ECE 277

GPU Programming

Group 7 - Optical Flow Estimation

Anand Kumar
A59026700

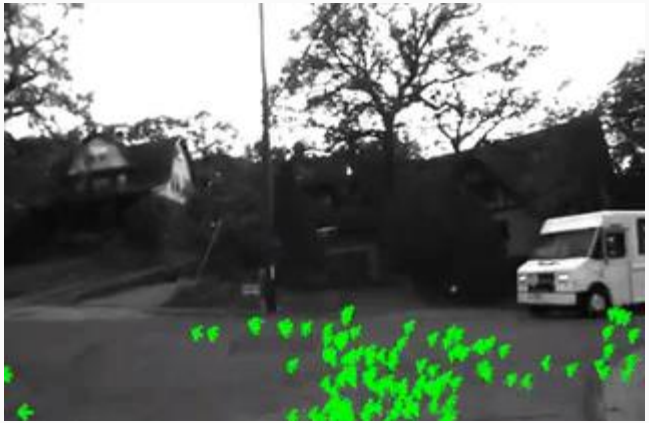
Nikhil Gandudi Suresh
A59022903

01 Optical Flow Introduction

A brief intro to optical flow

Optical Flow

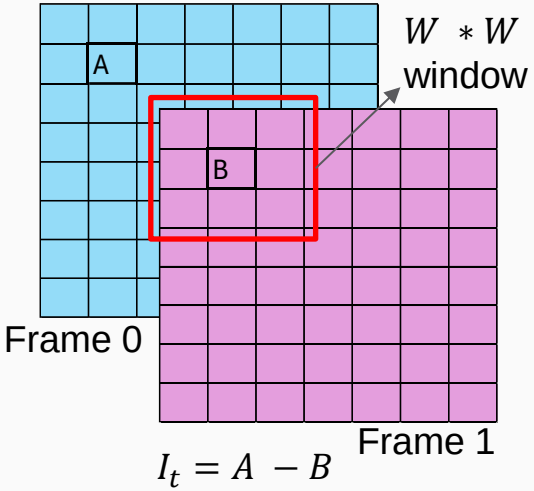
- Optical flow in computer vision describes the **pattern of apparent motion** of objects, surfaces, and edges in a visual scene caused by the relative movement.

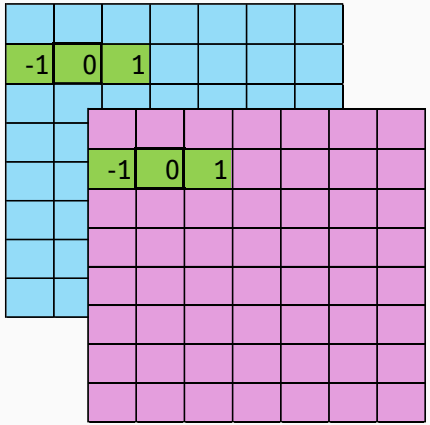


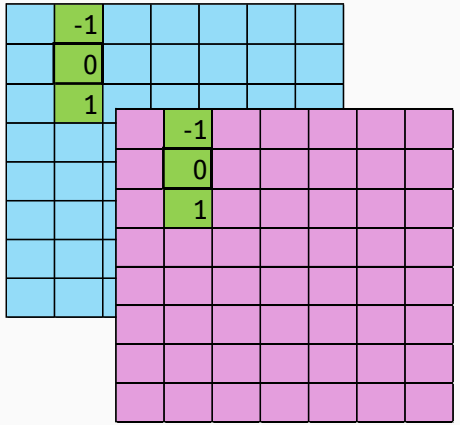
We are using **Lucas-Kanade Method** to estimate Optical Flow.

Lucas-Kanade Method

- Choose a small window $W * W$ around a pixel and compute three gradients - I_x, I_y, I_t







Computing Optical Flow

$$\begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} -\sum I_x I_t \\ -\sum I_y I_t \end{bmatrix},$$

$\sum$ represents sum over $W * W$ window

Closed Form solution of $(u, v)^T$

$$\begin{bmatrix} u \\ v \end{bmatrix} = \frac{1}{\Delta} \begin{bmatrix} \sum I_y^2 & -\sum I_x I_y \\ -\sum I_x I_y & \sum I_x^2 \end{bmatrix} \begin{bmatrix} -\sum I_x I_t \\ -\sum I_y I_t \end{bmatrix},$$

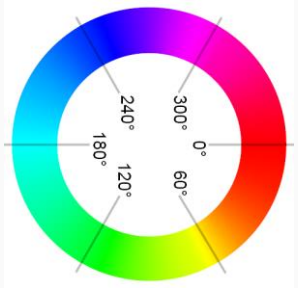
$$\Delta = \sum I_x^2 \sum I_y^2 - \left(\sum I_x I_y\right)^2$$

Displaying Optical Flow Vector

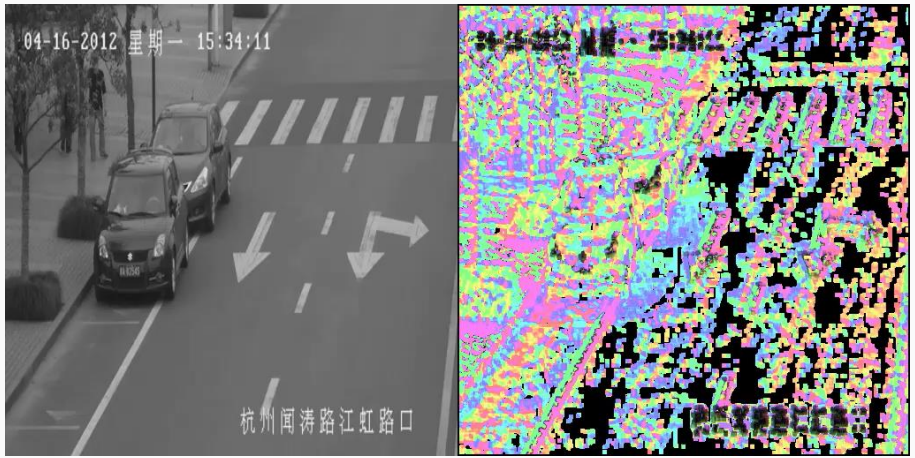
We calculate magnitude and phase to display optical flow vectors.

$$mag = \sqrt{(u^2+v^2)}$$

$$phase = \arctan\left(\frac{u}{v}\right) + \pi$$

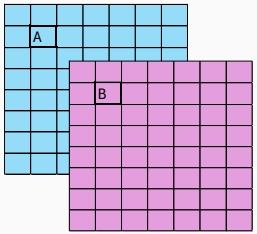


Color represents **phase** and
Intensity represents **magnitude**.

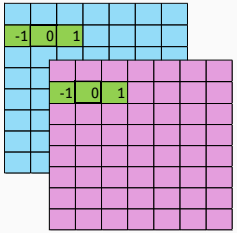


02 Kernels

Kernels



$I_t = A - B$



I_x

Kernel 1

To compute gradients, $I_x, I_y, \& I_t$

$$\begin{bmatrix} u \\ v \end{bmatrix} = \frac{1}{\Delta} \begin{bmatrix} \sum I_y^2 & -\sum I_x I_y \\ -\sum I_x I_y & \sum I_x^2 \end{bmatrix} \begin{bmatrix} -\sum I_x I_t \\ -\sum I_y I_t \end{bmatrix},$$
$$\Delta = \sum I_x^2 \sum I_y^2 - \left(\sum I_x I_y\right)^2$$

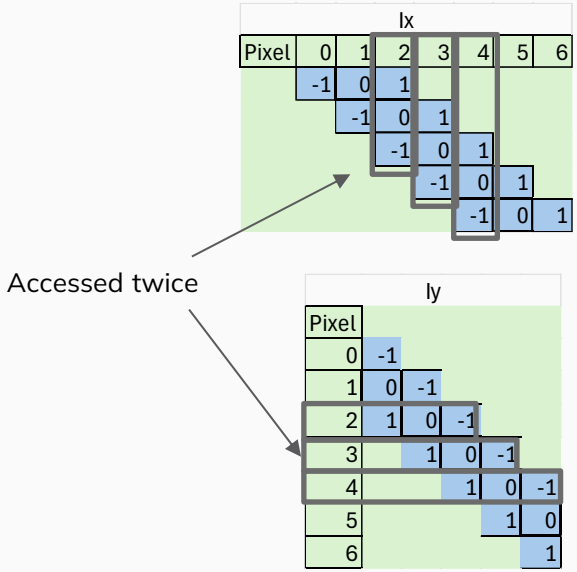
Kernel 2

To compute sum in $W * W$ region and compute optical flow vectors $u \ \& \ v$.

Kernel 1 – Computing Gradients

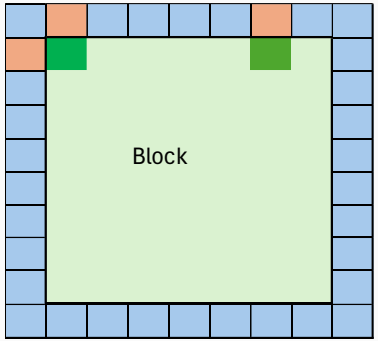
- To begin, a basic kernel is implemented to compute the gradients.
- It is **Coalesced Memory Access** but reading the same pixel twice.
- Using shared memory will improve performance.

```cpp
// kernel to get the Ix, Iy and It of two images
__global__ void cudaComputeGradients(float* Ix, float* Iy, float* It, const unsigned char* I1, const unsi
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    int idx = i + j * width;
    if (i > 0 && i < width - 1 && j > 0 && j < height - 1)
    {
        Ix[idx] = (I1[idx + 1] - I1[idx - 1] + I2[idx + 1] - I2[idx - 1]) / 4.0f;
        Iy[idx] = (I1[idx + width] - I1[idx - width] + I2[idx + width] - I2[idx - width]) / 4.0f;
        It[idx] = (I2[idx] - I1[idx]);
    }
}
```



Kernel 1 – Using Shared Memory

- To reduce access of global memory. We use shared memory to fetch all the data needed for gradient computation.
- Each thread copies one pixel.
- The threads on the edges copy one padded edge pixel also.
- The four corner threads copy three padded corner pixels as well.

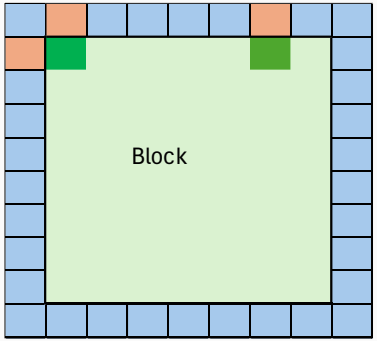



```
//kernel to get Ix, Iy and It of two images using shared memory
__global__ void cudaComputeGradients2(float* Ix, float* Iy, float
{

    __shared__ int I1_shared[BLOCK_SIZE + 2][BLOCK_SIZE + 2];
    __shared__ int I2_shared[BLOCK_SIZE + 2][BLOCK_SIZE + 2];

    int x = threadIdx.x + blockIdx.x * blockDim.x;
```

Shared Memory used per block (with Block Size of 32) –
= 34 x 34 x 4 x 2
= 9248 Bytes
= **9.248 kB**

Bank Conflicts Reduction

- **Implicit-padding** due to necessary padding for performing derivative for x, y & z.
- Only have to maintain **blockDim.x = 32.**



blockDim.x = 16

	Shared Memory				
	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	81,920	81,920	166,089	2.80	81,280
Shared Load Matrix	0	0			
Shared Store	64,384	64,384	80,768	0.34	16,384
Shared Store From Global Load	0	0	0	0	0
Shared Atomic	0	0	0	0	0
Other	-	-	362,220	7.12	0
Total	146,304	146,304	609,077	10.25	97,664

blockDim.x = 32

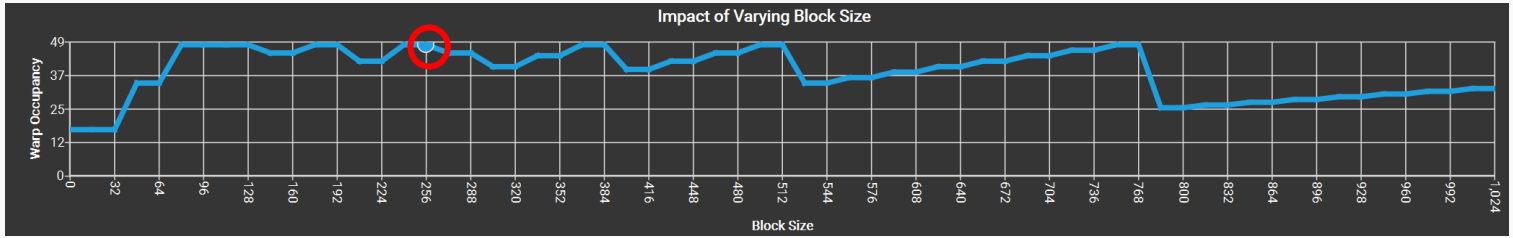
	Shared Memory				
	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	81,600	81,600	81,777	1.40	0
Shared Load Matrix	0	0			
Shared Store	51,136	51,136	51,136	0.22	0
Shared Store From Global Load	0	0	0	0	0
Shared Atomic	0	0	0	0	0
Other	-	-	330,802	6.32	0
Total	132,736	132,736	463,715	7.94	0

Optimal Block Size

- Utilized the warp occupancy graph to get the ideal block dimensions of :

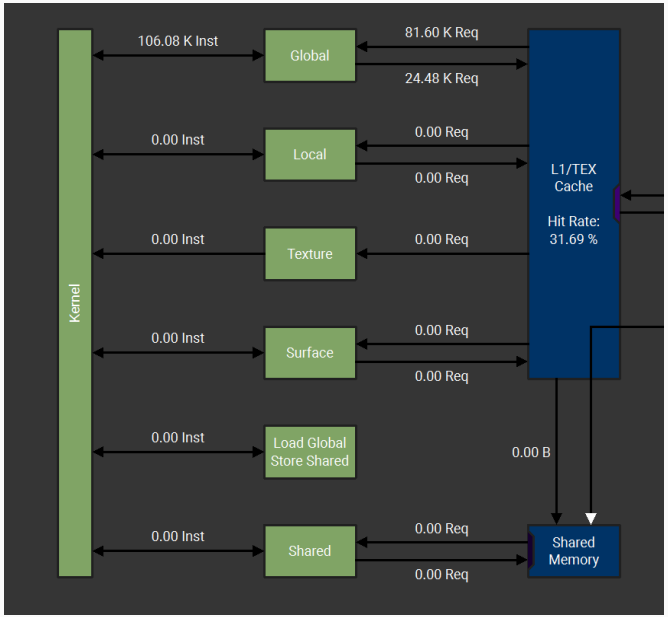
$$32 \times 8$$

$$Num\ Threads = 256$$

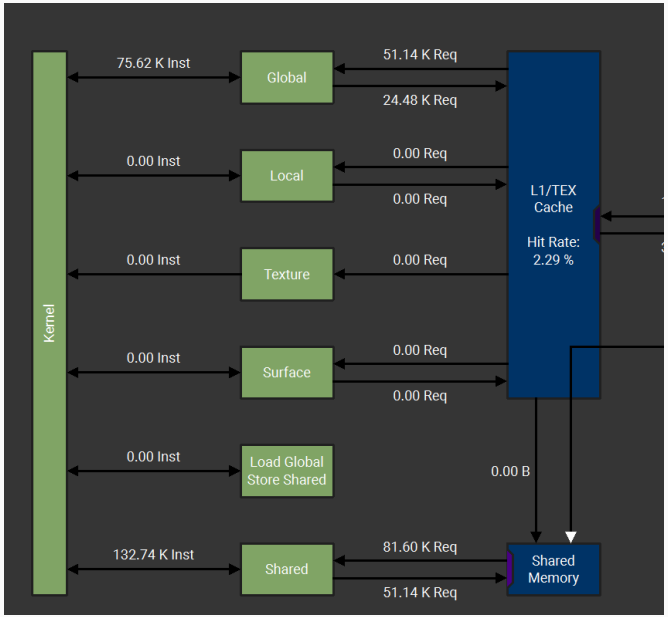


Kernel 1 - Memory Charts

Naïve GMEM

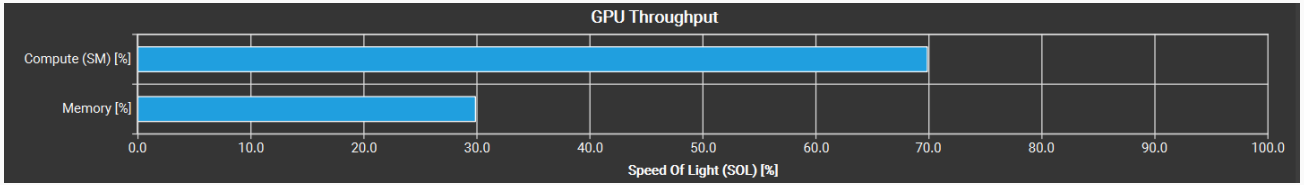


SMEM

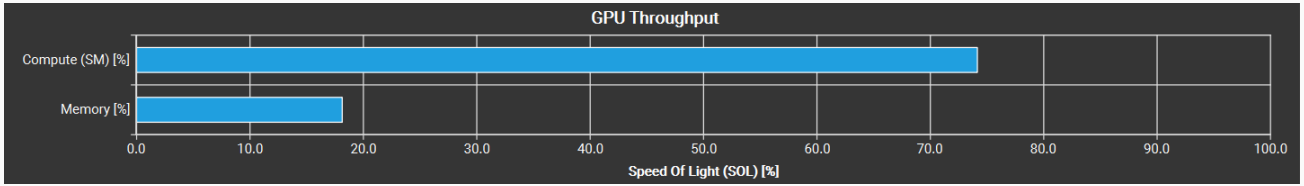


Analysis without L1 Cache

Naïve GMEM



SMEM



Kernel 1 – Loss from Shared Memory

- Performance loss using shared memory.
- Large L1 hit-rate for Naïve GMEM making it faster than SMEM.
- Gradients access a **single pixel only 2.5 times**, which is too low for efficient SMEM implementation.

	Naïve GMEM	SMEM
Duration [μs]	60.13	142.30
Elapsed Cycles [cycle]	102,668	243,202

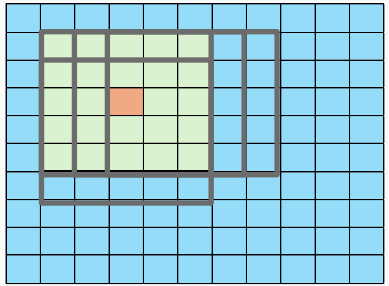
Naïve GMEM >> SMEM by ~2.5x

Kernel 2 – Computing Optical Flow

- Involves two steps –
  - Step1: Computing sum of $I_x^2, I_y^2, I_xI_y, I_xI_t, I_yI_t$ in a $W * W$ window.
  - Step2: Computing $(u, v)^T$ using the above computed sums.

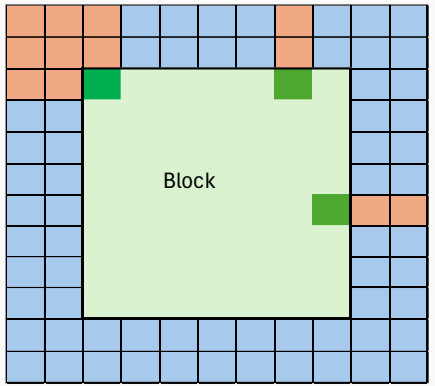
$$\begin{bmatrix} u \\ v \end{bmatrix} = \frac{1}{\Delta}\begin{bmatrix} \sum I_y^2 & -\sum I_xI_y \\ -\sum I_xI_y & \sum I_x^2 \end{bmatrix}\begin{bmatrix} -\sum I_xI_t \\ -\sum I_yI_t \end{bmatrix}, \Delta = \sum I_x^2 \sum I_y^2 - \left(\sum I_xI_y\right)^2$$

- At a given pixel, Step 2 is a closed form solution and can not be optimized further.
- Step1 has multiple threads accessing the same value from the global memory multiple times.



Kernel 2 – Using Shared Memory

- Similar to Kernel 1, we use shared memory to get all the computed $I_x, I_y, and\ I_t$.
- Corner Threads/Pixels fill in all the corner values as well.
- Edge Threads/Pixels fill in all the edge values as well.

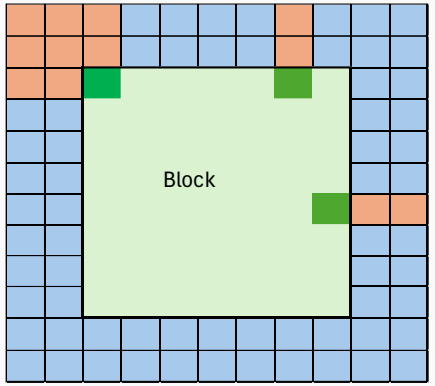



```
__global__ void cudaComputeOpticalFlow2(const float* Ix, const float* Iy, const float* It, in
{
    __shared__ float Ix_shared[BLOCK_SIZE + WIN_SIZE - 1][BLOCK_SIZE + WIN_SIZE - 1];
    __shared__ float Iy_shared[BLOCK_SIZE + WIN_SIZE - 1][BLOCK_SIZE + WIN_SIZE - 1];
    __shared__ float It_shared[BLOCK_SIZE + WIN_SIZE - 1][BLOCK_SIZE + WIN_SIZE - 1];
```

Shared Memory used per block (with Block Size of 32) –
= 36 x 36 x 4 x 3
= 15552 Bytes
= **15.552 kB**

Thread Divergence

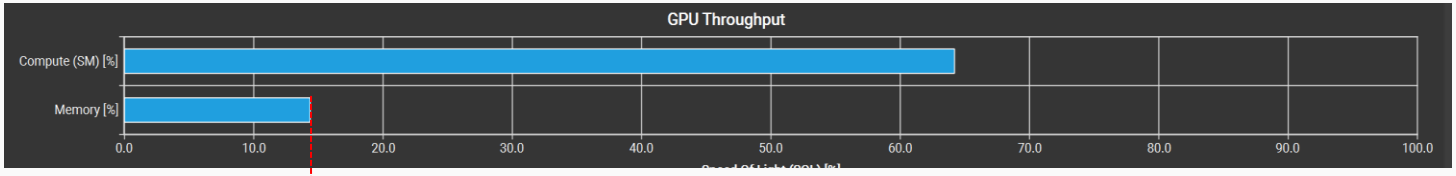
- Each thread fetches a different number of pixels as shown.
- All the threads in a block have to wait for corner thread.
- Memory transactions for SMEM will be higher due to padding.



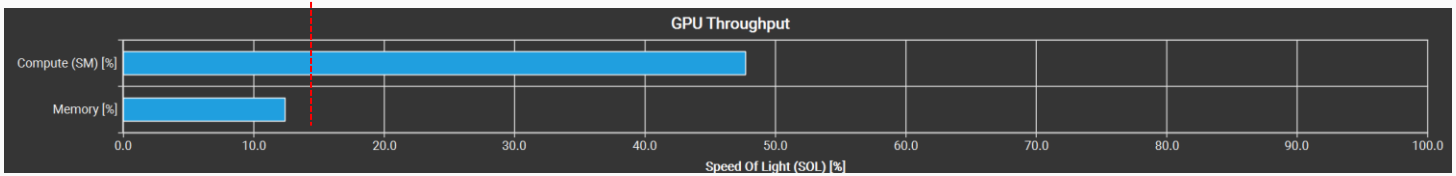
	Naïve GMEM	SMEM
Avg. Active Threads Per Warp	31.49	29.87

Effect of Thread Divergence

Naïve GMEM



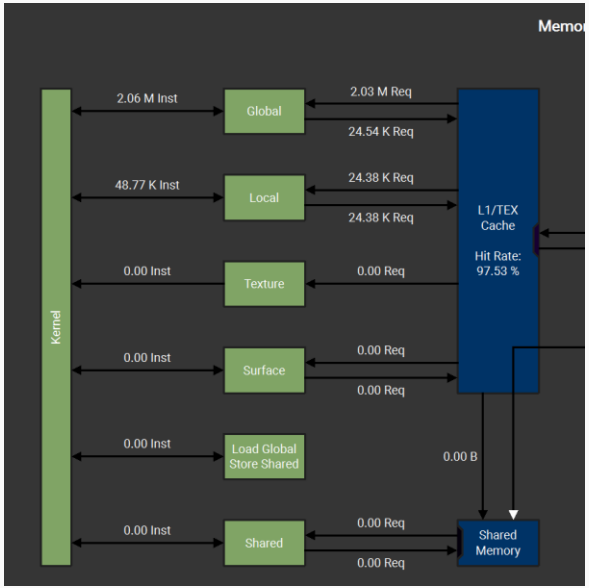
SMEM



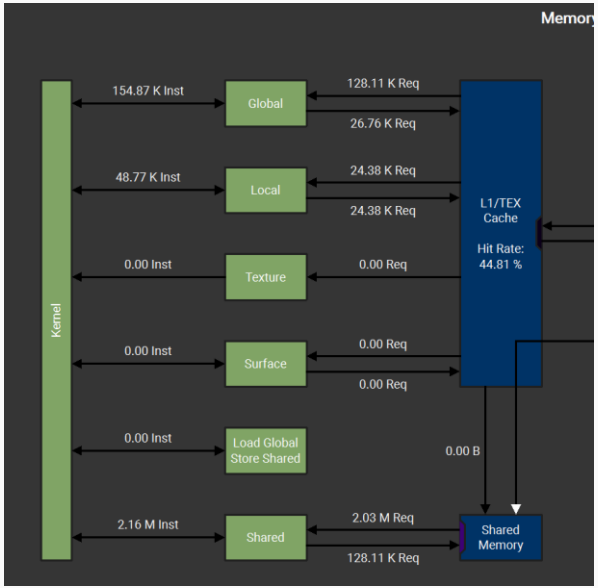
Effect on Memory Transactions

- Memory transactions for SMEM will be higher due to padding.

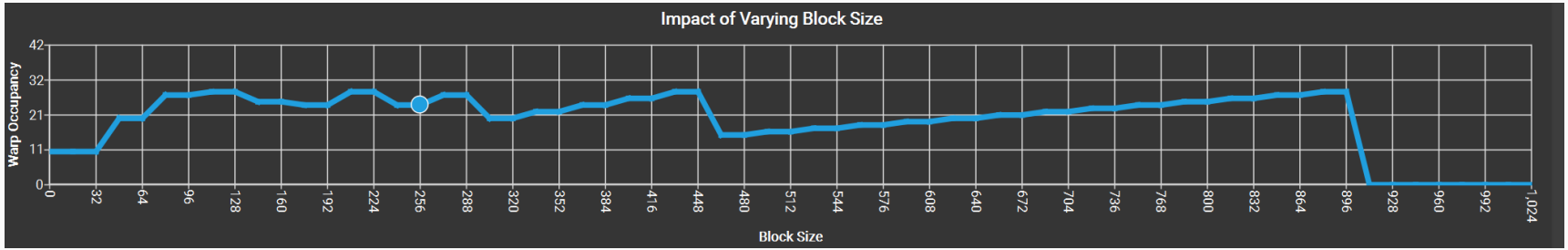
Naïve GMEM



SMEM



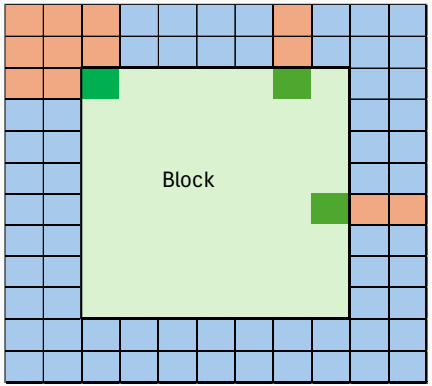
Optimal Block Size



- Shared memory limits the block size.
- 256 block size is the optimal which balances between the memory limitation per SM and occupancy.

Bank Conflicts Reduction

- **Implicit-padding** due to necessary.
- Only have to maintain **blockDim.x = 32.**



blockDim.x = 16

	Shared Memory				
	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	2,032,000	2,032,000	4,122,015	4.27	2,032,000
Shared Load Matrix	0	0			
Shared Store	285,552	285,552	334,512	0.09	48,960
Shared Store From Global Load	0	0	0	0	0
Shared Atomic	0	0	0	0	0
Other	-	-	4,781,051	5.21	0
Total	2,317,552	2,317,552	9,237,578	9.57	2,080,960

blockDim.x = 32

	Shared Memory				
	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	2,032,000	2,032,000	2,032,078	1.85	0
Shared Load Matrix	0	0			
Shared Store	128,112	128,112	128,112	0.03	0
Shared Store From Global Load	0	0	0	0	0
Shared Atomic	0	0	0	0	0
Other	-	-	4,450,740	4.14	0
Total	2,160,112	2,160,112	6,610,930	6.02	0

Kernel 2 – Loss from Shared Memory

- Similar to Kernel 1
- Large L1 hit-rate for Naïve GMEM making it faster than SMEM.
- Overhead of storing data in SMEM is bigger than the gain it provides, esp. with padding.

	Naïve GMEM	SMEM
Duration [ms]	0.72	2.68
Elapsed Cycles [cycle]	1,235,804	4,575,505

Naïve GMEM >> SMEM by ~4x

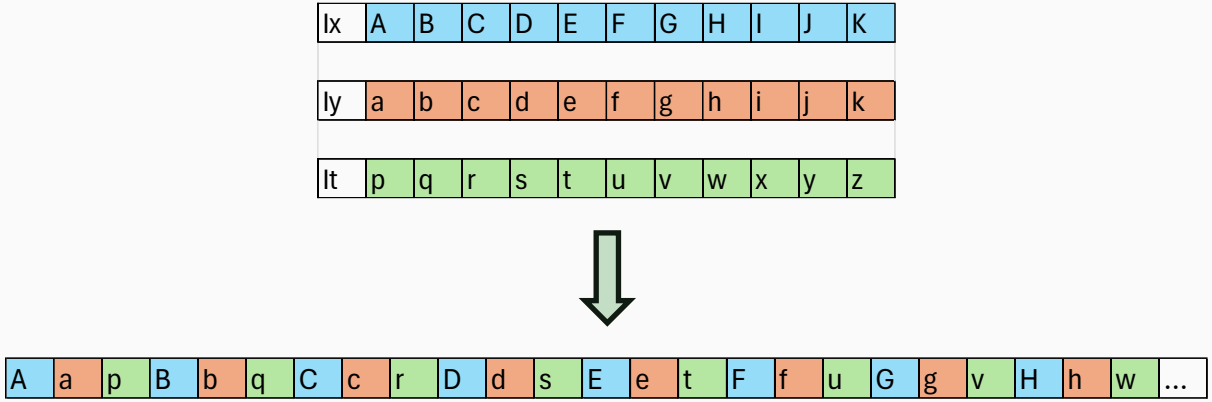
03 Further Optimizations

Optim 1. Improving Coalesced Access

- Consolidate instructions to fetch $I_x, I_y, and\ I_t$, as their positions are fixed, reducing the overall instruction count.
- Due to padding, there is uncoalesced memory access.

From the report, we observe access to **16%** of 9.74M of L2 sectors is **uncoalesced.**

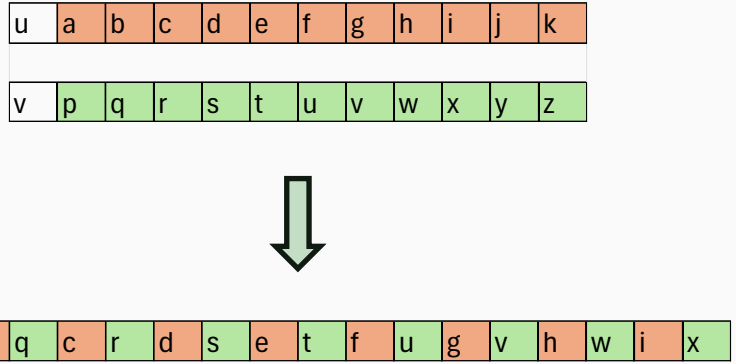
Optim. 1 - Array of Structures (Read)




```
ğľộắƫ Íẏ xîđƫḥ   ḥêîĝḥƫ
ğľộắƫ Íỳ xîđƫḥ   ḥêîĝḥƫ
ğľộắƫ Íƫ xîđƫḥ   ḥêîĝḥƫ

Şƫsụçƫ   ắľîĝŋ   ,`
 ğľộắƫ Íẏ
 ğľộắƫ Íỳ
 ğľộắƫ Íƫ
  Ğsắđ
Ğsắđ ĝsắđș xîđƫḥ   ḥêîĝḥƫ
```

- Improves coalesced read access to memory.

Optim. 1 - Array of Structures (Write)




```
ğľộắƫ ụ xîđƫḥ   ḥêîĝḥƫ
ğľộắƫ ŵ xîđƫḥ   ḥêîĝḥƫ

Şƫsụçƫ   ắľîĝŋ   ¨
 ğľộắƫ ụ
 ğľộắƫ ŵ
  Gľộx
Gľộx ụŵ xîđƫḥ   ḥêîĝḥƫ
```

- Improves coalesced write access to memory.

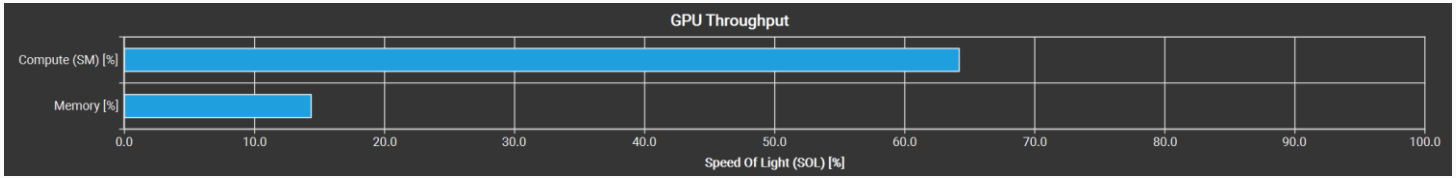
Results for Optim. 1 - AoS

	Naïve GMEM	AoS GMEM
Duration [μs]	722.75	441.92
Elapsed Cycles [cycle]	1,235,804	755,537
Uncoalesced Sectors (%)	16%	5%

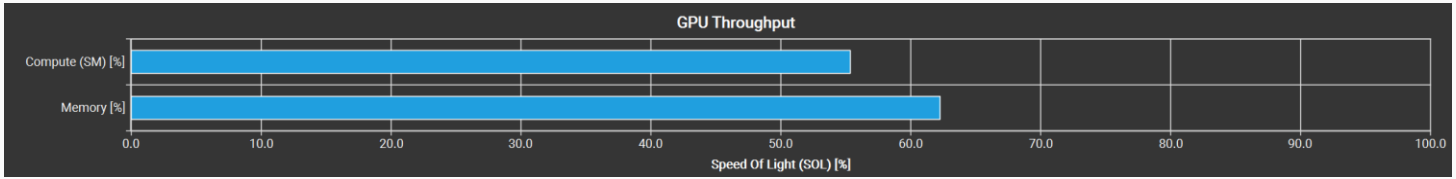
AoS GMEM >> Naïve GMEM by ~2x

Gain from AoS

Naïve GMEM

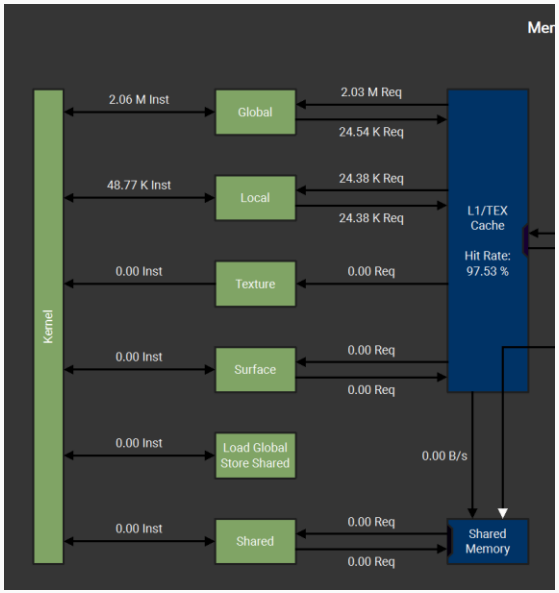


AoS GMEM

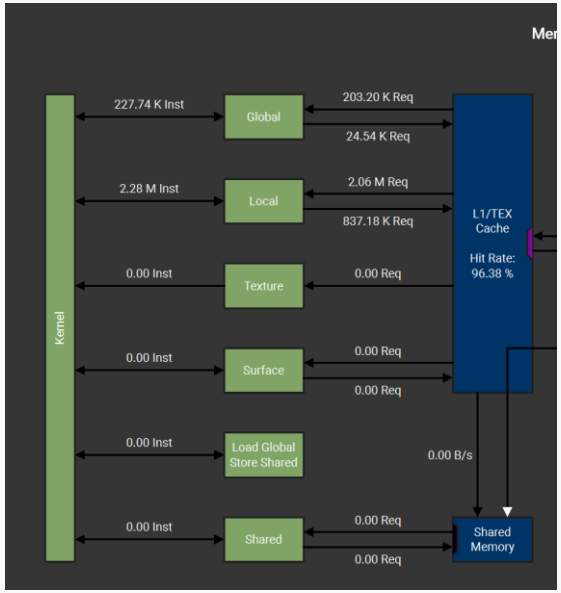


Gain from Optim. 1 – AoS (Logical)

Naïve GMEM

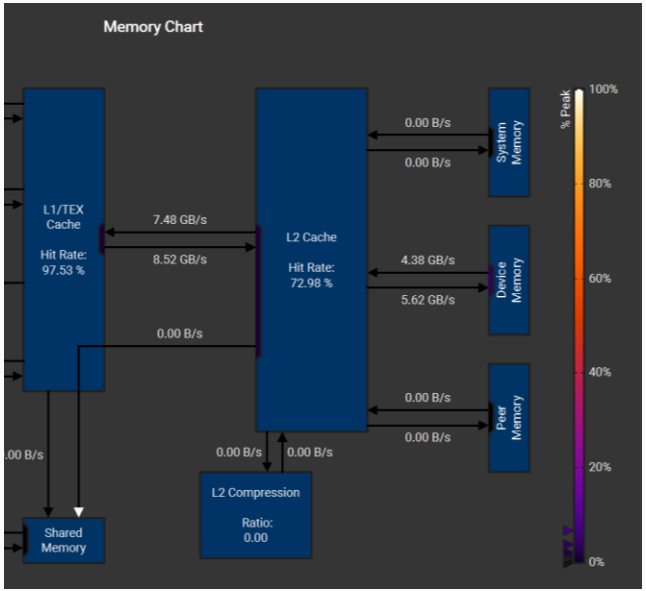


AoS GMEM

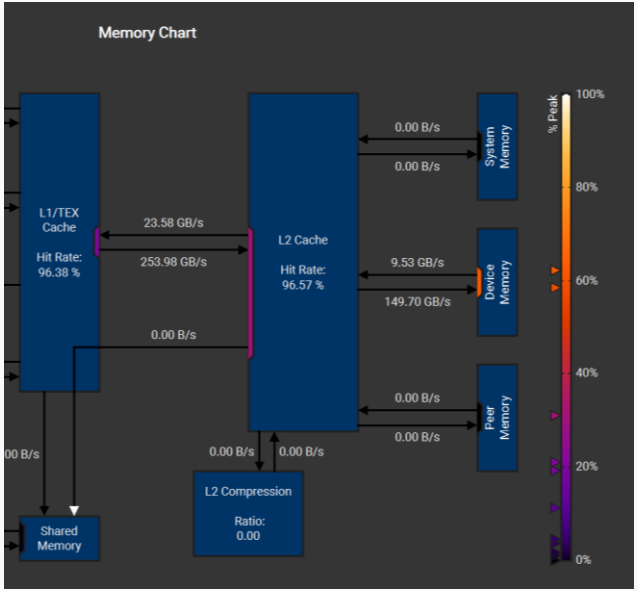


Gain from Optim. 1 – AoS (Physical)

Naïve GMEM



AoS GMEM

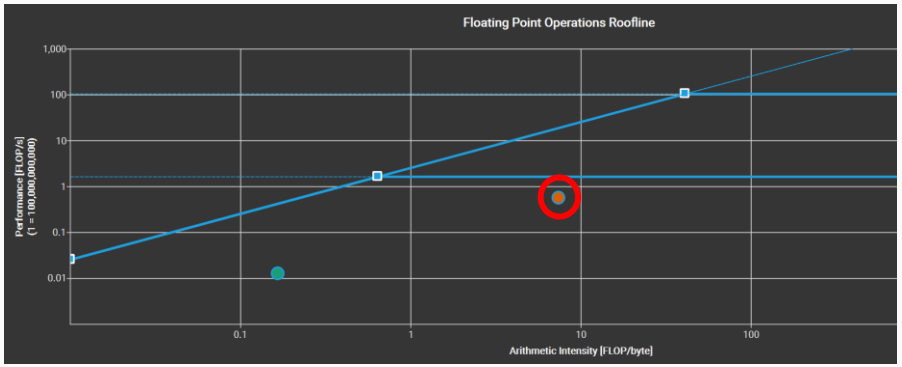


Optim. 2 – Utilizing DMA

- No significant improvement by allocating the host values using $cudaHostAlloc$.

Optim. 3 – Double Precision (Double)

- Since Kernel 2 is more resource heavy, we analyze the throughput roofline for it.

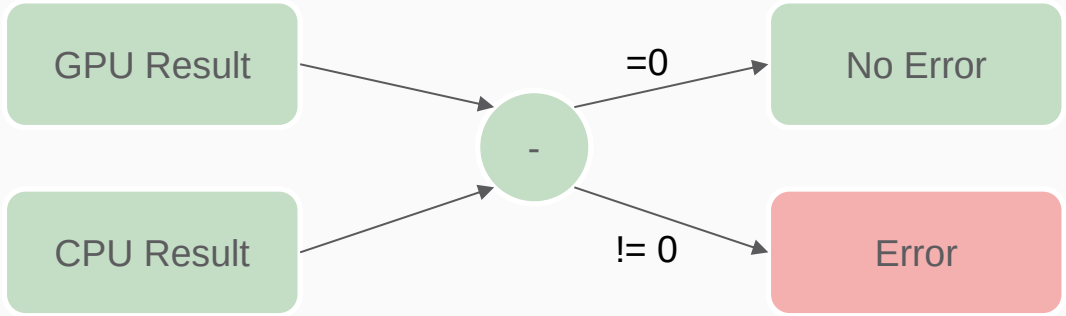


From the graph, our bottleneck is primarily compute based due to loops which potentially can be removed using tensor cores.

04 Sanity Check

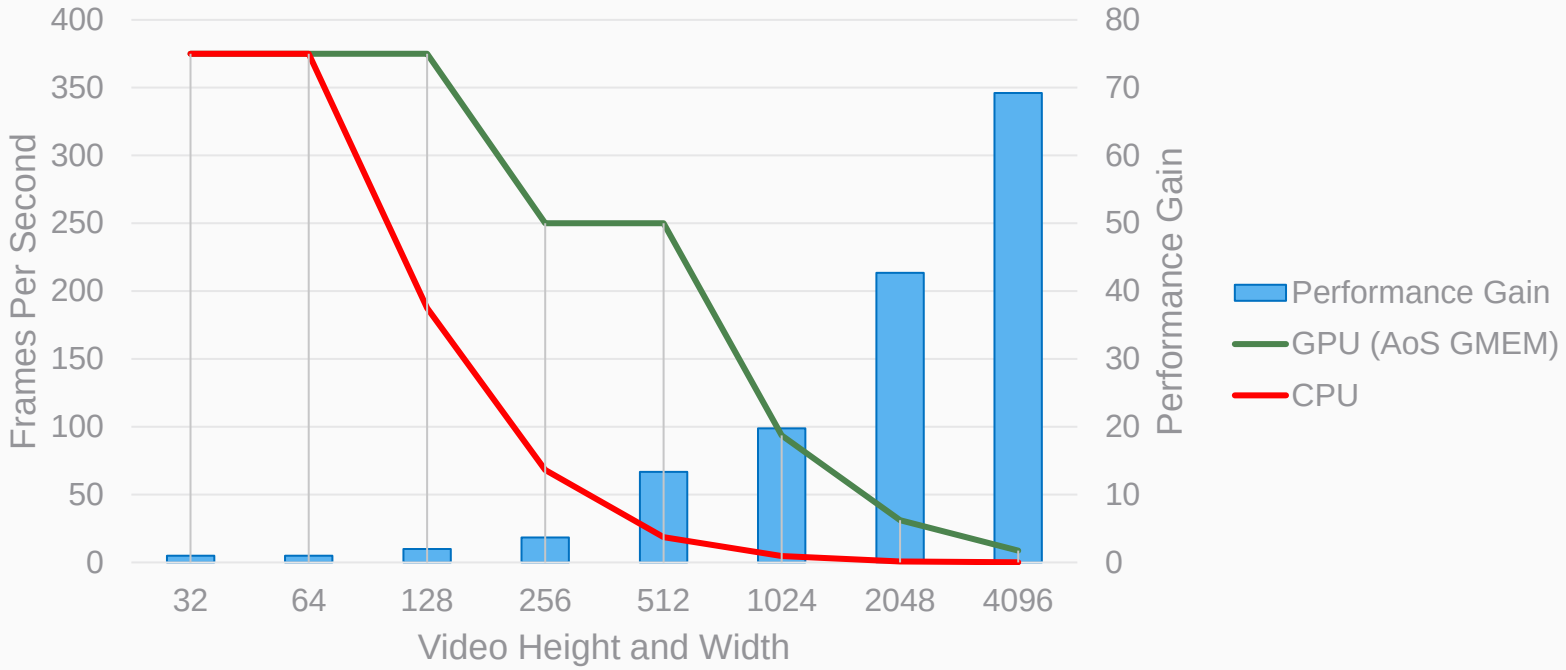
Sanity Check

- To ensure accurate GPU results and performance comparison, we build CPU implementation.
- Check results for each frame with CPU using DEBUG and CPU flags.
- This check is disabled for performance testing.



05 Results

CPU vs GPU Comparison

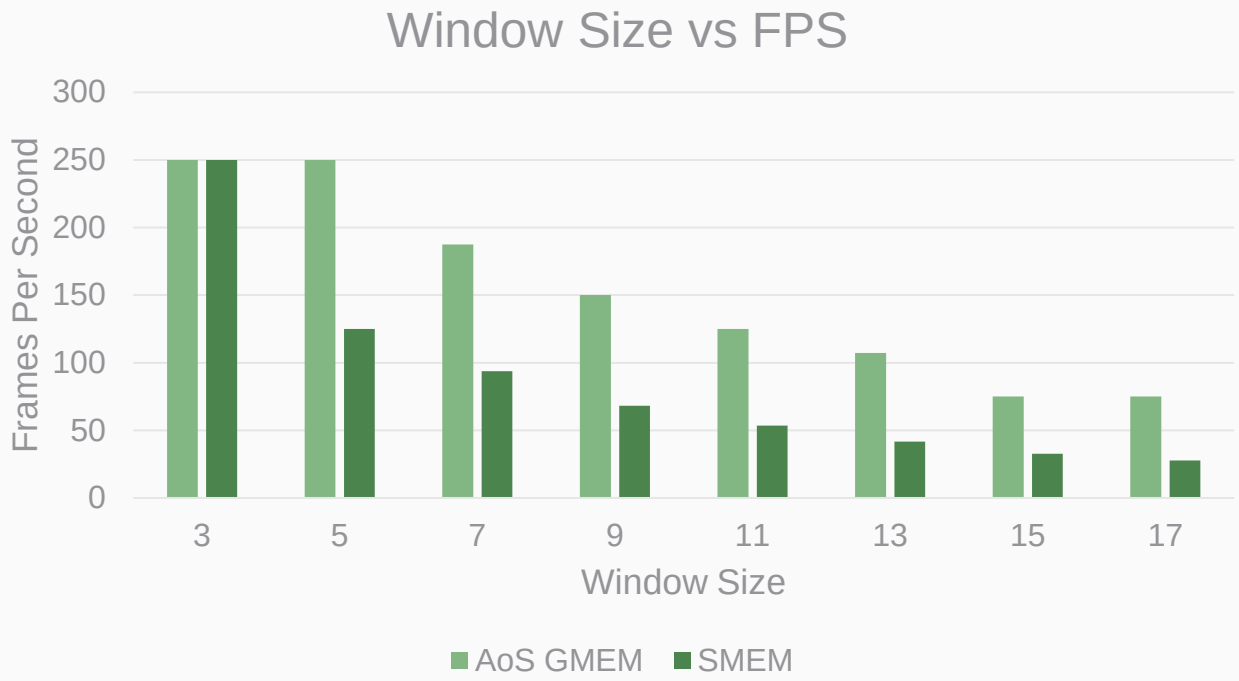


GPU Variations

- CPU, GPU(Naïve GMEM), GPU (SMEM), GPU (Naïve GMEM) + DMA & GPU (AoS GMEM)

Variation @ 512	FPS
CPU	18.75
GPU (Naïve GMEM)	187.5
GPU (SMEM)	125
GPU (AoS GMEM)	250

Effect of Changing Window Sizes



Visualization of Results with FPS

Input GPU CPU

*Rendering in CPU

Thank You