How to Write a Parallel GPU Application Using CUDA and Charm++

Presented by Lukasz Wesolowski

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

PARALLEL PROGRAMMING LAB
DEPT. OF COMPUTER SCIENCE, UNIVERSITY OF ILLINOIS

General Purpose GPUs

- Graphics cards adapted for general purpose programming
- Impressive floating point performance
  - 4.6 TFLOPs single precision (AMD Radeon HD 5970)
  - Compared to ~100 GFLOPs for a 3 GHz quad-core quad-issue CPU
- Typically require data parallelism for good performance

CUDA

- A popular hardware/software architecture for GPGPUs
- Supported on all NVIDIA GPUs
- Based on C, with extensions for large-scale data parallelism
- CPU is used to offload and manage units of GPU work

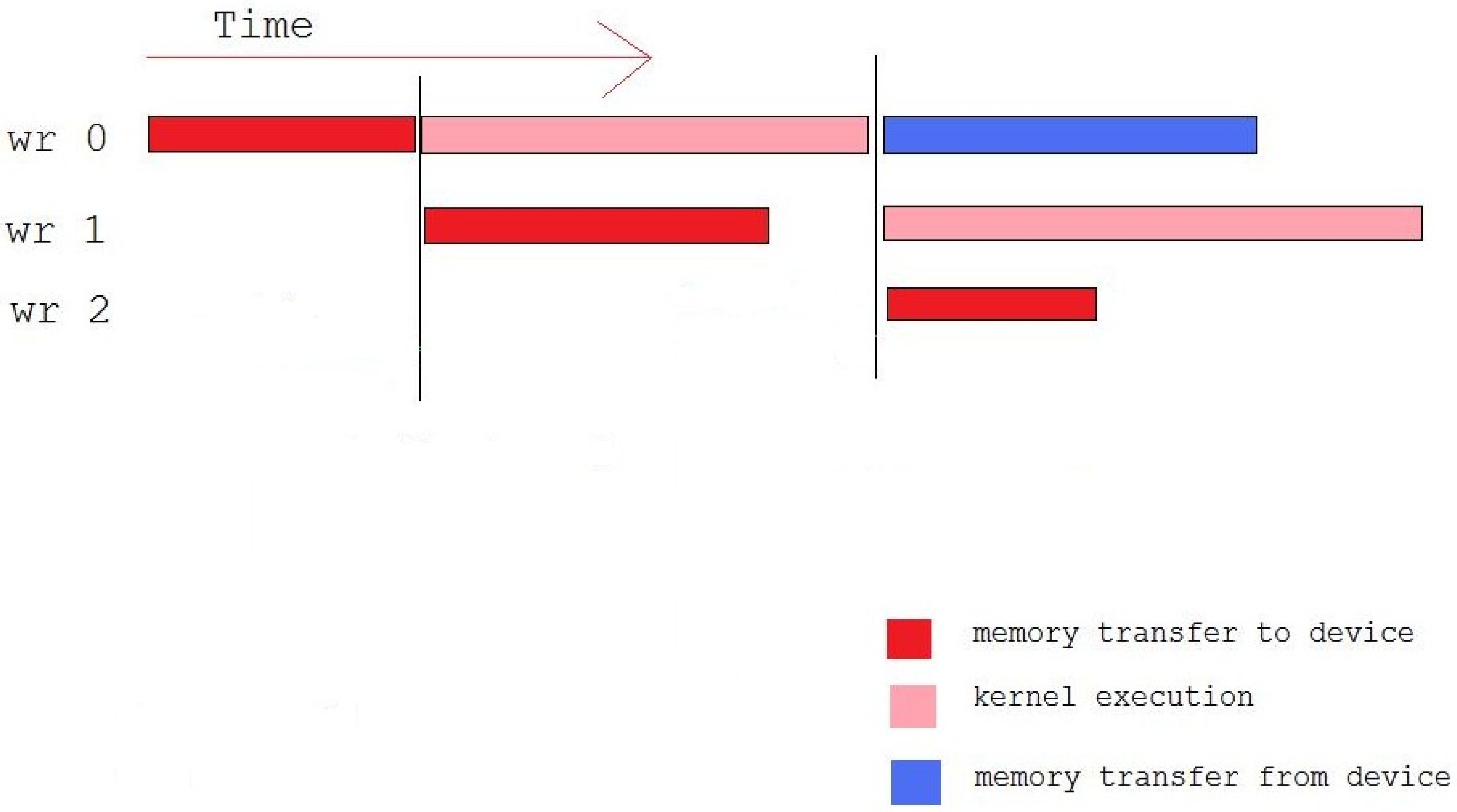
CUDA on Charm++

- Direct approach
  - User makes CUDA calls directly in Charm++
  - User assigns a unique CUDA stream for each chare and makes polling or synchronization calls
- Charm++ GPU Manager
  - User creates a GPU work request and submits it to the runtime system
  - Runtime system manages execution and returns control to the user

Overview of GPU Manager

- User enqueues requests specifying work to be executed on the GPU, associated buffers, and callback
- System transfers memory between CPU and GPU, executes request, and returns through a callback
- GPU operations performed asynchronously
- Three stage pipeline

Execution of Work Requests



Example Code

```
dataInfo *AInfo, *BInfo, *CInfo;

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

AInfo = &(matmul.bufferInfo[0]);
AInfo->bufferID = BUFFERS_PER_CHARE * myIndex + A_INDEX;
AInfo->transferToDevice = YES;
AInfo->transferFromDevice = NO;
AInfo->freeBuffer = YES;
AInfo->hostBuffer = h_A;
AInfo->size = size;

BInfo = &(matmul.bufferInfo[1]);
BInfo->bufferID = BUFFERS_PER_CHARE * myIndex + B_INDEX;
BInfo->transferToDevice = YES;
BInfo->transferFromDevice = NO;
BInfo->freeBuffer = YES;
BInfo->hostBuffer = h_B;
BInfo->size = size;

CInfo = &(matmul.bufferInfo[2]);
CInfo->bufferID = BUFFERS_PER_CHARE * myIndex + C_INDEX;
CInfo->transferToDevice = NO;
CInfo->transferFromDevice = YES;
CInfo->freeBuffer = YES;
CInfo->hostBuffer = h_C;
CInfo->size = size;

matmul.callbackFn = cb;
matmul.id = MATMUL_KERNEL;

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

enqueue(wrQueue, &matmul);
```

```
void kernelSelect(workRequest *wr) {

  switch (wr->id) {
  case 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));
    break;
  }
}
```