GPU programming in CUDA: Using multiple GPUs

Lukas Einkemmer
Department of Mathematics
University of Innsbruck

PRACE Autumn School, Innsbruck
Link to slides: `http://www.einkemmer.net/training.html`

Goals

Our goals in this section are

- Understanding asynchronous execution.
- How to use multiple GPUs
- What additional performance considerations need to be taken into account

Interleaving different tasks

Synchronous vs. Asynchronous

Usual CUDA execution model looks like

```
// Copy necessary data to host.
cudaMemcpy(..., cudaMemcpyHostToDevice);

// Launch one or multiple kernels.
kernel<<<num_threads, num_blocks>>>(...);

do_some_cpu_work();

// Copy results back to the host.
cudaMemcpy(..., cudaMemcpyDeviceToHost);
```

Kernel launches are asynchronous.

- `do_some_cpu_work` is executed concurrently with the kernel.

`cudaMemcpy` waits for the kernel to complete and then copies back the data.

Interleaving different tasks

There are situations were

- computation on the GPU
- moving data from and to the GPU
- moving data between two different GPUs
- doing computation on the CPU

can **take place in parallel**.

Data transfer is often slow. **We are going to discuss a way to hide that overhead.**

CUDA streams

CUDA operates with so-called **streams**.

A **stream** is a handle for a **sequence of operations that depend on each other**.

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

The default stream is 0. **By default all operations belong to the default stream.**

Interleaving computations on CPU and GPU

```
// Starts an memory operation in stream 0.
cudaMemcpyAsync(x_d, x_h, size, cudaMemcpyHostToDevice, 0);

// Waits for cudaMemcpyAsync to complete.
kernel<<<num_threads, num_blocks>>>(...);

// Runs concurrently with the kernel call.
do_some_cpu_work();
```

Since both **cudaMemcpyAsync and the kernel launch** are placed in the default stream (stream zero) they **run in sequence (the kernel after the copy)**.

Interleaving communication with computation

```
// Create two CUDA streams.
cudaStream_t stream1; cudaStreamCreate(&stream1);
cudaStream_t stream2; cudaStreamCreate(&stream2);

// Starts an asynchronous memory transfer in
// stream1.
cudaMemcpyAsync(x_d, x_h, size,
        cudaMemcpyHostToDevice, stream1);

// Executes concurrently with cudaMemcpyAsync
// (MUST not depend on x_d).
kernel<<<num_threads, num_blocks, 0, stream2>>>(...);

// Waits for cudaMemcpyAsync to complete.
kernel<<<num_threads, num_blocks, 0, stream1>>>(...);

// Copy results back to the host.
cudaMemcpy(x_h, x_d, size, cudaMemcpyDeviceToHost);
```

Synchronization

We can explicitly wait for the completion of a stream by calling

```
cudaStreamSynchronize(stream1);
```

Similar to `cudaDeviceSynchronize` but only applies to tasks in the stream.

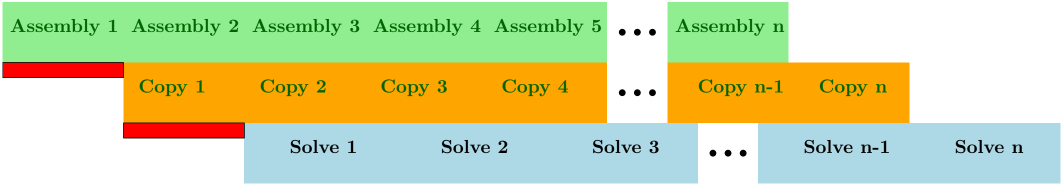
Interleave communication with computation

We have three tasks

- Matrix **assembly** which is **best done on the GPU**.
- **Copy** the assembled matrix to the CPU.
- **Solve** the resulting linear system on the CPU.

But: copy and solve almost take the same time.

Solution: split the data set into smaller chunks and do the assembly and copy asynchronously.



Multiple GPUs

Running on multiple GPUs

Available devices are numbered 0 to number_of_devices-1.

```
$ ./deviceQuery

Detected 4 CUDA Capable device(s)

Device 0: "Tesla V100-SXM2-16GB"
Device 1: "Tesla V100-SXM2-16GB"
Device 2: "Tesla V100-SXM2-16GB"
Device 3: "Tesla V100-SXM2-16GB"
```

Commands such as kernel launches/memory allocation/... are issued for the currently active device.

The active device can be changed as follows

```
cudaSetDevice(i);
// k_my_kernel is launched on device i
k_my_kernel<<<...>>>(...);
```

Running on multiple GPUs

The **active device can be changed even if async calls are still executing**.

```
cudaSetDevice(0);
k_my_kernel<<<...>>>(...);
cudaMemcpyAsync(...);
cudaSetDevice(1);
k_another_kernel<<<...>>>(...);
```

Synchronization

```
cudaSetDevice(0); cudaDeviceSynchronize();
cudaSetDevice(1); cudaDeviceSynchronize();
```

Call to `cudaDeviceSynchronize` only synchronizes the current CUDA context.

Recommendation: Use a dedicated stream for each GPU (in lieu of stream 0).

Running on multiple GPUs

We can set a device in a multi-threaded environment.

Common pattern: Use one thread for each GPU

```
// sequential program on the CPU

#pragma omp parallel for num_threads(4)
for(int i=0;i<4;i++)
{
    cudaSetDevice(i);

    cudaMemcpy(...);
    k_my_kernel<<<...>>>(...);
    cudaMemcpy(...);
}

// sequential program on the CPU
```

Common pattern: Use one MPI process per GPU.

Running on multiple GPUs

Beware that the currently active device is managed on a per thread basis.

WRONG!

```
cudaSetDevice(1);
#pragma omp parallel
{
    k_my_kernel<<<...>>>(...);
}
```

Correct.

```
cudaSetDevice(1);
#pragma omp parallel
{
    cudaSetDevice(1);
    k_my_kernel<<<...>>>(...);
}
```

Distribute your program to multiple GPUs

We are in a **distributed memory** environment now.

Even if Unified Memory allows us to read memory from all devices, access speed is not the same.

- Programmer has to think **how to distribute data** in order to minimize data movement between devices.
- Not dissimilar to MPI, although at a smaller scale.

What we can expect

- V100 main memory: 900 GB/s
- V100 NVLink device-to-device: 300 GB/s
- PCIe 3.0 16x: 16 GB/s

Mechanics

Data transfer **between GPUs** works in in the same way as data transfer between device and host.

```
cudaMemcpy(d_src, d_dest, sizeof(double)*n,
        cudaMemcpyDefault);
```

Devices that are involved in data transfer are inferred from pointer d_src and d_dest.

- No need to explicitly specify source and target device.

No guarantee that copy is device to device.

- Data flow could look like
  device 0 $\to$ host $\to$ device 1

Peer-to-peer transfer

Many modern GPU systems are able to directly (without involving the host) transfer data.

- This is called **peer-to-peer** transfer.

Enable peer-to-peer data transfer between device i and j

```
int is_able;
cudaSetDevice(i);
cudaDeviceCanAccessPeer(&is_able, i, j);
if(is_able)
    cudaDeviceEnablePeerAccess(j, 0);
```

Without Unified Virtual Addressing (UVA) we need to use

```
cudaMemcpyPeer(dst, dst_device_id, src, src_device_id, size);
```

Query the system

All the information we get from `deviceQuery` can be obtained programmatically.

```
int num;
cudaGetDeviceCount(&num); // number of CUDA devices

for(int i=0;i<num;i++) {
    // Query the device properties.
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, i);

    cout << "Device id: " << i << endl;
    cout << "Device name: " << prop.name << endl;
}
```

Exercise

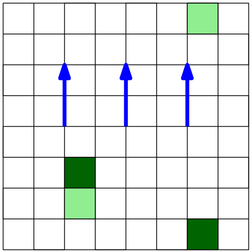
We have a **two-dimensional stencil code that solves an advection problem** (`exercise-multiplegpu.cu`).

- Advection in the y direction with periodic boundary conditions.
- Problem is setup such that we return to the original state.

Goal is to parallelize the program to two GPUs.

Exercise

Single GPU



2 GPUs