Improving the Performance of Charm++ Applications on GPU Systems

Jaemin Choi

Charm++ Workshop 2021
Oct 19, 2021
Charm++ on GPU Systems

- Chares can offload computational kernels to the GPU (e.g., CUDA)
- Need to maximize asynchrony to prevent chares from not yielding to other chares
  - CUDA streams
  - Charm++ Hybrid API (HAPI) for asynchronous completion notification
Computation-Communication Overlap
• Minimize synchronization for overlap
• Prioritize communication using CUDA stream priorities or coordination with CUDA events
• More details can be found in this [ESPM2’20 paper](https://example.com/)

Automatic Computation-Communication Overlap
Automatic Computation-Communication Overlap

- **MiniMD**: proxy app for molecular dynamics
  - Charm++ (decomposition, communication) and **Kokkos** (GPU kernels, host-device transfers)
  - Beats CUDA-aware MPI even without GPU-aware communication due to overlap
  - Limitation: overlap with overdecomposition does not improve performance at end of strong scaling
  - [https://github.com/minitu/miniMD/tree/charm/kokkos](https://github.com/minitu/miniMD/tree/charm/kokkos)
GPU-aware Communication
• **Productivity**: users can provide GPU buffers directly to the communication APIs

• **Performance**: direct transfers between GPUs (bypass host memory)

• Underlying technology: CUDA IPC, GPUDirect

• E.g., CUDA-aware MPI
Also, Adaptive MPI and Charm4py

How can we support all of our parallel programming models?

How do we retain message-driven execution?

Our approach: build on GPU support in UCX
  • Caveat: UCX tagged API caters to MPI send/recv semantics
Sender’s data is packed together with metadata (e.g., information about target chare & method)

• Message asynchronously sent to receiver

• Sits in receiver’s message queue until it is picked up by scheduler

```cpp
void Sender::foo() {
    // Send host buffer to a peer chare
    chare_proxy[peer].bar(1024, my_buf);
}
```

```cpp
void Receiver::bar(int count, double* buf) {
    // Scheduler calls this method after picking
    // up message from its message queue
    for (i = 0; i < count; i++) {
        f(buf[i]);
    }
}
GPU Messaging API

Sender Chare

```c++
void Sender::foo() {
    // Send GPU buffer to a peer chare
    char_proxy[peer].bar(1024, CkDeviceBuffer(my_buf));
}
```

1. Send host-side message
2. Send GPU buffer
   
Receiver Chare

```c++
// Post entry method: First called by the runtime
// Before receiving incoming GPU buffer
void Receiver::bar(int& count, double*& buf) {
    // Specify destination GPU buffer
    buf = recv_buf;
}
```

3. Post receive for GPU buffer

```
// Regular entry method: Called by the runtime
// once the GPU buffer has arrived
void Receiver::bar(int count, double* buf) {
    // Has access to received GPU buffer
    some_kernel<<<...>>>(count, buf);
}
```

- **Documentation**
- Builds on Zero Copy API to preserve message-driven execution
- Still need metadata on host memory
- **CkDeviceBuffer**
  - Contains information about GPU src/dst buffers
  - Sent to receiver together with other metadata
- Receiver posts separate receives for GPU data once host-side message arrives
Channel API

**Channels** can be created between a pair of chares (not constrained to GPU data)

- Exchange only data with explicit sends & receives (similar to MPI)
- Does not transfer control flow
- Reduces overhead from receive for GPU data being delayed
- Will be part of release 7.1
  - [https://github.com/UIUC-PPL/charm/pull/3484](https://github.com/UIUC-PPL/charm/pull/3484)

---

**Sender Chare**

```cpp
void Sender::foo() {
  // Send GPU buffer to a peer chare
  channel.send(data, size, &future);
  CkWaitFuture(future);
}
```

**Receiver Chare**

```cpp
void Receiver::bar() {
  // Receive GPU buffer
  channel.recv(data, size, &future);
  CkWaitFuture(future);
}
```

* Can also use Charm++ callbacks instead of futures
• Substantial improvements in latency & bandwidth
• TODO: Combine computation-communication overlap & GPU-aware communication
• More details in AsHES’21 paper
CharminG: A GPU-resident Runtime System
Motivation

• Computation is moving to GPU

• Program flow & communication are still driven by CPU
  • Overheads from interactions (e.g., synchronization) & data transfers between CPU and GPU
  • How do we utilize the upcoming direct GPU-NIC connections (e.g., OLCF Frontier) more efficiently?

• Can we improve performance by moving the entire execution to the GPU?

• Related work: Juggler [M. E. Belviranli, PPoPP ‘18]
  • Per-SM task scheduler
  • Task dependencies are resolved on the fly and entirely on the GPU
  • Limited to a single node
  • Not modularized, runtime system is embedded within the application
- Develop fully GPU-resident runtime system
- Using Charm++ principles
  - Overdecomposition
  - Asynchronous message-driven execution
  - Migratability
- Enable adaptive runtime features without interactions with host CPU
- Implemented working prototype
System Design

[ Current Prototype ]

[ Future ]
• Persistent kernel, single thread per GPU
• PE 0 (thread 0 on GPU 0) executes user’s main function
  • Creates chare objects and initiates program flow
    (invoke entry methods)
• All PEs keep receiving messages and executing entry methods until termination
  • New kernels launched using CUDA dynamic parallelism to perform user’s data parallel tasks
Message Queue

- Implemented as MPSC ring buffer with wrap-around to utilize fixed NVSHMEM allocation
  - Also working on SPSC-based implementation ($O(N^2)$ memory usage in exchange for less remote atomic operations)
- Producers (remote PEs)
  - Try to acquire space in the consumer's message queue using NVSHMEM atomics
  - Once acquired, transfer message using NVSHMEM one-sided put
- Consumer (local PE, scheduler)
  - Consumes messages starting from the lowest address
__global__ void jacobi_kernel(double* temp, double* new_temp,
    int block_width, int block_height) {
    int i = blockDim.x * blockIdx.x + threadIdx.x + 1;
    int j = blockDim.y * blockIdx.y + threadIdx.y + 1;
    if (i < block_height + 1 && j < block_width + 1) {
        new_temp[IDX(i,j)] = (temp[IDX(i,j)] + temp[IDX(i,j-1)]
            + temp[IDX(i,j+1)] + temp[IDX(i-1,j)] + temp[IDX(i+1,j)]) * 0.2;
    }
}

// Block is a chare object
struct Block : charming::chare {
    __device__ Block() {}    
    __device__ void send_boundaries();    
    __device__ void recv_ghost(void* arg);    
    __device__ void update();    
};

__device__ void Block::send_boundaries() {
    block_proxy->invoke(left_neighbor, 1, left_boundary, ghost_size);
    ...
}

__device__ void Block::recv_ghost(void* arg) {
    int dir = *(int*)arg;
    double* ghost = (double*)((int*)arg + 1);
    switch (dir) { ... } // Unpack if necessary
    if (++recv_count == neighbor_count) update();
}

__device__ void Block::update() {
    jacobi_kernel<<<grid_dim, block_dim>>>(...);
    cudaDeviceSynchronize();
    if (++iter == n_iters) charming::exit();
    else send_boundaries();
}
Jacobi2D Preliminary Performance

- Comparison against non-blocking CUDA-aware MPI based implementation
- Up to 64 nodes (256 NVIDIA V100 GPUs) on LLNL Lassen
- Much room for performance improvement
• Prototype working on NVIDIA GPUs
  • C++ templates to support user-defined chare types
  • NVSHMEM for device-initiated GPU communication
  • CUDA dynamic parallelism to launch new kernels

• Future work
  • Analyze and improve performance (communication, scheduler, launching of user kernels)
  • Explore computation-communication overlap with overdecomposition
• GPU features in Charm++
  • Asynchronous execution & completion notification using CUDA streams & HAPI
  • GPU-aware communication: GPU Messaging API, Channel API
• CharminG: GPU-resident runtime system
Thank You!