CS433A : Report | Design Exercise 3

> ***Submitted on : April 27, 2022 by Group No. 24***
>
> ***Dipanshu Garg | 190306***
>
> ***Gurbaaz Singh Nandra | 190349***

Deliverables

```
.
├── Q1.cu
├── Q2.cu
└── report.pdf

0 directories, 3 files
```

Question 1

Usage

```
nvcc -O3 Q1.cu -o Q1
./Q1 <size-of-vector-n> <number-of-threads-t>
```

Thread Grid Structure

Each thread block is a square tile. The entire `n*n` grid is divided into disjoint square tiles. Each thread block gets mapped to multiple square tiles.

Algorithm

We exploit the massive parallelisability of Gauss Seidel algorithm as each element can be parallely updated.

We use a sense-reversal barrier to synchronise all thread-blocks.

We maintain a global variable `diff` and update it atomically using `CUDA` inbuilt instruction `atomicAdd`.

Basic Program

Each thread in a thread block first computes its local `diff` in a temporary variable and then atomically adds the value to global `diff`. So every thread is performing an atomic operation, which is inefficient.

Optimisations with Tree Reductions and Shared Memory

We maintain a shared memory with dimensions equal to that of the thread block. Each thread in a thread block computes its local `diff` and updates it in the shared memory. Then we perform tree reduction using this shared memory. First, each of the warps compute thir local `diff` sum. Then we initialise another array of dimensions equal to `(TILE_SIZE*TILE_SIZE) / warp_size` which contains the local `diff` sums computed for each warp. Then we again perform tree reduction on these sums. Finally, the thread with `thread_id` = 0 will contain the local `diff` sum value for the entire thread block. Then we atomically add this value to our global `diff`. So unlike the basic program where there was one atomic operation per thread, here we are using only `1` atomic instruction per thread block. Because of this, we see a significant amount of speedup in the performance of optimised version, as shown below.

Comparison with OpenMP

We observe that our optimised version performs significantly better than the best performance of `OpenMP` program (thread count 4). This is because Gauss Seidel can be highly parallelised with `O(n^2)` threads, so `CUDA` is an ideal option here.

Additional approaches and thoughts

We also thought of using `1-dimensional` thread blocks rather than tiles, but we prefer tiles as that would help in faster convergence because each thread block will have the information of most updated `A[i]`'s (except for the border of thread block tile).

We also observed that despite being slower, `OpenMP` program on average takes lesser number of iterations (`300-350`) as compared to the optimised version of the `CUDA` program (`350-400`). This is because `OpenMP` always uses the most updated values of `A[i]` because of cache-coherence, which is not the case in `CUDA`.

Results

These results have been obtained on an `NVIDIA GeForce MX250 (2 GB)` machine.

Choice of `n` = 4096

Performance of Basic Program

Time (in microseconds)	Thread Count
12471846	512
7617310	1024
8814774	2048
18058290	4096
18354371	8192
17841806	16384
18420416	32768
18306904	65536
18372722	131072

Performance of Optimised Program

Time (in microseconds)	Thread Count
12980099	512
11031589	1024
7233197	2048
7092255	4096
6823287	8192
6988132	16384
7118372	32768
7121419	65536
7095654	131072

Performance of OpenMP Program

For evaluating equivalent `OpenMP` program, we are using an `8-core` machine. We used the file

OpenMP implementation with static block row assignment: omp_gauss-seidel_blockrow.c

from class demos (we commented out the part where `diff` is written to file for fair comparison)

Time (in microseconds)	Thread Count
45394445	1
25811253	2
19729098	4
29349812	8

Question 2

Usage

```
nvcc -O3 Q2.cu -o Q2
./Q2 <size-of-vector-n> <number-of-threads-t>
```

Thread Grid Structure

Here, we use one dimensional thread block. Each thread in a thread block gets mapped to computing a block of elements in the final vector `y`.

Algorithm

Basic Program

The basic program simply computes the value of `y[i]` by assigning the complete computation of `y[i]` to a particular thread. So the thread will iterate over all the columns in matrix `A` and compute the sum of element-wise product with the vector `x`, and assign it to `y[i]`. Clearly here we are not efficiently using cache, as multiple threads may suffer cache miss for the same `x[i]` elements.

Optimisations with Shared Memory

Here we use shared memory to cache a block of `x[i]` 's. Then the entire thread block uses these `x[i]`'s for further computation. After that all the thread in the thread blocks are synchronised and these cached `x[i]`'s are replaced with the next set of `x[i]`'s. So here we are ensuring that the cache `x[i]`'s are not evicted using shared memory.

We expected the optimised version to show good speedups due to cache effects but we observe that performance is not as good as that of the basic program. The reason could be that the synchronisation overhead after using each block of `x[i]`'s outweights the benefit of optimal caching. We tried different thread block sizes, but got similar outcomes.

Comparison with OpenMP

We observe that the best `OpenMP` program (thread count 4) performs better than both of the `CUDA` programs. This could be because here we have `O(n)` parallelisations. So the overhead of kernel launch neutralises the benefits of better parallelisation.

Results

These results have been obtained on an `NVIDIA GeForce MX250 (2 GB)` machine.

Choice of `n` = 16384

Performance of Basic Program

Time (in microseconds)	Thread Count
163006	512
165369	1024
149270	2048
171299	4096
171401	8192
173436	16384

Performance of Optimised Program

Time (in microseconds)	Thread Count
259751	512
228312	1024
180925	2048
504616	4096
390699	8192
280818	16384

Performance of OpenMP Program

Time (in microseconds)	Thread Count
405209	1
200577	2
134704	4
169942	8
209459	16