COMS3008A - Assignment One - Report

2561034

27 April 2024

Introduction

In this report we will be going through different runs of the Julia fractals set.

Details of each Implementation

1D row-wise parallel

The goal of the first implementation was to parallelise the Julia set Fractal calculation by distributing the workload across multiple threads. We take the approach of manual decomposition, by assigning rows of the image to individual threads. We use the OpenMP library to create and manage a team of threads where

```
omp_set_num_threads(NUM_THREADS)
```

controls the number of threads entering the parallel region and the outer loop

```
for (y=tid; y < DIM ; y = y + tthreads)
```

will iteratively assign rows of the image to threads based on their unique thread ID. Hence if there are four threads, Thread 0 processes rows 0, 4, 8, ... , Thread 1 processes rows 1, 5, 9, ... , and so on.
The calculations of each pixel in the Julia set are also independent which allows us to successfully parallelise this computation.
The inner loop

```
for (int x=0; x<DIM; x++)
```

processes the columns for each assigned row.
Since OpenMP offers implicit synchronisation at the end of the parallel region clean-up was not necessary and it makes sure that each thread completes its work before the program proceeds.
The image data ptr is also shared across all threads, where then threads collaborate to completely fill the image.

The optimisations that were applied is just a basic decomposition of the array and "handing them out" to the different threads.We also pre-compute the offset to reduce the calculations performed. We can do this since the base-offset stays the same for each iteration of x for every y iteration. So it is unnecessary to actually recompute it each time for each x iteration of y.

```
int baseOffset = y * DIM;
```

this is the base offset which only changes every outer iteration and

```
int offset = x + baseOffset;
```

is the normal offset.
This implementation is also well load balanced since we have distributed equal amounts of work to each thread.

1D column-wise parallel

The goal of the 1D column-wise implementation was similar to the 1D row-wise parallel implementations goals. We wanted to parallelise the Julia set fractal calculation in this case by distributing the workload across multiple threads by manual decomposition. We assign columns of the image to individual threads and we use the OpenMP library to create and manage the team of threads where

```
omp_set_num_threads(NUM_THREADS)
```

controls the number of threads entering the parallel region.
The outer loop in this case for ease of use is flipped to start with rows rather than columns. Hence we have

```
for (x=tid; x < DIM ; x = x + tthreads)
```

which will iterate over rows of the image and assign threads columns of the image to a unique thread ID. The distribution follows the same pattern as above where thread 0 processes columns 0, 4, 8, ... , Thread 1 processes columns 1, 5, 9, ... , and so on.
The inner loop

```
for (int y=0; y<DIM; y++)
```

processes the entire row of whichever column has been assigned. We had to change the Julia function call from

```
int juliaValue = julia( x, y );
```

to

```
int juliaValue = julia( y, x );
```

since i have flipped the entire for loop and if we don't change this we will get a mirrored image since it treats the rows as columns, the columns as rows and the x's as y's vice versa.
Since OpenMP offers implicit synchronisation at the end of the parallel region clean-up was not necessary and it makes sure that each thread completes its work before the program proceeds.
The image data ptr is also shared across all threads, where then threads collaborate to completely fill the image.
The optimisations that were applied is just a basic decomposition of the array and "handing them out" to the different threads.We also pre-compute the offset to reduce the calculations performed. We can do this since the base-offset stays the same for each iteration of x for every y iteration. So it is unnecessary to actually recompute it each time for each x iteration of y.

```
int baseOffset = y * DIM;
```

this is the base offset which only changes every outer iteration and

```
int offset = x + baseOffset;
```

is the normal offset.
This implementation is also well load balanced since we have distributed equal amounts of work to each thread.
This implementation also improves the Memory Access Pattern since each thread iterates over consecutive columns which improves memory access patterns, which can benefit from hardware caching.

2D Row Block Parallel

This implementation also parallelises the Julia set fractal calculation, but in this implementation we distribute the work over threads in a row-block pattern. This mean that each thread works on a block of rows in the image rather than a single row iteratively.

```
omp_set_num_threads(NUM_THREADS)
```

sets the number of threads entering the parallel region.

```
#pragma omp parallel private(tid, rows_per_thread, start_row, end_row, y)
```

sets the parallel region with the variables tid, `rows_per_thread`, `start_row`, `end_row`, y to be private to each thread. The main thread identification and work distribution is done by the following block of code:

```
tid = omp_get_thread_num();     // Get thread ID
int tthreads = omp_get_num_threads();  // Get total number of threads
```

```
rows_per_thread = DIM/tthreads;
start_row = tid * rows_per_thread;
end_row = start_row + rows_per_thread -1;

if(tid == NUM_THREADS -1 && remainder != 0){
    end_row += remainder;
}
```

each thread gets it unique thread id denoted tid and the total number of threads is tthreads. The image is then divided into roughly equal-sized blocks of rows denoted by `rows_per_thread`. Each thread gets a starting row `start_row` and an ending row `end_row`. The remainder of rows when the image dimension DIM is not evenly divisible by the number of threads is handled by :

```
if(tid == NUM_THREADS -1 && remainder != 0){
    end_row += remainder;
}
```

The row-wise work distribution is then handled by

```
for(y = start_row; y <= end_row ; y++){
    int baseOffset = y * DIM;
    for(int x = 0; x < DIM ; x++){
        int offset = x + baseOffset;
        // ... (rest of the calculation)
    }
}
```

The outer loop

```
for(y = start_row; y <= end_row ; y++)
```

each thread iterates over the block of rows it has been assigned.
The inner loop

```
for(int x = 0; x < DIM ; x++)
```

for each block of rows the thread also processes all columns.
We can also pre-calculate the offset in this implementation. Where since the baseOffset only changes per iteration of the rows in this case we can pre-calculate a base offset then use that for the actual offset per column in the inner loop. This reduces the number of total arithmetic calculations. With this implementation we also get good load balancing and flexibility. We are able to handle extra rows and assign somewhat equal workloads to each thread. We also get some more flexibility by changing the number of threads to find a good balance between our paralallization and overhead.

2D column-block parallel

This implementation is similar to the previous row-block implementation of parallelising the Julia set. The main difference is that is uses a column-block distribution where each thread is assigned a block of columns to process.

```
omp_set_num_threads(NUM_THREADS)
```

sets the number of threads entering the parallel region.

```
#pragma omp parallel private(tid, cols_per_thread, cols_per_thread, end_col, x)
```

sets the parallel region with the variables tid, `cols_per_thread`, `cols_per_thread`, `end_col`, x to be private to each thread. The main task identification and task decomposition is done by

```
tid = omp_get_thread_num();     // Get thread ID
int tthreads = omp_get_num_threads();  // Get total number of threads

cols_per_thread = DIM/tthreads;
start_col = tid * cols_per_thread;
end_col = start_col + cols_per_thread -1;

if(tid == NUM_THREADS -1 && remainder != 0){
    end_col += remainder;
}
```

each thread gets it unique thread id denoted *tid* and the total number of threads is *tthreads*. The image is then divided into roughly equal-sized blocks of rows denoted by `cols_per_thread`. Each thread gets a starting row `start_col` and an ending row `end_col`. The remainder of rows when the image dimension DIM is not evenly divisible by the number of threads is handled by :

```
for(x = start_col; x <= end_col ; x++){
    int baseOffset = x * DIM;
    for(int y = 0; y < DIM ; y++){
        int offset = y + baseOffset;
        // ... (rest of the calculation)
    }
}
```

The outer loop

```
for(x = start_col; x <= end_col ; x++)
```

is where each thread iterates over the columns assigned to its block.
The inner loop where

```
for(int y = 0; y < DIM ; y++)
```

each thread processes the rows given to its column. We can also pre-calculate the offset in this implementation. Where since the baseOffset only changes per iteration of the column in this case we can pre-calculate a base offset then use that for the actual offset per row in the inner loop. This reduces the number of total arithmetic calculations. Also to take note that we have just flipped the for loop. In essence we are rotating the image. To combat the incorrect output given we send the Julia function an x instead of a y and vice versa. Again with this implementation we get good load balancing since each of the threads receives a relatively equal number of rows, the memory access is also better since we can fine tune and tweak the number of threads to test performance.

Plots of Performance

In our performance analysis, we observed the following speedup trends across different parallelization strategies: All the below were run on a computer with 2 threads per core and 4 cores per socket and 1 socket.

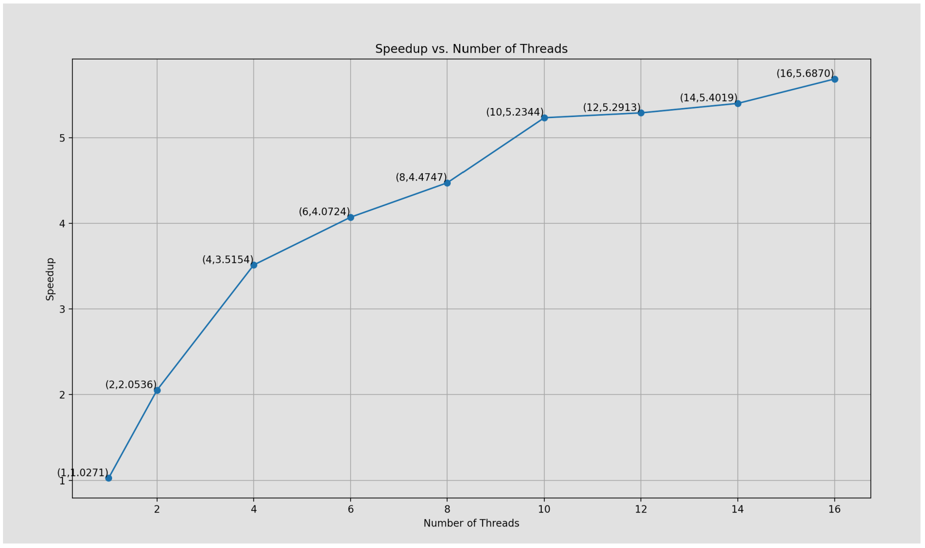


Figure 1: Plot of Speedup vs Threads for 1-D row-wise parallel

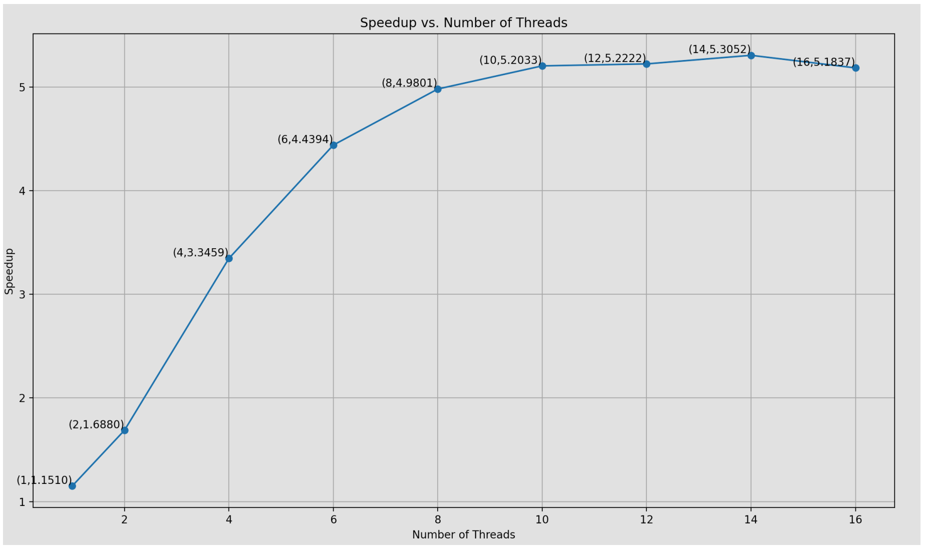


Figure 2: Plot of Speedup vs Threads for 1-D column-wise parallel

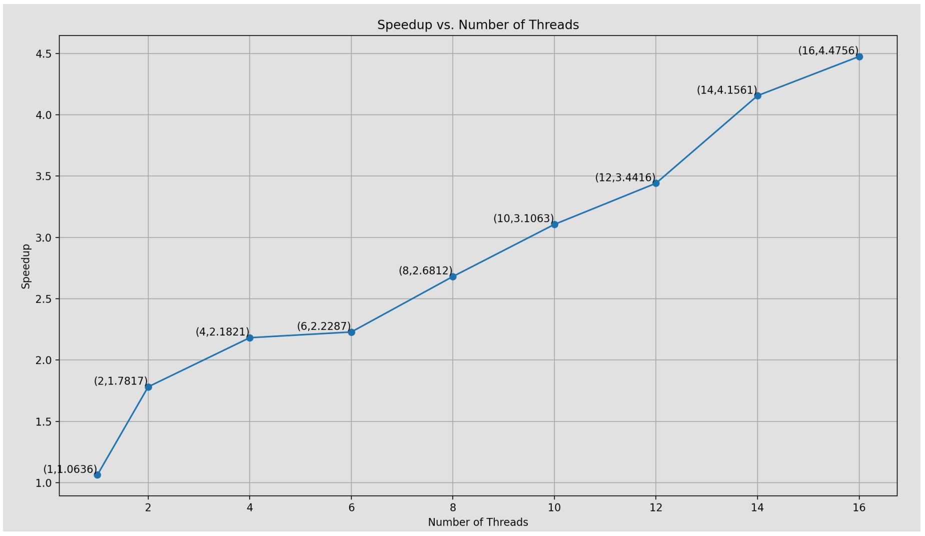


Figure 3: Plot of Speedup vs Threads for 2-D row-wise parallel

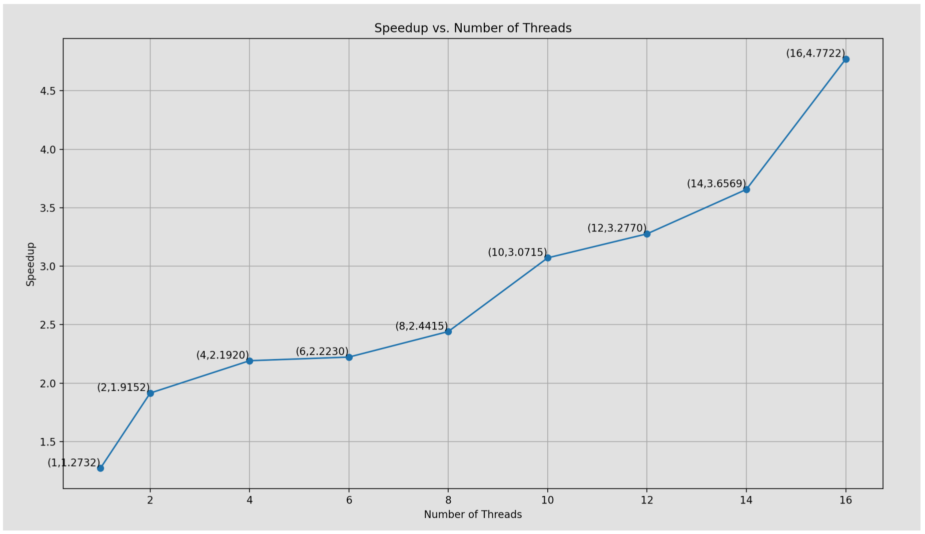


Figure 4: Plot of Speedup vs Threads for 2-D column-wise parallel

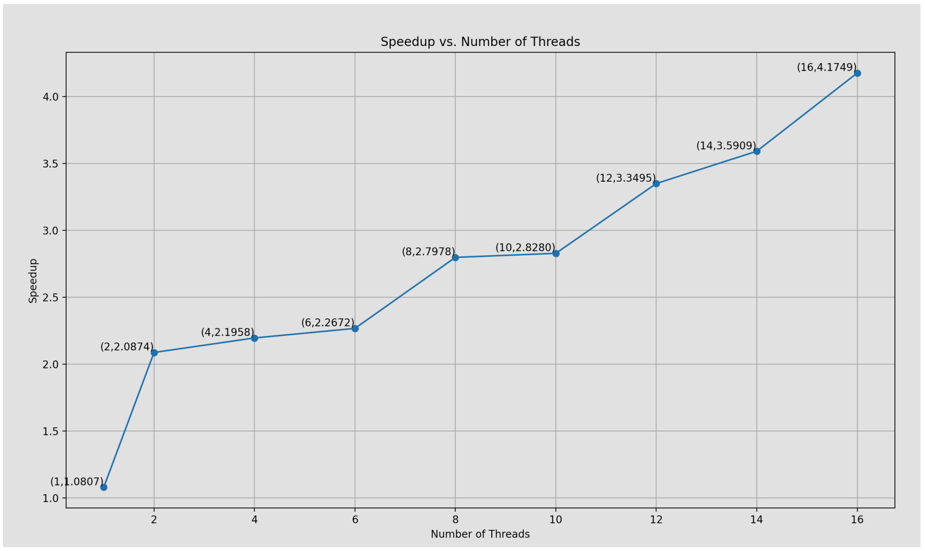


Figure 5: Plot of Speedup vs Threads using for

Analysis

1D row-wise VS 1D column-wise

We observe a difference between performances of 1D row-wise parallel and 1D column wise parallel. This can be attributed to the fact that the ptr array. I personally would have thought row-wise would have performed better than column-wise, since many modern CPU architectures store data in memory as row-major order. This translates to better and higher cache hits and locality of accesses. The data ptr is stored as

```
[row 0 elem 0] [row 0 elem 1] ... [row 0 elem N] [row 1 elem 0] [row 1 elem 1] ...
```

but since we assign a single row to a thread which contains DIM number of elements, so when each thread processes a complete row, it needs to jump across a large chunk of memory (DIM elements) to access the next row it's assigned. This disrupts the ideal sequential memory access pattern that benefits from hardware prefetching and caching. We also get alot of cache inefficiency, each time a thread jumps to the next row, it's very likely that the required data is not already present in the cache. This leads to cache misses, forcing the processor to fetch the data from slower main memory, incurring performance penalties.
With an example we have:

```
[R0G0B0A0] [R1G1B1A1] [R2G2B2A2] [R3G3B3A3]  // Row 0
[R4G4B4A4] [R5G5B5A5] [R6G6B6A6] [R7G7B7A7]  // Row 1
...
```

If a thread processes Row 0, it accesses memory sequentially. But once it finishes with Row 0, it needs to jump the length of an entire row (4 elements in this example) to begin processing Row 1. This jump disrupts the favorable memory access pattern. Hence in column-wise parallelization, each thread works on a contiguous chunk of memory within a column. This aligns better with the row-major layout, leading to more cache-friendly access patterns. The above is obvious when the code is profiled. The following is for the 1D row-wise parallel

```
perf stat -e cache-misses ./fractal

Performance counter stats for './fractal':

    62,511      cache-misses:u

   0.150456916 seconds time elapsed

   0.534318000 seconds user
   0.016191000 seconds sys
```

The following is now for 1D column-wise parallel:

```
perf stat -e cache-misses ./fractal

Performance counter stats for './fractal':

36,379      cache-misses:u

0.144710283 seconds time elapsed

0.543002000 seconds user
0.007985000 seconds sys
```

this proves our point and that the column-wise parallel gives almost half the amount of cache misses. Hence the way we store our data and access it and hand it out to threads play a role in our performance.

2D row-block VS 2D col-block

In this case the difference is not that much. Our findings still show that the 2D column-wise is slightly faster than the 2D row-wise. Once again we go to the way our memory is stored

```
[R0G0B0A0] [R1G1B1A1] ... [RnGnBnAn]
Pixel 0    Pixel 1          Pixel n
```

with our program the image is divided into sub-blocks (tiles) along rows. Each thread gets assigned a block of rows to process. Within a block, a thread iterates over pixels sequentially, which is good for cache utilization. However, when a thread finishes its block and moves to the next one, it jumps a significant distance in memory. Even though it will jump less than the 1D implementation in case of a high DIM we might have a lot of jumps. With the current implementation of DIM = 768 and 16 threads it divides well and not a lot of jumps are needed but as the program were to scale it would become more apparent. An example of this is:

```
[T1] [R0G0B0A0] [R0G1B1A1] [R1G0B0A0] [R1G1B1A1] ... // Thread 1's block (row-wise)
[T2] [R2G2B2A2] [R2G3B3A3] [R3G2B2A2] [R3G3B3A3] ... // Thread 2's block (row-wise)
```

Here, when T1 finishes its block, it needs to jump to the next row block, which involves traversing a large chunk of memory (n pixels in this example) corresponding to pixels in Thread 2's block. This jump disrupts cache locality and leads to cache misses.
With 2D column-block we have while iterating over a block's pixels, there might be some non-contiguous access within the block (depending on the block size and data alignment). However, when a thread finishes its block and moves to the next one, it jumps to a much smaller distance in memory compared to row-wise.

```
[R0G0B0A0] [R1G1B1A1] [R2G2B2A2] [R3G3B3A3]  // Thread 1's block (column-wise) - T1
[T1] [R0G0B0A1] [R1G1B1A1] [R2G2B2A1] [R3G3B3A1]  // T1 continued
[T2] [R0G0B0A2] [R1G1B1A2] [R2G2B2A2] [R3G3B3A2]  // Thread 2's block (column-wise)
[T2] [R0G0B0A3] [R1G1B1A3] [R2G2B2A3] [R3G3B3A3]  // T2 continued
```

Here, Thread 1 finishes its block (T1) and needs to jump to the next one . However, the jump distance is only the size of a single row (4 pixels) compared to the entire row block in row-wise partitioning. This smaller jump is more cache-friendly, leading to better performance. The cache misses are similar at this level of usage and sizing, shown by for 2D row-wise:

```
perf stat -e cache-misses ./fractal

Performance counter stats for './fractal':

38,704      cache-misses:u

0.150822966 seconds time elapsed

0.518501000 seconds user
0.003898000 seconds sys
```

and 2D column-wise:

```
perf stat -e cache-misses ./fractal

Performance counter stats for './fractal':
```

```
32,170      cache-misses:u

0.149987987 seconds time elapsed

0.493800000 seconds user
0.000000000 seconds sys
```

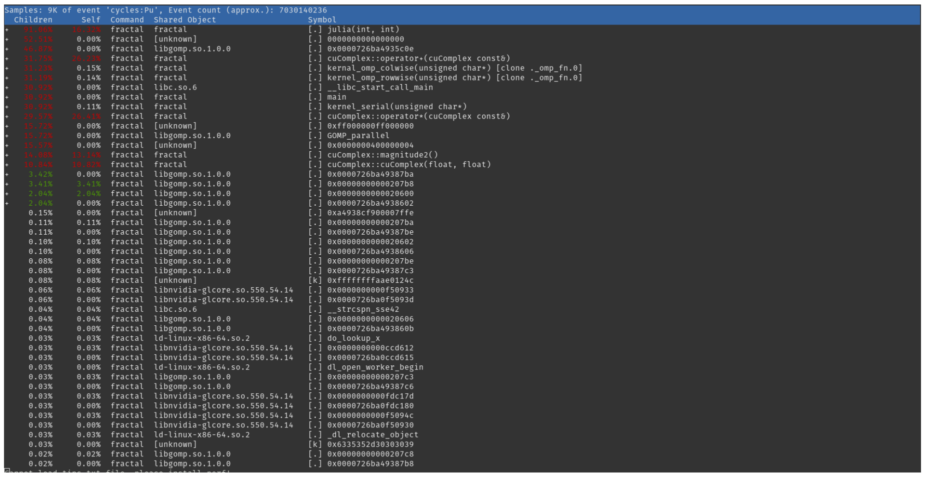
The difference is not much but as the program scales it becomes more apparent.

2 top performers

The 2 top performers are the 1D column-wise and 2D column wise. The 2 methods benefit from iterating over columns. This aligns with the row-major memory layout which increases the cache efficiency and reducing cache misses. The differences is that 1D column-wise divides the image into smaller chunks and each thread gets a contiguous set of columns. This gives us a rougher-grained parallelism but can lead to load imbalances if the DIM is not perfectly divisible by the number threads. The 2D column-wise on the other hand provides a finer-grained parallelism and work distribution. This can improve load balancing. The cache behaviour also plays a small role in this, since in 1D column-wise the threads work on long, contiguous lines of memory. With 2D column-wise the jump to the next block is relatively smaller which promotes cache-friendly patterns. They both outperform the other due to the fact that of memory alignment. Both implementations use sequential column-wise access, matching the row-major layout and enabling effective hardware pre-fetching and caching. They are also better load balanced for scalability, 2D column-wise often has a slight edge here due to the finer granularity of work load.

Increasing performance

Some further optimisations we can apply is some more loop unrolling. We can hand unroll the inner loop of the Julia function to process multiple pixels per iteration. We can also implement data alignments, this means that we can ensure that the start of each thread aligns with the cache boundaries. This can then increase performance by reducing partial cache line reads and writes - we have to pad the data for this to work. We can also further divide large column data blocks into smaller tiles that fit into the processors L1 or L2 cache which will then maximise cache reuses and minimises fetches from main memory. In the end the main bottle neck is the Julia function and its calls as seen below: we



```
Samples: 9K of event 'cycles:Pu', Event count (approx.): 7030140236
  Children      Self  Command  Shared Object                    Symbol
+   91.06%    16.32%  fractal  fractal                          [.] julia(int, int)
+   52.51%     0.00%  fractal  [unknown]                        [.] 0000000000000000
+   46.87%     0.00%  fractal  libgomp.so.1.0.0                 [.] 0x0000726ba4935c0e
+   31.75%    26.23%  fractal  fractal                          [.] cuComplex::operator+(cuComplex const&)
+   31.23%     0.15%  fractal  fractal                          [.] kernal_omp_colwise(unsigned char*) [clone ._omp_fn.0]
+   31.19%     0.14%  fractal  fractal                          [.] kernel_omp_rowwise(unsigned char*) [clone ._omp_fn.0]
+   30.92%     0.00%  fractal  libc.so.6                        [.] __libc_start_call_main
+   30.92%     0.00%  fractal  fractal                          [.] main
+   30.92%     0.11%  fractal  fractal                          [.] kernel_serial(unsigned char*)
+   29.57%    26.41%  fractal  fractal                          [.] cuComplex::operator*(cuComplex const&)
+   15.72%     0.00%  fractal  [unknown]                        [.] 0xff000000ff000000
+   15.72%     0.00%  fractal  libgomp.so.1.0.0                 [.] GOMP_parallel
+   15.57%     0.00%  fractal  [unknown]                        [.] 0x0000000400000004
+   14.08%    13.14%  fractal  fractal                          [.] cuComplex::magnitude2()
+   10.84%    10.82%  fractal  fractal                          [.] cuComplex::cuComplex(float, float)
+    3.42%     0.00%  fractal  libgomp.so.1.0.0                 [.] 0x0000726ba49387ba
+    3.41%     3.41%  fractal  libgomp.so.1.0.0                 [.] 0x00000000000207b8
+    2.04%     2.04%  fractal  libgomp.so.1.0.0                 [.] 0x0000000000020600
+    2.04%     0.00%  fractal  libgomp.so.1.0.0                 [.] 0x0000726ba4938602
     0.15%     0.00%  fractal  [unknown]                        [.] 0xa4938cf900007ffe
     0.11%     0.11%  fractal  libgomp.so.1.0.0                 [.] 0x00000000000207ba
     0.11%     0.00%  fractal  libgomp.so.1.0.0                 [.] 0x0000726ba49387be
     0.10%     0.10%  fractal  libgomp.so.1.0.0                 [.] 0x0000000000020602
     0.10%     0.00%  fractal  libgomp.so.1.0.0                 [.] 0x0000726ba4938606
     0.08%     0.08%  fractal  libgomp.so.1.0.0                 [.] 0x00000000000207be
     0.08%     0.00%  fractal  libgomp.so.1.0.0                 [.] 0x0000726ba49387c3
     0.08%     0.08%  fractal  [unknown]                        [k] 0xffffffffaae0124c
     0.06%     0.06%  fractal  libnvidia-glcore.so.550.54.14    [.] 0x0000000000f50933
     0.06%     0.00%  fractal  libnvidia-glcore.so.550.54.14    [.] 0x0000726ba0f5093d
     0.04%     0.04%  fractal  libc.so.6                        [.] __strcspn_sse42
     0.04%     0.04%  fractal  libgomp.so.1.0.0                 [.] 0x0000000000020606
     0.04%     0.00%  fractal  libgomp.so.1.0.0                 [.] 0x0000726ba493860b
     0.03%     0.03%  fractal  ld-linux-x86-64.so.2             [.] do_lookup_x
     0.03%     0.03%  fractal  libnvidia-glcore.so.550.54.14    [.] 0x0000000000ccd612
     0.03%     0.00%  fractal  libnvidia-glcore.so.550.54.14    [.] 0x0000726ba0ccd615
     0.03%     0.00%  fractal  ld-linux-x86-64.so.2             [.] dl_open_worker_begin
     0.03%     0.03%  fractal  libgomp.so.1.0.0                 [.] 0x00000000000207c3
     0.03%     0.00%  fractal  libgomp.so.1.0.0                 [.] 0x0000726ba49387c6
     0.03%     0.03%  fractal  libnvidia-glcore.so.550.54.14    [.] 0x0000000000fdc17d
     0.03%     0.00%  fractal  libnvidia-glcore.so.550.54.14    [.] 0x0000726ba0fdc180
     0.03%     0.03%  fractal  libnvidia-glcore.so.550.54.14    [.] 0x0000000000f5094c
     0.03%     0.00%  fractal  libnvidia-glcore.so.550.54.14    [.] 0x0000726ba0f50930
     0.03%     0.03%  fractal  ld-linux-x86-64.so.2             [.] _dl_relocate_object
     0.03%     0.00%  fractal  [unknown]                        [k] 0x6335352d30303039
     0.02%     0.02%  fractal  libgomp.so.1.0.0                 [.] 0x00000000000207c8
     0.02%     0.00%  fractal  libgomp.so.1.0.0                 [.] 0x0000726ba49387b8
```

Figure 6: Main usage of memory

could find ways to optimise the Julia function and its calls to further improve performance.