Group Assignment: Prefix Sum/Scan

Course Number	**MENG 3540**
Course Title	**Parallel Programming**
Semester/Year	**06/24**
Submission Due Date	**07/03/24**

Student Name	Student No.	Total Mark
Ian Cameron	**N00009345**	**/100**
Alexander Colatosti	**N01104675**	**/100**
Amelia Soon	**N01519374**	**/100**

* By signing above, you attest that you have contributed to this submission and confirm that all work you have contributed to this submission is your work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a ZERO on the work or possibly more severe penalties.
https://academic-regulations.humber.ca/2021-2022/17.0-ACADEMIC-MISCONDUCT

Group Assignment: Prefix Sum/Scan

SUMMARY

The purpose of this report is to research and familiarize ourselves with one of the fundamental parallel computations, prefix sum/scan, which is used in a variety of different parallel applications. The function of prefix sum/scan is to calculate the cumulative sum of each element of an array into an output array. This output array can then be used for sorting or searching, or in image processing it can help prepare data for filtering or morphological operations. Prefix sum/scan is often used to prepare data for other parallel operations, but can also be useful after data preparation.

Section I. – Original Code

Our basic code will take the values from Array 1 (input) and add then to the previous value in array 2 (output). For our application we would like to use prefix sum/scan to identify objects from an array of data from an environment scan (LiDAR), therefore, we must reduce the input data to identify changes in value and put them into an obstacle array. Since our application is to run object detection on lidar data for an autonomous vehicle, our main measurement factor is going to be the execution time on the computation.

Our simulated data is such that the maximum readable distance is 9. This means that if the the distance read by the LiDAR is 9, there is no obstacle; if the LiDAR reads a distance smaller than 9 (in the case of our sample data, these are represented by the value 3) it will indicate that there is an obstacle at that position.

```
//Populate with distances
for (int i = 0; i < N; i++) {
  int j = 8;
  if (i % j == 0) {
    data[i] = 3;
  }
  else {
    data[i] = 9;
  }
}
```

```
//Check if obstacle detected
for (int i = 0; i < N; i++) {
  if (data[i] < MAX) {
    h_obstacle[i] = 1;
  }
  else {
    h_obstacle[i] = 0;
  }
```

Given this data, we can apply the prefix sum to the obstacle array to determine the number of obstacles detected by the LiDAR sensor. A preliminary version of the code (called the brute force method) has each term *i* of the prefix sum matrix equal to the sum of the first to the (i-1)th terms.

Source Code

```
__global__ void scan1(int* X, int* Y, int InputSize) {
  __shared__ int XY[N];
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  Y[i] = 0;
  for (int j = 0; j <= i; j++) {
    Y[i] += X[j];
  }
}
```

Initial Results

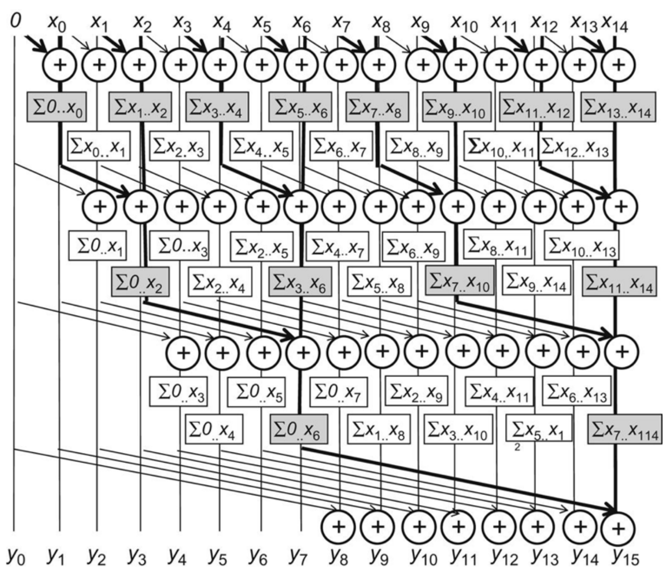
```
Time (Brute-Force): 0.295968 ms
Prefix Sum Array:
  1  1  1  1  1  1  1  1  2  2  2  2  . . .  31 31 31 31 32 32 32 32 32 32 32 32
 32 obstacles detected
```

Already we can see potential methods to optimize the code, such as minimizing the number of global writes by using a variable to record the sum instead of writing directly to the global memory in each loop; further optimizations are described in Section II.

Section II. – Optimizations

We have two optimization techniques that we will use to increase the performance of the prefix sum/scan operation. By applying various algorithms which modify the order in which calculations are executed, we can theoretically reduce the number of operations, global reads, and global writes that are executed by the kernel, resulting in a shorter execution time and smaller memory bandwidth.

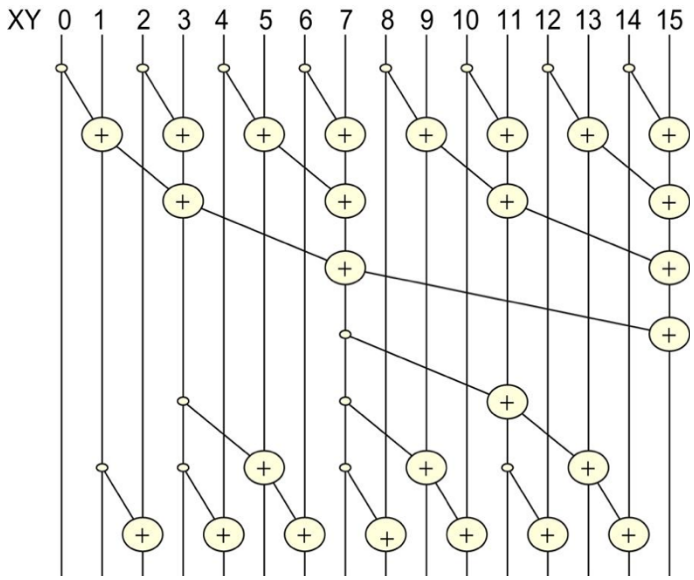
Both methods use some algorithm (Kogge-Stone or Brent-Kung) to create a reduction tree reducing the number of required operations, along with using shared memory to reduce the number of global memory reads.



Representation of Kogge-Stone algorithm (Source: Chang and Gomez-Luna)

Source Code for Kogge-Stone Method

```
__global__ void scan2(int* X, int* Y, int InputSize) {
  __shared__ int XY[N];
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < InputSize) {
    XY[threadIdx.x] = X[i];
  }
  for (int j = 1; j < blockDim.x; j *= 2) {
    __syncthreads();
    if (threadIdx.x >= j) {
      XY[threadIdx.x] += XY[threadIdx.x - j];
    }
    Y[i] = XY[threadIdx.x];
  }
}
```



Representation of Brent-Kung adder design (Source: Chang and Gomez-Luna)

Source Code for Brent-Kung Method

```
__global__ void scan3(int* X, int* Y, int InputSize) {
  __shared__ int XY[N];
  int i = 2 * blockIdx.x * blockDim.x + threadIdx.x;
  if (i < InputSize) {
    XY[threadIdx.x] = X[i];
  }
  if (i + blockDim.x < InputSize) {
    XY[threadIdx.x + blockDim.x] = X[i + blockDim.x];
  }

  for (int j = 1; j <= blockDim.x; j *= 2) {
    __syncthreads();
    int index = (threadIdx.x + 1) * 2 * j - 1;
    if (index < N) {
      XY[index] += XY[index - j];
    }
  }

for (int j = N / 4; j > 0; j /= 2) {
    __syncthreads();
    int index = (threadIdx.x + 1) * j * 2 - 1;
    if (index + j < N) {
      XY[index + j] += XY[index];
    }
  }

  __syncthreads();
```

```
  if (i < InputSize) {
    Y[i] = XY[threadIdx.x];
  }
  if (i + blockDim.x < InputSize) {
    Y[i + blockDim.x] = XY[threadIdx.x + blockDim.x];
  }
}
```

Results

```
Data Array:
3  9  9  9  9  9  9  9  3  9  9  9   . . .  9  9  9  9  3  9  9  9  9  9  9  9
Obstacle Array:
1  0  0  0  0  0  0  0  1  0  0  0   . . .  0  0  0  0  1  0  0  0  0  0  0  0

Time (Brute-Force): 1.320960 ms
Prefix Sum Array:
  1  1  1  1  1  1  1  1  2  2  2  2  . . . 127127127127128128128128128128128128
128 obstacles detected

Time (Brute-Force): 0.749376 ms
Prefix Sum Array:
  1  1  1  1  1  1  1  1  2  2  2  2  . . . 127127127127128128128128128128128128
128 obstacles detected

Time (Kogge-Stone): 0.068992 ms
Prefix Sum Array:
  1  1  1  1  1  1  1  1  2  2  2  2  . . . 127127127127128128128128128128128128
128 obstacles detected

Time (Brent-Kung): 0.068096 ms
Prefix Sum Array:
  1  1  1  1  1  1  1  1  2  2  2  2  . . . 127127127127128128128128128128128128
128 obstacles detected
```

Method	Floating Point Operations	Global Reads	Global Writes
Brute Force	$(N+1)N/2$	$(N+1)N/2$	$(N+1)N/2 + N$
Kogge-Stone	$N \log_2(N)$	N	$N \log_2(N)$
Brent-Kung	$2N - 2 - \log_2(N)$	2N	2N

Results and Review

The results from our optimization were conclusive in that they confirmed that our optimizations were both faster at returning the obstacle count than the brute force method. Running the different kernels through Nsight we can see that the Brute Force method had a runtime of 0.951 ms. Additionally we can verify that our two optimization methods have reduced the number of cycles required to run the operation.

Brute Force

GPU Speed Of Light Throughput — GPU Throughput Chart

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. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	0.80	Duration [usecond]	951.17
Memory Throughput [%]	0.24	Elapsed Cycles [cycle]	1,058,612
L1/TEX Cache Throughput [%]	9.83	SM Active Cycles [cycle]	26,345.08
L2 Cache Throughput [%]	0.24	SM Frequency [cycle/nsecond]	1.11
DRAM Throughput [%]	0.00	DRAM Frequency [cycle/nsecond]	7.01

Kogge-Stone

GPU Speed Of Light Throughput — All

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. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	1.28	Duration [usecond]	24.86
Memory Throughput [%]	0.53	Elapsed Cycles [cycle]	27,336
L1/TEX Cache Throughput [%]	22.57	SM Active Cycles [cycle]	639
L2 Cache Throughput [%]	0.35	SM Frequency [cycle/nsecond]	1.10
DRAM Throughput [%]	0.04	DRAM Frequency [cycle/nsecond]	6.92

Brent-Kung

GPU Speed Of Light Throughput — All

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. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	0.81	Duration [usecond]	39.17
Memory Throughput [%]	0.20	Elapsed Cycles [cycle]	43,175
L1/TEX Cache Throughput [%]	5.34	SM Active Cycles [cycle]	1,038.03
L2 Cache Throughput [%]	0.20	SM Frequency [cycle/nsecond]	1.10
DRAM Throughput [%]	0.02	DRAM Frequency [cycle/nsecond]	6.95

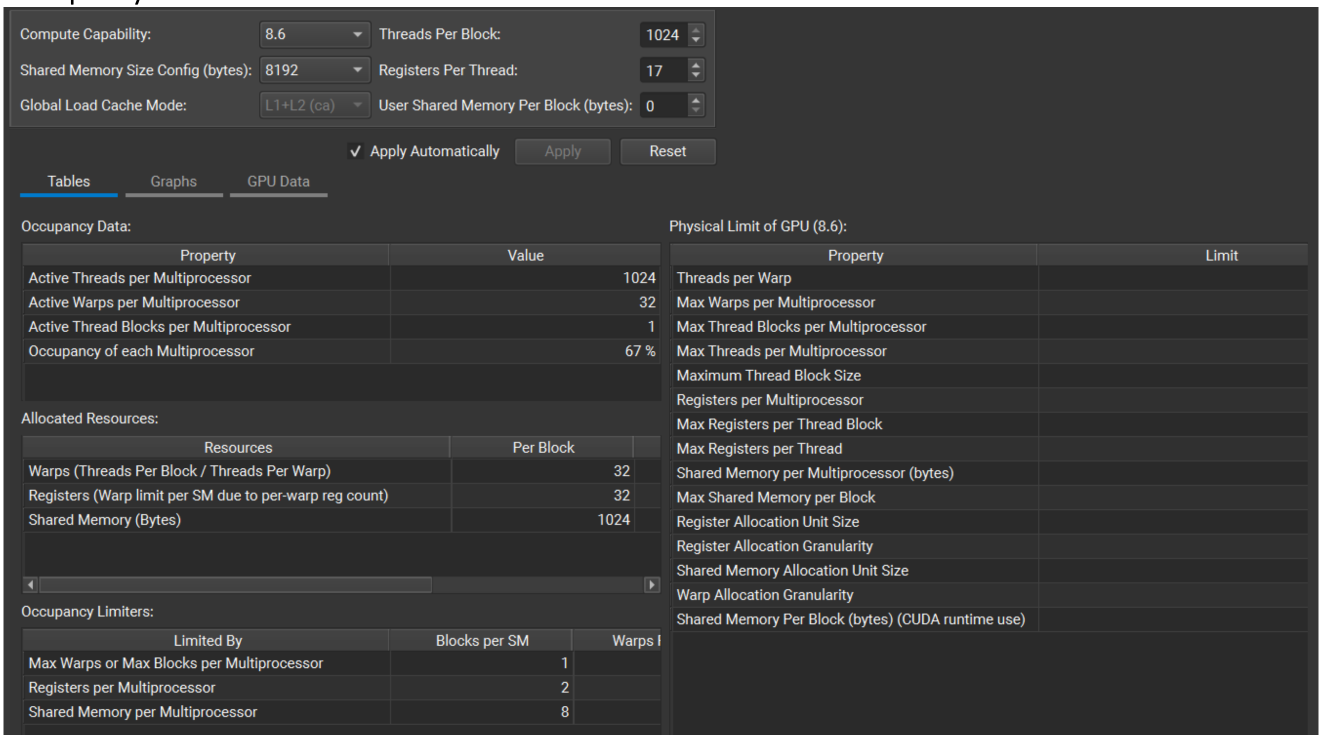
Running the code on Google Collab was also returning interesting results, at first we thought our optimizations were exponentially faster than the brute force. However, after investigation we found that the first operation run on google collab would always run slower than any subsequent operations. Therefore, we needed to run the brute force twice to get a baseline of the brute force that would be comparable to the optimizations. We can see that while the optimization does decrease the computation time the effectiveness between the Kogge-Stone method and the Brent-Kung method is similar in that they are both roughly 0.02 ms faster than the brute force method.

```
Array Size = 256

Data Array:
3  9  9  9  9  9  9  9  3  9  9  9   . . .   9  9  9  9  3  9  9  9  9  9  9  9
Obstacle Array:
1  0  0  0  0  0  0  0  1  0  0  0   . . .   0  0  0  0  1  0  0  0  0  0  0  0

Time (Brute-Force): 0.257664 ms
Prefix Sum Array:
  1  1  1  1  1  1  1  1  2  2  2  2  . . .  31 31 31 31 32 32 32 32 32 32 32 32
 32 obstacles detected

Time (Brute-Force): 0.044288 ms
Prefix Sum Array:
  1  1  1  1  1  1  1  1  2  2  2  2  . . .  31 31 31 31 32 32 32 32 32 32 32 32
 32 obstacles detected

Time (Kogge-Stone): 0.023936 ms
Prefix Sum Array:
  1  1  1  1  1  1  1  1  2  2  2  2  . . .  31 31 31 31 32 32 32 32 32 32 32 32
32 obstacles detected

Time (Brent-Kung): 0.025280 ms
Prefix Sum Array:
  1  1  1  1  1  1  1  1  2  2  2  2  . . .  31 31 31 31 32 32 32 32 32 32 32 32
32 obstacles detected
```

Nsight Results

Brute Force

Occupancy



Compute Capability: 8.6 | Threads Per Block: 1024
Shared Memory Size Config (bytes): 8192 | Registers Per Thread: 17
Global Load Cache Mode: L1+L2 (ca) | User Shared Memory Per Block (bytes): 0

✓ Apply Automatically | Apply | Reset

Tables | Graphs | GPU Data

Occupancy Data:

Property	Value
Active Threads per Multiprocessor	1024
Active Warps per Multiprocessor	32
Active Thread Blocks per Multiprocessor	1
Occupancy of each Multiprocessor	67 %

Allocated Resources:

Resources	Per Block
Warps (Threads Per Block / Threads Per Warp)	32
Registers (Warp limit per SM due to per-warp reg count)	32
Shared Memory (Bytes)	1024

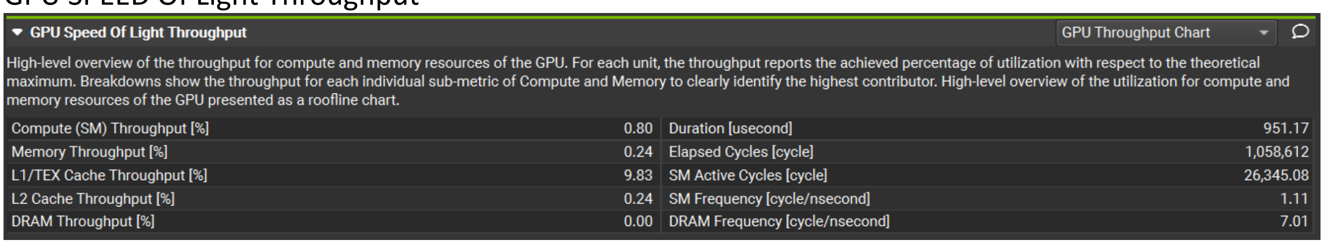
Occupancy Limiters:

Limited By	Blocks per SM	Warps I
Max Warps or Max Blocks per Multiprocessor	1	
Registers per Multiprocessor	2	
Shared Memory per Multiprocessor	8	

Physical Limit of GPU (8.6):

Property	Limit
Threads per Warp	
Max Warps per Multiprocessor	
Max Thread Blocks per Multiprocessor	
Max Threads per Multiprocessor	
Maximum Thread Block Size	
Registers per Multiprocessor	
Max Registers per Thread Block	
Max Registers per Thread	
Shared Memory per Multiprocessor (bytes)	
Max Shared Memory per Block	
Register Allocation Unit Size	
Register Allocation Granularity	
Shared Memory Allocation Unit Size	
Warp Allocation Granularity	
Shared Memory Per Block (bytes) (CUDA runtime use)	

GPU SPEED Of Light Throughput

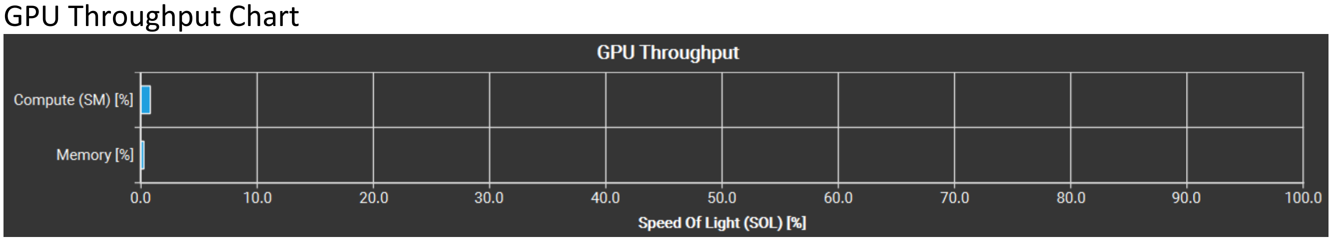


GPU Speed Of Light Throughput | GPU Throughput Chart

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. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	0.80	Duration [usecond]	951.17
Memory Throughput [%]	0.24	Elapsed Cycles [cycle]	1,058,612
L1/TEX Cache Throughput [%]	9.83	SM Active Cycles [cycle]	26,345.08
L2 Cache Throughput [%]	0.24	SM Frequency [cycle/nsecond]	1.11
DRAM Throughput [%]	0.00	DRAM Frequency [cycle/nsecond]	7.01

GPU Throughput Chart



GPU Throughput Breakdown

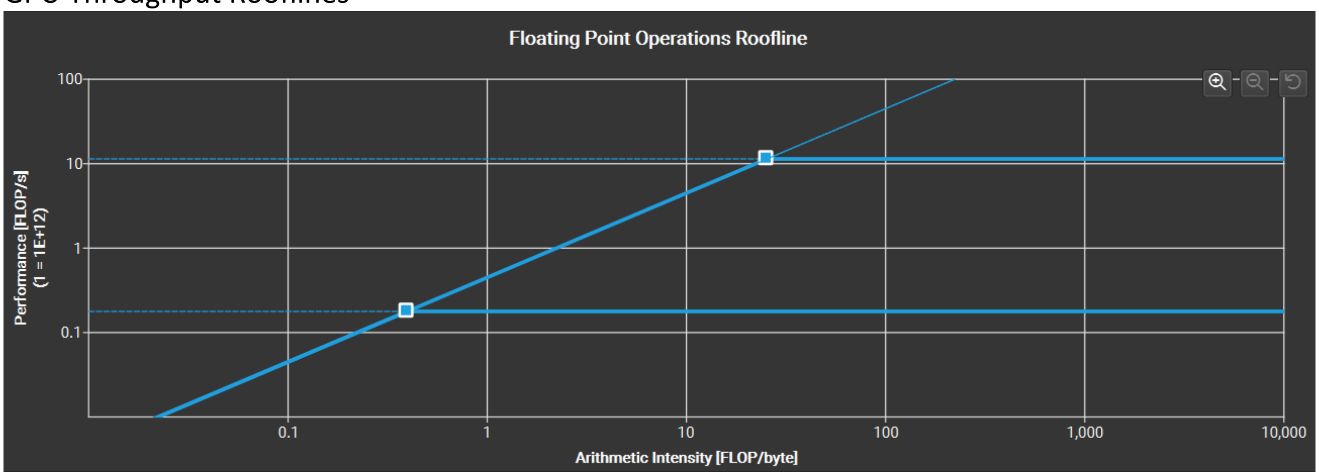
Compute Throughput Breakdown

SM: Pipe Alu Cycles Active [%]	0.80
SM: Issue Active [%]	0.53
SM: Inst Executed [%]	0.53
SM: Inst Executed Pipe Adu [%]	0.48
SM: Mio Pq Read Cycles Active [%]	0.28
SM: Mio Pq Write Cycles Active [%]	0.28
SM: Inst Executed Pipe Lsu [%]	0.24
SM: Mio Inst Issued [%]	0.24
SM: Mio2rf Writeback Active [%]	0.04
SM: Inst Executed Pipe Tex [%]	0.00
SM: Inst Executed Pipe Cbu Pred On Any [%]	0.00
IDC: Request Cycles Active [%]	0.00
SM: Pipe Fmaheavy Cycles Active [%]	0.00
SM: Pipe Fma Cycles Active [%]	0.00
SM: Inst Executed Pipe Ipa [%]	0
SM: Inst Executed Pipe Uniform [%]	0
SM: Inst Executed Pipe Xu [%]	0
SM: Pipe Fp64 Cycles Active [%]	0
SM: Pipe Tensor Cycles Active [%]	0

Memory Throughput Breakdown

L1: Lsuin Requests [%]	0.24
L2: Xbar2lts Cycles Active [%]	0.24
L2: T Sectors [%]	0.22
L1: M L1tex2xbar Req Cycles Active [%]	0.16
L1: Data Pipe Lsu Wavefronts [%]	0.13
L1: Lsu Writeback Active [%]	0.08
L2: T Tag Requests [%]	0.07
L2: D Sectors [%]	0.05
L1: Data Bank Reads [%]	0.04
L1: Data Bank Writes [%]	0.02
L2: Lts2xbar Cycles Active [%]	0.02
L1: Texin Sm2tex Req Cycles Active [%]	0.00
DRAM: Cycles Active [%]	0.00
DRAM: Dram Sectors [%]	0.00
L2: D Sectors Fill Device [%]	0.00
L1: M Xbar2l1tex Read Sectors [%]	0.00
L1: Data Pipe Tex Wavefronts [%]	0
L1: F Wavefronts [%]	0
L1: Tex Writeback Active [%]	0
L2: D Atomic Input Cycles Active [%]	0
L2: D Sectors Fill Sysmem [%]	0

GPU Throughput Rooflines



Kogge Stone

Occupancy

Compute Capability:	8.6	Threads Per Block:	1024
Shared Memory Size Config (bytes):	8192	Registers Per Thread:	16
Global Load Cache Mode:	L1+L2 (ca)	User Shared Memory Per Block (bytes):	4096

✓ Apply Automatically | Apply | Reset

Tables | Graphs | GPU Data

Occupancy Data:

Property	Value
Active Threads per Multiprocessor	1024
Active Warps per Multiprocessor	32
Active Thread Blocks per Multiprocessor	1
Occupancy of each Multiprocessor	67 %

Allocated Resources:

Resources	Per Block
Warps (Threads Per Block / Threads Per Warp)	32
Registers (Warp limit per SM due to per-warp reg count)	32
Shared Memory (Bytes)	5120

Occupancy Limiters:

Limited By	Blocks per SM	Warps l
Max Warps or Max Blocks per Multiprocessor	1	
Registers per Multiprocessor	4	
Shared Memory per Multiprocessor	1	

Physical Limit of GPU (8.6):

Property	Limit
Threads per Warp	
Max Warps per Multiprocessor	
Max Thread Blocks per Multiprocessor	
Max Threads per Multiprocessor	
Maximum Thread Block Size	
Registers per Multiprocessor	
Max Registers per Thread Block	
Max Registers per Thread	
Shared Memory per Multiprocessor (bytes)	
Max Shared Memory per Block	
Register Allocation Unit Size	
Register Allocation Granularity	
Shared Memory Allocation Unit Size	
Warp Allocation Granularity	
Shared Memory Per Block (bytes) (CUDA runtime use)	

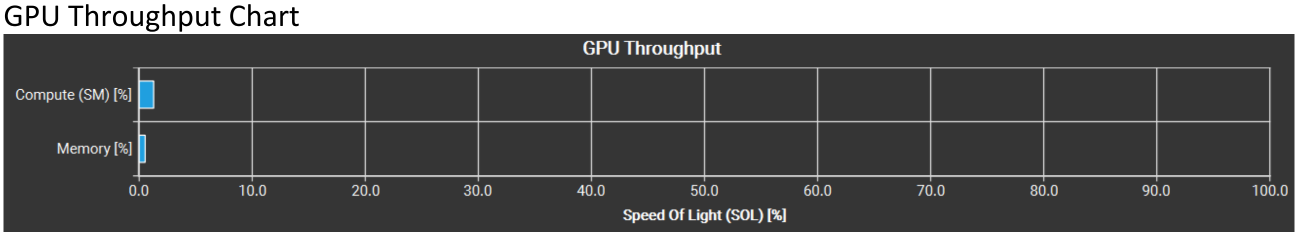
GPU SPEED Of Light Throughput

GPU Speed Of Light Throughput All

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. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	1.28	Duration [usecond]	24.86
Memory Throughput [%]	0.53	Elapsed Cycles [cycle]	27,336
L1/TEX Cache Throughput [%]	22.57	SM Active Cycles [cycle]	639
L2 Cache Throughput [%]	0.35	SM Frequency [cycle/nsecond]	1.10
DRAM Throughput [%]	0.04	DRAM Frequency [cycle/nsecond]	6.92

GPU Throughput Chart

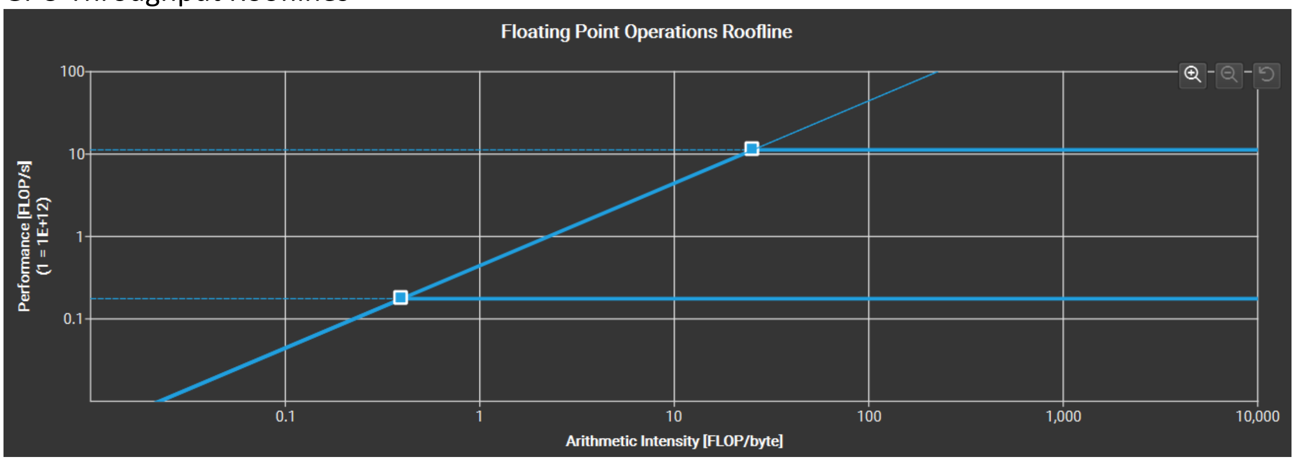


GPU Throughput Breakdown

Compute Throughput Breakdown	
SM: Pipe Alu Cycles Active [%]	1.28
SM: Issue Active [%]	0.88
SM: Inst Executed [%]	0.87
SM: Inst Executed Pipe Adu [%]	0.66
SM: Inst Executed Pipe Lsu [%]	0.53
SM: Mio Inst Issued [%]	0.40
SM: Mio Pq Read Cycles Active [%]	0.36
SM: Mio Pq Write Cycles Active [%]	0.36
SM: Mio2rf Writeback Active [%]	0.11
SM: Inst Executed Pipe Cbu Pred On Any [%]	0.10
IDC: Request Cycles Active [%]	0.02
SM: Pipe Fmaheavy Cycles Active [%]	0.02
SM: Pipe Fma Cycles Active [%]	0.01
SM: Inst Executed Pipe Tex [%]	0.01
SM: Inst Executed Pipe Ipa [%]	0
SM: Inst Executed Pipe Uniform [%]	0
SM: Inst Executed Pipe Xu [%]	0
SM: Pipe Fp64 Cycles Active [%]	0
SM: Pipe Tensor Cycles Active [%]	0

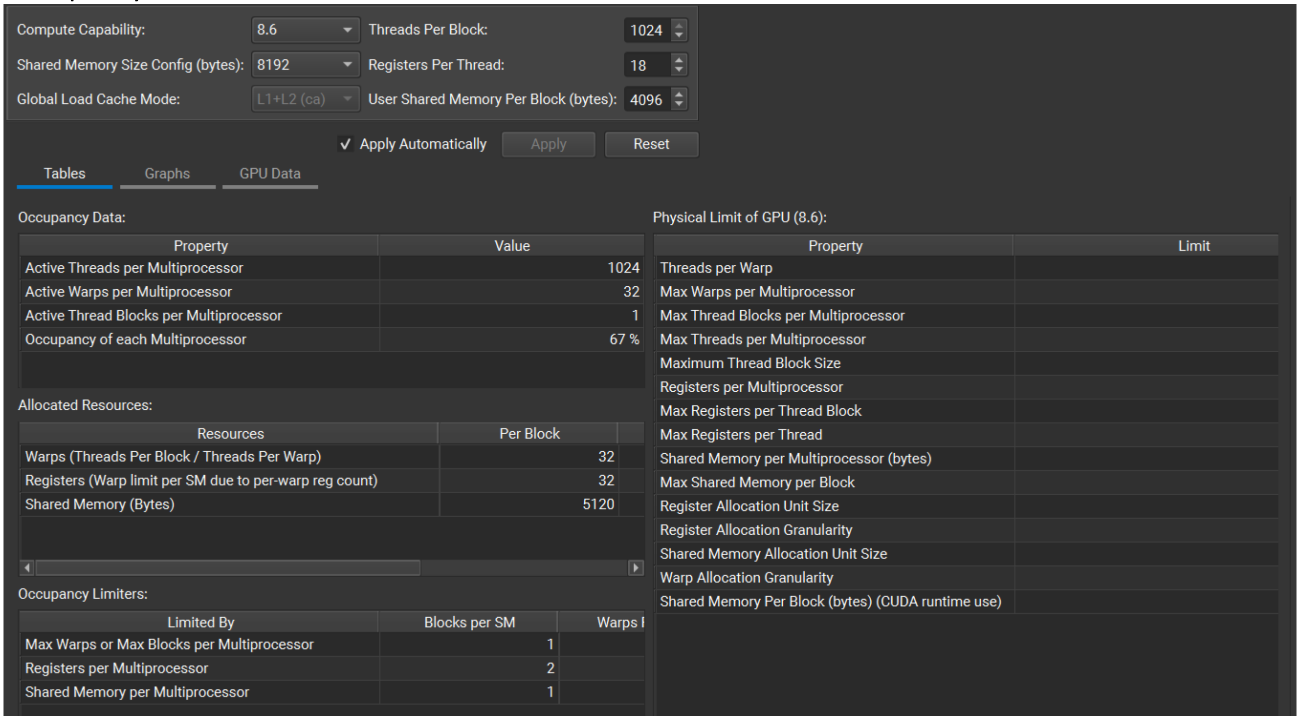
Memory Throughput Breakdown	
L1: Lsuin Requests [%]	0.53
L2: T Sectors [%]	0.35
L1: Data Pipe Lsu Wavefronts [%]	0.28
L1: Lsu Writeback Active [%]	0.20
L2: Xbar2lts Cycles Active [%]	0.19
L1: M L1tex2xbar Req Cycles Active [%]	0.13
L2: T Tag Requests [%]	0.07
L1: Data Bank Reads [%]	0.06
L2: D Sectors [%]	0.05
L2: Lts2xbar Cycles Active [%]	0.04
DRAM: Cycles Active [%]	0.04
L1: Texin Sm2tex Req Cycles Active [%]	0.04
DRAM: Dram Sectors [%]	0.03
L1: Data Bank Writes [%]	0.03
L2: D Sectors Fill Device [%]	0.02
L1: M Xbar2l1tex Read Sectors [%]	0.01
L1: Data Pipe Tex Wavefronts [%]	0
L1: F Wavefronts [%]	0
L1: Tex Writeback Active [%]	0
L2: D Atomic Input Cycles Active [%]	0
L2: D Sectors Fill Sysmem [%]	0

GPU Throughput Rooflines



Brent Kung

Occupancy



Compute Capability:	8.6	Threads Per Block:	1024
Shared Memory Size Config (bytes):	8192	Registers Per Thread:	18
Global Load Cache Mode:	L1+L2 (ca)	User Shared Memory Per Block (bytes):	4096

✓ Apply Automatically Apply Reset

Tables Graphs GPU Data

Occupancy Data:

Property	Value
Active Threads per Multiprocessor	1024
Active Warps per Multiprocessor	32
Active Thread Blocks per Multiprocessor	1
Occupancy of each Multiprocessor	67 %

Allocated Resources:

Resources	Per Block
Warps (Threads Per Block / Threads Per Warp)	32
Registers (Warp limit per SM due to per-warp reg count)	32
Shared Memory (Bytes)	5120

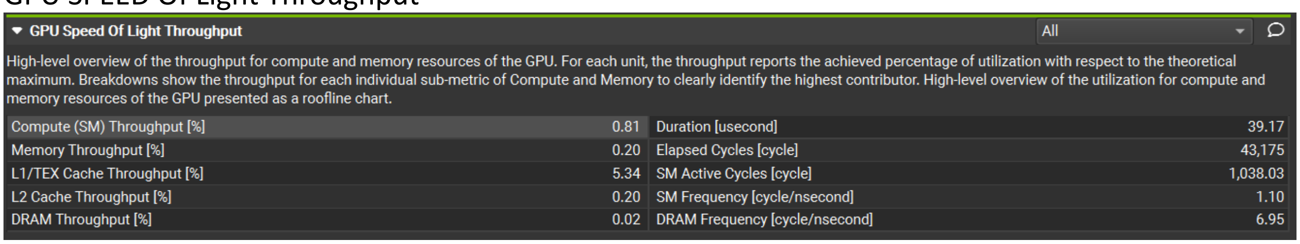
Occupancy Limiters:

Limited By	Blocks per SM	Warps I
Max Warps or Max Blocks per Multiprocessor	1	
Registers per Multiprocessor	2	
Shared Memory per Multiprocessor	1	

Physical Limit of GPU (8.6):

Property	Limit
Threads per Warp	
Max Warps per Multiprocessor	
Max Thread Blocks per Multiprocessor	
Max Threads per Multiprocessor	
Maximum Thread Block Size	
Registers per Multiprocessor	
Max Registers per Thread Block	
Max Registers per Thread	
Shared Memory per Multiprocessor (bytes)	
Max Shared Memory per Block	
Register Allocation Unit Size	
Register Allocation Granularity	
Shared Memory Allocation Unit Size	
Warp Allocation Granularity	
Shared Memory Per Block (bytes) (CUDA runtime use)	

GPU SPEED Of Light Throughput

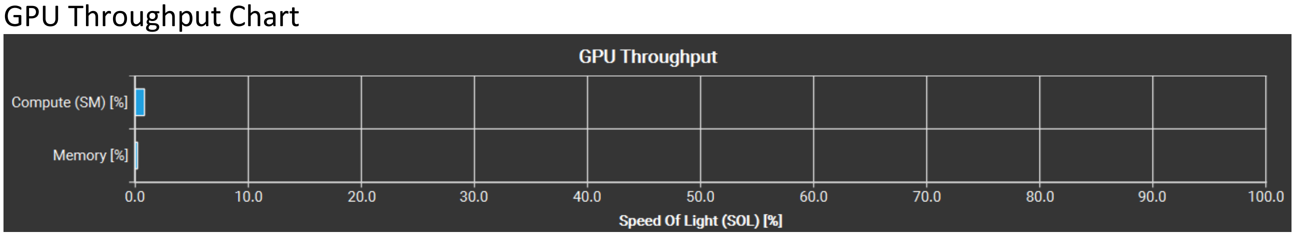


GPU Speed Of Light Throughput All

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. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	0.81	Duration [usecond]	39.17
Memory Throughput [%]	0.20	Elapsed Cycles [cycle]	43,175
L1/TEX Cache Throughput [%]	5.34	SM Active Cycles [cycle]	1,038.03
L2 Cache Throughput [%]	0.20	SM Frequency [cycle/nsecond]	1.10
DRAM Throughput [%]	0.02	DRAM Frequency [cycle/nsecond]	6.95

GPU Throughput Chart



GPU Throughput Breakdown

Compute Throughput Breakdown

SM: Pipe Alu Cycles Active [%]	0.81
SM: Issue Active [%]	0.59
SM: Inst Executed [%]	0.59
SM: Inst Executed Pipe Adu [%]	0.17
SM: Inst Executed Pipe Lsu [%]	0.13
SM: Inst Executed Pipe Cbu Pred On Any [%]	0.13
SM: Mio Inst Issued [%]	0.10
SM: Inst Executed Pipe Xu [%]	0.10
SM: Pipe Fmaheavy Cycles Active [%]	0.10
SM: Mio Pq Read Cycles Active [%]	0.05
SM: Mio Pq Write Cycles Active [%]	0.05
SM: Pipe Fma Cycles Active [%]	0.05
SM: Mio2rf Writeback Active [%]	0.03
IDC: Request Cycles Active [%]	0.01
SM: Inst Executed Pipe Tex [%]	0.00
SM: Inst Executed Pipe Ipa [%]	0
SM: Inst Executed Pipe Uniform [%]	0
SM: Pipe Fp64 Cycles Active [%]	0
SM: Pipe Tensor Cycles Active [%]	0

Memory Throughput Breakdown

L2: T Sectors [%]	0.20
L1: Lsuin Requests [%]	0.13
L1: Data Pipe Lsu Wavefronts [%]	0.13
L1: Lsu Writeback Active [%]	0.05
L2: Xbar2lts Cycles Active [%]	0.04
L2: Lts2xbar Cycles Active [%]	0.03
L2: T Tag Requests [%]	0.03
DRAM: Cycles Active [%]	0.02
L1: Texin Sm2tex Req Cycles Active [%]	0.02
DRAM: Dram Sectors [%]	0.02
L1: M L1tex2xbar Req Cycles Active [%]	0.01
L2: D Sectors Fill Device [%]	0.01
L2: D Sectors [%]	0.01
L1: M Xbar2l1tex Read Sectors [%]	0.01
L1: Data Bank Reads [%]	0.01
L1: Data Bank Writes [%]	0.00
L1: Data Pipe Tex Wavefronts [%]	0
L1: F Wavefronts [%]	0
L1: Tex Writeback Active [%]	0
L2: D Atomic Input Cycles Active [%]	0
L2: D Sectors Fill Sysmem [%]	0

GPU Throughput Rooflines