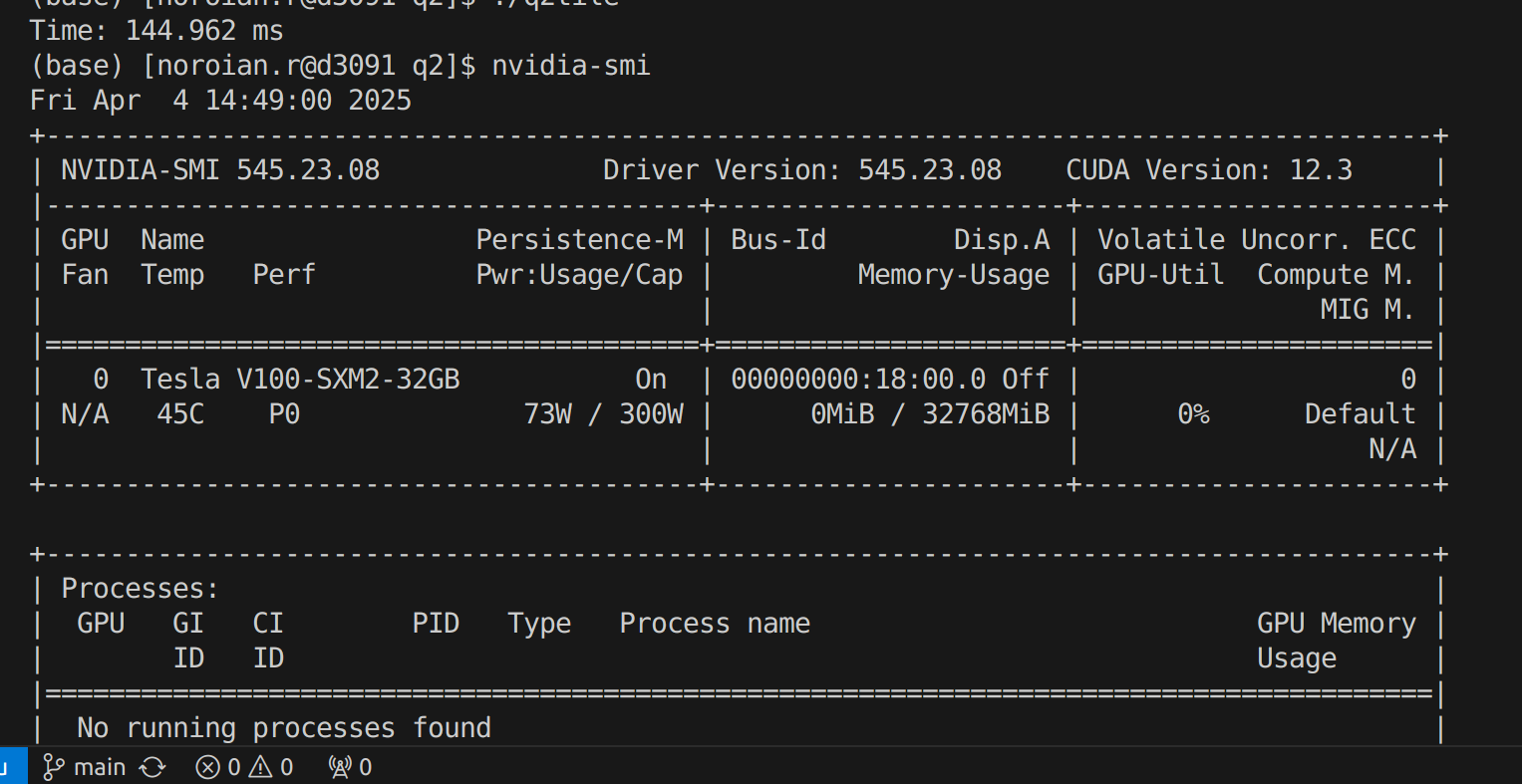
Q2:

2. (40) The code below carries out a “nearest neighbor” or “stencil” computation. This class of algorithm appears frequently in image processing and visualization applications. The memory reference pattern for matrix b exhibits reuse in 3 dimensions. Your task is to develop a C/CUDA version of this code that initializes b on the host and then uses tiling on the GPU to exploit locality in GPU shared memory across the 3 dimensions of b:  
#define n 32  
float a[n][n][n], b[n][n][n];  
for (i=1; i<n-1; i++)  
 for (j=1; j<n-1; j++)  
 for (k=1; k<n-1; k++) {  
 a[i][j][k]=0.75\*(b[i-1][j][k]+b[i+1][j][k]+b[i][j-1][k]  
 + b[i][j+1][k]+b[i][j][k-1]+b[i][j][k+1]);  
}

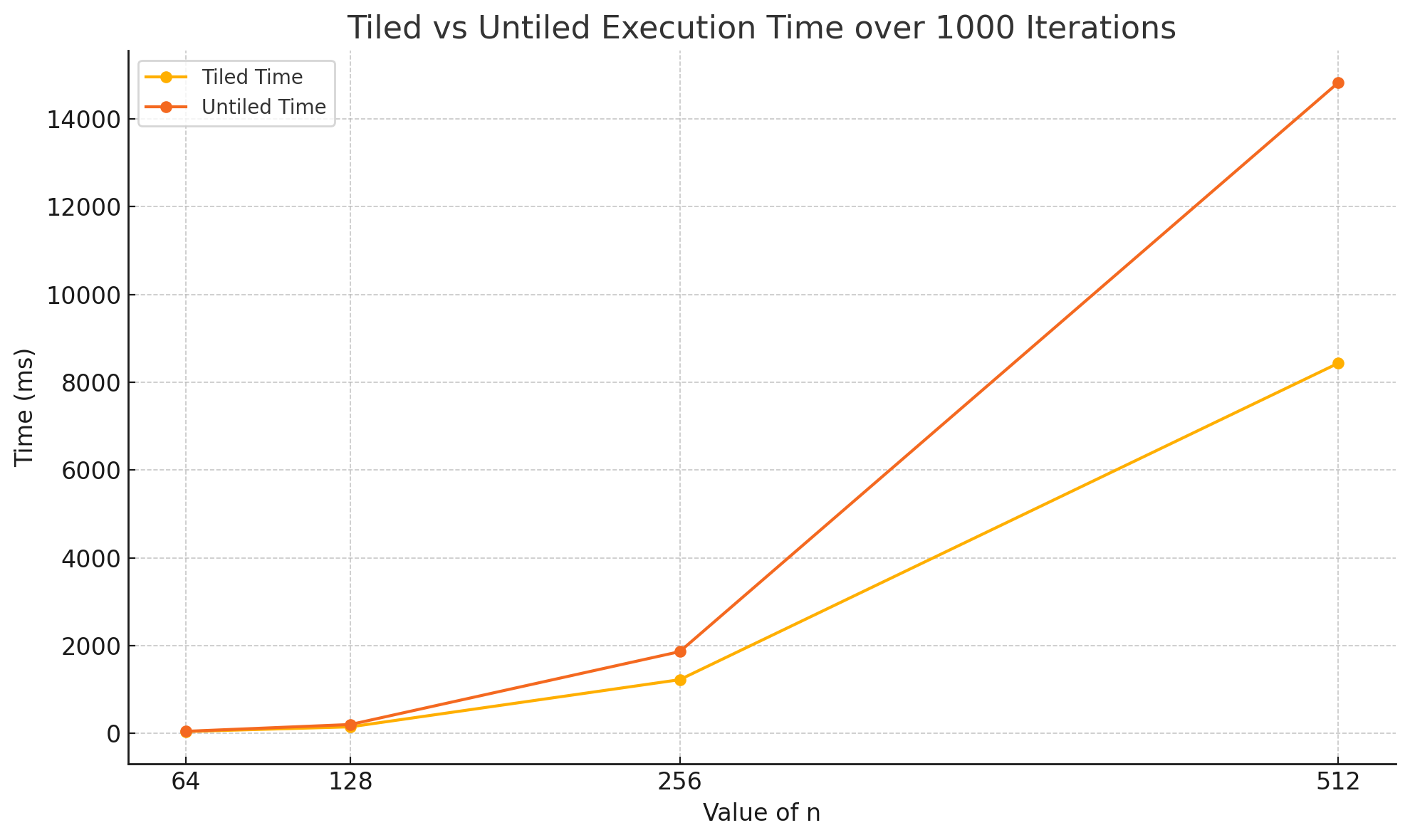
a.) Evaluate the performance of computing a tiled versus non-tiled implementation in your GPU application. Explore what happens when you change the value of n. Consider the performance for at least two additional values for n.

With stencil computation, optimizations can be made with spatial locality in memory, as elements in the 3D matrix, outside the edges, use the indices above and below, in front and behind, and to the right and left. This means that many indices close in memory use the same values, and so they can be stored in a shared memory 3D array. This is done on the GPU during kernel execution to enable tiling. Though the tile size is set to 8, the block size is specified to have a 3D array of 10 to ensure that a halo of values is saved for the right indices. The shared memory then loads the values in the original b[][][] array and ensuring that they are within the bounds of the array size, completes the computation to add them to a[][][].

I used a V100 GPU with the follow specifications:



|  |  |  |
| --- | --- | --- |
| Value of n | Tiled time 1000 iterations (ms) | Untiled time 1000 iterations(ms) |
| 64 | 45.5 | 50.2 |
| 128 | 152.3 | 205.4 |
| 256 | 1226.4 | 1865.2 |
| 512 | 8432.3 | 14822 |



Based on these results, it’s evident that tiling provided definitive benefits, especially over larger scale computations that benefitted from repeated haloing in shared memory. Tiling, in this case, decreases cache misses as chunks of memory shared between threads in a block are all used together. Memory locality, for a 3D problem at a large scale, introduces high latency that results in slower computational times. These, as shown by the graph, are not linear, as the increase of n results in 2^3 difference in the size of the 3D array.

Nvidia Nsight was also used here, and it is the reason behind selecting the tile size of 8, as this was determined to maximize throughput for this computation. It was especially beneficial at higher values of n, as shown by the graph.

b.) Explore and report on other optimizations to accelerate your code on the GPU further.

In my implementation there is somewhat inefficient shared memory, as some of the halo is uninitialized due to imperfect buffering in the tile size for the shared memory array.

Generally speaking, to optimize stencil computation, multiple other optimizations can be made.

Based on Nvidia Nsight results, it was clear that warp divergence was a somewhat large bottleneck, as there exist multiple conditionals within the kernel execution that may disrupt performance. To prevent this, precomputation of the integers used in the computation would be valuable and register local could be reused.

The shared memory access pattern could also be improved to avoid bank conflicts. Rather than +2 for the halo, +1 may be a good alternative. This could be experimented further as well. It may be beneficial for the 3D shared memory to be a size of base 2 to enable better cache utilization and wrap deployment.

Lastly, it would be valuable to test this on different GPUs. I ran this code on a Tesla V100, but as described in question 3, this has older generation tensor cores and smaller memory compared to the A100 and H100 GPUs. To see acceleration, it would be beneficial to investigate how speedup looks with these memory hierarchies, and with streaming multiprocessors that possess newer, advanced cores for deployment.