Maximizing GPU Throughput Across Multiple Streams – Tips and Tricks

Chuck Seberino
Roche Sequencing Solutions, Santa Clara
Discussion Today

- Why use GPU streams?
- Stream Basics
- Example use cases
- cudaMemcpyAsync
- Custom Thrust allocator

Examples used in this presentation is available at:
https://github.com/chuckseberino/CCT.git
Why Use GPU Streams?

• Use streams when you have more than one kernel that can be executed simultaneously
  • Could be several compute tasks for an aggregated result
  • Could be completely independent work products

• Better utilization of resources – shared memory, compute, thread blocks

• Provides more opportunities for kernel scheduler to insert more work when other kernels stall
Basics of Stream Usage

Create additional streams:
• cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking)

Issue kernel/CUDA calls on proper stream:
• kernel<<grid, block, shm, stream>>>(args)
• cudaMemcpyAsync(dst, src, size, kind, stream)

Create and use events for synchronization:
• cudaEventCreate(), cudaEventRecord(), cudaStreamWaitEvent()

When using more than one stream, never use default stream:
• Remove implicit synchronization with default stream
• Makes it easier to debug default stream problems
• Helps to identify and fix synchronization bugs
• Able to verify in NVVP correct behavior
First Priority – Schedule “Enough” Work

• Make sure there are always 16-32x the number of threads queued
  • 4,000 cores = 64k to 128k threads of work
  • Provides enough work to allow the kernel scheduler to maximize functional units and hide memory latency.

• What if my kernel doesn’t use that much parallelism?

• What if my kernel uses (much) more than 32x?
  • Limited return or even degradation in performance
  • Reduce parallelism by making “fatter” threads
Example 1 – Combine Components

**Problem:** One or more kernels don’t individually create enough work, but they are independent calculations

**Solution:** Run them concurrently and synchronize their completion

- Create a separate stream for each component
- Place an event record in each stream after kernel call
- Have the aggregation stream wait on all event records of component streams

- Events work across GPU devices and CPU threads
  - Make sure that a `cudaStreamWaitEvent()` is issued after the `cudaEventRecord()` has been placed in the stream.
  - Particularly important when working across CPU threads.
    - Use CPU synchronization primitives to guarantee order.
Example 1 - Parallelize Along Work Components

- Kernel{1-4} create independent sub-results that are aggregated in Kernel0.

- Increased utilization of GPU!
Example 2 – “Too Much” Parallelism

- Column sum operation with 32M elements
  - Run on Quadro P6000 with 3840 cores

This example gets slower with increased threads!
Example 3 – Resource Utilization

**Problem:** One kernel requires large amount of shared memory, limiting occupancy
- Maxwell & Pascal have 48KB or 64KB of shared memory
  - A block size of 1024 gives *only 48(64) bytes of memory per thread* - *12(16) floats*
  - Reduce block size to get more memory per thread
  - 4x increase in shared memory per thread requires 4x reduction in occupancy

**Solution:** Given that another independent kernel is available that requires no shared memory, run it in a separate stream

Examples – median, percentile, sort, histogram, transpose
cudaMemcpyAsync Potential Pitfall

• From CUDA C Best Practices Guide Chapter 9.1:
  • “In contrast with cudaMemcpy(), the asynchronous transfer version requires pinned host memory …”

• What happens if I try to use cudaMemcpyAsync() with non-pinned memory?

• Calling cudaMemcpyAsync() with pageable memory works, but …
  • Copy operation gets serialized on GPU along with kernel launches - no copy engine overlap with kernels
  • Host doesn’t block on call though
  • Can examine in Visual Profiler
cudaMemcpyAsync Pinned
... vs. cudaMemcpyAsync Paged
Using Thrust

• Thrust is a great API that provides STL-like primitives
  • Because it behaves like standard algorithms, it is also limited in how it passes data back to the caller.
  • If a thrust function requires temporary memory, OR it passes back a result as the return value, then it will allocate and free CUDA memory

```
cudaMalloc/cudaFree every time! Serializes kernels!
```
Be Careful of Thrust Allocations!

- By using a custom allocator, you can control creation and deletion.

Calls `cudaMalloc` once the first time, then reuses on subsequent calls.
General Practice to Keep GPU Busy

1. Provide enough work for the GPU
   • Create ~16x more threads than physical cores to provide enough opportunities for the scheduler to hide latency.

2. Use multiple streams to increase utilization of resources
   • Balance ALU, Shared Memory, I/O

3. Minimize warp divergence
   • Multiple streams do not help divergence. Conditional code gets disabled by thread mask
Thank You

• Source code is available:
  • [https://github.com/chuckseberino/CCT](https://github.com/chuckseberino/CCT)
    • GPU wrapper
    • Custom Thrust allocator (per stream)
    • Examples used in this presentation

• We are hiring GPU developers!