# GPU Interoperability for Standard Template Adaptive Parallel Library

**Shaoshao Xiong, Prahit Yaugand, Swarup Majumder, Nihar Kalode, Francisco Coral, Lawrence Rauchwerger**

Department of Computer Science, College of Engineering, University of Illinois at Urbana-Champaign

Parasol
Smarter Computing.
University of Illinois Urbana-Champaign

## Introduction

### What is STAPL?

The Standard Template Adaptive Parallel Library (STAPL) is a framework designed for parallel programming in C++. STAPL offers a collection of parallel algorithms that resemble those in the C++ Standard Template Library (STL), but are specifically adapted for execution across multiple processors.

### Motivation

GPUs have been used to extensively increase computation speed for parallel programming. Integrating GPUs in STAPL algorithms will likely lead to speedups due to the massively parallel nature of GPUs.

### Objectives

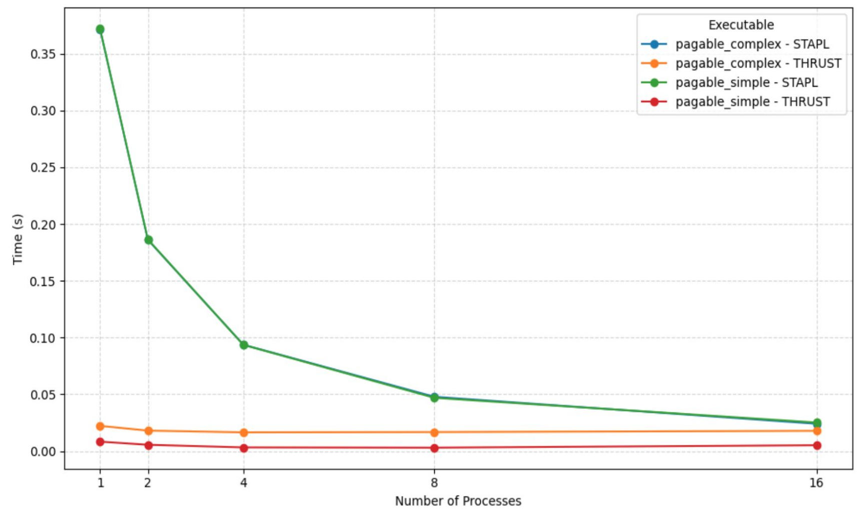
The objective was to study the scalability and interoperability of STAPL algorithms using multiple GPUs and implement STAPL algorithms with CUDA Thrust library. The research further aimed to evaluate how the complexity of a work function would affect the speedup.

## Testbench

- 8 Nvidia H100 GPU (Campus Cluster)
- 10 Million Integers Array Input Size
- for_each algorithm
- Simple function: $x = x + (2^{14} \% 3 + 1) * 50 + 3^{10}$
- Complex function: x = sinf(x + i * 0.001f) for i in [0,999]
- 64 sample tests

## STAPL CPU vs GPU Initial Results

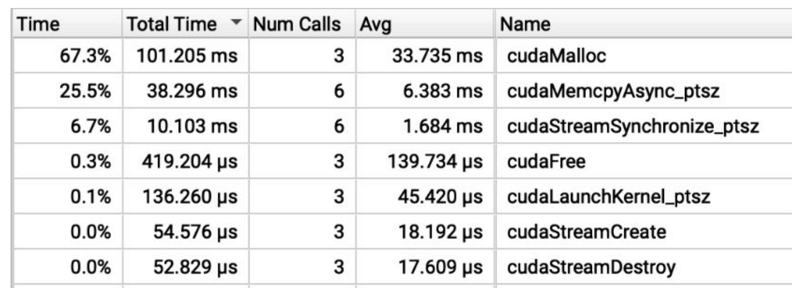
- CPU - Splits array into segments based on process count and performs computation on each segment
- GPU - Copies each segment of array into GPU; performs computation with GPU kernels; copies back data to CPU
- CPU complex time is not visible as the fastest runtime takes at least 10 seconds.



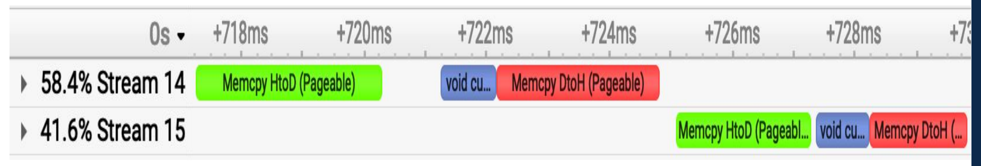
## Discovering the Bottlenecks

### Nvidia Nsight Profiler

- Pageable memory operations (default cudaMemcpy) consume significant time
- CUDA uses lazy initialization
- cudaMemcpy blocks the CPU

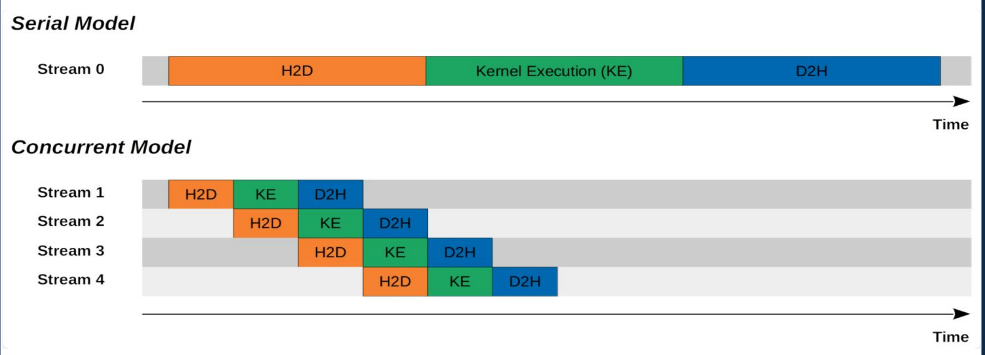


| Time | Total Time | Num Calls | Avg | Name |
|---|---|---|---|---|
| 67.3% | 101.205 ms | 3 | 33.735 ms | cudaMalloc |
| 25.5% | 38.296 ms | 6 | 6.383 ms | cudaMemcpyAsync_ptsz |
| 6.7% | 10.103 ms | 6 | 1.684 ms | cudaStreamSynchronize_ptsz |
| 0.3% | 419.204 μs | 3 | 139.734 μs | cudaFree |
| 0.1% | 136.260 μs | 3 | 45.420 μs | cudaLaunchKernel_ptsz |
| 0.0% | 54.576 μs | 3 | 18.192 μs | cudaStreamCreate |
| 0.0% | 52.829 μs | 3 | 17.609 μs | cudaStreamDestroy |



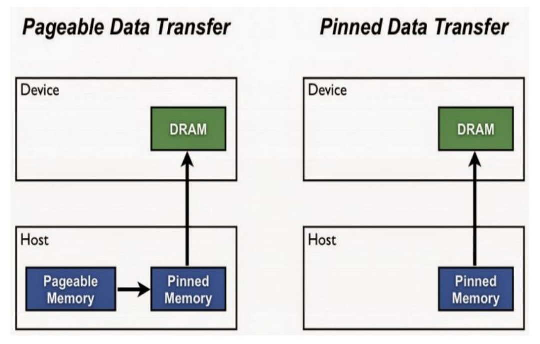
### Solution: Concurrency with CUDA Streams

- Operations from different streams may be interleaved
- Does not block the CPU with async API functions
- Streams with pinned memory allow for concurrent copies between host (CPU) and device (GPU)



## Pinned Memory vs Pageable Memory

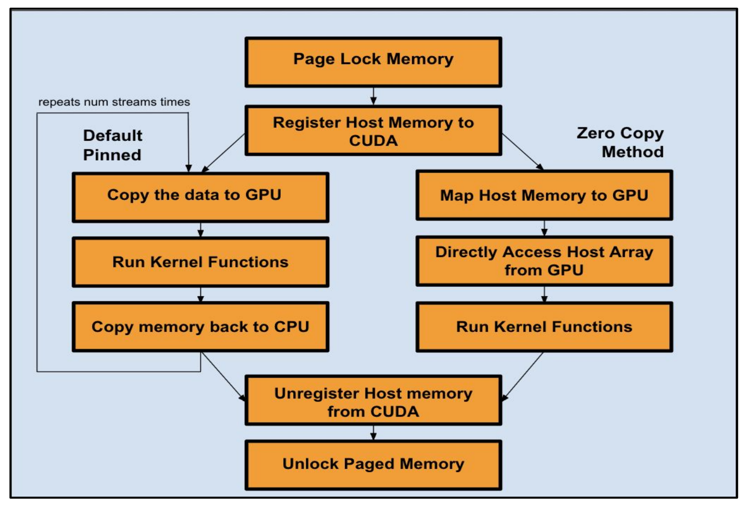
- Pinned (page-locked) memory allows for faster copies between hosts and devices
- Asynchronously copies data between GPU and CPU
- Used with streams to overlap execution



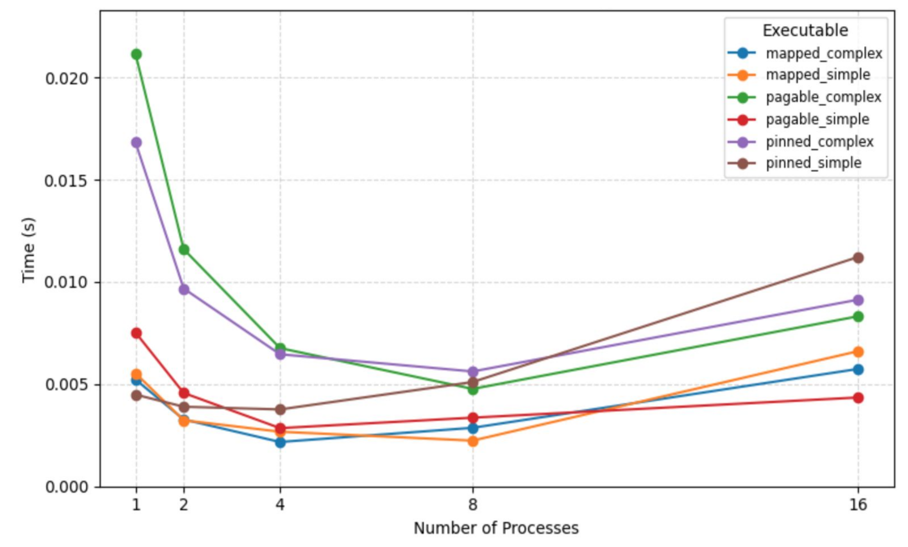
### Pinned Memory Workflow

The diagram shows the differences between two approaches to using pinned memory.

- Default pinned- data copied to GPU global memory
- Mapped (Zero copy) - GPU reads and writes directly to and from host memory under unified address space



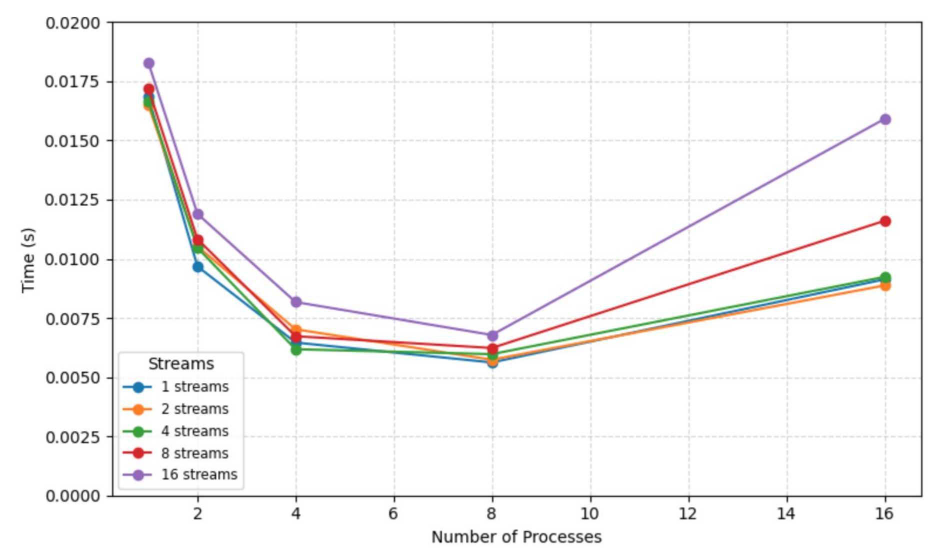
## Pageable vs Pinned vs Mapped (GPU)



### Results

This graph shows that mapped memory shows a significant speedup over pageable memory, especially for complex functions. Strong scaling is only present for up to 8 processes due to the presence of only 8 GPUs in the testing suite.

## Pinned with complex function (GPU)

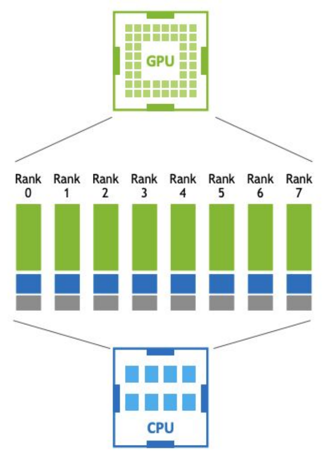


### Results

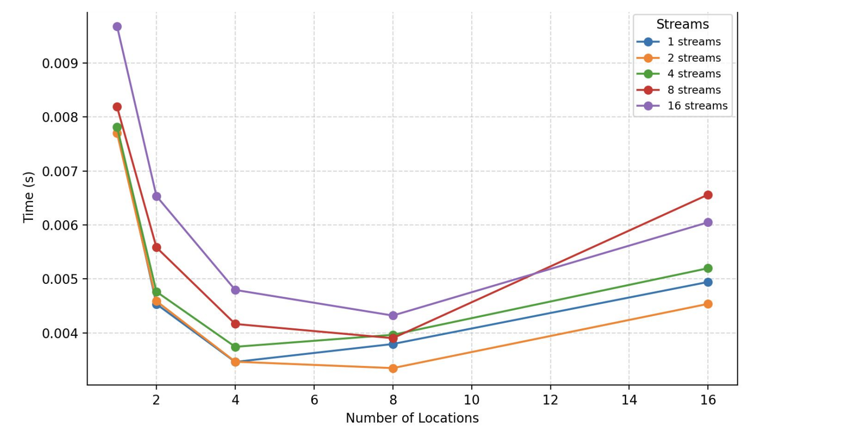
This graph shows that the number of streams did not have a significant effect on the timings for the complex function. This is due to the overhead in stream creation and destruction cancelling with the speedup from split computation.

## Multi-Process-Services

- Allow multiple CUDA processes to share GPU compute resources
- Concurrently maps multiple locations (processing unit) onto a single GPU
- Maximizes GPU occupancy when a single process doesn't fully utilize the resources



### Pageable(Complex Function)



## Conclusions and Future Plans

Based on the complexity of the functor, GPUs have provided 10-100x computation speedups compared to CPU execution. Although scaling is not as strong compared to STAPL CPU locations, this research has shown that performance gains are significant and scaling is still present. Future research can focus on implementing other STAPL algorithms in CUDA. Additionally, speedups through inter-GPU communication can be achieved by utilizing the NCCL GPU communication library and NVIDIA's topology-aware GPU selection. Investigating the effects of combining caching and zero-copy memory on GPUs and physical partitioning of GPU resources using Multi-Instance GPU would also be beneficial.

## Acknowledgements

We would like to thank Dr. Coral, Professor Rauchwerger, Marie, and the Parasol Lab for providing us with the opportunity to learn, gain experience, and form new connections through the SRP.

## References

- NVIDIA Developer Blog
- CUDA by Example Book
- CUDA Runtime API Documentation
- STAPL Library
- NSight Systems documentation

I ILLINOIS