1 Algorithm

The algorithm is standard block-wise matrix multiplication slightly modified to exploit maximum parallelism. Let the matrix size be n; block size be m, and $T = n/m$. Let the input matrices be $A_{n\times n}$ and $B_{n\times n}$ and the output matrix be $C_{n\times n}$. All three matrices are stored in row-major order and a boolean array for each (of size T^2), indicating a non-zero element's presence in that block.

1. T^2 blocks are created each containing m^2 threads. Keeping threads dependent on m will not be an issue as $m \leq 8$, and the maximum number of threads per block on the K40 GPU is 1024.
2. Each thread calculates its corresponding element of the output matrix. This is done through the standard matrix multiplication method. This does not create any race conditions as data is being read simultaneously but no thread attempts to write at the same location as another thread.
3. The final value of the element is then truncated to fit into a 4 byte unsigned integer. If the value is non-zero, the block's corresponding position in the array is set to true. At first glance it may seem that this would create a race condition but each thread that attempts to write at this location would definitely writing the same value so the correctness of the final value would not be affected.

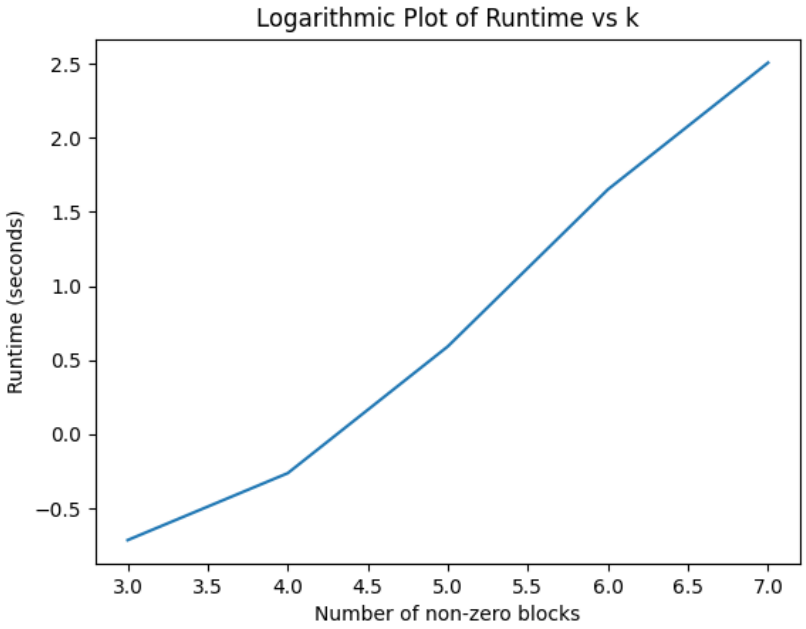
2 CUDA Features Used

1. `cudaMalloc` is used to allocate device memory.
2. `cudaMemcpy` is used to copy memory from host to device and vice-versa.
3. `cudaDeviceSynchronize` is used to ensure that all device threads complete before the host continues its code.
4. The in-built variables `blockIdx` and `threadIdx` are used to find the block index and cell index respectively.

3 Performance

This algorithm gives good performance even for large matrices. In this case, the primary factor in determining the time complexity is the values of k (non-zero blocks) for both A and B, as the computation is done only for these blocks and skipped for the others.

n	m	k	Runtime
256	4	1000	0.193
2048	8	10000	0.546
4096	4	100000	3.913
16384	8	1000000	44.841
16384	4	10000000	321.053



Note: The plot is logarithmic due to the large variations in the values of k.

As observable from the graph, the runtime has an almost linear relation with the value of k. As $k \in O(n^2)$, the algorithm turns out to have a final complexity of $O(n^2)$. This is a great improvement from the standard complexity of $O(n^3)$.