UNIVERSITY OF WOLVERHAMPTON

6CS005 High Performance Computing

Lecture 11

More on GPU Programming

Overview

- **Computational Thinking in Parallel Programming**
- **MultiGPUs**
- **CUDA Using Python**
  - **pyCUDA**
  - **Numba**

Computational Thinking in Parallel Programming

- Parallel computing requires that
  - The problem can be decomposed into sub-problems that can be safely solved at the same time
  - The programmer structures the code and data to solve these sub-problems concurrently
- The goals of parallel computing are
  - To solve problems in less time (strong scaling), and/or
  - To solve bigger problems (weak scaling), and/or
  - To achieve better solutions (advancing science)

The problems must be large enough to justify parallel computing and to exhibit exploitable concurrency.

Shared Memory vs. Message Passing

- We have focused on shared memory parallel programming
  - This is what CUDA (and OpenMP, OpenCL) is based on
  - Future massively parallel microprocessors are expected to support shared memory at the chip level
- The programming considerations of message passing model is quite different!
  - However, you will find parallels for almost every technique you learned in this course
  - Need to be aware of space-time constraints

Data Sharing

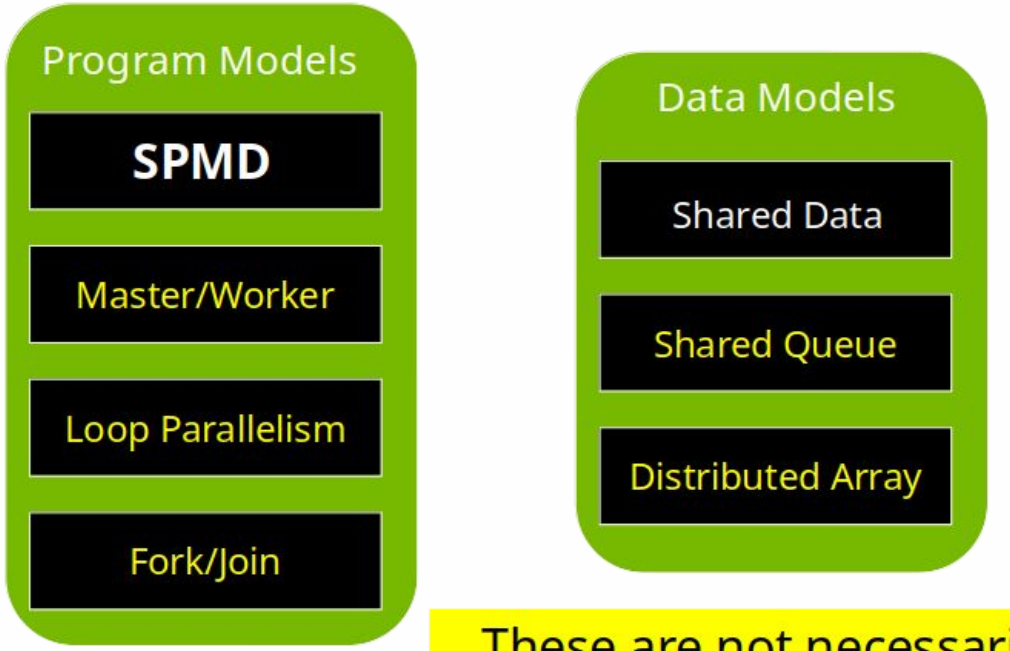
- Data sharing can be a double-edged sword
  - Excessive data sharing drastically reduces advantage of parallel execution
  - Localized sharing can improve memory bandwidth efficiency
- Efficient memory bandwidth usage can be achieved by synchronizing the execution of task groups and coordinating their usage of memory data
  - Efficient use of on-chip, shared storage and datapaths
- Read-only sharing can usually be done at much higher efficiency than read-write sharing, which often requires more synchronization
- Many:Many, One:Many, Many:One, One:One

Synchronization

- Synchronization == Control Sharing
- Barriers make threads wait until all threads catch up
- Waiting is lost opportunity for work
- Atomic operations may reduce waiting
    - Watch out for serialization

- Important: be aware of which items of work are truly independent

Parallel Programming Coding Styles – Program and Data Models



These are not necessarily mutually exclusive.

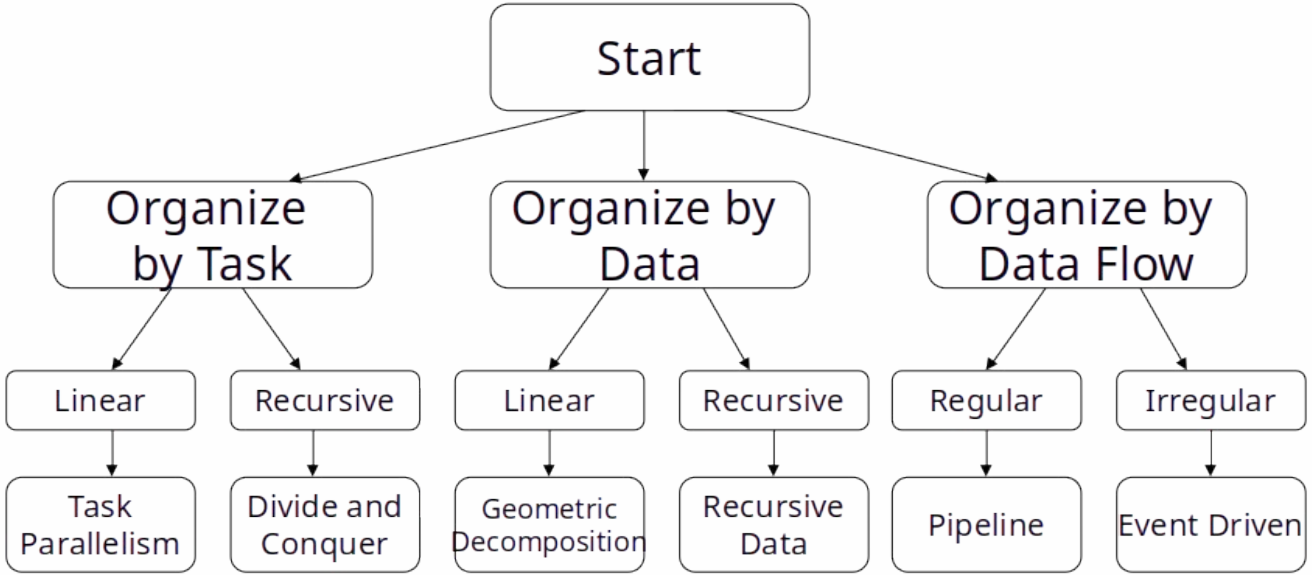
Program Models

- SPMD (Single Program, Multiple Data)
  - All PE's (Processor Elements) execute the same program in parallel, but has its own data
  - Each PE uses a unique ID to access its portion of data
  - Different PE can follow different paths through the same code
  - This is essentially the CUDA Grid model (also OpenCL, MPI)
  - SIMD is a special case – WARP used for efficiency
- Master/Worker
- Loop Parallelism
- Fork/Join

Program Models

- SPMD (Single Program, Multiple Data)
- Master/Worker (OpenMP, OpenACC, TBB)
  - A Master thread sets up a pool of worker threads and a bag of tasks
  - Workers execute concurrently, removing tasks until done
- Loop Parallelism (OpenMP, OpenACC, C++AMP)
  - Loop iterations execute in parallel
- Fork/Join (Posix p-threads)
  - Most general, generic way of creation of threads

Algorithm Structure



Mattson, Sanders, Massingill, *Patterns for Parallel Programming*

More on SPMD

- Dominant coding style of scalable parallel computing
  - MPI code is mostly developed in SPMD style
  - Many OpenMP code is also in SPMD (next to loop parallelism)
  - Particularly suitable for algorithms based on task parallelism and geometric decomposition.
- Main advantage
  - Tasks and their interactions visible in one piece of source code, no need to correlated multiple sources

SPMD is by far the most commonly used pattern for structuring massively parallel programs.

Typical SPMD Program Phases

- Initialize
  - Establish localized data structure and communication channels
- Obtain a unique identifier
  - Each thread acquires a unique identifier, typically range from 0 to N-1, where N is the number of threads.
  - Both OpenMP and CUDA have built-in support for this.
- Distribute Data
  - Decompose global data into chunks and localize them, or
  - Sharing/replicating major data structure using thread ID to associate subset of the data to threads
- Run the core computation
  - More details in next slide…
- Finalize
  - Reconcile global data structure, prepare for the next major iteration

Core Computation Phase

- Thread IDs are used to differentiate behavior of threads
  - Use thread ID in loop index calculations to split loop iterations among threads
    - Potential for memory/data divergence
  - Use thread ID or conditions based on thread ID to branch to their specific actions
    - Potential for instruction/execution divergence

Both can have very different performance results and code complexity depending on the way they are done.

Typical SPMD Program Phases

- Initialize
  - Establish localized data structure and communication channels
- Obtain a unique identifier
  - Each thread acquires a unique identifier, typically range from 0 to N-1, where N is the number of threads.
  - Both OpenMP and CUDA have built-in support for this.
- Distribute Data
  - Decompose global data into chunks and localize them, or
  - Sharing/replicating major data structure using thread ID to associate subset of the data to threads
- Run the core computation
  - More details in next slide…
- Finalize
  - Reconcile global data structure, prepare for the next major iteration

Making Science Better, not just Faster

or… in other words:

There will be no Nobel Prizes or Turing Awards awarded for “just recompile” or using more threads

Conclusion: Three Options

- Good: “Accelerate” Legacy Codes
  - Recompile/Run
  - Call CUBLAS/CUFFT/thrust/matlab/PGI pragmas/etc.
  - => good work for domain scientists (minimal CS required)
- Better: Rewrite / Create new codes
  - Opportunity for clever algorithmic thinking
  - => good work for computer scientists (minimal domain knowledge required)
- Best: Rethink Numerical Methods & Algorithms
  - Potential for biggest performance advantage
  - => Interdisciplinary: requires CS and domain insight
  - => Exciting time to be a computational scientist

Think, Understand… then, Program

- Think about the problem you are trying to solve
- Understand the structure of the problem
- Apply mathematical techniques to find solution
- Map the problem to an algorithmic approach
- Plan the structure of computation
  - Be aware of in/dependence, interactions, bottlenecks
- Plan the organization of data
  - Be explicitly aware of locality, and minimize global data
- Finally, write some code! (this is the easy part)

Future Studies

- More complex data structures
- More scalable algorithms and building blocks
- More scalable math models

- Thread-aware approaches
  - More available parallelism
- Locality-aware approaches
  - Computing is becoming bigger, and everything is further away

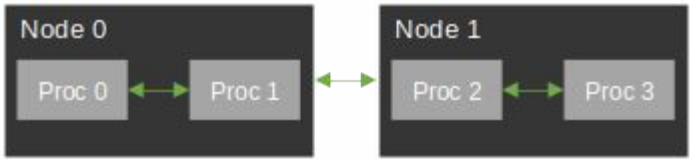
UNIVERSITY OF WOLVERHAMPTON

MultiGPU

Differences between process and threaded parallelism

Process parallelism

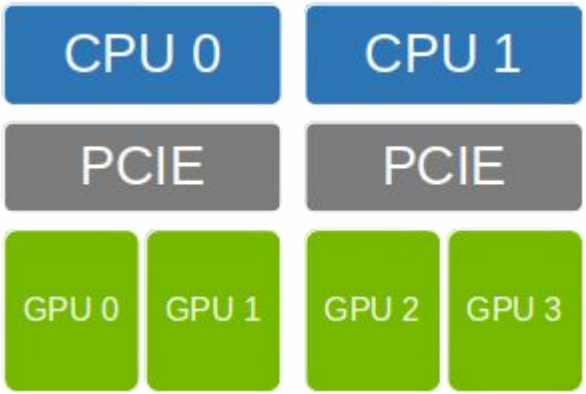
- Uses local memory forcing each parallel worker to have its own memory space.
  - It needs explicit data sharing routines creating communication overhead.
- It's controlled by a main process.
- It can scale to multiple computers.
- Example: MPI



Threaded parallelism

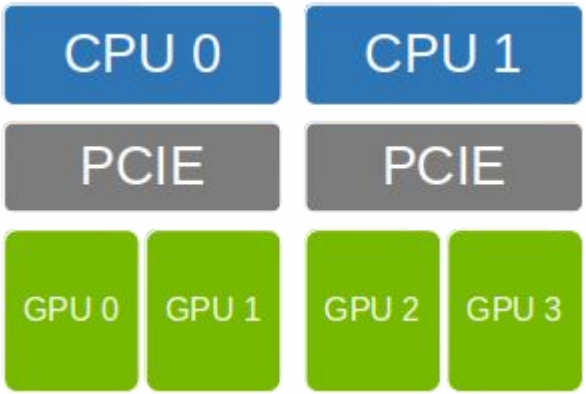
- Uses shared memory, enabling all parallel workers to access the same memory space.
- It's controlled by a main thread, with all threads running on a single process.
- It can't scale beyond a single computer.
- Example: OpenMP

Schematic of a Multi GPU system



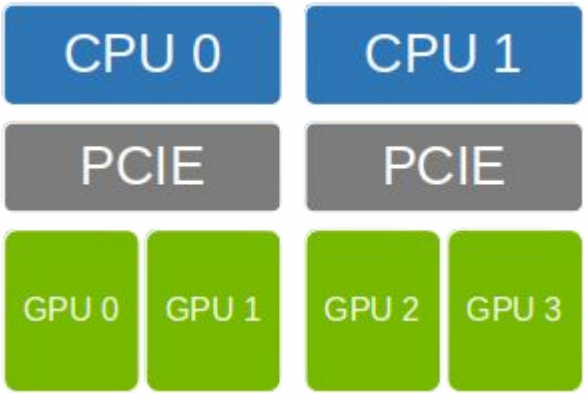
- The figure represents a system with 2 CPU's and 4 GPU's.
- GPU's are numbered from 0 to n-1, where n is the number of GPU's.
- The CUDA driver always starts with a default active device.
- There are two broad types of Multi GPU communication:
  - Through the PCIE bus
  - Through NVLINK

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

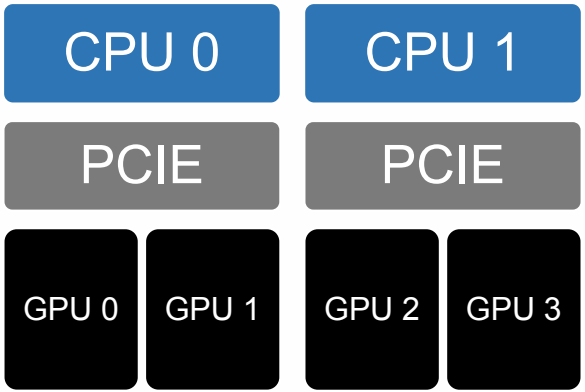
Putting it all together, vecAdd

```
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
```

Putting it all together, vecAdd

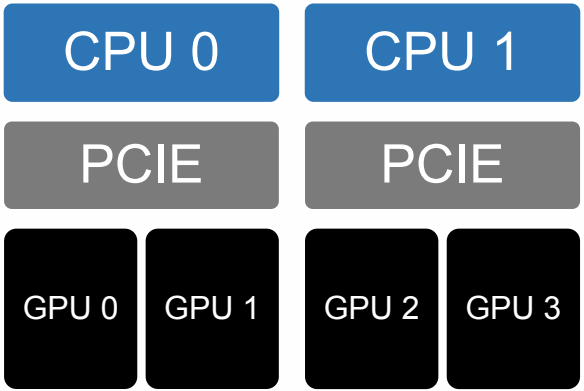
```
// Memory initialization on the Host and memory transfers

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);
```

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



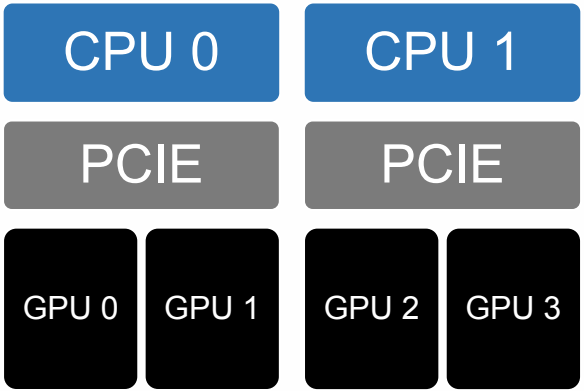
- cudaMemcpyPeerAsync()
  - Six parameters
    - Pointer to destination region on the destination device
    - Destination device id
    - Pointer to source region on the source device
    - Source device id
    - Number of bytes copied
    - CUDA stream

  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



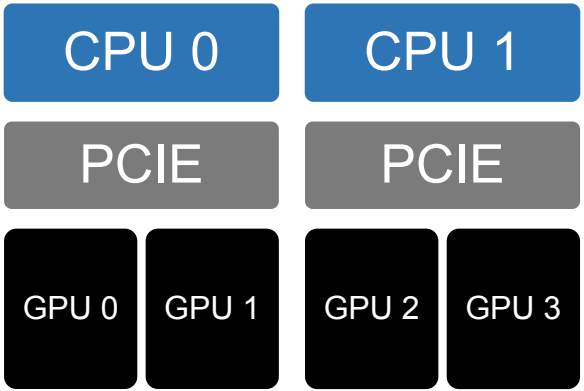
- If the flag cudaDevAttrUnifiedAddressing is set to 1, then you may copy regions between devices using the traditional cudaMemcpy API function, setting the copy kind to cudaMemcpyDefault.

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

Example: peer transfer cudaMemcpy

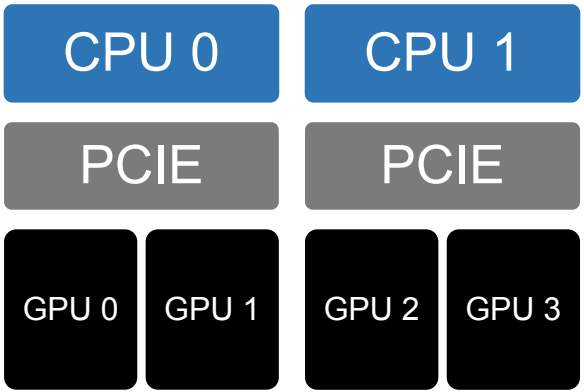
```
float *A0, *A1;
int size;
int unifiedAddr_flag0 = 0;
int unifiedAddr_flag1 = 0;

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 disabling implicit peer memory access



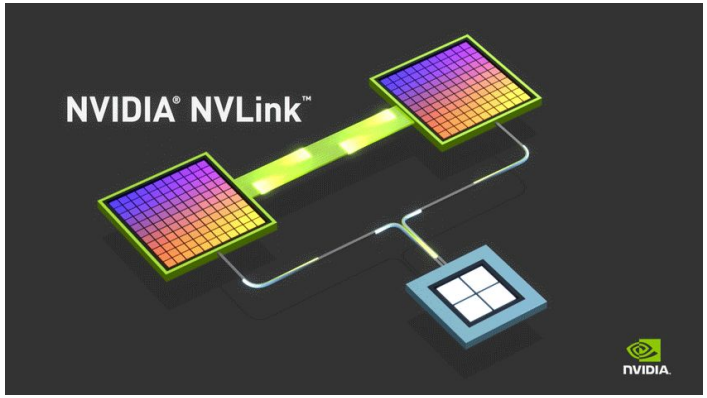
- cudaDeviceEnablePeerAccess:
  - Two parameters:
    - Device id to enable access to from the current active device.
    - Int flag set to 0.
  - This call is not symmetric.
  - Returns error cudaErrorInvalidDevice if not possible.
- cudaDeviceDisablePeerAccess:
  - One parameter:
    - Device id to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

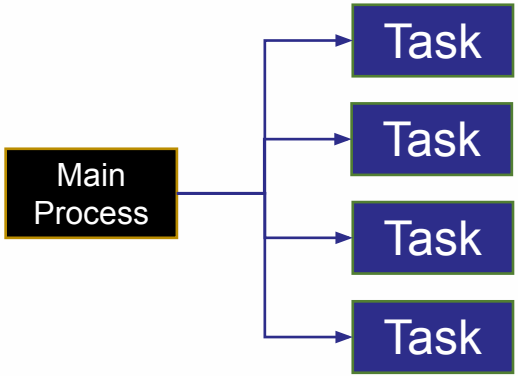
NVLINK

- 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



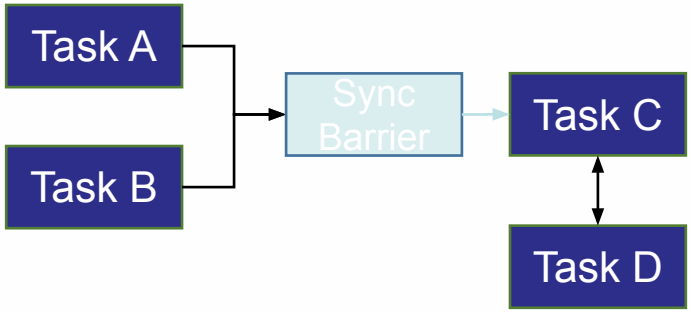
Batch processing:

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



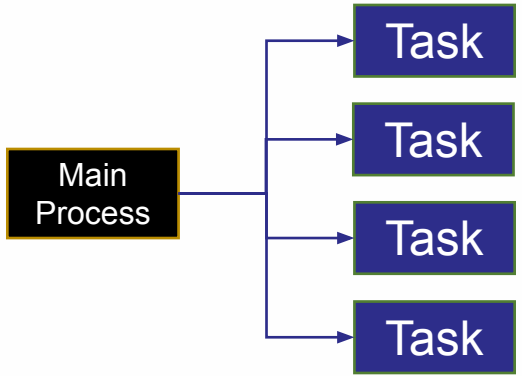
Cooperative patterns:

• 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.

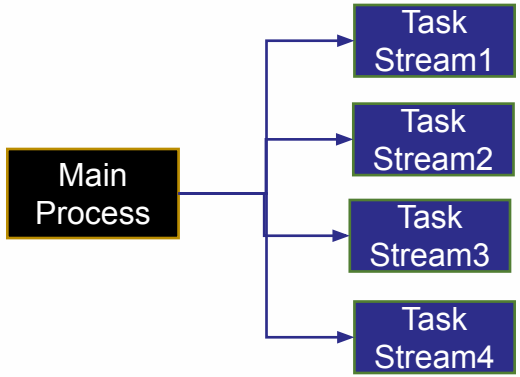


Batch processing

Typical operations to accomplish batch processing:

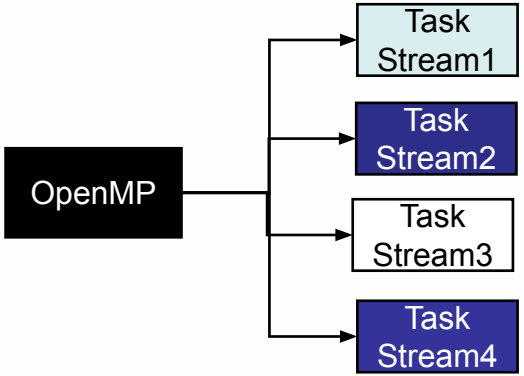
1. Get number of available devices.
2. Considering the number of devices and number of desired tasks allocate, initialize and copy the memory need it by the algorithm.
3. Create CUDA streams for each of the tasks to be executed concurrently.
4. Launch in parallel the kernel.

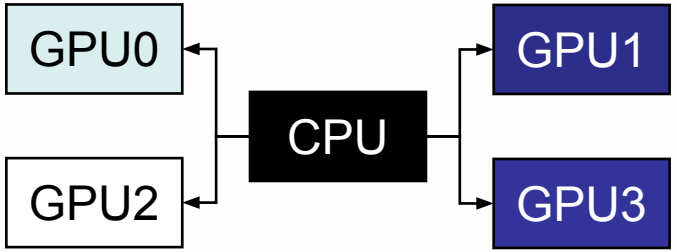
* Remember to set the device at the beginning of each group of operations.



Batch processing example with OpenMP

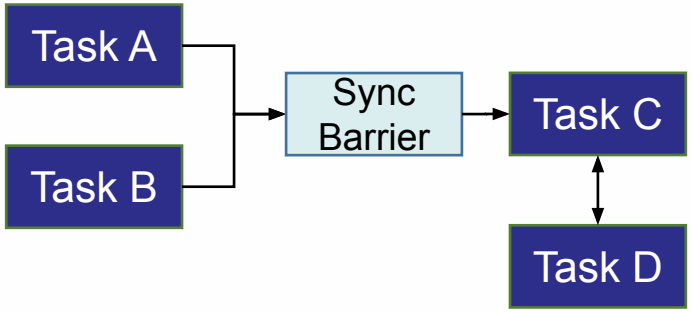
```
int deviceCount;
cudaGetDeviceCount(& deviceCount);
std::vector<cudaStream_t> streams(deviceCount);
#pragma omp parallel for num_threads(deviceCount)
for(int dev = 0; dev < deviceCount; ++dev) {
    cudaSetDevice(dev);
    cudaStreamCreate(&streams[i]);
    // Allocate, initialize and transfer memory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





Cooperative patterns

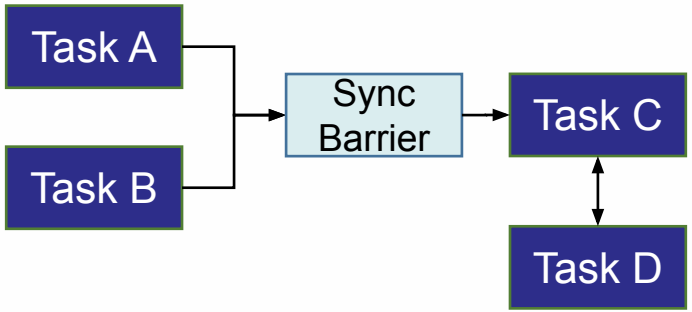
- Might have unavoidable syncing points causing tasks to wait and thus wasting compute resources.
- In some cases even when massive amounts of input data it might not reach 100% resource usage.
- It's common on applications with steps to reach a goal like iterative algorithms.



Cooperative patterns

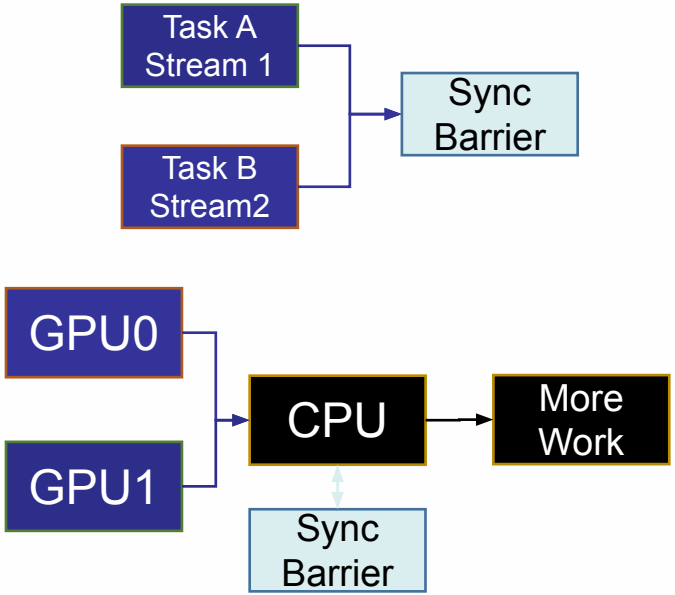
- With cooperative patterns there is no single fit solution like with batch processing.
- Thus the process consists in a loop of:

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



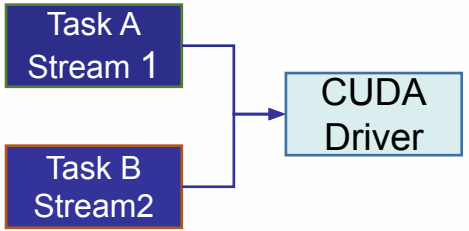
Syncing patterns on different streams with OpenMP

```
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
}
```



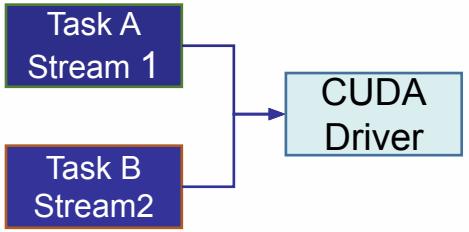
Multi-GPU Syncing patterns with Cooperative Groups

- 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

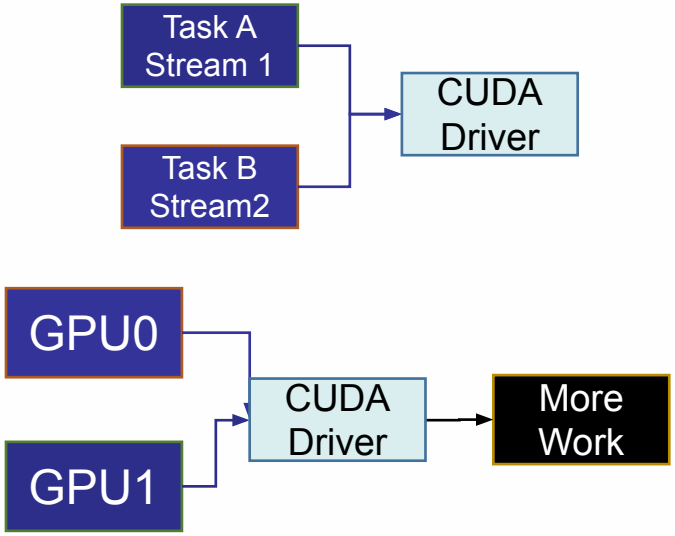
Launching a kernel with Multi-GPU syncing and Cooperative Groups requires using the API function:

- 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(…);
```



UNIVERSITY OF WOLVERHAMPTON

CUDA Using Python

CUDA C/C++ vs. pyCUDA vs. Numba

CUDA C/C++:

- The most common, performant, and flexible way to utilize CUDA
- Accelerates C/C++ applications

pyCUDA:

- Exposes the entire CUDA C/C++ API
- Is the most performant CUDA option available for Python
- Requires writing C code in your Python, and in general, a lot of code modifications

CUDA C/C++ vs. pyCUDA vs. Numba

Numba:

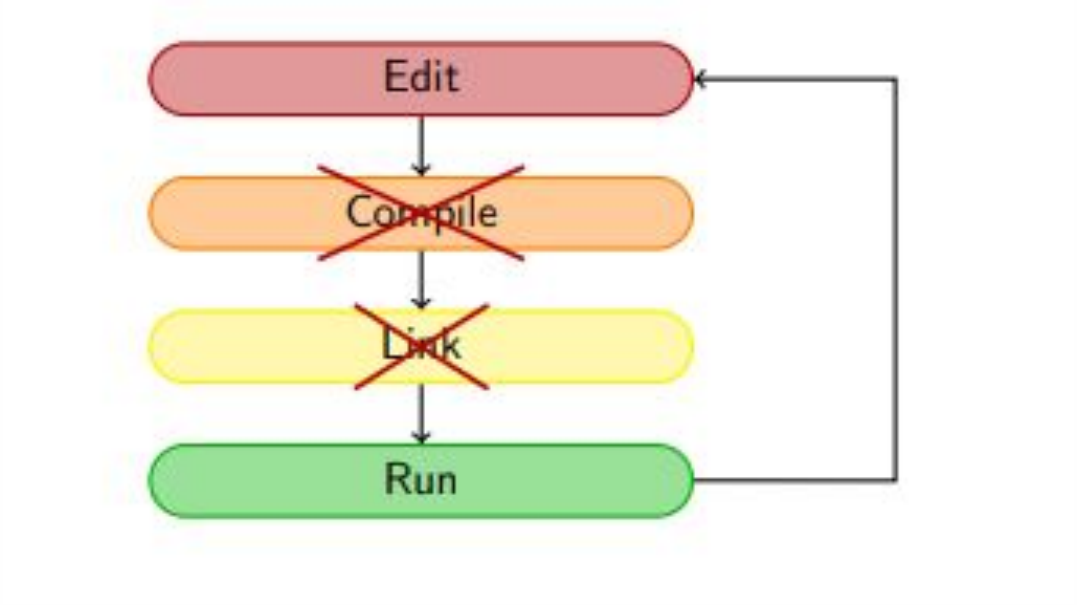
- Potentially less performant than pyCUDA
- Does not (yet?) expose the entire CUDA C/C++ API
- Still enables massive acceleration, often with very little code modification
- Allows developers the convenience of writing code directly in Python
- Also optimizes Python code for the CPU

```
import pycuda.driver as cuda
import pycuda.autoinit
import pycuda.compiler
import numpy as np

a = np.random.randn(4, 4).astype(np.float32) # Create a random 4x4 array
of float32

a_gpu = cuda.mem_alloc(a.nbytes) # Allocate memory on the GPU


cuda.memcpy_htod(a_gpu, a) # Copy data from host to device
```

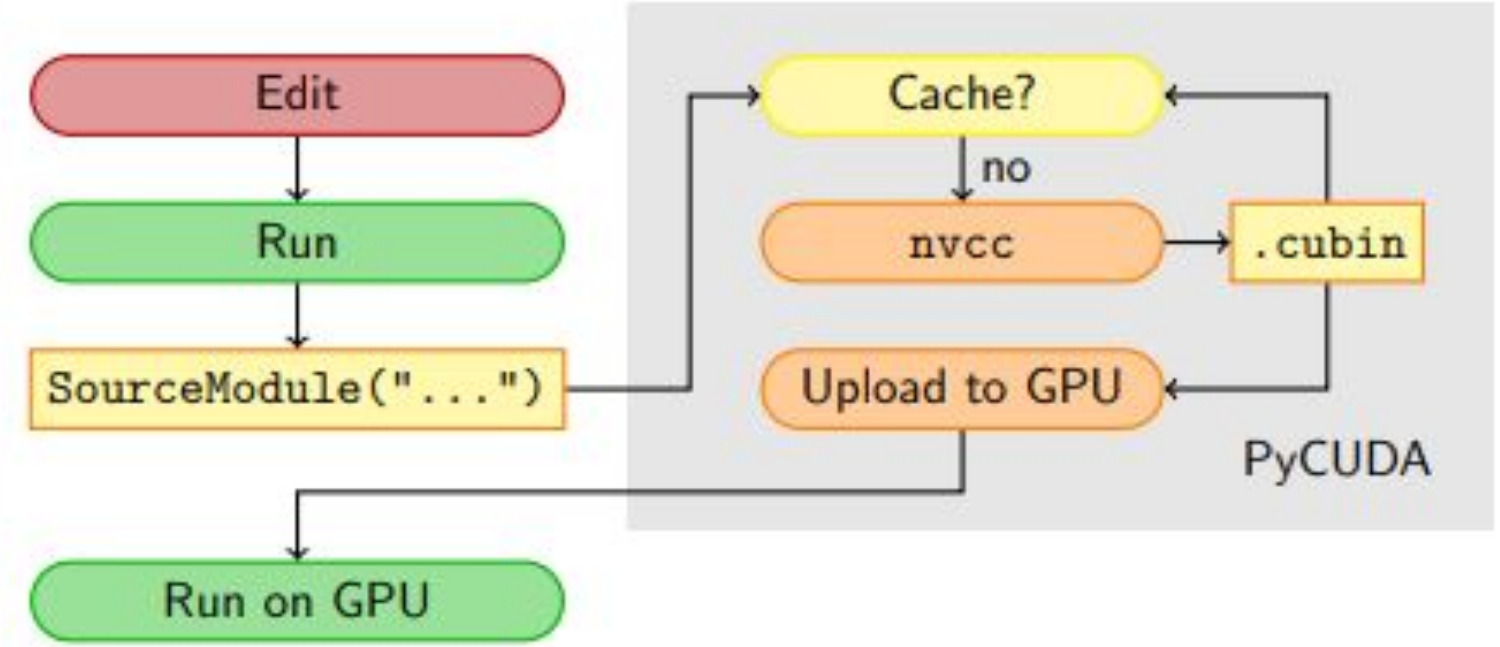
```
# Define the CUDA kernel
mod = pycuda.compiler.SourceModule("""
__global__ void twice(float *a)
{
    int idx = threadIdx.x + threadIdx.y * 4;
    a[idx] *= 2;
}
""")

func = mod.get_function("twice") # Get the kernel function
func(a_gpu, block=(4, 4, 1)) # Execute the kernel
a_doubled = np.empty_like(a) # Prepare to retrieve the result
cuda.memcpy_dtoh(a_doubled, a_gpu)# Copy data from device to host
print(a_doubled)
print(a)
```

pyCUDA: Scripting: Interpreted, not Compiled

Program Creation Workflow:



PyCUDA: Workflow



PyCUDA Philosophy

- Provide complete access
- Automatically manage resources
- Provide abstractions
- Check for and report errors automatically
- Integrate tightly with numpy

Numba

- Numba is a just-in-time, type-specializing, function compiler for accelerating numerically-focused Python for either a CPU or GPU.

- **Function Compiler:** Numba compiles individual Python functions into optimized, faster versions, not entire applications or parts of functions. It acts as a Python module, complementing your interpreter.

- **Type-Specializing:** It generates specialized implementations for specific data types, offering significant speed improvements compared to generic Python functions.

Numba

- **Just-in-Time (JIT) Compilation:** Functions are translated at their first call, enabling optimal performance by targeting the actual argument types used. Ideal for interactive workflows, like Jupyter notebooks.

- **Numerically-Focused:** Optimized for numerical data types (e.g., int, float, complex), with limited support for strings. Best performance is achieved with NumPy arrays.

First Steps with Numba

- Use the @jit decorator to compile Python functions for the CPU, enabling simple and efficient optimization.

```
from numba import jit
import math

@jit
def hypot(x, y):
    x = abs(x);
    y = abs(y);
    t = min(x, y);
    x = max(x, y);
    t = t / x;
    return x * math.sqrt(1+t*t)
```

Benchmarking with Numba

- Use ***%timeit*** in Jupyter notebooks to measure function performance accurately.
- Numba optimizations often show significant speed-ups, but:
  - Numba Overhead: Small, fast functions may run slower than Python's built-ins due to function call overhead.
  - Key Insight: Benchmark functions to confirm speed-ups, especially for frequently used operations.

Benchmarking with Numba

User-defined Python function

```
%timeit hypot.py_func(3.0, 4.0)
623 ns ± 2.09 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Numba

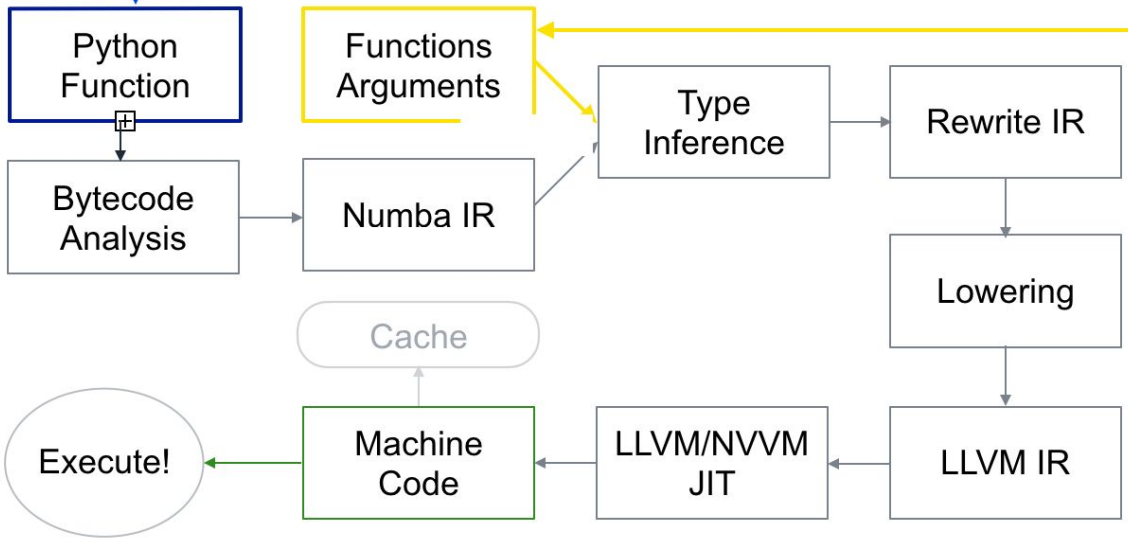
```
%timeit hypot(3.0, 4.0)
180 ns ± 0.0222 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

Python builtin function

```
%timeit math.hypot(3.0, 4.0)
113 ns ± 0.0662 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

How Numba Works

```
@jit
def do_math(a, b):
    …
>>> do_math(x, y)
```



Numba Limitations and Best Practices

- **Unsupported Features:** Numba cannot compile all Python code (e.g., no dictionary support). Unsupported features default to slower object mode.
- **Best Practice:** Use nopython mode (@jit(nopython=True) or @njit) for optimal performance, as it enforces strict type specialization.
- **Benchmark and Debug:** Always benchmark your code and refer to the Numba documentation for supported features.

UNIVERSITY OF WOLVERHAMPTON

End of Lecture 10