Question – 2

Assignment - 5

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

Answer:

(a). In Non-tiled implementation each thread computes one element of matrix a directly using elements from matrix b. The performance penalty of repeated global memory accesses may not be as apparent when the arrays are smaller (e.g., when N is 32). However, as N increases, the total volume of data becomes significantly larger, and the cost of global memory accesses begins to dominate.

For **N = 32**; in non-tiled kernel, this is the execution time:

```
[sridhar.pray@explorer-01 ~]$ nano naive_stencil.cu
[sridhar.pray@explorer-01 ~]$ nvcc naive_stencil.cu -o naive_stencil -lcudart
[sridhar.pray@explorer-01 ~]$ ./naive_stencil
N Value: 32
Native (non-tiled) execution time: 0.006154 ms
```

For **N = 64**; in non-tiled kernel, this is the execution time:

```
[sridhar.pray@explorer-01 ~]$ nvcc naive_stencil.cu -o naive_stencil -lcudart
[sridhar.pray@explorer-01 ~]$ ./naive_stencil
N Value: 64
Native (non-tiled) execution time: 0.007276 ms
```

For **N = 96**; in non-tiled kernel, this is the execution time:

```
[sridhar.pray@explorer-01 ~]$ nano naive_stencil.cu
[sridhar.pray@explorer-01 ~]$ nvcc naive_stencil.cu -o naive_stencil -lcudart
[sridhar.pray@explorer-01 ~]$ ./naive_stencil
N Value: 96
Native (non-tiled) execution time: 0.007309 ms
```

_

In the tiled version, each block of threads loads a tile of b into shared memory prior to computing elements of a. The objective of this method is to minimize the global memory bandwidth by utilizing data reuse within the tile.

For **N = 32**; in tiled kernel this is the execution time:

```
[sridhar.pray@explorer-01 ~]$ nano tailed_stencil.cu
[sridhar.pray@explorer-01 ~]$ nvcc tailed_stencil.cu -o tailed_stencil -lcudart
[sridhar.pray@explorer-01 ~]$ ./tailed_stencil
Value of N: 32
Tiled execution time: 0.006297 ms
```

For **N = 64**; in tiled kernel this is the execution time:

```
[sridhar.pray@explorer-01 ~]$ ./tailed_stencil
Value of N: 64
Tiled execution time: 0.007459 ms
```

For **N = 96**; in tiled kernel this is the execution time:

```
[sridhar.pray@explorer-01 ~]$ ./tailed_stencil
Value of N: 96
Tiled execution time: 0.007927 ms
```

Comparison:

- **Non-Tiled vs Tiled Execution Time**: For the same value of N, the tiled implementation typically exhibits slightly longer execution times than the non-tiled version. This indicates that the benefits anticipated from improved cache utilization and reduced global memory access may not be compensated for by the overhead associated with tiling (such as setting up the tiles and boundary conditions) for the range of N values tested.
- **Scalability as N increases**: Both implementations exhibit an increase in execution time as N increases. Nevertheless, the increase is not overly significant, suggesting a certain degree of scalability. The overhead may become more significant as N increases further, as seen by the slightly sharper increase in the tiled version.

(b): There are many other ways of optimizations to accelerate the GPU further:

- **Dynamic Parallelism**: It allows kernel that is operating on the GPU is capable of launching other kernels without the need to return control to the host. This can assist in the optimization of your code by:
    - **Improving load balancing**: kernels can spawn additional work based on data-dependent conditions which helps in better distribution of workload across GPU.
    - **Enabling nested parallelism**: Dynamic parallelism can effectively handle hierarchical tasks in complex methods, making sure that fine-grained subtasks are handled right away by the GPU.
- **Loop unrolling**: It helps by letting the compiler see more independent processes and lowering the overhead that comes with controlling loops, such as increasing counters and checking for exit conditions. This extra instruction-level parallelism can let the GPU schedule more work at the same time, which can boost total throughput and lower latency in parts of your code that are memory- or compute-bound.
- **Asynchronous data transfers**: It allows the GPU to move data between host and device memory without blocking kernel execution. This means that while one part of the GPU is busy executing a kernel, another part can simultaneously transfer data needed for subsequent computations. This overlap of computation and data movement reduces idle time, improves resource utilization, and ultimately leads to a reduction in overall execution time.
- **Prefetching**: When you prefetch, you put data into faster memory (like shared memory or registers) before you use it. In other words, when the GPU thread needs the data, it is already there, cutting down on the time it has to wait for memory delays. Basically, prefetching combines data transfer with computation, which cuts down on stalls and boosts total throughput.
- **Resource Utilization**: Using GPU resources like registers and shared memory in the best way possible can have a direct effect on speed. If you use too much shared memory per block, you can't run as many blocks at once. Similarly, if you use too many registers per thread, you can't run as many threads per block.