Parallel Ray Tracer

Group 8:

311553060 周睦鈞
109550156 曾偉杰
109550059 黃彥傑

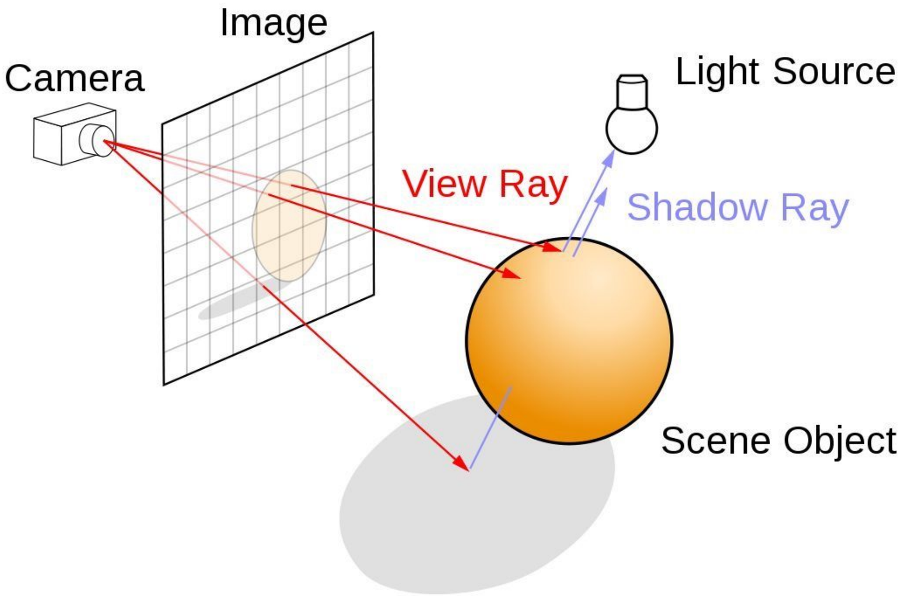
Outline

- Introduction
- Problem Statement
- Proposed Solution
- Evaluation
- Conclusion

Introduction

Introduction

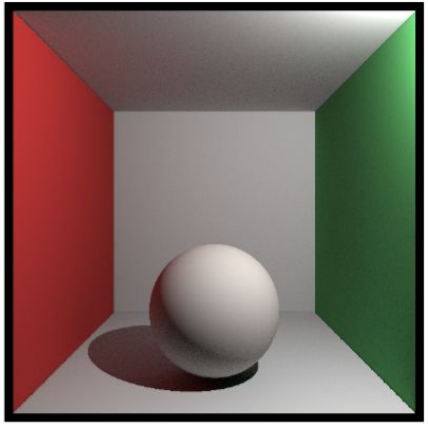
- Ray tracing is a rendering method that simulates the behavior of light to create realistic images



Problem Statement

Problem Statement

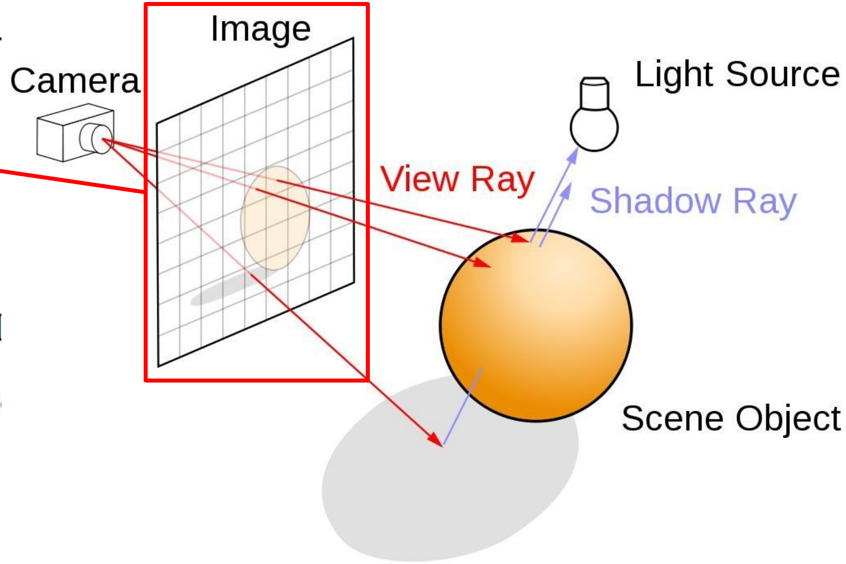
- Serial version of the Ray Tracer requires rendering one pixel before proceeding to the next, resulting in significant time wastage.
- Utilize parallelism methods to minimize computational costs.



Ray Tracing Renderer

Algorithm 1: The Conceptual Workflow of Ray Tracing

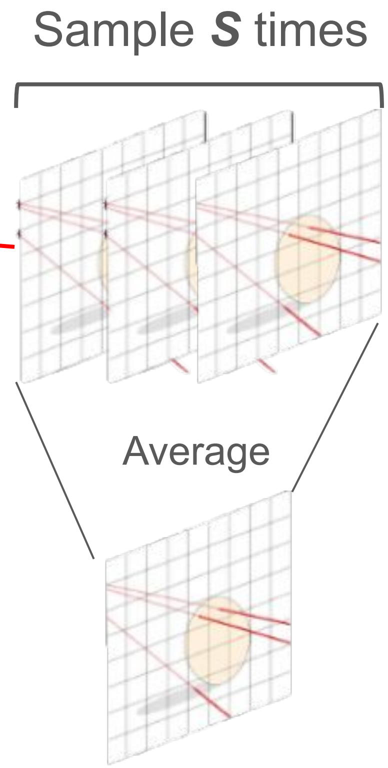
```
for p = 1 to P do
    Image[p] = [0 0 0]
    for s = 1 to S do
        Ray = GenerateRay(p)
        for b = 1 to B do
            Initialize IntersectionResult
            for g = 1 to G do
                IntersectionResult = IntersectionTesting(Ray, GeometricObjects[
            if IntersectionResult.Hit then
                Image[p] = Image[p] + CalculateColor(LightSource, Ray, Interse
                Ray = GenerateNextRay(Ray, IntersectionResult)
            else
                break
    Image[p] = Image[p] / S
```



Ray Tracing Renderer

Algorithm 1: The Conceptual Workflow of Ray Tracing

```
for p = 1 to P do
    Image[p] = [0 0 0]
    for s = 1 to S do
        Ray = GenerateRay(p)
        for b = 1 to B do
            Initialize IntersectionResult
            for g = 1 to G do
                IntersectionResult = IntersectionTesting(Ray, GeometricObjects[
            if IntersectionResult.Hit then
                Image[p] = Image[p] + CalculateColor(LightSource, Ray, Interse
                Ray = GenerateNextRay(Ray, IntersectionResult)
            else
                break
    Image[p] = Image[p] / S
```

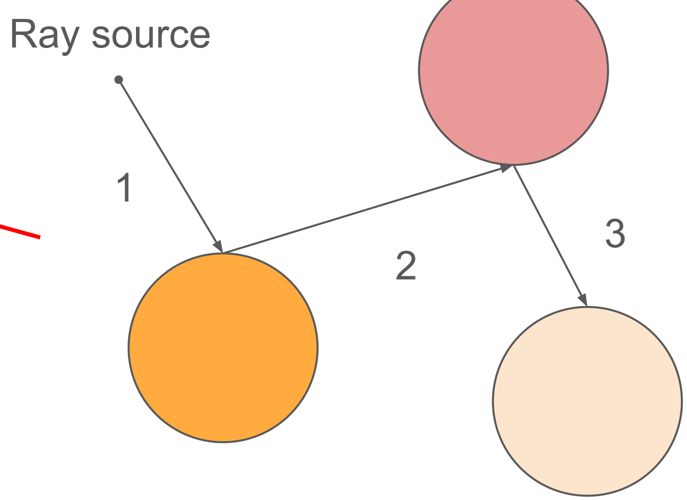


Ray Tracing Renderer

Algorithm 1: The Conceptual Workflow of Ray Tracing

```
for p = 1 to P do
    Image[p] = [0 0 0]
    for s = 1 to S do
        Ray = GenerateRay(p)
        for b = 1 to B do
            Initialize IntersectionResult
            for g = 1 to G do
                IntersectionResult = IntersectionTesting(Ray, GeometricObjects[
            if IntersectionResult.Hit then
                Image[p] = Image[p] + CalculateColor(LightSource, Ray, Interse
                Ray = GenerateNextRay(Ray, IntersectionResult)
            else
                break
    Image[p] = Image[p] / S
```

Bounce ***B*** times per ray

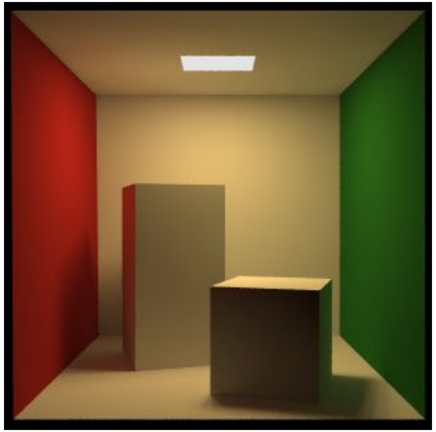


Ray Tracing Renderer

Algorithm 1: The Conceptual Workflow of Ray Tracing

```
for p = 1 to P do
    Image[p] = [0 0 0]
    for s = 1 to S do
        Ray = GenerateRay(p)
        for b = 1 to B do
            Initialize IntersectionResult
            for g = 1 to G do
                IntersectionResult = IntersectionTesting(Ray, GeometricObjects[
            if IntersectionResult.Hit then
                Image[p] = Image[p] + CalculateColor(LightSource, Ray, Interse
                Ray = GenerateNextRay(Ray, IntersectionResult)
            else
                break
    Image[p] = Image[p] / S
```

Number of Objects ***G***



Ray Tracing Renderer

Algorithm 1: The Conceptual Workflow of Ray Tracing

```
for p = 1 to P do
    Image[p] = [0 0 0]
    for s = 1 to S do
        Ray = GenerateRay(p)
        for b = 1 to B do
            Initialize IntersectionResult
            for g = 1 to G do
                IntersectionResult = IntersectionTesting(Ray, GeometricObjects[
            if IntersectionResult.Hit then
                Image[p] = Image[p] + CalculateColor(LightSource, Ray, Interse
                Ray = GenerateNextRay(Ray, IntersectionResult)
            else
                break
    Image[p] = Image[p] / S
```

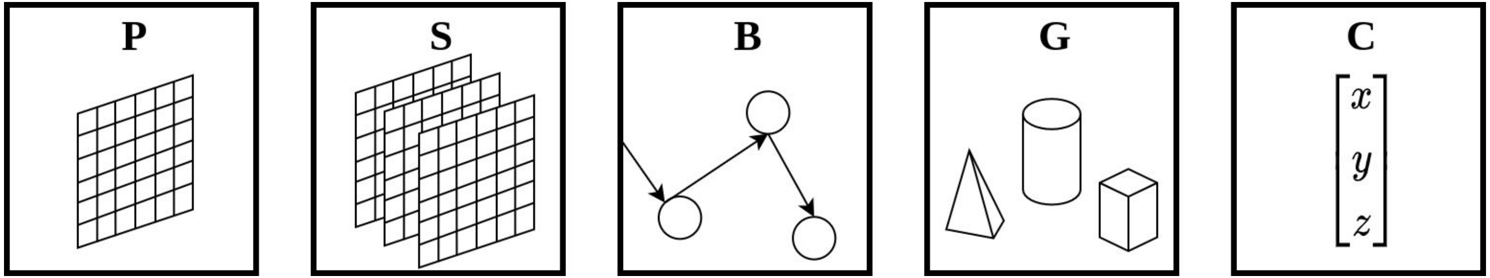
C vector components

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Ray Tracing Renderer

Algorithm 1: The Conceptual Workflow of Ray Tracing

```
for p = 1 to P do
    Image[p] = [0 0 0]
    for s = 1 to S do
        Ray = GenerateRay(p)
        for b = 1 to B do
            Initialize IntersectionResult
            for g = 1 to G do
                IntersectionResult = IntersectionTesting(Ray, GeometricObjects[g], IntersectionResult)
            if IntersectionResult.Hit then
                Image[p] = Image[p] + CalculateColor(LightSource, Ray, IntersectionResult)
                Ray = GenerateNextRay(Ray, IntersectionResult)
            else
                break
    Image[p] = Image[p] / S
```

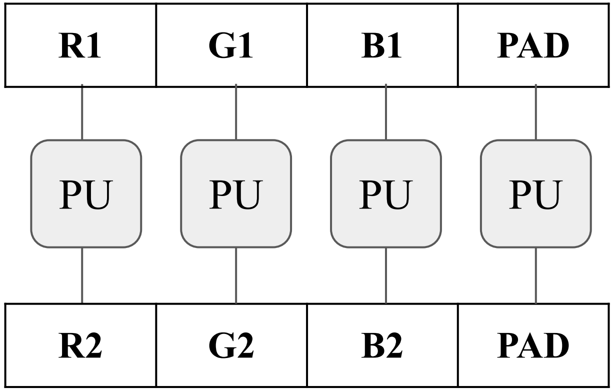


Proposed Solution

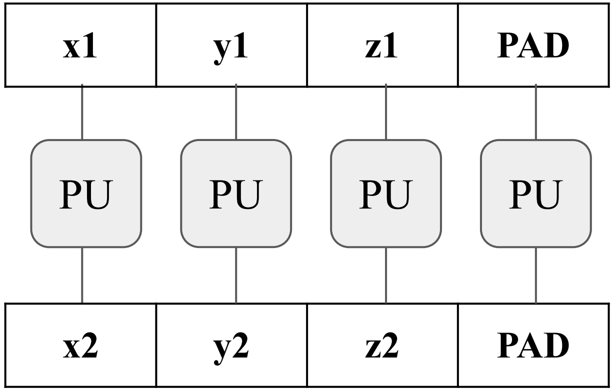
Proposed Solution (SIMD)

- Parallelize vector computation on arithmetic operations of floating point values
  - Addition, Multiplication, Dot Product, …

RGB computation



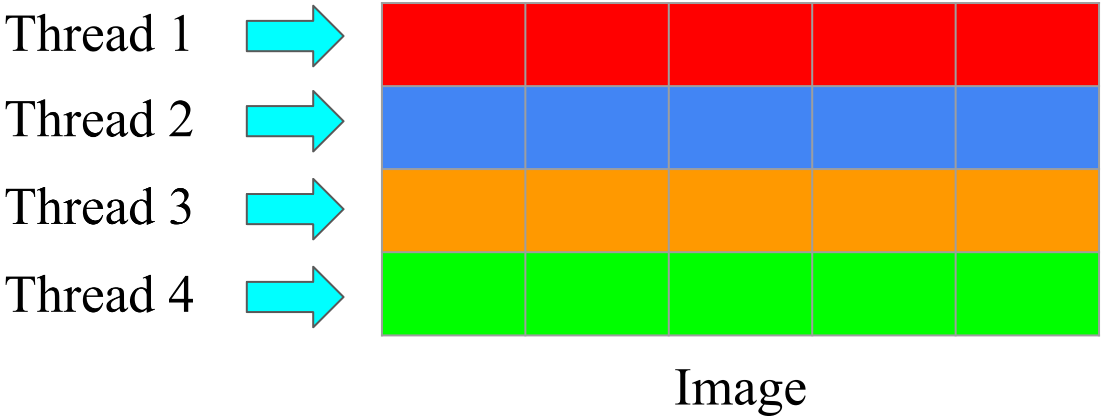
Coordinates computation



Proposed Solution (OpenMP)

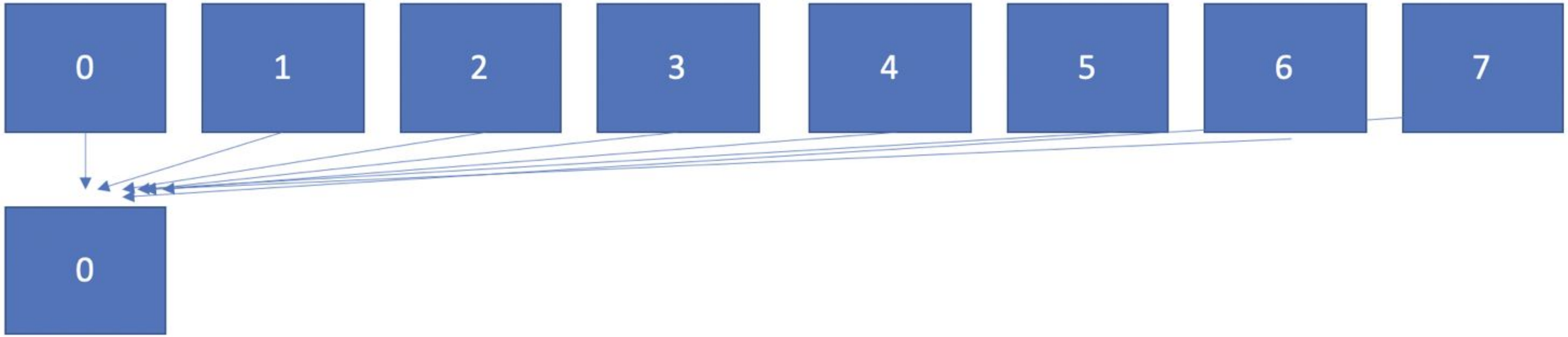
- Utilize the dynamic scheduling provided by OpenMP to dynamically allocate threads for parallel computation of each row in the image.

```
#pragma omp parallel for schedule(dynamic)
```



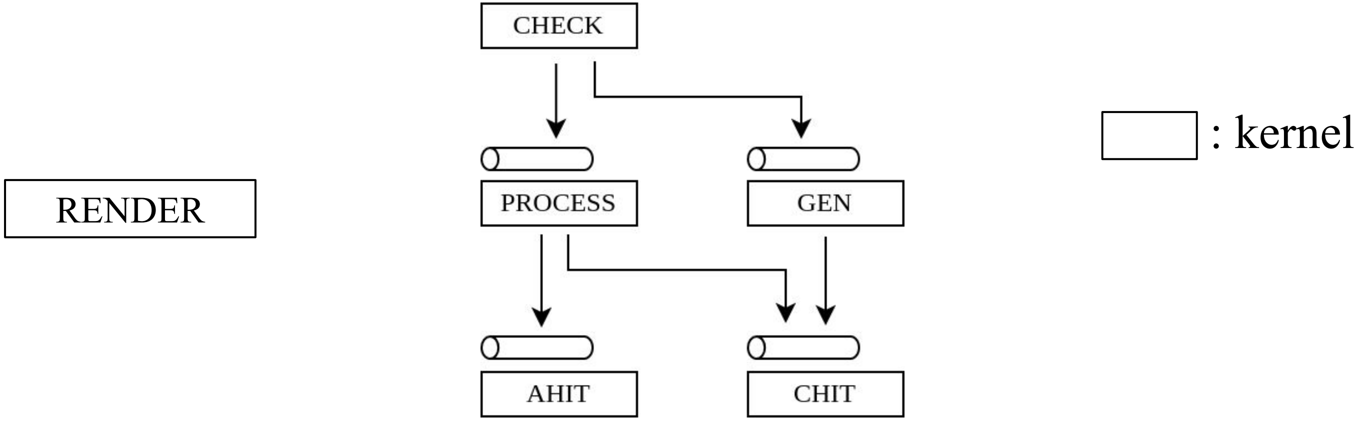
Proposed Solution (MPI)

- Distribute the rows of the image evenly among different servers for parallel computation.
- Finally, consolidate all calculations to generate the ultimate rendered image.



Proposed Solution (CUDA Mega/Split)

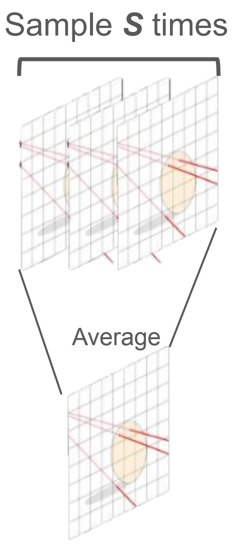
- We developed two versions of CUDA implementations to accelerate ray tracing
  - 1. Mega: Consolidates computations into a single “mega” kernel
  - 2. Split: Divides mega kernel into smaller kernels

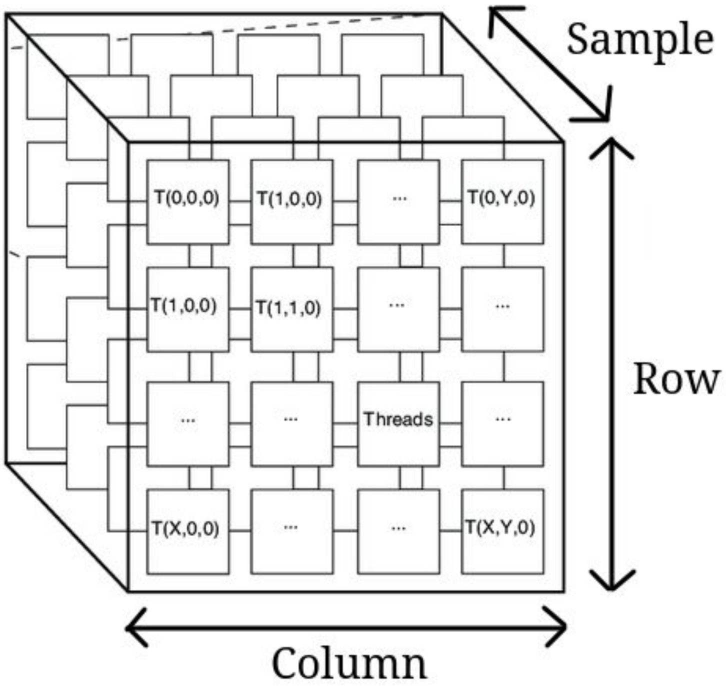


CUDA Mega CUDA Split

Proposed Solution (CUDA Mega)

- The CUDA Mega simply assign each CUDA thread a sample to work with

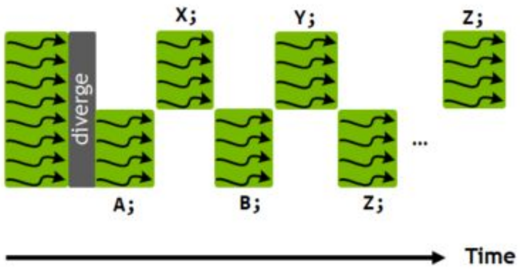




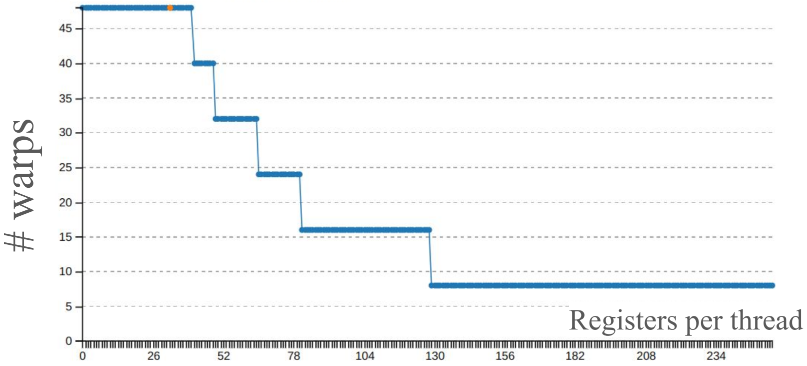
Proposed Solution (CUDA Mega)

- Disadvantages of mega kernel:
  - Divergence in control flow
  - High register usage ⇒ decreased concurrent warps

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



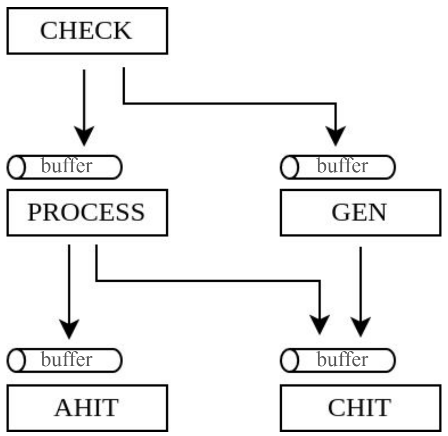
CUDA occupancy
Impact of Varying Register Count Per Thread

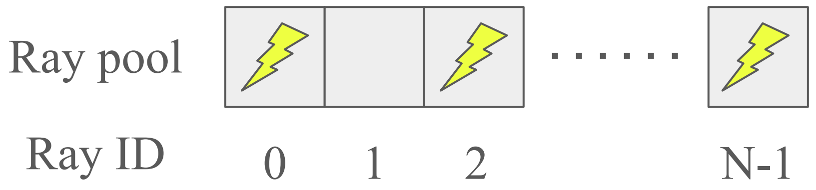


Megakernels Considered Harmful: Wavefront Path Tracing on GPUs [Laine et al., HPG 2013]

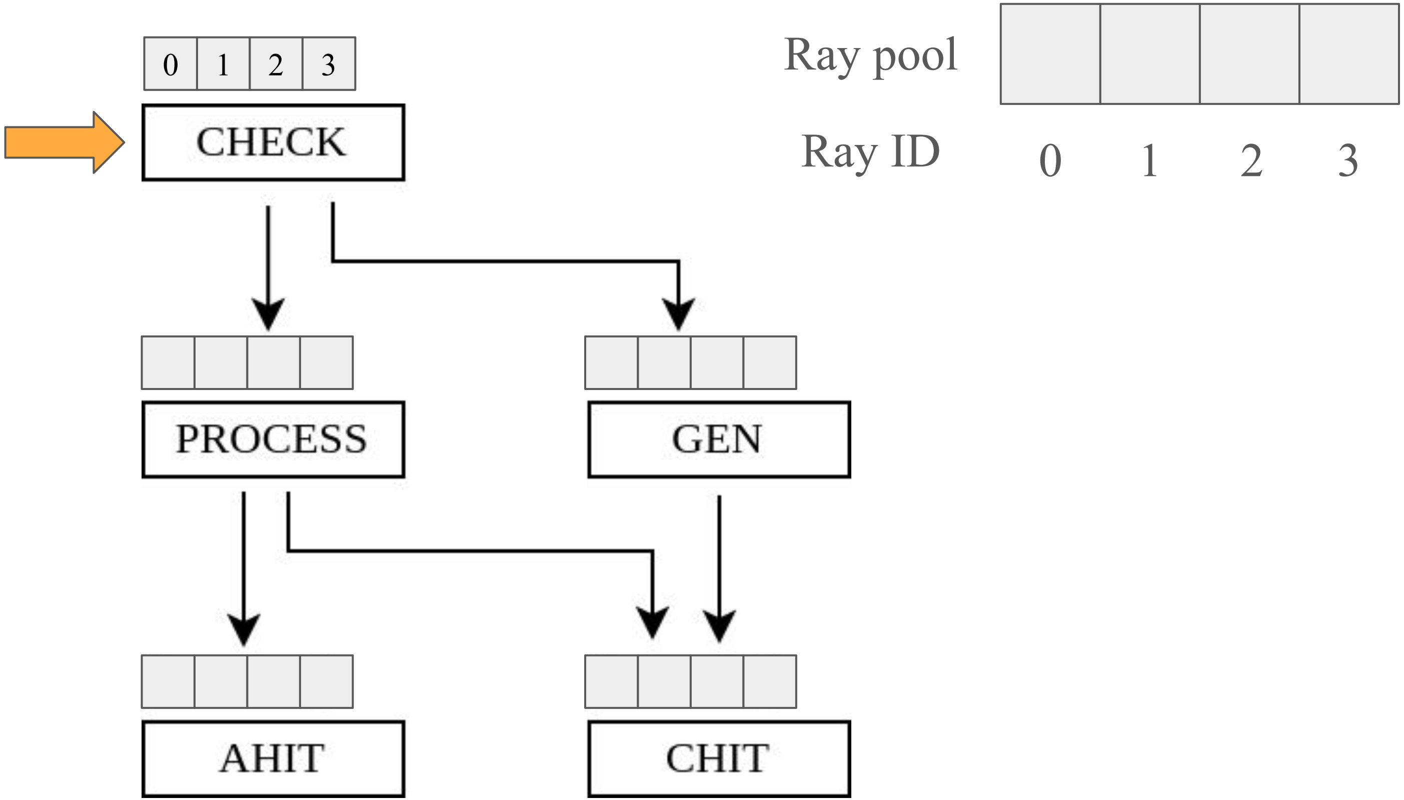
Proposed Solution (CUDA Split)

- We split mega kernel into five small kernels
  - CHECK, PROCESS, GEN, AHIT, CHIT
  - Rays are stored in ray pool
  - Different kernels communicate by passing ray ID to the target kernel's buffer

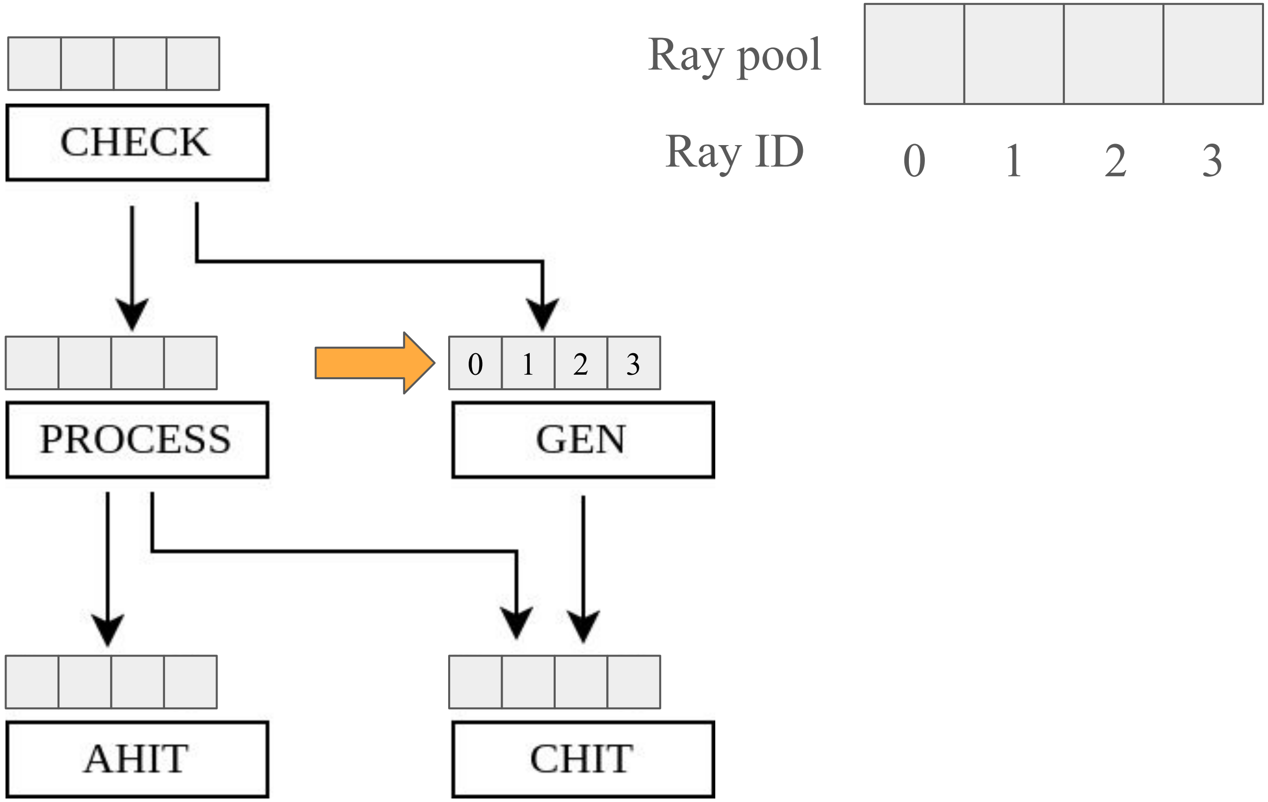




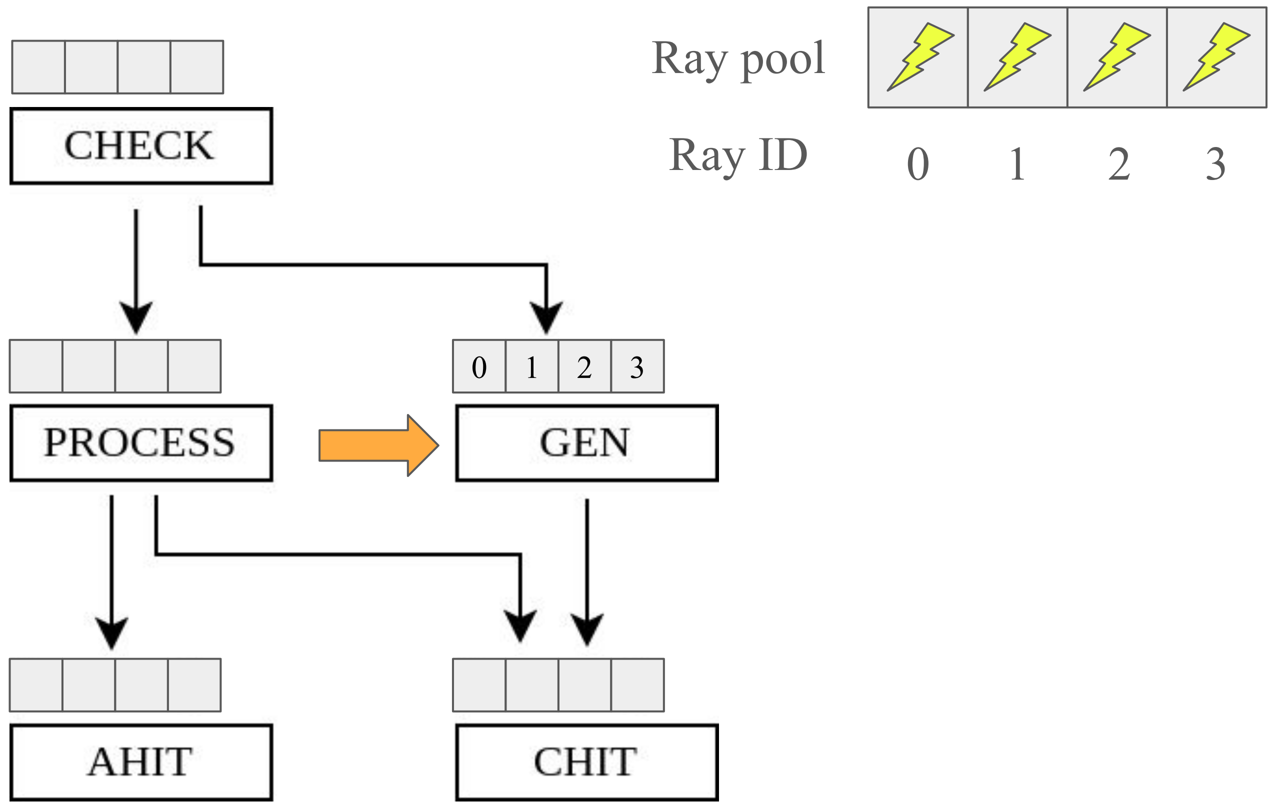
Proposed Solution (CUDA Split)



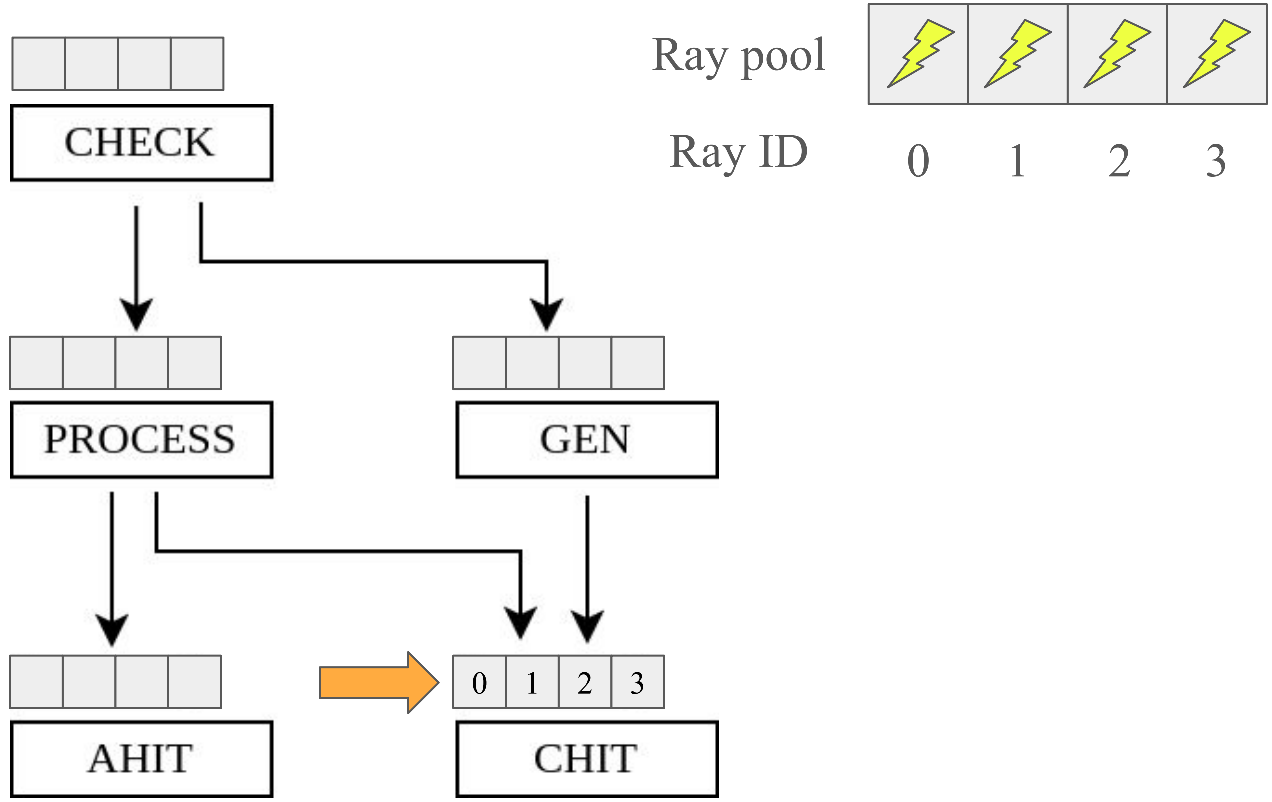
Proposed Solution (CUDA Split)



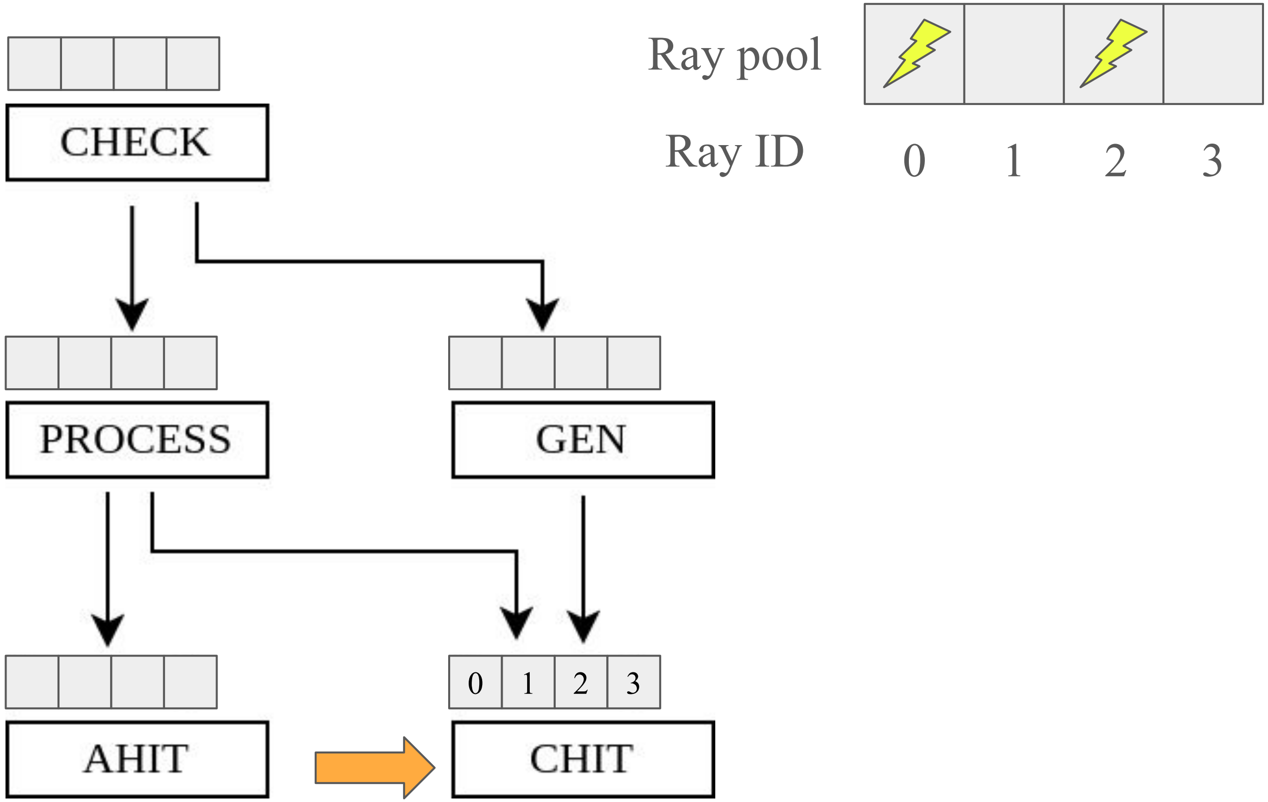
Proposed Solution (CUDA Split)



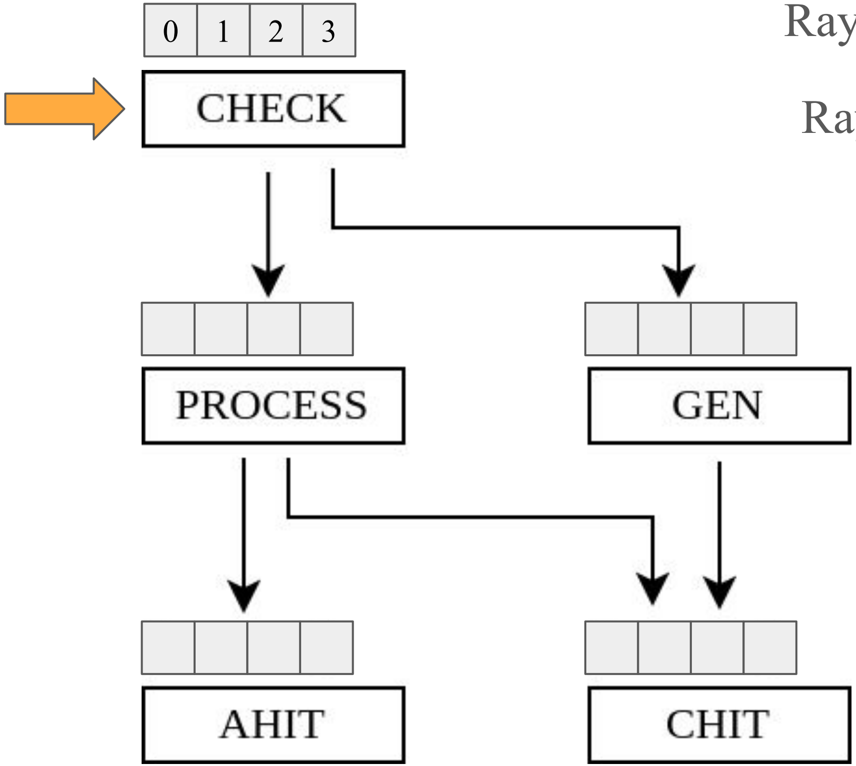
Proposed Solution (CUDA Split)

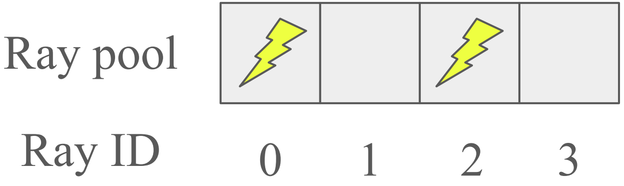


Proposed Solution (CUDA Split)

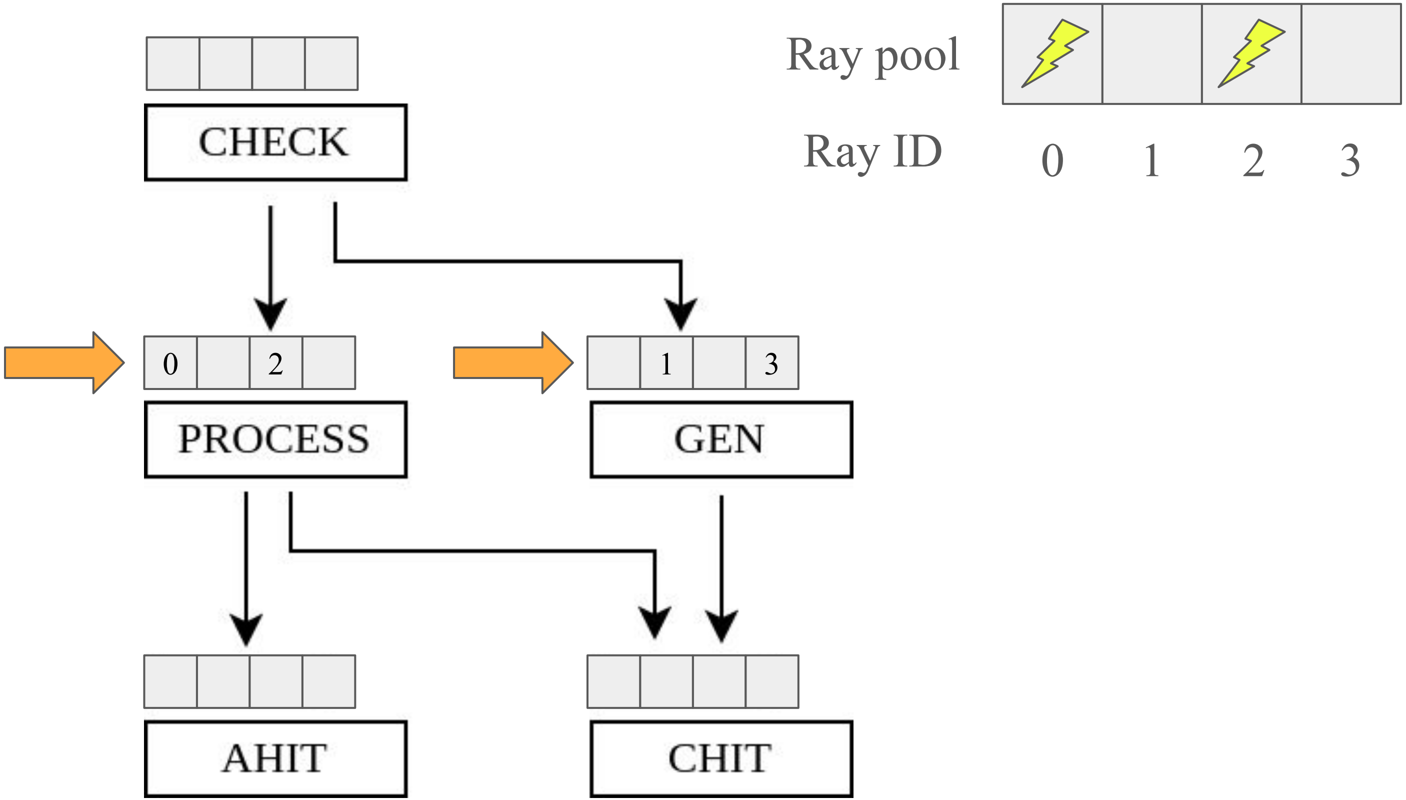


Proposed Solution (CUDA Split)

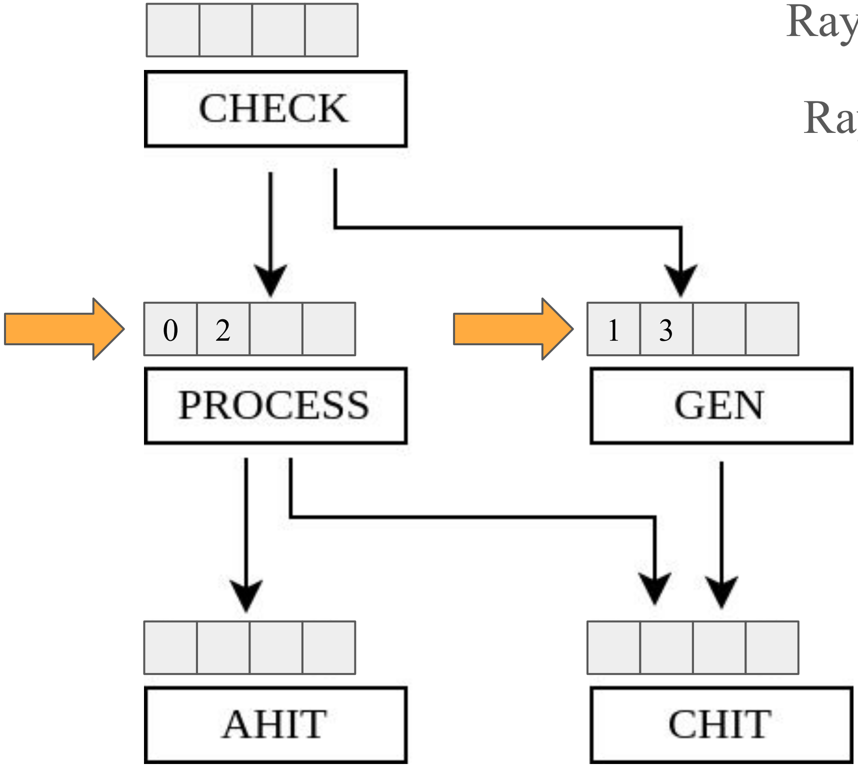


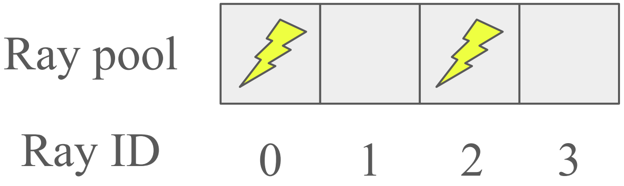


Proposed Solution (CUDA Split)



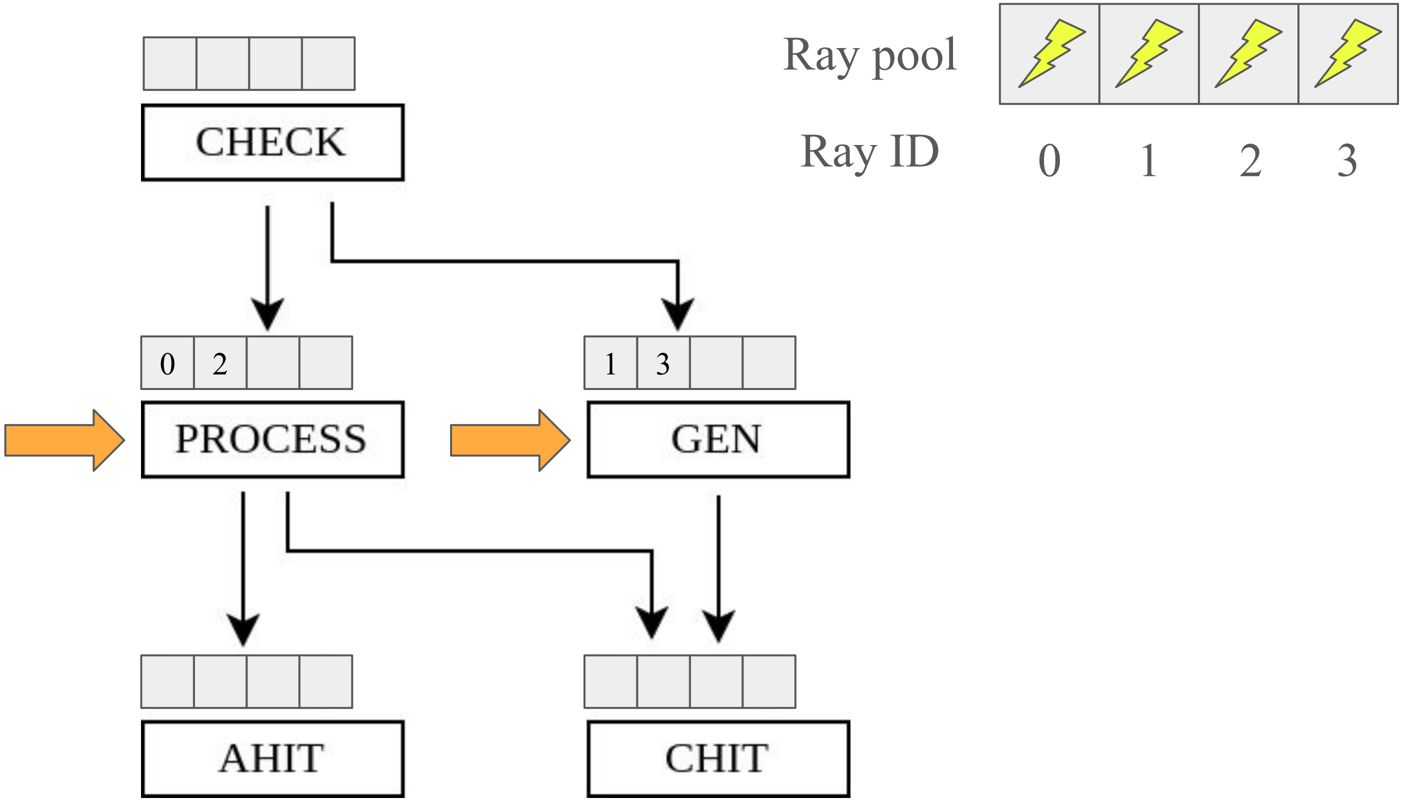
Proposed Solution (CUDA Split)



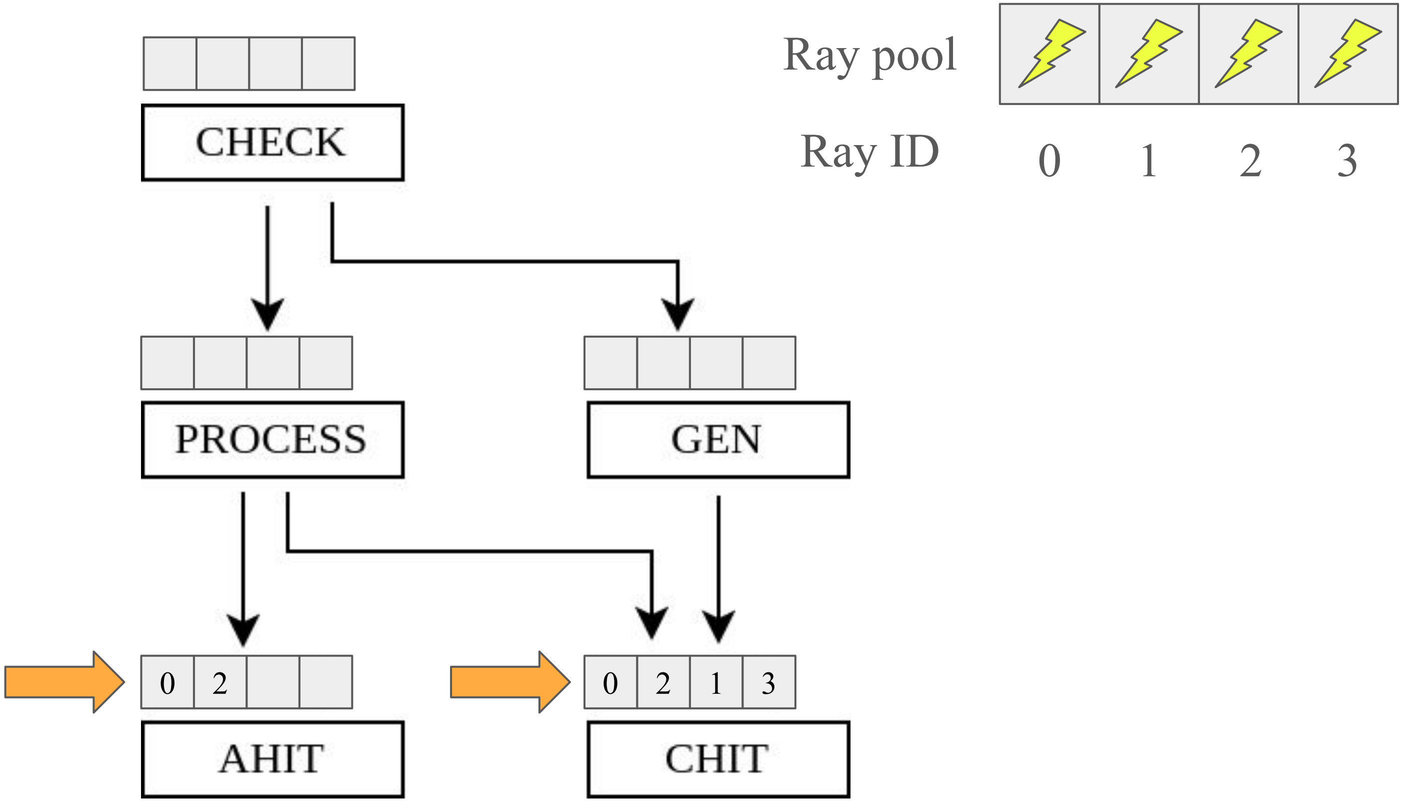


cub::DeviceSelect::Flagged()

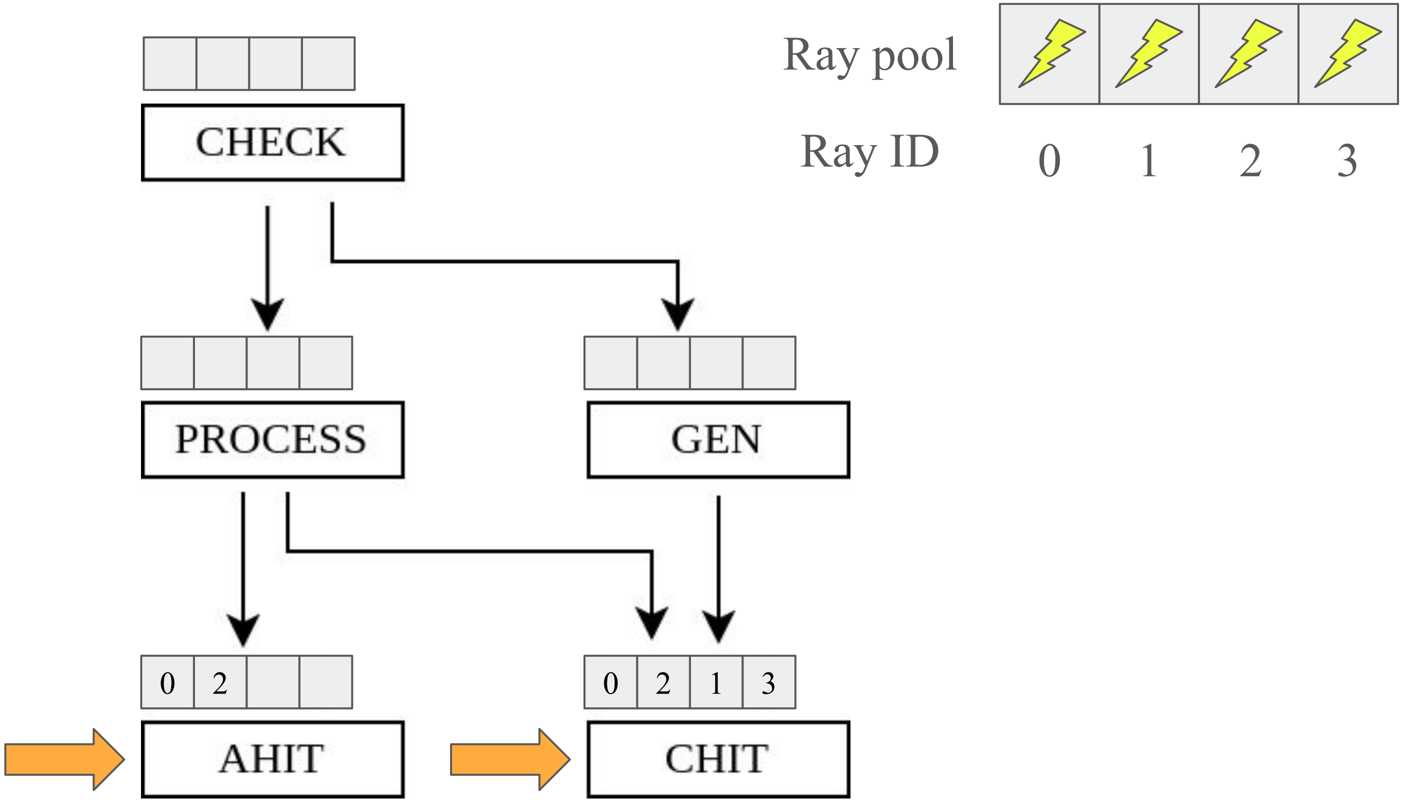
Proposed Solution (CUDA Split)



Proposed Solution (CUDA Split)



Proposed Solution (CUDA Split)



Proposed Solution (CUDA Split)

- Register usage before/after splitting kernels

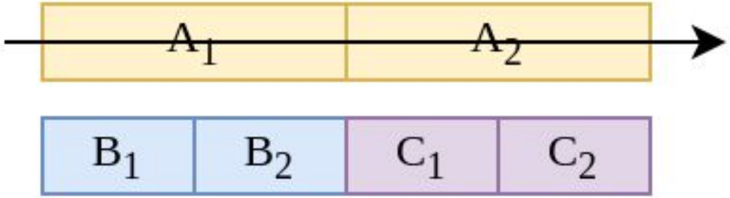
Kernel		Register Usage
Mega		65
Split	CHECK	14
	PROCESS	39
	GEN	28
	AHIT	32
	PHIT	43

Proposed Solution (CUDA Split)

- Coalescing memory access
  - array-of-structure ⇒ structure-of-array

array-of-structure

```
struct {
    double A;
    float B, C;
} S[2];
```

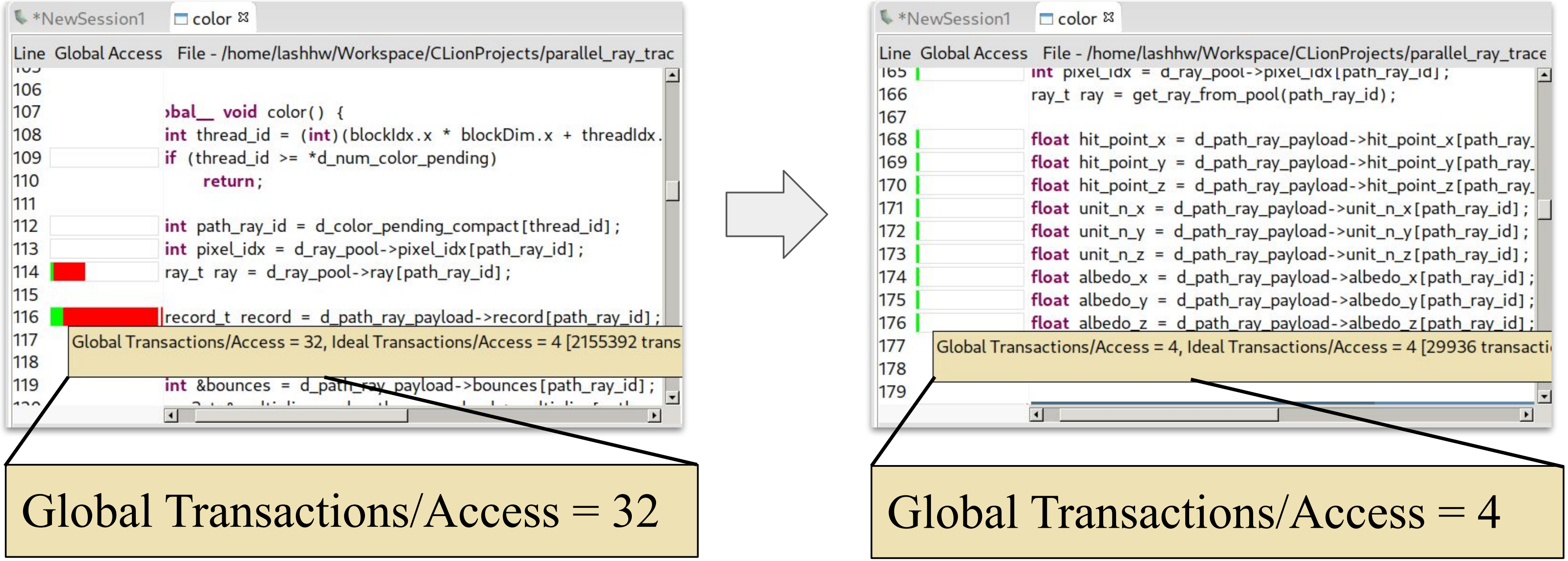


structure-of-array

```
struct {
    double A[2];
    float B[2], C[2];
} S;
```

Proposed Solution (CUDA Split)

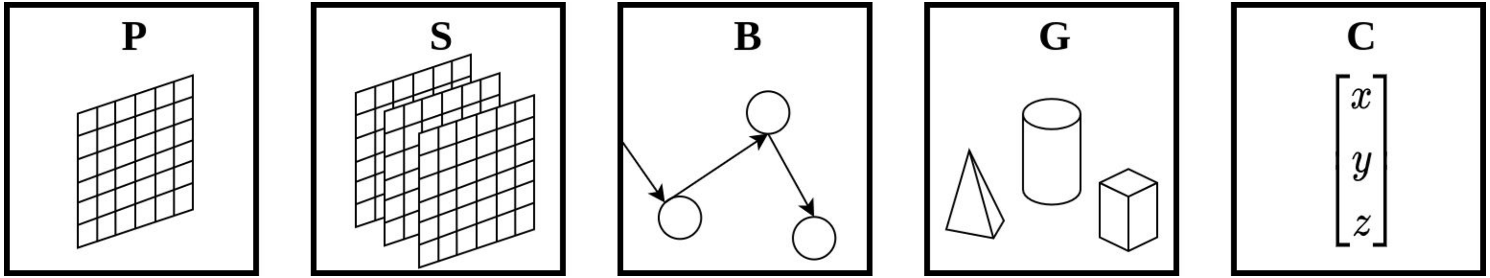
- array-of-structure ⇒ structure-of-array: 1.09x speedup!



Comparison of Parallelized Loops

Method	Parallized Loop
SIMD	C
OpenMP	P
MPI	P
CUDA	P, S

$\Rightarrow$ SIMD can be combined with OpenMP and MPI



Evaluation

Platform

OS	Ubuntu 20.04.2
CPU	Intel(R) Core(TM) i5-7500 @ 3.4GHz
GPU	NVIDIA Tesla T4
RAM	12GB
GCC	9.4.0
NVCC	10.1.243
Open MPI	4.0.3
CUDA	12.2
OpenMP	4.5

Evaluation

- Basic setting:
  - Image size: 600 x 600
  - Samples per pixel: 512
  - Maximum bounces per ray: 4

- Factors:
  - Image size
  - Samples per pixel
  - Maximum bounces per ray

Implementation	SIMD	Time (s)	Speedup
Serial	✗	102.46	1.00x
	✓	95.00	1.08x
OpenMP*	✗	39.65	2.58x
	✓	38.90	2.63x
MPI†	✗	15.75	6.51x
	✓	15.65	6.55x
CUDA (Mega)	✗	6.64	15.43x
CUDA (Split)	✗	3.12	32.84x

*4 threads †8 processes

Experiment 1

- Different image size *(P)*

	150 x 150		300 x 300		600 x 600	
	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
Serial	5.99	1.00x	25.08	1.00x	95.00	1.00x
OpenMP*	2.98	2.01x	9.70	2.58x	38.90	2.44x
MPI†	3.52	1.70x	4.59	5.46x	15.65	6.55x
CUDA	0.39	15.36x	0.92	27.26x	3.12	30.45x

*4 threads †8 processes

Experiment 2

- Different number of samples per pixel *(S)*

	256		512		1024	
	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
Serial	47.43	1.00x	95.00	1.00x	189.46	1.00x
OpenMP*	24.22	1.96x	38.90	2.44x	77.79	2.44x
MPI†	8.32	5.7x	15.65	6.55x	30.29	6.25x
CUDA	1.64	28.92x	3.12	30.25x	6.02	31.47x

*4 threads †8 processes

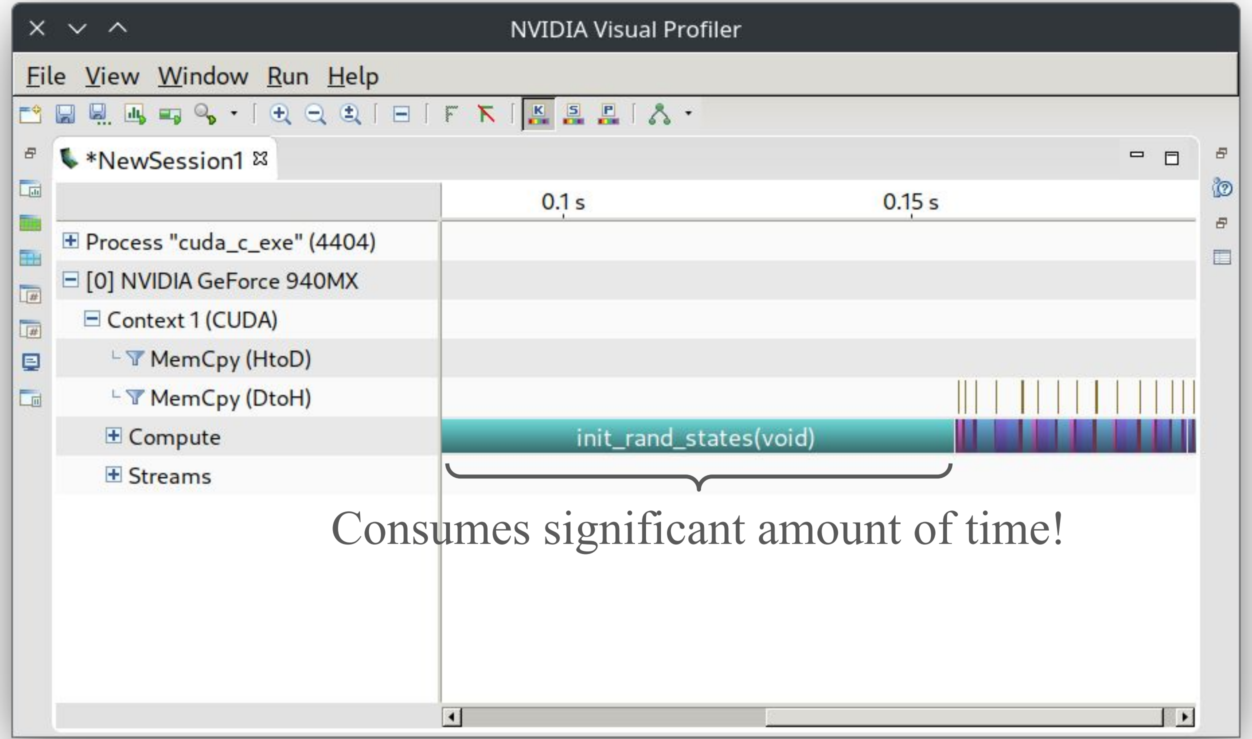
Experiment 3

- Different number of bounces per ray *(B)*

	2		4		8	
	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
Serial	49.78	1.00x	95.00	1.00x	146.39	1.00x
OpenMP*	21.83	2.28x	38.90	2.44x	60.15	2.43x
MPI†	7.94	6.27x	15.65	6.55x	24.82	5.90x
CUDA	2.21	22.52x	3.12	30.45x	4.02	36.42x

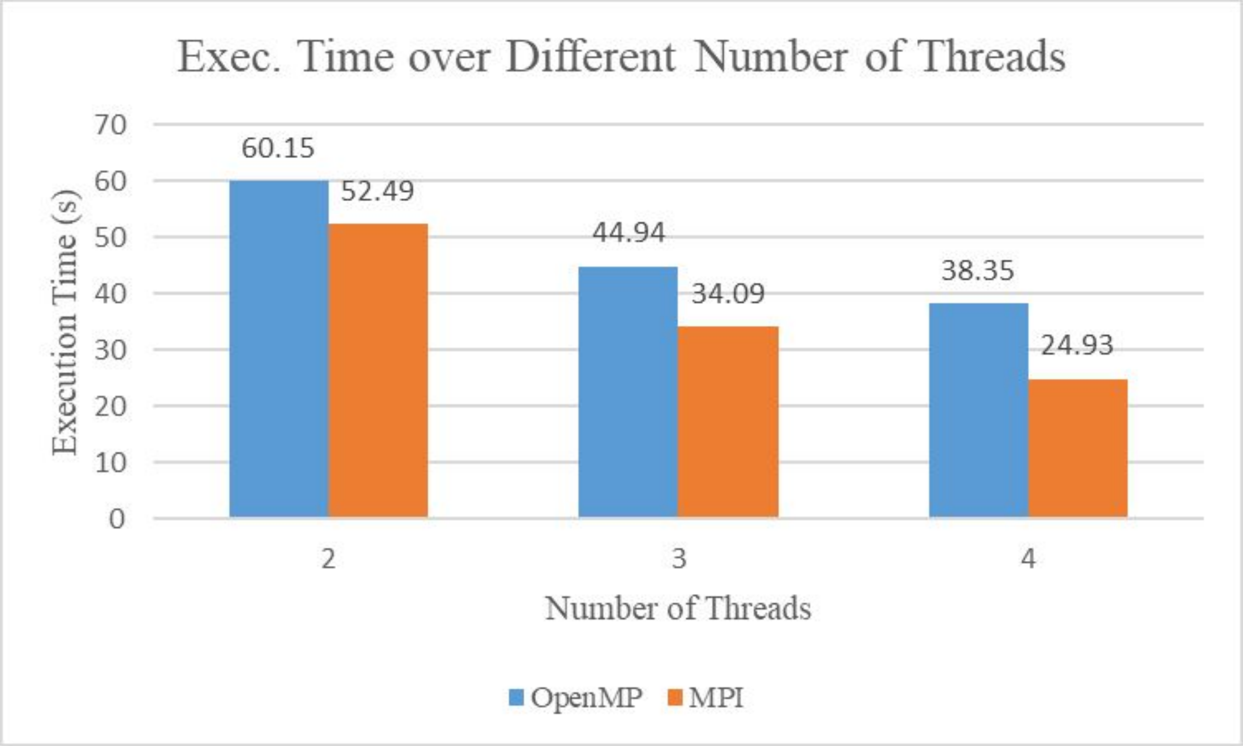
*4 threads †8 processes

Experiment 3



Experiment 4

- Compare OpenMP with MPI at same number of threads



Conclusion

Conclusion

- Our study demonstrated that CUDA significantly outperformed CPU methods in parallelizing a ray tracing renderer.

- Notably, the split kernel approach of CUDA showed the highest speedup, highlighting the effectiveness of parallel computing.