Optimisation of the naive matrix multiplication and heat conduction kernels pt. 1 – Jacek Jagosz

When having a working CUDA C code, there are 3 easy ways to make it run faster:

1. Optimising grid size so the occupancy of threads is as high as possible – CUDA Occupancy Calculator can be helpful here
2. When the current implementation might suffer from gaps in the data access, where the next threads don't work on data that is physically close to each other, then coalessing the memory is helpful. One way to achieve that is tiling
3. When the performance is limited by a large amounts of page faults and the data size we are working on at the same size is not too big memory prefetching or even manually moving the data instead of using managed memory should bring big performance gains.

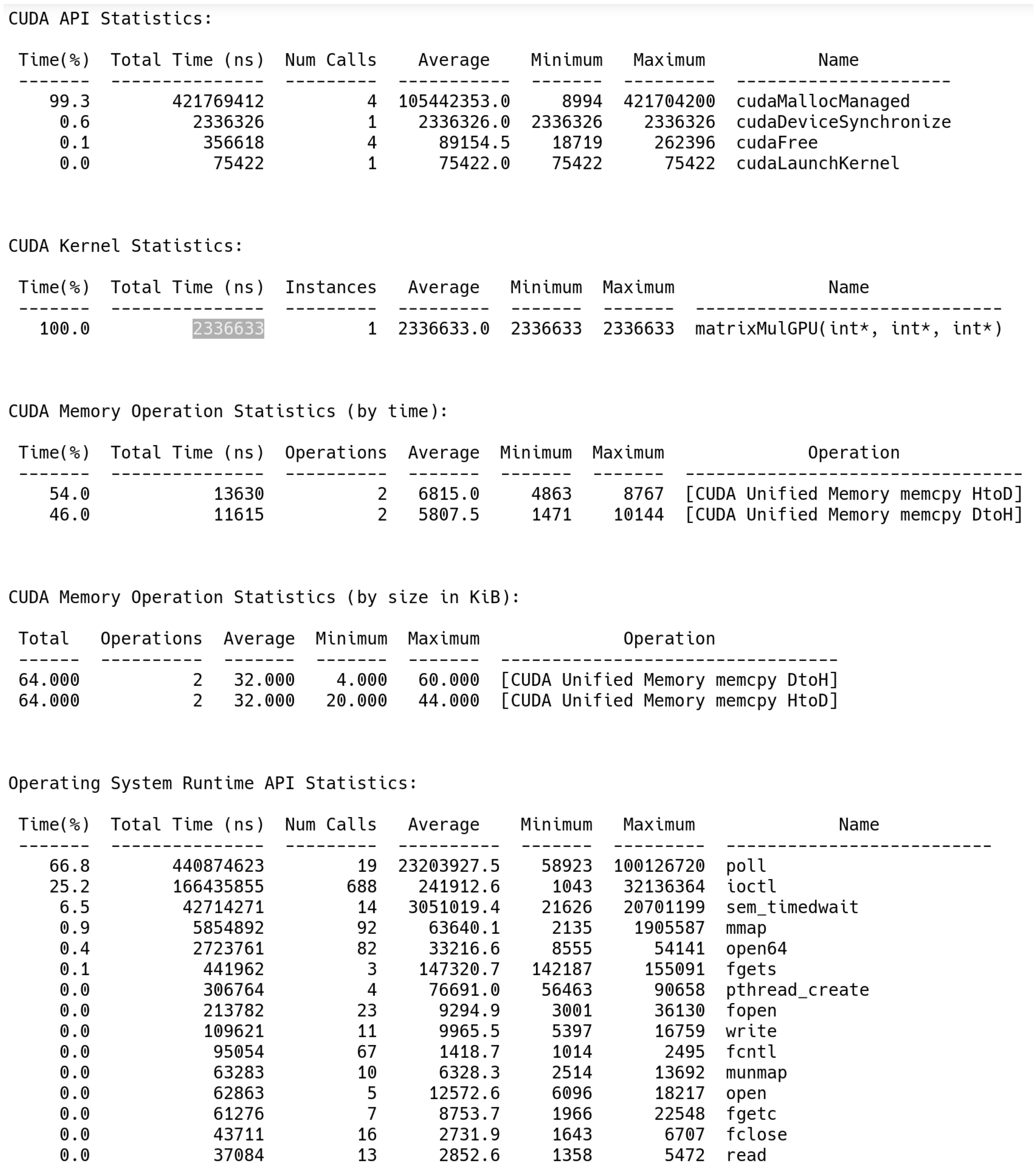
In this report we will focus on the first, most basic optimisation.
De device we are using is:

```
Device name: Tesla T4
Number of SMs: 40
Compute Capability Major: 7
Compute Capability Minor: 5
Warp Size: 32
```

Tesla T4 is a Turing GPU which uses the same die as GeForce RTX 2080 SUPER, but somewhat cut-down. Each SM allows for 64 threads, so 2 warps. Shared memory capacity per SM is 64KB. This will all be important later when analysing the performance differences

The calculations were performed on the AWS cloud shared with us from Nvidia for this course. The starting point for each was the solution and then I tweaked the block dimensions and used `nsys` to check the kernel performance.

```
!nsys profile --stats=true matrix-multiply-2d
```



```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls    Average    Minimum   Maximum          Name
 -------  ---------------  ---------  -----------  -------  ---------  ---------------------
    99.3        421769412          4  105442353.0     8994  421704200  cudaMallocManaged
     0.6          2336326          1    2336326.0  2336326    2336326  cudaDeviceSynchronize
     0.1           356618          4      89154.5    18719     262396  cudaFree
     0.0            75422          1      75422.0    75422      75422  cudaLaunchKernel



CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances   Average   Minimum  Maximum              Name
 -------  ---------------  ---------  ---------  -------  -------  ------------------------------
   100.0          2336633          1  2336633.0  2336633  2336633  matrixMulGPU(int*, int*, int*)



CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations  Average  Minimum  Maximum             Operation
 -------  ---------------  ----------  -------  -------  -------  ---------------------------------
    54.0            13630           2   6815.0     4863     8767  [CUDA Unified Memory memcpy HtoD]
    46.0            11615           2   5807.5     1471    10144  [CUDA Unified Memory memcpy DtoH]



CUDA Memory Operation Statistics (by size in KiB):

 Total   Operations  Average  Minimum  Maximum             Operation
 ------  ----------  -------  -------  -------  ---------------------------------
 64.000           2   32.000    4.000   60.000  [CUDA Unified Memory memcpy DtoH]
 64.000           2   32.000   20.000   44.000  [CUDA Unified Memory memcpy HtoD]



Operating System Runtime API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum             Name
 -------  ---------------  ---------  ----------  -------  ---------  --------------------------
    66.8        440874623         19  23203927.5    58923  100126720  poll
    25.2        166435855        688    241912.6     1043   32136364  ioctl
     6.5         42714271         14   3051019.4    21626   20701199  sem_timedwait
     0.9          5854892         92     63640.1     2135    1905587  mmap
     0.4          2723761         82     33216.6     8555      54141  open64
     0.1           441962          3    147320.7   142187     155091  fgets
     0.0           306764          4     76691.0    56463      90658  pthread_create
     0.0           213782         23      9294.9     3001      36130  fopen
     0.0           109621         11      9965.5     5397      16759  write
     0.0            95054         67      1418.7     1014       2495  fcntl
     0.0            63283         10      6328.3     2514      13692  munmap
     0.0            62863          5     12572.6     6096      18217  open
     0.0            61276          7      8753.7     1966      22548  fgetc
     0.0            43711         16      2731.9     1643       6707  fclose
     0.0            37084         13      2852.6     1358       5472  read
```

2D Matrix multiplication kernel total time:
`533 396ns 32 x 32 block of threads`
`321 913ns` `- 16 x 16 block threads - default`
`345 208ns 8 x 8 block threads`
`Data size: 2x 16 384`

As expected the default configuration was the most performant. Each SM got assigned 4 times as many threads as it can handle at once, but this is because a lot of time is spent waiting on data, so assigning

more threads increases SM occupation, to a certain point of course, as we can see by 32 x 32 setup leading to regression in performance. My assumption is that no the whole data can fit in SM's shared memory at this point.
But because the data access patter is not linear here we would benefit considerably by tiling, and then I presume not as many threads would be needed to achieve maximum performance, as data access would be more optimised.

```
Thermal conductivity kernel total time:
4 534 028ns 32 x 32
4 170 747ns 32 x 16 - default
2 260 498ns 16 x 16
2 202 122ns 16 x 8
1 863 674ns 8 x 8
1 991 376ns 8 x 4
2 165 206ns 4 x 4
Data size: 80 000
```

This kernel's performance turned out to be much more interesting. Not only the default configuration was far from the most performant, but also the most performant one had exactly the same amount of threads per SM as it can natively support. So clearly here over-assigning threads doesn't lead to more SM utilisation. Even assigning half the threads the SM can handle didn't considerably reduce the performance. My guess is that here we are dealing with dataset too big to fit in the shared memory as data size is 2x 80 000 is more than 64KB, so we are cache limited.
Because we have so much data adding more threads won't improve performance as we are running out of this local memory really quickly.

Conclusion: It is beneficial to increase thread number per SM to increase SM utilisation, but each algorythm has its own sweet spot, and adding to many will cause regressions in performance. Also some knowledge about the GPU is beneficial at creating an educated guess at what numbers will work or not.