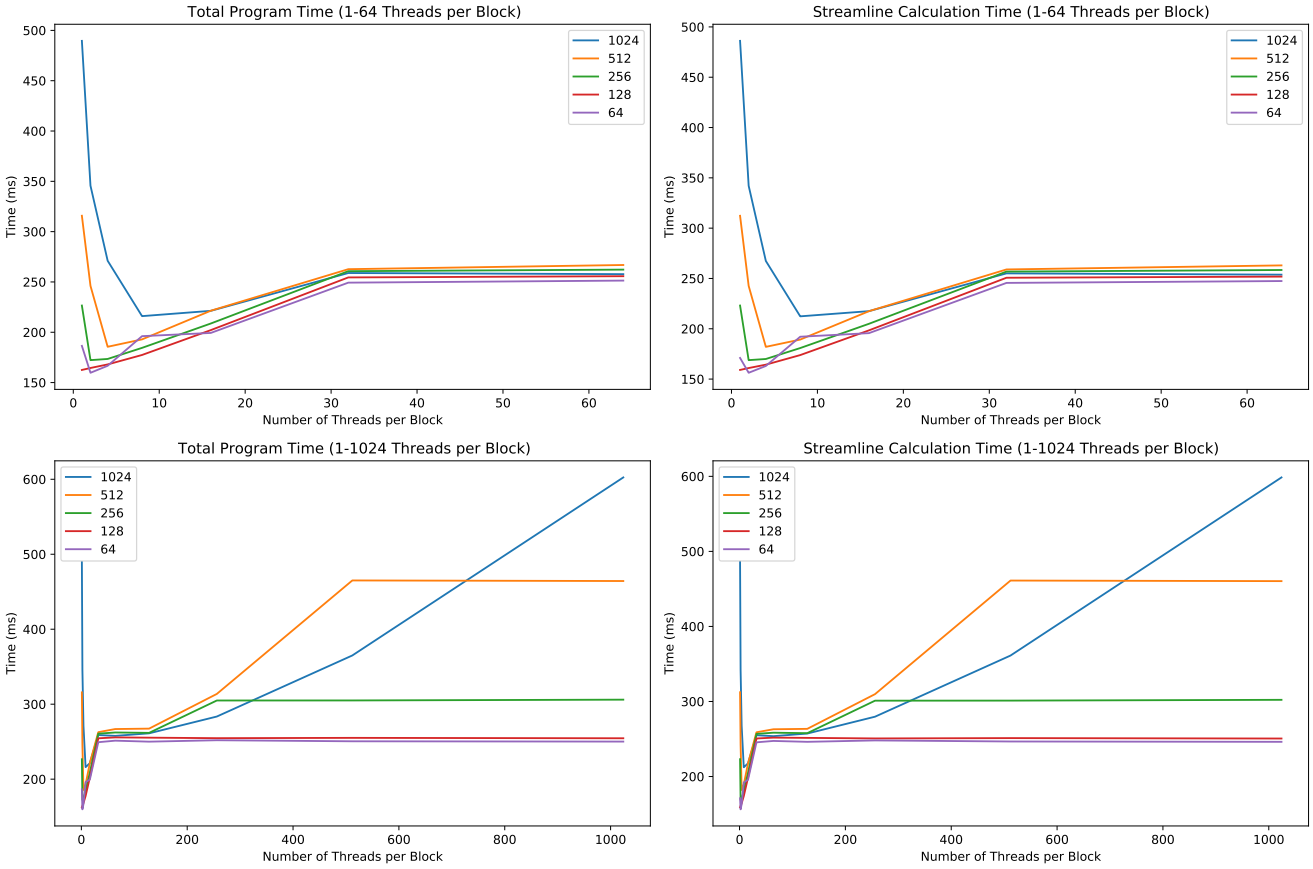


Figure 3: Timing for GPU Implementation

Comparison

It is interesting to compare all of the methods, see Figure 4. I was surprised to see the GPU be outperformed by the CPU, however; this application is not taking full advantage of the GPU. At most I am using 1024 threads which is not much for a GPU. There is a lot of performance being left on the table here. It is also interesting to see that threads and MPI are pretty even in their performance. This is likely because their implementations are very similar and there is not much communication between processes so their execution is comparable.

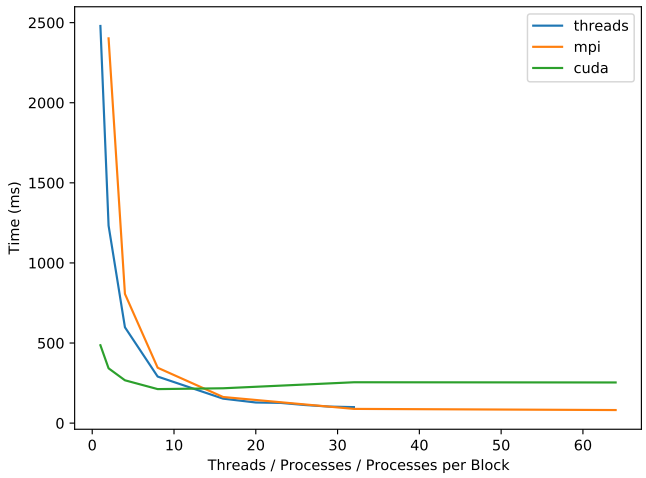
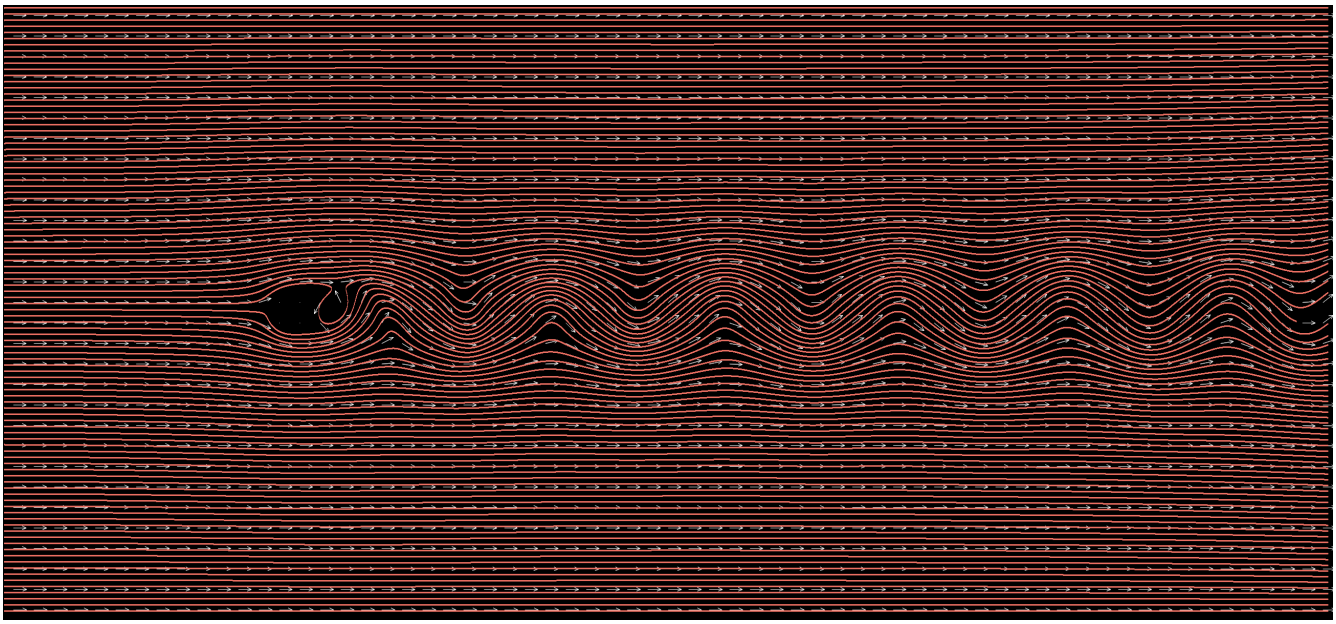
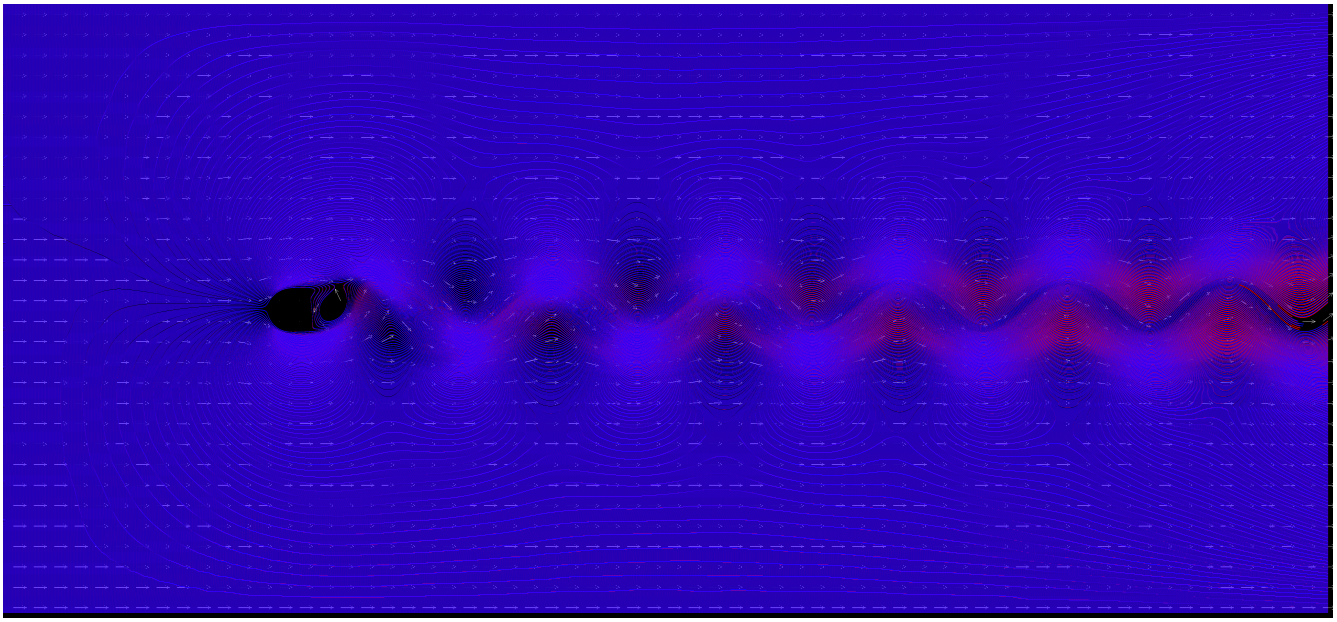


Figure 4: Timing comparison for all methods (1024 streamlines)

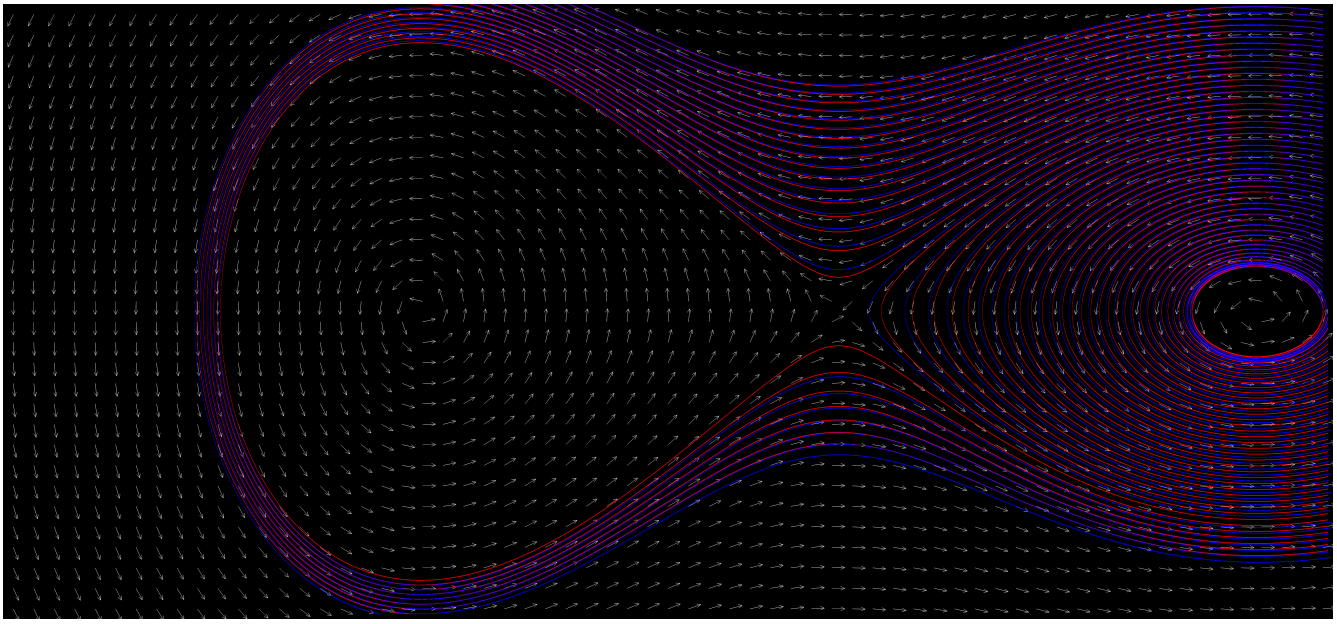
Appendix A

The following are images of vector fields produced by the programs

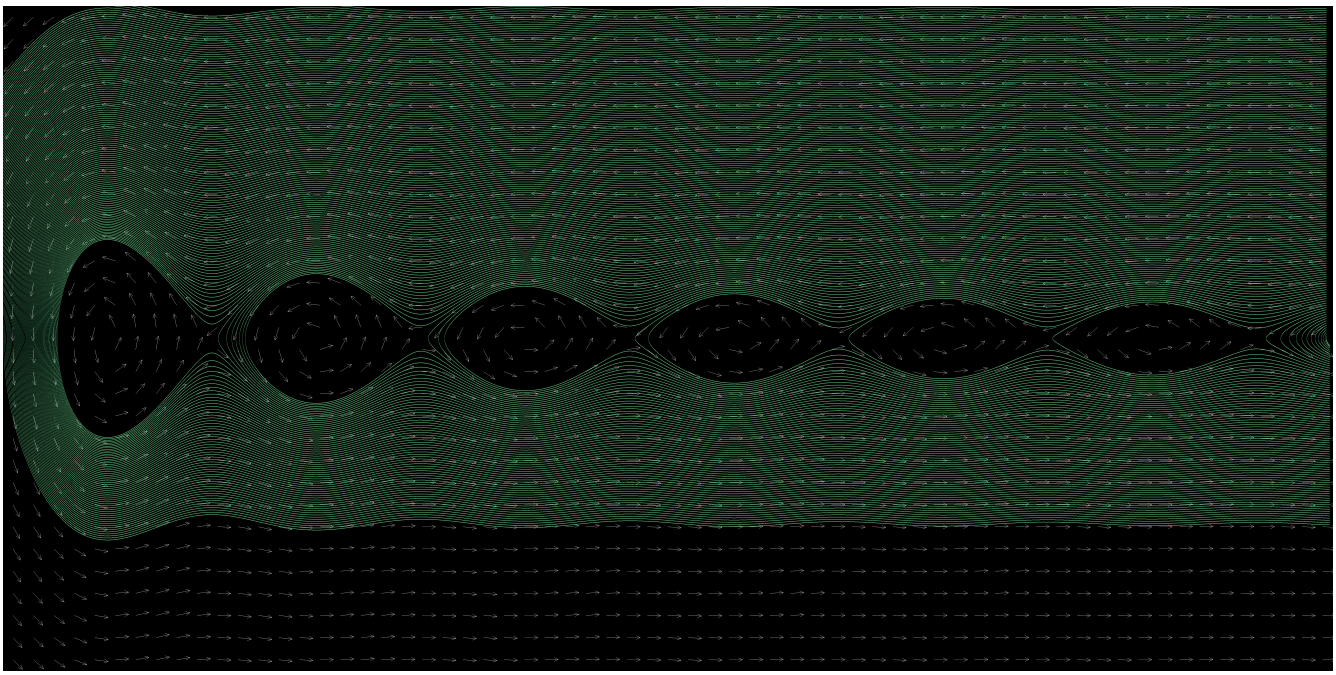




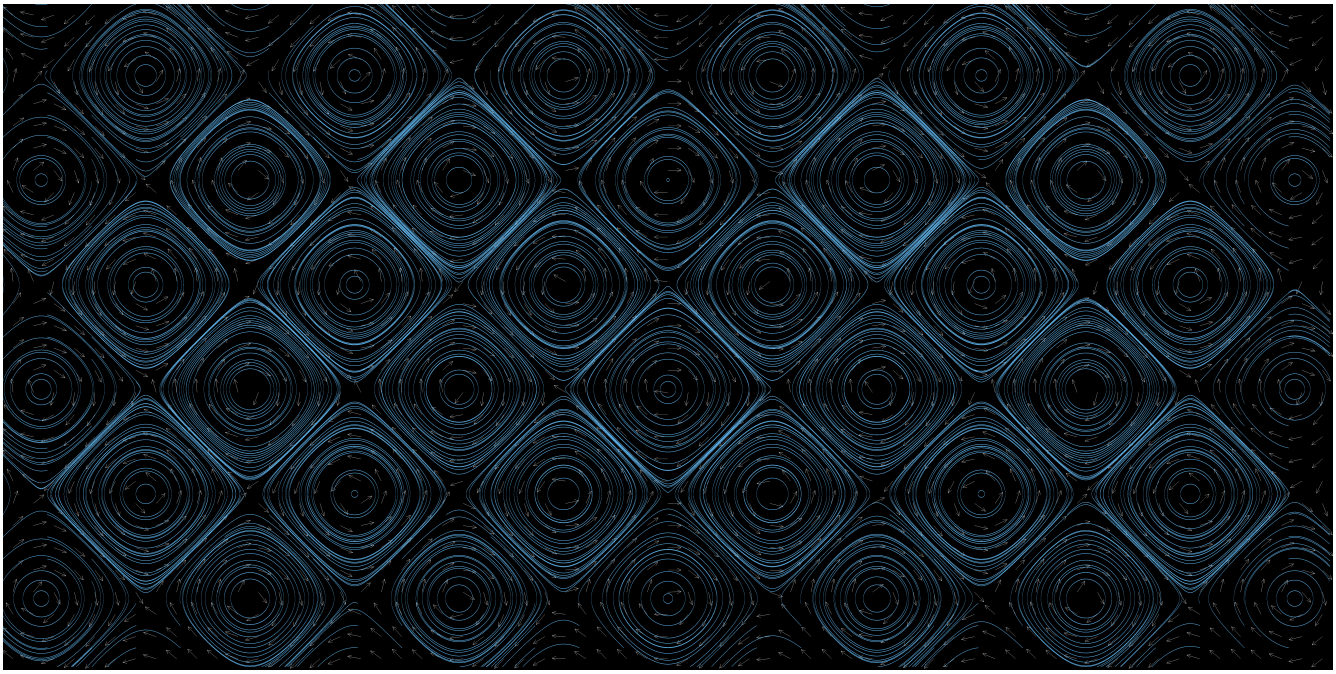
Red: Euler's Method | Blue: Runge Kutta



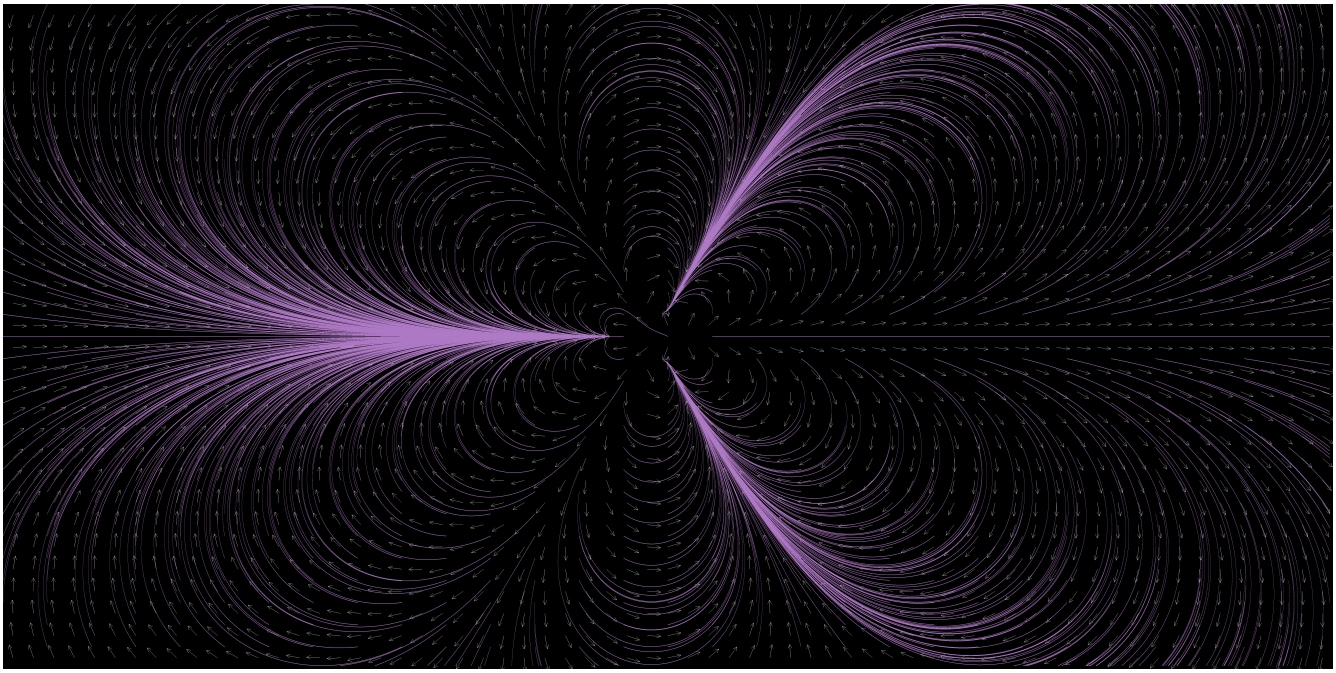
Red: Euler's Method | Blue: Runge Kutta $(x, y) \to (xy, 10sin(x))$



$(x, y) \rightarrow (xy, 10sin(x))$



$(x, y) \rightarrow (cos(y), sin(x))$



$(x, y) \to c = x + iy, z = 0, f(z) = z^2 + c$ for 3 iterations