Topics

Topics

- Dynamic Parallelism
- Multi-GPU Processing
- Warp Voting

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

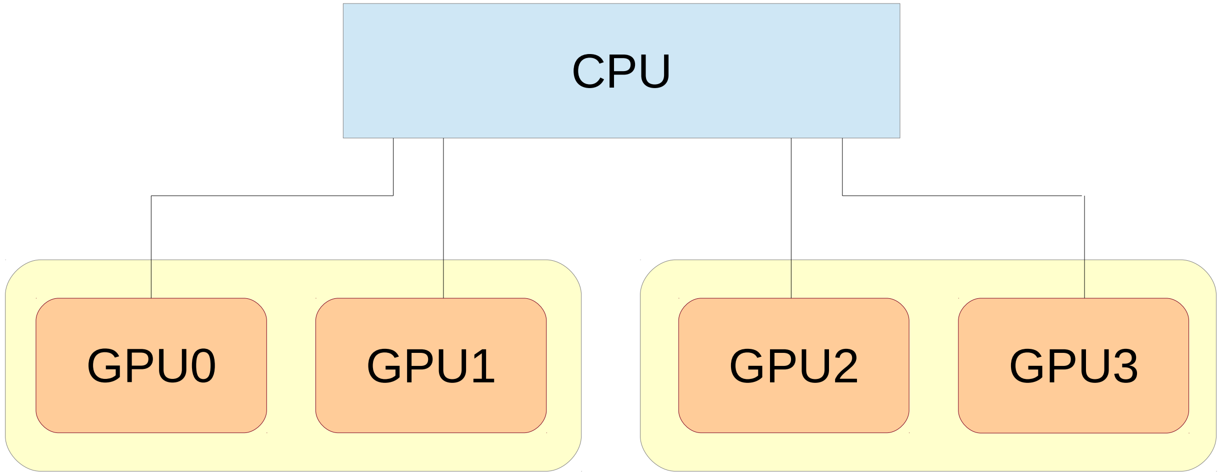
- Parent kernel is associated with a parent grid.
- Child kernel**s** are associated with child grids.
- Parent and child kernels may execute asynchronously.
- A parent grid is not complete unless all its children have completed.

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.

Why Multi-GPU?

- Having multiple CPU-GPU handshakes should suffice?



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();
```

Multiple Devices

- cudaGetDeviceCount(&c);
  - Identify the number of devices.
- cudaDeviceCanAccessPeer(&can, from, to);
  - Can from device access to device?
- cudaDeviceEnablePeerAccess(peer, ...);
- While at the hardware level, the relation seems symmetric, the programming interface enforces asymmetry.
- Maximum 8 peer connections per device.
- Need 64 bit application.

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);
}
```

Warp Voting

- __all(predicate);
  - If all warp threads satisfy the predicate.
- __any(predicate);
  - If any warp threads satisfies the predicate.
- __ballot(predicate);
  - Which warp threads satisfy the predicate.
  - Generalizes *__all* and *__any*.

Warp Voting

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

__global__ void K() {
        unsigned val = __all(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>

__global__ void K() {
        unsigned val = __any(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>

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

        return 0;
}
```

FFFFFFFF
FFFFFFFF
FFFFFFFF
F

Warp Voting

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

__global__ void K() {
        unsigned val = __ballot(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 warp, but not all.
  - The contention is high.
  - Can be optimized with __ballot.
- Leader collects warp-count.
  - __ballot() 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 Consolidation

Original code

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

Optimized code

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

Warp Voting for atomics

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

__global__ void K() {
        unsigned val = __ballot(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
```

Conditional Warp Voting

- If a warp-voting function is executed within a conditional, some threads may be masked, and they would not participate in the voting.

```
if (threadIdx.x % 2 == 0) {
    unsigned mask = __ballot(threadIdx.x < 100);
    if (threadIdx.x % 32 == 0) printf("%d\n", __popc(mask));
}
```

```
16
16
16
2
```