Special Topics

Rupesh Nasre.

IIT Madras

January 2020

Topics

- Dynamic Parallelism
- Multi-GPU Processing
- Peer Access
- Unified Virtual Memory
- PTX
- Warp-voting
- Heterogeneous Programming

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;
}
```

```
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
```

```
$ nvcc dynpar.cu
error: calling a __global__ function("Child") from a __global_        y allowed
on the compute_35 architecture or above
$ nvcc -arch=sm_35 dynpar.cu
error: kernel launch from __device__ or __global__ functions        lation
mode
$ nvcc -arch=sm_35 -rdc=true dynpar.cu
$ a.out
```

```
#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 startir
        }
}
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.
- Recall *preamble*, *work*, and *postamble* from Streams.

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.
- Recall threadfence.

```
__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] is guaranteed to be visible to the child.

Without this barrier, only thread 0 is guaranteed to see the child modifications.

What happens if the two __syncthreads() are removed?

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: Synchronization

- Child grids launched into the same stream are executed in-order.
  - Effects of the previous are visible to the next.
- Events can be used to create dependencies across streams.
- Streams and events created within a grid exist **within thread block scope**. They have undefined behavior when used outside the thread-block where they are created.
  - Thus, all threads of a thread-block by default launch kernels into the same default stream.

DP: Synchronization

- Grids launched within a thread-block in the default stream are executed sequentially.
  - The next grid starts after the previous completes.
  - This happens even if grids are launched by different threads in the parent thread-block.
- To achieve more concurrency, we can use named streams.

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

__global__ void Child(int parent) {
	printf("\tparent %d, child %d\n", parent, threadIdx.x + blockIdx.x * blockDim.x);
}
__global__ void Parent() {
	unsigned id = threadIdx.x + blockIdx.x * blockDim.x;
	printf("parent %d\n", id);
	Child<<<2, 2>>>(id);
	cudaDeviceSynchronize();
}
int main() {
	Parent<<<3, 4>>>();
	cudaDeviceSynchronize();

	return 0;
}
```

- There are 12 child kernels, corresponding to parents (0, 0) through (2, 3).
- Child kernels for parents (i, 0), (i, 1), (i, 2) and (i, 3) execute serially.
- Child kernels for parents in different blocks execute concurrently, such as (0, 0), (1, 0), and (2, 1).
- Since there are three thread blocks at the parent level, maximum three child kernels can be running concurrently in the default stream.

```
__global__ void Child(int parent) {
        printf("\tparent %d, child %d\n", parent, threadIdx.x + blockIdx.x * blockDim.x);
}
__global__ void Parent() {
        unsigned id = threadIdx.x + blockIdx.x * blockDim.x;
        printf("parent %d\n", id);
        cudaStream_t ss;
        cudaStreamCreateWithFlags(&ss, cudaStreamNonBlocking);
        Child<<<2, 2, 0, ss>>>(id);
        cudaDeviceSynchronize();
}
int main() {
        Parent<<<3, 4>>>();
        cudaDeviceSynchronize();

        return 0;
}
```

Source: dynpar5.cu

- All kernel calls are independent (even those launched by threads within the same block).
- However, the number of concurrent kernels is decided by the number of resources available and that supported by your GPU.

DP Overheads

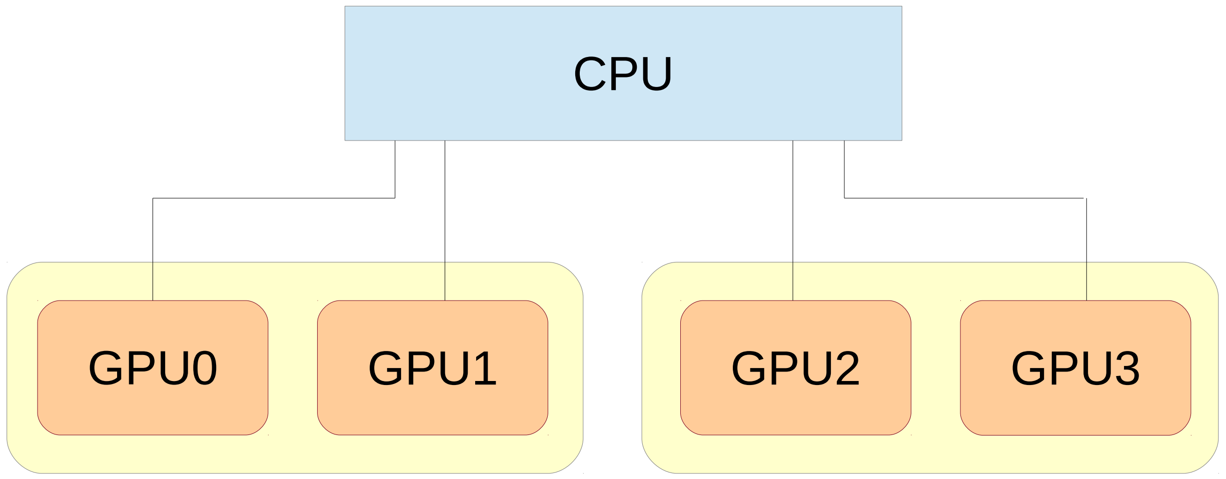
- To launch a kernel, CUDA driver and runtime parse parameters, buffer their values, and issue kernel dispatch.
- Kernels waiting to execute are inserted in a pending buffer, modeled as fixed-sized + variable-sized pools. The latter has higher management overheads.
- If parent explicitly synchronizes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

Topics

- ✓ Dynamic Parallelism
- Multi-GPU Processing
- Peer Access
- Unified Virtual Memory
- PTX
- Warp-voting
- Heterogeneous Programming

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?

```
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);
}
```

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.

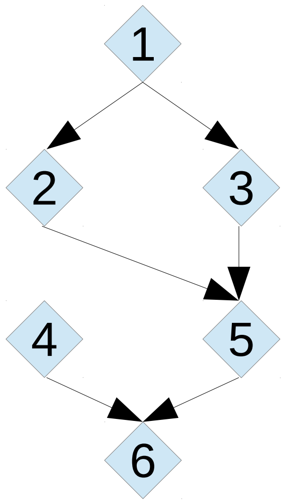
```
int main() {
        cudaStream_t s0, s1;
        cudaEvent_t e0, e1;

        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;
}
```

This ensures that K4 does not start before K1 completes.

Can be graphically modeled as a *dependence edge*.

Classwork

- Implement inter-device
- Simulate the following node represents a kerr



```
K1, s1
cudaEventRecord(e1, s1);

cudaStreamWaitEvent(s2, e1, 0);
K2, s2
cudaEventRecord(e2, s2);

cudaStreamWaitEvent(s3, e1, 0);
K3, s3
cudaEventRecord(e3, s3);

K4, s4
cudaEventRecord(e4, s4);

cudaStreamWaitEvent(s5, e2, 0);
cudaStreamWaitEvent(s5, e3, 0);
K5, s5
cudaEventRecord(e5, s5);

cudaStreamWaitEvent(s6, e4, 0);
cudaStreamWaitEvent(s6, e5, 0);
K6, s6
```

Number of events = Number of source nodes of edges
Number of waits = Number of edges

Peer Access

- In general, GPUs have different addr. spaces.
- In PA, 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);
```

All allocations from device **0** would be accessible by device **1**.

Common Memories

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

Topics

- ✓ Dynamic Parallelism
- ✓ Multi-GPU Processing
- ✓ Peer Access
- ✓ Unified Virtual Memory
- PTX
- Warp-voting
- Heterogeneous Programming

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;
}
```

```
.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 Voting

- __all(predicate);
  - If all warp threads satisfy the predicate.
- __any(predicate);
  - If any warp thread 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;
}
```

Source: voting.cu

```
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 block, but not all.
  - The contention is high.
  - Can be optimized with __ballot.
- Leader election
  - Can be thread 0 of each warp (threadIdx.x % 32 == 0)
- Leader collects warp-count.
  - __ballot() provides a mask; how do we count bits?
  - __popc(mask) provides the count.
  - __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>

__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
```

Warp Consolidation

Original code

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

Optimized code

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

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(
                    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(
                    a[threadIdx.x] == 0
                ||  threadIdx.x % 3 != 0
                );
```

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
```

Implementing Warp Voting

- Implement __any, __all, __ballot.
  - Check where you need atomics.
- Extend these intrinsics for a thread block.
  - __ballot can return the count.
- Extend across all GPU threads.
- Extend for multi-GPU case.

printf Notes

- Final formatting happens on the host.
  - Behavior is dependent on the host-side printf.
- Has a limit of 33 arguments
  - including format string.
- The associated buffer is fixed-size and circular.
  - May get overwritten if there is huge output.
- Buffer flush happens on:
  - Kernel launches, device / stream synchronization, blocking memcpy, prior to a callback, etc.
  - Note that it doesn't get flushed on program exit.

Kernel Launch Bounds

- Compiler tries to identify the register usage to generate spill-code.
- Programmer can help specify the resource usage.
- If compiler's register usage is lower, it aggressively uses more registers to hide single-thread instruction latency.
- If compiler's register usage is higher, it reduces register usage and uses more local memory.
- Kernel launch fails with more threads per block.

```
__global__ void
__launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor)
K(...) { ... }
```

Compiler Optimizations

- nvcc defaults to *-O3*
  - Performs simple optimizations (*O1*)
  - Performs aggressive intra-procedural opti. (*O2*)
  - Performs inlining (*O3*)
- Compilation time is proportional to O-level.
- Execution time is inversely proportional to O-level.

Loop Unrolling

- By default, compiler unrolls small loops with a known trip count.
- User can control unrolling using #pragma unroll.
- The directive must be placed prior to the loop.

```
__global__ void K(int *a, int N) {
        for (unsigned ii = 0; ii < N; ++ii) {
                a[ii] = ii + 1;
        }
}
```

```
BB0_1:
        add.s32         %r5, %r5, 1;
        st.global.u32   [%rd5], %r5;
        add.s64         %rd5, %rd5, 4;
        setp.lt.u32     %p2, %r5, %r3;
        @%p2 bra            BB0_1;
```

Loop Unrolling

```
__global__ void K(int *a, int N) {
        #pragma unroll
        for (unsigned ii = 0; ii < N; ++ii) {
                a[ii] = ii + 1;
        }
}
```

No change

```
__global__ void K(int *a, int N) {
        for (unsigned ii = 0; ii < N; ++ii) {
                a[ii] = ii + 1;
        }
}
```

```
BB0_1:
        add.s32          %r5, %r5, 1;
        st.global.u32    [%rd5], %r5;
        add.s64          %rd5, %rd5, 4;
        setp.lt.u32      %p2, %r5, %r3;
        @%p2 bra             BB0_1;
```

Loop Unrolling

```
__global__ void K(int *a, int N) {
        #pragma unroll 2
        for (unsigned ii = 0; ii < N; ++ii) {
                a[ii] = ii + 1;
        }
}
```

```
__global__ void K(int *a, int N) {
        for (unsigned ii = 0; ii < N; ++ii) {
                a[ii] = ii + 1;
        }
}
```

```
BB0_3:
        ...
        add.s64         %rd4, %rd1, %rd3;
        add.s32         %r20, %r22, 1;
        st.global.u32   [%rd4], %r20;
        mul.wide.u32    %rd5, %r20, 4;
        add.s64         %rd6, %rd1, %rd5;
        add.s32         %r22, %r22, 2;
        st.global.u32   [%rd6], %r22;
        add.s32         %r21, %r21, 2;
        setp.ne.s32     %p3, %r21, 0;
        @%p3 bra        BB0_3;
```

```
struct S1_t { static const int value = 4; };
template <int X, typename T2>
__device__ void foo(int *p1, int *p2) {
    #pragma unroll
    for (int i = 0; i < 12; ++i)
        p1[i] += p2[i]*2;

    #pragma unroll (X+1)
    for (int i = 0; i < 12; ++i)
        p1[i] += p2[i]*4;

    #pragma unroll 1
    for (int i = 0; i < 12; ++i)
        p1[i] += p2[i]*8;

    #pragma unroll (T2::value)
    for (int i = 0; i < 12; ++i)
        p1[i] += p2[i]*16;
}
__global__ void bar(int *p1, int *p2) {
    foo<7, S1_t>(p1, p2);
}
```

11 times

7 times

Not unrolled

3 times

Find the number of times each loop is unrolled.

Topics

- ✓ Dynamic Parallelism
- ✓ Multi-GPU Processing
- ✓ Peer Access
- ✓ Unified Virtual Memory
- ✓ PTX
- ✓ Warp-voting
- Heterogeneous Programming

Heterogeneous Programming

- Multiple types of devices work together.
- For instance, CPU and GPU
  - and not multiple GPUs.
- Heterogeneous programming does not require either of them to be parallel.
  - But together it is asynchronous (between one CPU thread and one GPU thread).
- We have already tasted heterogeneous programming.
  - CPU-GPU synchronization

Heterogeneous Programming

- In general, CPU can be parallel, GPU can be parallel, and together they all can work.
- GPU parallelization is achieved using CUDA or OpenCL or …
- CPU parallelization is achieved using OpenMP or pthreads or ...

CUDA OpenMP

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

int main() {
        #pragma omp parallel
        {
                printf("Thread started.\n");
        }

        return 0;
}
```

```
$ nvcc -Xcompiler -fopenmp -lgomp omp.cu
$ a.out
Thread started.          // 32 times
```

CUDA OpenMP

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

int main() {
        omp_set_num_threads(4);
         #pragma omp parallel
         {
                  printf("Thread started.\n");
         }

         return 0;
}
```

```
$ a.out
Thread started.            // 4 times
```

CUDA OpenMP

```
#include <stdio.h>
#include <omp.h>
#include <cuda.h>
__global__ void K() {
        printf("in K: %d\n", threadIdx.x);
}
int main() {
        omp_set_num_threads(4);
         #pragma omp parallel
         {
           K<<<1, 1>>>();
           CudaDeviceSynchronize();
         }
        return 0;
}
```

4 CPU threads each launches a kernel on the same GPU (in default stream).

CUDA OpenMP

```
#pragma omp parallel for
for (int i = 0; i < 10; ++i)
{
        K<<<1, 1>>>();
        cudaDeviceSynchronize();
}
```

- Partitions iterations across the available threads (in this case 10).
- The amount of work done remains the same.
- Chunk-size changes.
- Improves parallelism.

CUDA OpenMP

- By default, variables are assumed to be shared.
- Index variables are thread-local.
- Variables can be marked local explicitly, using `private(v1, v2, …)` construct.

Classwork

- Write a CUDA OpenMP program that
  - Creates and initializes an array on CPU.
  - Partitions the array into 10 parts.
  - Launches 10 CPU threads.
  - Each thread makes the partition accessible to the corresponding GPU kernels.
  - Each thread waits for its kernel.
  - In the end, one of the CPU threads prints the completion message.

API

```
int nthreads = omp_get_num_threads();
omp_get_thread_num();  // tid in team
omp_get_wtime(); // portable per-thread time
```

Environment Variables:

OMP_NUM_THREADS

```
#define N 100
__global__ void K(int *a, int start, int end) {
        printf("start = %d, end = %d\n", start, end);
}
int main() {
        int a[N];
        int ii;

        omp_set_num_threads(10);
        #pragma omp parallel
        {
          #pragma omp ~~parallel~~ for
          for (ii = 0; ii < N; ++ii)
                a[ii] = ii;

          int nthreads = omp_get_num_threads();
          int perthread = N / nthreads;
          int start = perthread * omp_get_thread_num();
          int end = start + perthread;
          K<<<1, 1>>>(a, start, end);
          cudaDeviceSynchronize();
        }
        printf("All over.\n");
        return 0;
}
```

Master Thread

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

int main(int argc, char *argv[]) {
     int nthreads, tid;

     #pragma omp parallel private(tid)
     {
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);

        if (tid == 0) {
          nthreads = omp_get_num_threads();
          printf("Number of threads = %d\n", nthreads);
        }

   }  /* All threads join master thread and terminate */

 }
```

OpenMP Synchronization

```
#pragma omp master
{
    Stmt;
}   // executed only by the master thread.

#pragma omp critical
{
    Stmt;
}   // executed by a thread at a time
```

OpenMP Synchronization

```
#include <omp.h>
 main(int argc, char *argv[]) {
   int x;
   x = 0;

   #pragma omp parallel shared(x)
   {
    #pragma omp critical
    x = x + 1;
   }
 }
```

Correct the program.

Reduction

```
#include <omp.h>
main(int argc, char *argv[])  {
  int   i, n, chunk;
  float a[100], b[100], result;

  n = 100;
  chunk = 10;
  result = 0.0;
  for (i=0; i < n; i++) {
    a[i] = i * 1.0;
    b[i] = i * 2.0;
  }

  #pragma omp parallel for  reduction(+:result)
     for (i = 0; i < n; i++)
      result = result + (a[i] * b[i]);

  printf("Final result= %f\n", result);
}
```

OpenMP Synchronization

#pragma omp barrier

- Global barrier across all CPU threads.

#pragma omp atomic

- Mini-critical

#pragma omp flush

- Similar to thread-fence

Classwork

- Convert the following CUDA code to equivalent OpenMP program.

```
__device__ int sum = 0;
__global__ void K(int *a) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    a[tid] = tid + 1;
    atomicAdd(&sum, a[tid]);
}
int main() {
    int *a;
    cudaMalloc(&a, sizeof(int) * 12);
    K<<<3, 4>>>(a);
    cudaDeviceSynchronize();
    return 0;
}
```

Classwork

```
#define N 12

int sum = 0;
int main() {
	int *a = (int *)malloc(sizeof(int) * N);

	#pragma omp parallel for
	for (int ii = 0; ii < N; ++ii) {
		a[ii] = ii + 1;
		sum += a[ii];
	}

	printf("%d\n", sum);
	return 0;
}
```

Source: omp5.cu

```
$ a.out
20
$ a.out
45
$ a.out
14
```

Classwork

```
#define N 12

int sum = 0;
int main() {
	int *a = (int *)malloc(sizeof(int) * N);

	#pragma omp parallel for
	for (int ii = 0; ii < N; ++ii) {
		a[ii] = ii + 1;
	#pragma omp parallel for reduction(+:sum)
		sum += a[ii];
	}

	printf("%d\n", sum);
	return 0;
}
```

error: for statement expected before ‘sum’

Classwork

```c
#define N 12

int sum = 0;
int main() {
	int *a = (int *)malloc(sizeof(int) * N);

	#pragma omp parallel for reduction(+:sum)
	for (int ii = 0; ii < N; ++ii) {
		a[ii] = ii + 1;
		sum += a[ii];
	}

	printf("%d\n", sum);
	return 0;
}
```

```
$ a.out
78
$ a.out
78
$ a.out
78
```

Share Work

- Total work to be done = N
- [0..N/2) on GPU, with N/2 threads
- [N/2..N) on CPU, with N/2 threads
- Use the same operator applied on individual elements.
- Avoid explicit memory copies.

```
#define N 10
__device__ __host__ void fun(int *a, int ii) {
        a[ii] = ii + 1;
}
__global__ void K(int *a) {
        fun(a, threadIdx.x);
}
int main() {
        int *a;
        cudaHostAlloc(&a, sizeof(int) * N, 0);
        K<<<1, N/2>>>(a);

        #pragma omp parallel for
        for (int ii = N/2; ii < N; ++ii)
                fun(a, ii);
        cudaDeviceSynchronize();

        for (int ii = 0; ii < N; ++ii)
                printf("a[%d] = %d\n", ii, a[ii]);
        return 0;
}
```

Back to 1..10

- CPU launches 5 threads (C0..C4).
- GPU launches 5 threads (G0..G4).
- They together print numbers 1..10 in sorted order.
- C0 prints 1, then G0 prints 2, then C1 prints 3, …, and finally G4 prints 10.

Barriers

- #pragma omp barrier
- There is implicit barrier at the end of parallel for.
- There is implicit barrier at the end of parallel region.
- Implicit barrier can be removed using nowait.
  - #pragma omp for nowait

master versus single

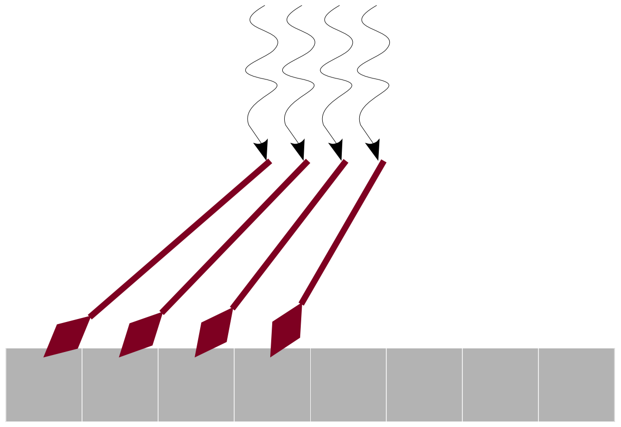
- Master block is executed by the master thread alone.
- Single block is also executed by one of the threads.
- However, it is unknown which thread would execute it.
- Critical is executed by one thread at a time, but is executed by all threads.

Sequential Order

- `#pragma omp for ordered`
- Enforces sequential order
- Useful for debugging, fallback, incremental parallelization

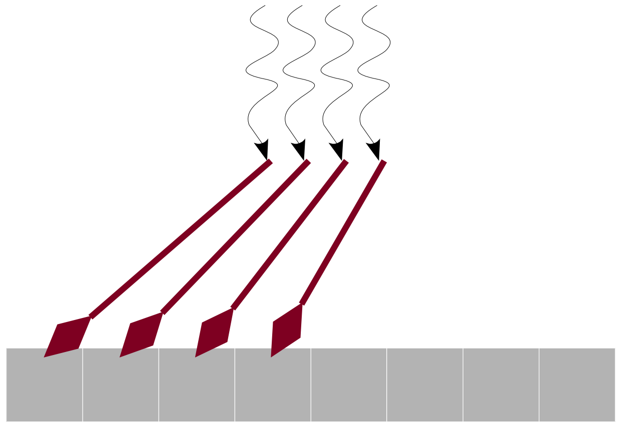
False Sharing

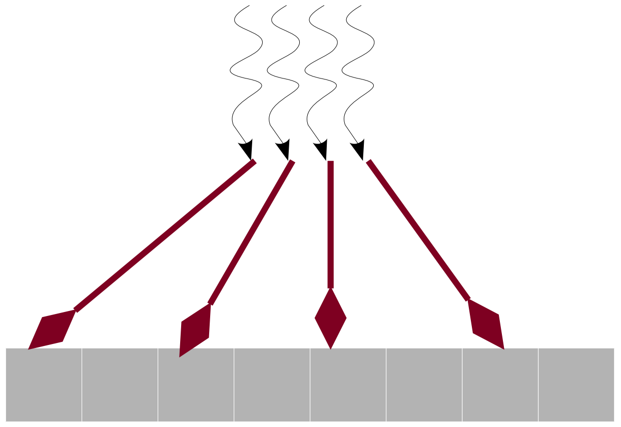
- `a[i] = i + 1;`
- Threads access nearby memory locations.
- Cache flushes unnecessarily.
- Can be reduced with striding.



False Sharing and Coalescing

- Striding and coalescing contradict each other.
- What is good for multi-core CPU may not be good for many-core GPU.
- Need data transformation between devices.


Coalesced


Strided

Schedule

- #pragma omp parallel for schedule(X)
  - static: compile-time
  - dynamic: run-time with work-queue
  - guided: special dynamic, chunk-size starts large, then shrinks

Loop-Carried Dependence

```
int i, j, A[MAX];
j = 5;

for (i = 0; i < MAX; i++) {
   j +=2;
  A[i] = big(j);
}
```

```
int i, A[MAX];

#pragma omp parallel for
for (i = 0; i < MAX; i++) {
  int j = 5 + 2*i;
  A[i] = big(j);
}
```

Loop carried dependence on j.
What is the effect of parallelization?
Is there a way to parallelize better?

OpenACC

- Directive-based approach
  - Similar to OpenMP
- Falls between auto-parallelization and hand-tuned parallelization.
  - Performance
  - Efforts

OpenACC Directives

```
#pragma acc parallel loop
for (...) { }
```

Compiler automatically generates the kernel.

```
#pragma acc parallel loop reduction(+:sum)
#pragma acc kernels
```

and others...

Needs *pgcc* compiler.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main() {
  long long int i, n = 10000000000;
  double start_time, end_time;
  double x, pi;
  double sum = 0.0;
  double step = 1.0 / (double) n;

  #pragma acc data copyin(step, sum) copyout(sum)
  {
    start_time = omp_get_wtime();

    #pragma acc kernels loop private(i, x) reduction(+:sum)
    for (i = 0; i < n; i++) {
          x = (i + 0.5) * step;
          sum +=  4.0 / (1.0 + x * x);
    }
    pi = step * sum;
    end_time = omp_get_wtime();
  }
  printf("pi = %17.15f\n", pi);
  printf("%g seconds\n", (double)(end_time - start_time));
  return 0;
}
```

Source: https://www.pgroup.com/userforum/viewtopic.php?t=4759

parallel loop versus kernels

- Requires analysis by the programmer
- Can parallelize what a compiler may miss
- Straightforward extension from OpenMP

- Compiler performs the analysis
- Can cover more code with a single directive
- Provides compiler additional leeway to optimize

OpenCL

- A framework for parallel programming of heterogeneous systes.
  - Language
  - Compiler
  - Runtime
- Standardized by Khronos group
  - Includes IBM, AMD, Intel, NVIDIA, Apple, TI, Sony, ...
- Uses extended C.

Typical OpenCL Program

- Kernels in a separate file or in a string
- CPU code (main)
  1. Read kernels from the file in a string
  2. Get platform and device information
  3. Create OpenCL context
  4. Create command queue
  5. Create memory buffer
  6. Create kernel program from source string
  7. Compile the kernel!
  8. Execute the kernel
  9. Copy results from the kernel

OpenCL Example

```
__kernel void hello(__global char* string) {
  string[0] = 'H';
  string[1] = 'e';
  string[2] = 'l';
  string[3] = 'l';
  string[4] = 'o';
  string[5] = ',';
  string[6] = ' ';
  string[7] = 'W';
  string[8] = 'o';
  string[9] = 'r';
  string[10] = 'l';
  string[11] = 'd';
  string[12] = '!';
  string[13] = '\0';
  printf("%s: %d\n", string, get_global_id(0));
}
```

opencl.cl

```
int main()
{
FILE *fp;
char fileName[] = "./opencl.cl";
char *source_str;
size_t source_size;
...
/* Load the source code containing the kernel*/
fp = fopen(fileName, "r");
if (!fp) {
  fprintf(stderr, "Failed to load kernel.\n");
  exit(1);
}
...
```

opencl.c

```
/* Get Platform and Device Info */
ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
&ret_num_devices);

/* Create OpenCL context */
context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);

/* Create Command Queue */
command_queue = clCreateCommandQueue(context, device_id, 0, &ret);

/* Create Memory Buffer */
memobj = clCreateBuffer(context, CL_MEM_READ_WRITE,MEM_SIZE *
sizeof(char), NULL, &ret);

/* Create Kernel Program from the source */
program = clCreateProgramWithSource(context, 1, (const char **)&source_str,
(const size_t *)&source_size, &ret);
```

```
/* Build Kernel Program */
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

/* Create OpenCL Kernel */
kernel = clCreateKernel(program, "hello", &ret);

/* Set OpenCL Kernel Parameters */
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobj);

/* Execute OpenCL Kernel */
size_t global_size = 8;
size_t local_size = 4;
ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global_size,
&local_size, 0, NULL, NULL);

/* Copy results from the memory buffer */
ret = clEnqueueReadBuffer(command_queue, memobj, CL_TRUE, 0,
MEM_SIZE * sizeof(char),string, 0, NULL, NULL);

/* Display Result */
puts(string);
```

OpenCL Compilation

- gcc

```
gcc   -I  /usr/local/cuda/include/
      -L /usr/local/cuda/lib64/
      -lOpenCL opencl.c
```

- a.out

```
Hello, World!: 4
Hello, World!: 5
Hello, World!: 6
Hello, World!: 7
Hello, World!: 0
Hello, World!: 1
Hello, World!: 2
Hello, World!: 3
Hello, World!
```

Learning Outcomes

- ✓ Dynamic Parallelism
- ✓ Multi-GPU Processing
- ✓ Peer Access
- ✓ Unified Virtual Memory
- ✓ PTX
- ✓ Warp-voting
- ✓ Heterogeneous Programming