Assignment 1 Submission Structure

Submission Requirements

PDF File

- **Name**: `DD2360HT25_HW1_Group2.pdf`
- **Content**:
  - Each group member's contributions
  - Answers to each exercise

ZIP File Structure

The zip file should contain code for each programming question in separate folders:

```
assignment1/
├── Q[1]/
│   └── README.md
├── Q[2]/
│   ├── vecAdd.cu
│   ├── Makefile
│   └── README.md
│   └── batch_test.sh
│   └── plot.py
├── Q[3]/
│   ├── vecMult.cu
│   ├── Makefile
│   └── README.md
│   └── batch_test.sh
│   └── plot.py
└── Q[4]/
    └── BFS_openmp_vs_cuda.csv
    └── hotspot_openmp_vs_cuda.csv
    └── README.md
```

Group Member Contribution

- Shitong Guo: 50%, Q[2], Q[3]
- Jinye Gong: 50% Q[1], Q[4]

Q[1] Reflection on GPU-accelerated Computing：

- Questions:

1. Explain in your own words three main architectural differences between GPUs and CPUs

   The GPU is a throughput Oriented Architecture, and the CPU is a latency Oriented Architecture.

   - GPU uses simple processing units

  Compared with the CPU, the GPU has more arithmetic (compute) units but relatively simpler control logic. This is because CPUs need complex mechanisms that can support complicated exception handling, while GPUs are mainly used to accelerate the parallel parts of programs and handle relatively single-purpose tasks, so the required control logic is simpler.

  - GPU uses hardware threads

    Compared with the CPU's software-implemented threads that are managed by the operating system, the GPU implements threads directly in hardware. Each SM has its own control unit and caches, which greatly increases the number of threads a GPU can handle. Compared with CPUs, it has stronger parallel processing capability and scalability.

  - GPU uses SIMD execution

    GPUs mainly adopt SIMD execution, so they can process as much parallel data as possible in a single cycle. Traditional single-core CPUs, on the other hand, mostly use SISD to execute sequentially; to enhance performance, CPUs use techniques like superscalar execution, which is more suitable for instructions with branch divergence and irregular memory access. But in essence CPUs are still sequential, whereas GPUs are designed for parallel computation, so they use SIMD execution.

2. Check the latest Top500 list (published at [https://top500.org/Links to an external site.](https://top500.org/)) that ranks the top 500 most powerful supercomputers in the world. In the top 10, how many supercomputers use GPUs? Report the name of the supercomputers and their GPU vendor (Nvidia, AMD, ...) and model.

   There are 9 supercomputers use GPUs.

   - TOP1

     El Capitan

     - 43,808 AMD fourth Gen EPYC 24C "Genoa" 24-core 1.8 GHz CPUs
     - 43,808 AMD Instinct MI300A GPUs

   - TOP2

     Frontier

     - 9,472 AMD third Gen Epyc 7713 "Trento" 64 core 2 GHz CPUs
     - 37,888 AMD Instinct MI250X GPUs

   - TOP3

     Aurora

     - 21248 intel Xeon CPU Max 9470 52C 2.4GHz
     - 63744 Intel Data Center GPU Max

   - TOP4

     JUPITER Booster

     - around 24,000 NVIDIA GH200 Grace Hopper superchips
     - each GH200 Grace Hopper superchip conclude one NVIDIA Grace CPU and one NVIDIA Hopper GPU

   - TOP5

     Eagle

     - 3600 Intel Xeon Platinum 8480C CPUs

        - 14,400 NVIDIA H100 GPUs
    - TOP6

      HPC6

        - 3,472 AMD third Gen Epyc 7713 "Trento" 64 core 2 GHz CPUs
        - 13,888 AMD Instinct MI250X GPUs
    - TOP7

      Supercomputer Fugaku

        - Fujitsu A64FX 48C 2.2GHz CPUs
    - TOP8

      Alps

        - around 10,752 NVIDIA GH200 Grace Hopper superchips
        - each GH200 Grace Hopper superchip conclude one NVIDIA Grace CPU and one NVIDIA Hopper GPU
    - TOP9

      LUMI

        - 4,096 AMD third Gen Epyc 7713 "Trento" 64 core 2 GHz CPUs
        - 11,912 AMD Instinct MI250X GPUs
    - TOP10

      Leonardo

        - 3,456 Intel Xeon Platinum 8358 CPUs
        - 13,824 NVIDIA A100 GPUs
3. One main advantage of GPU is its **power efficiency**, which can be quantified by *Performance/Power*, e.g., throughput as in FLOPS per watt power consumption. Calculate the power efficiency for the top 10 supercomputers.
    - power efficiency：

      El Capitan: 1,742.00 PFlop/s ÷ 29,581 kW = 0.05889 PFlop/s/kW = **58.89 GFLOP/s/W**

      Frontier: 1,353.00 PFlop/s ÷ 24,607 kW = 0.05498 PFlop/s/kW = **54.98 GFLOP/s/W****

      Aurora: 1,012.00 PFlop/s ÷ 38,698 kW = 0.02615 PFlop/s/kW = **26.15 GFLOP/s/W**

      JUPITER Booster: 793.40 PFlop/s ÷ 13,088 kW = 0.06062 PFlop/s/kW = **60.62 GFLOP/s/W**

      HPC6: 477.90 PFlop/s ÷ 8,461 kW = 0.05648 PFlop/s/kW = **56.48 GFLOP/s/W**

      Eagle It doesn't offer its Power

      Fugaku: 442.01 PFlop/s ÷ 29,899 kW = 0.01478 PFlop/s/kW = **14.78 GFLOP/s/W**

      Alps: 434.90 PFlop/s ÷ 7,124 kW = 0.06105 PFlop/s/kW = **61.05 GFLOP/s/W**

      LUMI: 379.70 PFlop/s ÷ 7,107 kW = 0.05343 PFlop/s/kW = **53.43 GFLOP/s/W**

      Leonardo: 241.20 PFlop/s ÷ 7,494 kW = 0.03219 PFlop/s/kW = **32.19 GFLOP/s/W**

Q[2] Submission Status

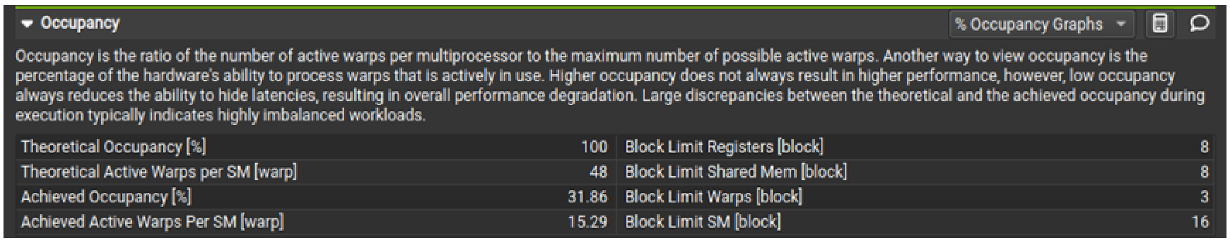
- Questions:
    1. **See the VecAdd.cu.**

```
    //@@ 1. Allocate in host memory.
    float *h_a = (float *)malloc(size);
    float *h_b = (float *)malloc(size);
    float *h_c = (float *)malloc(size);
    float *h_c_ref = (float *)malloc(size);

    //@@ 3. Initialize host memory.
    for (int i = 0; i < N; i++) {
        h_a[i] = (float)(rand() % 100) / 10.0f;
        h_b[i] = (float)(rand() % 100) / 10.0f;
        h_c[i] = 0.0f;
        h_c_ref[i] = 0.0f;
    }

    //@@ 2. Allocate in device memory.
    float *d_a, *d_b, *d_c;
    CHECK(cudaMalloc((void **)&d_a, size));
    CHECK(cudaMalloc((void **)&d_b, size));
    CHECK(cudaMalloc((void **)&d_c, size));

    // Time: Copy from host to device
    double iStart = cpuSecond();
    //@@ 4. Copy from host memory to device memory.
    CHECK(cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice));
    double iElaps = cpuSecond() - iStart;
    double timeH2D = iElaps * 1000.0;  // Convert to milliseconds
    printf("Host to Device time: %f ms\n", timeH2D);

    //@@ 5. Initialize thread block and thread grid.
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;  // Ceiling division

    printf("Threads per block: %d\n", threadsPerBlock);
    printf("Blocks per grid: %d\n", blocksPerGrid);
    printf("Total threads: %d\n", blocksPerGrid * threadsPerBlock);

    //@@ 6. Invoke the CUDA kernel.
    iStart = cpuSecond();
    vectorAddGPU<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, N);
    CHECK(cudaGetLastError());
    cudaDeviceSynchronize();  // Wait for all GPU threads to complete
    iElaps = cpuSecond() - iStart;
    double timeKernel = iElaps * 1000.0;  // Convert to milliseconds
    printf("Kernel execution time: %f ms\n", timeKernel);

    // Time: Copy from device to host
    iStart = cpuSecond();
    //@@ 7. Copy results from GPU to CPU.
    CHECK(cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost));
    iElaps = cpuSecond() - iStart;
    double timeD2H = iElaps * 1000.0;  // Convert to milliseconds
    printf("Device to Host time: %f ms\n", timeD2H);

    printf("Total GPU time: %f ms\n", timeH2D + timeKernel + timeD2H);

    // Compute CPU reference
    iStart = cpuSecond();
    vectorAddCPU(h_a, h_b, h_c_ref, N);
    iElaps = cpuSecond() - iStart;
    double cpuTime = iElaps * 1000.0;  // Convert to milliseconds
    printf("CPU execution time: %f ms\n", cpuTime);

    //@@ 8. Compare the results with the CPU reference result.
    float maxError = 0.0f;
    int errorCount = 0;
    for (int i = 0; i < N; i++) {
        float error = fabsf(h_c[i] - h_c_ref[i]);
        if (error > maxError) {
            maxError = error;
        }
        if (error > 1e-5) {
            errorCount++;
        }
    }
    printf("Max error: %f\n", maxError);
    if (errorCount > 0) {
        printf("Warning: %d elements have error > 1e-5\n", errorCount);
    } else {
        printf("Results match CPU reference (within tolerance)\n");
    }

    //@@ 10. Free device memory.
    CHECK(cudaFree(d_a));
    CHECK(cudaFree(d_b));
    CHECK(cudaFree(d_c));

    //@@ 9. Free host memory.
    free(h_a);
    free(h_b);
    free(h_c);
    free(h_c_ref);

```

2. **(N-1) operations; 2N reads.**

3. 256 threads per block, **total 2 blocks and 512 threads**.

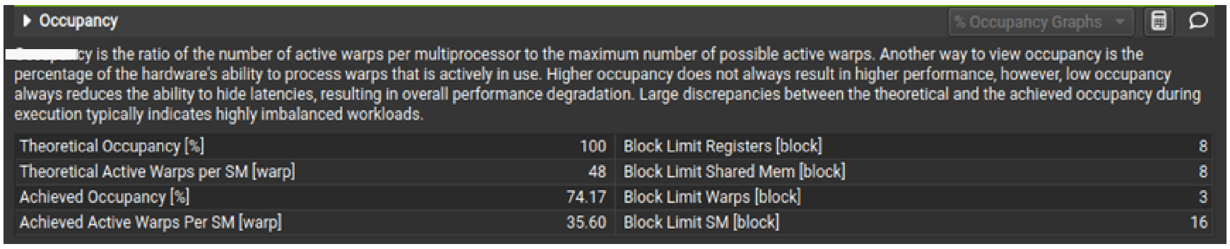
4. The achieved occupancy is **31.86%**.



Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	100	Block Limit Registers [block]	8
Theoretical Active Warps per SM [warp]	48	Block Limit Shared Mem [block]	8
Achieved Occupancy [%]	31.86	Block Limit Warps [block]	3
Achieved Active Warps Per SM [warp]	15.29	Block Limit SM [block]	16

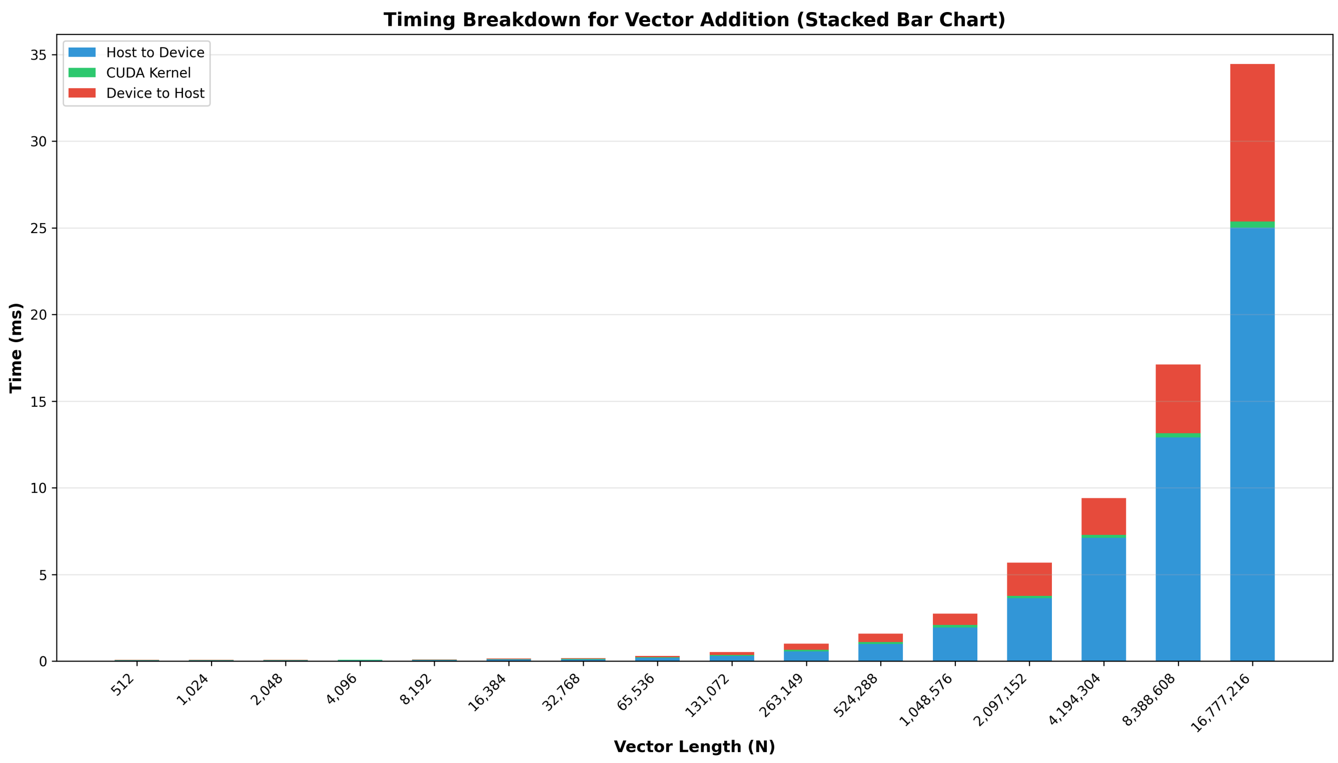
5. My program **still works** with the number since that I use the ceiling for the calculation of block numbers, i.e., `int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock`.
6. **Total 1028 blocks and 263168 threads**.
7. The achieved occupancy is **74.17%**.



[redacted]cy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	100	Block Limit Registers [block]	8
Theoretical Active Warps per SM [warp]	48	Block Limit Shared Mem [block]	8
Achieved Occupancy [%]	74.17	Block Limit Warps [block]	3
Achieved Active Warps Per SM [warp]	35.60	Block Limit SM [block]	16

8. See the stacked bar chart below:



Q[3] Submission Status

- Questions:

1.
```
FLOPs = 2 × numARows × (numAColumns-1) × numBColumns
```

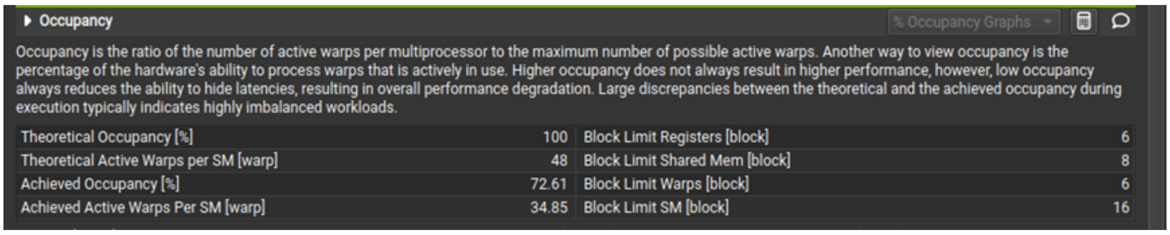
Explanation:

- Each element $C[i][j]$ requires computing the dot product of row i of A and column j of B
- For each element $C[i][j]$:

        - `numAColumns` multiplications: $A[i][k] \times B[k][j]$ for k = 0 to `numAColumns`-1
        - (`numAColumns` - 1) additions: summing the products
        - Total: `numAColumns` multiplications + (`numAColumns` - 1) additions ≈ 2 × `numAColumns` operations per element
    - Total elements in C: `numARows` × `numBColumns`
2.
    ```
    Total reads = 2 × numARows × numAColumns × numBColumns
    ```

    Explanation:

    - Each thread computes one element $C[i][j]$
    - Each thread reads:
        - One row of A: `numAColumns` reads ($A[i][k]$ for k = 0 to `numAColumns`-1)
        - One column of B: `numAColumns` reads ($B[k][j]$ for k = 0 to `numAColumns`-1)
        - Total per thread: 2 × `numAColumns` reads
    - Number of threads computing: `numARows` × `numBColumns`
3.
    1. **Thread block size:** 16 × 16 = 256 threads per block
    2. **Grid size:**
        1. X dimension: (`numBColumns` + 16 - 1) / 16 = (32 + 15) / 16 = 2 blocks
        2. Y dimension: (`numARows` + 16 - 1) / 16 = (128 + 15) / 16 = 8 blocks
    3. **Total 2 × 8 = 16 blocks**, **and total 16 blocks × 256 threads = 4096 threads**.
    4. The achieved occupancy is **72.61%**

    Occupancy

    Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	100	Block Limit Registers [block]	6
Theoretical Active Warps per SM [warp]	48	Block Limit Shared Mem [block]	8
Achieved Occupancy [%]	72.61	Block Limit Warps [block]	6
Achieved Active Warps Per SM [warp]	34.85	Block Limit SM [block]	16

4. My program **still works** with the number, since that I use the ceiling for the calculation of block numbers, i.e.,

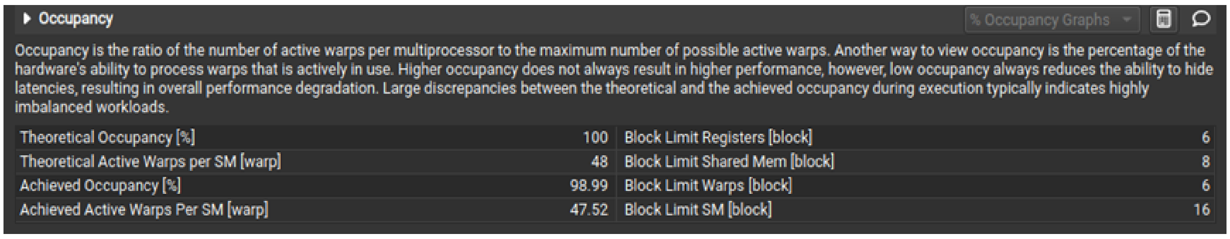
    ```
    dim3 blocksPerGrid((numBColumns + threadsPerBlock.x - 1) / threadsPerBlock.x,
                       (numARows + threadsPerBlock.y - 1) / threadsPerBlock.y);
    ```

    1. **Thread block size:** 16 × 16 = 256 threads per block
    2. **Grid size:**
        1. X dimension: (`numBColumns` + 16 - 1) / 16 = (8197 + 15) / 16 = 513 blocks
        2. Y dimension: (`numARows` + 16 - 1) / 16 = (1024 + 15) / 16 = 64 blocks
        3. Total: 513 × 64 = 32832 blocks
    3. **Total CUDA threads launched:** 32832 blocks × 256 threads = **8404992 threads**
    4. **Threads that actually compute results:** 1024 × 8197 = **8393728 threads**

Explanation:

1. For A(1024×8191) × B(8191×8197), all 8393728 threads compute results
2. The grid is sized to cover all elements of C, and since dimensions are not multiples of block size, 8404992 - 8393728 = 11264 threads are idle.

5. The achieved occupancy is **98.99%**

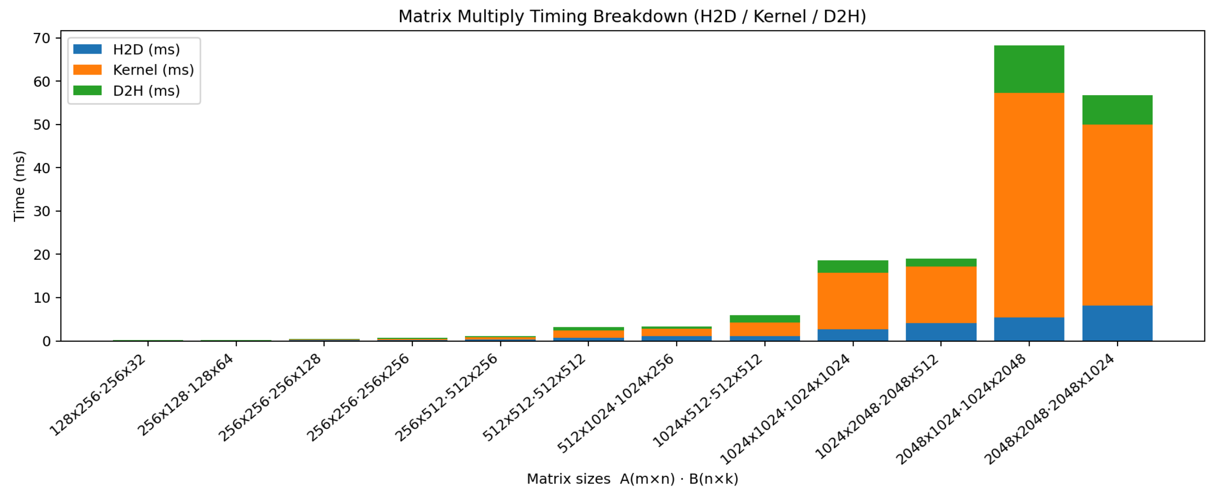


▸ Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

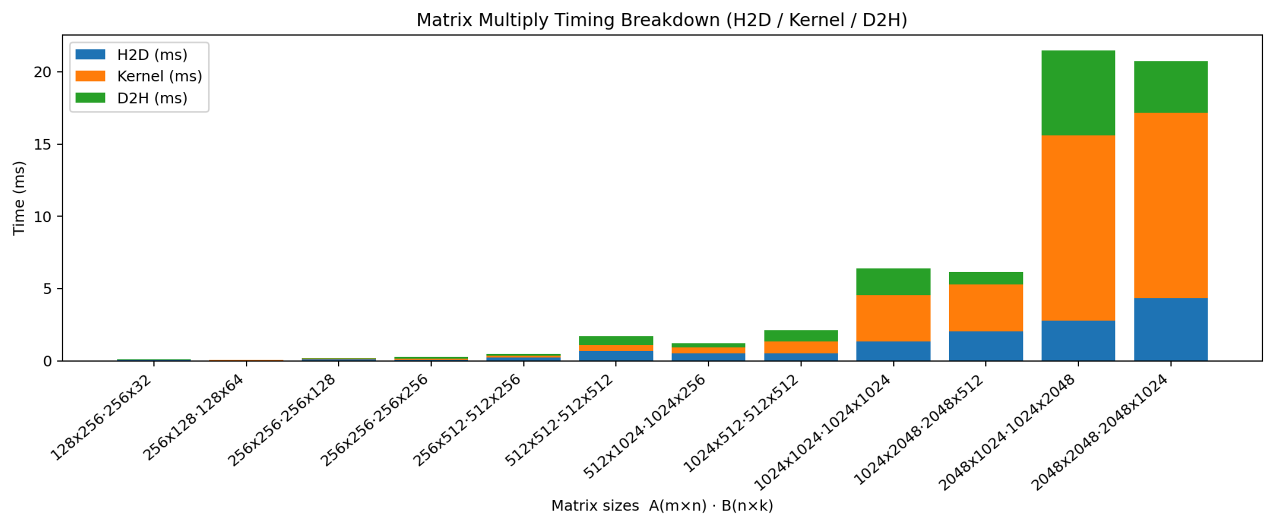
Theoretical Occupancy [%]	100	Block Limit Registers [block]	6
Theoretical Active Warps per SM [warp]	48	Block Limit Shared Mem [block]	8
Achieved Occupancy [%]	98.99	Block Limit Warps [block]	6
Achieved Active Warps Per SM [warp]	47.52	Block Limit SM [block]	16

5.



- I found that with the increase of `FLOPs = 2 × numARows × (numAColumns - 1) × numBColumns`, the overall time cost increases.
- In small matrix calculations, the data transmission time is dominates the overall time. With the size of the matrix increases, the kernel runtime increases significantly, and gradually become the dominant, indicating the advantage of parallel computation.
- Kernel computation scales cubically with problem size, while PCIe transfers (H2D and D2H) scale quadratically. As the matrices grow, data transfer increases steadily, but the kernel—due to its computational load and device-memory bandwidth limits—grows faster and becomes the dominant cost.

6.



- The overall result is the quite the same as the computation with datatype of `double`. While comparing between the two datatypes, since the size of `double` is 2 times of the size of `float`, the overall runtime of `float` operation is 0.3 times of `double` operation .
- After switching from double to float, the bytes moved over PCIe and DRAM are halved, and FP32 has higher hardware throughput with better occupancy, so all three time components decrease. Among them, the kernel drops the most—being limited by both memory bandwidth and compute—so the entire bar becomes significantly shorter.

Q[4] Submission Status

- Questions:

  1.Compile both OMP and CUDA versions of your selected benchmarks. Do you need to make any changes in Makefile?

  No.

2. Ensure the same input problem is used for OMP and CUDA versions. Report and compare their execution time.

Benchmark	Dataset/Size	OpenMP Real (s)	CUDA Real (s)	Speedup (CPU/GPU)
BFS	graph1MW_6.txt	0.64	2.48	0.26
BFS	graph4096.txt	0.06	1.71	0.04
BFS	graph65536.txt	0.05	1.74	0.03
hotspot	64×64	0.14	1.70	0.08
hotspot	512×512	0.51	1.82	0.28
hotspot	1024×1024	0.00	2.21	inf

3. Do you observe expected speedup on GPU compared to CPU? Explain your observations.

There’s no end-to-end speedup on the GPU; it’s consistently slower than the OpenMP CPU runs. Two things dominate: (1) overheads dwarf the work at these sizes—each BFS/hotspot step launches many short kernels, with kernel-launch/synchronization and H2D/D2H copies adding fixed costs that CPUs don’t pay; and (2) the workloads aren’t a good fit for GPU throughput. BFS has highly irregular, pointer-chasing accesses with branch divergence and atomics, while hotspot is a bandwidth-bound stencil with low arithmetic intensity, so ALUs sit under-utilized. In contrast, the CPU keeps data in caches, has low launch overhead, and handles irregular control flow well, so for these small/medium inputs the CPU’s wall-clock wins even though the GPU has higher peak FLOPS.