# Programming Hybrid CPU-GPU Clusters with Unicorn

Subodh Kumar
IIT Delhi

Part of Tarun Beri's PhD thesis

# Parallel Programming is Hard

- Problem decomposition
- Processor mapping
- Scheduling
- Communicated and synchronization
- Tuning to hardware
- Maintainable and portable code
- Programmer productivity
- Scalability
- Managing multiple types of parallelism
  - accelerator, shared memory, cluster, message passing
- Thread model is non-deterministic
- Low level locking prone to deadlocks and livelocks
- Large numbers still trained on sequential models of computation
  - No effective bridging model

# About Unicorn

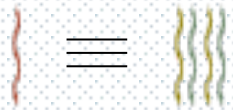
- Multi-node, Multi-CPU, Multi-GPU programming framework
  - Shared memory style
  - Bulk synchronous
- For coarse grained compute-intensive workloads
- Designed to adapt to the totality of effective network, memory and compute throughputs of devices

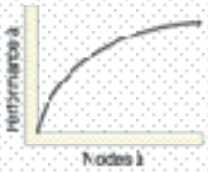
Current implementation
- Assumes flat network topology
- No elaborate matching of device capability to workload

# Unicorn's Goals

- To design a cluster programming model which is:

| Simple | Hetero-geneous | Scalable | Unified | Abstract |
|---|---|---|---|---|
| Complexity largely in sequential native code | Works on hybrid CPU-GPU clusters | Performance increases with cluster nodes | Common API for CPU/GPU | Agnostic to network topology/ data organization |





# Programming with Unicorn

- Plug-in existing sequential, parallel CPU and CUDA code
  - Debugging complexity “near” sequential/native code
- Shared memory style but deterministic execution
  - No data races
  - Check-in/check-out memory semantics with conflict resolution
  - No explicit data transfers in user code
    - Internally, latency-hiding with compute-communication overlap is first class citizen
  - Automatic load balancing and scheduling
- Code agnostic to data placement, organization and generation
  - No requirement of user binding data or computations to nodes

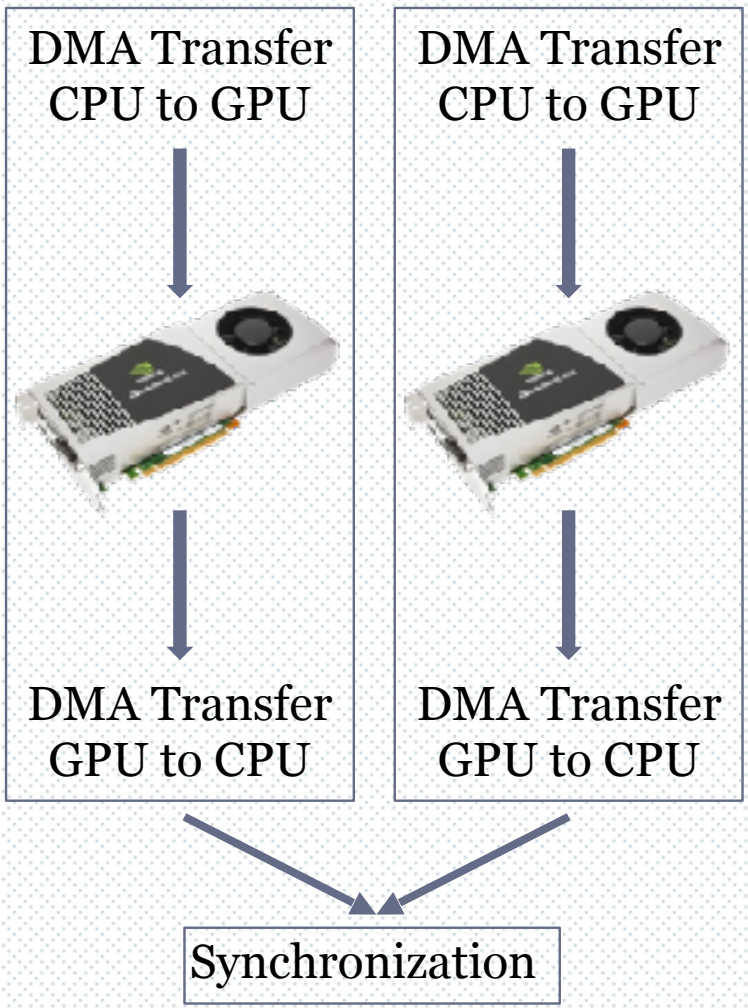
# Many Competing Approaches

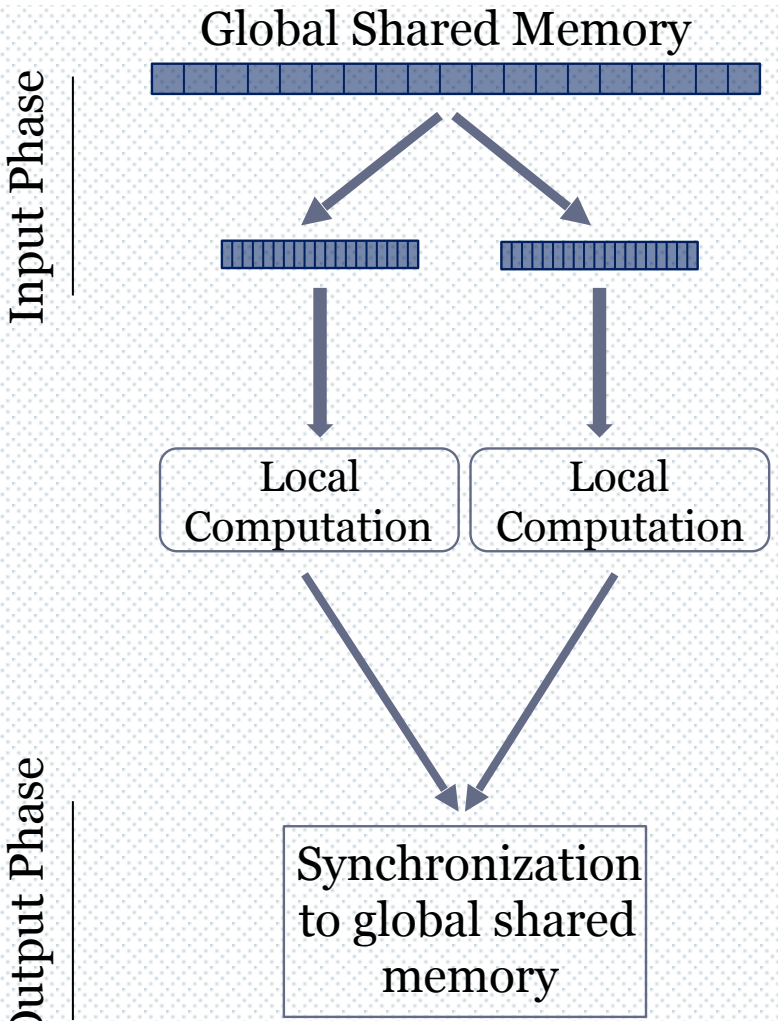
- Task graph partitioning
- Express units of work
- Directives
- Loop parallelization and scheduling
- Distributed address space (PGAS)

**Data Scheduling**

Legion
Global Arrays
X10
Split-C
Cilk
POP-C++
Titanium
Sequoia
MPI
Globus
MPI-ACC
StarPU-MPI
Phalanx
Charm++/G-Charm
Mekong

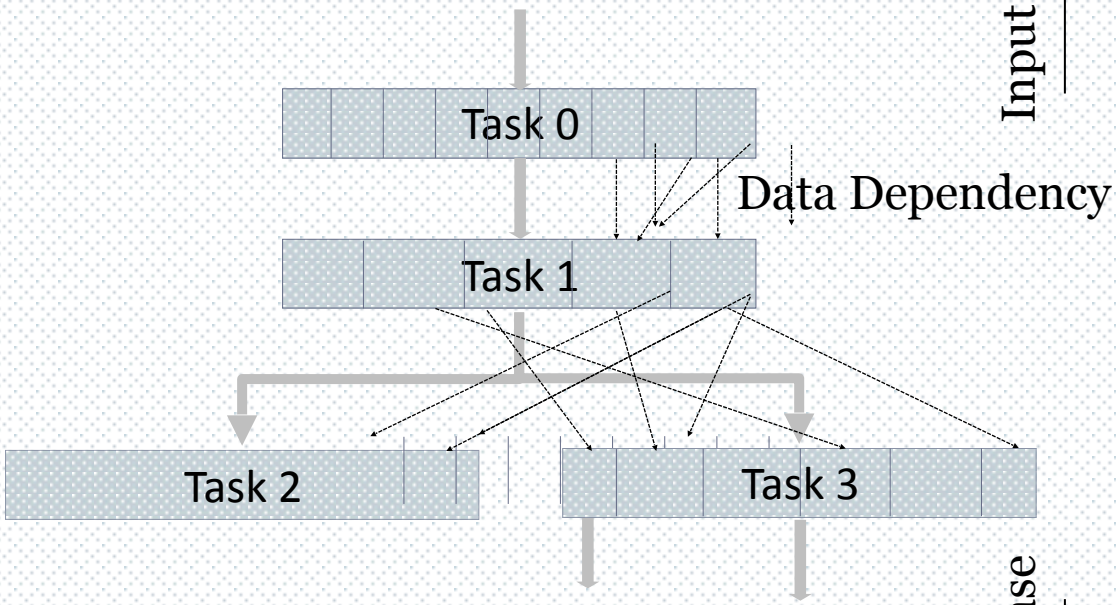
# Bulk Synchronous?



# Unicorn Programming Model

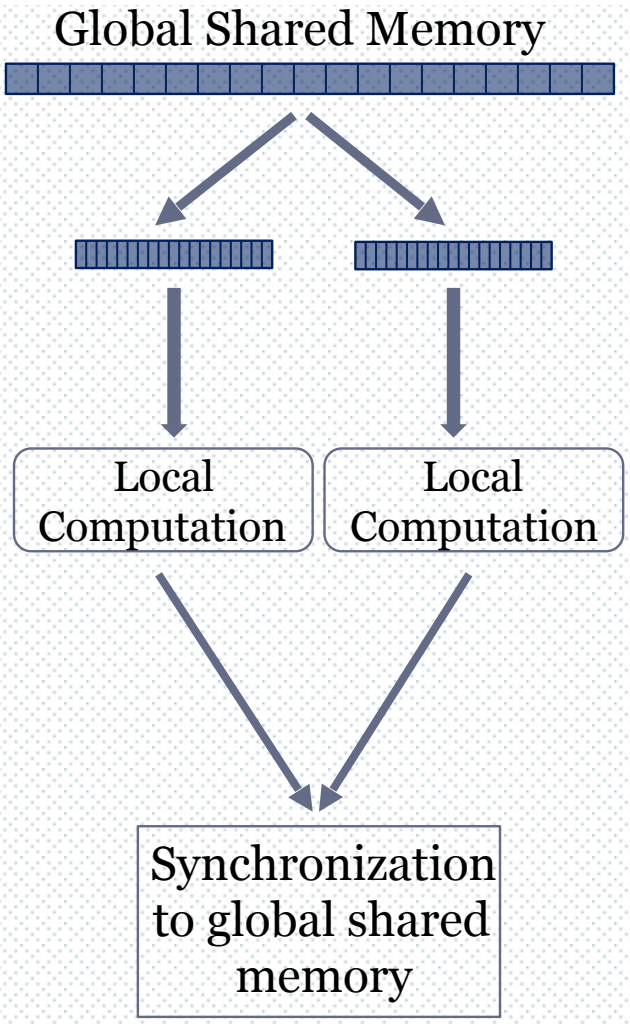
A parallel program is a graph of tasks



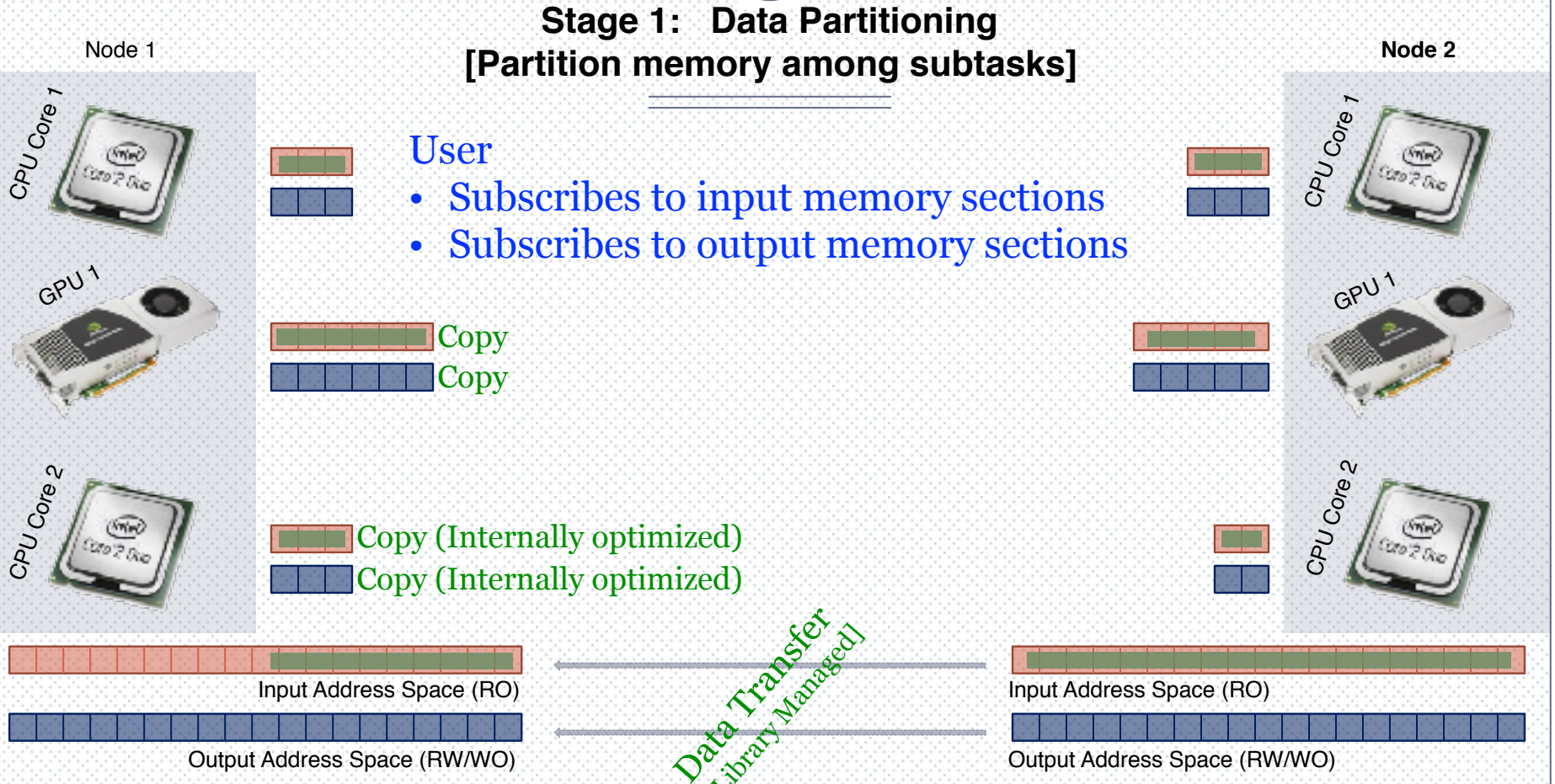

Tasks are divided into multiple concurrently executable subtasks

Input Phase

Output Phase

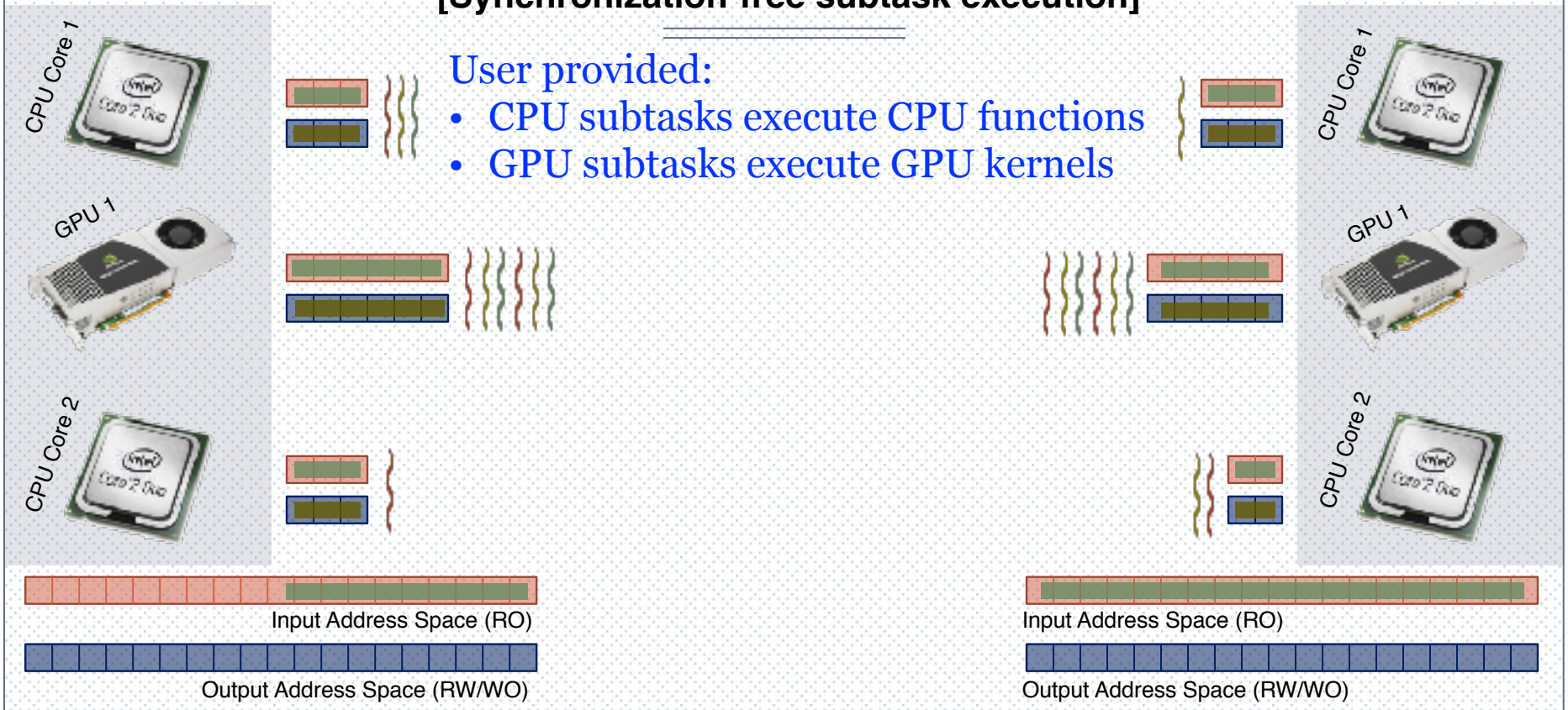


# Unicorn – Data Partitioning



# Unicorn – Subtask Execution

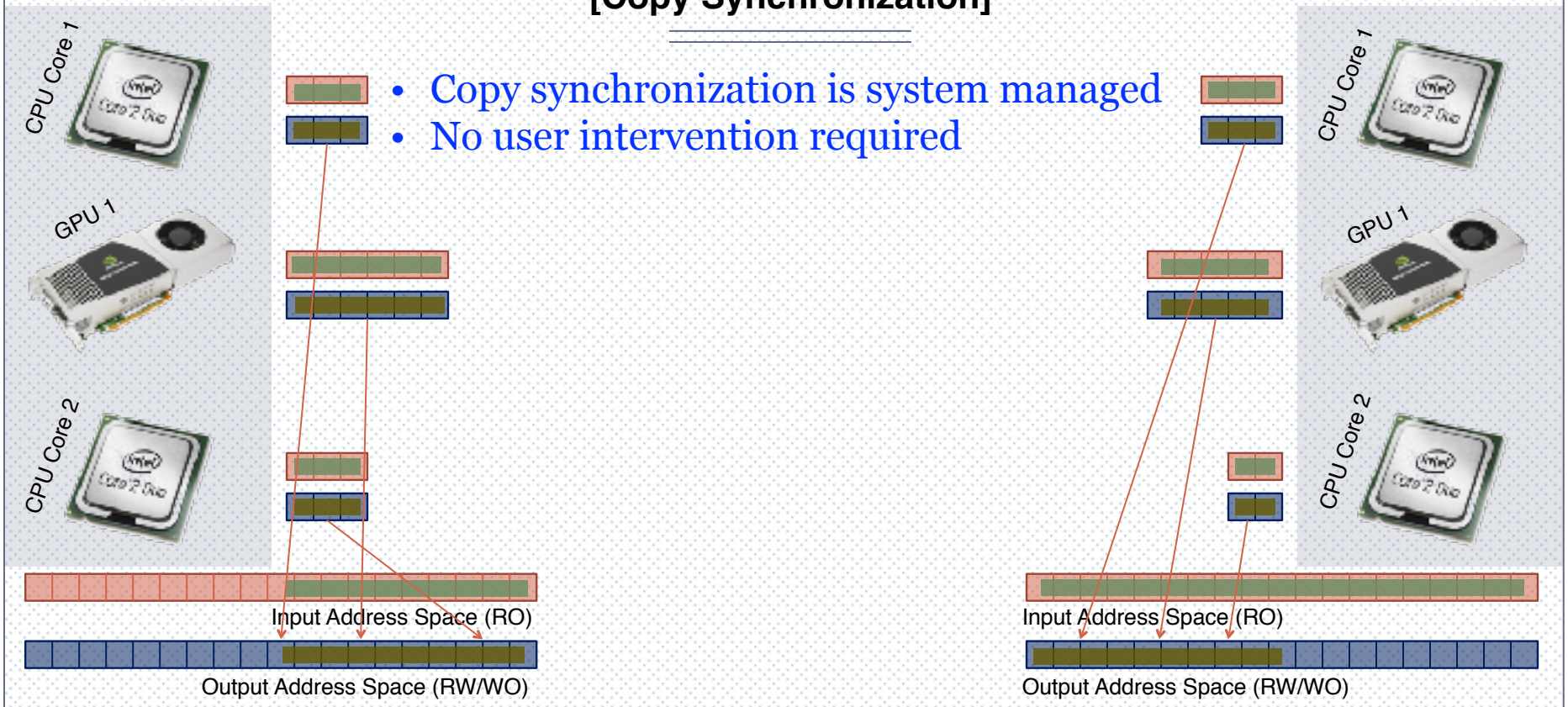
**Stage 2: Computation**
**[Synchronization-free subtask execution]**



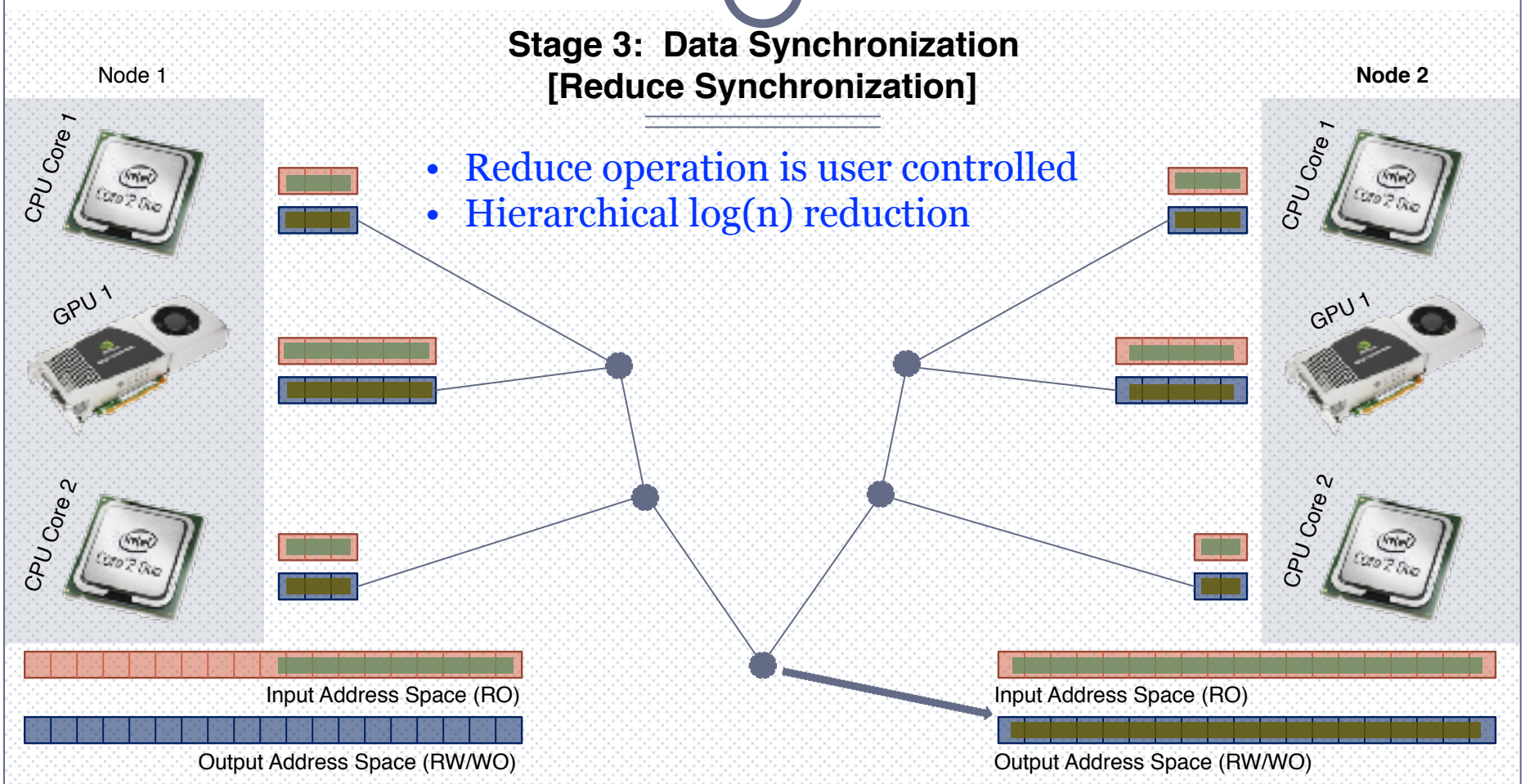
# Unicorn – Data Synchronization

**Stage 3: Data Synchronization [Copy Synchronization]**

- Copy synchronization is system managed
- No user intervention required



# Unicorn – Data Reduction



# What a program looks like?

```
struct complex { float real, imag; };
struct fft_conf { size_t rows, cols; };

fft_1d(matrix_rows, matrix_cols)
{
    key = "FFT";
    register_callback(key, SUBSCRIPTION, fft_subscription);
    register_callback(key, CUDA, "fft_cuda", "fft.cu");

    if(get_host() == 0) // Submit task from single host
    {
        size = matrix_rows * matrix_cols * sizeof(complex);
        input = malloc_shared(size);
        output = malloc_shared(size);

        initialize_input(input); // application data

        // create task with one subtask per row
        nsubtasks = matrix_rows;
        task = create_task(key, nsubtasks, fft_conf(matrix_rows, matrix_cols));

        bind_address_space(task, input, READ_ONLY);
        bind_address_space(task, output, WRITE_ONLY);

        submit_task(task);
        wait_for_task_completion(task);
    }
}
```

Define task callbacks

Allocate address spaces

Create task

Bind address spaces to task

Submit task for asynchronous execution

# Pillars of Unicorn

- Understand the flow of data and balance load
  - Pipeline data delivery and computation
- Parallelize at multiple levels
  - Inner loops often data parallel
    - Scientific computation
  - Coarse grained outer level
- Sandbox computation
  - No data race
  - Transactional semantics
  - Data reduction, assimilation, re-organization

# Runtime Optimizations

- Distributed directory maintenance
  - Non-coherent opportunistic lock-free updates
  - MPI style views
- Schedule data pre-fetch (among nodes and to/from GPU)
  - Software GPU cache
  - Hierarchical steal, Pro-active steal, granularity adjustment
  - Locality aware scheduling
    - Local estimate of partial subtask affinity
    - Register top affinities with Claim central
    - Time to fetch non-resident, rather than size of resident data
  - Locality-aware work stealing
- Automatic device conglomeration
- Network message merging, compression, etc.

# Experimental Setup

## Node Configuration (14 node cluster)

Intel Xeon X5650 2.67 GHz CPUs
2 CPUs with six cores each
64 GB main memory
2 Tesla M2070 GPUs
32Gbps InfiniBand (QDR) network

**Total number of devices in the cluster = 196**

# Experiments

## Image Convolution

24-bit RGB image
$2^{16}$ x $2^{16}$ pixels
31 x 31 filter
1024 subtasks

## Matrix Multiplication

$2^{16}$ x $2^{16}$ matrices
1024 subtasks

## 2D C2C FFT

61440 x 61440 matrix
1 row FFT task
1 column FFT task
120 subtasks/task

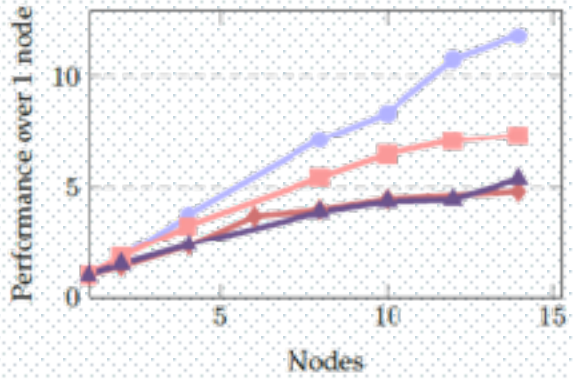
## Block LU Decomposition

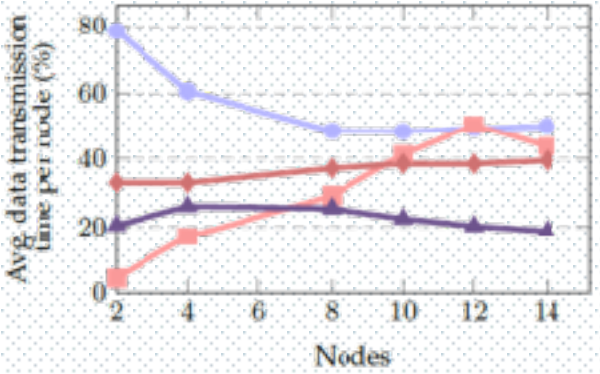
$2^{16}$ x $2^{16}$ matrix
3 tasks per iteration
32 iterations
1 sequential task/iteration
Min/Max/Avg subtasks in a task = 1/961/121.7

## Page Rank

500 million web pages
20 outlinks per page (max)
Web dump stored on NFS
25 iterations
250 subtasks/iteration

# Performance Results

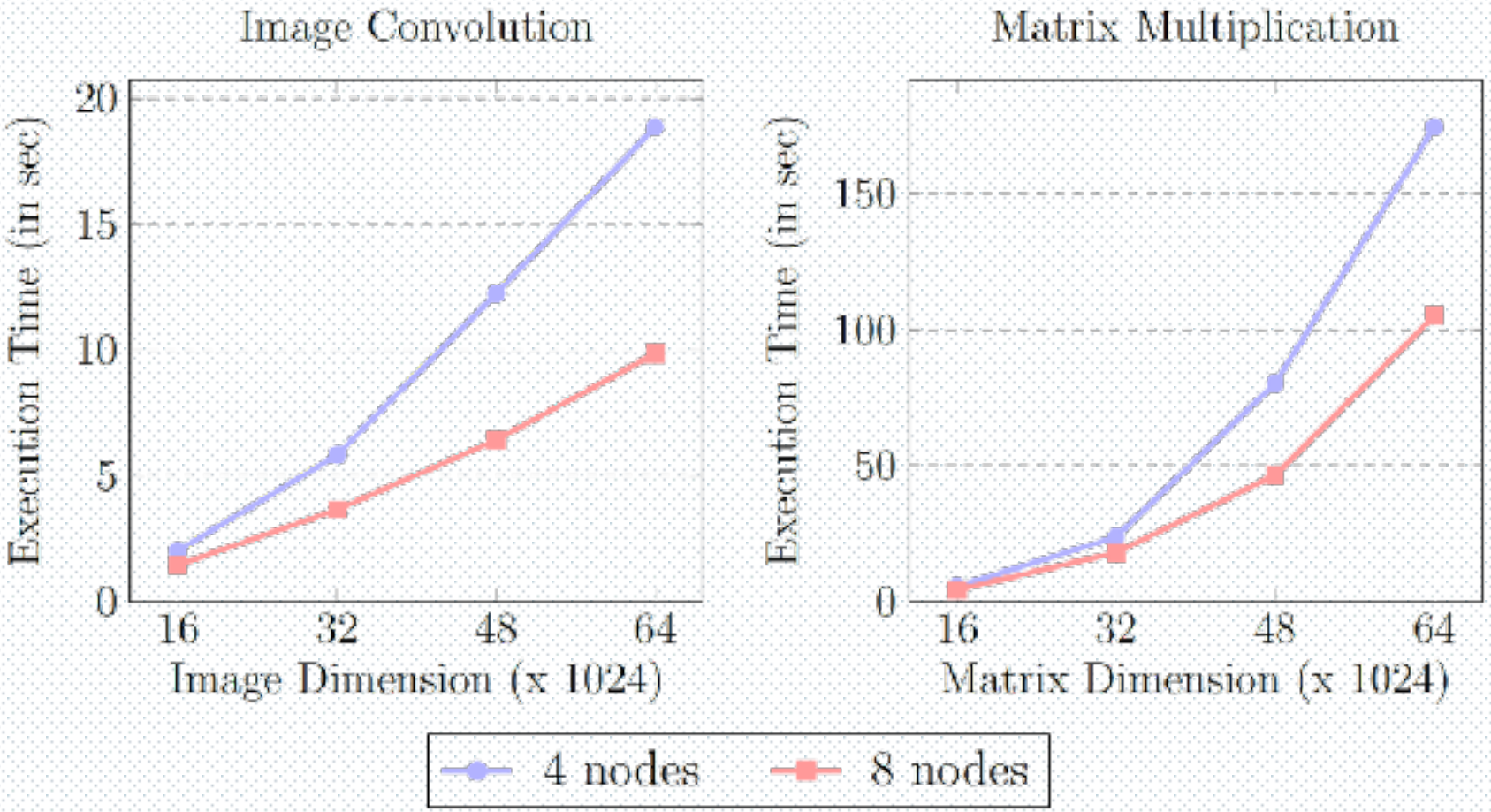



| Nodes | Data transferred (GB) | | | | | | Data transfer events (x 1000) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | 4 | 8 | 10 | 12 | 14 | 2 | 4 | 8 | 10 | 12 | 14 |
| | 6.1 | 9.3 | 10.6 | 10.7 | 11.0 | 11.4 | 1.55 | 3.2 | 4.46 | 4.97 | 5.12 | 5.38 |
| | 17.0 | 61.5 | 140.0 | 186.5 | 224.0 | 248.5 | 0.07 | 0.21 | 0.48 | 0.54 | 0.67 | 0.66 |
| | 40.0 | 79.2 | 142.9 | 172.0 | 198.0 | 222.5 | 7.53 | 15.3 | 25.4 | 30.3 | 32.1 | 36.7 |
| | 24.7 | 45.7 | 50.3 | 49.5 | 53.0 | 51.2 | 1.83 | 3.97 | 11.18 | 10.84 | 10.93 | 14.58 |

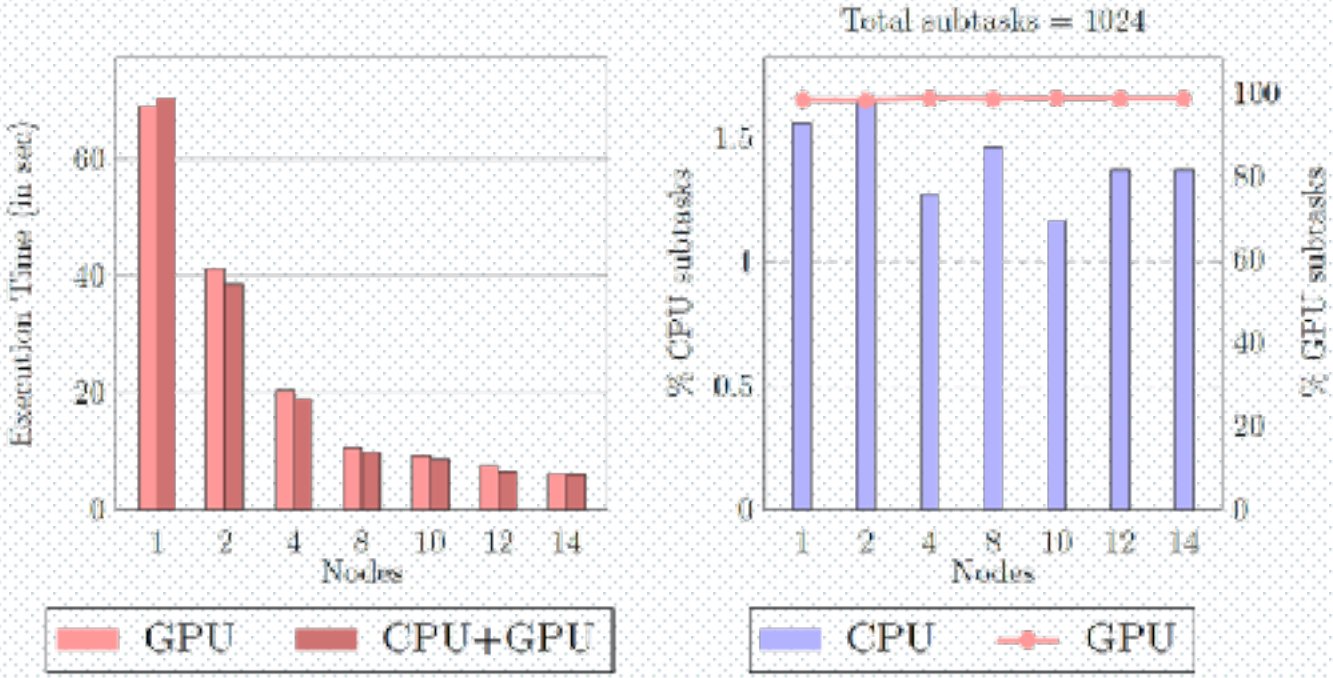
Image Convolution
Matrix Multiplication
LU Decomposition
2D FFT

# Scaling with increasing problem size



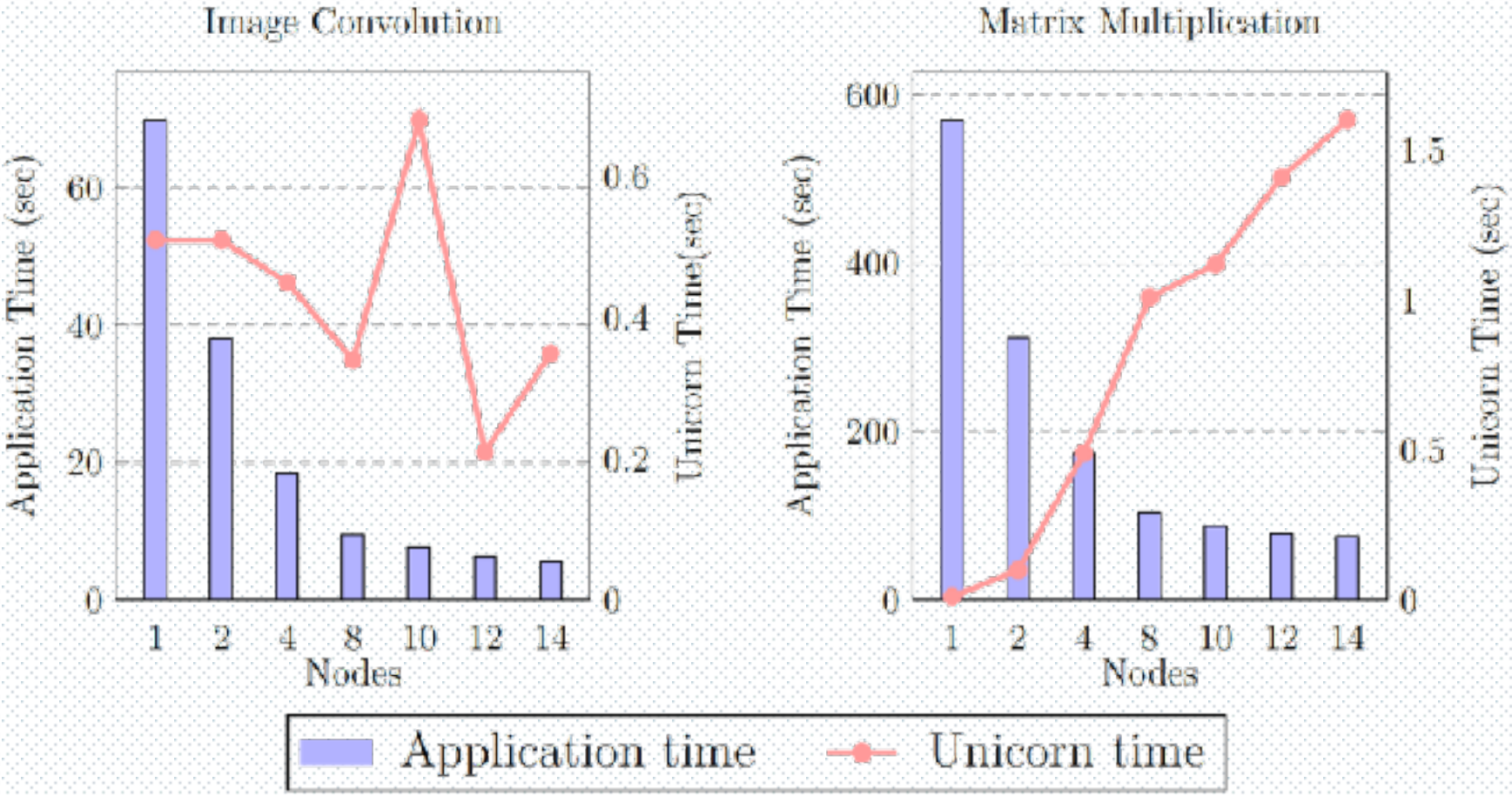
# GPU versus CPU+GPU

## Image Convolution



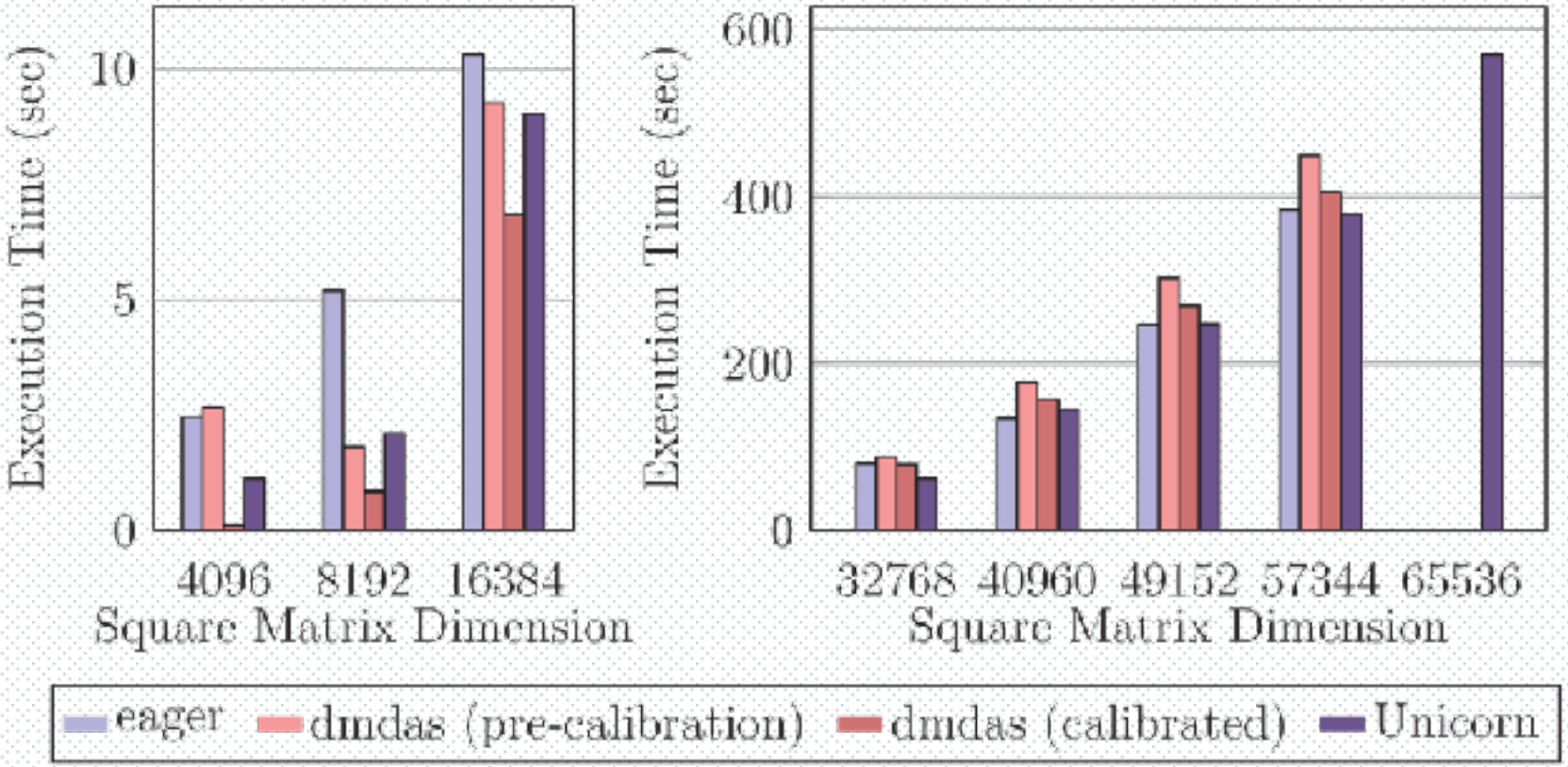

Lower Execution time is better

# Unicorn Time versus Application Time



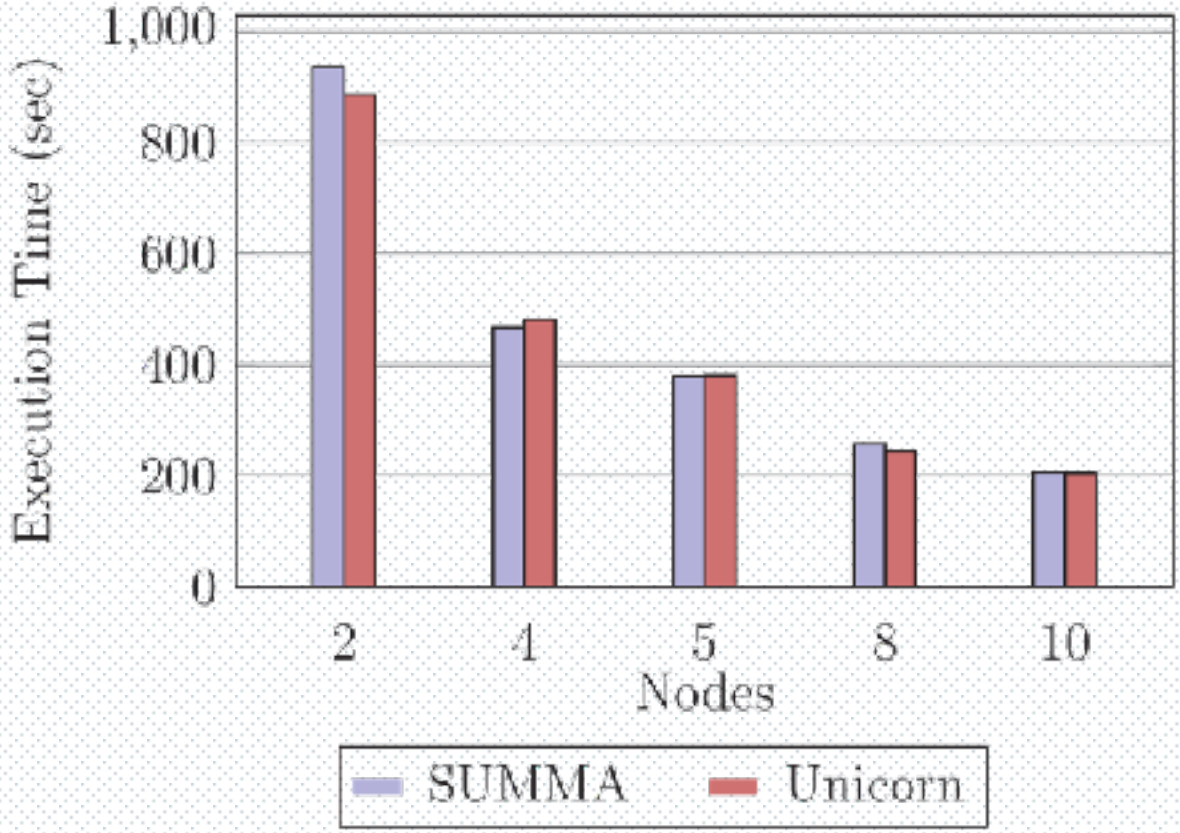
# Unicorn versus StarPU (one node)

## Matrix Multiplication



# Unicorn versus SUMMA

## Matrix Multiplication



# Concluding Remarks

- Unicorn is suitable for
  - Coarse grained tasks decomposable into concurrently executable subtasks
  - Defer synchronization, with lazy conflict resolution
- Unicorn model does not work efficiently with tasks
  - Having non-deterministic memory access pattern
  - Requiring fine-grained/frequent communication
- Unicorn could use
  - Directives and language based constructs
  - Inter-job and IO scheduling
  - Support asynchronous updates
  - Adapt to newer architecture, GPU-aware MPI etc.

For more details, please visit:

http://www.cse.iitd.ac.in/~subodh/unicorn.html

Thank you