# Matrix  Transpose
Ali yazdanpanah
9831116

### Overview

The focus of this homework is parrallelizing transpose function of matrices with single and double precision values. First I will go through serial impelementation. Then I'll get to parallel implementation accelerated using NVIDIA K80 GPU provided by google colab. Serial version is also ran on google colab.

## Serial

First lets take a look at the structure of the code.

### Structure

Three files where used in implementing serial version:

- matrix.h:
  - This is the main header file that provides required includes and defines matrix structs used in both parallel and serial version. Both double and integer matrices are defined like this:

    ```
    typedef struct {
    int rows;
    int cols;
    double * data;
    } double_matrix;

    typedef struct {
    int rows;
    int cols;
    int * data;
    } int_matrix;
    ```

- transpose.c
  - This file provides functions required for serial version (Some functions are used in parallel version too), description of each function is listed below:
    - `double_matrix * newDoubleMatrix(int rows, int cols):`
      - This functions returns a matrix pointer with double values and takes number of rows and columns as input. Values are assigned randomly
    - `int_matrix * newIntMatrix(int rows, int cols):`
      - This functions returns a matrix pointer with integer values and takes number of rows and columns as input. Values are assigned randomly
    - `#define ELEM(mtx, row, col) \mtx->data[(col-1) * mtx->rows + (row-1)]:`
      - Pointer to element in matrix by row and column location
    - `int printDoubleMatrix(double_matrix * mtx):`
      - Used for printing a matrix with double values
    - `int printIntMatrix(int_matrix * mtx):`
      - Used for printing a matrix with int values
    - `int transposeDouble(double_matrix * in, double_matrix * out):`
      - Writes the transpose of a matrix with double values in into matrix out. Returns 0 if successful, -1 if either in or out is NULL, and -2 if the dimensions of in and out are incompatible.
    - `int transposeInt(int_matrix * in, int_matrix * out):`
      - Writes the transpose of a matrix with int values in into matrix out. Returns 0 if successful, -1 if either in or out is NULL, and -2 if the dimensions of in and out are incompatible.
    - `int setDoubleElement(double_matrix * mtx, int row, int col, double val):`
      - Sets the (row, col) element of double mtx to val. Returns 0 if successful, -1 if mtx is NULL, and -2 if row or col are outside of the dimensions of mtx.
    - `int setIntElement(int_matrix * mtx, int row, int col, int val):`
      - Sets the (row, col) element of int mtx to val. Returns 0 if successful, -1 if mtx is NULL, and -2 if row or col are outside of the dimensions of mtx.
    - `double_matrix * copyDoubleMatrix(double_matrix * mtx):`
      - Copies a double matrix. Returns NULL if mtx is NULL.
    - `int_matrix * copyIntMatrix(int_matrix * mtx):`
      - Copies an int matrix. Returns NULL if mtx is NULL.
    - `int deleteDoubleMatrix(double_matrix * mtx):`
      - Deletes a double matrix. Returns 0 if successful and -1 if mtx is NULL.
    - `int deleteIntMatrix(int_matrix * mtx):`
      - Deletes an int matrix. Returns 0 if successful and -1 if mtx is NULL.

- Makefile
  - Makefile is used to compile the code.

- Chart.py
  - A simple python script which generates charts using matploitlib

## Runtime

Now lets walk through running the serial version using google colab.
There is a whole section in google colab notebook dedicated to seeing CPU specifications, Also the detailed specifications of CPU are liseted below:

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 85
model name      : Intel(R) Xeon(R) CPU @ 2.00GHz
stepping        : 3
microcode       : 0x1
cpu MHz         : 2000.172
cache size      : 39424 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx
pdpe1gb rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2
x2apic movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch invpcid_single ssbd ibrs ibpb stibp fsgsbase tsc_adjust
bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx avx512f avx512dq rdseed adx smap clflushopt clwb avx512cd avx512bw avx512vl xsaveopt xsavec
xgetbv1 xsaves arat md_clear arch_capabilities
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs taa itlb_multihit
bogomips        : 4000.34
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:

processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 85
model name      : Intel(R) Xeon(R) CPU @ 2.00GHz
stepping        : 3
microcode       : 0x1
cpu MHz         : 2000.172
cache size      : 39424 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 1
apicid          : 1
initial apicid  : 1
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx
pdpe1gb rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2
x2apic movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch invpcid_single ssbd ibrs ibpb stibp fsgsbase tsc_adjust
bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx avx512f avx512dq rdseed adx smap clflushopt clwb avx512cd avx512bw avx512vl xsaveopt xsavec
xgetbv1 xsaves arat md_clear arch_capabilities
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs taa itlb_multihit
bogomips        : 4000.34
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

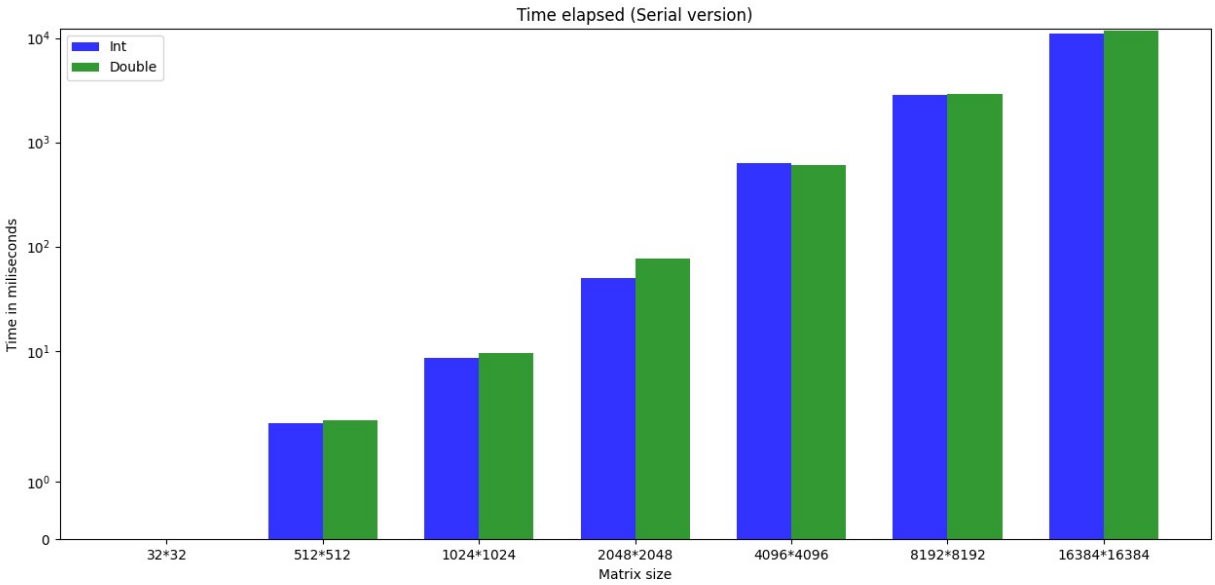
There is a whole section in google colab notebook dedicated to seeing Memory specifications, Also the detailed specifications of Memory are liseted below:

```
MemTotal:       13333540 kB
MemFree:         1002948 kB
MemAvailable:   12428748 kB
Buffers:          140972 kB
Cached:         11101676 kB
SwapCached:            0 kB
Active:           880100 kB
Inactive:       10806316 kB
Active(anon):     417400 kB
Inactive(anon):      340 kB
Active(file):     462700 kB
Inactive(file): 10805976 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:              1256 kB
Writeback:             0 kB
AnonPages:        443540 kB
Mapped:           219972 kB
Shmem:               948 kB
Slab:             526376 kB
SReclaimable:     481260 kB
SUnreclaim:        45116 kB
KernelStack:        3816 kB
PageTables:         6284 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:     6666768 kB
Committed_AS:    2551564 kB
VmallocTotal:   34359738367 kB
VmallocUsed:           0 kB
VmallocChunk:          0 kB
Percpu:              912 kB
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
ShmemPmdMapped:        0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:               0 kB
DirectMap4k:      132284 kB
DirectMap2M:     7206912 kB
DirectMap1G:     8388608 kB
```

Now Lets take a look at profiling details of 16384*16384 serial version using gprof. (Due to low runtime of the smaller matrices versions the larger matrices provide better insights):

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00  235.23                 main [1]
              117.25    0.00      10/10          transposeDouble [2]
              110.72    0.00      10/10          transposeInt [3]
                4.29    0.00       2/2           newDoubleMatrix [4]
                2.97    0.00       2/2           newIntMatrix [5]
                0.00    0.00       2/2           deleteDoubleMatrix [6]
                0.00    0.00       2/2           deleteIntMatrix [7]
-----------------------------------------------
              117.25    0.00      10/10          main [1]
[2]     49.8  117.25    0.00      10         transposeDouble [2]
-----------------------------------------------
              110.72    0.00      10/10          main [1]
[3]     47.1  110.72    0.00      10         transposeInt [3]
-----------------------------------------------
                4.29    0.00       2/2           main [1]
[4]      1.8    4.29    0.00       2         newDoubleMatrix [4]
-----------------------------------------------
                2.97    0.00       2/2           main [1]
[5]      1.3    2.97    0.00       2         newIntMatrix [5]
-----------------------------------------------
                0.00    0.00       2/2           main [1]
[6]      0.0    0.00    0.00       2         deleteDoubleMatrix [6]
-----------------------------------------------
                0.00    0.00       2/2           main [1]
[7]      0.0    0.00    0.00       2         deleteIntMatrix [7]
-----------------------------------------------
```

As you can see 49.8% of the total runtime is dedicated to transpose int matrix and 47.1% to transpose double matrix. And the goal of the homework is to parallelize these two functions. Also a detailed chart of the runtimes for each matrix size is provided below:



# Parallel

First lets take a look at the structure of the code.

### Structure

Three files where used in implementing serial version:

- matrix.h:
  - This is the main header file that provides required includes and defines matrix structs used in both parallel and serial version. Both double and integer matrices are defined like this:

    ```
    typedef struct {
    int rows;
    int cols;
    double * data;
    } double_matrix;

    typedef struct {
    int rows;
    int cols;
    int * data;
    } int_matrix;
    ```

- transpose.cu
  - This file provides functions required for serial version (Some functions are used in parallel version too), description of each function is listed below:
    - `double_matrix * newDoubleMatrix(int rows, int cols);`
      - This functions returns a matrix pointer with double values and takes number of rows and columns as input. Values are assigned randomly
    - `int_matrix * newIntMatrix(int rows, int cols);`
      - This functions returns a matrix pointer with integer values and takes number of rows and columns as input. Values are assigned randomly
    - `#define ELEM(mtx, row, col) \mtx->data[(col-1) * mtx->rows + (row-1)];`
      - Pointer to element in matrix by row and column location
    - `int printDoubleMatrix(double_matrix * mtx);`
      - Used for printing a matrix with double values
    - `int printIntMatrix(int_matrix * mtx);`
      - Used for printing a matrix with int values
    - `int transposeDouble(double_matrix * in, double_matrix * out);`
      - Writes the transpose of a matrix with double values in into matrix out. Returns 0 if successful, -1 if either in or out is NULL, and -2 if the dimensions of in and out are incompatible.
    - `int transposeInt(int_matrix * in, int_matrix * out);`
      - Writes the transpose of a matrix with int values in into matrix out. Returns 0 if successful, -1 if either in or out is NULL, and -2 if the dimensions of in and out are incompatible.
    - `int setDoubleElement(double_matrix * mtx, int row, int col, double val);`
      - Sets the (row, col) element of double mtx to val. Returns 0 if successful, -1 if mtx is NULL, and -2 if row or col are outside of the dimensions of mtx.
    - `int setIntElement(int_matrix * mtx, int row, int col, int val);`
      - Sets the (row, col) element of int mtx to val. Returns 0 if successful, -1 if mtx is NULL, and -2 if row or col are outside of the dimensions of mtx.
    - `double_matrix * copyDoubleMatrix(double_matrix * mtx);`
      - Copies a double matrix. Returns NULL if mtx is NULL.
    - `int_matrix * copyIntMatrix(int_matrix * mtx);`
      - Copies an int matrix. Returns NULL if mtx is NULL.
    - `int deleteDoubleMatrix(double_matrix * mtx);`
      - Deletes a double matrix. Returns 0 if successful and -1 if mtx is NULL.
    - `int deleteIntMatrix(int_matrix * mtx);`
      - Deletes an int matrix. Returns 0 if successful and -1 if mtx is NULL.

- `__global__ void transposeDiagonalInt(int *odata, int *idata, int width,int height):`
  - Kernel to compute transpose of a given int matrix in form of an array
- `__global__ void transposeDiagonalDouble(double *odata, double *idata, int width,int height):`
  - Kernel to compute transpose of a given double matrix in form of an array

- Makefile
  - Makefile is used to compile the code.

**Runtime**

Now lets walk through running the seria version using google colab.
There is a whole section in google colab notebook dedicated to seeing GPU specifications, Also the detailed specifications of GPU are liseted below:

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.82       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   39C    P0    27W / 250W |      0MiB / 16280MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

There is a whole section in google colab notebook dedicated to seeing Memory specifications, Also the detailed specifications of Memory are liseted below:

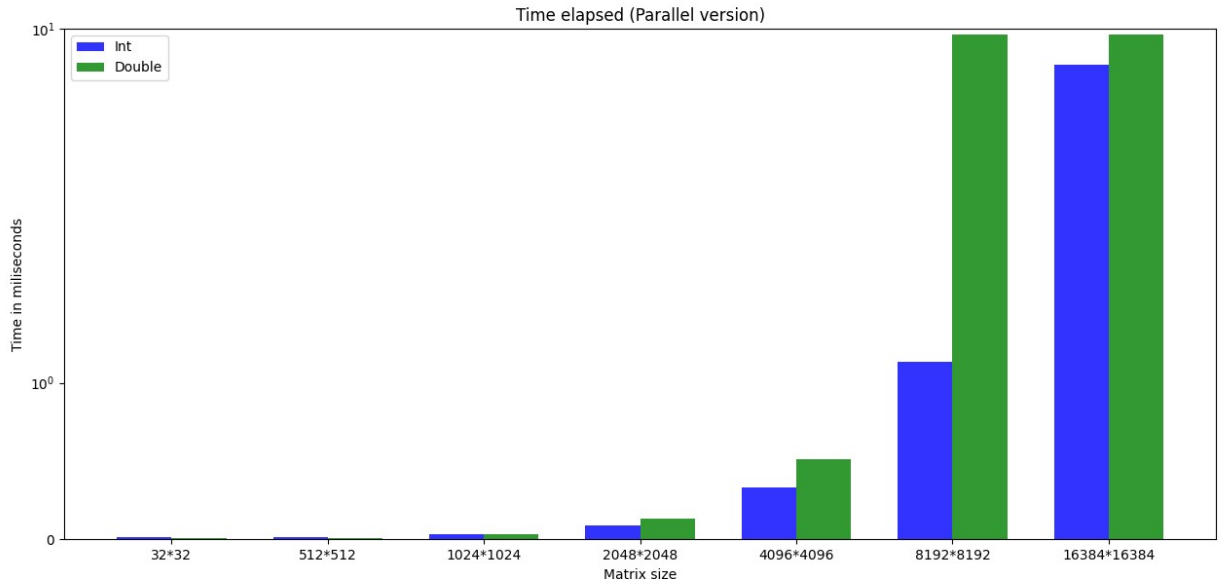
```
MemTotal:       13333540 kB
MemFree:         1002948 kB
MemAvailable:   12428748 kB
Buffers:          140972 kB
Cached:         11101676 kB
SwapCached:            0 kB
Active:           880100 kB
Inactive:       10806316 kB
Active(anon):     417400 kB
Inactive(anon):      340 kB
Active(file):     462700 kB
Inactive(file): 10805976 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:              1256 kB
Writeback:             0 kB
AnonPages:        443540 kB
Mapped:           219972 kB
Shmem:               948 kB
Slab:             526376 kB
SReclaimable:     481260 kB
SUnreclaim:        45116 kB
KernelStack:        3816 kB
PageTables:         6284 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:     6666768 kB
Committed_AS:    2551564 kB
VmallocTotal:   34359738367 kB
VmallocUsed:           0 kB
VmallocChunk:          0 kB
Percpu:              912 kB
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
ShmemPmdMapped:        0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:               0 kB
DirectMap4k:      132284 kB
DirectMap2M:     7206912 kB
DirectMap1G:     8388608 kB
```

Now Lets take a look at profiling details of 16384*16384 serial version using gprof. (Due to low runtime of the smaller matrices versions the larger matrices provide better details):

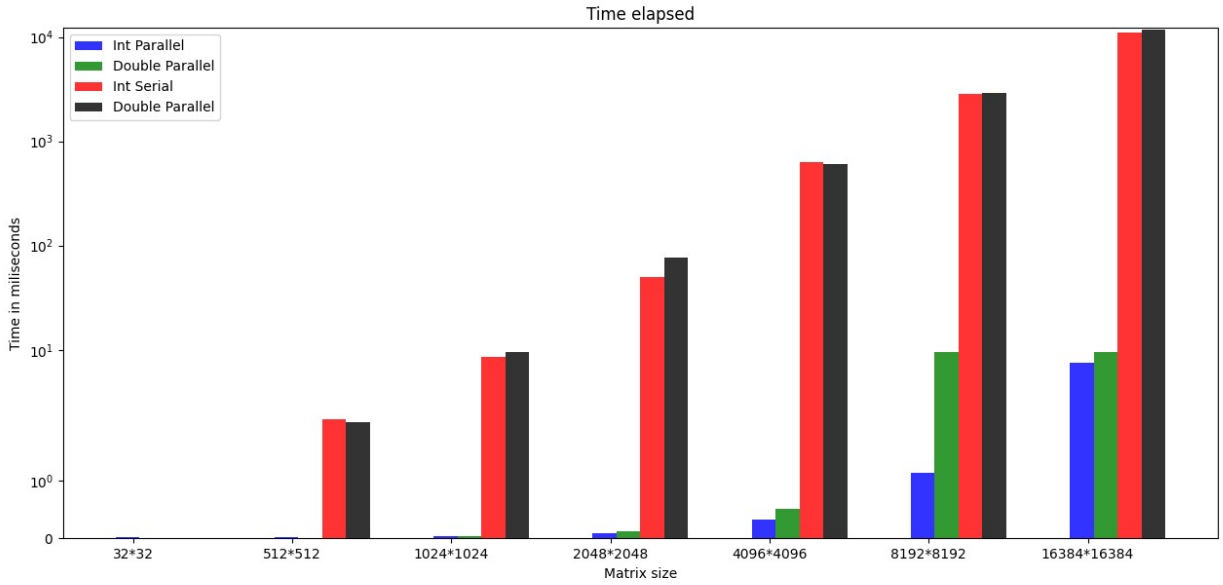
```
==12580== Profiling application: ./transpose-parallel 16384 16384
==12580== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput  SrcMemType  DstMemType
Device   Context    Stream  Name
379.21ms  171.07ms                    -               -         -         -         -  1.0000GB  5.8454GB/s    Pageable      Device  Tesla
P100-PCIE         1         7  [CUDA memcpy HtoD]
555.20ms  368.89ms                    -               -         -         -         -  2.0000GB  5.4217GB/s    Pageable      Device  Tesla
P100-PCIE         1         7  [CUDA memcpy HtoD]
924.10ms  7.4604ms        (1024 1024 1)       (16 16 1)        11  1.0625KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalInt(int*, int*, int, int) [113]
931.56ms  7.4542ms        (1024 1024 1)       (16 16 1)        11  1.0625KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalInt(int*, int*, int, int) [114]
939.02ms  7.4531ms        (1024 1024 1)       (16 16 1)        11  1.0625KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalInt(int*, int*, int, int) [115]
946.47ms  7.4488ms        (1024 1024 1)       (16 16 1)        11  1.0625KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalInt(int*, int*, int, int) [116]
953.92ms  7.4550ms        (1024 1024 1)       (16 16 1)        11  1.0625KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalInt(int*, int*, int, int) [117]
961.38ms  7.4516ms        (1024 1024 1)       (16 16 1)        11  1.0625KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalInt(int*, int*, int, int) [118]
968.83ms  7.4502ms        (1024 1024 1)       (16 16 1)        11  1.0625KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalInt(int*, int*, int, int) [119]
976.28ms  7.4510ms        (1024 1024 1)       (16 16 1)        11  1.0625KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalInt(int*, int*, int, int) [120]
983.73ms  7.4541ms        (1024 1024 1)       (16 16 1)        11  1.0625KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalInt(int*, int*, int, int) [121]
991.19ms  7.4527ms        (1024 1024 1)       (16 16 1)        11  1.0625KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalInt(int*, int*, int, int) [122]
998.80ms  9.5668ms        (1024 1024 1)       (16 16 1)        11  2.1250KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalDouble(double*, double*, int, int) [124]
1.00836s  9.5454ms        (1024 1024 1)       (16 16 1)        11  2.1250KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalDouble(double*, double*, int, int) [125]
1.01791s  9.5402ms        (1024 1024 1)       (16 16 1)        11  2.1250KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalDouble(double*, double*, int, int) [126]
1.02745s  9.5492ms        (1024 1024 1)       (16 16 1)        11  2.1250KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalDouble(double*, double*, int, int) [127]
1.03700s  9.5473ms        (1024 1024 1)       (16 16 1)        11  2.1250KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalDouble(double*, double*, int, int) [128]
1.04655s  9.5557ms        (1024 1024 1)       (16 16 1)        11  2.1250KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalDouble(double*, double*, int, int) [129]
1.05611s  9.5594ms        (1024 1024 1)       (16 16 1)        11  2.1250KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalDouble(double*, double*, int, int) [130]
1.06568s  9.5514ms        (1024 1024 1)       (16 16 1)        11  2.1250KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalDouble(double*, double*, int, int) [131]
1.07523s  9.5517ms        (1024 1024 1)       (16 16 1)        11  2.1250KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalDouble(double*, double*, int, int) [132]
1.08478s  9.5530ms        (1024 1024 1)       (16 16 1)        11  2.1250KB        0B         -           -           -           -  Tesla
P100-PCIE         1         7  transposeDiagonalDouble(double*, double*, int, int) [133]
1.09446s  96.472ms                    -               -         -         -         -  1.0000GB  10.366GB/s      Device    Pageable  Tesla
P100-PCIE         1         7  [CUDA memcpy DtoH]
1.19106s  197.30ms                    -               -         -         -         -  2.0000GB  10.137GB/s      Device    Pageable  Tesla
P100-PCIE         1         7  [CUDA memcpy DtoH]

Regs: Number of registers used per CUDA thread. This number includes registers used internally by the CUDA driver and/or tools and can be
more than what the compiler shows.
SSMem: Static shared memory allocated per CUDA block.
DSMem: Dynamic shared memory allocated per CUDA block.
SrcMemType: The type of source memory accessed by memory operation/copy
DstMemType: The type of destination memory accessed by memory operation/copy
==12580== Generated result file: /content/matrix-transpose/results.nvprof


------------------------------------------------
```

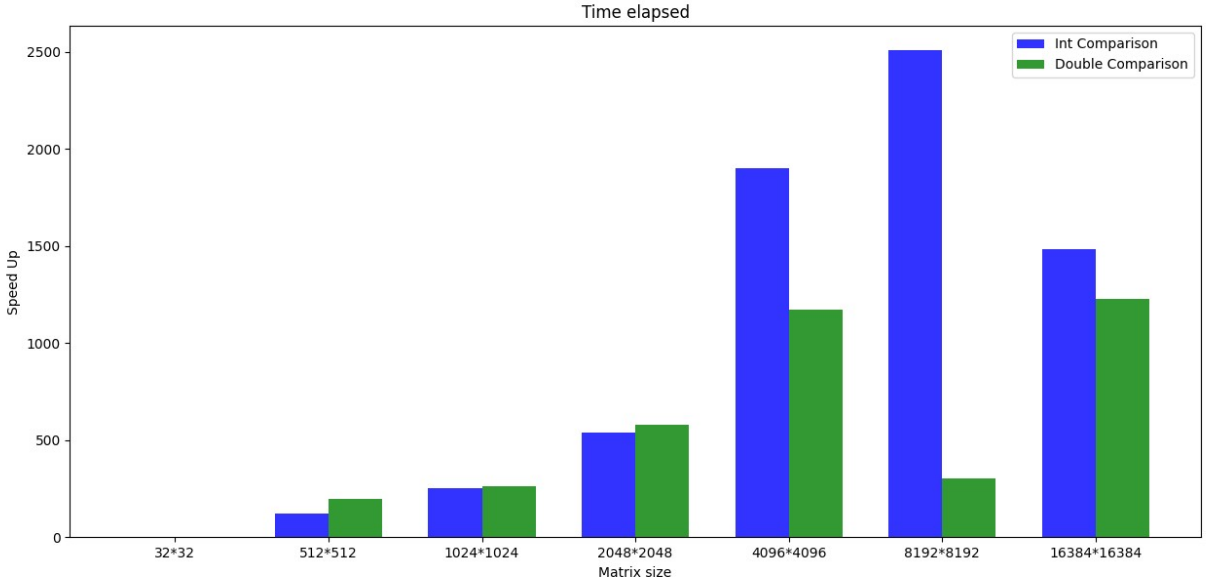
As you can see runtime of transpose functions have very significant speed ups. How parallelization of this functions work are described in the next section, below are charts of runtimes on GPU and comparsion between serial and parallel version.

## Time elapsed in prallel version:



Comparisons:



Last but not least lets take a brief look at parallelizing method. Each thread executing the kernel transposes four elements from one column of the input matrix to their transposed locations in one row of the output matrix. But what makes this implementation unique compared to naive implementations?

## First memory access is Coalesced:

because device memory has a much higher latency and lower bandwidth than on-chip memory, special attention must be paid to how global memory accesses are performed, in our case loading data from idata and storing data in odata. All global memory accesses by a half-warp of threads can be coalesced into one or two transactions if certain criteria are met. These criteria depend on the compute capability of the device, which can be determined, for example, by running the deviceQuery SDK example. For compute capabilities of 1.0 and 1.1, the following conditions are required for coalescing:

- threads must access either 32- 64-, or 128-bit words, resulting in either one transaction (for 32- and 64-bit words) or two transactions (for 128-bit words)
- All 16 words must lie in the same aligned segment of 64 or 128 bytes for 32- and 64-bit words, and for 128-bit words the data must lie in two contiguous 128 byte aligned segments
- The threads need to access words in sequence. If the k-th thread is to access a word, it must access the k-th word, although not all threads need to participate.

For devices with compute capabilities of 1.2, requirements for coalescing are relaxed. Coalescing into a single transaction can occur when data lies in 32-, 64-, and 128-byte aligned segments, regardless of the access pattern by threads within the segment. In general, if a half-warp of threads access N segments of memory, N memory transactions are issued. In a nutshell, if a memory access coalesces on a device of compute capability 1.0 or 1.1, then it will coalesce on a device of compute capability 1.2 and higher. If it doesn't coalesce on a device of compute capability 1.0 or 1.1, then it may. either completely coalesce or perhaps result in a reduced number of memory transactions, on a device of compute capability 1.2 or higher. For both the simple copy and naïve transpose, all loads from idata coalesce on devices with any of the compute capabilities discussed above. For each iteration within the i-loop, each half warp reads 16 contiguous 32-bit words, or one half of a row of a tile. Allocating device memory through cudaMalloc() and choosing TILE_DIM to be a multiple of 16 ensures alignment with a segment of memory, therefore all loads are coalesced. Coalescing behavior differs between the simple copy and naïve transpose kernels when writing to odata. For the simple copy, during each iteration of the i-loop, a half warp writes one half of a row of a tile in a coalesced manner. In the case of the naïve transpose, for each iteration of the i-loop a half warp writes one half of a column of floats to different segments of memory, resulting in 16 separate memory transactions, regardless of the compute capability. The way to avoid uncoalesced global memory access is to read the data into shared memory, and have each half warp access noncontiguous locations in shared memory in order to write contiguous data to odata. There is no performance penalty for noncontiguous access patters in shared memory as there is in global memory, however the above procedure requires that each element in a tile be accessed by different threads, so a __synchthreads() call is required to ensure that all reads from idata to shared memory have completed before writes from shared memory to odata commence.

## Shared Memory Bank Conflicts:

Shared memory is divided into 16 equally-sized memory modules, called banks, which are organized such that successive 32-bit words are assigned to successive banks. These banks can be accessed simultaneously, and to achieve maximum bandwidth to and from shared memory the threads in a half warp should access shared memory associated with different banks. The exception to this rule is when all threads in a half warp read the same shared memory address, which results in a broadcast where the data at that address is sent to all threads of the half warp in one transaction. One can use the warp_serialize flag when profiling CUDA applications to determine whether shared memory bank conflicts occur in any kernel. In general, this flag also reflects use of atomics and constant memory, however neither of these are present in my example. The coalesced transpose uses a 32x32 shared memory array of doubles or ints. For this sized array, all data in columns k and k+16 are mapped to the same bank. As a result, when writing partial columns from tile in shared memory to rows in odata the half warp experiences a 16-way bank conflict and serializes the request. A simple to avoid this conflict is to pad the shared memory array by one column: __shared__ int/double tile[TILE_DIM][TILE_DIM+1]; The padding does not affect shared memory bank access pattern when writing a half warp to shared memory, which remains conflict free, but by adding a single column now the access of a half warp of data in a column is also conflict free. The performance of the kernel, now coalesced and memory bank conflict free.

## Partition Camping:

Just as shared memory is divided into 16 banks of 32-bit width, global memory is divided into either 6 partitions (on 8- and 9-series GPUs) or 8 partitions (on 200- and 10-series GPUs) of 256-byte width. We previously discussed that to use shared memory effectively, threads within a half warp should access different banks so that these accesses can occur simultaneously. If threads within a half warp access shared memory though only a few banks, then bank conflicts occur. To use global memory effectively, concurrent accesses to global memory by all active warps should be divided evenly amongst partitions. The term partition camping is used to describe the case when global memory accesses are directed through a subset of partitions, causing requests to queue up at some partitions while other partitions go unused.

While coalescing concerns global memory accesses within a half warp, partition camping concerns global memory accesses amongst active half warps. Since partition camping concerns how active thread blocks behave, the issue of how thread blocks are scheduled on multiprocessors is important. When a kernel is launched, the order in which blocks are assigned to multiprocessors is determined by the one-dimensional block ID defined as: bid = blockIdx.x + gridDim.x*blockIdx.y; which is a row-major ordering of the blocks in the grid. Once maximum occupancy is reached, additional blocks are assigned to multiprocessors as needed. How quickly and the order in which blocks complete cannot be determined, so active blocks are initially contiguous but become less contiguous as execution of the kernel progresses. If we return to our matrix transpose and look at how tiles in our 2048x2048 matrices map to partitions on a GPU, as depicted in the figure below, we immediately see that partition camping is a problem.

idata

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 64 | 65 | 66 | 67 | 68 | 69 |
| 128 | 129 | 130 | ... | | |
| | | | | | |
| | | | | | |
| | | | | | |

odata

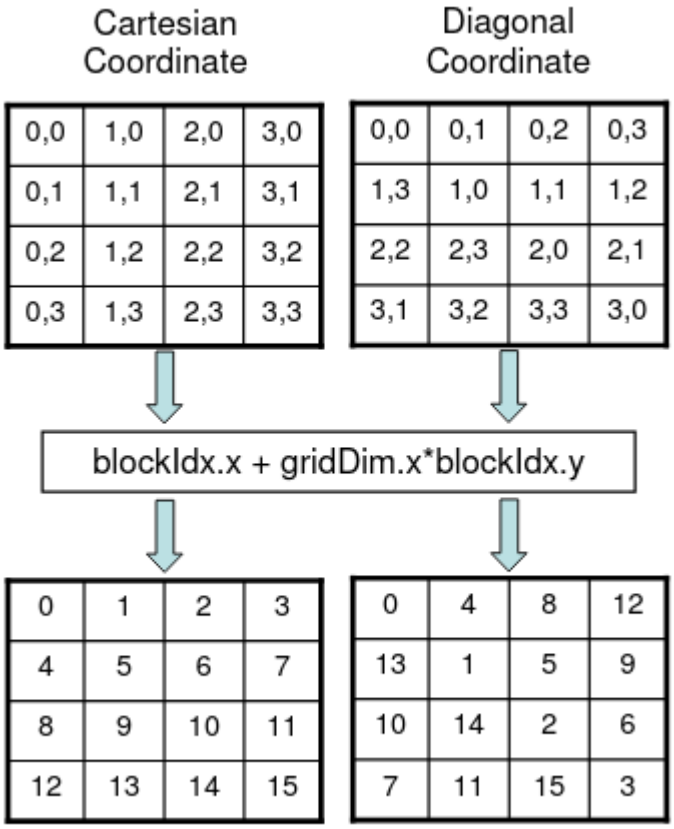
| | | | | | |
|---|---|---|---|---|---|
| 0 | 64 | 128 | | | |
| 1 | 65 | 129 | | | |
| 2 | 66 | 130 | | | |
| 3 | 67 | ... | | | |
| 4 | 68 | | | | |
| 5 | 69 | | | | |

With 8 partitions of 256-byte width, all data in strides of 2048 bytes (or 512 floats) map to the same partition. Any float matrix with an integral multiple of 512 columns, such as our 2048x2048 matrix, will contain columns whose elements map to a single partition. With tiles of 32x32 floats (or 128x128 bytes), whose one-dimensional block IDs are shown in the figure, all the data within the first two columns of tiles map to the same partition, and likewise for other pairs of tile columns (assuming the matrices are aligned to a partition segment). Combining how the matrix elements map to partitions, and how blocks are scheduled, we can see that concurrent blocks will be accessing tiles row-wise in idata which will be roughly equally distributed amongst partitions, however these blocks will access tiles column-wise in odata which will typically access global memory through just a few partitions. Having diagnosed the problem as partition camping, the question now turns to what can be done about it. Just as with shared memory, padding is an option. Adding an additional 64 columns (one partition width) to odata will cause rows of a tile to map sequentially to different partitions. However, such padding can become prohibitive to certain applications. There is a simpler solution that essentially involves rescheduling how blocks are executed.

## Diagonal block reordering:

While the programmer does not have direct control of the order in which blocks are scheduled, which is determined by the value of the automatic kernel variable blockIdx, the programmer does have the flexibility in how to interpret the components of blockIdx. Given how the components blockIdx are named, i.e. x and y, one generally assumes these components refer to a cartesian coordinate system. This does not need to be the case, however, and one can choose otherwise. Within the cartesian interpretation one could swap the roles of these two components, which would eliminate the partition camping problem in writing to odata, however

this would merely move the problem to reading data from idata. One way to avoid partition camping in both reading from idata and writing to odata is to use a diagonal interpretation of the components of blockIdx: the y component represents different diagonal slices of tiles through the matrix and the x component indicates the distance along each diagonal. Both cartesian and diagonal interpretations of blockIdx components are shown in the top portion of the diagram below for a 4x4-block matrix, along with the resulting one-dimensional block ID on the bottom.




Before we discuss the merits of using the diagonal interpretation of blockIdxcomponents in the matrix transpose, we briefly mention how it can be efficiently implemented using a mapping of coordinates. This technique is useful when writing new kernels, but even more so when modifying existing kernels to use diagonal (or other) interpretations of blockIdx fields. If blockIdx.x and blockIdx.y represent the diagonal coordinates, then (for block-square matrixes) the corresponding cartesian coordinates are given by the following mapping: blockIdx_y = blockIdx.x; blockIdx_x = (blockIdx.x+blockIdx.y)%gridDim.x; One would simply include the previous two lines of code at the beginning of the kernel, and write the kernel assuming the cartesian interpretation of blockIdxfields, except using blockIdx_x and blockIdx_y in place of blockIdx.xand blockIdx.y, respectively, throughout the kernel.

**Sincerely yours**
**Ali Yazdanpanah**