

## GPU programming - assignment 3

Dr. Cherif Salama

### Team members:

1. Mina Ashraf Gamil - 900182973
2. Mohamed Ayman Mohamed - 900182267

**GPU Model:** GeForce RTX 3090

```
Fri Nov  4 19:33:12 2022
+-----+
| NVIDIA-SMI 520.61.05      Driver Version: 520.61.05      CUDA Version: 11.8      |
+-----+
| GPU  Name      Persistence-M| Bus-Id      Disp.A  Volatile Uncorr. ECC  | | |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M.  |
|                               |             |            | MIG M.   |
+-----+
| 0  NVIDIA GeForce ...  On   | 00000000:02:00.0  On   |                N/A  |
| 0%   41C     P8    25W / 350W |      366MiB / 24576MiB |      6%     Default  |
|                               |                |                N/A  |
+-----+
+-----+
| Processes:                               |
| GPU  GI  CI      PID   Type  Process name          GPU Memory  |
|           ID  ID                  |              Usage  |
|-----|
| 0    N/A N/A      950    G    /usr/lib/xorg/Xorg        194MiB  |
| 0    N/A N/A     1459    G    /usr/bin/gnome-shell       38MiB  |
| 0    N/A N/A    224141    G    ...RendererForSitePerProcess  131MiB  |
+-----+
● cse-p07-g07f@csep07g07f:~/GPU_programming_with_CUDA$ nvidia-smi --query-gpu=name
--format=csv,noheader
NVIDIA GeForce RTX 3090
```

## GeForce and TITAN Products

GPU	Compute Capability
GeForce RTX 3090 Ti	8.6
GeForce RTX 3090	8.6

We are using the 8.6

For the full-table, check this link:

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#features-and-technical-specifications>

Note that the KB and K units used in the following table correspond to 1024 bytes (i.e., a KiB) and 1024 respectively.

Table 15. Technical Specifications per Compute Capability

Technical Specifications	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0	8.6	8.7	8.9	9.0
Maximum number of resident grids per device ( <a href="#">Concurrent Kernel Execution</a> )			32		16	128	32	16	128	16			128			
Maximum dimensionality of grid of thread blocks											3					
Maximum x-dimension of a grid of thread blocks											$2^{31-1}$					
Maximum y- or z-dimension of a grid of thread blocks											65535					
Maximum dimensionality of a thread block											3					
Maximum x-, y- or z-dimension of a block											1024					
Maximum z-dimension of a block											64					
Maximum number of threads per block											1024					
Warp size											32					
Maximum number of resident blocks per SM	16								32			16	32	16	24	32
Maximum number of resident warps per SM								64			32	64	48	64		
Maximum number of resident threads per SM								2048			1024	2048	1536	2048		
Number of 32-bit registers per SM	64 K	128 K									64 K					
Maximum number of 32-bit registers per thread block		64 K			32 K		64 K		32 K				64 K			
Maximum number of 32-bit registers per thread											255					
Maximum amount of shared memory per SM	48 KB	112 KB	64 KB	96 KB	64 KB	96 KB	64 KB	96 KB	64 KB	164 KB	100 KB	164 KB	100 KB	228 KB		
Maximum amount of shared memory per thread block <a href="#">34</a>				48 KB					96 KB	96 KB	64 KB	163 KB	99 KB	163 KB	99 KB	227 KB
Number of shared memory banks									32							
Maximum amount of local memory per thread									512 KB							
Constant memory size									64 KB							
Cache working set per SM for constant memory			8 KB			4 KB				8 KB						
Cache working set per SM for texture memory			Between 12 KB and 48 KB			Between 24 KB and 48 KB		32 - 128 KB		32 or 64 KB	28KB - 192 KB	28KB - 128 KB	28KB - 192 KB	28KB - 128 KB	28KB - 256 KB	
Maximum width for a 1D texture reference bound to a CUDA array			65536								131072					
Maximum width for a 1D texture reference bound to linear memory			$2^{27}$			$2^{28}$		$2^{27}$		$2^{28}$	$2^{27}$				$2^{28}$	
Maximum width and number of layers for a 1D layered texture reference			16384 x 2048								32768 x 2048					
Maximum width and height for a 2D texture reference bound to a CUDA array			65536 x 65536								131072 x 65536					

# Task 1

Experiment 1:

Image Dimension: 6000x4000x1

Using **Top Sobel**:



**Results:**

CPU took: **841137869 ns**

Thus, CPU GFLOPS: **5.42123e+08**

**GPU**

GPU convolution took **948311** nanoseconds (**WITHOUT DATA TRANSFER**)

GPU convolution took **122397442** nanoseconds (**WITH DATA TRANSFER**)

GPU GFLOPS (without Data Transfer): **4.80855e+11**

GPU GFLOPS (with Data Transfer): **3.72557e+09**

We can see that data transfer is the main bottleneck for slowing the parallel algorithm that much. With Data Transfer, It is faster than the CPU by **6.87**, while GPU without Data Transfer is much faster than the CPU by **886.9**

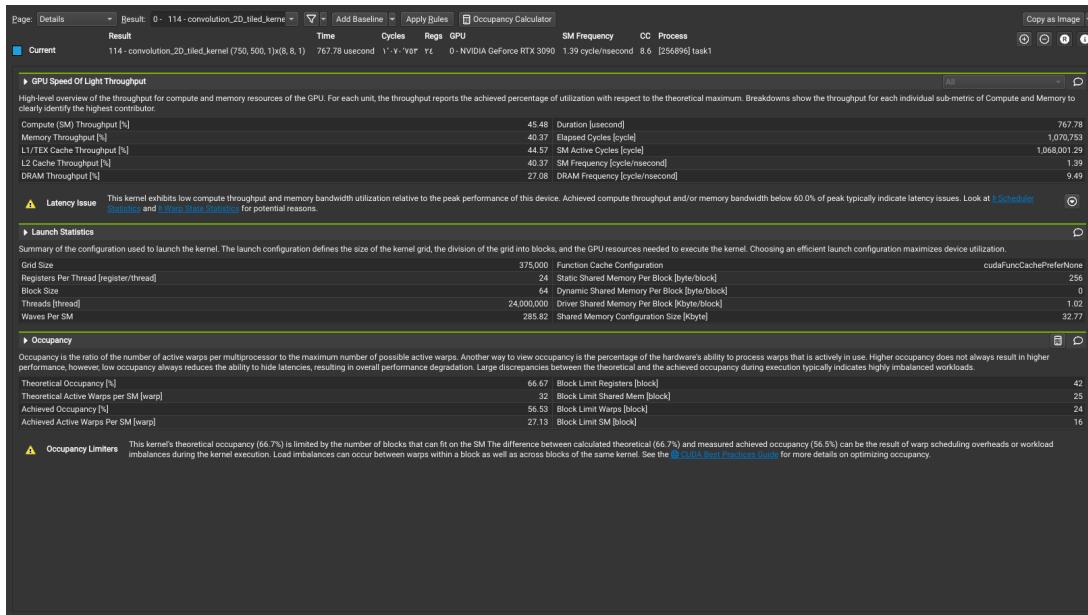
```

● cse-p07-g07f@csep07g07f:~/Ayman and Mina Again/GPU_programming_with_CUDA/assignment 3/Task1$ nvcc -ljpeg --disable-warnings task1.cu -o task1 && ./task1
    enter a number to choose a convolution kernel:
    1. Blur
    2. Emboss
    3. Outline
    4. Sharpen
    5. Left Sobel
    6. Right Sobel
    7. Top Sobel
    8. Bottom Sobel

image dimensions: 6000x4000x1
image basename: 1.jpg
Applying kernel emboss
CPU time: 841137869 ns
CPU GFLOPS: 5.42123e+08
result image saved to 1.jpg_emboss.jpeg
GPU kernel duration: 948311 ns
GPU total duration: 122397442 ns
GPU GFlops without Data Transfer: 4.80855e+11
GPU GFlops with Data Transfer: 3.72557e+09
gpu result image saved to 1.jpg_emboss_gpu.jpeg
CPU is the same as GPU results

```

## GPU Utilization profiling and occupancy analysis: (ZOOM IN for more readability)



Experiment 2:

**Results:**

Using **Bottom Sobel**:



CPU took: **635346104 ns**

Thus, CPU GFLOPS: **4.81021e+08**

**GPU**

GPU convolution took **929513** nanoseconds (**WITHOUT DATA TRANSFER**)

GPU convolution took **99401179** nanoseconds (**WITH DATA TRANSFER**)

GPU GFLOPS (without Data Transfer): **3.2879e+11**

GPU GFLOPS(with Data Transfer): **3.07456e+09**

We can see that data transfer is the main bottleneck for slowing the parallel algorithm that much. With Data Transfer, It is faster than the CPU by **6.3**, while GPU without Data Transfer is much faster than the CPU by **683.5**

```

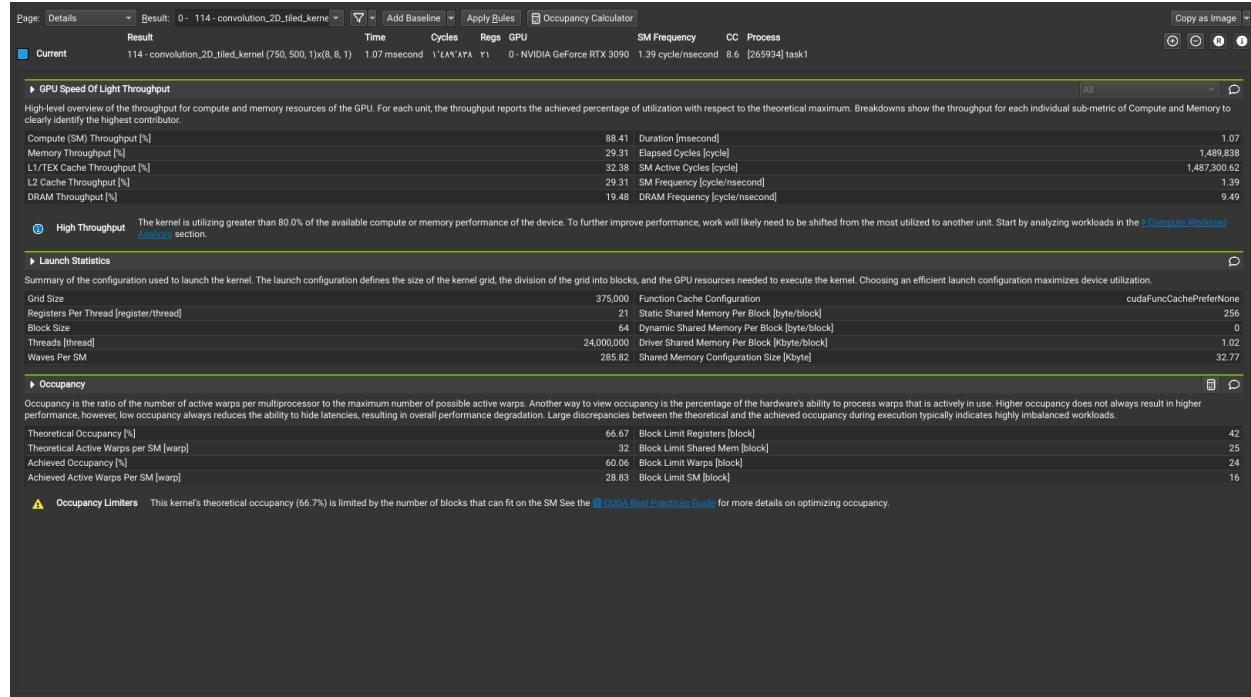
● cse-p07-g07f@csep07g07f:~/Ayman_and_Mina/Again/GPU_programming_with_CUDA/assignment_3/Task1$ nvcc -ljpeg --disable-warnings task1.cu -o task1 && ./task1
enter a number to choose a convolution kernel:
1. Blur
2. Emboss
3. Outline
4. Sharpen
5. Left Sobel
6. Right Sobel
7. Top Sobel
8. Bottom Sobel

image dimensions: 4928x3264x1
image basename: husky.jpeg
Applying kernel bottom_sobel
CPU time: 635346104 ns
CPU GFLOPS: 4.81021e+08
result image saved to husky.jpeg_bottom_sobel.jpeg
GPU kernel duration: 929513 ns
GPU total duration: 99401179 ns
GPU GFlops without Data Transfer: 3.2879e+11
GPU GFlops with Data Transfer: 3.07456e+09
gpu result image saved to husky.jpeg_bottom_sobel.gpu.jpeg
CPU is the same as GPU results

○ cse-p07-g07f@csep07g07f:~/Ayman_and_Mina/Again/GPU_programming_with_CUDA/assignment_3/Task1$ █

```

## GPU Utilization profiling and occupancy analysis: (ZOOM IN for more readability)



### Experiment 3: Outline



CPU took: **2695977 ns**

Thus, CPU GFLOPS: **4.61867e+08**

### GPU

GPU convolution took **24940** nanoseconds (**WITHOUT DATA TRANSFER**)

GPU convolution took **85843748** nanoseconds (**WITH DATA TRANSFER**)

GPU GFLOPS (without Data Transfer): **4.99272e+10**

GPU GFLOPS(with Data Transfer): **1.45052e+07**

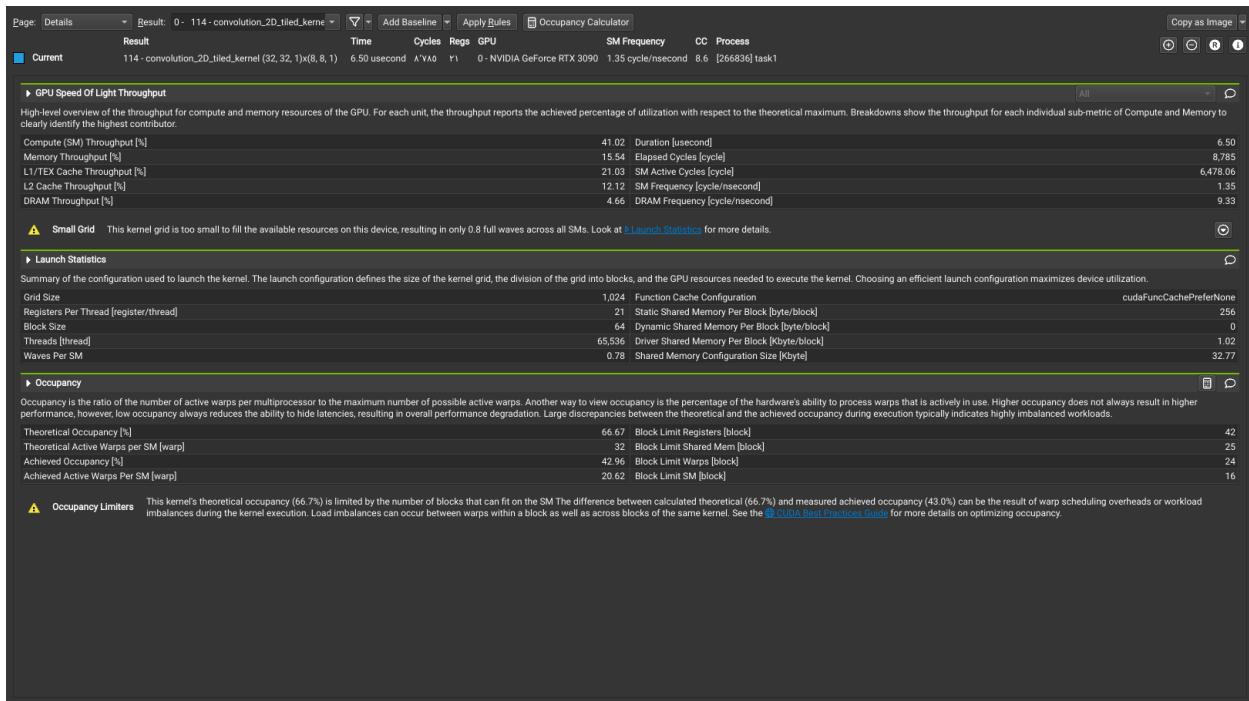
We can see that data transfer is the main bottleneck for slowing the parallel algorithm that much. With Data Transfer, CPU is faster than GPU with data transfer, while GPU without Data Transfer is much faster than the CPU by **108.5. Compared** to other experiments, we can see that GPU is much faster with larger images.

```
• cse-p07-g07f@csep07g07f:~/Ayman_and_Mina/Again/GPU_programming_with_CUDA/assignment_3/Task1$ nvcc -ljpeg --disable-warnings task1.cu -o task1 && ./task1
enter a number to choose a convolution kernel:
1. Blur
2. Emboss
3. Outline
4. Sharpen
5. Left Sobel
6. Right Sobel
7. Top Sobel
8. Bottom Sobel

image dimensions: 256x256x1
image basename: lena.jpeg
Applying kernel outline
CPU time: 2695977 ns
CPU GFLOPS: 4.61867e+08
result image saved to lena.jpeg_outline.jpeg
GPU kernel duration: 24940 ns
GPU total duration: 85843748 ns
GPU Gflops without Data Transfer: 4.99272e+10
GPU Gflops with Data Transfer: 1.45052e+07
gpu result image saved to lena.jpeg_outline_gpu.jpeg
CPU is the same as GPU results

• cse-p07-g07f@csep07g07f:~/Ayman_and_Mina/Again/GPU_programming_with_CUDA/assignment_3/Task1$
```

## GPU Utilization profiling and occupancy analysis: (ZOOM IN for more readability)



# Task 2

## Experiment 1:

Using 6000x4000 Image size

CPU took: **0.100483 seconds (100483205 ns)**

Thus, CPU GFLOPS: **9.52928e+08**

## GPU

Total Number Of Blocks: **6 \* 4000 + 4 \* 6000 = 48000**

Total number of threads: **1024 \* 6 \* 4000 + 4 \* 1024 \* 6000 = 49152000**

GPU 2D-PrefixSum took **4719036** nanoseconds (WITHOUT DATA TRANSFER)

GPU 2D-PrefixSum took **150374009** nanoseconds (WITH DATA TRANSFER)

GPU GFLOPS (without Data Transfer): **1.45052e+11**

GPU GFLOPS(with Data Transfer): **4.52022e+09**

We can see that data transfer is the main bottleneck for slowing the parallel algorithm that much. With Data Transfer, It is faster than the CPU by **4.8**, while GPU without Data Transfer is much faster than the CPU by **76.4**

```
cse-p07-g07f@csep07g07f:~/Ayman_and_Mina/Again/GPU_programming_with_CUDA/assignm
ent 3/Task2$ nvcc -ljpeg --disable-warnings task2.cu -o task2 && ./task2
5 2 5 2
3 6 3 6
5 2 5 2
3 6 3 6

Results:

5 7 12 14
8 16 24 32
13 23 36 46
16 32 48 64

Test passed
Test passed!!
image dimensions: 6000x4000x1
image basename: 1.jpg
CPU GFLOPS: 9.52928e+08
CPU duration: 100742149 ns
The Generalized Kernel of GPU
GPU duration without Data Transfer: 4709688 ns
GPU duration with Data Transfer: 150622138 ns
GPU GFLOPS Without Transfer: 1.4534e+11
GPU GFLOPS with Transfer: 4.54453e+09
Comparing 24000000 elements
Test passed
```

Test Query functionality:

```

Computing 24000000 elements
Test passed
The GPU Test Passed
Enter the number of queries to execute :
1
Enter query 0 :
Enter A(x1,y1)
0 0
Enter B(x2,y2)
1 0
Enter C(x3,y3)
0 1
Enter D(x4,y4)
1 1
Result: 11

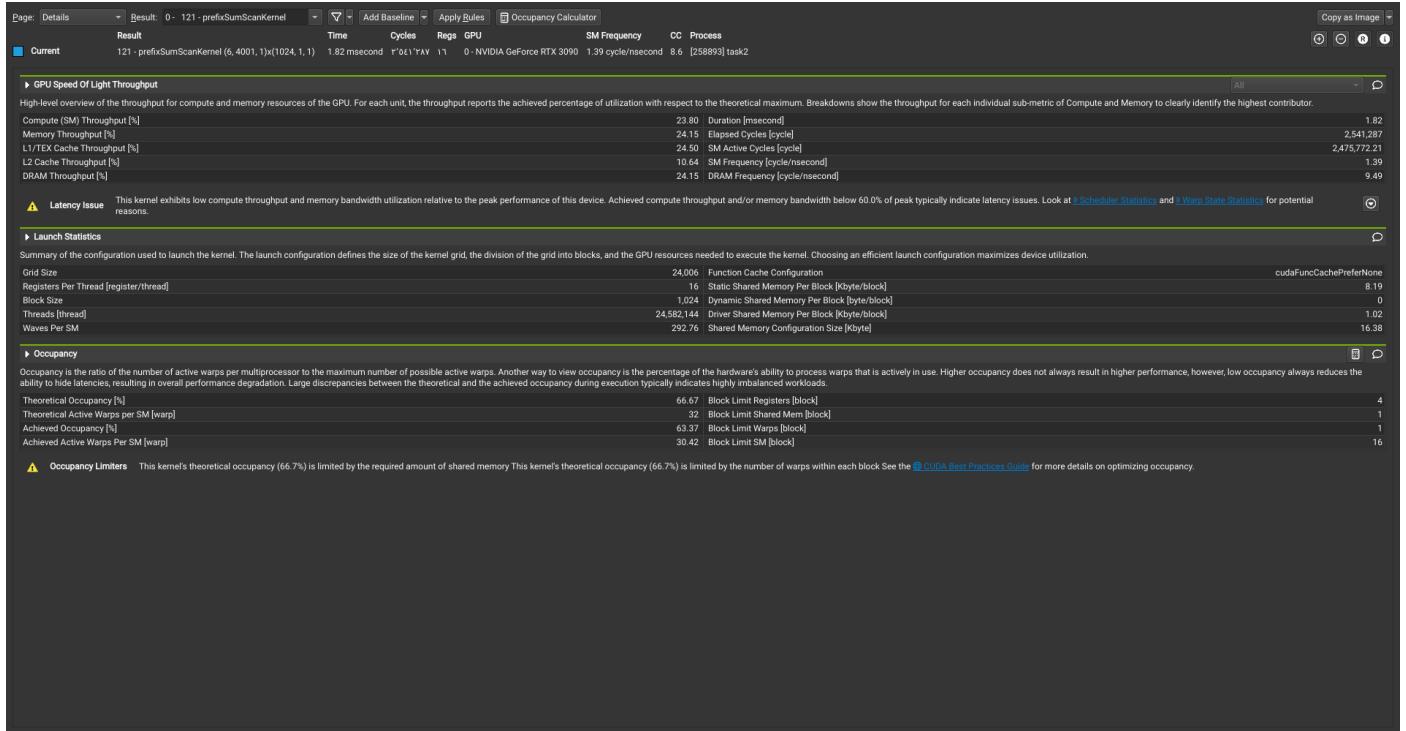
```

cse-p07-g07f@csep07g07f:~/Ayman\_and\_Mina/Again/GPU\_programming\_with\_CUDA/assignment\_3/Task2\$

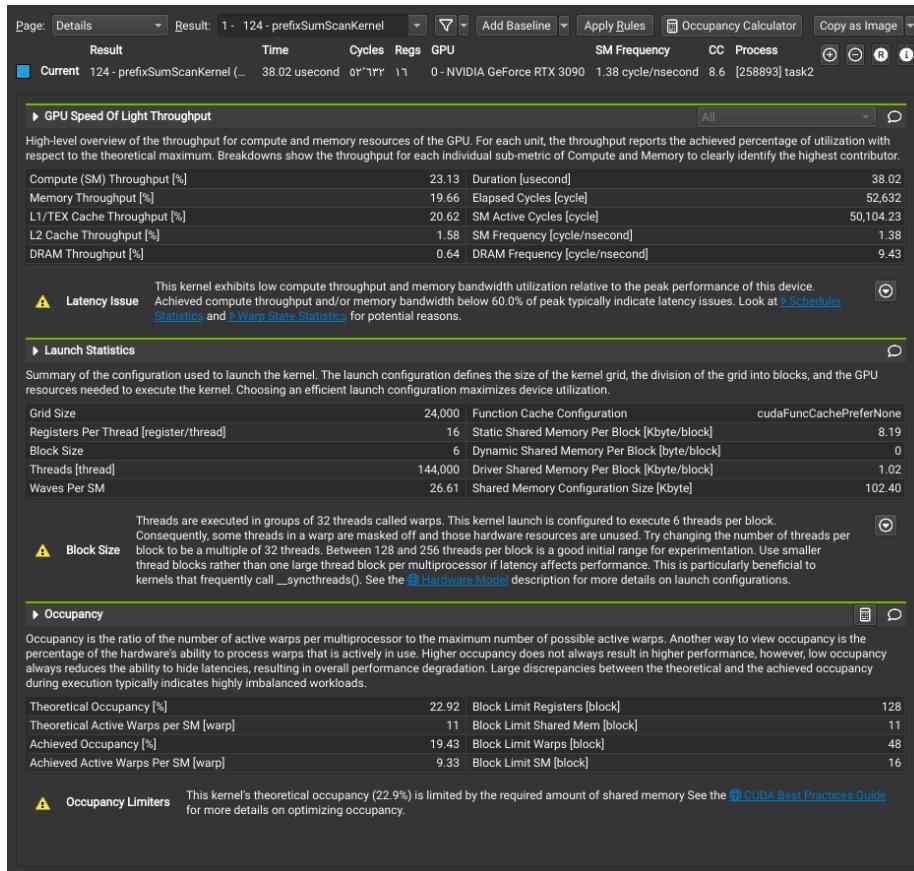
## GPU Utilization profiling and occupancy analysis: (ZOOM IN for more readability): 8 kernels: 1 for row, 1, col, 2x prefix\_sum to section, 2x propagate, 2x transpose.

Page	Summary	Result: 0 - 121 - prefixSumScanKernel	▼	Add Baseline	Apply Rules	Occupancy Calculator	Save as PDF					
ID	Issues	Detailed Function Name	Demangled Name	Process	Device N	Grid Size	Block Size	Cycles [cycle]	Duration [msecond]	Compute Thro	Memory Throughput	# Registers [register/thread]
-	-	Y prefixSumScanKernel	prefixSumScanKernel([213819] ta...)	NVIDI...	6,4001	1	1024, 1, 1	2,540,739	1.82	23.80	23.87	16
\	\	Y prefixSumScanKernel	prefixSumScanKernel([213819] ta...)	NVIDI...	6,4000	1	6, 1, 1	52,617	0.04	23.11	19.65	16
\	\	Y addRowSectionsKernel	addRowSectionsKernel([213819] ta...)	NVIDI...	6,4001	1	1024, 1, 1	745,027	0.54	23.57	78.32	16
\	\	Y efficientTransposeKernel	efficientTransposeKernel([213819] ta...)	NVIDI...	188,125	1	32, 32, 1	655,116	0.48	16.79	89.04	16
\	\	Y prefixSumScanKernel	prefixSumScanKernel([213819] ta...)	NVIDI...	4,6000	1	1024, 1, 1	2,538,756	1.82	23.82	23.90	16
\	\	Y prefixSumScanKernel	prefixSumScanKernel([213819] ta...)	NVIDI...	4,6000	1	4, 1, 1	41,914	0.03	20.65	18.16	16
\	\	Y addRowSectionsKernel	addRowSectionsKernel([213819] ta...)	NVIDI...	4,6000	1	1024, 1, 1	786,769	0.57	22.59	74.34	16
\	\	Y efficientTransposeKernel	efficientTransposeKernel([213819] ta...)	NVIDI...	125,188	1	32, 32, 1	707,400	0.52	15.56	82.47	16

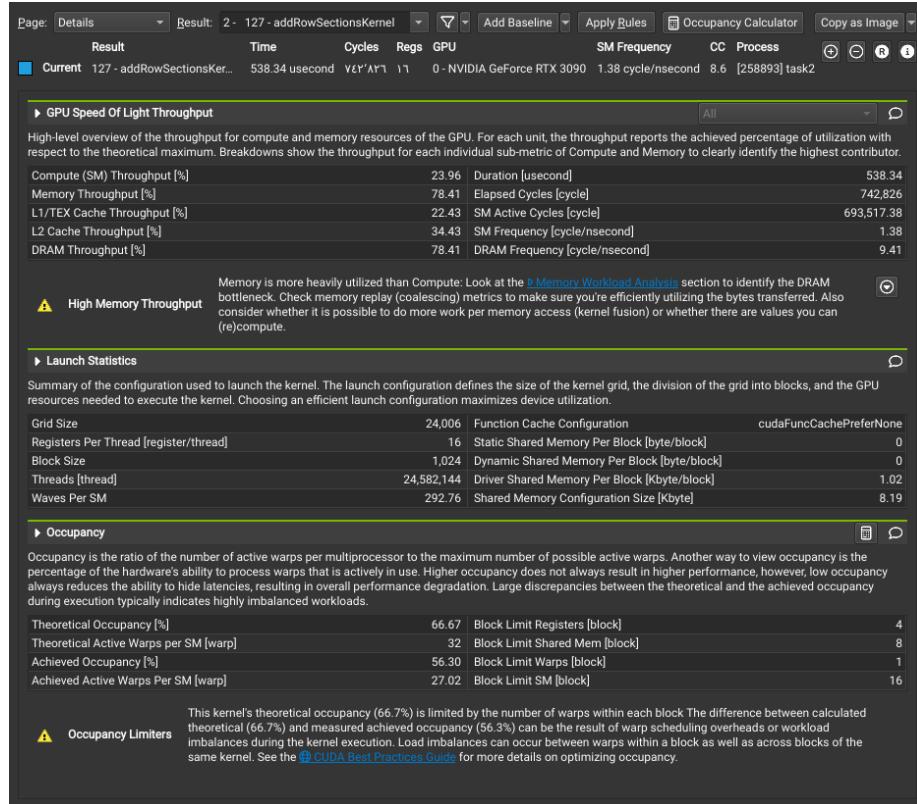
First Kernel:



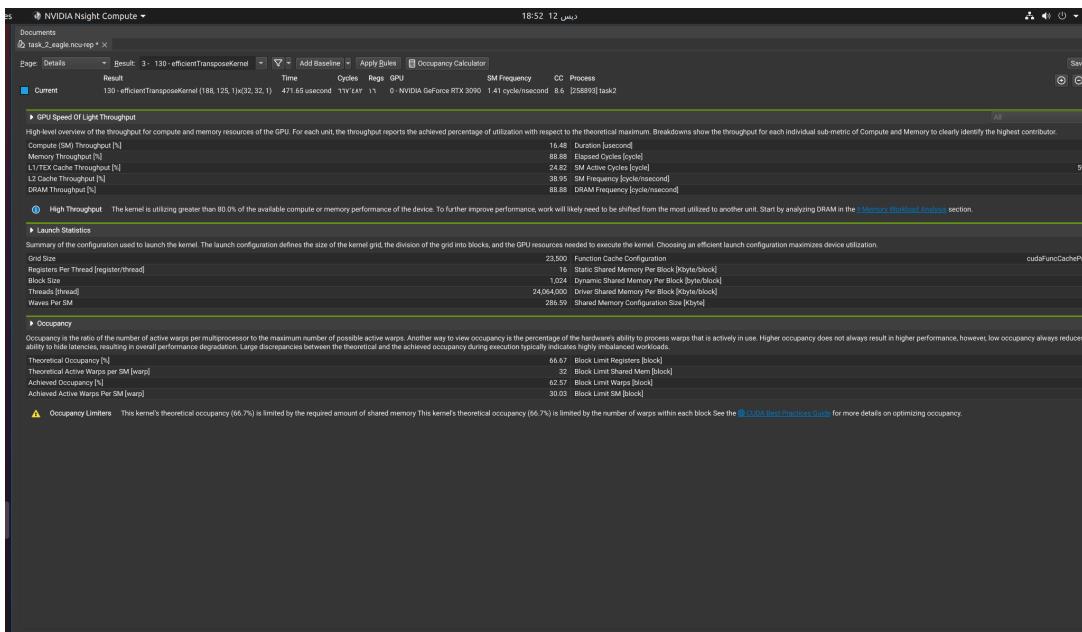
## Second Kernel:



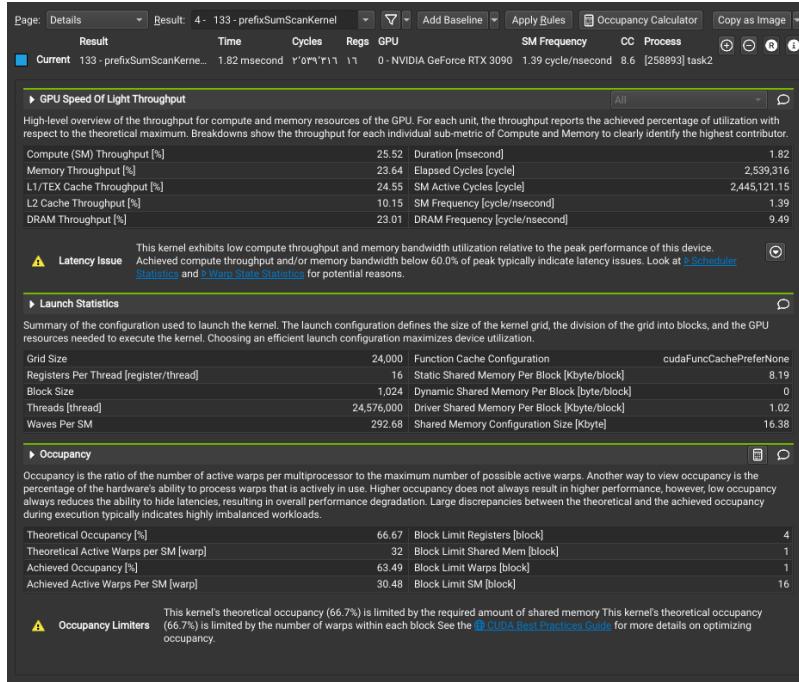
## Third Kernel:



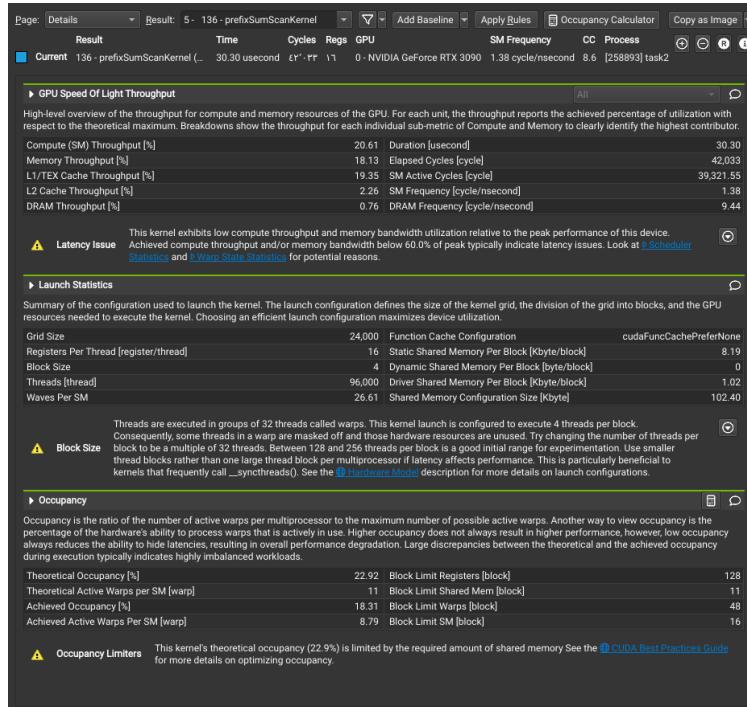
## Fourth Kernel:



## Fifth Kernel:



## Sixth Kernel:



## Seventh Kernel:

Page: Details Result: 6 - 139 - addRowSectionsKernel Add Baseline Apply Rules Occupancy Calculator Copy as Image

Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current 139 - addRowSectionsKer...	565.86 usecond	YY9'AY0	11	0 - NVIDIA GeForce RTX 3090	1.38 cycle/nsecond	8.6	[258893] task2

### GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor.

	Compute (SM) Throughput [%]	Memory Throughput [%]	L1/TEX Cache Throughput [%]	L2 Cache Throughput [%]	DRAM Throughput [%]	Duration [usecond]	Elapsed Cycles [cycle]	SM Active Cycles [cycle]	SM Frequency [cycle/nsecond]	DRAM Frequency [cycle/nsecond]
Current	22.82	74.98	21.31	32.91	74.98	565.86	779,875	729,567.84	1.38	9.39

**⚠️ High Memory Throughput** Memory is more heavily utilized than Compute. Look at the [Memory Workload Analysis](#) section to identify the DRAM bottleneck. Check memory replay (coalescing) metrics to make sure you're efficiently utilizing the bytes transferred. Also consider whether it is possible to do more work per memory access (kernel fusion) or whether there are values you can (re)compute.

### Launch Statistics

Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.

Grid Size	Function Cache Configuration	cudaFuncCacheNone
Registers Per Thread [register/thread]	16 Static Shared Memory Per Block [byte/block]	0
Block Size	1,024 Dynamic Shared Memory Per Block [byte/block]	0
Threads [thread]	24,576,000 Driver Shared Memory Per Block [kbyte/block]	1.02
Waves Per SM	292.68 Shared Memory Configuration Size [kbyte]	8.19

### 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 [%]	Block Limit Registers [block]	4
Theoretical Active Warps per SM [warp]	32 Block Limit Shared Mem [block]	8
Achieved Occupancy [%]	55.37 Block Limit Warps [block]	1
Achieved Active Warps Per SM [warp]	26.58 Block Limit SM [block]	16

**⚠️ Occupancy Limiters** This kernel's theoretical occupancy (66.7%) is limited by the number of warps within each block. The difference between calculated theoretical (66.7%) and measured achieved occupancy (55.4%) can be the result of warp scheduling overheads or workload imbalances during the kernel execution. Load imbalances can occur between warps within a block as well as across blocks of the same kernel. See the [CUDA Best Practices Guide](#) for more details on optimizing occupancy.

## Eighth Kernel:

Page: Details Result: 7 - 142 - efficientTransposeKernel Add Baseline Apply Rules Occupancy Calculator Copy as Image

Result Time Cycles Regs GPU SM Frequency CC Process

Current 142 - efficientTransposeKe... 503.58 usecond Y-Y-ZY- 11 0 - NVIDIA GeForce RTX 3090 1.39 cycle/nsecond 8.6 [258893] task2

**GPU Speed Of Light Throughput**

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor.

Compute (SM) Throughput [%]	15.67	Duration [usecond]	503.58
Memory Throughput [%]	83.07	Elapsed Cycles [cycle]	702,420
L1/TEX Cache Throughput [%]	23.49	SM Active Cycles [cycle]	640,326.22
L2 Cache Throughput [%]	36.40	SM Frequency [cycle/nsecond]	1.39
DRAM Throughput [%]	83.07	DRAM Frequency [cycle/nsecond]	9.49

**High Throughput** The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing DRAM in the [Memory Workload Analysis](#) section.

**Launch Statistics**

Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.

Grid Size	23,500	Function Cache Configuration	cudaFuncCacheNone
Registers Per Thread [register/thread]	16	Static Shared Memory Per Block [Kbyte/block]	8.45
Block Size	1,024	Dynamic Shared Memory Per Block [byte/block]	0
Threads [thread]	24,064,000	Driver Shared Memory Per Block [Kbyte/block]	1.02
Waves Per SM	286.59	Shared Memory Configuration Size [Kbyte]	16.38

**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 [%]	66.67	Block Limit Registers [block]	4
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]	1
Achieved Occupancy [%]	62.87	Block Limit Warps [block]	1
Achieved Active Warps Per SM [warp]	30.18	Block Limit SM [block]	16

**Occupancy Limiters** This kernel's theoretical occupancy (66.7%) is limited by the required amount of shared memory. This kernel's theoretical occupancy (66.7%) is limited by the number of warps within each block. See the [CUDA Best Practices Guide](#) for more details on optimizing occupancy.

## Experiment 2:

Using 4000x2667 image size

CPU took: **47088192 ns**

Thus, CPU GFLOPS: **9.06214e+08**

## GPU

GPU 2D-PrefixSum took **3334478** nanoseconds (**WITHOUT DATA TRANSFER**)

GPU 2D-PrefixSum took **117054717** nanoseconds (**WITH DATA TRANSFER**)

GPU GFLOPS (without Data Transfer): **8.75027e+10**

GPU GFLOPS(with Data Transfer): **2.49264e+09**

We can see that data transfer is the main bottleneck for slowing the parallel algorithm that much. With Data Transfer, It is faster than the CPU by **2.8**, while GPU without Data Transfer is much faster than the CPU by **96.6**

```
Test passed
Test passed!!
image dimensions: 4000x2667x1
image basename: greek.jpg
CPU GFLOPS: 9.06214e+08
CPU duration: 47088192 ns
The Genralized Kernel of GPU
GPU duration without Data Transfer: 3334478 ns
GPU duration with Data Transfer: 117054717 ns
GPU GFLOPS Without Transfer: 8.75027e+10
GPU GFLOPS with Transfer: 2.49264e+09
Comparing 10668000 elements
Test passed
The GPU Test Passed
cse-p07-g07f@csep07g07f:~/Ayman_and_Mina Again/GPU_programming_with_CUDA/assignment_3/Task2$
```

## Test Query functionality:

```
The GPU Test Passed
Enter the number of queries to execute :
1
Enter query 0 :
Enter A(x1,y1)
5 5
Enter B(x2,y2)
100 5
Enter C(x3,y3)
5 100
Enter D(x4,y4)
100 100
Result: 1037511
cse-p07-g07f@csep07g07f:~/Ayman_and_Mina Again/GPU_programming_with_CUDA/assignment_3/Task2$
```

## GPU Utilization profiling and occupancy analysis: (ZOOM IN for more readability):

I will add just an overview here.

Page	Summary	Result: 0 - 121 - prefixSumScanKernel	▼	Add Baseline	Apply Rules	Occupancy Calculator	Copy as Image					
ID	Issues Detected	Function Name	Demangled Name	Process	Device N	Grid Size	Block Size	Cycles [cycle]	Duration [usecond]	Compute Thro	Memory Throughput	# Registers [register/thread]
-	1	Y prefixSumScanKernel	prefixSumScanKernel([on... [266514] ta... NVIDI...	4, 2668,	1	1024, 1, 1	1,133,217	812.70	23.72	23.79		16
1	Y prefixSumScanKernel	prefixSumScanKernel([on... [266514] ta... NVIDI...	4, 2667,	1	4, 1, 1	20,864	15.26	18.52	16.25			16
Y	Y addRowSectionsKernel	addRowSectionsKernel([on... [266514] ta... NVIDI...	4, 2668,	1	1024, 1, 1	353,519	250.14	22.39	74.05			16
Y	Y efficientTransposeKernel	efficientTransposeKernel([on... [266514] ta... NVIDI...	125, 84,	1	32, 32, 1	323,634	233.25	15.21	79.61			16
t	Y prefixSumScanKernel	prefixSumScanKernel([on... [266514] ta... NVIDI...	3, 4000,	1	1024, 1, 1	1,274,310	913.63	23.73	23.76			16
o	Y prefixSumScanKernel	prefixSumScanKernel([on... [266514] ta... NVIDI...	3, 4000,	1	3, 1, 1	22,683	16.35	18.79	16.78			16
Y	Y addRowSectionsKernel	addRowSectionsKernel([on... [266514] ta... NVIDI...	3, 4000,	1	1024, 1, 1	364,225	263.39	24.17	70.57			16
v	Y efficientTransposeKernel	efficientTransposeKernel([on... [266514] ta... NVIDI...	84, 125,	1	32, 32, 1	299,468	214.59	16.38	85.71			16

# Task 3

## Experiment 1: with 255 bins

834x390x3 image Size

### Results:

CPU took: **3191393 ns**

Thus, CPU GFLOPS: **2.03836e+08**

### GPU

GPU convolution took **89650** nanoseconds (**WITHOUT DATA TRANSFER**)

GPU convolution took **101840242** nanoseconds (**WITH DATA TRANSFER**)

GPU GFLOPS (without Data Transfer): **7.25622e+09**

GPU GFLOPS (with Data Transfer): **6.38765e+06**

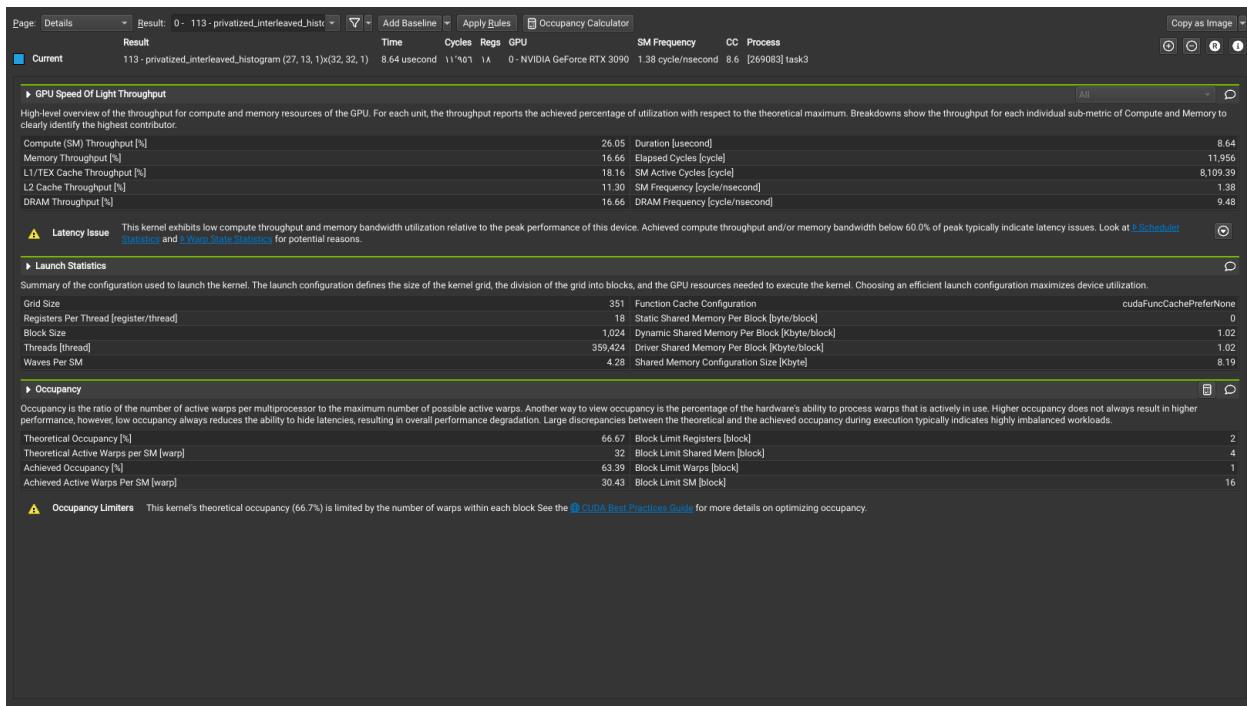
We can see that data transfer is the main bottleneck for slowing the parallel algorithm that much. With Data Transfer, It is not faster than CPU, as the image is very small. However, GPU without Data Transfer is much faster than the CPU by **35.7**

```
● cse-p07-g07f@csep07g07f:~/Ayman_and_Mina/Again/GPU_programming_with_CUDA/assignment_3/Task3$ nvcc -ljpeg --disable-warnings task3.cu -o task3 && ./task3 ./man.jpg
1: test_compute_histogram passed!
2: test_compute_histogram passed!!

Image width: 834
Image height: 390
Image channels: 3
Image size: 975780

Please enter the number of histogram bins to use: 255
CPU Results
CPU compute histogram time: 3191393 ns
CPU compute histogram GFLOPS: 2.03836e+08
GPU Results
GPU compute histogram time without Transfer: 89650 ns
GPU compute histogram time with Transfer: 101840242 ns
GPU compute histogram GFLOPS without Transfer: 7.25622e+09
GPU compute histogram GFLOPS with Transfer: 6.38765e+06
GPU compute histogram passed!
○ cse-p07-g07f@csep07g07f:~/Ayman_and_Mina/Again/GPU_programming_with_CUDA/assignment_3/Task3$
```

## GPU Utilization profiling and occupancy analysis: (ZOOM IN for more readability):



## Experiment 2: with 256 bins

615x418x3 image Size

CPU took: **555549 ns**

Thus, CPU GFLOPS: **9.25463e+08**

## GPU

GPU convolution took **87684** nanoseconds (**WITHOUT DATA TRANSFER**)

GPU convolution took **90681306** nanoseconds (**WITH DATA TRANSFER**)

GPU GFLOPS (without Data Transfer): **5.86356e+09**

GPU GFLOPS(with Data Transfer): **5.66975e+06**

We can see that data transfer is the main bottleneck for slowing the parallel algorithm that much. With Data Transfer, It is not faster than CPU, as the image is very small. However, GPU without Data Transfer is much faster than the CPU by **35.7**

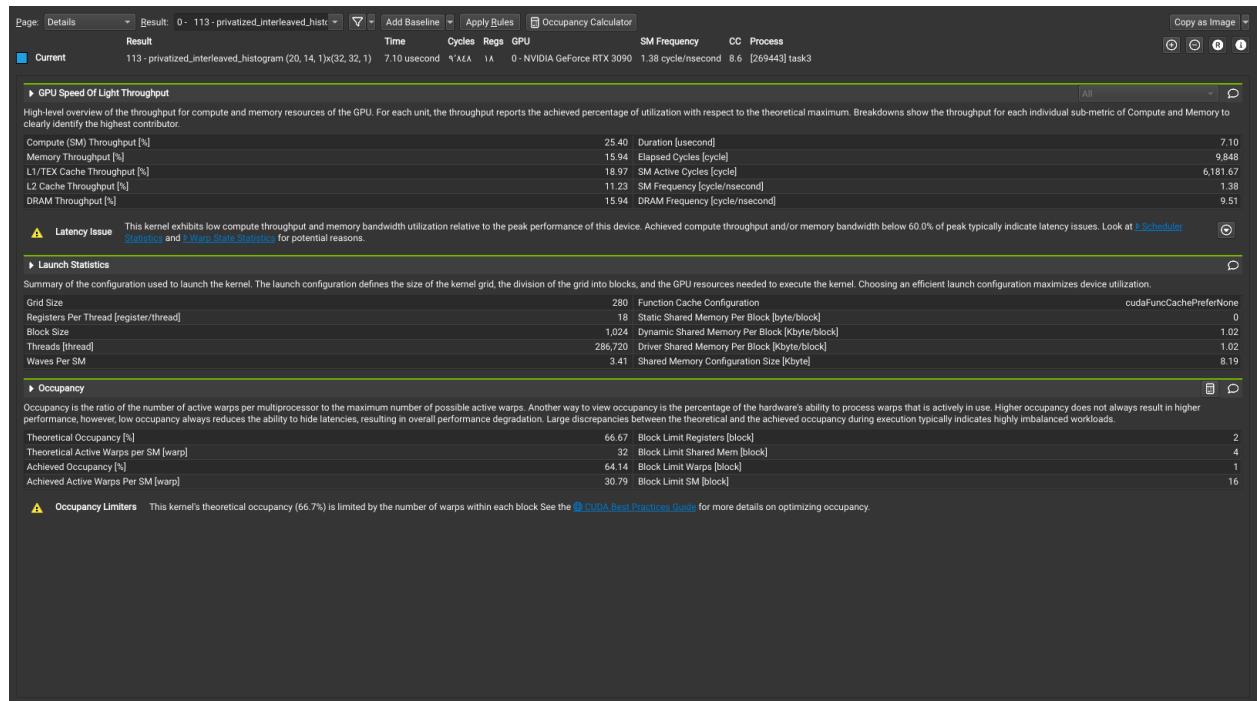
```
cse-p07-g07f@sep07g07f:~/Ayman_and_Mina/Again/GPU_programming_with_CUDA/assignment_3/Task3$ nvcc -ljpeg --disable-warnings task3.cu -o task3 && ./task3
1: test_compute_histogram passed!!
2: test_compute_histogram passed!!

Image width: 615
Image height: 418
Image channels: 3
Image size: 771210

Please enter the number of histogram bins to use: CPU Results
CPU compute histogram time: 555549 ns
CPU compute histogram GFLOPS: 9.25463e+08
GPU Results
GPU compute histogram time without Transfer: 87684 ns
GPU compute histogram time with Transfer: 90681306 ns
GPU compute histogram GFLOPS without Transfer: 5.86356e+09
GPU compute histogram GFLOPS with Transfer: 5.66975e+06
GPU_compute_histogram passed!!

cse-p07-g07f@sep07g07f:~/Ayman_and_Mina/Again/GPU_programming_with_CUDA/assignment_3/Task3$
```

## GPU Utilization profiling and occupancy analysis: (ZOOM IN for more readability):



### Experiment 3: 150 bins

6000x4000x3 image Size

CPU took: **50854556 ns**

Thus, CPU GFLOPS: **9.43868e+08**

### GPU

GPU convolution took **315192** nanoseconds (**WITHOUT DATA TRANSFER**)

GPU convolution took **93244658** nanoseconds (**WITH DATA TRANSFER**)

GPU GFLOPS (without Data Transfer): **1.52288e+11**

GPU GFLOPS(with Data Transfer): **5.14775e+08**

We can see that data transfer is the main bottleneck for slowing the parallel algorithm that much. With Data Transfer, It is not faster than CPU, as the image is very small. However, GPU without Data Transfer is much faster than the CPU by **161.34**

```
● cse-p07-g07f@csep07g07f:~/Ayman_and_Mina/Again/GPU_programming_with_CUDA/assignment_3/Task3$ nvcc -ljpeg --disable-warnings task3.cu -o task3 && ./task3
1: test_compute_histogram passed!!
2: test_compute_histogram passed!!

Image width: 6000
Image height: 4000
Image channels: 3
Image size: 72000000

Please enter the number of histogram bins to use: CPU Results
CPU compute histogram time: 50854556 ns
CPU compute histogram GFLOPS: 9.43868e+08
GPU Results
GPU compute histogram time without Transfer: 315192 ns
GPU compute histogram time with Transfer: 93244658 ns
GPU compute histogram GFLOPS without Transfer: 1.52288e+11
GPU compute histogram GFLOPS with Transfer: 5.14775e+08
GPU_compute_histogram passed!!

cse-p07-g07f@csep07g07f:~/Ayman_and_Mina/Again/GPU_programming_with_CUDA/assignment_3/Task3$ █
Live Share
You, now Ln 215, Col 23 Spaces: 4 UTF-8 LF
```

## GPU Utilization profiling and occupancy analysis: (ZOOM IN for more readability):

