Advanced Topics in Numerical Analysis: High Performance Computing Assignment 4 (due Apr. 18, 2022) Anav Prasad, ap7152

Handing in your homework: Hand in your homework as for the previous homework assignments (git repo with Makefile), answering the questions by adding a text or a LaTeX file to your repo.

1. **Matrix-vector operations on a GPU.** Write CUDA code for an inner product between two given (long) vectors on a GPU. Then, generalize this code to implement a matrix-vector multiplication (no blocking needed here) on the GPU. Check the correctness of your implementation by performing the same computation on the CPU and compare them. Report the memory band your code obtains on different GPUs[1].

   Solution:
   For **inner-product**, the vector size was chosen as 2^{24} because in the ranges of $2^1 - 2^{10}$, the CPU bandwidths and GPU bandwidths are comparable. In the ranges of $2^{11} - 2^{20}$, CPU performs better than the GPU. Finally, at sizes $> 2^{20}$, GPU parallizations finally become significant enough to improve the overall time as seen below.
   Note that the values of bandwidth's are in GB/s.

   - **inner-product results:**
     Vector Size: 2^{24}

Compute Server	GPU	CPU Bandwidth	GPU Bandwidth
cuda1	GTX TITAN Black	12.0025	62.7648
cuda2	RTX 2080 Ti	10.955	131.298
cuda3	TITAN V	5.0364	204.313
cuda4	GTX TITAN X	5.21459	66.8682
cuda5	GTX TITAN Z	8.89145	51.611

   For **matrix-vector-product**, on the other hand, GPU parallelized code almost always performs at least a little better than the CPU parallelized code at varied ranges of matrix and vector sizes. The given matrix size shows decent distinction between the two bandwidths.

[1]The cuda{1–5}.cims.nyu.edu compute servers at the Institute have different Nvidia GPUs, for an overview see the list of compute servers available at the Institute: https://cims.nyu.edu/webapps/content/systems/resources/computeservers.

- **matrix-vector-product results:**
  Matrix Size: 500×2^{18}
  Vector Size: 2^{18}

Compute Server	GPU	CPU Bandwidth	GPU Bandwidth
cuda1	GTX TITAN Black	13.7252	37.0873
cuda2	RTX 2080 Ti	5.11759	60.84
cuda3	TITAN V	5.6556	90.4507
cuda4	GTX TITAN X	6.3598	39.2735
cuda5	GTX TITAN Z	12.341	31.9898

2. **2D Jacobi method on a GPU.** Implement the 2D Jacobi method as discussed in the 2nd homework assignment using CUDA. Check the correctness of your implementation by performing the same computation on the CPU and compare them.
   Solution:
   Machine Details:

```
Compute Server cuda3.cims.nyu.edu
Device Name: NVIDIA TITAN V
Computation Capability: 7.0
Memory: 12.6528
```

Number of Threads Set = 4

N	#Max Iterations	CPU Time	GPU Time
100	10000	1.01629s	0.342811s
1000	10000	9.07443s	6.78653s

3. **Update on final projection** Describe with a few sentences the status of your final project: What tasks that you formulated in the proposal have you worked on? Did you run into unforeseen issues?

   Solution:
   Collaborators:

   - Aditya Kashilkar, ask9126
   - Sai Himal Allu, sa6783

   The following checkpoints have been completed:

- `Checkpoint 1: Benchmark the serialized implementation of WFC algorithm`

  A few sample images:

  - `chess.bmp`:

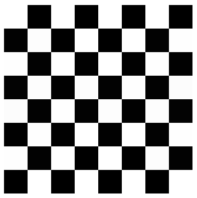

Figure 1: Reference bitmap: chess.bmp

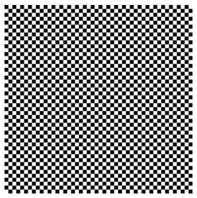

Figure 2: Larger candidate bitmap generated by the algorithm

    Time taken to process and generate the larger bitmap from `chess.bmp`: 0.504s

  - `city.bmp`:

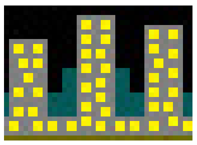

Figure 3: Reference bitmap: city.bmp

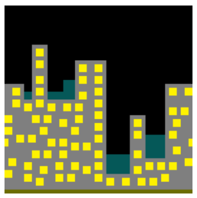

Figure 4: Larger candidate bitmap generated by the algorithm

    Time taken to process and generate the larger bitmap from `city.bmp`: 1.928s

  - `flowers.bmp`:

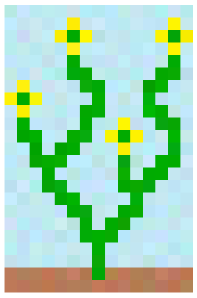

Figure 5: Reference bitmap: flowers.bmp

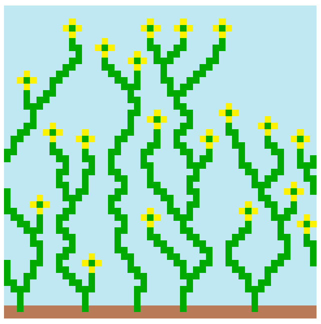

Figure 6: Larger candidate bitmap generated by the algorithm

    Time taken to process and generate the larger bitmap from `flowers.bmp`: 1.388s

  Total Time taken to process all 54 sample images: 54.671533

4. `Checkpoints 2 and 3:`
   - `-> Analyze the reference code for optimal parallelization avenues.`
   - `-> Implement basic OpenMP optimizations.`

   Although there are simple optimisation routines possible for the multiple for-loops used in the code-base, there is some amount of non-triviality involved with optimising it.

   While the code compiles and, the results are holding up, the speedups reported are not that great. We reason this might be due to the fact that there are a significant number of branching operations in our serial code which is negating some of the benefits of parallelization.

   We are currently looking into avenues for reducing the number of branching instructions through some optimisation tricks