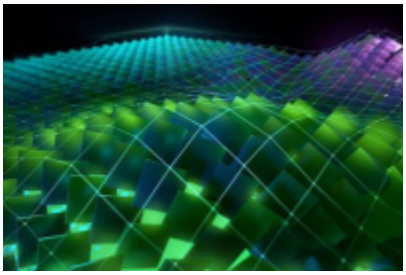
Controlling Data Movement to Boost Performance on the NVIDIA Ampere Architecture | NVIDIA Developer Blog

https://developer.nvidia.com/blog/controlling-data-movement-to-boost-performance-on-ampere-architecture/

Matthieu Tardy Wed Apr, 14 08:41

The NVIDIA Ampere architecture provides new mechanisms to control data movement within the GPU and CUDA 11.1 puts those controls into your hands. These mechanisms include asynchronously copying data into shared memory and influencing residency of data in L2 cache.



This post walks through how to use the asynchronous copy feature, and how to set up your algorithms to overlap asynchronous copies with computations.

Applications stage data through shared memory

Applications with large data and computational intensity on that data are accelerated by copying data from global to shared memory, performing computations on data in shared memory, and copying results back to global memory.

- Linear algebra algorithms copy a submatrix of a global matrix to shared memory, factorize the submatrix while in shared memory, and then copy the updated submatrix back into the global matrix.
- Finite difference algorithms copy a subgrid of a global grid into shared memory, compute with that subgrid, and then copy the subgrid back into the global grid.

Asynchronous data movement

You've long had the ability to asynchronously copy data between CPU memory and GPU global memory using `cudaMemcpyAsync`. For more information, see the Developer Blog post from 2012, How to Overlap Data Transfers in CUDA C/C++.

CudaDMA was an early effort to give developers asynchronous data movement between global and shared memory. CudaDMA uses extra data movement warps dedicated to copy operations while primary warps perform computations. With the NVIDIA Ampere architecture, you can now asynchronously copy data between GPU global memory and shared memory by using `cuda::memcpy_async` and not tie up threads to shepherd data movement.

These asynchronous data movement features enable you to overlap computations with data movement and reduce total execution time. With `cudaMemcpyAsync`, data movement between CPU memory and GPU global memory can be overlapped with kernel execution. With `cuda::memcpy_async`, data movement from GPU global memory to shared memory can be overlapped with thread execution.

A better journey through the memory hierarchy

Prior to `cuda::memcpy_async`, copying data from global to shared memory was a two-step process. First, a thread block copied data from global memory into registers and then the thread block copied that data from registers into shared memory. This resulted in the data taking a long journey through the memory hierarchy.

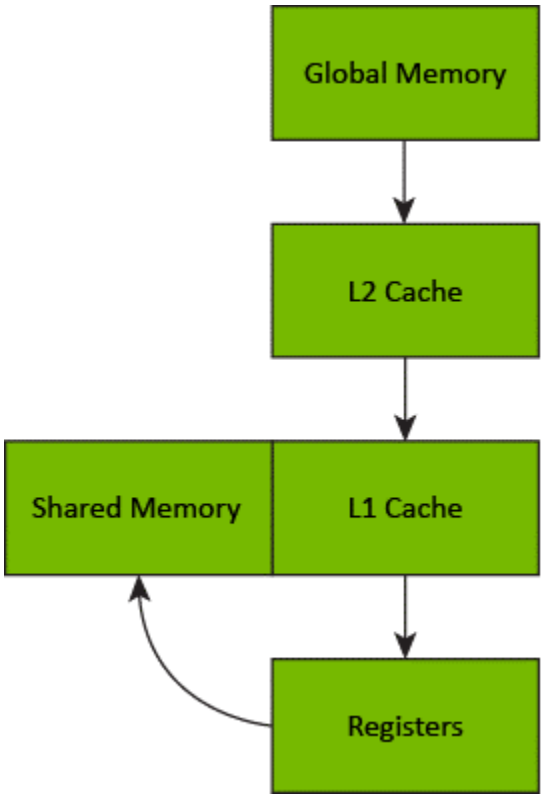


Figure 1. A long journey through the memory hierarchy.

With `cuda::memcpy_async`, the thread block no longer stages data through registers, freeing the thread block from the task of moving data and freeing registers to be used by computations.

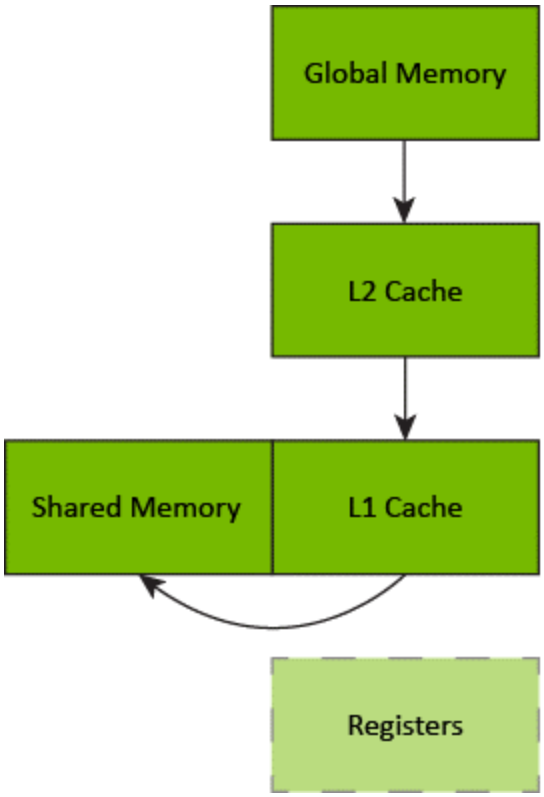


Figure 2. A better journey through the memory hierarchy.

Stages of asynchronous copy operations

Prior to `cuda::memcpy_async`, a thread block copied a batch of data from global to shared memory, computed on that batch, and then iterated to the next batch. A batch can be a contiguous region of memory or be scattered into a data structure such as into the boundary of a finite difference grid. Each thread within a thread block copied one or more elements of the batch and then all threads synchronized (`_syncthreads` or `cooperative_group::sync`) to wait for all element-copy operations to complete.

The pattern for asynchronously copying data is similar. Each thread calls `cuda::memcpy_async` one or more times to submit an asynchronous copy operation for elements within a batch and then all threads wait for the submitted copy operations to complete. Asynchronous data movement enables multiple batches to be "in flight" at the same time.

A thread block can use asynchronous data movement to pipeline (for example, double buffer) its iteration through a large data structure by submitting *N* stages of asynchronous data movement, waiting for the oldest stage to complete, computing with that batch of shared memory, and submitting a new stage before waiting for the next oldest batch to complete.

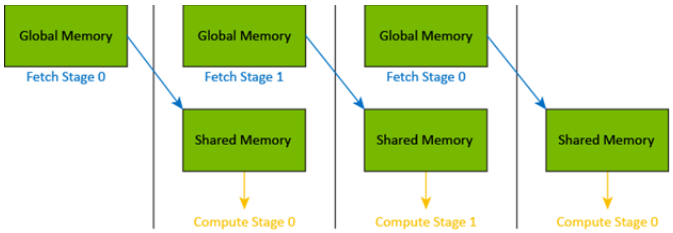


Figure 3. A sequence of asynchronous copy batches and computations pipelined in two stages.

Copy and compute: From synchronous to pipelined

You work through code changes to a simple example kernel that copies data from global to shared memory and then computes on that data. With the final code change, the kernel is overlapping the asynchronous copy of data with computations.

Staging data through shared memory

Most applications using shared memory to perform computation on a subset of a larger data set can be represented by the following algorithmic pattern. For each subset of the dataset:

1. Fetch the subset datum from global memory.
2. Store it in shared memory.
3. Perform some computation on it and eventually copy it back to global memory.

The following code example is a direct implementation of such an algorithm.

```
#include <cooperative_groups.h>

template <typename T>
__global__ void example_kernel(T * global1, T * global2, size_t subset_count)
{
    extern __shared__ T shared[];
    auto group = cooperative_groups::this_thread_block();

    for (size_t subset = 0; subset < subset_count; ++subset) {
        shared[group.thread_rank()               ] = global1[subset * group.size() +
group.thread_rank()];
        shared[group.size() + group.thread_rank()] = global2[subset * group.size() +
group.thread_rank()];

        group.sync(); // Wait for all copies to complete

        compute(shared);

        group.sync();
    }
}
```

Introducing asynchronous copies

For trivially copyable types, this algorithm can be straightforwardly improved using the new Ampere-accelerated CUDA 11.1 facilities.

```
#include <cooperative_groups.h>
#include <cooperative_groups/memcpy_async.h>

template <typename T>
__global__ void example_kernel(T * global1, T * global2, size_t subset_count)
{
    extern __shared__ T shared[];
    auto group = cooperative_groups::this_thread_block();

    for (size_t subset = 0; subset < subset_count; ++subset) {
        cooperative_groups::memcpy_async(group, shared,
                                         &global1[subset * group.size()], sizeof(T) *
group.size());
        cooperative_groups::memcpy_async(group, shared + group.size(),
                                         &global2[subset * group.size()], sizeof(T) *
group.size());

        cooperative_groups::wait(group); // Wait for all copies to complete

        compute(shared);

        group.sync();
    }
}
```

Here, you use `cooperative_groups::memcpy_async` paired with `cooperative_groups::wait` as a drop-in replacement for `memcpy` and `cooperative_groups::group::sync`.

This new version has several advantages:

- Asynchronous memcpy does not use any registers, which means less register pressure and better occupancy
- Asynchronous memcpy allows for optimal data copies without involving the compiler's dynamic loop unrolling

Arrive-wait barrier interoperability

The CUDA 11.1 `memcpy_async` APIs also offer the possibility of synchronizing asynchronous data transfers using asynchronous barriers.

For more information about asynchronous barriers, see cuda/std/barrier, cuda/barrier in the libcu++ documentation.

```
#include <cooperative_groups.h>
#include <cuda/barrier>

template <typename T>
__global__ void example_kernel(T * global1, T * global2, size_t subset_count)
{
    extern __shared__ T shared[];
    auto group = cooperative_groups::this_thread_block();

    // Create a synchronization object (C++20 barrier)
    __shared__ cuda::barrier<cuda::thread_scope::thread_scope_block> barrier;
    if (group.thread_rank() == 0) {
        init(&barrier, group.size());
    }
    group.sync();

    for (size_t subset = 0; subset < subset_count; ++subset) {
        cuda::memcpy_async(group, shared,
                           &global1[subset * group.size()], sizeof(T) * group.size(),
barrier);
        cuda::memcpy_async(group, shared + group.size(),
                           &global2[subset * group.size()], sizeof(T) * group.size(),
barrier);

        barrier.arrive_and_wait(); // Wait for all copies to complete

        compute(shared);

        barrier.arrive_and_wait();
    }
}
```

Overlapping global-to-shared copies with compute

Finally, with some limited restructuring, you can asynchronously prefetch data for subset *N*+1 while computing subset *N* using a two-stage `cuda::pipeline` statement.

```
#include <cooperative_groups.h>
#include <cuda/pipeline>

template <typename T>
__global__ void example_kernel(T * global1, T * global2, size_t subset_count)
{
    extern __shared__ T s[];
    constexpr unsigned stages_count = 2;
    auto group = cooperative_groups::this_thread_block();
    T * shared[stages_count] = { s, s + 2 * group.size() };

    // Create a synchronization object (cuda::pipeline)
    __shared__ cuda::pipeline_shared_state<cuda::thread_scope::thread_scope_block,
stages_count> shared_state;
    auto pipeline = cuda::make_pipeline(group, &shared_state);

    size_t fetch;
    size_t subset;
    for (subset = fetch = 0; subset < subset_count; ++subset) {
        // Fetch ahead up to stages_count subsets
        for (; fetch < subset_count && fetch < (subset + stages_count); ++fetch ) {
            pipeline.producer_acquire();
            cuda::memcpy_async(group, shared[fetch % 2],
                               &global1[fetch * group.size()], sizeof(T) * group.size(),
pipeline);
            cuda::memcpy_async(group, shared[fetch % 2] + group.size(),
                               &global2[fetch * group.size()], sizeof(T) * group.size(),
pipeline);
            pipeline.producer_commit(); // Commit the fetch-ahead stage
        }
        pipeline.consumer_wait(); // Wait for 'subset' stage to be available

        compute(shared[subset % 2]);

        pipeline.consumer_release();
    }
}
```

This pipelining scheme enables the improvement of two aspects compared to the previous version:

- Data-transfer synchronization points are now explicitly specified using `pipeline::producer_commit`
- Global-to-shared data transfers are overlapped with compute work

Summary

CUDA 11.1 provides a foundational development environment for building applications with the NVIDIA Ampere GPU architecture. You can access all the new features of CUDA 11.1 on either powerful server platforms built on the NVIDIA A100 or consumer GPUs with the GeForce RTX-30 series or Quadro RTX series. Use CUDA 11.1 to take control of your data movement.

Get started today by downloading CUDA 11.1:

Figure 4. Different ways to get CUDA 11.1.