Multi-GPU Execution

For Multi-GPU Execution, the first thing we need to consider is what is the support that we have today. If we use something like CUDA, how can we control multiple GPUs with one program. You probably want to write only one program to control multiple GPUs. You can also write multiple programs where each program control one GPU, but it will be difficult for these programs to communication with each other. So eventually, you want to have multiple GPUs to work on one program. Then this program controls multiple GPUs.

There are 3 different ways that we can control multiple GPUs, depending on how we divided it.

1. The first way is called stream-based method.
2. The second way is thread-based method.
3. And the third way is MPI based method.

Thread-based method and MPI based method are not very different because MPI will help you launch the thread so that you don't have to launch the thread yourself. You can use MPI based methods to synchronize all those threads. You don't have to use methods like threads.join() to control multiple threads. Other than that, they are basically the same. We are going to focus on stream-based method.

So how can we write a stream-based multi-GPU program? There are a few APIs that we need to be familiar with. They are cuda device APIs.

- CudaGetDeviceCount()
- CudaGetDeviceProperty()
- CudaSetDevice()

These are the 3 most important APIs.

Before you run a program, you probably want to know how many GPUs are available. Then you want to know the property of the devices. Some machines may have 2 different GPUs and they may have different properties, so you may want to select the best GPU to use. Finally, use CudaSetDevice(id) to select which device you would like to use. You have to provide something like the id to do the selection. For example, when id=1, you select the first GPU. This is something called a thread-local variable. It's not a function, but a variable. There is a thread-local variable running behind the function. You set the device to 1, you are only impacting your current thread. If you have multiple threads, they may have different devices. After you set the device, you will call something like

- cudaMalloc()
- cudaMemcpy()
- launchKernel()

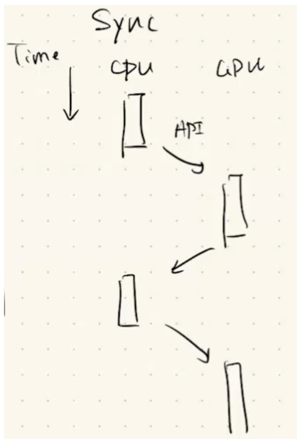
And all these perations will run the particular GPU you set. If you call the setDevice function again to select another GPU, the following operations will be executed on that GPU. By using these APIs, you can control multiple GPUs.

You can consider steams as command queue. For example, when you do cudaMemCpy, which is a synchronized operation, meaning once the function returns, the copy is already completed.

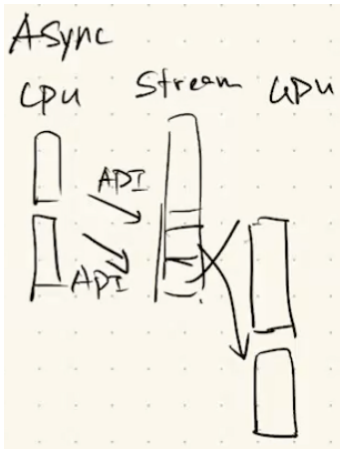
There is another version of this API called cudaMemCpyAsync. It's an asynchronized API which means when you call it, it will return immediately before the action is completed. What it is actually doing is that it will inject a command into this stream(command queue). And it will be played after all commands in front of it complete. So if you call this asynchronous API, you need to provide the stream id so that you can specify which stream you want this command to be added to. Also when we run launch kernel, if we run vec_add<<<1,1,__>>>, it will be inserted into stream 0. Asynchronized API, default stream.

Vec_add<<<1,1,stream_id>>> specify the stream_id if you don't want to insert into the default stream. When we call this API, we're actually adding a command into this stream. There is also another argument named sq-size (scratch pad size). Aftering adding a lot of command to this command queue, run cudaStreamSync(). If we call this API, this function will only return if there's no command in the stream. It's another way of writing a multi-GPU program that you insert a lot of command into a stream and then you call cudaStreamSync(). All the commands will be inserted into this stream. When this function returns, it gurantees that all the commands are completed. You got a result.

Why useful? You don't want your CPU to keep waiting there.
If you use synchronized way, your CPU may run for a while and then not doing anything while your GPU is executing. Like this picture.



But if you are using asynchronized way, your CPU may run for a while and then launch API. Now that the API is actually in a stream, your GPU will pick it up. And CPU may do something more useful and call another API to insert another command into this GPU, then GPU will pick up another command to execute. So we are using both the CPU and GPU, like this picture.



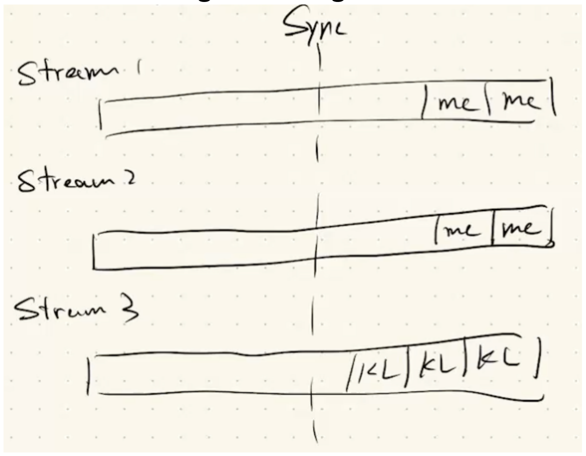
Because the command is already in the stream, the GPU can pick up new command faster than waiting for the control flow to be handed over to the CPU and launch another command. Gap on GPU is smaller. The GPU may have higher utilization. This is another very common way to write a GPU program that can improve GPU utilization. Actually, for most of the high performance computing programs, that involves a lot of kernel cores like machine learning workloads, asynchronized way is a dominant way to write a GPU program.

Streams' Unique Property –use case 1

- Commands within the same stream are guaranteed to execute sequentially.
- Commands from different streams can be executed concurrently.

One particular way that you may think of the utilization of this property is that we can let the GPU do the memory copy while it's doing computation. For example, in deep learning, while we are training one batch of data, we may copy another batch of images to this GPU. In each GPU, we have 2 DM engines and each DM engine can be responsible for memory copy. So 2 memory copies can happen at the same time, while one kernel is being executed.

For example, stream 1 does mc, mc … stream2 does mc, mc… and stream 3 does kl,kl,… But at some point, to do the synchronization together, so that we don't launch a kernel when data hasn't arrived and we don't do something or writing the data that's not supposed to overwrite.

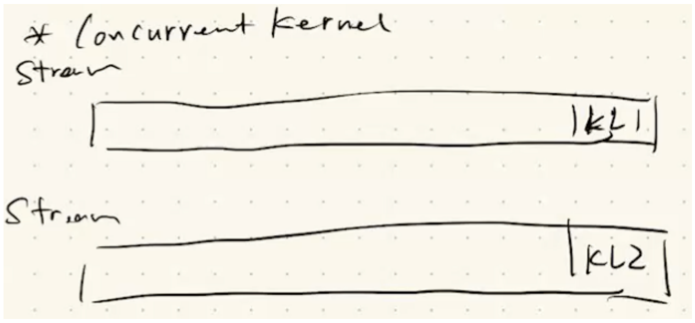


Concurrent kernel is another way to have multiple streams. –use case 2

- 2 kernels, execute at the same time.

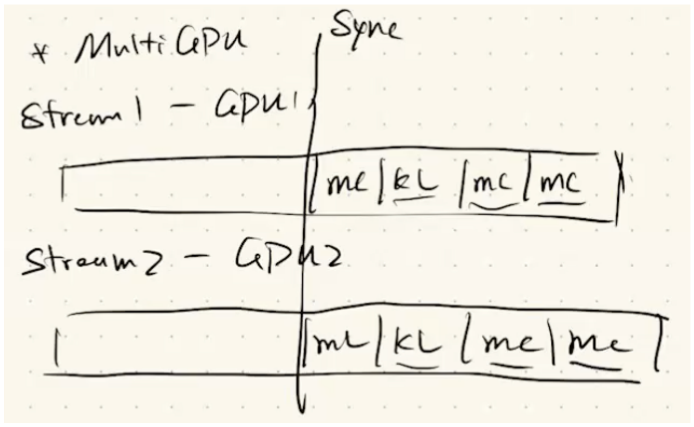
In this 2 frames, because these 2 kernels can be executed concurrently, these 2 kernels can be handled by GPU at the same time. If one kernel really required a lot of computational resources, and one kernel require a lot of memory access, they may not be competing for the resources and the overall execution time can be reduced.

For most of the time, if they are competing for memory bandwidth or cache sizes, then the overall execution time maybe slower. So concurrent kernel is something still being researched without a definite conclusion.



3rd use case multi GPU

- You may allocate stream1 to be associated with GPU1, stream 2 running on GPU2.

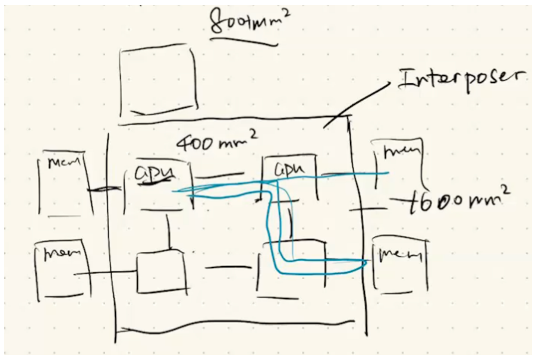


So if you have 2 streams, you can insert those command into streams. Then you do a synchronization to wait for all of them to complete. In this case, because these commands are in different stream commands. In this case, because these commands are in different stream commands, and commands in different streams are excluded concurrently, we can double the computational power available in the whole system.

It's the programmer's responsibility to define all the details in GPU programming. The programmer has the most control. But there's one drawback of this design, it requires a lot of effort. If we have a single GPU program we have no way to directly run it on a multi GPU platform.

MCM GPU

Multi-chip module GPU, (chiplet technology, rather than produce a large chip like 800mm, we produce 4 400mm chips, then we combine 4 chips together on a piece of silicon. Logic is implemented in an underlying layer called interposer.). These 4 chips have some way to communicate with each other. We get much larger chip with less cost.

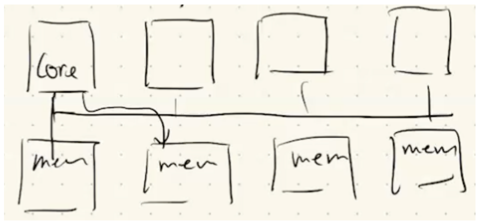


NUMA

It's really annoying to a lot of programmer to have those many chips in there. You start to have something called NUMA(non uniform memory access) effect.

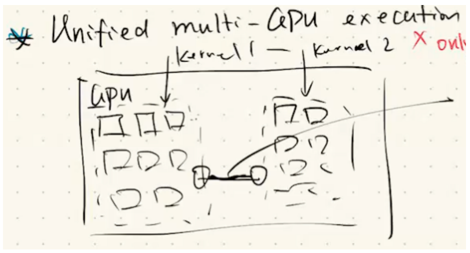
UMA: access from any core to any address has similar latency. Typically, if you consider accessing your own memory and a remote memory, there are no latency differences.

NUMA: You may have different memory chips that is associated with different GPUs. So if the chip you access is own memory, it may have 100 cycle latency. If this chip you access is remote, you may have 1000 cycle latency. Now you do need to consider where I want to place the data, where I want to place to compute to minimize the overall latency. It's getting more and more unreasonable to let programmers to control where to place their data and where to place the memory.



Unified multi-GPU execution.

The academia came up with a unified multi-GPU. This type of design only exists in research papers. I don't see any industrial standard that or any production that supports this feature. So if we want to evaluate this feature, it is basically only available in simulators, you have to use simulators to support this feature.



In our simulator, we configure the network to create long latency links to separate groups of resources. They are connected using a single link that has lower bandwidth and longer latency. Then there's another group of resources. And they all communicate through this particular channel. But logically, they are still a single GPU. You can only launch one kernel at the same time. And this one kernel is executing on its whole system.

So far, 2 kernels are not supported by other simulators, it's only supported by MGPUSIM, that's why MGPUSIM is so unique in multi-GPU simulation.

But the academia is still interested in the unified multi-GPU execution. So in this case, if we exclude a multiple GPU execution mode, the driver is responsible for placing data and computing threads.

Design patterns

- **Singleton**

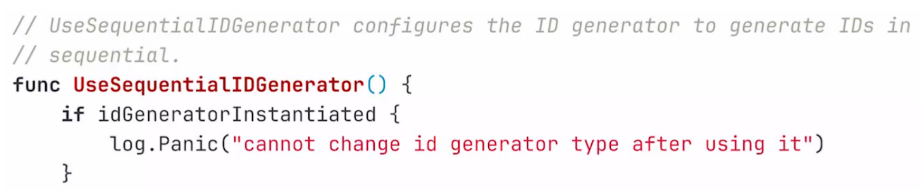
A bad design pattern that you should avoid as much as possible.

In Multi2Sim , there is a single GPU which is a singleton. In your whole program, you may have only one instance of this particular class. At the time multi2sim was beginning, it was reasonable, because we just want to be able to build a simulation and nobody was doing multi-GPU computation at that time. And it's actually sufficient to do single GPU execution.

Then my argument is you never know what is a single instance. For example, we may think while we're executing one problem, we may actually run your one simulation.

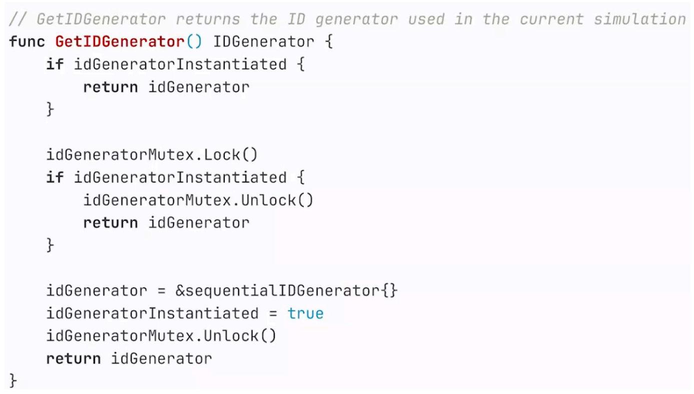
In Akita and MGPUSim, there is only one singleton. Even for that singleton, it's hard to tell whether it's a good design or not.

Example:



```go
// UseSequentialIDGenerator configures the ID generator to generate IDs in
// sequential.
func UseSequentialIDGenerator() {
    if idGeneratorInstantiated {
        log.Panic("cannot change id generator type after using it")
    }
```

In every programming language, you have a particular way to implement singleton. Every programming language can support singleton. Here, we have an ID generator which is a global instance of the ID generator. For every event and message, we have an ID which is for tracing purposes. Then we can assign task for that particular element, either an event or a message. No 2 messages should have the same ID.

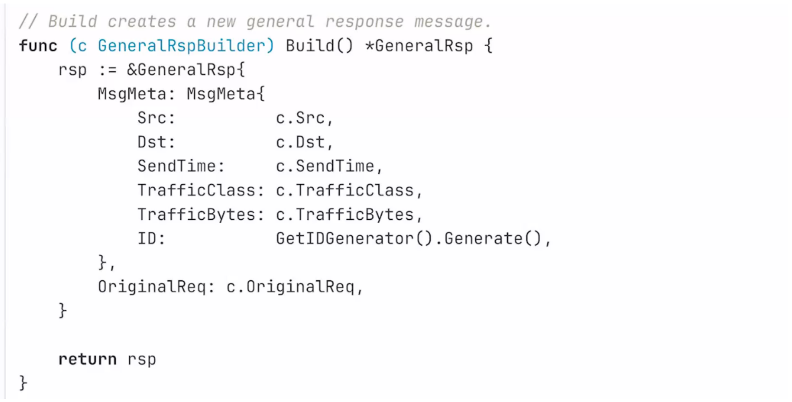


```go
// GetIDGenerator returns the ID generator used in the current simulation
func GetIDGenerator() IDGenerator {
    if idGeneratorInstantiated {
        return idGenerator
    }

    idGeneratorMutex.Lock()
    if idGeneratorInstantiated {
        idGeneratorMutex.Unlock()
        return idGenerator
    }

    idGenerator = &sequentialIDGenerator{}
    idGeneratorInstantiated = true
    idGeneratorMutex.Unlock()
    return idGenerator
}
```

This is a global function which doesn't belong to any class. The GetIDGenerator here is actually retrieving the global instance of this ID generator. There is always only one ID generator running there. This is an application of singleton.



```go
// Build creates a new general response message.
func (c GeneralRspBuilder) Build() *GeneralRsp {
    rsp := &GeneralRsp{
        MsgMeta: MsgMeta{
            Src:          c.Src,
            Dst:          c.Dst,
            SendTime:     c.SendTime,
            TrafficClass: c.TrafficClass,
            TrafficBytes: c.TrafficBytes,
            ID:           GetIDGenerator().Generate(),
        },
        OriginalReq: c.OriginalReq,
    }

    return rsp
}
```

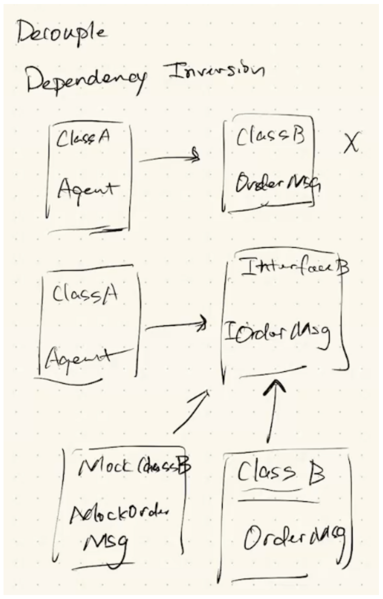
These generated ID are guaranteed to be unique in the simulation. It's reproducible: if you run the simulation again, same message will always get the same ID.
You want your simulation to be deterministic for 2 reasons:

1. Debugging. You want this bug to be reproducible. It's super annoying if you can't reproduce a bug.
2. Reproducible results.

Quick suggestion: avoid using singleton as much as possible.

- **Factory**
  Assume design an agent. There is function called like MakeOrderDecision().
  Msg = new OrderMessage(_,_);

  When we talk about design patterns and design principles, we want to decouple as much as possible.



For example, Class A doesn't need to know the existence of class B, it only needs to know the existence of Interface B. And class B needs to know the existence of Interface B.
This is dependency inversion. Advantage is for testing purposes. This only a small case and will be more useful in larger cases.

Agent may have a field called
-IOrderingFactory -> strategy pattern
-Dependency Injection
    Mock Factory
    Real Facotry
-MakeOrderDecision
    Msg = agent.IOrderMsgFactory.Produce()

Advantage: class A is not coupled with class B. class A only needs to be aware of this interface, then we can change class B's code without modifying class A's code. Class A with only depend on an interface.

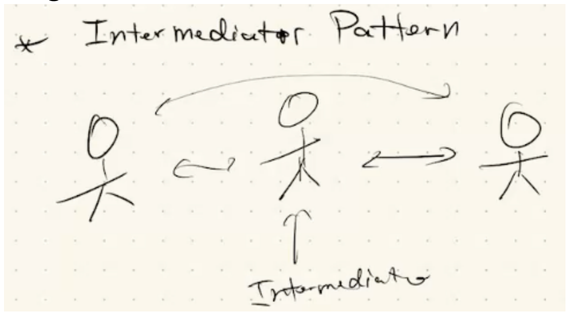
Let's talk about why for most of the cases, it's not necessary. Take object oriented programming instances, you should consider 2 different kinds of classes. You should have one type of class that focuses more on behavior.
Class-Behavior
RoB-Agent -CU -MemoryController
--internal states->Buffer

Class-Data-Pure Data In Java, this kind of class has a particular name called POJO(plain old java object). It only contains fields and setter/getter like methods.
Example:
Circle
        Radius
        getArea()
Events

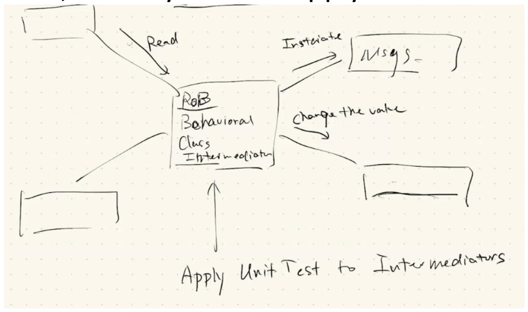
There is almost no way for us to get an error in this get error implementation. So we can trust the implementation of the event, which only contains data, getter and setter method. It would be too complex if we consider mocks. When we create instances within our behavior classes, we should avoid creating another behavior, another complex class. We should only instantiate simple data classes like events and messages. And so far there is no violation of this rule. And our behavior classes never create something super complex. And where we create behavior classes, complex classes are instantiated.

- **Intermediate Pattern**

Intermediate is basically a pattern that two people negotiate. It is very easy to have conflict. In this case, you may have a third person in the middle who is an intermediate to avoid those 2 people directly communicating with each other.



In software engineering, intermediate is basically a pattern we can consider ROB as intermediator there. And the intermediate pattern is behavioral class. This ROB class is controlling messages and controlling some other data fields. They can't change their internal values, only the behavioral classes have the capability of changing the internal states of those fields. This ROB may read data from one side then instantiate or change the value of another object. So when we run unit test, we only need to apply unit test to intermediators.


Read
Instaiate
Msgs
ROB
Behavioral
Class
Intermediator
Change the value
Apply Unit Test to Intermediators