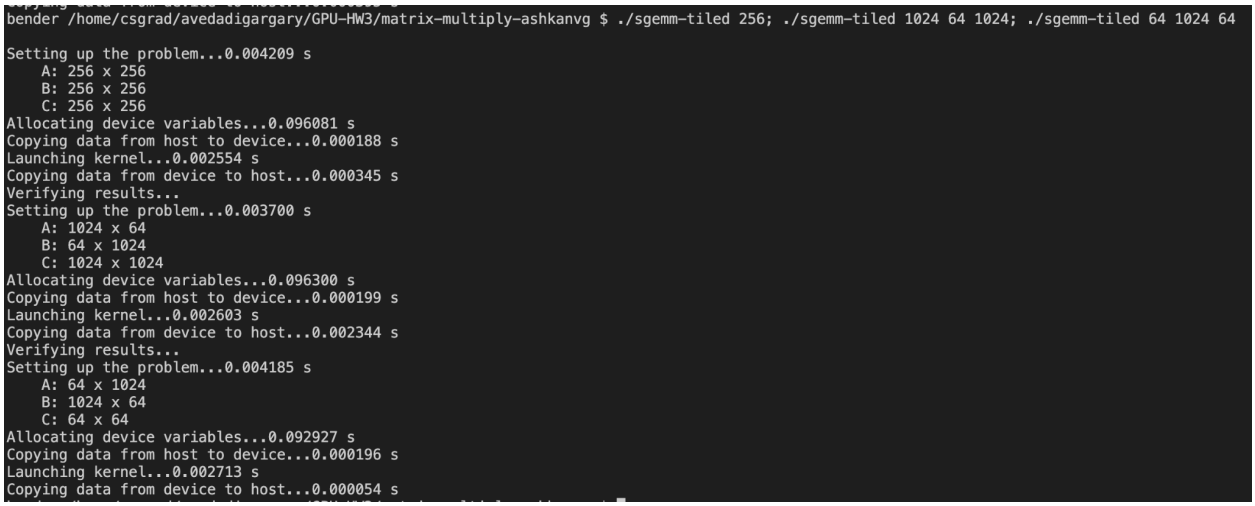
Ashkan Vedadi Gargary | **aveda002 | 862352576**

1. **On Bender, compare the execution time of a 256 x 256 square matrix multiplication compared to a 1024 x 64 and 64 x 1024 rectangular matrix multiply. All input matrices have 65k entries. What do you observe? Which is faster? Can you explain the observed behavior? Tip: You may want to comment out the verify() function in main.cu when timing this question.**



It shows that allocating device variables and copying from host to device is almost the same.

However, launching the kernel function for the first setup is faster than the second setup. The reason behind this is because of the amount of context switching in the second and third setup. And also, the 256x256 matrix multiplication has less number of operations per load than 1024x64. And 64*1024 has fewer operations than other setups.

Also, it shows that copying data from device to host at the end for 64x64 is better than others and 256x256 is faster than 1024x1024 which is because of the size.

2. **Conceptual Question: For a 64 square tiled matrix multiplication, how many times is each element of the input matrices loaded from global memory? Assume 16x16 tiles.**

   4 times

   Conceptual Question: For a 64 square non-tiled matrix multiplication, how many times is each element of the input matrices loaded from global memory?

   64 times

3. **GPGPU-Sim related question: In this part, we will compare the execution of a 128x128 square tiled matrix multiplication across different tile sizes. Run `./sgemm-tiled 128` in GPGPU-Sim with `TILE_SIZE` of 8, 16 (default), and 32. Fill the following table:**

Tile size	8	16	32	Note
gpu_tot_sim_cycle	41445	26626	55956	Total cycles
gpu_tot_ipc	401.6442	447.9663	398.2100	Instruction per cycle
gpgpu_n_load_insn	524288	262144	131072	Total loads to global memory
gpgpu_n_store_insn	16384	16384	16384	Total stores to global memory
gpgpu_n_shmem_insn	4718592	4456448	4325376	Total accesses to shared memory

4. **Which tile size resulted in the least number of accesses to global memory? Which tile size resulted in the most number of accesses to global memory? What is the reasoning behind this observation?**
    - Least number of accesses to global memory: 32 tile sizes
    - Most number of accesses to global memory: 8 tile size
    - Having fewer tiles will provide better results in accesses in global memory. When you use a larger tile size, like 32, each thread in the block can process more data, which means fewer blocks are needed. This results in fewer accesses to global memory, as more of the data can be loaded into the faster shared memory and reused across multiple threads in the same block. On the other hand, a smaller tile size, like 8, means that each thread processes less data, requiring more blocks. This leads to more access to global memory, as less data can be loaded into shared memory and reused. Also, it's important to note that the optimal tile size can depend on the specific GPU architecture and the nature of the data and computations.
5. **Which tile size performed the fastest, which tile size performed the slowest? Why do you think that is?**
    - The fastest: 16 size tile
        - It has the fewest total cycles (gpu_tot_sim_cycle: 26626) and the highest instructions per cycle (gpu_tot_ipc: 447.9663).
    - The Slowest: 32 size tile
        - It has the most total cycles (gpu_tot_sim_cycle: 55956) and the lowest instructions per cycle (gpu_tot_ipc: 398.2100).
    - A tile size of 16 seems to strike a good balance for this particular workload and GPU architecture. It allows for efficient use of shared memory (gpgpu_n_shmem_insn: 4456448), reducing the need for slower global memory accesses (gpgpu_n_load_insn: 262144), while still having enough threads to keep the GPU cores busy. On the other hand, a tile size of 32 might be too large, leading to less efficient use of shared memory and more contention for resources, resulting in more total cycles. Conversely, a tile size of 8 might be too small, leading to underutilization of the GPU cores and more frequent global memory accesses, which are slower than shared memory accesses.