Recent Advances in Heterogeneous Computing using Charm++

Jaemin Choi, Michael Robson

Parallel Programming Laboratory
University of Illinois Urbana-Champaign

April 12, 2018

Heterogeneous Computing

- Computing with different types of devices
- In this talk: using GPUs to boost performance
- **GPU**
  - Throughput oriented
  - Data parallel (SIMD)
  - Many simple, low frequency cores
  - Teraflops of computing power
  - Separate memory (GDDR or HBM)
  - Data transfer overhead
- Now a critical factor of performance

Figure: NVIDIA Tesla V100[1]

[1] Image source: `https://www.nvidia.com/en-us/data-center/tesla-v100`

How to Utilize GPUs in Charm++

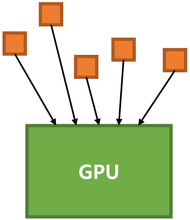
1. Use CUDA directly
   - Let each chare offload (small) kernels
   - Or manually aggregate data at a synchronization point and offload one big kernel
2. Use **GPU Manager** library of Charm++
   - Why? What good is it?

Problems with Using GPUs in Charm++

- Due to **overdecomposition** and **asynchrony**

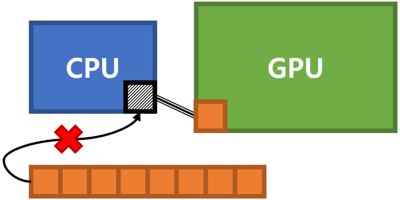
1. Granularity of work
2. Blocking offload API
3. Responsiveness

Problem 1: Granularity of Work



- Each chare is fine-grained
- Contain little data and work $\rightarrow$ small kernels
- Kernels should be able to execute concurrently
- Or need to aggregate kernels

Problem 2: Blocking Offload API



- Commonly used CUDA API are blocking
  - E.g. `cudaDeviceSynchronize()`, `cudaStreamSynchronize()`
- PEs are implemented as persistent threads on CPU cores
- Blocking call thus prevents another chare from executing
- Another problem: number of concurrent kernels limited to the number of PEs
- Offload API should be **non-blocking** for Charm++

Problem 3: Responsiveness

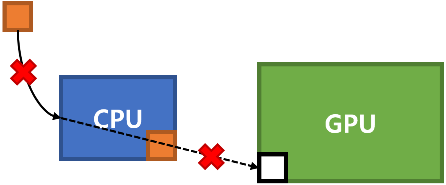


Figure: Slow initiation

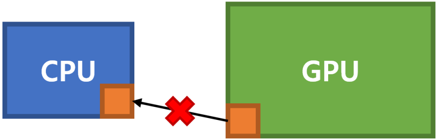


Figure: Slow response

1. Slow initiation
   - Method offloading work must wait if target PE is busy (even if the GPU is free)
2. Slow response
   - Handling completed GPU work delayed if target PE is busy

Current GPU Manager

- Addresses Problem 2 (blocking offload API)
- User constructs and submits a `WorkRequest` object, specifying
  - Data buffers and directions of transfer
  - Kernel to be executed and its specifications (e.g. grid size, block size)
- Runtime tracks `WorkRequests`, overlapping data transfers with kernel execution
  - But does NOT overlap multiple kernel executions
  - Because only one CUDA stream is used for kernels
- Execution continues without blocking after `WorkRequest` submission
- 3 CUDA streams used internally: Data-in, Kernel, Data-out
- Problems
  - Only one CUDA stream for all kernels
  - Unnecessarily complex API

New GPU Manager: Release 6.9.0

- Partially addresses Problem 1 (granularity of work)
  - Allows kernels to execute in separate CUDA streams
  - Runtime support for kernel aggregation is ongoing research
- Non-blocking feature implemented using CUDA events
- Much simpler API (almost identical to CUDA API)
  - **Hybrid API**: `hapi` prefix instead of `cuda`
  - `hapiAddCallback()`: invoke provided Charm++ callback function when data transfer/kernel execution completes, replaces `cudaStreamSynchronize()`
- Ongoing research to address Problem 3 (responsiveness)

Non-blocking Implementation of Offloading

- Use CUDA events to detect completion of GPU work
- Each PE maintains a queue of events
- Queue is checked in the scheduler before choosing what to execute next
- Charm++ callback invoked on completion to continue program flow
- Impractical for the user to implement
    - Unclear where in the program flow the queue should be checked
    - Unclear how frequent the checking should occur
- Alternative: CUDA callback, but single callback thread becomes a bottleneck

Matmul Code Comparison: Current GPU Manager

```c
void run_MATMUL_KERNEL(workRequest *wr, cudaStream_t kernel_stream, void **devBuffers) {
    printf("MATMUL KERNEL");
    matrixMul<<< wr->dimGrid, wr->dimBlock, wr->smemSize, kernel_stream >>>
      ((ElementType *) devBuffers[wr->bufferInfo[C_INDEX].bufferID],
       (ElementType *) devBuffers[wr->bufferInfo[A_INDEX].bufferID],
       (ElementType *) devBuffers[wr->bufferInfo[B_INDEX].bufferID],
       *((int *) wr->userData), *((int *) wr->userData));
}

void cudaMatMul(ElementType *h_A, ElementType *h_B, ElementType *h_C,
                void *cb, int matrixSize) {
  int size = matrixSize * matrixSize * sizeof(ElementType);
  dataInfo *AInfo, *BInfo, *CInfo;

  workRequest matmul;
  dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
  matmul.dimGrid = dim3(ceil((float)matrixSize / threads.x),
                        ceil((float)matrixSize / threads.y));
  matmul.dimBlock = threads;
  matmul.smemSize = 0;
  matmul.nBuffers = 3;
  matmul.bufferInfo = (dataInfo *) malloc(matmul.nBuffers * sizeof(dataInfo));

  AInfo = &(matmul.bufferInfo[0]);
  AInfo->transferToDevice = YES;
  AInfo->transferFromDevice = NO;
  AInfo->freeBuffer = YES;
  AInfo->hostBuffer = h_A;
  AInfo->size = size;

  BInfo = &(matmul.bufferInfo[1]);
  BInfo->transferToDevice = YES;
  BInfo->transferFromDevice = NO;
  BInfo->freeBuffer = YES;
  BInfo->hostBuffer = h_B;
  BInfo->size = size;

  CInfo = &(matmul.bufferInfo[2]);
  CInfo->transferToDevice = NO;
  CInfo->transferFromDevice = YES;
  CInfo->freeBuffer = YES;
  CInfo->hostBuffer = h_C;
  CInfo->size = size;

  matmul.callbackFn = cb;
  matmul.traceName = "matmul";
  matmul.runKernel = run_MATMUL_KERNEL;

  matmul.userData = malloc(sizeof(int));
  memcpy(matmul.userData, &matrixSize, sizeof(int));

  enqueue(&matmul);
}
```

Matmul Code Comparison: CUDA, New GPU Manager

```
void cudaMatMul(ElementType *h_A, ElementType *h_B, ElementType *h_C,
                ElementType *d_A, ElementType *d_B, ElementType *d_C,
                cudaStream_t stream, int matrixSize) {
  int size = matrixSize * matrixSize * sizeof(ElementType);
  dim3 block(BLOCK_SIZE, BLOCK_SIZE);
  dim3 grid(ceil((float)matrixSize / block.x),
            ceil((float)matrixSize / block.y));

  cudaMemcpyAsync(d_A, h_A, size, cudaMemcpyHostToDevice, stream);
  cudaMemcpyAsync(d_B, h_B, size, cudaMemcpyHostToDevice, stream);

  matrixMul<<<grid, block, 0, stream>>>(d_C, d_A, d_B, matrixSize, matrixSize);

  cudaMemcpyAsync(h_C, d_C, size, cudaMemcpyDeviceToHost, stream);

  cudaStreamSynchronize(stream);
}
```

Figure: CUDA

```
void cudaMatMul(ElementType *h_A, ElementType *h_B, ElementType *h_C,
                ElementType *d_A, ElementType *d_B, ElementType *d_C,
                void *cb, int matrixSize) {
  int size = matrixSize * matrixSize * sizeof(ElementType);
  dim3 block(BLOCK_SIZE, BLOCK_SIZE);
  dim3 grid(ceil((float)matrixSize / block.x),
            ceil((float)matrixSize / block.y));

  cudaStream_t stream = hapiGetStream();

  hapiCheck(hapiMemcpyAsync(d_A, h_A, size, cudaMemcpyHostToDevice, stream));
  hapiCheck(hapiMemcpyAsync(d_B, h_B, size, cudaMemcpyHostToDevice, stream));

  matrixMul<<<grid, block, 0, stream>>>(d_C, d_A, d_B, matrixSize, matrixSize);
  hapiCheck(cudaPeekAtLastError());

  hapiCheck(hapiMemcpyAsync(h_C, d_C, size, cudaMemcpyDeviceToHost, stream));

  hapiAddCallback(stream, cb);
}
```

Figure: New API

Performance Evaluation: Test Environment

- Single compute node of OLCF Titan
- Up to 8 cores of AMD Opteron 6274 CPU
- 32GB DDR3 memory
- NVIDIA Tesla K20X GPU

Performance Evaluation: `busywait`

- Benchmark designed to validate new GPU Manager
- Tasks (kernels on GPU) busywait both on CPU and GPU
- Vary how much work out of total is offloaded, and how long they take
- 3 configurations of task duration:
  - CPU 1 ms, GPU 10 ms
  - CPU 10 ms, GPU 1 ms
  - CPU 10 ms, GPU 10 ms
- 8 PEs, 16 chares per PE, 128 chares total, 100 iterations
- 32 concurrent kernels with new GPU Manager (vs. 8 without)
- Up to **4.79x** speedup compared to directly using CUDA
- Effectiveness of runtime support depends on application characteristics

Performance Evaluation: `busywait`

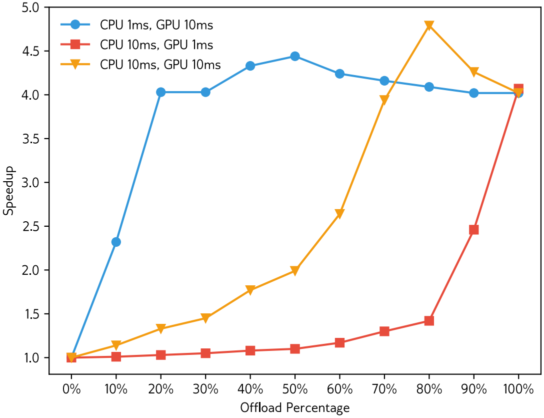


Figure: Speedup of `busywait` benchmark

Performance Evaluation: `stencil2d`

- 2D 5-point iterative stencil benchmark
- Evaluate effectiveness under realistic workload
- 16,384 x 16,384 grid, decomposed into 512 x 512 blocks (chares)
- 8 PEs, 128 chares per PE, 1,024 chares total, 100 iterations
- Vary percentage of chares that offload work to GPU
- 32 concurrent kernels with new GPU Manager (vs. 8 without)
- Up to **2.75x** speedup compared to directly using CUDA

Performance Evaluation: `stencil2d`

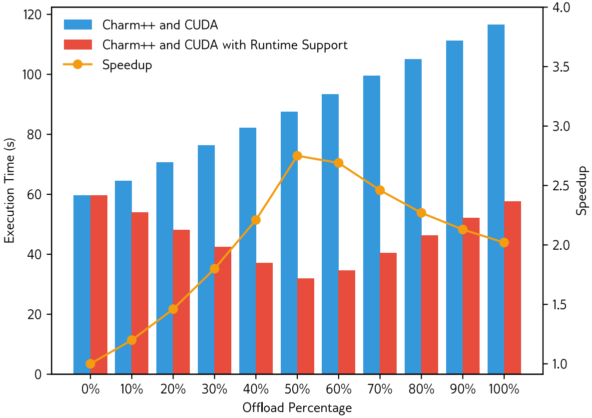


Figure: Execution Time and Speedup of `stencil2d` benchmark

GPU Applications: ChaNGa

- Cosmological N-body simulations
- Leverages GPU Manager
- Offloads physics kernels
- Active work in optimization

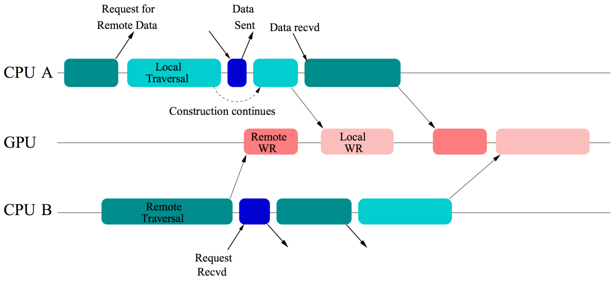


Figure: ChaNGa GPU Manager Design

GPU Applications: Recent ChaNGa Results

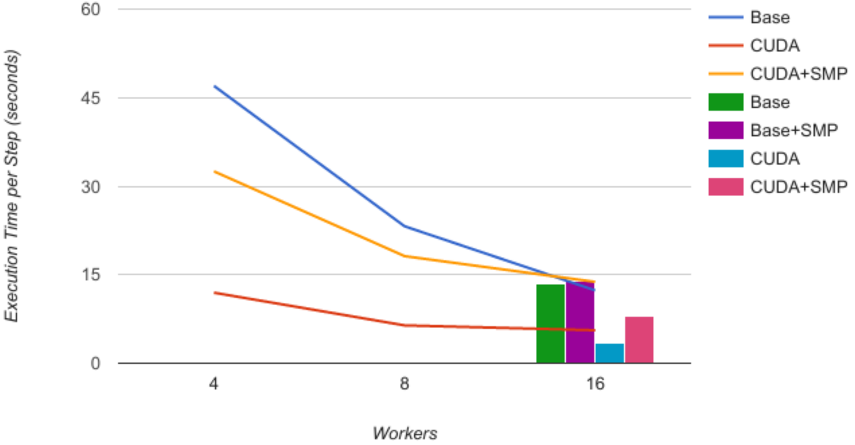


Figure: ChaNGa dwf1 on 4 XK Nodes of BlueWaters

GPU Applications: ChaNGa GPU Tree Walk

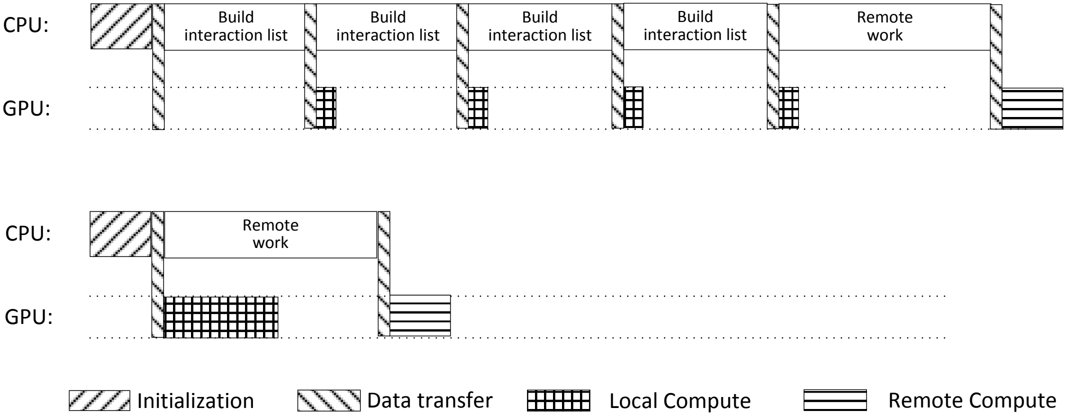
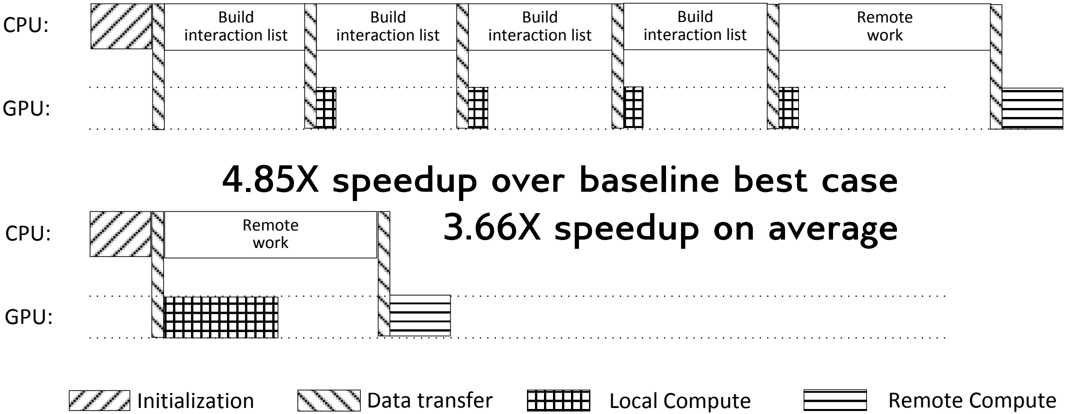


Figure: Strategy Comparison

Jianqiao Liu, Purdue University

GPU Applications: ChaNGa GPU Tree Walk

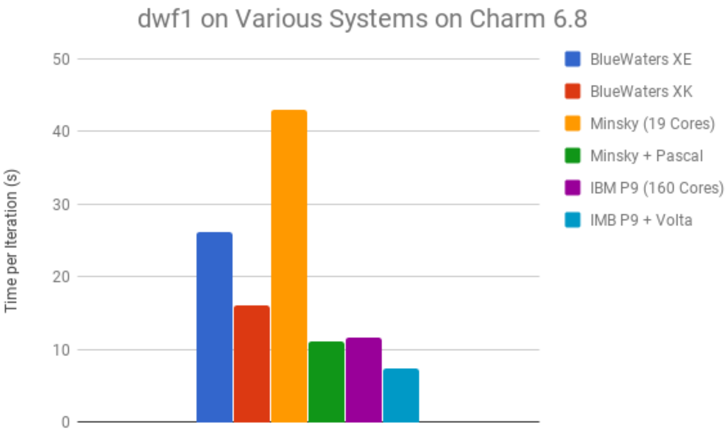


4.85X speedup over baseline best case

3.66X speedup on average

Figure: Strategy Comparison

GPU Applications: ChaNGa on GPU Generations



Mert Hidayetoglu, University of Illinois

Conclusion

- New GPU Manager: presented as a ACM SRC poster at SC'17
- 3 main issues with using GPUs in Charm++
  1. Granularity
  2. Blocking
  3. Responsiveness
- Mostly resolved issue #2, but need more work on issues #1 and #3
- Interesting research topics with fine-grain tasks and GPUs
- Increasing importance of accelerators even for irregular applications

Thank You