GPU Teaching Kit

Accelerated Computing

Module 24 – Multi-GPU

Lecture 24.1 – OpenMP

Objective

- To learn some of the main OpenMP features
  - OpenMP programming model
  - Differences between the threaded and process parallel programming model
  - Parallel regions
  - Synchronization barriers
  - Special sections

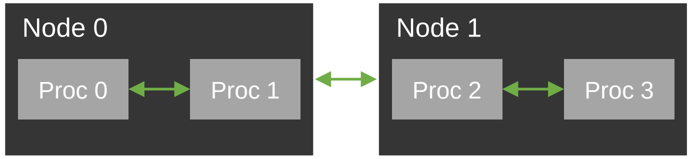
OpenMP

- A parallel programming model based on the concepts of multithreading and shared memory
- OpenMP programs consists on annotations written into the serial code, called directives
- Needs a compiler with OpenMP support, which will translate the directives to the actual parallel code
- Highly portable across systems
- Limited to a single computer, however it can handle several processors

Differences between process and threaded parallelism

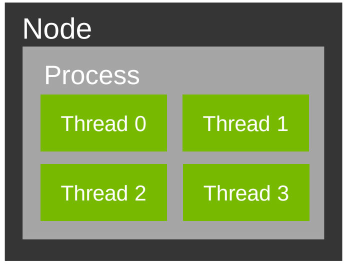
Process parallelism

- Uses local memory forcing each parallel worker to have its own memory space.
  - It needs explicit data sharing routines creating communication overhead.
- It's controlled by a main process.
- It can scale to multiple computers.
- Example: MPI (see Module 18)



Threaded parallelism

- Uses shared memory, enabling all parallel workers to access the same memory space.
- It's controlled by a main thread, with all threads running on a single process.
- It can't scale beyond a single computer.
- Example: OpenMP

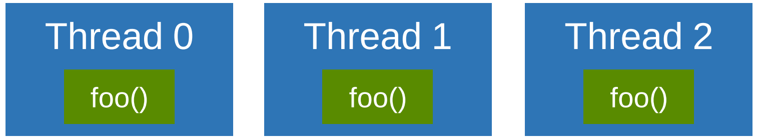


OpenMP parallel regions

- OpenMP is a directive based parallel programming model, meaning we annotate the code to introduce parallelism.
- The directive to create a parallel region is:
  - #pragma omp parallel
  - This pragma is added right before the block of code to parallelize.
  - This pragma runs the instructions inside the code block in each parallel thread.

- Example parallel behavior of calling a function inside a parallel region:

```
#pragma omp parallel
{
    foo();
}
```



Visualization of the behavior of the parallel region

Parallelizing a “for” loop with OpenMP

– OpenMP is aware that in many cases “for” loops are a source for introducing parallelism, that is why it has special directives for loops

– To parallelize a for loop exists the directive:
  – #pragma omp parallel for
  – This pragma is added right before the for loop to parallelize.
  – An OpenMP compiler will identify the pragma and create a program where the loop is executed in parallel.

– Serial version of the vector sum algorithm:

```
for(int i = 0; i < n; ++i)  {
  c[i] = a[i] + b[i];
}
```

– Parallel OpenMP version of the vector sum algorithm:

```
#pragma omp parallel for
for(int i = 0; i < n; ++i)  {
  c[i] = a[i] + b[i];
}
```

#pragma omp parallel for

- Each variable used inside the loop has a kind determining if it’s accessible by other threads: shared, private, firstprivate, lastprivate
- Shared variables are accessible by every thread in the region
- Private variables are local to each thread and are not initialized
- Firstprivate behave similarly to private variables but are initialized with the value before the parallel region
- Unless otherwise stated, every variable outside the loop will be considered shared and every inside the loop will be considered private

OpenMP Synchronization

- By default every parallel region will wait until the region is finished before executing other instructions. To change this behavior use the nowait option when declaring the parallel region
- #pragma omp barrier specifies a synchronization point equivalent to __syncthreads() in CUDA, the pragma is not associated to a block of code

```
#pragma omp parallel for
for(int i = 0; i < n; ++i) {
  c[i] = a[i] + b[i];
  …
#pragma omp barrier
  c[i] += i >= 1 ? c[i - 1] : 0;
}
```

OpenMP Sections

- Critical sections:
  - Are executed by only one thread at a time
  - Are associated to a block of code
  - #pragma omp critical
- Single sections:
  - Are executed by only one thread of the team
  - Are associated to a block of code
  - #pragma omp single

```
int a = 0;
int b = 0;
#pragma omp parallel num_threads(4)
{
#pragma omp critical
  a += 1 ;
#pragma omp single
  b += 1;
}
```

- a final value is 4, because each thread executed the section one at a time
- b final value is 1, because only one thread executed the section

OpenMP API

- OpenMP in addition to the directives have an API, to control or obtain certain runtime parameters.
- As is the case with CUDA, OpenMP allows users to set the number of threads, get the number of threads and uniquely identify each thread.

- omp_set_num_threads(int):
  - Sets the desired number of threads to be used by the OpenMP runtime on subsequent parallel regions.
- omp_get_num_threads():
  - Returns the number of threads being used in the parallel region.
- omp_get_thread_num():
  - Returns a numeric identifier for the calling thread, with different threads having different identifiers.

Matrix Vector Multiplication OpenMP example

```
void tileMultiply(double* A, double* x, double y, int rows, int cols, int tileSize){
    int rid = omp_get_thread_num() * tileSize;            // Get the first row in the tile to compute
    for(int r = rid; r < min(rows, rid + tileSize); ++r ) {   // Iterate through the rows in the tile
        double sum = 0;
        for(int c = 0; c < cols; ++c)
            sum += A[r * cols + c] * x[c];                 // Accumulate the dot product between r-row and x
        y[r] = sum;                                        // Store the result into the result
    }
}
…
#pragma omp parallel                                       // Create a parallel region
{
    int tnum = omp_get_num_threads();                      // Get the number of threads executing the region
    int tileSize = (rows + tnum - 1) / tnum;               // Calculate the number of rows in a tile
    tileMultiply(A, x, t, rows, cols, tileSize);           // Dispatch the function calculating the multiplication
}
```

Compiling an OpenMP program

- OpenMP compilers typically will ignore the OpenMP pragmas unless we specify the OpenMP flag.
- To use the API functions you need to include in the appropriate file:
  - #include <omp.h>
- Examples:
  - GCC & Clang: to compile an OpenMP annotated source file we use:
    - gcc <file to compile> -fopenmp
    - clang <file to compile> -fopenmp
    - Additionally if there are multiple OpenMP libraries, you may specify which version to use, for example –lgomp uses the GNU implementation.
  - NVCC: to compile an OpenMP annotated source file we use:
    - nvcc <options> -Xcompiler -fopenmp

GPU Teaching Kit

Accelerated Computing

GPU Teaching Kit

Accelerated Computing

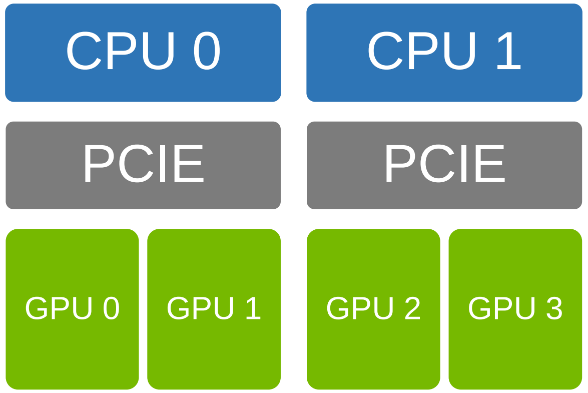
Module 24 – Multi-GPU

Lecture 24.2 – Multi GPU Introduction I

Objective

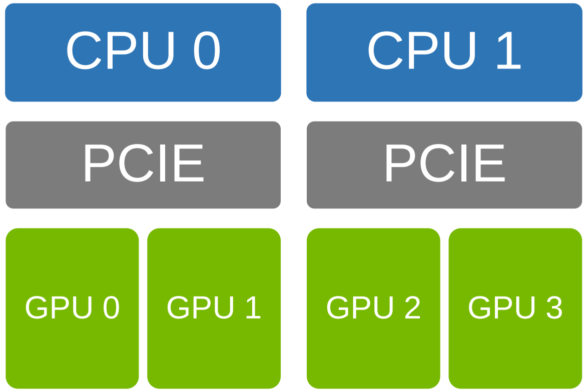
- To learn the important concepts involved on the use of systems with Multiple GPU's
  - Devices information
  - Memory allocation
  - Kernel launching

Schematic of a Multi GPU system



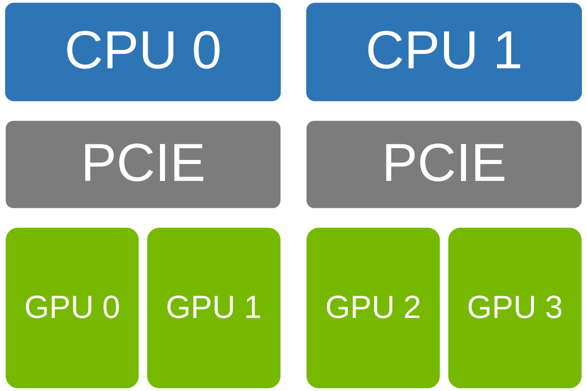
- The figure represents a system with 2 CPU's and 4 GPU's.
- GPU's are numbered from 0 to n-1, where n is the number of GPU’s.
- The CUDA driver always starts with a default active device.
- There are two broad types of Multi GPU communication:
  - Through the PCIE bus
  - Through NVLINK

CUDA host API calls for Multi GPU's



- cudaSetDevice()
  - Set GPU device to use for device code execution on the active host thread.
  - Requires one parameter:
    - An int with the device id number
  - This function doesn't affect other host threads, meaning that setting the device on one thread will not set the device in other host threads. Also doesn't affect previous async calls.

CUDA host API calls for Multi GPU's



- cudaGetDevice()
  - Get GPU device being currently used by the active host thread.
  - Requires one parameter:
    - An int pointer to store the device id
- cudaGetDeviceCount()
  - Get the number of CUDA-capable devices in the system.
  - Requires one parameter:
    - An int pointer to store the device count

CUDA host API calls for Memory allocation with Multiple GPU’s

To allocate or associate memory with a specific device using non-Managed CUDA-API calls, it’s necessary to call cudaSetDevice() before doing the allocation call.

- cudaMalloc()
  - Allocates an object in the device global memory
  - Two parameters
    - Address of a pointer to the allocated object
    - Size of allocated object in terms of bytes

- cudaHostAlloc()
  - Allocates pinned memory on the host
  - Three parameters
    - Address of pointer to the allocated memory
    - Size of the allocated memory in bytes
    - Host Alloc flags

CUDA host API calls for Memory allocation with Multiple GPU's Unified Memory

- If the flag cudaDevAttrConcurrentManagedAccess is set in all devices, then it's not necessary to call cudaSetDevice before the cudaMallocManaged call.
- If the flag is not set but devices can access each others memory, then calling cudaSetDevice before the cudaMallocManaged call will establish the context for the managed memory on the active device.
  - With other devices accessing the data via PCIE at reduced bandwidth.

CUDA runtime calls affected by cudaSetDevice

- If cudaSetDevice() was called before a kernel launching call, the kernel will execute in the active device.
  - It’s crucial that every non managed memory being used in the kernel resides in the active device, otherwise an error will occur.

- If cudaSetDevice() was called before a cudaStreamCreate(), then the stream will be associated with the active device.

- The synchronization functions: cudaDeviceSynchronize(), cudaStreamSynchronize() are also affected by cudaSetDevice(), synchronizing tasks only for the active device on the active host thread

Putting it all together, vecAdd

```
float *m_A0, float *m_B0, *m_A1, float *m_B1, int n;
int size = n * sizeof(float);

cudaSetDevice(0);                                        // Will set the active device to 0
cudaMalloc((void**) &m_A0, size);                    // Will allocate memory on device 0
cudaMalloc((void**) &m_B0, size);                    // Will allocate memory on device 0

cudaSetDevice(1);                                        // Will set the active device to 1
cudaMalloc((void**) &m_A1, size);                    // Will allocate memory on device 1
cudaMalloc((void**) &m_B1, size);                    // Will allocate memory on device 1

// Memory initialization on the Host and memory transfers

cudaSetDevice(0);                                        // Set the device for kernel execution
vecAdd<<<gridDim, blockDim>>>(m_A0,m_B0);

cudaSetDevice(1);                                        // Set the device for kernel execution
vecAdd<<< gridDim, blockDim>>>(m_A1,m_B1);

cudaFree(m_A0); cudaFree(m_B0);
cudaFree(m_A1); cudaFree(m_B1);
```

GPU Teaching Kit

Accelerated Computing

GPU Teaching Kit

Accelerated Computing

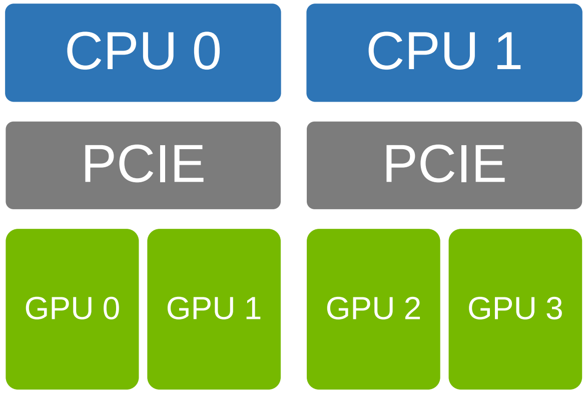
Module 24 – Multi-GPU

Lecture 24.3 – Multi GPU Introduction II

Objective

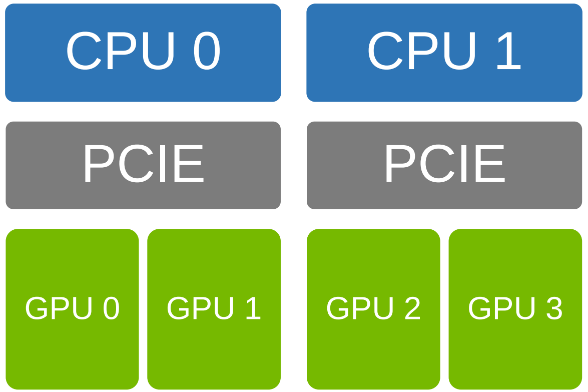
- To learn the important concepts involved on the use of systems with Multiple GPU's
  - Memory transfers
  - NVLINK

Memory transfers in a Multi GPU setup



- Involves transferring memory regions from one device to another, e.g. GPU0 to GPU1.
- There are three ways to do it:
  - Fully explicit memory transfers using cudaMemcpyPeerAsync, which requires the specification of the peer devices.
  - Partially explicit memory transfers using cudaMemcpy, relying on the unified address system.
  - Implicit peer memory access performed by the driver, without the need of explicit transfers.
- Not all three possibilities are available in every system.

Explicit peer memory transfers CUDA host API functions



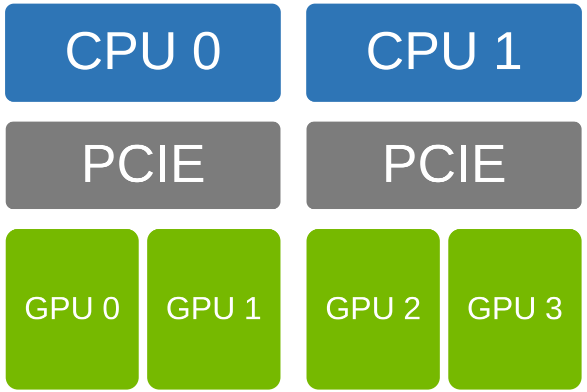
- cudaMemcpyPeerAsync()
  - Six parameters
    - Pointer to destination region on the destination device
    - Destination device id
    - Pointer to source region on the source device
    - Source device id
    - Number of bytes copied
    - CUDA stream

Transfer between devices is asynchronous

Example: peer transfer cudaMemcpyPeerAsync

```
float *A0, *A1;
int size;

cudaSetDevice(0);  // Set active device to 0
cudaMalloc((void**) &A0, size);// Allocate memory on device 0

cudaSetDevice(1); // Set active device to 1
cudaMalloc((void**) &A1, size);  // Allocate memory on device 1

 // Initialize region A0 on device 0

cudaMemcpyPeerAsync(A1, 1, A0, 0, size, stream); // Copy the data on A0 on device 0 to the region A1 on device 1

cudaSetDevice(1);  // Set the device for kernel execution
kernel<<<gridDim, blockDim, 0, stream>>>(A1); // Perform computations on A1

cudaFree(A0);// Free A0 region
cudaFree(A1);  // Free A1 region
```

Explicit peer memory transfers CUDA host API functions



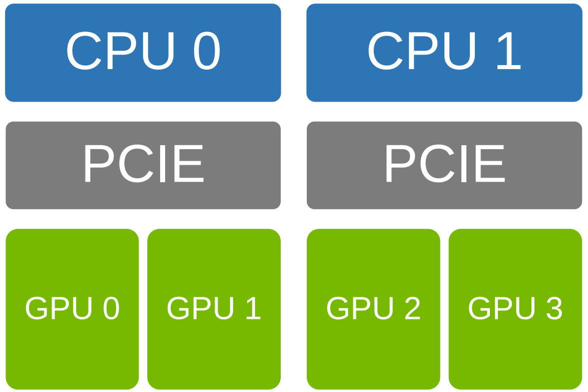
- If the flag cudaDevAttrUnifiedAddressing is set to 1, then you may copy regions between devices using the traditional cudaMemcpy API function, setting the copy kind to cudaMemcpyDefault.

- To check if the flag is set you can use the API function:
  - cudaDeviceGetAttribute()

Example: peer transfer cudaMemcpy

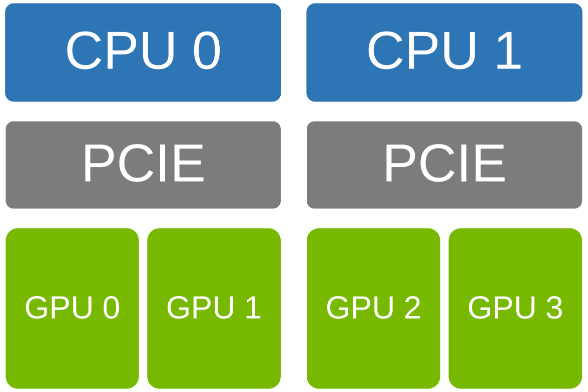
```
float *A0, *A1;
int size;
int unifiedAddr_flag0 = 0;
int unifiedAddr_flag1 = 0;

cudaSetDevice(0);// Set active device to 0
cudaMalloc((void**) &A0, size);// Allocate memory on device 0
cudaSetDevice(1);// Set active device to 1
cudaMalloc((void**) &A1, size);// Allocate memory on device 1

// Initialize region A0 on device 0

cudaDeviceGetAttribute(unifiedAddr_flag0, cudaDevAttrUnifiedAddressing, 0); // Check if unified addressing is
available on dev 0
cudaDeviceGetAttribute(unifiedAddr_flag1, cudaDevAttrUnifiedAddressing, 1); // Check if unified addressing is
available on dev 0

if( unified_addressing_flag0 == 1 && unified_addressing_flag1 == 1 )
 cudaMemcpy(A1, A0, size, cudaMemcpyDefault);  // Copy the data on A0 on device 0 to the region A1 on device
1
else
 // Throw error indicating the copy couldn't be performed
```

Implicit peer memory access



- To query if implicit peer memory access are enabled use:
- cudaDeviceCanAccessPeer:
  - Three parameters:
    - Int pointer to place to store the flag.
    - Device id of device trying to access peer
    - Device id of peer device
  - This call is not symmetric, meaning that if the canAccess flag is set to 1 for deviceA and deviceB, it may not be set to 1 for deviceB and deviceA.

Enabling and disableing implicit peer memory access



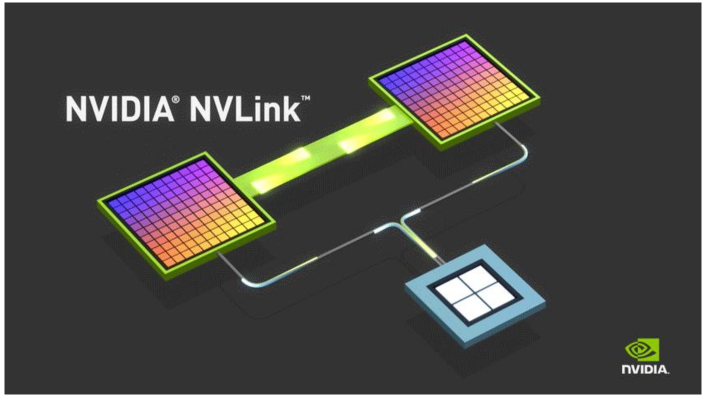
- cudaDeviceEnablePeerAccess:
  - Two parameters:
    - Device id to enable access to from the current active device.
    - Int flag set to 0.
  - This call is not symmetric.
  - Returns error cudaErrorInvalidDevice if not possible.
- cudaDeviceDisablePeerAccess:
  - One parameter:
    - Device id to disable access to from the current device.
  - This call is not symmetric.

Example: implicit peer access

```
float *ptrA;  // Pointer to memory region on device devA

int devA;
int devB;
int BcanAccessA = 0;

cudaError_t error;

cudaDeviceCanAccessPeer(&BcanAccessA, devB, devA);  // Check if devB can access devA memory.

cudaSetDevice(devB);  // Set the current active device to devB
if(BcanAccessA == 0)
 error = cudaDeviceEnablePeerAccess(devA, 0);  // Enable peer accesses to devA memory

if(error == cudaSuccess) {
  kernel<<<gridDim, blockDim, 0, stream>>>(ptrA);  // Access ptrA on device devA from device devB
}
cudaDeviceDisablePeerAccess(devA); // Disable peer access to devA, this call is not needed.
```

NVLINK

– Is a proprietary interconnect technology developed by NVIDIA
– Provides higher memory bandwidth communication between GPUS than PCIE communication
– NVLINK on the Tesla V100 delivers a 300 GB/s communication data rate, whereas the typical PCIE 3.0 link delivers only 32 GB/s



GPU Teaching Kit

Accelerated Computing

GPU Teaching Kit

Accelerated Computing

Module 24 – Multi-GPU

Lecture 24.4 – Multi GPU patterns with OpenMP & Cooperative Groups

Objective

- To learn the important concepts involved on how to use parallel frameworks in systems with Multiple GPU’s
  - Batch processing
  - Cooperative patterns algorithms
  - OpenMP
  - NVIDIA Cooperative Groups

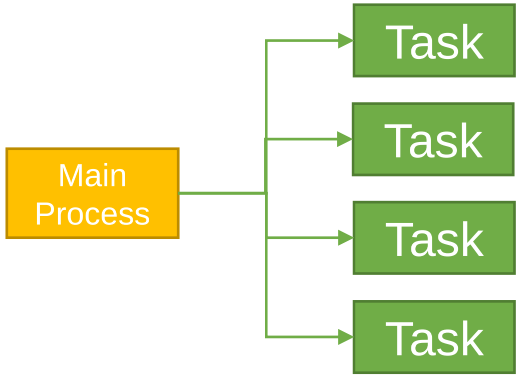
Common parallel patterns in a Multi-GPU environment

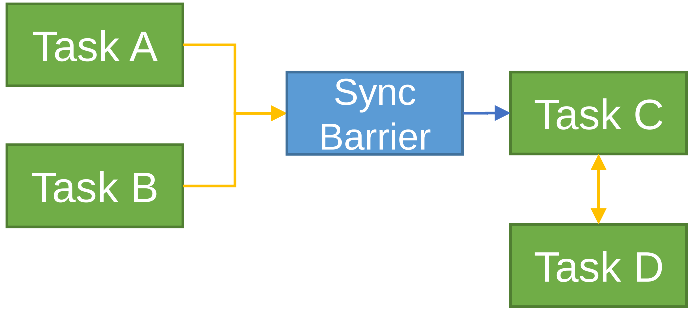
Batch processing:

– Execute the same independent task multiple times with different data.

Cooperative patterns:

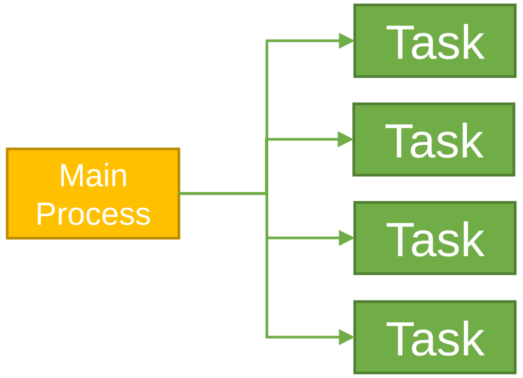
– Tasks need to cooperate between each other to collectively reach a goal.





Batch processing

- Is an embarrassingly parallel pattern.
- With enough data it is usual we can achieve 100% usage of the compute resources.
- It’s common on video and image processing applications, where we need to apply the same operation to lots of different data.

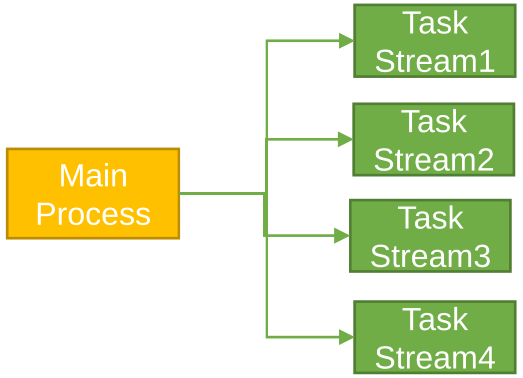


Batch processing

Typical operations to accomplish batch processing:

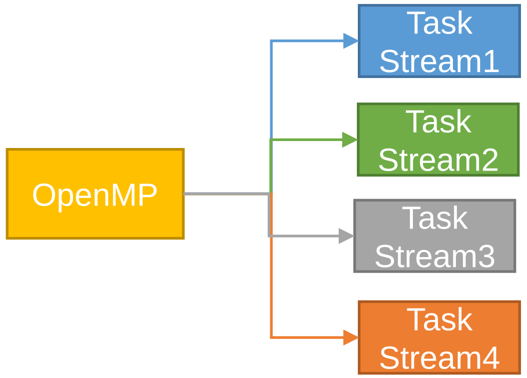
1. Get number of available devices.
2. Considering the number of devices and number of desired tasks allocate, initialize and copy the memory need it by the algorithm.
3. Create CUDA streams for each of the tasks to be executed concurrently.
4. Launch in parallel the kernel.

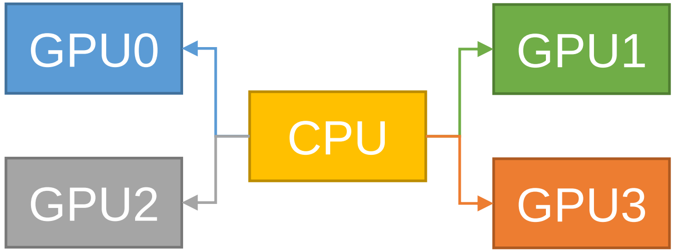
* Remember to set the device at the beginning of each group of operations.



Batch processing example with OpenMP

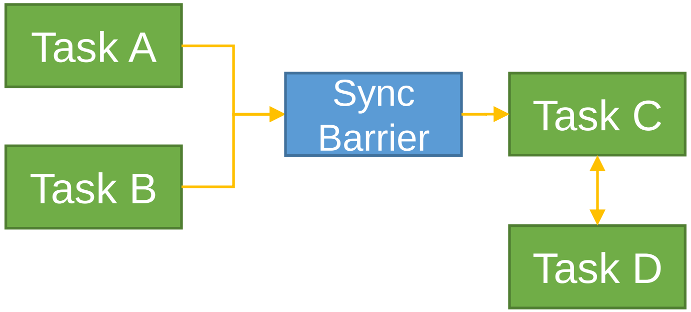
```
  int deviceCount;
  cudaGetDeviceCount(& deviceCount);
  std::vector<cudaStream_t> streams(deviceCount);
#pragma omp parallel for num_threads(deviceCount)
  for(int dev = 0; dev < deviceCount; ++dev) {
      cudaSetDevice(dev);
      cudaStreamCreate(&streams[i]);
      // Allocate, initialize and transfer memory
  }
#pragma omp parallel for num_threads(deviceCount)
  for(int dev = 0; dev < deviceCount; ++dev) {
      cudaSetDevice(dev);
      kernel<<<gridDim, blockDim, streams[i]>>>(…);
  }
```





Cooperative patterns

- Might have unavoidable syncing points causing tasks to wait and thus wasting compute resources.
- In some cases even when massive amounts of input data it might not reach 100% resource usage.
- It's common on applications with steps to reach a goal like iterative algorithms.

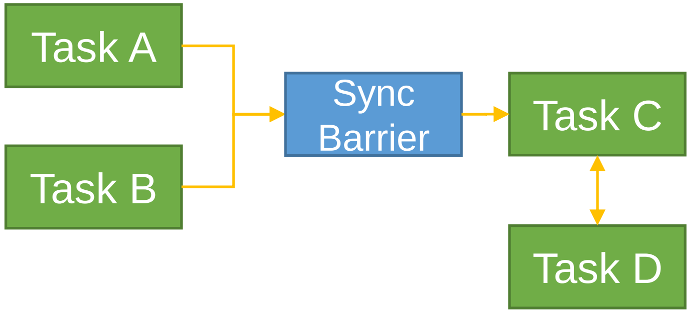


Cooperative patterns

– With cooperative patterns there is no single fit solution like with batch processing.

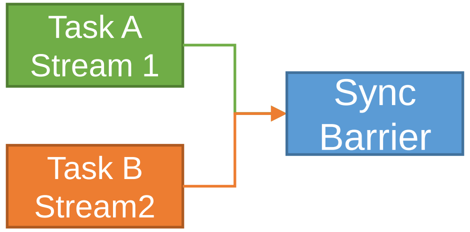
– Thus the process consists in a loop of:

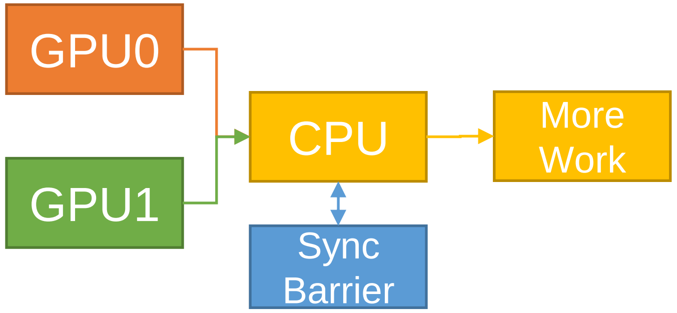
1. Launching the code in parallel.
2. Profiling it.
3. Analyzing and removing bottlenecks.



Syncing patterns on different streams with OpenMP

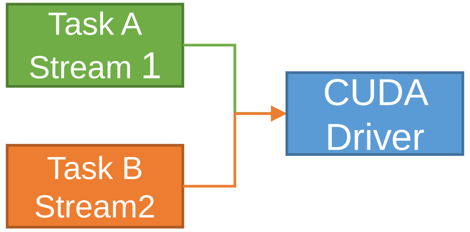
```
  std::vector<cudaStream_t> streams;
  // Initialization of the streams on each device.
#pragma omp parallel
{
  // Launch the different kernels on the streams.
#pragma omp for num_threads(streams.size())
  for(auto& stream : streams)
      cudaStreamSynchornize(stream);
#pragma omp barrier
}
```





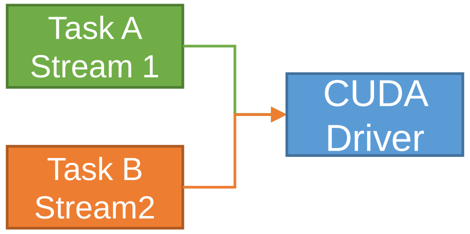
Multi-GPU Syncing patterns with Cooperative Groups

- Cooperative Groups is a C++-CUDA high level abstraction to perform syncing across different parallel granularities (Threads, Blocks, Grids, and Devices).
- Multi-GPU syncing with cooperatives groups requires:
  - Devices with the exact same compute capability.
  - Compute capability of 6 or higher.
  - Executing the same kernel across all devices.



Multi-GPU Syncing patterns with Cooperative Groups

- To enable the use of Cooperative Groups we need to include the file cooperative_groups.h and use the namespace cooperative_groups.
- The kernels needs to be compiled using separate compilation and then linked with the –rdc=true flag.
- You also need to ensure that MPS is disabled and CU_DEVICE_ATTRIBUTE_COOPERATIVE_MULTI_DEVICE_LAUNCH is set in the device properties using cuDeviceGetAttribute the API function.



Multi-GPU Syncing patterns with Cooperative Groups

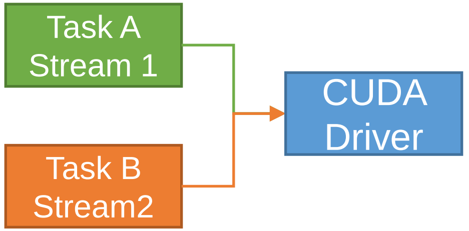
Launching a kernel with Multi-GPU syncing and Cooperative Groups requires using the API function:

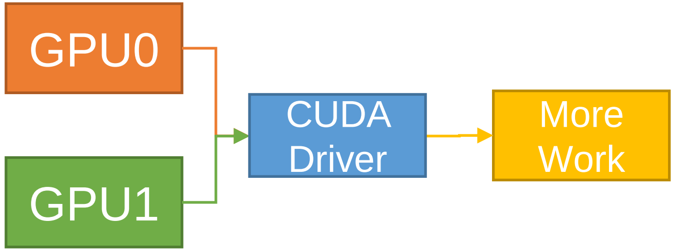
- cudaLaunchCooperativeKernelMultiDevice:
  - The first parameter is an array of CUDA_LAUNCH_PARAMS, where except for kernel params and the stream, all other fields needs be the same.
  - The number of devices to use.

```
typedef struct CUDA_LAUNCH_PARAMS_st {
  CUfunction function;      // Kernel to launch.
  unsigned int gridDimX;   // Grid dimensions.
  unsigned int gridDimY;
  unsigned int gridDimZ;
  unsigned int blockDimX; // Block dimensions.
  unsigned int blockDimY;
  unsigned int blockDimZ;
  unsigned int sharedMemBytes; // Shared memory size.
  CUstream hStream;         // Stream to perform the work
  void **kernelParams;      // Kernel parameters
} CUDA_LAUNCH_PARAMS;
```

Multi-GPU Syncing patterns with Cooperative Groups

```
#include <cooperative_groups.h>
using namespace cooperative_groups;
void __global__ kernel(…) {
    // Work
    multi_grid_group multi_grid = this_multi_grid();
    multi_grid.sync();
   // Work
}
//  Work
cudaLaunchCooperativeKernelMultiDevice(…);
```





GPU Teaching Kit

Accelerated Computing

GPU Teaching Kit

Accelerated Computing

Module 24 – Multi-GPU

Lecture 24.5 – Multi-GPU Heat Equation

Objective

- To learn how to parallelize a basic Finite Difference scheme for the Heat Equation using Multiple GPUs in 2D
  - Learn what is the Heat Equation
  - Learn about the Finite Difference Method (FDM)
  - Learn caveats of synchronization

Heat Equation in 2D

– A partial differential equation describing heat diffusion over time on a surface given by the formula:

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right)$$

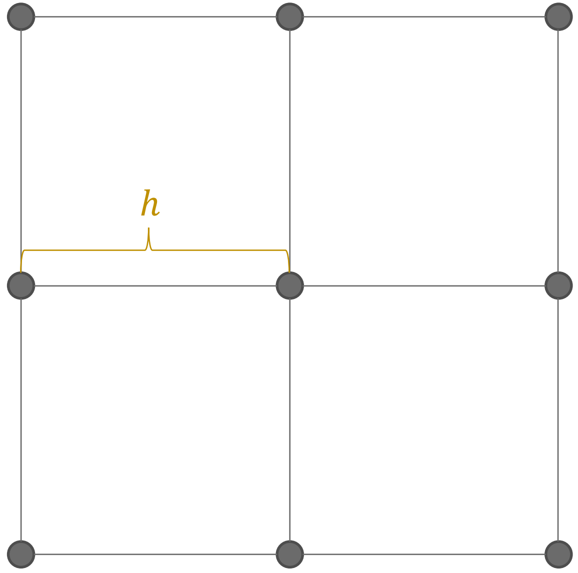
where, T is the temperature in the surface, α the heat diffusion coefficient, t time and x, y the spatial variables

Finite Difference Method (FDM)

- Numerical method typically used for solving differential equations on a grid, which discretizes the equation by replacing derivatives with differences between numbers
- If a grid with equidistant points is separated by a distance h, the following are the approximations:

$$\frac{dy}{dx}(x_0) = \frac{y(x_0+h) - y(x_0)}{h}$$

$$\frac{d^2y}{dx^2}(x_0) = \frac{y(x_0+h) - 2y(x_0) - y(x_0-h)}{h^2}$$



Heat Equation and FDM

– Using finite differences here is the discretization for the heat equation:

$$T_{i,j}^{n+1} = T_{i,j}^{n} + \hat{\alpha}(DTx + DTy)$$

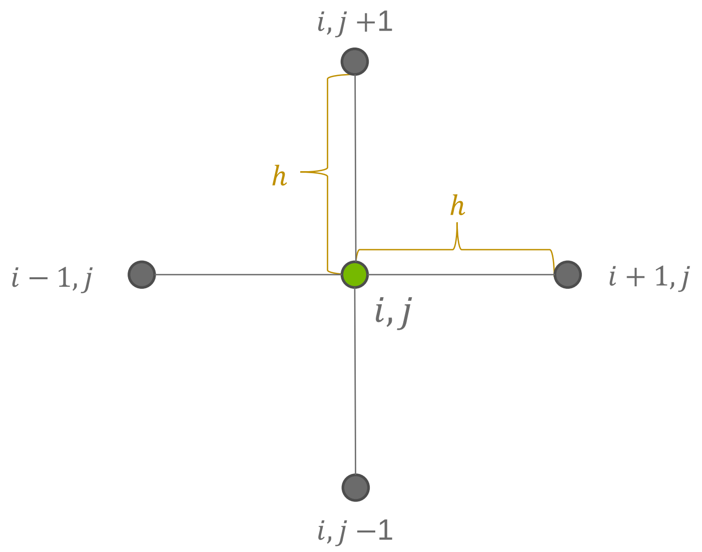
$$DTx = T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n}$$

$$DTy = T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n}$$

where n refers to the time iteration and $\hat{\alpha}$ is given by:

$$\hat{\alpha} = \frac{g}{h^2}\alpha$$

and g is the size of the time step



Parallelizing the Heat equation across Multiple GPUs

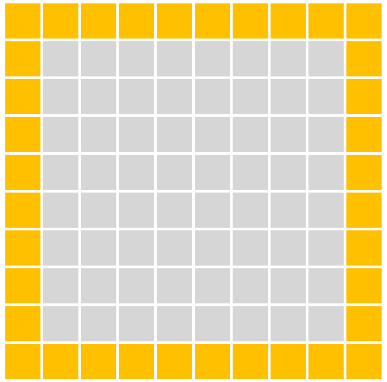
Description of the problem with boundary conditions

- For the equation to have a unique solution there exists the need to establish boundary conditions
- In this case the boundary conditions describe a heat source acting on the boundary
- The full problem is given by:

$$\frac{\partial T}{\partial t} = \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right)$$

$$T(x,y,t) = g(x,y), \forall\,(x,y) \in \partial\Omega$$

where $\partial\Omega$ is the boundary and $g(x,y)$ is known function



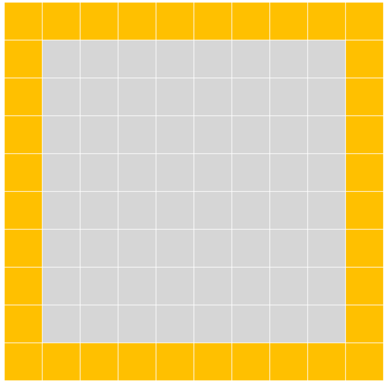
⬜ $int\ \Omega$

🟨 $\partial\Omega$

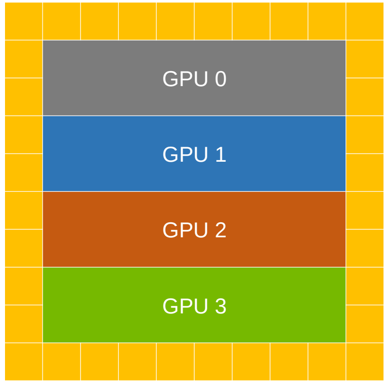
$$\Omega = int\ \Omega\ \cup \partial\Omega$$

Parallelization Strategy for Multiple GPUs

- The idea is to partition the grid horizontally into subdomains of roughly the same size
- As shown in the figure, each subdomain marked in grey, blue, red and green will run on GPUs 0-3
- Yellow denotes the boundary and will remain unchanged due to the boundary conditions



Unpartitioned Grid



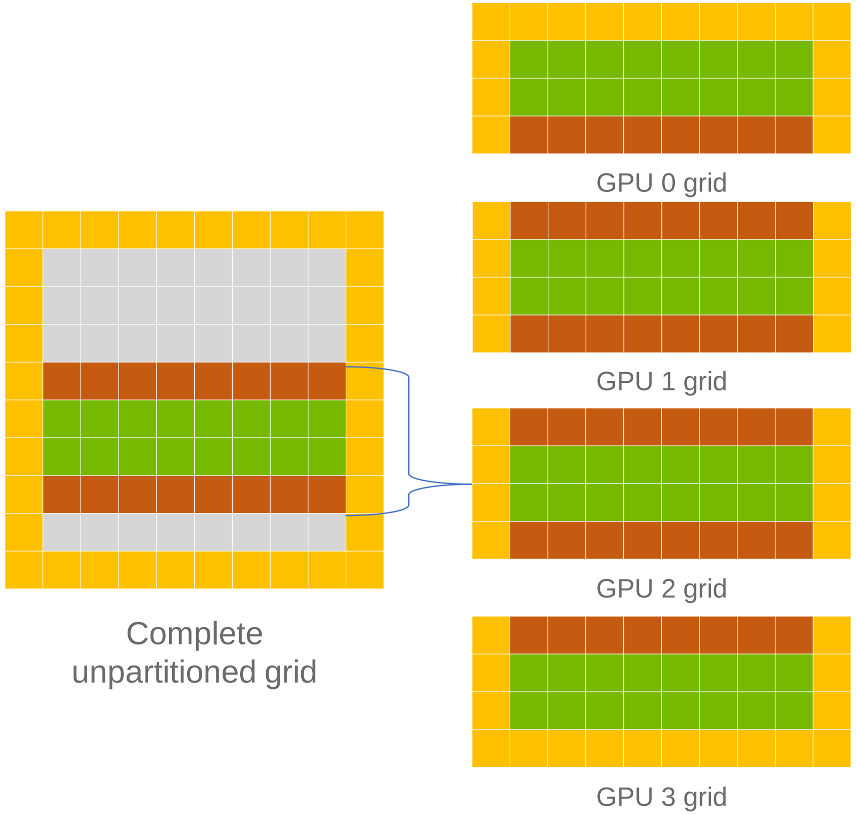
Partitioned Grid

Memory partition of the grid

- Recall that in order to compute a grid point, we need information from the points at:

  $(i,j),(i-1,j),(i+1,j),(i,j-1),(i,j+1)$

- Each GPU must have the grid points required to update the time step (the green rows), boundary region (yellow) and halo regions (top or bottom orange rows)
- After performing a time step update, the devices must synchronize the halos between each other



Outline of the full algorithm

- Initialize the boundary with boundary conditions
- Partition the domain
- Transfer each subdomain
- Repeat n times:
  - Compute the finite difference approximation in slide 4 in each device
  - Create a barrier for waiting for the computations to be finished
  - Transfer the upper and lower halos to the upper and lower devices respectively and synchronize
- Transfer back the result to the host

Notes on the CUDA kernel

- In occasions where the amount of operations performed by individual threads is not high -like in the case of the heat equation, is beneficial to loop on the operations to take advantage that the thread is executing instead of launching a new thread
- Usually this is accomplished by setting a smaller grid size than needed, for example:

$$gridSize = \left\lfloor \frac{n - 1 + blockDim\ * nit}{blockDim\ * nit} \right\rfloor$$

where n is the number of elements and nit the number of iterations

Notes on the CUDA kernel

Vector add kernel where each thread computes only one add operation:

```
__global__ void add(double* a, double* b, int n) {
  int tid = threadIdx.x + blockDim.x * blockIdx.x;
  if(tid < n)
    a[tid] += b[tid]
}
```

Vector add kernel where each thread computes multiple add operations:

```
__global__ void add(double* a, double* b) {
  int tid = threadIdx.x + blockDim.x * blockIdx.x;
  for( ; tid < n; tid += blockDim.x * gridDim.x)
    a[tid] += b[tid]
}
```

GPU Teaching Kit

Accelerated Computing