Maximizing GPU Efficiency in Extreme Throughput Applications

The Fairmont San Jose | October 2, 2009, 2:00PM | Joe Stam
Motivation

• GPUs have dedicated memory which has 5-10X the bandwidth of CPU memory, this is a tremendous advantage

• New developers are sometimes discouraged by the perceived overhead of transferring data between GPU and CPU memory.

Today we’ll show how to properly transfer data in high throughput applications, and reduce or eliminate the transfer burden.
AGENDA

- Asynchronous APIs
- Data Acquisition
- CUDA Streams
- “Zero-Copy”
Typical Approach

1. **Copy Data from CPU Memory to GPU Memory**
2. **Run CUDA Kernel(s)**
3. **Copy Data from GPU memory to CPU Memory**

- **PCIe (5 GB/s)**
- **5-10 GB/s**
- **50-80 GB/s**

*Averaged observed bandwidth*
Synchronous Functions

• Standard CUDA C functions are **Synchronous**
• Kernel launches are:
  – Runtime API: Asynchronous
  – Driver API: `cuLaunchGrid()` or `cuLaunchGridAsync()`
• Synchronous functions block on any prior asynchronous kernel launches
Example

cudaMemcpy(...);

myKernel<<<grid, block>>>(...);

cudaMemcpy(...);

Doesn’t return until copy is complete

Returns immediately

Waits for myKernel to complete, then starts copying. Doesn’t return until copy is complete.

cudaDeviceSetFlags() function sets behavior. Tradeoff between CPU cycles and response speed

• cudaDeviceScheduleSpin
• cudaDeviceScheduleYield
• cudaDeviceBlockingSync

Driver API has equivalent context creation flags
Asynchronous APIs

• All Memory operations can also be asynchronous, and return immediately

• Memory must be allocated as ‘pinned’ using
  – cuMemHostAlloc()
  – cudaHostAlloc()
  – Older version of these functions cuMemAllocHost() cudaMallocHost() also work, but don’t have option flags

PINNED memory allows direct DMA transfers by the GPU to and from system memory. It’s locked to a physical address
Asynchronous APIs (Cont.)

• Copies & Kernels are queued up in the GPU
• Any launch overhead is overlapped

• Synchronous calls should be done outside critical sections — some of these are expensive!
  – Initialization
  – Memory allocations
  – Stream / Event creation
  – Interop resource registration
Example

cudaMemcpyAsync (void * dst,
void * src,
size_t count,
enum cudaMemcpyKind kind,
cudaStream_t stream)

cudaMemcpyAsync (…);
myKernel<<grid,block>>>(…);
cudaMemcpyAsync (…);

CPU does other stuff here

cudaThreadSynchronize ();

More on streams soon, for now assume stream = 0

Returns immediately
Returns immediately
Returns immediately
Waits for everything on the GPU to finish, then returns
Events Can Be Used to Monitor Completion

- `cudaEvent_t / CUevent`
  - Created by `cudaEventCreate() / cuEventCreate()`

```c
cudaEvent_t HtoDdone;
cudaEventCreate(&HtoDdone, 0);
cudaMemcpyAsync(dest, source, bytes, cudaMemcpyHostToDevice, 0);
cudaEventRecord(HtoDdone);
myKernel<<<grid, block>>>(...);
cudaMemcpyAsync(dest, source, bytes, cudaMemcpyDeviceToHost, 0);
cudaEventSynchronize(HtoDdone);
```

CPU can do stuff here

```c
cudaThreadSynchronize();
```

Waits just for everything before `cuEventRecord(HtoDdone)` to complete, then returns

The first memory copy is done, so the memory at `source` could be used again by the CPU

Waits for everything on the GPU to finish, then returns
Acquiring Data From an Input Device
Strategy: Overlap Acquisition With Transfer
Strategy: Overlap Acquisition With Transfer

- Allocate 2 pinned CPU buffers, ping-pong between them

```c
int bufNum = 0;
void * pCPUbuf[2];
... Allocate buffers
while (!done)
{
    cudaMemcpyAsync(pGPUbuf, pCPUbuf[(bufNum+1)%2], size,
                    cudaMemcpyHostToDevice, 0);
    myKernel1<<<...>>>(GPUbuf...);
    myKernel2<<<...>>>(GPUbuf...);
    ... other GPU stuff, all asynchronous
    GrabMyFrame(pCPUbuf[bufNum]);
    ... other CPU stuff
    cudaMemcpyAsync(pGPUbuf, pCPUbuf[(bufNum+1)%2], size,
                    cudaMemcpyHostToDevice, 0);
    myKernel1<<<...>>>(GPUbuf...);
    myKernel2<<<...>>>(GPUbuf...);
    ... other GPU stuff, all asynchronous
    GrabMyFrame(pCPUbuf[bufNum]);
    ... other CPU stuff
    cudaMemcpyAsync(pGPUbuf, pCPUbuf[(bufNum+1)%2], size,
                    cudaMemcpyHostToDevice, 0);
    myKernel1<<<...>>>(GPUbuf...);
    myKernel2<<<...>>>(GPUbuf...);
    ... other GPU stuff, all asynchronous
    bufNum++; bufNum %= 2;
}
```
CUDA Streams

- NVIDIA GPUs with Compute Capability >= 1.1 have a dedicated DMA engine
- DMA transfers over PCIe can be concurrent with CUDA kernel execution*
- Streams allows independent concurrent in-order queues of execution
  - cudaStream_t, CUstream
  - cudaMemcpy2DAsync cannot overlap.
- Multiple streams exist within a single context, they share memory and other resources

*1D Copies only! cudaMemcpy2DAsync cannot overlap.
Stream Parameter

• All Async function varieties have a stream parameter

• Runtime Kernel Launch
  
  <<< GridSize, BlockSize, SMEM Size, Stream>>>

• Driver API
  
  cuLaunchGridAsync(function, width, height, stream)

• Copies & Kernel launches with the same stream parameter execute in-order
CUDA Streams

Independent Tasks

<table>
<thead>
<tr>
<th>TASK A</th>
<th>TASK B</th>
</tr>
</thead>
<tbody>
<tr>
<td>COPY A1</td>
<td>COPY B1</td>
</tr>
<tr>
<td>KERNEL A1</td>
<td>COPY B2</td>
</tr>
<tr>
<td>KERNEL A2</td>
<td>KERNEL B1</td>
</tr>
<tr>
<td>KERNEL A3</td>
<td>COPY B3</td>
</tr>
<tr>
<td>COPY A2</td>
<td>COPY B4</td>
</tr>
</tbody>
</table>

Scheduling on GPU

Copy Engine

- COPY A1
- COPY B1
- COPY B2
- COPY A2
- COPY B3
- COPY B4

Compute Engine

- KERNEL A1
- KERNEL A2
- KERNEL A3
- KERNEL B1

GPU TECHNOLOGY CONFERENCE

© 2009 NVIDIA CORPORATION
Avoid Serialization!

**WRONG WAY!**

CudaMemcpyAsync(A1..., StreamA);
KernelA1<<<..., StreamA>>>();
KernelA2<<<..., StreamA>>>();
KernelA3<<<..., StreamA>>>();
CudaMemcpyAsync(A2..., StreamA);

CudaMemcpyAsync(B1..., StreamB);
CudaMemcpyAsync(B2..., StreamB);
KernelB1<<<..., StreamB>>>();
CudaMemcpyAsync(B2..., StreamB);
CudaMemcpyAsync(B2..., StreamB);

• Engine queues are filled in the order code is executed
**CORRECT WAY!**

CudaMemcpyAsync(A1..., StreamA);
KernelA1<<<..., StreamA>>>();
KernelA2<<<..., StreamA>>>();
KernelA3<<<..., StreamA>>>();

CudaMemcpyAsync(B1..., StreamB);
CudaMemcpyAsync(B2..., StreamB);
KernelB1<<<..., StreamB>>>();

CudaMemcpyAsync(A2..., StreamA);
CudaMemcpyAsync(B2..., StreamB);
CudaMemcpyAsync(B3..., StreamB);
CudaMemcpyAsync(B4..., StreamB);

**CUDA Programming**

**CUDA** is a parallel computing platform and programming model created by **NVIDIA**. It enables **general-purpose** computing on **graphics processing units** (GPUs) inside **graphics cards**. This allows developers to use **CUDA** to harness the power of the GPU for general-purpose computing tasks, such as data analysis, machine learning, and scientific simulations.

In this diagram, we see the correct way to order the stream code for computations involving both **Stream A** and **Stream B**. The *Copy Engine* and *Compute Engine* are represented by different columns, indicating the order in which the code should be executed to ensure efficient and correct data transfer and processing.

- **Copy Engine** includes the operations for copying data between different streams or devices.
- **Compute Engine** involves the execution of kernel functions on the GPU.

The diagram shows how to manage data transfers and kernel executions in a way that optimizes performance by minimizing idle time and maximizing the use of the GPU's resources.

The diagram also highlights the importance of **asynchronous** operations, denoted by `CudaMemcpyAsync`, which allow for overlapping computation and data transfer, thereby improving the overall efficiency of the code execution.
Revisit Our Data I/O Example

Add 3-way Overlap:
- Acquisition
- CPU-GPU transfer
- Compute
3-Way Overlap

- As before, allocate two CPU buffers
- Also allocate two GPU buffers

```c
int bufNum = 0;
void * pCPUbuf[2];
void * pGPUbuf[2];
cudaStream_t copyStream;
cudaStream_t computeStream;

// Allocate Buffers
cudaHostAlloc(&pCPUbuf[0], size, 0);
cudaHostAlloc(&pCPUbuf[1], size, 0);
cudaMalloc(&pGPUbuf[0], size, 0);
cudaMalloc(&pGPUbuf[1], size, 0);

// Create Streams
cudaStreamCreate(&copyStream, 0);
cudaStreamCreate(&computeStream, 0);
```
while (!done)
{
    cudaMemcpyAsync(pGPUbuf[bufNum], pCPUbuf[(bufNum+1)%2], size,
                    cudaMemcpyHostToDevice, copyStream);
    myKernel1<<<gridSz, BlockSz, 0, computeStream>>>(pGPUbuf[(bufNum+1)%2]...);
    myKernel2<<<gridSz, BlockSz, 0, computeStream>>>(pGPUbuf[(bufNum+1)%2]...);
    ... other GPU stuff, all asynchronous
    GrabMyFrame(pCPUbuf[bufNum]);
    ... other CPU stuff
    cudaThreadSynchronize();
    bufNum++; bufNum %= 2;
}
What About Readback?
while (!done)
{
    cudaMemcpyAsync(pGPUbuf[bufNum], pCPUbuf[(bufNum+1)%3], size, cudaMemcpyHostToDevice, copyStream);

    cudaMemcpyAsync(pGPUbuf[bufNum+2], pCPUbuf[(bufNum+2)%3], size, cudaMemcpyDeviceToHost, copyStream);

    myKernel1<<<gridSz, BlockSz, 0, computeStream>>>(pGPUbuf[(bufNum+1)%3]...);
    myKernel2<<<gridSz, BlockSz, 0, computeStream>>>(pGPUbuf[(bufNum+1)%3]...);

    ... other GPU stuff, all asynchronous

    GrabMyFrame(pCPUbuf[bufNum]);
    ... other CPU stuff

    cudaThreadSynchronize();
    bufNum++; bufNum %= 3;
}
while (!done)
{
    cudaMemcpyAsync(pGPUbuf[bufNum], pCPUbuf[(bufNum+1) % 3], size, cudaMemcpyHostToDevice, uploadStream);
    cudaMemcpyAsync(pGPUbuf[bufNum+2], pCPUbuf[(bufNum+2) % 3], size, cudaMemcpyDeviceToHost, downloadStream);
    myKernel1<<<gridSz, BlockSz, 0, computeStream>>>(pGPUbuf[(bufNum+1) % 3]...);
    myKernel2<<<gridSz, BlockSz, 0, computeStream>>>(pGPUbuf[(bufNum+1) % 3]...);
    ... other GPU stuff, all asynchronous
    GrabMyFrame(pCPUbuf[bufNum]);
    ... other CPU stuff
    cudaThreadSynchronize();
    bufNum++; bufNum %= 3;
}
Host Memory Mapping, a.k.a “Zero-Copy”

The easy way to achieve copy/compute overlap!

1. **Enable Host Mapping**
   *Runtime:* `cudaSetDeviceFlags()` with `cudaDeviceMapHost` flag
   *Driver:* `cuCtxCreate()` with `CU_CTX_MAP_HOST`

2. **Allocate pinned CPU memory**
   *Runtime:* `cudaHostAlloc()`, use `cudaHostAllocMapped` flag
   *Driver:* `cuMemHostAlloc()` use `CUDA_MEMHOSTALLOC_DEVICEMAP`

3. **Get a CUDA device pointer to this memory**
   *Runtime:* `cudaHostGetDevicePointer()`
   *Driver:* `cuMemHostGetDevicePointer()`

4. **Just use that pointer in your kernels!**

*Check the `canMapHostMemory` / `CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY` device property flag to see if Zero-Copy is available.

Note: For Ion™ and other Unified Memory Architecture (UMA) GPUs zero-copy eliminates data transfer altogether!
Zero-Copy Guidelines

• Data is transferred over the PCIe bus automatically, but it’s slow

• Use when data is only read/written once

• Use for very small amounts of data (new variables, CPU/GPU communication)

• Use when compute/memory ratio is very high and occupancy is high, so latency over PCIe is hidden

• Coalescing is critically important!
NVIDIA NEXUS

The first development environment for massively parallel applications.

Hardware GPU Source Debugging
Platform-wide Analysis
Complete Visual Studio integration

Timeline trace is excellent for analyzing streams!

Register for the Beta here at GTC!

Beta available October 2009
Releasing in Q1 2010
Questions?