Advanced Topics: Streams, Multi-GPU, Tools, Libraries, etc.

Streams

- Until now, we have largely focused on massively data-parallel execution on GPUs.

- Task parallelism (also available on CPUs) is also possible on GPUs.

- Rather than simultaneously computing the same function on lots of data (data parallelism), task parallelism involves doing two or more completely different tasks in parallel.

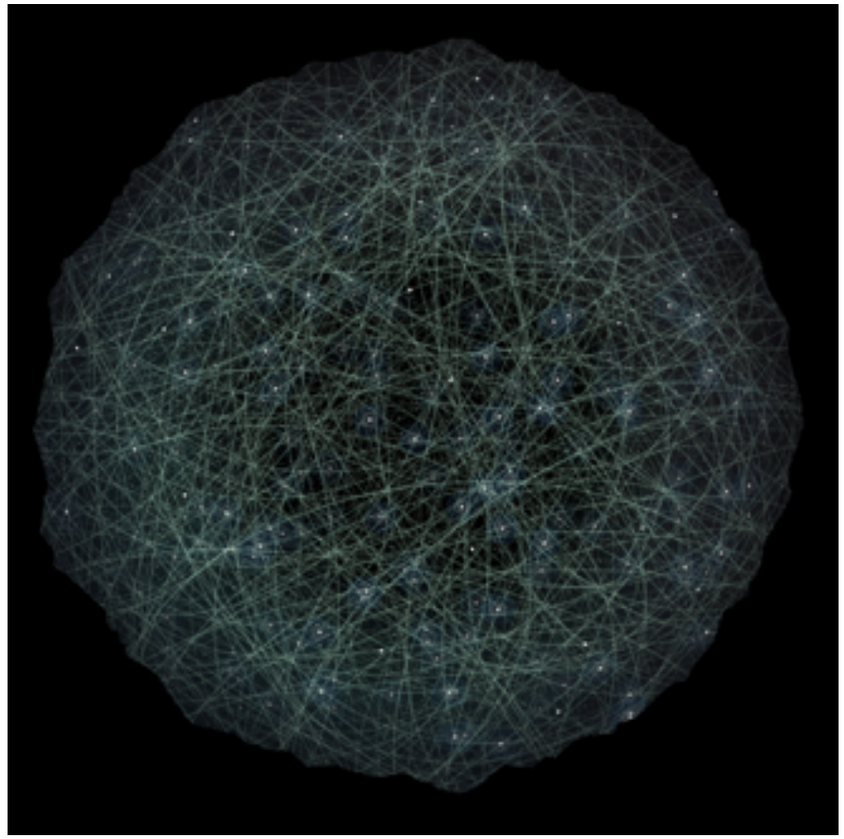
- Task parallelism is not as flexible as data parallelism, but it allows the extraction of even more optimization from GPU-based implementation of algorithms.

Page-Locked Host Memory

- Recall: To allocate memory on the GPU, we used cudaMalloc(). On the host, we used malloc().

- CUDA has another option: cudaHostAlloc(). While malloc allocates a standard pageable host memory, cudaHostAlloc() allocates a buffer of page-locked (or pinned) host memory.

- Page-locked buffers are guaranteed to remain in physical memory by the operating system. Therefore, it will never page out to a disk.
  - Uses real addresses rather than virtual ones so memory bandwidth is higher.
  - Page locked memory will reduce memory available to the operating system, so it will run out of memory more quickly.



CUDA Stream

- A CUDA Stream is a sequence of operations (commands) that are executed in order.

- CUDA streams can be created and executed together and interleaved although the “program order” is always maintained within each stream.

- Streams proved a mechanism to overlap memory transfer and computations operations in different stream for increased performance if sufficient resources are available.

Creating a Stream

- Done by creating a stream object and associated it with a series of CUDA commands that then becomes the stream. CUDA commands have a stream pointer as an argument.

- Cannot use regular cudaMemcpy with streams, need asynchronous commands for concurrent operation.

- cudaMemcpyAsync is an asynchronous version of cudaMemcpy that copies date to/from host and the device.
  - May return before copy complete
  - A stream argument specified.
  - Needs "page-locked" memory.

- Multiple calls to cudaStreamCreate can create multiple streams. Then multiple kernels (even different ones!) can be executed on each stream.

```
cudaStream_t stream1;
cudaStreamCreate(&stream1);

cudaMemcpyAsync(…,  stream1);
MyKernel<<< grid, block, stream1>>>(…);
cudaMemcpyAsync(… , stream1);
```

Naive Stream Example

```
#define SIZE (N*20)
…
int main(void) {
  int *a, *b, *c;
  int *dev_a, *dev_b, *dev_c;

  cudaMalloc( (void**)&dev_a, N * sizeof(int) );
  cudaMalloc( (void**)&dev_b, N * sizeof(int) );
  cudaMalloc( (void**)&dev_c, N * sizeof(int) );

  // paged-locked allocation
  cudaHostAlloc((void**)&a,SIZE*sizeof(int),cudaHostAllocDefault);
  cudaHostAlloc((void**)&b,SIZE*sizeof(int),cudaHostAllocDefault);
  cudaHostAlloc((void**)&c,SIZE*sizeof(int),cudaHostAllocDefault);

  // generate data
  for(int i=0;i<SIZE;i++) {
       a[i] = rand();
       b[i] = rand();
  }

  // loop over data in chunks of two
  for(int i=0;i < SIZE;i+= N {
       cudaMemcpyAsync(dev_a,a+i,N*sizeof(int),cudaMemcpyHostToDevice,stream);
       cudaMemcpyAsync(dev_b,b+i,N*sizeof(int),cudaMemcpyHostToDevice,stream);
       MyKernel<<<(int)ceil(N/1024)+1,1024,0,stream>>>(dev_a,dev_b,dev_c);
       cudaMemcpyAsync(c+1,dev_c,N*sizeof(int),cudaMemcpyDeviceToHost,stream);
  }
  cudaStreamSynchronize(stream); // wait for stream to finish
  cudaStreamDestroy(stream); // because we care
  return 0;
}
```

Multiple Streams v.1.0

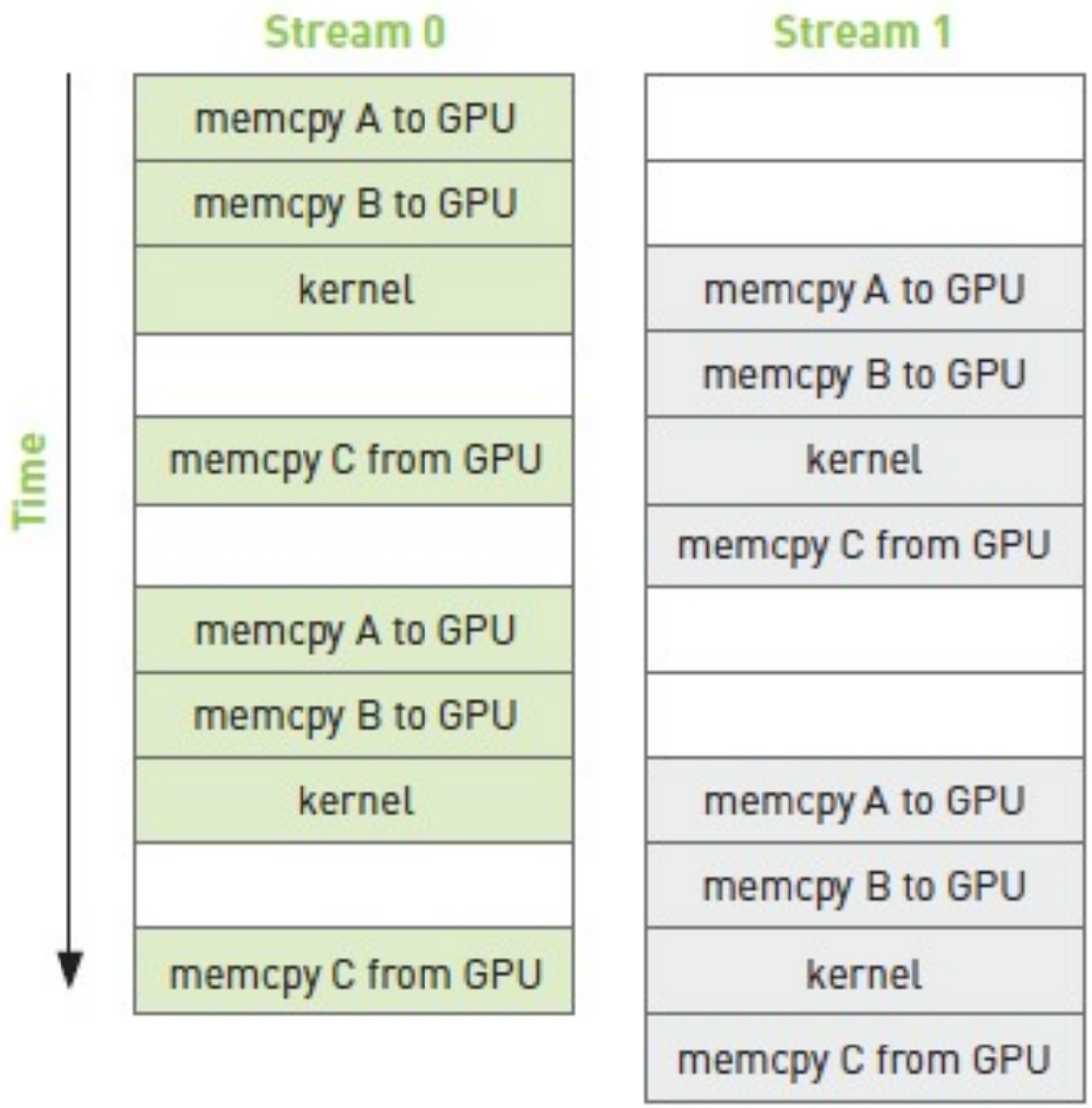
```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);


int *dev_a1, *dev_b1, *dev_c1; // stream 1 mem ptrs
int *dev_a2, *dev_b2, *dev_c2; // stream 2 mem ptrs

//stream 1
cudaMalloc( (void**)&dev_a1, N * sizeof(int) );
cudaMalloc( (void**)&dev_b1, N * sizeof(int) );
cudaMalloc( (void**)&dev_c1, N * sizeof(int) );
//stream 2
cudaMalloc( (void**)&dev_a2, N * sizeof(int) );
cudaMalloc( (void**)&dev_b2, N * sizeof(int) );
cudaMalloc( (void**)&dev_c2, N * sizeof(int) );
…
for(int i=0;i < SIZE;i+= N*2 {           // loop over data in chunks
  // stream 1
  cudaMemcpyAsync(dev_a1,a+i,N*sizeof(int),cudaMemcpyHostToDevice,stream1);
  cudaMemcpyAsync(dev_b1,b+i,N*sizeof(int),cudaMemcpyHostToDevice,stream1);
  kernel<<<(int)ceil(N/1024)+1,1024,0,stream1>>>(dev_a,dev_b,dev_c);
  cudaMemcpyAsync(c+1,dev_c1,N*sizeof(int),cudaMemcpyDeviceToHost,stream1);
  //stream 2
  cudaMemcpyAsync(dev_a2,a+i,N*sizeof(int),cudaMemcpyHostToDevice,stream2);
  cudaMemcpyAsync(dev_b2,b+i,N*sizeof(int),cudaMemcpyHostToDevice,stream2);
  kernel<<<(int)ceil(N/1024)+1,1024,0,stream2>>>(dev_a,dev_b,dev_c);
  cudaMemcpyAsync(c+1,dev_c2,N*sizeof(int),cudaMemcpyDeviceToHost,stream2);
}
cudaStreamSynchronise(stream1); // wait for stream1 to finish
cudaStreamSynchronise(stream2); // wait for stream2 to finish
```

Multiple Streams

- The main goal is to overlap the memory transfer with the kernel execution.

- One of the reasons why this works is because the cudaMemcpyAsync is a request for memory transfer, but the code continues on.

- Note: Sometimes, the most well-intended "enhancements" do nothing more than introduce complications to the code.



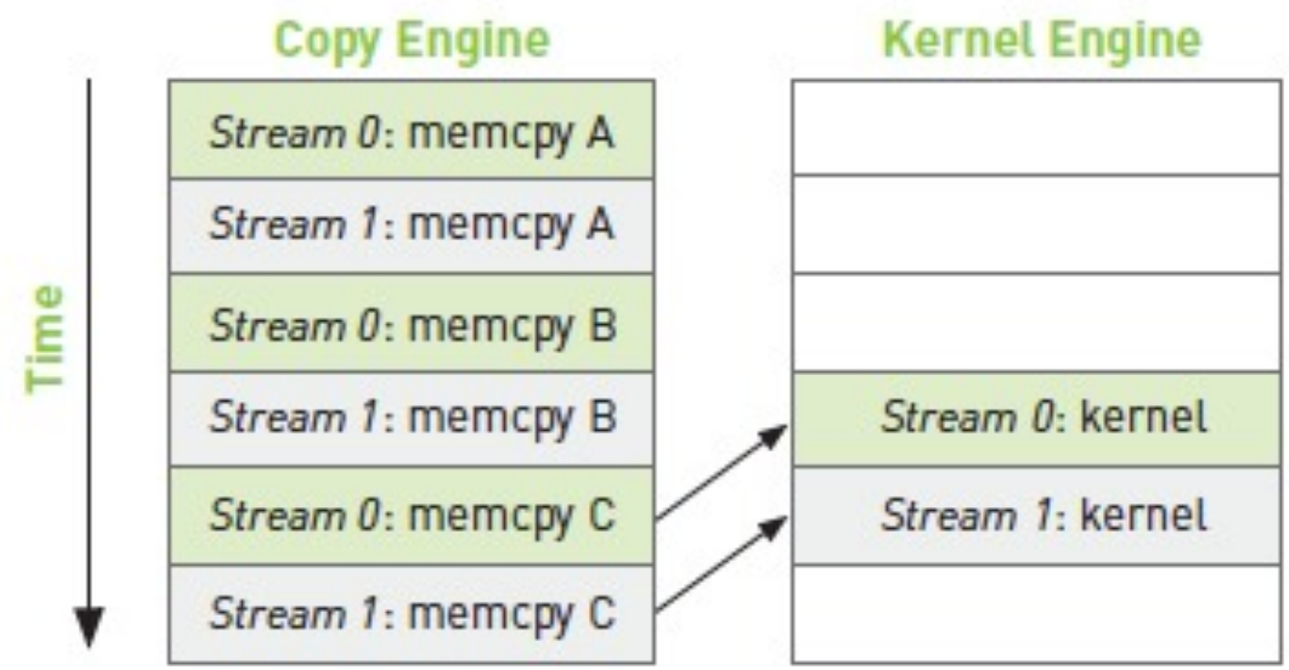
Multiple Streams v.2.0

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);


int *dev_a1, *dev_b1, *dev_c1; // stream 1 mem ptrs
int *dev_a2, *dev_b2, *dev_c2; // stream 2 mem ptrs

//stream 1
cudaMalloc( (void**)&dev_a1, N * sizeof(int) );
cudaMalloc( (void**)&dev_b1, N * sizeof(int) );
cudaMalloc( (void**)&dev_c1, N * sizeof(int) );
//stream 2
cudaMalloc( (void**)&dev_a2, N * sizeof(int) );
cudaMalloc( (void**)&dev_b2, N * sizeof(int) );
cudaMalloc( (void**)&dev_c2, N * sizeof(int) );
…
for(int i=0;i < SIZE;i+= N*2 {           // loop over data in chunks
  // interweave stream 1 and steam 2
  cudaMemcpyAsync(dev_a1,a+i,N*sizeof(int),cudaMemcpyHostToDevice,stream1);
  cudaMemcpyAsync(dev_a2,a+i,N*sizeof(int),cudaMemcpyHostToDevice,stream2);
  cudaMemcpyAsync(dev_b1,b+i,N*sizeof(int),cudaMemcpyHostToDevice,stream1);
  cudaMemcpyAsync(dev_b2,b+i,N*sizeof(int),cudaMemcpyHostToDevice,stream2);
  kernel<<<(int)ceil(N/1024)+1,1024,0,stream1>>>(dev_a,dev_b,dev_c);
  kernel<<<(int)ceil(N/1024)+1,1024,0,stream2>>>(dev_a,dev_b,dev_c);
  cudaMemcpyAsync(c+1,dev_c1,N*sizeof(int),cudaMemcpyDeviceToHost,stream1);
  cudaMemcpyAsync(c+1,dev_c2,N*sizeof(int),cudaMemcpyDeviceToHost,stream2);
}
cudaStreamSynchronise(stream1); // wait for stream1 to finish
cudaStreamSynchronise(stream2); // wait for stream2 to finish
```

Multiple Streams

- If we assume that the memory copies and the kernel execution times are about the same, the memory transfer will properly overlap with the kernel calls.

- In practice, timing the overlap requires experimenting, but it can (under optimal conditions) effectively remove the memory transfer latency by hiding it.



Why Multi-GPU Programming?

- Many HPC systems contain multiple GPUs.
  - Servers: S2070 = 4 GPUs
  - Desktops: GreenFlashXX = 2 GPUs each connected by 3-way SLI (Scalable Link Interface)
- Additional processing power to divide work between GPUs.
- Reduced memory transfer latency.
- Additional Memory
  - ~~Some~~ Many interesting problems do not fit into a single GPU.

Multi-GPU Memory

- GPUs do not share global memory.
  - A kernel executing on one GPU cannot access memory on another GPU.
- Inter-GPU communication
  - Application code is responsible for transferring data between GPUs as necessary.
  - Data travels across the PCIe bus, even when GPUs are connected to the same PCIe switch.

Multiple Device Management

- GPUs have consecutive integer IDs, starting with 0.

- cudaGetDeviceCount(int *num_devices);
  - Command is designed to determine whether multi-GPU usage is possible.

- cudaSetDevice(int device_id)
  - Device selection within the code by specifying the identifier and making CUDA kernels run on the selected GPU.
  - If command is not called, the default behavior is device_id = 0.

```
int main(int argc, char **argv) {
  int deviceCount;

  cudaGetDeviceCount(&deviceCount);

  if (deviceCount < 2) {
    printf("Error: Only %d GPUs found.\n",
            deviceCount);
  }

  // ...
  cudaSetDevice(1); // use second GPU
  // ...
}
```

Multiple Device Kernel Calls

- The CPU-GPU context must be established before commands are issued to the GPU.

- The cudaSetDevice command sets the context by which the commands are issues to a specific GPU.

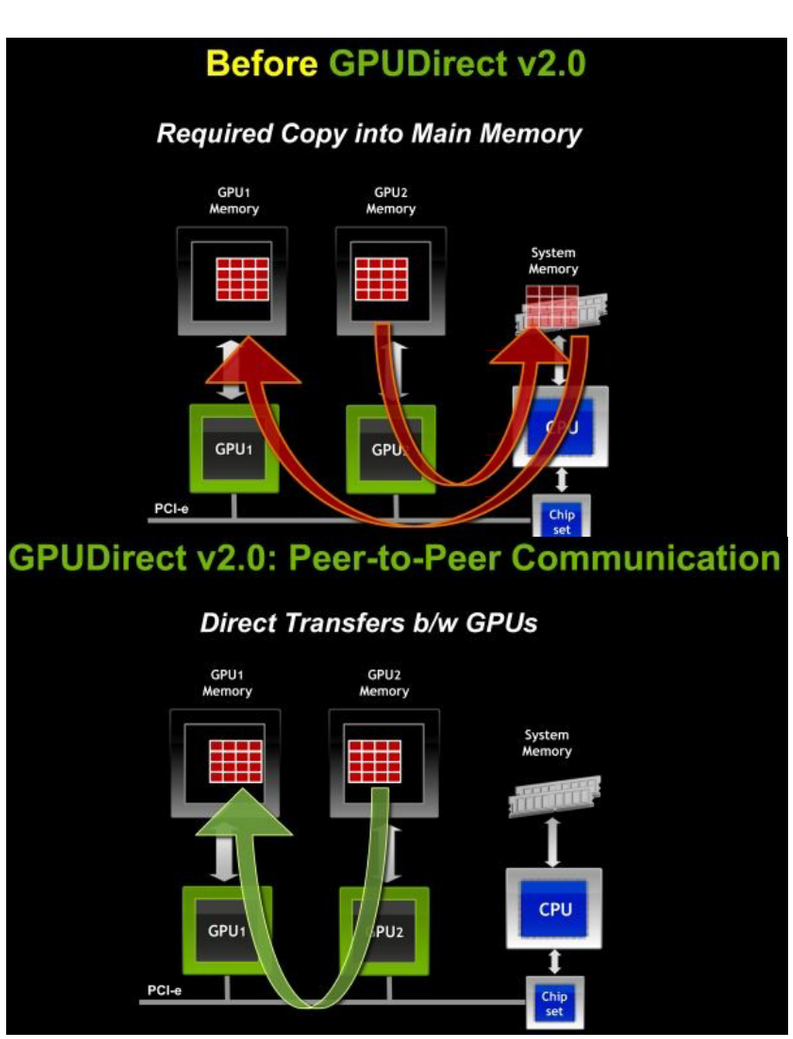
- cudaDeviceSynchronize() waits until all preceding commands in all streams of all host threads have completed.

- Note that all of each of the GPUs can execute different kernels too, just like with streams.

```
int size = 1024 * sizeof(int);

// set device 0 as current
cudaSetDevice(0);

float *p0;
// allocate memory on device 0
cudaMalloc(&p0, size);
// launch kernel on device 0
MyKernel<<<grid, block>>>(p0);

// set device 1 as current
cudaSetDevice(1);

float *p1;
// allocate memory on device 1
cudaMalloc(&p1, size);
// launch kernel on device 1
MyKernel<<<grid, block>>>(p1);

// you know there had to be one of these
cudaDeviceSynchronize();
```

Multiple Device Memory Transfer

- Peer to peer memory copy must be done explicitly.
- It used to be done via a direct copy to CPU main memory (pre-Fermi).
- Now direct transfers between GPUs.




```
int size = 1024 * sizeof(int);

// set device 0 as current
cudaSetDevice(0);

float *p0;
// allocate memory on device 0
cudaMalloc(&p0, size);
// launch kernel on device 0
MyKernel<<<grid, block>>>(p0);

// set device 1 as current
cudaSetDevice(1);

float *p1;
// allocate memory on device 1
cudaMalloc(&p1, size);
// copy p0 to p1 from device 0 to 1
cudaMemcpyPeer(p1, 1, p0, 0, size);
// launch kernel on device 1
MyKernel<<<grid, block>>>(p1);

// you know there had to be one of these
cudaDeviceSynchronize();
```

CUDA Tools + Libraries

- CUFFT - Fast Fourier Transform library (NVIDIA, free)
- CUBLAS - Basic Linear Algebra Subprograms (NVIDIA, free)
- CUDPP - CUDA Data Parallel Primitives Library (UC Davis, free)
  - Parallel Scan, Sort, Reduction, etc.
- CULA - CUDA implementation of industry standard Linear Algebra Package (LAPACK).
  - QR factorization, linear system solver, singular value decomposition, least squared solvers, etc.
- Language wrappers
  - JCUDA (Java), PyCUDA (Python)
- etc. etc. etc.