Advanced CUDA programming: asynchronous execution, memory models, unified memory

January 2020

Caroline Collange
Inria Rennes – Bretagne Atlantique
https://team.inria.fr/pacap/members/collange/
caroline.collange@inria.fr
Agenda

- Asynchronous execution
  - Streams
  - Task graphs

- Fine-grained synchronization
  - Atomics
  - Memory consistency model

- Unified memory
  - Memory allocation
  - Optimizing transfers
Asynchronous execution

- By default, most CUDA function calls are *asynchronous*
  - Returns immediately to CPU code
  - GPU commands are queued and executed in-order
- Some commands are synchronous by default
  - cudaMemcpy(..., cudaMemcpyDeviceToHost)
  - Asynchronous version: cudaMemcpy\textit{Async}
- Keep it in mind when checking for errors and measuring timing!
  - Error returned by a command may be caused by an earlier command
  - Time taken by kernel\texttt{<<<>>> launch} is meaningless
- To force synchronization: cuThreadSynchronize()
Asynchronous transfers

- Overlap CPU work with GPU work

Can we do better?
Multiple command queues / streams

Diagram:
- Application: Fat binary
- CUDA Runtime
- GPU Driver
- CUDA Runtime API: cudaXxx functions
- CUDA Driver API: cuYyy functions
- User mode
- Kernel mode
- Command queues
  - push
  - pop

Push and pop operations occur between the GPU and CPU.
**Streams: pipelining commands**

- **Command queues** in OpenCL
  - Commands from the same stream run in-order
  - Commands from different streams run out-of-order

![Diagram showing pipelining commands](image)
Overlapping CPU-GPU communication and computation:
Direct Memory Access (DMA) copy engine runs CPU-GPU memory transfers in background

- Requires page-locked memory
- Some Tesla GPUs have 2 DMA engines or more: simultaneous send + receive + inter-GPU communication

Concurrent kernel execution

- Start next kernel before previous kernel finishes
- Mitigates impact of load imbalance / tail effect

Example

Serial kernel execution

```
Kernel<<<5,,,a>>>
Kernel<<<4,,,b>>>
```

Concurrent kernel execution

```
Kernel<<<5,,,a>>>
Kernel<<<4,,,b>>>
```

```c
a block 0 | a 3 | b 0 | b 3
```

```c
a 1 | a 4 | b 1
```

```c
a 2
```

```c
b 2
```

```c
b 0 | b 3
```

```c
a 1 | a 4 | b 1
```

```c
a 2
```
Page-locked memory

- By default, allocated memory is *pageable*
  - Can be swapped out to disk, moved by the OS...

- DMA transfers are only safe on *page-locked* memory
  - Fixed virtual → physical mapping
  - `cudaMemcpy` needs an intermediate copy: slower, **synchronous only**

- `cudaMallocHost` allocates page-locked memory
  - Mandatory when using streams

- Warning: page-locked memory is a limited resource!
Streams: example

- Send data, execute, receive data

```c
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice, stream[i]);
MyKernel <<<100, 512, 0, stream[i]>>>(outputDevPtr + i * size, inputDevPtr + i * size, size);
cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size, cudaMemcpyDeviceToHost, stream[i]);
```

for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice, stream[i]);
Streams: alternative implementation

cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                     size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    MyKernel<<<100, 512, 0, stream[i]>>> (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                     size, cudaMemcpyDeviceToHost, stream[i]);
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);

- Which one is better?
Events: synchronizing streams

- Schedule synchronization of one stream with another
  - Specify dependencies between tasks

```c
cudaEvent_t e;
cudaEventCreate(&e);
kernel1<<<,,a>>>();
cudaEventRecord(e, a);
cudaStreamWaitEvent(b, e);
kern2el2<<<,,b>>>();
cudaEventDestroy(e);
```

- Measure timing

```c
cudaEventRecord(start, 0);
kernel<<<>>>();
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```
Scheduling data dependency graphs

- With streams and events, we can express task dependency graphs
  - Equivalent to threads and events (e.g. semaphores) on CPU

- Example:
  - 2 GPU streams: \( a \) \( b \) and 1 CPU thread: \( \) 
  - Where should we place events?
Scheduling data dependency graphs

```
kernel1<<<,,a>>>();
cudaEventRecord(e1, a);
kernel2<<<,,b>>>();
cudaStreamWaitEvent(b, e1);
cudaMemcpyAsync(,,,b);
cudaEventRecord(e2, b);
kernell3<<<,,a>>>();
cudaEventRecord(e3, a);
cudaEventSynchronize(e2);

CPU code

cudaStreamWaitEvent(b, e3);
kernell4<<<,,b>>>();
```
Agenda

- Asynchronous execution
  - Streams
  - Task graphs

- Fine-grained synchronization
  - Atomics
  - Memory consistency model

- Unified memory
  - Memory allocation
  - Optimizing transfers
From streams and events to task graphs

- Limitations of scheduling task graphs with streams
  - Sub-optimal scheduling: GPU runtime has no vision of tasks ahead
  - Must pay various initialization overheads when launching each task

- **New** alternative since CUDA 10.0: cudaGraph API
  - Build an in-memory representation of the dependency graph offline
  - Let the CUDA runtime optimize and schedule the task graph
  - Launch the optimized graph as needed

- Two ways we can build the dependency graph
  - Record a sequence of asynchronous CUDA calls
  - Describe the graph explicitly
Recording asynchronous CUDA calls

- Capture a sequence of calls on a stream, instead of executing them
  
  > cudaGraph_t graph;
  > cudaStreamBeginCapture(stream);

  ... CUDA calls on stream

  > cudaStreamEndCapture(stream, &graph);

- Good for converting existing asynchronous code to task graphs
  
  - (Almost) no code rewrite required

- Supports any number of streams (except default stream 0)
  
  - Follows dependencies to other streams through events
  - Capture all streams that have dependency with first captured stream

- Need all recorded calls to be asynchronous and bound to a stream
  
  - CPU code needs to be asynchronous to be recorded too!
Surround asynchronous code with capture calls

cudaGraph_t graph;
cudaStreamBeginCapture(a);

erg1<<<,,a>>>();
cudaEventRecord(e1, a);
kernel2<<<,,b>>>();
cudaStreamWaitEvent(b, e1);
cudaMemcpyAsync(,,b);

erg3<<<,,a>>>();
cudaEventRecord(e3, a);
cudaLaunchHostFunc(b, cpucode, params);

cudaStreamWaitEvent(b, e3);
kernel4<<<,,b>>>();

cudaStreamEndCapture(a, &graph);

Will capture stream a, and dependent streams: b
Recording asynchronous CUDA calls

- Records only asynchronous calls: can't use immediate synchronization

```c
cudaGraph_t graph;
cudaStreamBeginCapture(a);

ekernel1<<<,,,a>>>();
cudaEventRecord(e1, a);
kernel2<<<,,,b>>>();
cudaStreamWaitEvent(b, e1);
cudan_memcpyAsync(,,,b);

kernel3<<<,,,a>>>();
cudaEventRecord(e3, a);

cudalLaunchHostFunc(b, cpucode, params);

cudaStreamWaitEvent(b, e3);
kernel4<<<,,,b>>>();

cudaStreamEndCapture(a, &graph);
```

- Make the call to CPU code asynchronous too, on stream b using `cudaLaunchHostFunc`
Describing the graph explicitly

- Add nodes to the graph: kernels, memcpy, host call...

```c
cudaGraph_t graph;
cudaGraphCreate(&graph, 0);

cudaGraphNode_t k1, k2, k3, k4, mc, cpu;

cudaGraphAddKernelNode(&k1, graph, 0, 0, // no dependency yet
paramsK1, 0);
...

cudaGraphAddKernelNode(&k4, graph, 0, 0, paramsK4, 0);

cudaGraphAddMemcpyNode(&mc, graph, 0, 0, paramsMC);

cudaGraphAddHostNode(&cpu, graph, 0, 0, paramsCPU);
```
Passing kernel parameters

- Node creation functions take parameters as a structure pointer
- e.g. for kernel calls

```c
__host__ cudaError_t
cudaGraphAddKernelNode(cudaGraphNode_t* pGraphNode,
cudaGraph_t graph, const cudaGraphNode_t* pDependencies,
size_t numDependencies, const cudaKernelNodeParams* pNodeParams)
```

```c
struct cudaKernelNodeParams
{
    void* func;             // Kernel function
    dim3 gridDim;
    dim3 blockDim;
    unsigned int sharedMemBytes;
    void **kernelParams;    // Array of pointers to arguments
    void **extra;           // (low-level alternative to kernelParams)
};
```

- `kernelParams` point to memory that will contain parameters when the graph is eventually executed
Describing the graph explicitly

- Add dependencies between nodes

```c
cudaGraph_t graph;
cudaGraphCreate(&graph, 0);

cudaGraphNode_t k1, k2, k3, k4, mc, cpu;

... Add nodes

cudaGraphAddDependencies(graph, &k1, &k3, 1); // kernel1 → kernel3
cudaGraphAddDependencies(graph, &k1, &mc, 1); // kernel1 → memcpy
cudaGraphAddDependencies(graph, &k2, &mc, 1); // kernel2 → memcpy
cudaGraphAddDependencies(graph, &mc, &cpu, 1); // memcpy → cpu
cudaGraphAddDependencies(graph, &k3, &k4, 1); // kernel3 → kernel4
cudaGraphAddDependencies(graph, &cpu, &k4, 1); // cpu → kernel4
```
Instantiating and running the graph

- Instantiate the graph to create an executable graph
- Launch executable graph on a stream

```c
cudaGraph_t graph;
... build or record graph

cudaGraphExec_t exec;
cudaGraphInstantiate(&exec, graph, 0, 0, 0);

cudaGraphLaunch(exec, stream);

cudaStreamSynchronize(stream);
```

- Once a graph is instantiated, its topology cannot be changed
- Kernel/memcpy/call... **parameters** can still be changed using `cudaGraphExecUpdate` or `cudaGraphExec{Kernel,Host,Mempy,Memset}NodeSetParams`
Agenda

- Asynchronous execution
  - Streams
  - Scheduling dependency graphs

- Fine-grained synchronization
  - Atomics
  - Memory consistency model

- Unified memory
  - Memory allocation
  - Optimizing transfers
Inter-thread/inter-warp communication

- **Barrier**: simplest form of synchronization
  - Easy to use
  - Coarse-grained

- **Atomics**
  - Fine-grained
  - Can implement *wait-free* algorithms
  - May be used for blocking algorithms (locks) among *warps* of the *same block*
  - From Volta (CC≥7.0), also among threads of the same warp

- **Communication through memory**
  - Beware of consistency
Atomics

- Read, modify, write in one operation
  - Cannot be mixed with accesses from other thread

- Available operators
  - Arithmetic: \texttt{atomic\{Add,Sub,Inc,Dec\}}
  - Min-max: \texttt{atomic\{Min,Max\}}
  - Synchronization primitives: \texttt{atomic\{Exch,CAS\}}
  - Bitwise: \texttt{atomic\{And,Or,Xor\}}

- On global memory, and shared memory

- Performance impact in case of contention
  - Atomic operations to the same address are serialized
Example: reduction

- After local reduction inside each block, use atomics to accumulate the result in global memory.

Complexity?

Time including kernel launch overhead?
Example: compare-and-swap

- Use case: perform an arbitrary associative and commutative operation atomically on a single variable
- atomicCAS(p, old, new) does atomically
  - if *p == old then assign *p ← new, return old
  - else return *p

```c
__shared__ unsigned int data;
unsigned int old = data; // Read once
unsigned int assumed;
do {
    assumed = old;
    newval = f(old, x); // Compute
    old = atomicCAS(&data, old, newval); // Try to replace
} while(assumed != old); // If failed, retry
```
Memory consistency model

- Nvidia GPUs (and compiler) implement a relaxed consistency model
  - No global ordering between memory accesses
  - Threads may not see the writes/atomics in the same order

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>write A</td>
<td>read B</td>
</tr>
<tr>
<td>(valid data)</td>
<td>New value of B</td>
</tr>
<tr>
<td>write B</td>
<td>read A</td>
</tr>
<tr>
<td>(ready flag)</td>
<td>Old value of A</td>
</tr>
<tr>
<td></td>
<td>(invalid data)</td>
</tr>
</tbody>
</table>

- Need to enforce explicit ordering

Details and precise specifications of CUDA memory consistency model:
Enforcing memory ordering: fences

- `__threadfence_block`
- `__threadfence`
- `__threadfence_system`
  - Make writes preceding the fence appear before writes following the fence for the other threads at the block / device / system level
  - Make reads preceding the fence happen after reads following the fence

```
T1
write A
__threadfence()
write B
T2
read B  Old/New value of B
__threadfence()
read A  New value of A
```

- Declare shared variables as `volatile`
to make writes visible to other threads (prevents compiler from removing “redundant” read/writes)
- `__syncthreads` implies `__threadfence_block`
Floating-point atomics

- `atomicAdd` supports floating-point operands

Remember
- Floating-point addition is not associative
- Thread scheduling is not deterministic

Without FP atomics
- Evaluation order is independent of thread scheduling
- You should expect deterministic result for a fixed combination of GPU, driver, and runtime parameters (thread block size, etc.)

With FP atomics
- Evaluation order depends on thread scheduling
- You will get different answers from one run to the other
Agenda

- Asynchronous execution
  - Streams
  - Scheduling dependency graphs

- Fine-grained synchronization
  - Atomics
  - Memory consistency model

- Unified memory
  - Memory allocation
  - Optimizing transfers
Unified memory and other features

OK, we lied

- You were told
  - CPU and GPU have distinct memory spaces
  - Blocks cannot communicate
  - You need to synchronize threads inside a block
- This is the least-common denominator across all CUDA GPUs

- This is not true (any more). We now have:
  - Device-mapped, Unified virtual address space, Unified memory
  - Global and shared memory atomics
  - Dynamic parallelism
  - Warp-synchronous programming with intra-block thread groups
  - Grid level and multi-grid level thread groups
Unified memory

- Allocate memory using `cudaMallocManaged`
  - Pointer is accessible from both CPU and GPU
  - The CUDA runtime will take care of the transfers
  - No need for `cudaMemcpy` any more
  - Behaves as if you had a single memory space

- Suboptimal performance: on-demand page migration
  - Optimization: perform copies in advance using `cudaMemPrefetchAsync` when access patterns are known
VectorAdd using unified memory

```c
int main()
{
    int numElements = 50000;
    size_t size = numElements * sizeof(float);

    float *A, *B, *C;
    cudaMallocManaged((void **)&A, size);
    cudaMallocManaged((void **)&B, size);
    cudaMallocManaged((void **)&C, size);
    Initialize(A, B);

    int blocks = numElements;
    vectorAdd2<<<blocks, 1>>>(A, B, C);
    cudaDeviceSynchronize();
    Display(C);

    cudaFree(A);
    cudaFree(B);
    cudaFree(C);
}
```

Explicit CPU-GPU synchronization is now mandatory! Do not forget it. *(We all make this mistake once)*
How it works: on-demand paging

- Managed memory pages mapped in both CPU and GPU spaces
  - Same virtual address
  - Not necessarily allocated in physical memory
- Typical flow
  - **1.** Data allocated in CPU memory
How it works: on-demand paging

- Managed memory pages mapped in both CPU and GPU spaces
  - Same virtual address
  - Not necessarily allocated in physical memory
- Typical flow
  - 1. Data allocated in CPU memory
  - 2. GPU code touches unallocated page, triggers page fault
How it works: on-demand paging

- Managed memory pages mapped in both CPU and GPU spaces
  - Same virtual address
  - Not necessarily allocated in physical memory
- Typical flow
  - 1. Data allocated in CPU memory
  - 2. GPU code touches unallocated page, triggers page fault
  - 3. Page fault handler allocates page in GPU mem, copies contents
How it works: on-demand paging

- Managed memory pages mapped in both CPU and GPU spaces
  - Same virtual address
  - Not necessarily allocated in physical memory

- Typical flow
  - 1. Data allocated in CPU memory
  - 2. GPU code touches unallocated page, triggers page fault
  - 3. Page fault handler allocates page in GPU mem, copies contents
  - 4. If GPU modifies page contents, invalidate CPU copy
     Next CPU access will cause data to be copied back from GPU mem
Prefetching data

On-demand paging has overhead

- Solution: load data in advance using `cudaMemPrefetchAsync`

```c
...
cudaStream_t stream;
cudaStreamCreate(&stream);

cudaMemPrefetchAsync(A, size, gpuId, stream);
cudaMemPrefetchAsync(B, size, gpuId, stream);

vectorAdd2<<<numElements, 1, 0, stream>>>(A, B, C);

cudaMemPrefetchAsync(C, size, cudaCpuDeviceId, stream);

cudaDeviceSynchronize();
Display(C);
...
```

- Performance similar to manual memory management
  - Supports asynchronous copies, tolerates sloppy synchronization
Controlling data placement

- System may have multiple GPU memory spaces
  - Specify destination of prefetch
    
    ```
    cudaMemPrefetchAsync(A, size, gpuId, s);
    ```

- Diagram showing CPU, HOST memory, GPU0, GPU1, and their device memories.
References

- Nikolay Sakharnykh. *Maximizing Unified Memory Performance in CUDA*. 
  [Link](https://devblogs.nvidia.com/parallelforall/maximizing-unified-memory-performance-cuda/)