Topics

Rupesh Nasre.

IIT Madras
January 2025

Topics

- Dynamic Parallelism
- Unified Virtual Memory
- Multi-GPU Processing
- Peer Access
- Heterogeneous Processing
- ...

Dynamic Parallelism

- Useful in scenarios involving nested parallelism.

```
for i …
    for j = f(i) …
        work(j)
```

  - Algorithms using hierarchical data structures
  - Algorithms using recursion where each level of recursion has parallelism
  - Algorithms where work naturally splits into independent batches, and each batch involves parallel processing
- Not all nested parallel loops need DP.

```
#include <stdio.h>
#include <cuda.h>
__global__ void Child(int father) {
        printf("Parent %d -- Child %d\n", father, threadIdx.x);
}
__global__ void Parent() {
        printf("Parent %d\n", threadIdx.x);
        Child<<<1, 5>>>(threadIdx.x);
}
int main() {
        Parent<<<1, 3>>>();
        cudaDeviceSynchronize();
        return 0;
}
```

$ nvcc dynpar.cu

error: calling a __global__ function("Child") from a __global__ function("Parent") is only allowed on the compute_35 architecture or above

$ nvcc -arch=sm_35 dynpar.cu

error: kernel launch from __device__ or __global__ functions requires separate compilation mode

$ nvcc -arch=sm_35 -rdc=true dynpar.cu

$ a.out

```
#include <stdio.h>
#include <cuda.h>
__global__ void Child(int father) {
        printf("Parent %d -- Child %d\n", father, threadIdx.x);
}
__global__ void Parent() {
        printf("Parent %d\n", threadIdx.x);
        Child<<<1, 5>>>(threadIdx.x);
}
int main() {
        Parent<<<1, 3>>>();
        cudaDeviceSynchronize();
        return 0;
}
```

```
Parent 0
Parent 1
Parent 2
Parent 0 -- Child 0
Parent 0 -- Child 1
Parent 0 -- Child 2
Parent 0 -- Child 3
Parent 0 -- Child 4
Parent 1 -- Child 0
Parent 1 -- Child 1
Parent 1 -- Child 2
Parent 1 -- Child 3
Parent 1 -- Child 4
Parent 2 -- Child 0
Parent 2 -- Child 1
Parent 2 -- Child 2
Parent 2 -- Child 3
Parent 2 -- Child 4
```

```
#include <stdio.h>
#include <cuda.h>

#define K 2

__global__ void Child(int father) {
        printf("%d\n", father + threadIdx.x);
}
__global__ void Parent() {
        if (threadIdx.x % K == 0) {
                Child<<<1, K>>>(threadIdx.x);
                printf("Called childen with starting %d\n", threadIdx.x);
        }
}
int main() {
        Parent<<<1, 10>>>();
        cudaDeviceSynchronize();

        return 0;
}
```

```
0
1
Called childen with starting 0
Called childen with starting 2
Called childen with starting 4
Called childen with starting 6
Called childen with starting 8
2
3
4
5
6
7
8
9
```

DP: Computation

- Parent kernel is associated with a parent grid.
- Child kernel**s** are associated with child grids.
- Parent and child kernels may execute asynchronously.
- A parent grid is not complete unless all its children have completed.

DP: Memory

- Parent and children **share** global and constant memory.
- But they have **distinct** local and shared memories.
- All global memory operations in the parent **before** child's launch are visible to the child.
- All global memory operations of the child are visible to the parent **after** the parent synchronizes on the child's completion.

```
__global__ void child_launch(int *data) {
    data[threadIdx.x] = data[threadIdx.x] + 1;
}
__global__ void parent_launch(int *data) {
    data[threadIdx.x] = threadIdx.x;
    __syncthreads();

    if (threadIdx.x == 0) {
        child_launch<<< 1, 256 >>>(data);
        cudaDeviceSynchronize();
    }
    __syncthreads();
}
void host_launch(int *data) {
    parent_launch<<< 1, 256 >>>(data);
    cudaDeviceSynchronize();
}
```

Without this barrier, only `data[0..31]` are guaranteed to be visible to the child (preVolta).

Without this barrier, only warp 0 is guaranteed to see the child modifications (with CDS permitted).

What happens if the two __syncthreads() are removed?

CDS is disallowed on the device.

From CUDA Programming Guide

- Explicit synchronization with child kernels from a parent block (i.e. using cudaDeviceSynchronize() in device code) is deprecated in CUDA 11.6 and removed for compute_90+ compilation.
- For compute capability < 9.0, compile-time opt-in by specifying -DCUDA_FORCE_CDP1_IF_SUPPORTED is required to continue using cudaDeviceSynchronize() in device code.
- Note that this is slated for full removal in a future CUDA release.

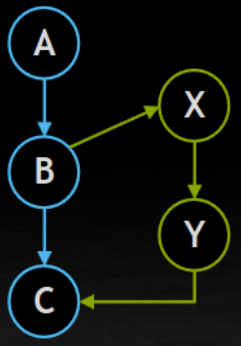
Is there a way to see the effects of a child kernel?

```
__global__ void tail_launch(int *data) {
   data[threadIdx.x] = data[threadIdx.x]+1;
}
__global__ void child_launch(int *data) {
      data[threadIdx.x] = data[threadIdx.x] + 1;
}
__global__ void parent_launch(int *data) {
      data[threadIdx.x] = threadIdx.x;
      __syncthreads();

      if (threadIdx.x == 0) {
            child_launch<<< 1, 256 >>>(data);
            cudaDeviceSynchronize();
            tail_launch<<< 1, 256, 0, cudaStreamTailLaunch >>>(data);
      }
}
void host_launch(int *data) {
      parent_launch<<< 1, 256 >>>(data);
      cudaDeviceSynchronize();
}
```

tail_launch is launched only at the end of parent_launch.

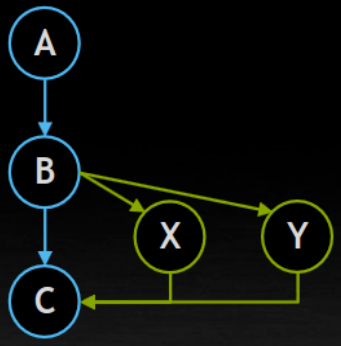
Special Streams

A, B, C kernels are launched from the host in the same stream. B launches X and Y.



Per-Thread stream

X & Y execute **sequentially**, similar to existing stream launch



Fire-and-forget

X & Y execute **independently** as if launched in separate streams

Tail launch

X & Y execute sequentially **after** parent kernel completes

```
__global__ void B() {
    X <<< ..., cudaStreamTailLaunch >>>();
    Y <<< ..., cudaStreamTailLaunch >>>();
}
```

Note: FireAndForget, TailLaunch streams cannot be used with events.

Stephen Jones, NVIDIA

```
// In this example, Z will launch only after P, Q, and R complete.
__global__ void R( ... ) {
  Z<<<..., cudaStreamTailLaunch>>>(...);
  P<<<..., cudaStreamPerThread>>>(...);
  Q<<<..., cudaStreamFireAndForget>>>(...)
}
```

Local and Shared Memory

- It is illegal to pass pointer to shared or local memory.

```
int x_array[10]; // Creates x_array in parent's local memory
child_launch<<< 1, 1 >>>(x_array);
```

- Argument passed should be pointers to global memory: *cudaMalloc*, *new* or global *__device__*.

```
// Correct access
__device__ int value;
__device__ void x() {
      value = 5;
      child<<< 1, 1 >>>(&value);
}
```

Kernel can be called from a device function.

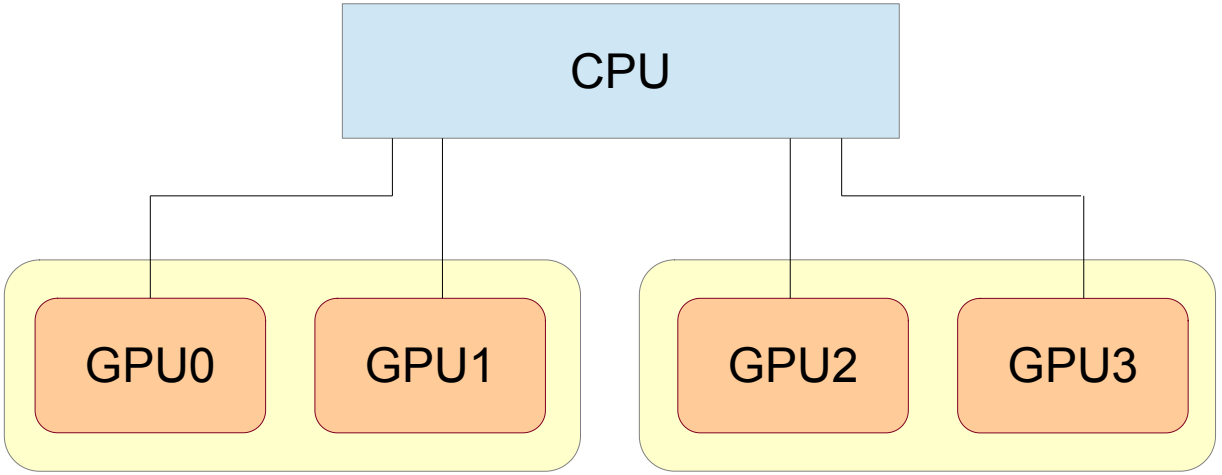
DP Overheads

- To launch a kernel, CUDA driver and runtime parse parameters, buffer their values, and issue kernel dispatch.
- Kernels waiting to execute are inserted in a pending buffer, modeled as fixed-sized + variable-sized pools. The latter has higher management overheads.
- (pre-11.6) If parent explicitly synchronizes with the child, to free resources for the execution of the children, parent kernels may be swapped to global memory.

Multi-GPU Processing

Why Multi-GPU?

- Having multiple CPU-GPU handshakes should suffice?
  - Pro: Known technology to communicate across CPUs
  - Con: If GPU is the primary worker, we pay too much for additional CPUs



Multiple Devices

- In general, a CPU may have different types of devices, with different compute capabilities.
- However, they all are nicely numbered from 0..N-1.
- *cudaSetDevice*(i)

What is wrong with this code from parallelization perspective?

```
cudaSetDevice(0);
K1<<<...>>>();
cudaMemcpy();
cudaSetDevice(1);
K2<<<...>>>();
cudaMemcpy();
```

```
cudaSetDevice(0);
K1<<<...>>>();
cudaMemcpyAsync();
cudaSetDevice(1);
K2<<<...>>>();
cudaMemcpyAsync();
```

Enumerate Devices

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp, device);
  printf("Device %d has compute capability %d.%d.\n",
         device, deviceProp.major, deviceProp.minor);
}
```

```
Device 0 has compute capability 7.5.
Device 1 has compute capability 6.0.
```

Kernels in Streams

- Device memory allocations, kernel launches are made on the currently set device.
- Streams and events are created in association with the currently set device.

```
cudaSetDevice(0); // Set device 0 as current
cudaStream_t s0;
cudaStreamCreate(&s0); // Create stream s0 on device 0
MyKernel<<<100, 64, 0, s0>>>(); // Launch kernel on device 0 in s0

cudaSetDevice(1); // Set device 1 as current
cudaStream_t s1;
cudaStreamCreate(&s1); // Create stream s1 on device 1
MyKernel<<<100, 64, 0, s1>>>(); // Launch kernel on device 1 in s1

// This kernel launch will fail:
MyKernel<<<100, 64, 0, s0>>>(); // Launch kernel on device 1 in s0
```

MemCopies in Streams

- A memory copy succeeds even if it is issued to a stream that is not associated to the current device.
- Each device has its own default stream.
  - Commands to default streams of different devices may execute concurrently.

```
cudaSetDevice(0); // Set device 0 as current
cudaStream_t s0;
cudaStreamCreate(&s0);

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

// This memory copy is alright.
cudaMemcpyAsync(dst, src, size, H2D, s0);
```

```
cudaSetDevice(0);
K<<<...>>>();
cudaSetDevice(1);
K<<<...>>>();

// The two kernels may
// execute concurrently.
```

Events

- cudaEventRecord() expects the event and the stream to be associated with the same device.
- cudaEventElapsedTime() expects the two events to be from the same device.
- cudaEventSynchronize() succeeds even if the input event is associated with a device different from the current device.
- cudaStreamWaitEvent() succeeds even if the stream and event are associated to different devices.
  - This can be used for inter-device synchronization.

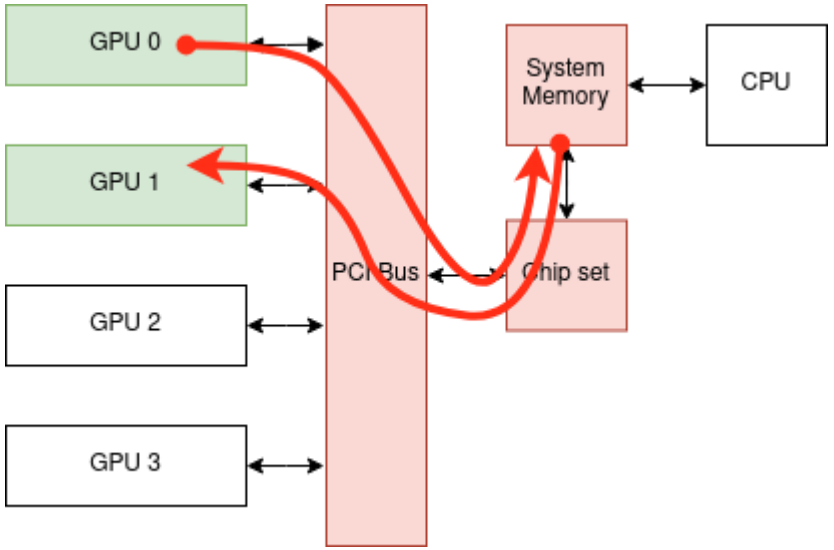
```
int main() {
        cudaStream_t s0, s1;
        cudaEvent_t e0, e1;

        cudaSetDevice(0);
        cudaStreamCreate(&s0);
        cudaEventCreate(&e0);

        K1<<<1, 1, 0, s0>>>();

        K2<<<1, 1, 0, s0>>>();

        cudaSetDevice(1);
        cudaStreamCreate(&s1);
        cudaEventCreate(&e1);

        K3<<<1, 1, 0, s1>>>();

        K4<<<1, 1, 0, s1>>>();

        cudaDeviceSynchronize();

        cudaSetDevice(0);
        cudaDeviceSynchronize();
        return 0;
}
```

What does this program do?

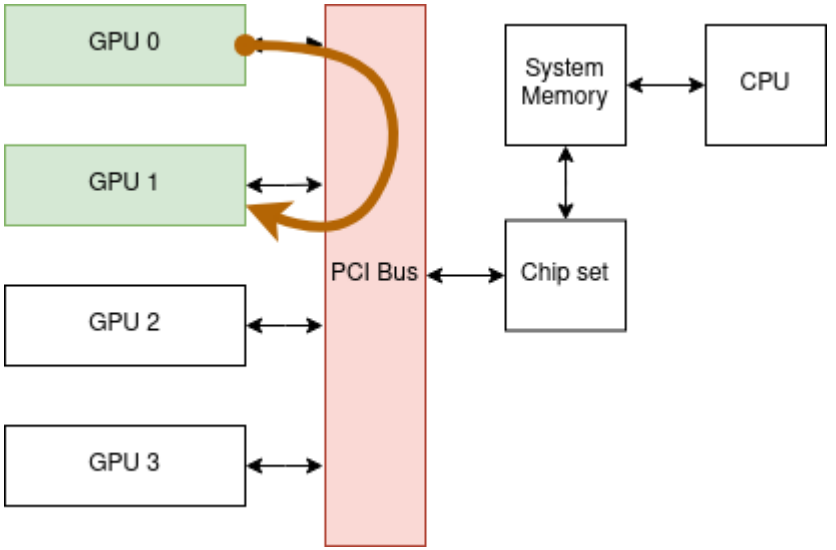
Now ensure that K4 does not start before K1 completes. Use events.

```
int main() {
        cudaStream_t s0, s1;
        cudaEvent_t e0, e1;

        cudaSetDevice(0);
        cudaStreamCreate(&s0);
        cudaEventCreate(&e0);

        K1<<<1, 1, 0, s0>>>();
        cudaEventRecord(e0, s0);
        K2<<<1, 1, 0, s0>>>();

        cudaSetDevice(1);
        cudaStreamCreate(&s1);
        cudaEventCreate(&e1);

        K3<<<1, 1, 0, s1>>>();
        cudaStreamWaitEvent(s1, e0, 0);
        K4<<<1, 1, 0, s1>>>();
        cudaDeviceSynchronize();

        cudaSetDevice(0);
        cudaDeviceSynchronize();
        return 0;
}
```

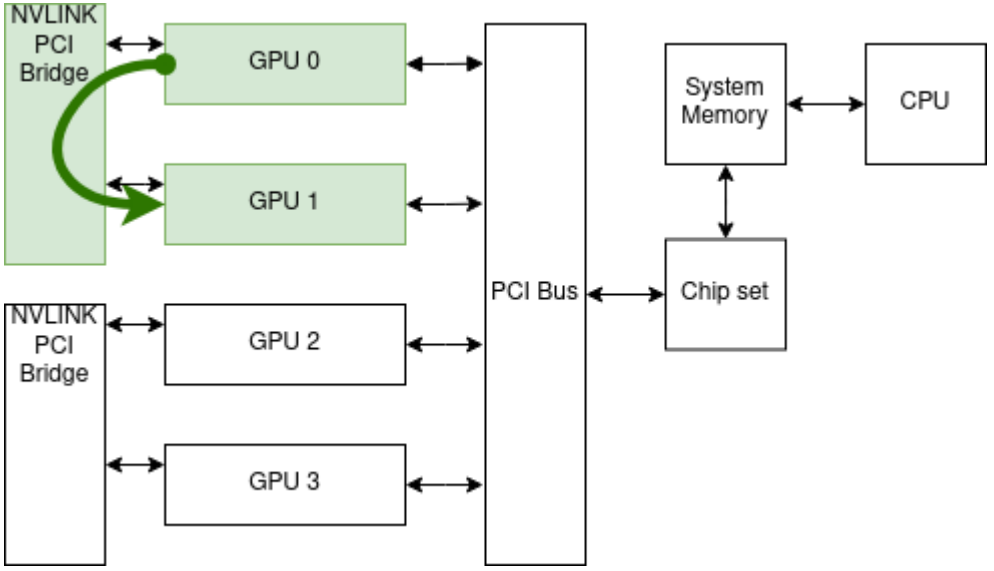
Peer Access



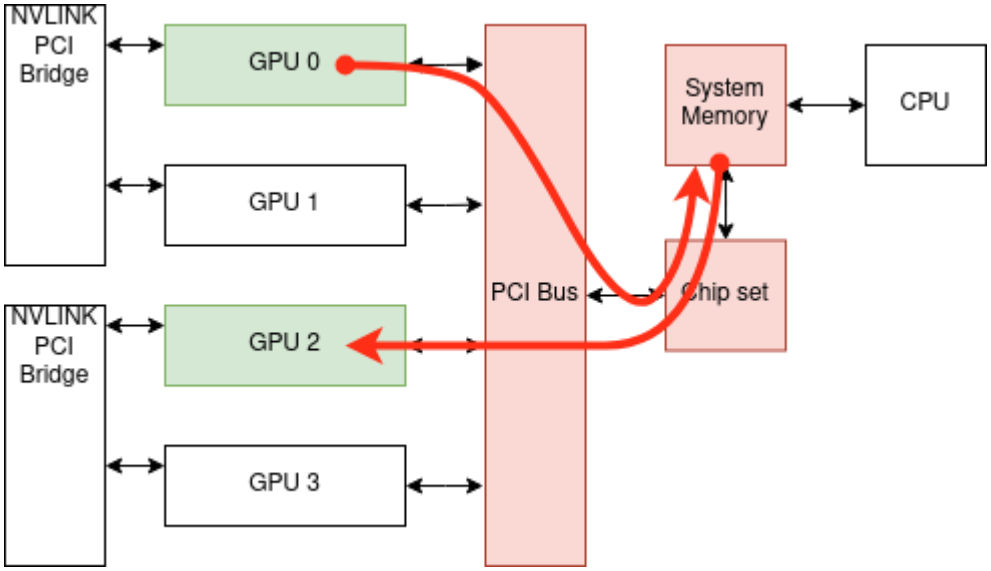
Access via host



Peer Access via PCI Bus

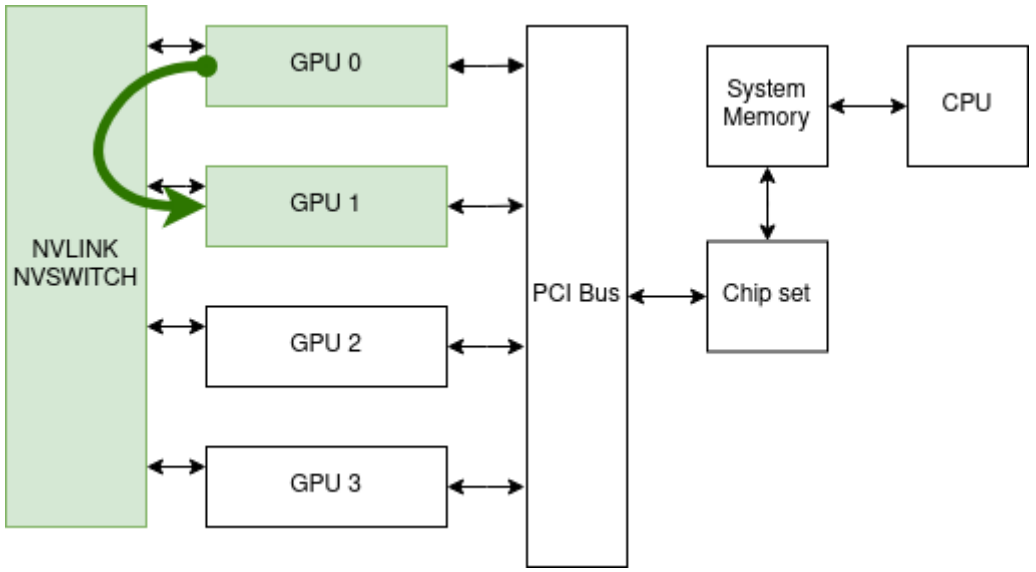


Peer Access via NVLink PCI Bridge

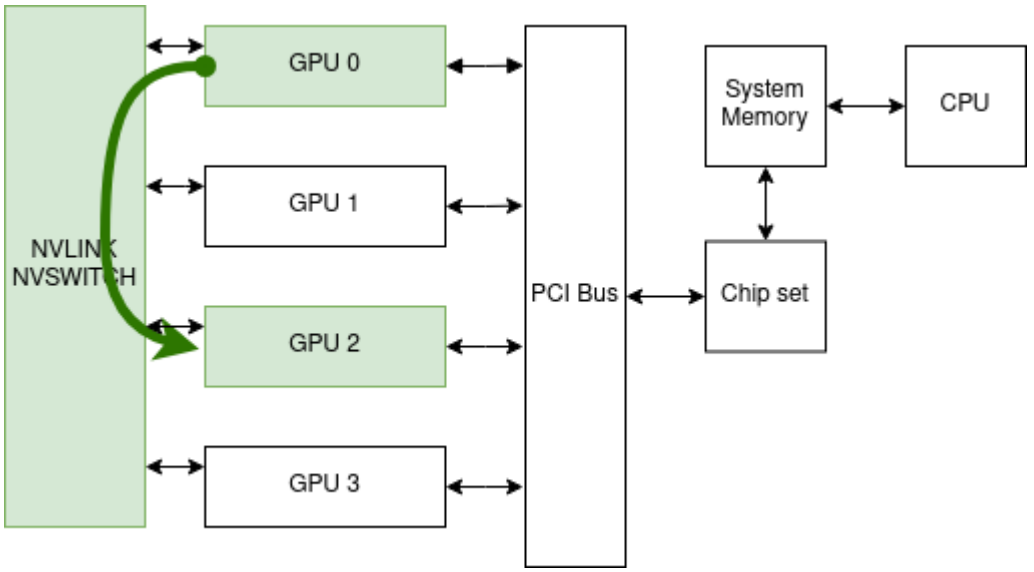


Non-Peer Access via Host

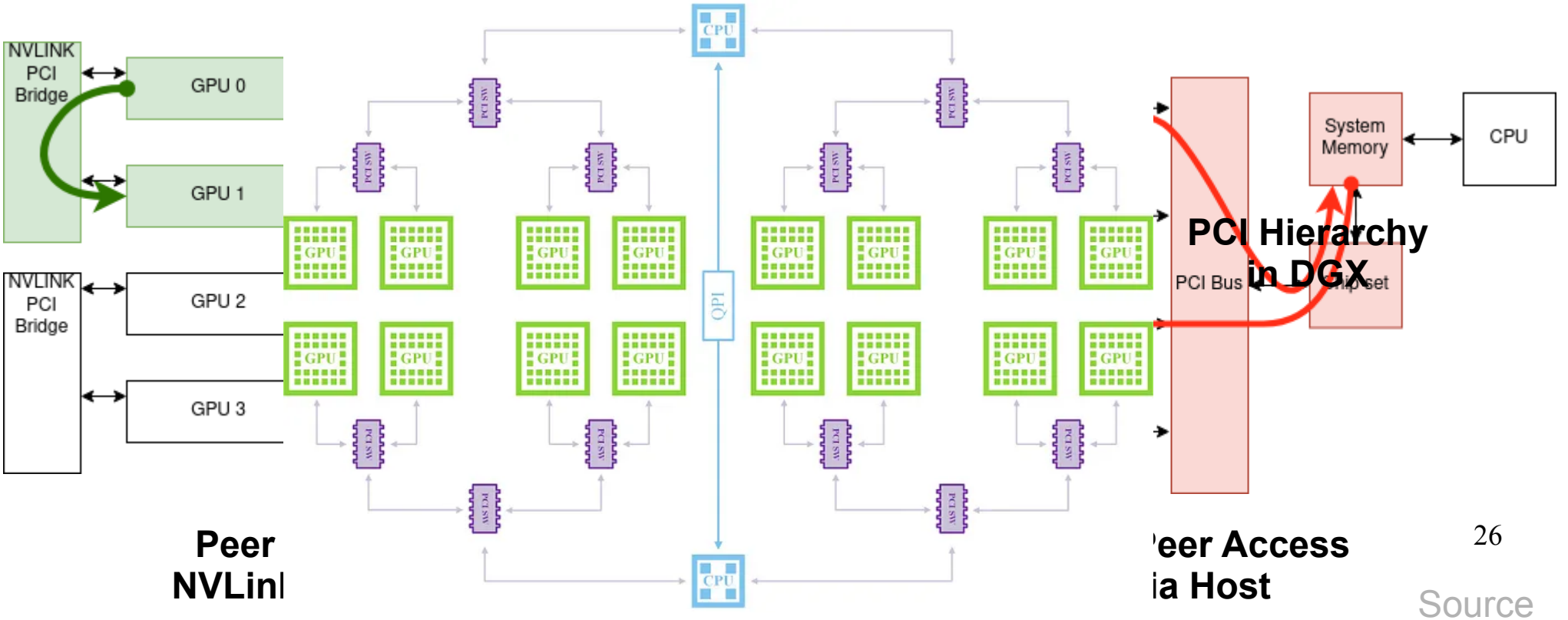
Source



Peer Access via NVLink Switch



Non-local Peer Access via NVLink Switch



Peer NVLink

Peer Access via Host

Source

Peer Access

- A kernel on one device can dereference a pointer to the memory on another device.
- This gets internally implemented by unifying virtual address spaces of the devices.

```
cudaSetDevice(0);
float *p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size);
K<<<1000, 128>>>(p0);

cudaSetDevice(1);
cudaDeviceEnablePeerAccess(0, 0);
K<<<1000, 128>>>(p0);
```

The same API works irrespective of the connection being via PCI-e, NVLink Bridge, or NVLink Switch.

```
__global__ void K(float *ptr) {
        printf("%.f ", ptr[threadIdx.x]);
        ptr[threadIdx.x] = threadIdx.x;
}
int main() {
        float *ptr;
        cudaSetDevice(0);
        cudaMalloc(&ptr, sizeof(float) * 10);
        K<<<1, 10>>>(ptr);
        cudaDeviceSynchronize(); printf("\n");

        cudaSetDevice(1);
        // cudaDeviceEnablePeerAccess(0, 0);  // uncomment this line.
        K<<<1, 10>>>(ptr);
        cudaDeviceSynchronize(); printf("\n");
        cudaError_t err = cudaGetLastError();
        printf("%d, %s, %s\n", err, cudaGetErrorName(err), cudaGetErrorString(err));

        return 0;
}
```

```
0 0 0 0 0 0 0 0 0 0

77, cudaErrorIllegalAddress, an illegal memory access was encountered
```

```
0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9
0, cudaSuccess, no error
```

```
cudaSetDevice(0);
float *p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size);
K<<<1000, 128>>>(p0);

cudaSetDevice(1);
cudaDeviceEnablePeerAccess(0, 0);
float *p1;
cudaMalloc(&p1, size);
K<<<1000, 128>>>(p1);
K<<<1000, 128>>>(p0);

cudaSetDevice(0);
cudaDeviceEnablePeerAccess(1, 0);
K<<<1000, 128>>>(p1);
cudaMemcpyPeer(p1, 1, p0, 0, size);
```

Works for both the previous allocations as well as future ones.

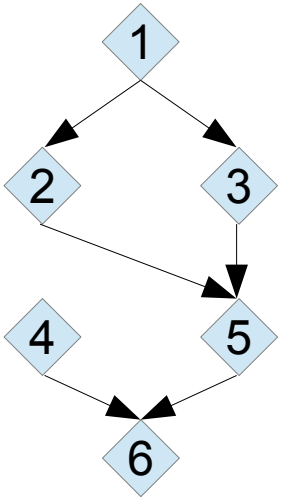
Cost of cudaMalloc

- cudaMalloc call is about O(log(N)), where N is the number of prior allocations on the current GPU
- With peer-access, it becomes O(D * log(N)), where D is the number of devices with the peer access.
  - The allocation needs to be mapped to all the enabled devices.
  - Indicates that the mapping is not done in parallel.

Source

Classwork

- Implement inter-device barrier using events.
- Simulate the following dependency graph. Each node represents a kernel on a different device.



Common Memories

Name	API	Allocation	Auto-Synced?
Pinned Memory	*cudaHostAlloc*	Host	Yes
Unified Virtual Addressing	*cudaMallocManaged*	Device	No
Unified Memory	*cudaMallocManaged*	Device	Yes

PTX

- Parallel Thread Execution
- Assembly Language for CUDA

```
__global__ void K() {
	printf("in K\n");
}
int main() {
	K<<<1, 1>>>();
	cudaDeviceSynchronize();

	return 0;
}
```

```
// Generated by NVIDIA NVVM Compiler
//
// Compiler Build ID: CL-21124049
// Cuda compilation tools, release 8.0, V8.0.44
// Based on LLVM 3.4svn
//

.version 5.0
.target sm_35
.address_size 64

	// .globl	_Z1Kv
.extern .func  (.param .b32 func_retval0) vprintf
(
	.param .b64 vprintf_param_0,
	.param .b64 vprintf_param_1
)
;
.global .align 1 .b8 $str[6] = {105, 110, 32, 75, 10,
0};
```

PTX

- Parallel Thread Execution
- Assembly Language for CUDA

```
__global__ void K() {
        printf("in K\n");
}
int main() {
        K<<<1, 1>>>();
        cudaDeviceSynchronize();

        return 0;
}
```

nvcc --ptx file.cu

```
.visible .entry _Z1Kv()
{
        ...
        mov.u64         %rd1, $str;
        cvta.global.u64         %rd2, %rd1;
        mov.u64         %rd3, 0;
        // Callseq Start 0
        {
        .reg .b32 temp_param_reg;
        // <end>}
        .param .b64 param0;
        …
        call.uni (retval0),
        vprintf,
        (param0, param1);
        ld.param.b32    %r1, [retval0+0];
        }// Callseq End 0
        ret;
}
```

Variables

- Usual registers, temporaries, etc. are used in PTX also.
- Some special variables are present:
  - threadIdx gets mapped to %tid. This is a predefined, read-only, per-thread special register.
  - blockDim gets mapped to %ntid.
  - %warpid, %nwarpid are available in PTX.
  - %smid, %nsmid are available.
  - %total_smem_size: static + dynamic

Synchronization Constructs

- bar, barrier
  - Variations on scope
- membar, fence
  - Variations on strictness
- atom.op {.and, .or, .xor, .cas, .min, ...}

Warp Level Functions

- Warp threads can communicate
  - via shared or global memory
- Warp-level intrinsics
  - communicate via registers
  - fast
- Use bitmask
  - 0xFFFFFFFF (thirty two 1s)

Active Mask

```
#include <stdio.h>
#include <cuda.h>

#define N 32

__global__ void K() {
        printf("%X\n", __activemask());
}
int main() {
        K<<<1, N>>>();
        cudaDeviceSynchronize();

        return 0;
}
```

```
FFFFFFFF
FFFFFFFF
...
32 times
```

This output is practically always seen.
But is not guaranteed by the documentation.
(Volta onward)

__activemask provides a mask indicating which threads are currently executing this instruction.

Active Mask

```
#include <stdio.h>
#include <cuda.h>

#define N 32

__global__ void K() {
        if (threadIdx.x == 0) printf("%X\n", __activemask());
}
int main() {
        K<<<1, N>>>();
        cudaDeviceSynchronize();

        return 0;
}
```

1

- __activemask() indicates which threads are executing this instruction *at this time*.
- It shows which warp-threads have converged. It does not cause threads to converge.

Warp Voting

- __all_sync(mask, predicate);
  - If all warp threads satisfy the predicate.
- __any_sync(mask, predicate);
  - If any warp threads satisfies the predicate.

return 0 or 1

- __ballot_sync(mask, predicate);
  - Which warp threads satisfy the predicate.

return a mask

Threads in the mask are waited for for convergence, and then the operation is performed.

Warp Voting

```
#include <stdio.h>
#include <cuda.h>

#define mask 0xFFFFFFFF

__global__ void K() {
        unsigned val = __all_sync(mask, threadIdx.x < 100);
        if (threadIdx.x % 32 == 0) printf("%X\n", val);
}
int main() {
        K<<<1, 128>>>();
        cudaDeviceSynchronize();

        return 0;
}
```

```
1
1
1
0
```

Warp Voting

```
#include <stdio.h>
#include <cuda.h>

#define mask 0xFFFFFFFF

__global__ void K() {
        unsigned val = __any_sync(mask, threadIdx.x < 100);
        if (threadIdx.x % 32 == 0) printf("%X\n", val);
}
int main() {
        K<<<1, 128>>>();
        cudaDeviceSynchronize();

        return 0;
}
```

```
1
1
1
1
```

Warp Voting

```
#include <stdio.h>
#include <cuda.h>

#define mask 0xFFFFFFFF

__global__ void K() {
        unsigned val = __ballot_sync(mask, threadIdx.x < 100);
        if (threadIdx.x % 32 == 0) printf("%X\n", val);
}
int main() {
        K<<<1, 128>>>();
        cudaDeviceSynchronize();

        return 0;
}
```

```
FFFFFFFF
FFFFFFFF
F
FFFFFFFF
```

Warp Voting

```
#include <stdio.h>
#include <cuda.h>

#define mask 0xFFFFFFFF

__global__ void K() {
        unsigned val = __ballot_sync(mask, threadIdx.x % 2 == 0);
        if (threadIdx.x % 32 == 0) printf("%X\n", val);
}
int main() {
        K<<<1, 128>>>();
        cudaDeviceSynchronize();

        return 0;
}
```

```
55555555
55555555
55555555
55555555
```

Warp Voting for atomics

- `if (condition) atomicInc(&counter, N);`
  - Executed by many threads in a block, but not all.
  - The contention is high.
  - Can be optimized with `__ballot_sync`.
- Leader election
  - Can be thread 0 of each warp (threadIdx.x % 32 == 0)
  - If leader should be one of the threads satisfying the condition, `__ffs` helps.
- Leader collects warp-count.
  - `__ballot_sync()` provides a mask; how do we count bits?
  - `__popc(mask)` returns the number of set bits.
  - `__ffs(mask)` returns the first set bit (from lsb).
- Leader performs a single `atomicAdd`.
  - Reduces contention.

Warp Voting for atomics

```
#include <stdio.h>
#include <cuda.h>

#define mask 0xFFFFFFFF

__global__ void K() {
        unsigned val = __ballot_sync(mask, threadIdx.x < 100);
        if (threadIdx.x % 32 == 0) printf("%d\n", __popc(val));
}
int main() {
        K<<<1, 128>>>();
        cudaDeviceSynchronize();

        return 0;
}
```

```
32
32
32
4
```

Warp Consolidation

Original code

```
if (condition) atomicInc(&counter, N);
```

Optimized code

```
unsigned mask = __ballot_sync(0xFFFFFFFF, condition);
if (threadIdx.x % 32 == 0)
    atomicAdd(&counter, __popc(mask));
```

Classwork

- Return the mask if every third thread of a warp has `a[threadIdx.x] == 0`.
  - What should be the mask if `a` is initialized to all 0?

```
unsigned mask = __ballot_sync(0xFFFFFFFF,
                    threadIdx.x % 3 == 0 && a[threadIdx.x] == 0
                );
```

This code forces other threads to return 0.
Ideally, other threads should be don't care.

```
unsigned mask = __ballot_sync(0xFFFFFFFF,
                    threadIdx.x % 3 == 0 && a[threadIdx.x] == 0
                ||  threadIdx.x % 3 != 0
                );
```

Implementing Warp Voting

- Simulate __any_sync, __all_sync, __ballot_sync.
  - Check where you need atomics.
- Extend these intrinsics for a thread block.
- Extend across all GPU threads.
- Extend for multi-GPU case.

In this course...

- Basic GPU Programming
  - Computation, Memory, Synchronization, Debugging
- Advanced GPU Programming
  - Streams, Heterogeneous computing, Functions
- Topics in GPU Programming
  - Unified virtual memory, multi-GPU, peer access
- Case Study
  - To be discussed