Optimisation of the naive matrix multiplication pt. 2 – 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 exercise we have a piece of code that already contains all the above optimisations, so I will experiment with data types to see how they affect the performance. The starting point is `cuda.opencl-9.11.2022/workdir_2022/matrixMul` project
De device we were using previously in AWS cloud was:

```
Device name: Tesla T4
Number of SMs: 40
Compute Capability Major/Minor: 7.5
Warp Size: 32
VRAM Size: 16GB
```

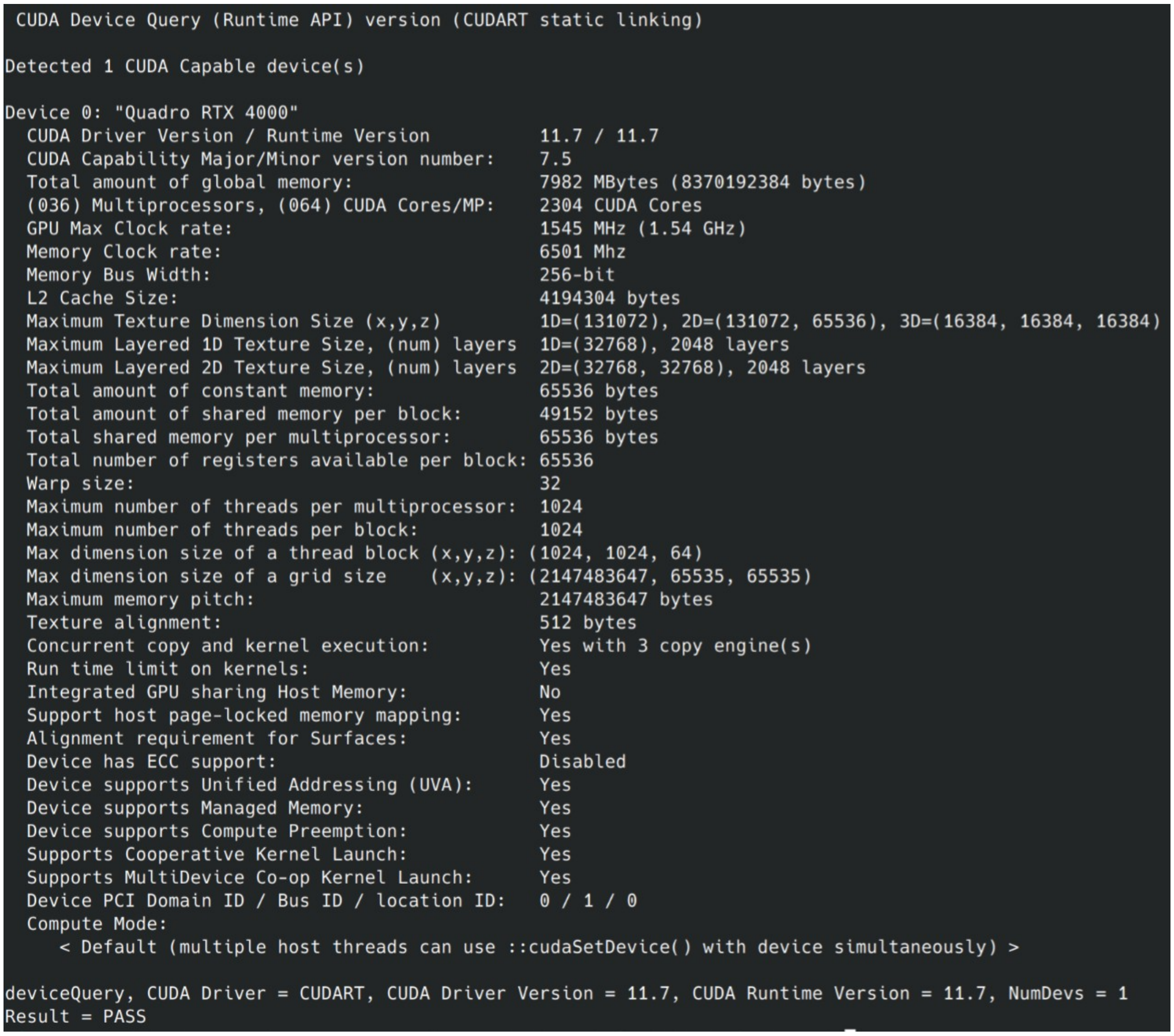
And the GPU our lab computers are equipped with is:

```
Device name: Quadro RTX 4000
Number of SMs: 36
Compute Capability Major/Minor: 7.5
Warp Size: 32
VRAM Size: 8GB
```

Both GPUs are not only use the same architecture, but both use the same silicon, that can be also seen in the RTX 2080 Super. Both have disabled SMs compared to the desktop GPU, the Quadro being cut down more. Tesla T4 also had twice the video memory than the consumer GPU, while quadro uses the desktop configuration. But the algorythm we are analysing today operates on way too small datasets for the VRAM capacity to make any difference. What is more important is that all the other parameters per the SM are exactly the same, which means all the characteristics we measured in the last report should be the same for this GPU as well.
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.

Conclusions from the last report was that 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. When the threads are mostly waiting for data increasing number of threads is helpful, while if each thread is operating on a lot of data then the size of shared memory starts to be the limiting factor and reducing number of threads leads to speedup.



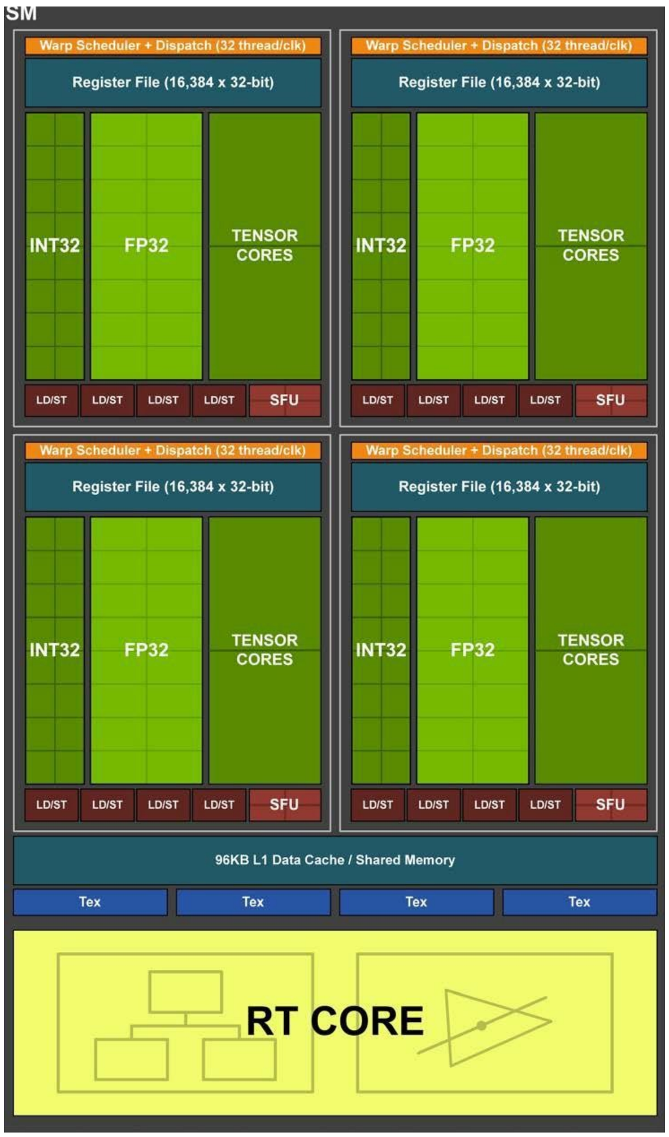
```
 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Quadro RTX 4000"
  CUDA Driver Version / Runtime Version          11.7 / 11.7
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 7982 MBytes (8370192384 bytes)
  (036) Multiprocessors, (064) CUDA Cores/MP:    2304 CUDA Cores
  GPU Max Clock rate:                            1545 MHz (1.54 GHz)
  Memory Clock rate:                             6501 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 4194304 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.7, CUDA Runtime Version = 11.7, NumDevs = 1
Result = PASS
```

We will start with the most optimised algorythm, the tiled one. In it we don't directly choose number of threads per block, as it is tied to the tile dimensions, and the whole generated matrix dimensions. So the only options we have without changing the algorythm are 32x32 and 16x16 tile and blocks of thread dimensions, with the bigger

```
Blocksize = 32: Perf= 583.90 GFlop/s, Time= 0.224 msec, WorkgroupSize= 1024
threads/block, MatrixA(320,320), MatrixB(640,320)
Blocksize = 16: Perf= 444.04 GFlop/s, Time= 0.037 msec, WorkgroupSize= 256
threads/block, MatrixA(160,160), MatrixB(320,160)
```

And while with the naive implementation the 16x16 was the most performant and 32x32 faced a significant performance regression, here the larger one gets noticeably moew GFlop/s.
I assume this is because the shared memory is allocated manually in this algorythm, as well as thanks to tiling only the tile needs to be allocated for each thread, which means the precious local memory is

used much more frugally, so more threads can be used in each SM without running out of local memory. But it could also mean the larger tile size is more optimal, or the larger datasize means there is some constant overhead, and the longer workload takes the less fraction of total time it is taking up. Although I find the last example the least likely.
From now on we will use the faster 32 blocksize as a base.


SM
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
Tex
RT CORE

Next I decided to check how this architecture can handle multiplication of different data types. Turing support concurrent FP 32 and INT 32, but I wonder about their performance.

```
583.90 GFlop/s – fp32 – default
583.99 GFlop/s – int32
```

The int32 and fp32 have the same performance, which matches the documentation.
Meanwhile the fp64 we can expect to be worse, not just because of the theoretical speed the hardware could do, but GPU makers many years ago started to limit its performance on consumer cards, so companies who need those levels of precision have to buy pro-tier GPUs.
he FP64 TFLOP rate is 1/32nd the TFLOP rate of FP32 operations on consumer cards[1], including QUADROs.

```
158.78 GFlop/s – fp64
```

But for some reason the calculation is only 3.67 times slower. I assume that is because the multiplication takes up really small part of the whole kernel runtime, the rest is the overhead. So increasing the accuracy doesn’t increase the runtime as much as could be expected by just reading the GPU specification. Because of the same reason reducing the precision to int16 fp16 or even lower will only bring small improvement.

```
459.94 GFlop/s – int8
```

After changing the algorythm to use int8 it actually got slower, while according to the documentation it should get 16 times faster. I don’t have a clear answer why that happened. Maybe because this operation should be performed on the Tensor cores, and those are not used by default, so instead all data has to be converted to int32 and then calculated the usual way? Or CUDA SDK has a problem with builtin C data type int8_t and some other library needs to be used?

Conclusion: Common data types FP32 and INT32 are multiplied at the same speed. Even though FP64 rate is 32 times slower than those, in our case the kernel run less than 4 times slower. That is because the multiplication operation is a small part of the whole runtime. Which means increasing accuracy doesn’t bring as big of a performance hit as could first be expected. Moreover I encountered a performance regression when lowering accuracy to INT8.

Sources:

1. Turing SM diagram and info about different data types operation rates - https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/