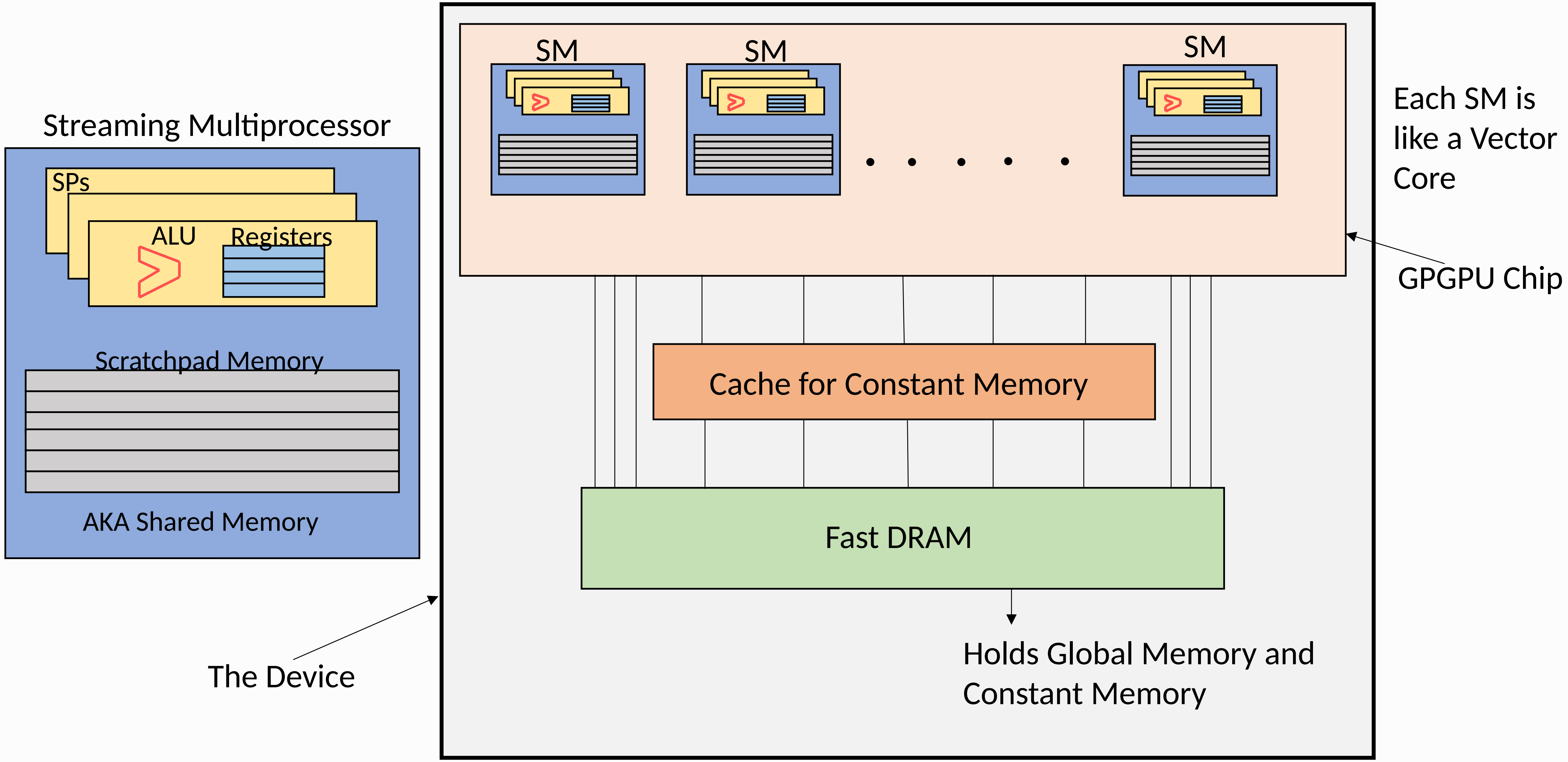
GPUs and General Purposing of GPUs :

- Graphics Processing Unit (GPU)
- Original purpose: high speed rendering(?) i.e. video games, etc
- Optimized for being good at math
- Result: High memory BW and many "cores"
- Brook Streaming Language from Stanford
  - Ian Buck et al paper is worth a read
  - The idea of specialized kernels
    - Running on specialized devices

In this paper, we present Brook for GPUs, a system for general-purpose computation on programmable graphics hardware. Brook extends C to include simple data-parallel constructs, enabling the use of the GPU as a streaming coprocessor.

- NVIDIA and AMD (and Intel's integrated graphics)
- Programming: CUDA, OpenCL, and OpenMP



Schematic GPGPUs

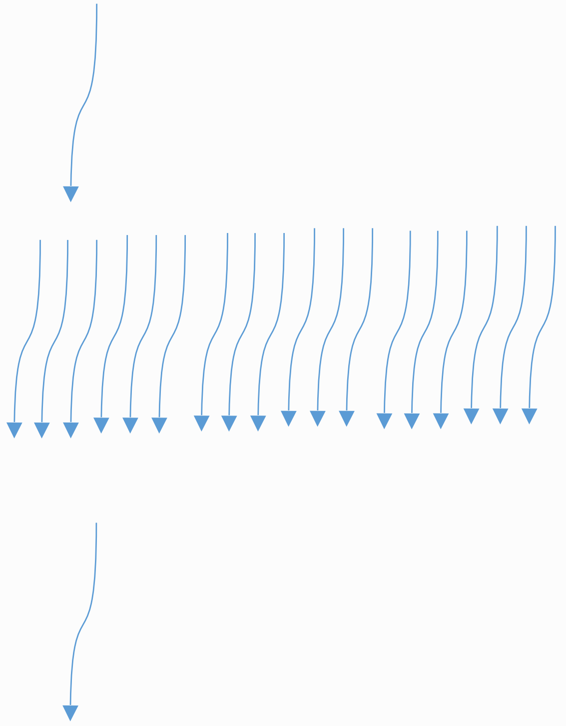
CUDA

- We will present a very simple, over-simplified, overview
- Explicit resource-aware programming
- What you specify
  - Data transfers
  - Data parallel kernel/s, expressed in form of threads
    - Each thread does the action specified by the kernel
  - The total number of threads are grouped into teams called “blocks”
  - Kernel calls specify the number of blocks , and number of threads per block

Programming Model Overview

- Host (serial)
- Launches device functions (parallel)
- Control can return asynchronously
- Memory?
  - Device memory
  - “Unified” memory
- Overlap
  - It is possible to overlap data transfer of one kernel with computation of another

- Serial
- Parallel
- Serial



Simple CUDA Program

```
#include <stdio.h>


void hello() {
  printf("Hello, world!\n");
}

int main() {
  hello();
}
```

```
$ gcc hello.c
$ ./a.out
Hello, world!
```

Simple CUDA Program

```
#include <stdio.h>

__global__
void hello() {
  printf("Hello, world!\n");
}

int main() {
  hello<<<1,1>>>();
}
```

```
$ gcc hello.c
$ ./a.out
Hello, world!
```

```
$ nvcc hello.cu
$ ./a.out
Hello, world!
```

Blocks

- Basic parallel unit
- Threads in a block can assume access to a common shared memory region (scratchpad).
- Analogous to processes
- Blocks grouped into grid
- Asynchronous

```
int main() {
  hello<<<128,1>>>();
}

$ ./a.out
Hello, world!
Hello, world!
…
Hello, world!
```

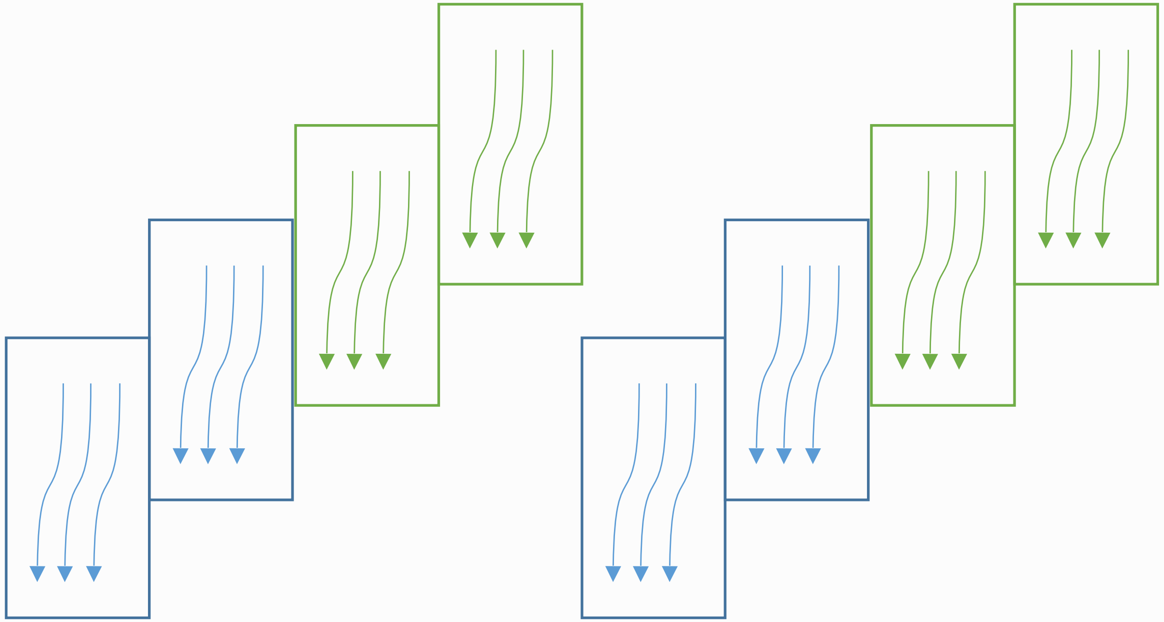
Threads

- Sub-division of a block (shared memory)
- Analogous to OpenMP threads
- Grouped into warps (shared execution)
- Level of synchronization and communication

```
int main() {
  hello<<<1,128>>>();
}

$ a./out
Hello, world!
Hello, world!
…
Hello, world!
```

Warps

- Groupings of threads
- All execute same instruction (SIMT)
- One miss, all miss
- Thread divergence, No-Ops
- Analogous to vector instructions
- Scheduling unit



Combining Blocks, Warps, and Threads

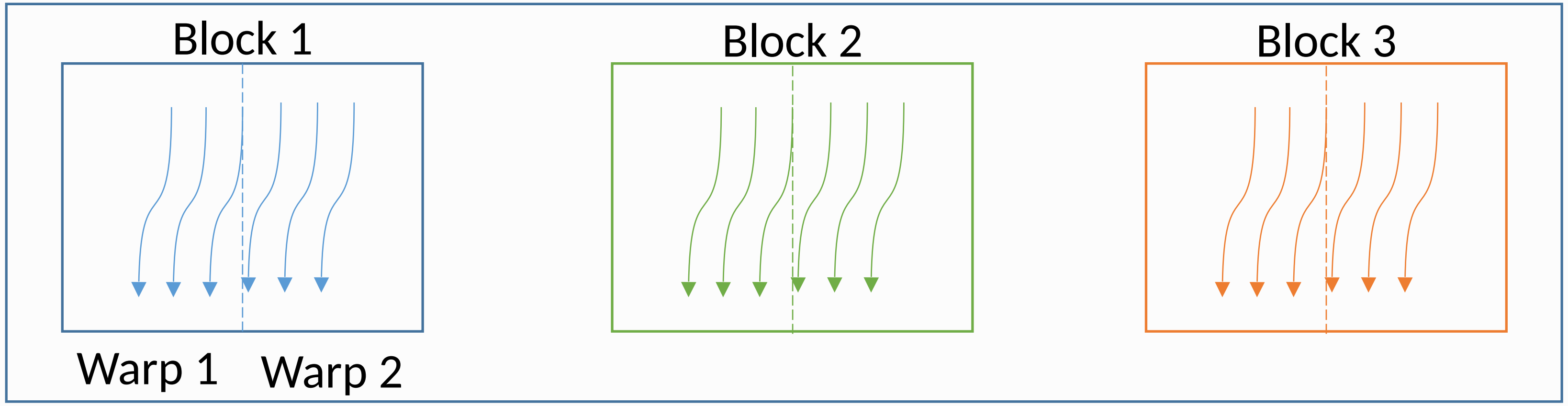
Number of Blocks

Number of Threads per Block

`KernelFunc<<<3,6>>>(…);`

Block Dimension = 6

For this picture, assume a warp has 3 threads.. (in reality, its almost always 32.. It’s a device dependent parameter)



Thread Index	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5
Global Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

If you specify blocksize that’s not a multiple of warpsize, the system will leave some cuda cores in a warp idle)

Illustrative Example

```
__global__
void vecAdd(int* A, int* B, int* C) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  C[i] = A[i] + B[i];
}
…
int main() {
  // Unified memory allocation
  vecAdd<<<VEC_SZ/512,512>>>(A, B, C);
}
```

blockIdx.x is my block's serial number

blockDim.x is the number of threads per block

threadIdx.x is my thread's id in my block

Number of Blocks

Number of Threads per Block

Using CUDA kernels from Chares

- Charm++ is not a compiler.. So it won't write CUDA code for you
    - OpenACC, OpenMP, … will write kernels for you
- So the main question is how can you fire CUDA kernels and manage dependencies
- Of course, you could just use CUDA as it is
    - But: when you fire a kernel, then, you are blocking the processor and not allowing other chares to make progress
- You first need an API/Abstraction to fire kernels asynchronously and get callbacks when they are done
    - This is provided by HAPI (Hybrid API)
    - In addition: allocate/free memory on device, and
    - Support for transferring data from/to device (instead of bringing it to host DRAM)

Following Slides by Jaemin Choi

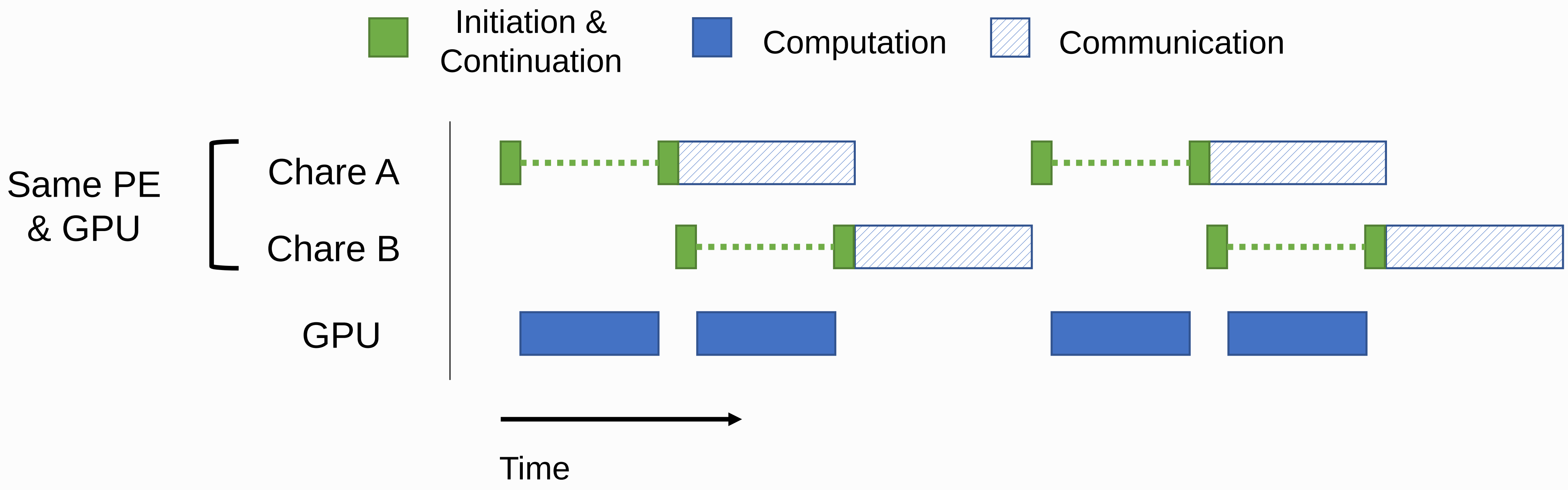
So, to use CUDA kernels in Charm++

- You write your own kernels
- Allocate cuda streams using HAPI calls
- Allocate device memory using HAPI calls
- Fire kernels on specific streams that you wish to use
- Asynchronous Completion support: Insert callbacks into the streams so your chare can be notified of completion using HAPI calls
- Use device-to-device communication using our layer:
  - CkDeviceBuffer and post method(GPU communication API)
  - Channel API

Following Slides by Jaemin Choi

Automatic Computation-Communication Overlap

Exploiting Overlap on GPUs



- Computational work offloaded to the GPU
- Initiation of kernels (+ data transfers) & subsequent continuation on the host CPU (PE)
- Little overlap with naive implementation... Why?

Need for Asynchrony

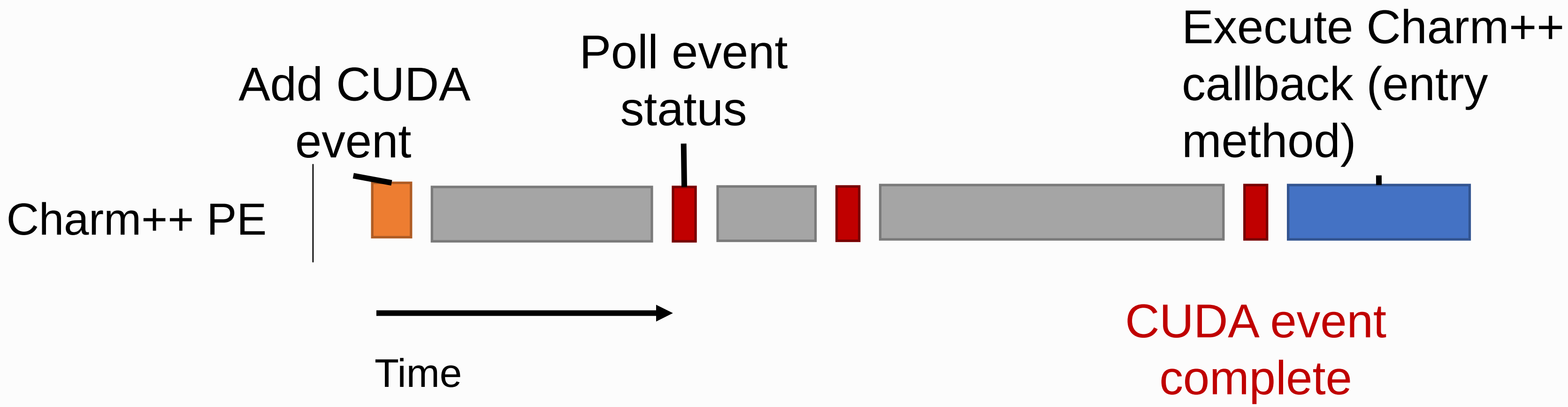
- Using CUDA stream synchronization to wait for kernel completion
  - Slow synchronization performance
  - Prevents host scheduler from doing anything else
  - Limits amount of attainable overlap
- Other asynchronous completion notification mechanisms from CUDA?
  - CUDA Callback: CUDA-generated thread collides with Charm++ runtime threads, does not have access to Charm++ functionalities and data structures
  - CUDA Events: How should the user poll the status of the events?
- Need support from the Charm++ runtime system

HAPI Callback: Asynchronous Completion Notification

- Provided in the **Hybrid API (HAPI)** module of Charm++
- hapiAddCallback(cudaStream_t stream, CkCallback cb)
- Tell Charm++ runtime to execute Charm++ callback (entry method) when previous operations in the CUDA stream complete
- Two mechanisms based on CUDA Callback & Events

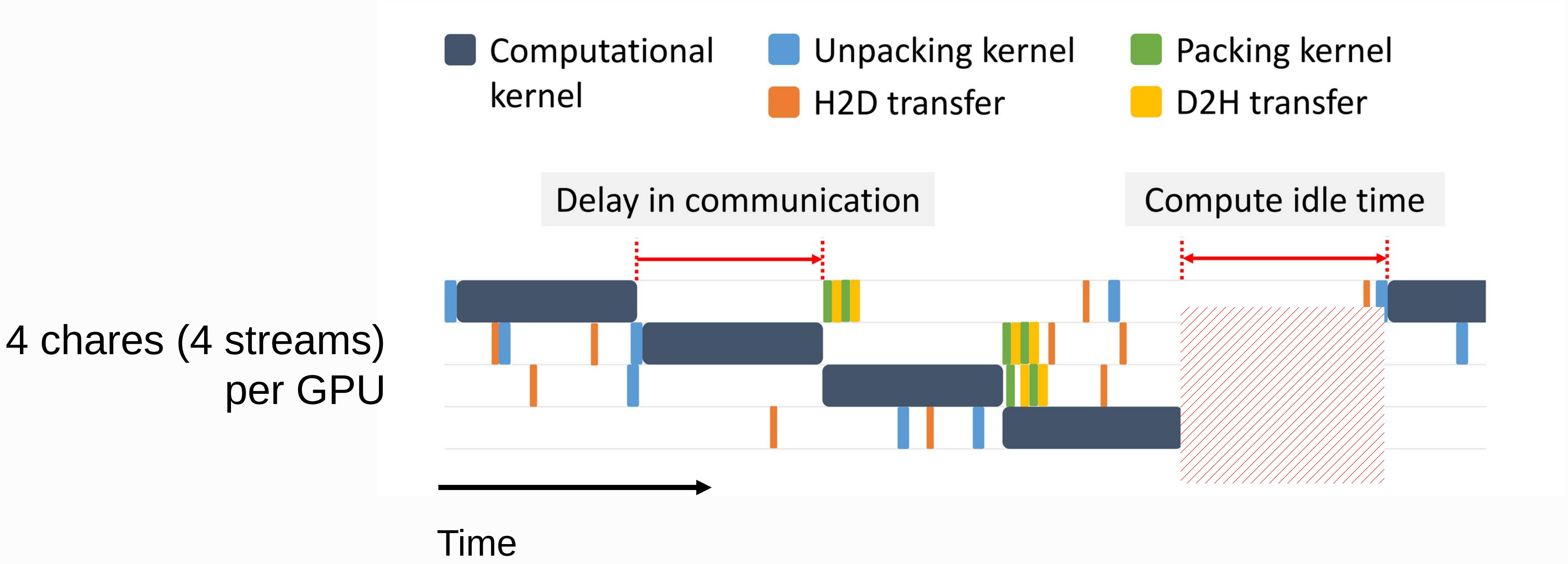
```
void hapiAddCallback(cudaStream_t stream, CkCallback* callback);

void hapiCreateStreams();
cudaStream_t hapiGetStream();

void* hapiPoolMalloc(int size);
void hapiPoolFree(void* ptr);

hapiCheck(code);
```

CUDA Event-Based HAPI Callback



- CUDA Event-based
  - Create and add CUDA event
  - Scheduler polls for status of CUDA event (poll frequency configurable)
  - When CUDA event completes, execute Charm++ callback (entry method)
  - Faster performance vs. CUDA Callback-based, used as the default

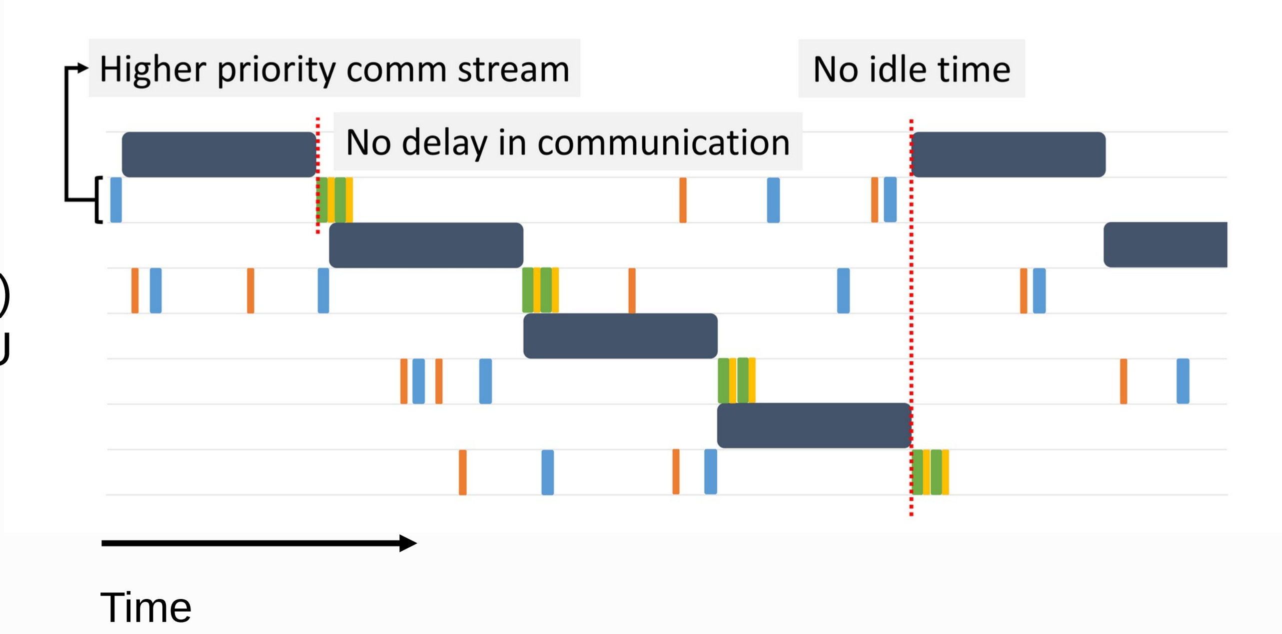
Need for Communication Priority



- With overdecomposition, communication and related operations (e.g., packing/unpacking kernels, host-device transfers) may be delayed
- Need to prioritize communication-related operations

CUDA Streams with Priority

4 chares (8 streams) per GPU



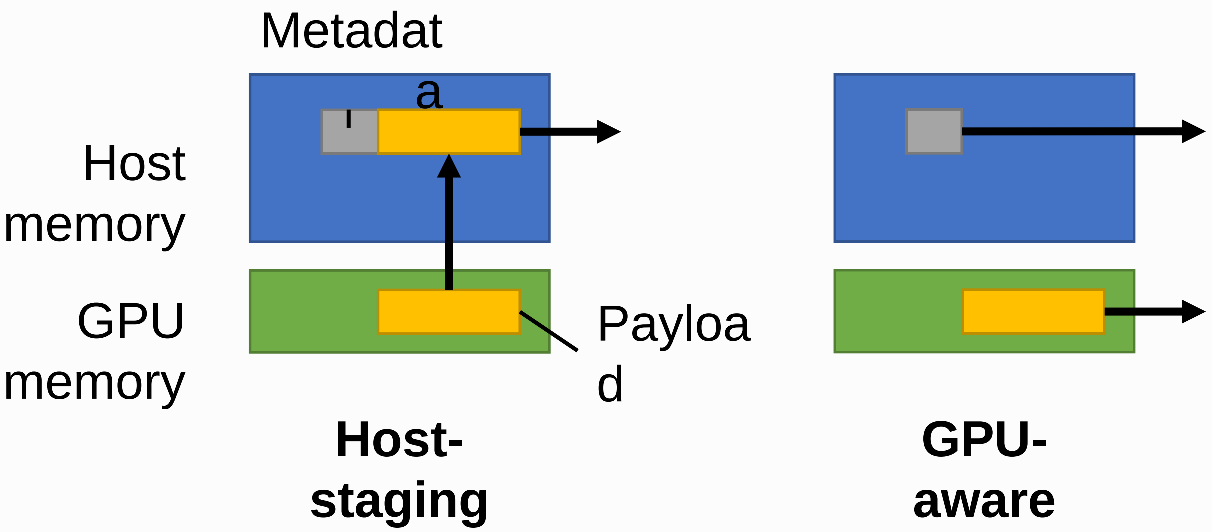
- Use a separate high priority CUDA stream for communication-related operations
- Reduces delay in initiating asynchronous communication
- Reduces idle time & increases compute utilization

Streams scheme

- If you use the previous scheme of 3 streams per chare and if you have a large number of chares per process, you may cauae overheads due to multiplexing of streams on system resources
- Consider the schemes of previous slides as suggestions for best practices, and vary the number of streams accordingly
  - Experiment with them

GPU-Aware Communication

GPU-Aware Message-Driven Execution



- Charm++ messages are constructed in host memory
    - Metadata + User payload
    - If user payload is in GPU memory, it needs to be moved to host memory beforehand
- Schedulers run on host CPUs
- Separate metadata and GPU payload!
    - Metadata needed for message-driven execution is sent without the payload
    - GPU payload is sent separately

GPU Messaging API

Sender Chare

```
void Chare::foo() {
  // Invoke entry method with GPU payload
  chare_proxy[peer].bar(8,
CkDeviceBuffer(my_buf));
}
```

1 Send metadata message 2 Send GPU buffer

3 Metadata message arrival

Receiver Chare

```
// Post entry method
void Chare::bar(int& count, double*& buf) {
  // Specify destination GPU buffer
  buf = recv_buf;
}
```

4 Post receive for GPU buffer

5 GPU buffer arrival

```
// Regular entry method
void Chare::bar(int count, double* buf) {
  // GPU buffer has been received
  some_kernel<<<...>>>(count, buf);
}
```

- Want to send buffer in GPU memory
- Wrap inside CkDeviceBuffer to notify runtime system that this is a GPU buffer
- Runtime sends message with metadata, and separately sends source GPU buffer (both with UCX but different code paths)
- On host-side message arrival, post entry method is first executed to determine destination GPU buffer
- Receive for incoming GPU buffer is posted

Channel API

Sender Chare

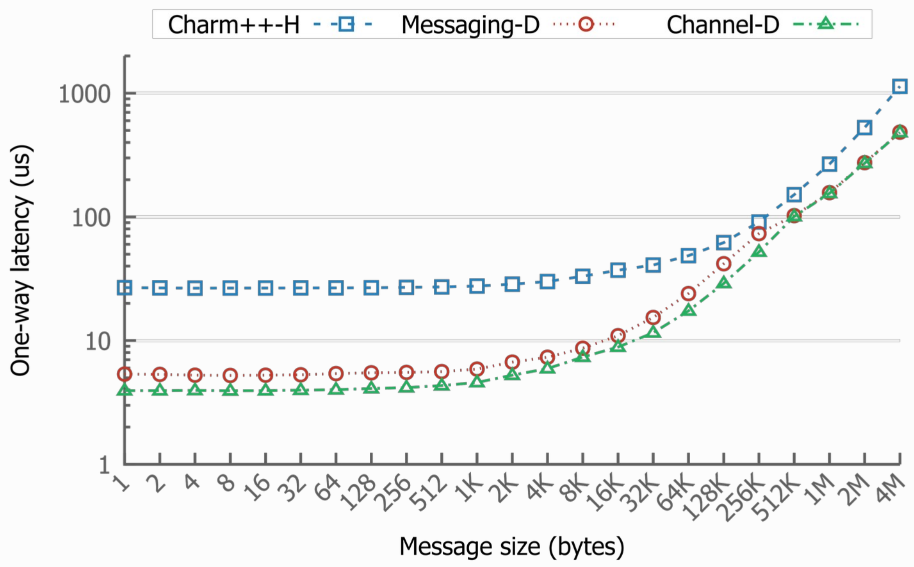
```
void Chare::foo() {
  channel.send(buf, size,
CkCallbackResumeThread());
}
```

Receiver Chare

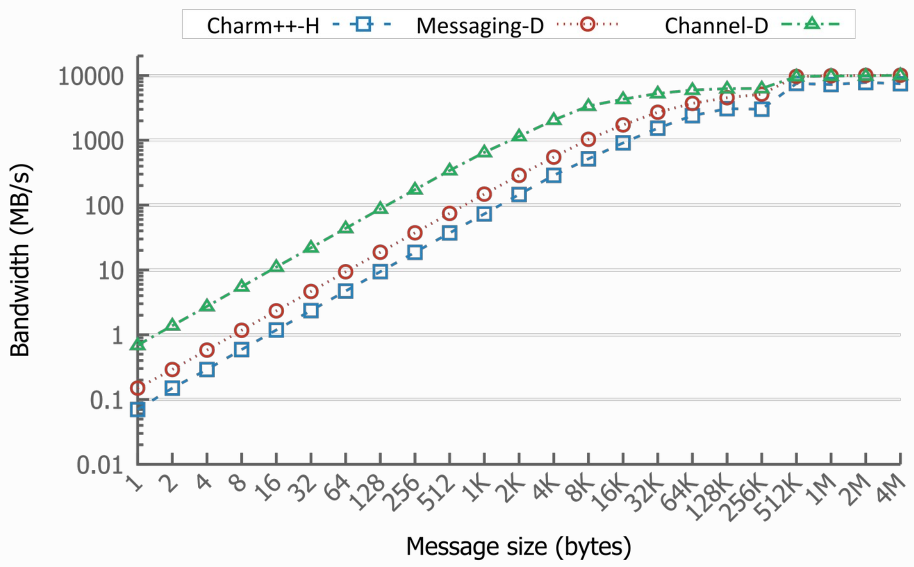
```
void Chare::bar() {
  channel.recv(buf, size,
CkCallbackResumeThread())
}
```

- GPU Messaging API suffers from additional latency due to metadata message & delayed receive
- A **channel** is established between a pair of chares
- Use two-sided send & receive semantics on channel
- Instead of transferring execution flow, **only transfer data**
- Charm++ callbacks can be passed for asynchronous completion notification
- Improved performance with direct interface to UCX

Pingpong Performance



Latenc y



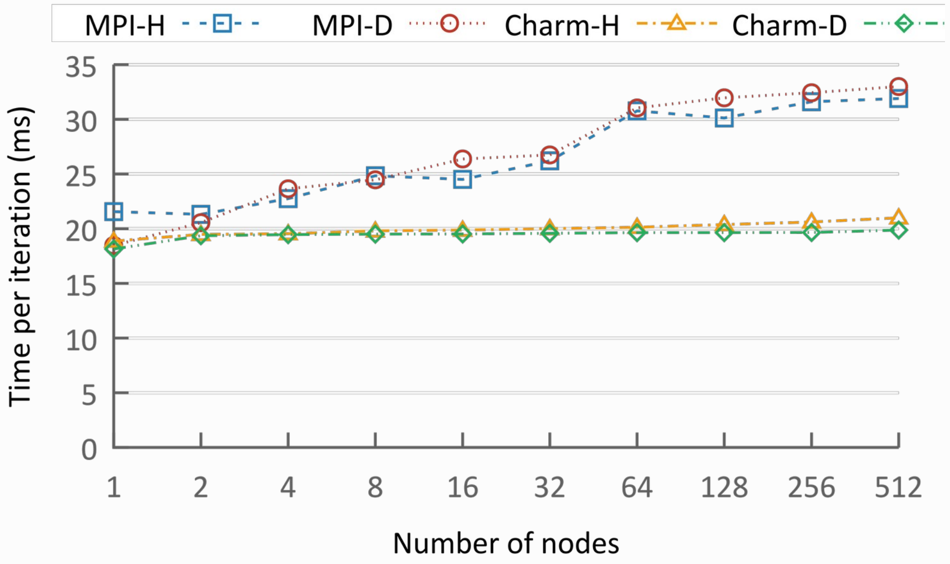
Bandwid th

- Charm++ pingpong benchmark on 2 nodes of OLCF Summit (GPU source/destination buffers)
- Latency & bandwidth substantially improve with GPU-aware communication
- Results with AMPI, Charm4py and Jacobi3D proxy application in thesis

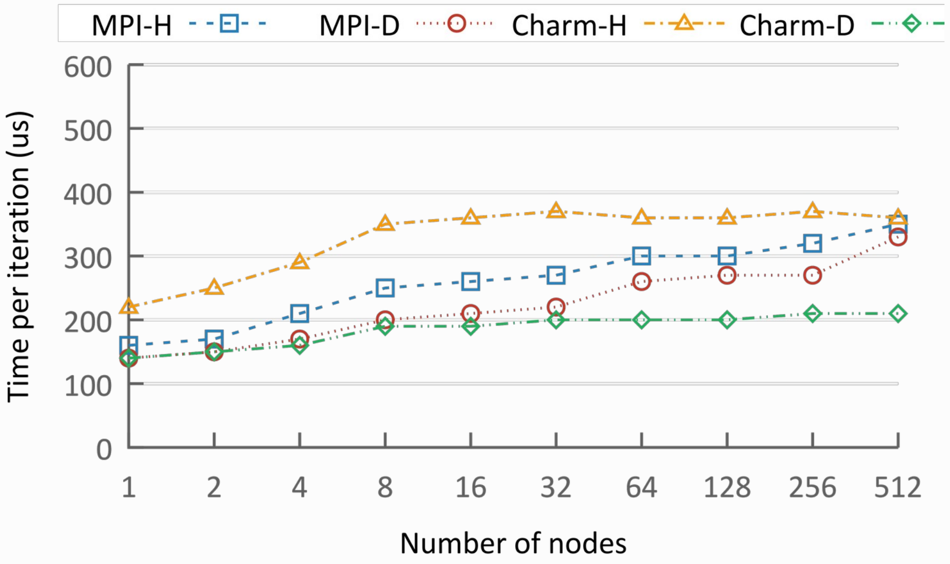
Combining Overlap & GPU-Aware Communication

- Overdecomposition-driven **automatic computation-communication overlap** on GPUs
  - Effective hiding of communication latency especially with weak scaling
  - Limitations with strong scaling due to overheads associated with finer granularity
- Integrating **GPU-aware communication** into message-driven execution
  - Improves raw communication performance
  - Less effective with large messages, due to switching to host-staging
- **Combine overlap & GPU-aware communication** for performance synergy
  - Hide as much communication as possible with automatic overlap
  - Reduce exposed communication costs with GPU-aware communication
  - **Effective in both weak and strong scaling**

Jacobi3D: Weak Scaling



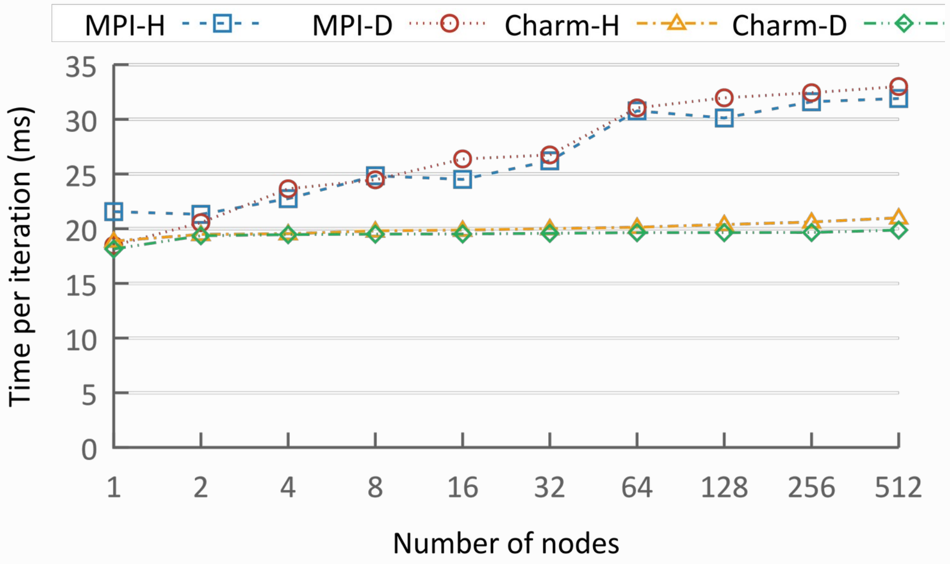
Big: 1,536 x 1,536 x 1,536 per node



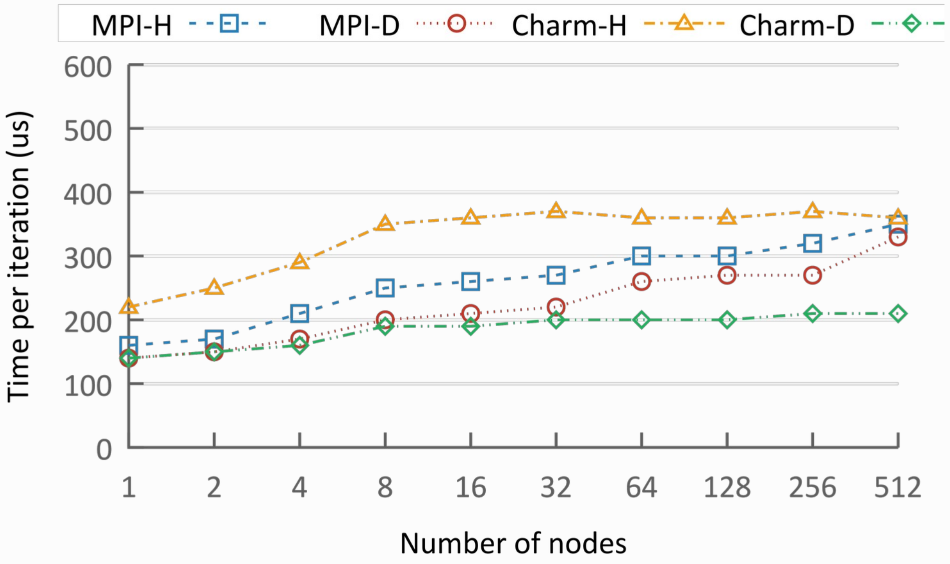
Small: 192 x 192 x 192 per node

- Big: Computation-communication overlap provides almost perfect weak scaling
  - Best performing ODFs: ODF-4 for Charm-H, ODF-2 for Charm-D
  - Small room for improvement with GPU-aware communication (Charm-D vs. Charm-H)
  - CUDA-aware MPI doesn't improve performance from 4 nodes due to pipelined host-staging protocol

Jacobi3D: Weak Scaling



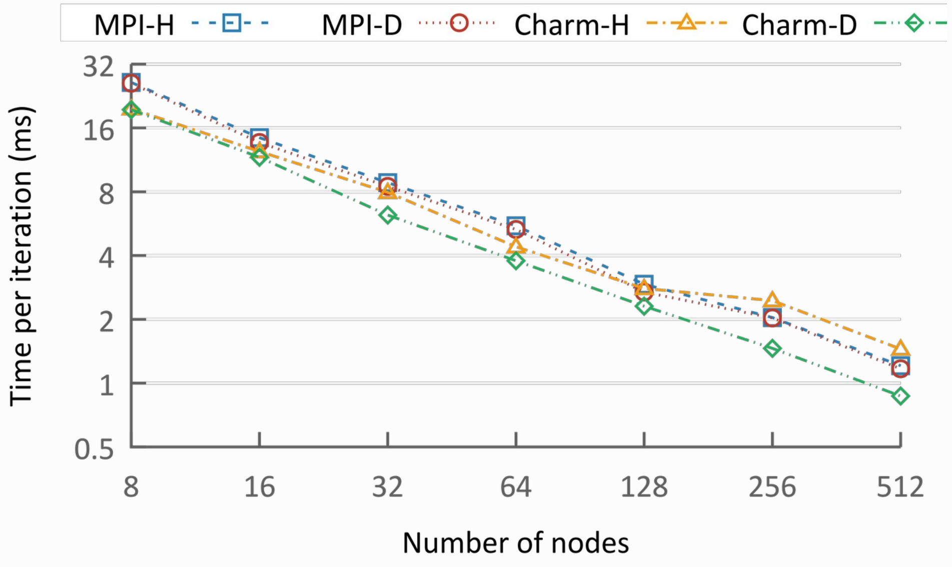
Big: 1,536 x 1,536 x 1,536 per node



Small: 192 x 192 x 192 per node

- Small: Performance gains from GPU-aware communication
  - Overdecomposition does not improve performance (no automatic overlap)
  - Due to fine-grained overheads with small problem size
  - Issue with CUDA-aware IBM Spectrum MPI performance at large scale

Jacobi3D: Strong Scaling



Global grid: 3,072 x 3,072 x 3,072

- **Combination of overlap & GPU-aware communication** provides the best performance and scalability
  - Best performing ODF for Charm++ decreases with scale, due to finer granularity
  - Charm-H: ODF-4 → ODF-2 → ODF-1, Charm-D: ODF-2