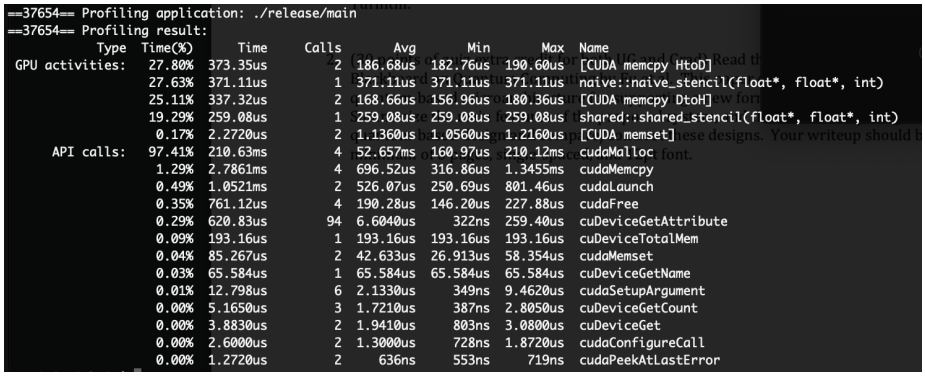
Memory Reuse through shared memory in CUDA

The stenciling program provided exhibits reuse of memory in 3 dimenions meaning each thread will be re-reading the same block of memory multiple times in order to carry out the same computation. This can be optimized through the use of caching in shared memory. Shared memory exists as a cache within each SM and therefore, is much less expensive to access than the off chip global memory. By having each thread read their data into shared memory and performing future access from shared memory rather than global memory, faster access speeds can be achieved.

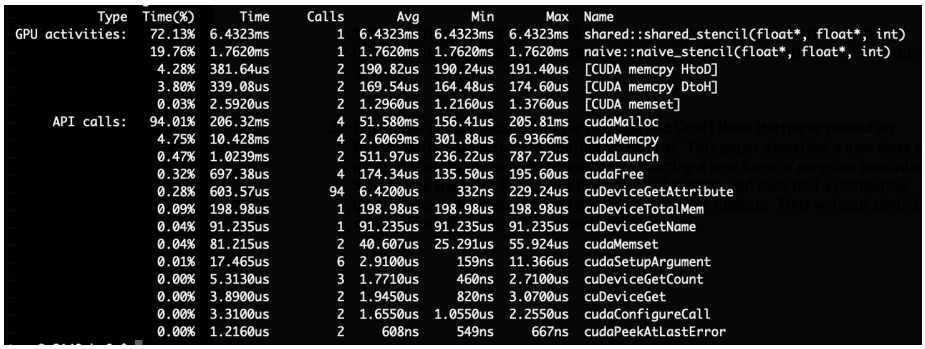
It is important to note that loading into shared memory is not free, especially around the fringes of computation. At the edge, a number of branch instructions take place which cause latencies to be exposed in the program. In our example, we have a small stencil size (6 reads, 2 in each direction) which may be realtively small compared to the branch overheads.

From the timing however, we still see that the `shared_stencil` implementation runs slightly faster than the naive implementaiton.



```
==37654== Profiling application: ./release/main
==37654== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   27.80%  373.35us         2  186.68us  182.76us  190.60us  [CUDA memcpy HtoD]
                   27.63%  371.11us         1  371.11us  371.11us  371.11us  naive::naive_stencil(float*, float*, int)
                   25.11%  337.32us         2  168.66us  156.96us  180.36us  [CUDA memcpy DtoH]
                   19.29%  259.08us         1  259.08us  259.08us  259.08us  shared::shared_stencil(float*, float*, int)
                    0.17%  2.2720us         2  1.1360us  1.0560us  1.2160us  [CUDA memset]
      API calls:   97.41%  210.63ms         4  52.657ms  160.97us  210.12ms  cudaMalloc
                    1.29%  2.7861ms         4  696.52us  316.86us  1.3455ms  cudaMemcpy
                    0.49%  1.0521ms         2  526.07us  250.69us  801.46us  cudaLaunch
                    0.35%  761.12us         4  190.28us  146.20us  227.88us  cudaFree
                    0.29%  620.83us        94  6.6040us     322ns  259.40us  cuDeviceGetAttribute
                    0.09%  193.16us         1  193.16us  193.16us  193.16us  cuDeviceTotalMem
                    0.04%  85.267us         2  42.633us  26.913us  58.354us  cudaMemset
                    0.03%  65.584us         1  65.584us  65.584us  65.584us  cuDeviceGetName
                    0.01%  12.798us         6  2.1330us     349ns  9.4620us  cudaSetupArgument
                    0.00%  5.1650us         3  1.7210us     387ns  2.8050us  cuDeviceGetCount
                    0.00%  3.8830us         2  1.9410us     803ns  3.0800us  cuDeviceGet
                    0.00%  2.6000us         2  1.3000us     728ns  1.8720us  cudaConfigureCall
                    0.00%  1.2720us         2     636ns     553ns     719ns  cudaPeekAtLastError
```

The use of tiling, seperate from shared memory is also interesting to explore. The above nvprof shows the timing from a 16x16x1 tiled implementation where tiles exist in 2 dimensions. The profiler shows that the average timing for these the two implemenations was on the order of 300-400 us. By comparison, tiling with a tile size of 1x1x1 (i.e. no tiling) runs on the order of 7ms!. A significant slow down.



Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	72.13%	6.4323ms	1	6.4323ms	6.4323ms	6.4323ms	shared::shared_stencil(float*, float*, int)
	19.76%	1.7620ms	1	1.7620ms	1.7620ms	1.7620ms	naive::naive_stencil(float*, float*, int)
	4.28%	381.64us	2	190.82us	190.24us	191.40us	[CUDA memcpy HtoD]
	3.80%	339.08us	2	169.54us	164.48us	174.60us	[CUDA memcpy DtoH]
	0.03%	2.5920us	2	1.2960us	1.2160us	1.3760us	[CUDA memset]
API calls:	94.01%	206.32ms	4	51.580ms	156.41us	205.81ms	cudaMalloc
	4.75%	10.428ms	4	2.6069ms	301.88us	6.9366ms	cudaMemcpy
	0.47%	1.0239ms	2	511.97us	236.22us	787.72us	cudaLaunch
	0.32%	697.38us	4	174.34us	135.50us	195.60us	cudaFree
	0.28%	603.57us	94	6.4200us	332ns	229.24us	cuDeviceGetAttribute
	0.09%	198.98us	1	198.98us	198.98us	198.98us	cuDeviceTotalMem
	0.04%	91.235us	1	91.235us	91.235us	91.235us	cuDeviceGetName
	0.04%	81.215us	2	40.607us	25.291us	55.924us	cudaMemset
	0.01%	17.465us	6	2.9100us	159ns	11.366us	cudaSetupArgument
	0.00%	5.3130us	3	1.7710us	460ns	2.7100us	cuDeviceGetCount
	0.00%	3.8900us	2	1.9450us	820ns	3.0700us	cuDeviceGet
	0.00%	3.3100us	2	1.6550us	1.0550us	2.2550us	cudaConfigureCall
	0.00%	1.2160us	2	608ns	549ns	667ns	cudaPeekAtLastError

Tiling through the use of large block sizes helps speed up the runtime for a number of reasons. Individual blocks correspond to a physical SM. Since SMs run the batches of threads groups of 32 (a unit known as a warp), running a block with less than 32 threads is just an inefficient use of hardware. Furthermore, when programs attempt to access similar pieces of data, (temporal locality), that data is primed in the data pipeline and left in the cache.