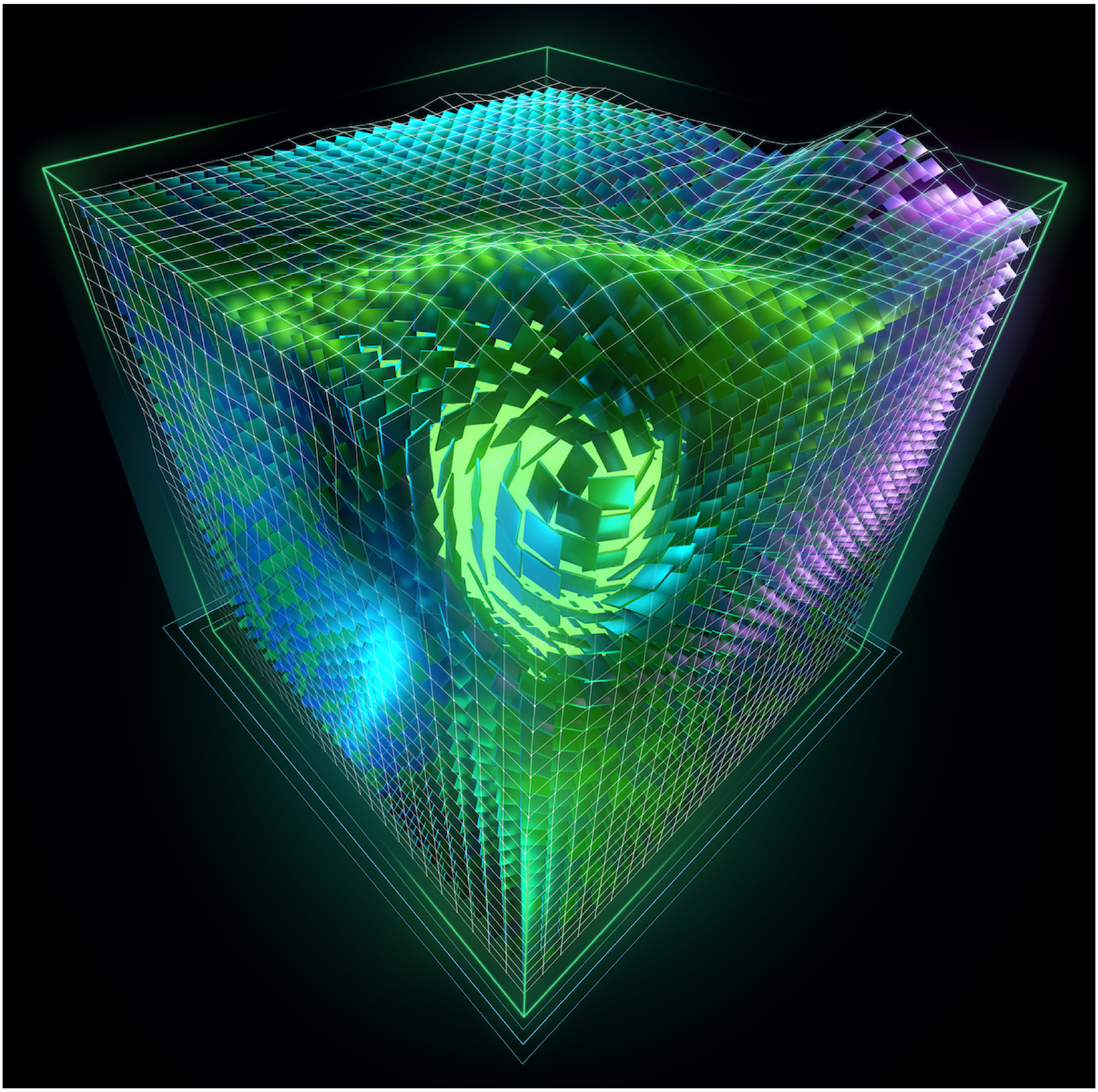
DEVELOPER BLOG



HPC

Dec 04, 2012

How to Optimize Data Transfers in CUDA C/C++

By Mark Harris

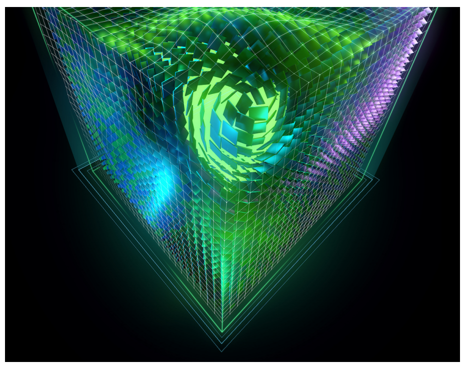
Tags: accelerated computing, Beginner, CUDA, CUDA C/C++, Memory, Profiling

Discuss (9)

In the previous three posts of this CUDA C & C++ series we laid the groundwork for the major thrust of the series: how to optimize CUDA C/C++ code. In this and the following post we begin our discussion of code optimization with how to efficiently transfer data between the host and device. The peak bandwidth between the device memory and the GPU is much higher (144 GB/s on the NVIDIA Tesla C2050, for example) than the peak bandwidth between host memory and device memory (8 GB/s on PCIe x16 Gen2). This disparity means that your implementation of data transfers

338
Shares

- Batching many small transfers into one larger transfer performs much better because it eliminates most of the per-transfer overhead.
- Data transfers between the host and device can sometimes be overlapped with kernel execution and other data transfers.



We investigate the first three guidelines above in this post, and we dedicate the next post to overlapping data transfers. First I want to talk about how to measure time spent in data transfers without modifying the source code.

Measuring Data Transfer Times with nvprof

To measure the time spent in each data transfer, we could record a CUDA event before and after each transfer and use `cudaEventElapsedTime()`, as we described in a previous post. However, we can get the elapsed transfer time without instrumenting the source code with CUDA events by using nvprof, a command-line CUDA profiler included with the CUDA Toolkit (starting with CUDA 5). Let's try it out with the following code example, which you can find in the Github repository for this post.

```
int main()
{
    const unsigned int N = 1048576;
    const unsigned int bytes = N * sizeof(int);
    int *h_a = (int*)malloc(bytes);
    int *d_a;
    cudaMalloc((int**)&d_a, bytes);

    memset(h_a, 0, bytes);
    cudaMemcpy(d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(h_a, d_a, bytes, cudaMemcpyDeviceToHost);

    return 0;
}
```

To profile this code, we just compile it using `nvcc`, and then run `nvprof` with the program filename as an argument.

```
$ nvcc profile.cu -o profile_test
$ nvprof ./profile_test
```

When I run on my desktop PC which has a GeForce GTX 680 (GK104 GPU, similar to a Tesla K10), I get the following output.

```
$ nvprof ./a.out
======== NVPROF is profiling a.out...
======== Command: a.out
======== Profiling result:
Time(%)     Time  Calls      Avg      Min      Max Name
  50.08 718.11us      1 718.11us 718.11us 718.11us [CUDA memcpy DtoH]
  49.92 715.94us      1 715.94us 715.94us 715.94us [CUDA memcpy HtoD]
```

As you can see, `nvprof` measures the time taken by each of the CUDA memcpy calls. It reports the average, minimum, and maximum time for each call (since we only run each copy once, all times are the same). `nvprof` is quite flexible, so make sure you check out the documentation.

`nvprof` is new in CUDA 5. If you are using an earlier version of CUDA, you can use the older "command-line profiler", as Greg Ruetsch explained in his post How to Optimize Data Transfers in CUDA Fortran.

Minimizing Data Transfers

We should not use only the GPU execution time of a kernel relative to the execution time of its CPU implementation to decide whether to run the GPU or CPU version. We also need to consider the cost of moving data across the PCI-e bus, especially when we are initially porting code to CUDA. Because CUDA's heterogeneous programming model uses both the CPU and GPU, code can be ported to CUDA one kernel at a time. In the initial stages of porting, data transfers may dominate the overall execution time. It's worthwhile to keep tabs on time spent on data transfers separately from time spent in kernel execution. It's easy to use the command-line profiler for this, as we already demonstrated. As we port more of our code, we'll remove intermediate transfers and decrease the overall execution time correspondingly.

Pinned Host Memory

Host (CPU) data allocations are pageable by default. The GPU cannot access data directly from pageable host memory, so when a data transfer from pageable host memory to device memory is invoked, the CUDA driver must first allocate a temporary page-locked, or "pinned", host array, copy the host data to the pinned array, and then transfer the data from the pinned array to device memory, as illustrated below.

As you can see in the figure, pinned memory is used as a staging area for transfers from the device to the host. We can avoid the cost of the transfer between pageable and pinned host arrays by directly allocating our host arrays in pinned memory. Allocate pinned host memory in CUDA C/C++ using cudaMallocHost() or cudaHostAlloc(), and deallocate it with cudaFreeHost(). It is possible for pinned memory allocation to fail, so you should always check for errors. The following code excerpt demonstrates allocation of pinned memory with error checking.

338
Shares

Data transfers using host pinned memory use the same cudaMemcpy() syntax as transfers with pageable memory. We can use the following "bandwidthtest" program (also available on Github) to compare pageable and pinned transfer rates.

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

// Convenience function for checking CUDA runtime API results
// can be wrapped around any runtime API call. No-op in release builds.
inline
cudaError_t checkCuda(cudaError_t result)
{
#if defined(DEBUG) || defined(_DEBUG)
  if (result != cudaSuccess) {
    fprintf(stderr, "CUDA Runtime Error: %s\n",
            cudaGetErrorString(result));
    assert(result == cudaSuccess);
  }
#endif
  return result;
}

void profileCopies(float        *h_a,
                   float        *h_b,
                   float        *d,
                   unsigned int  n,
                   char         *desc)
{
  printf("\n%s transfers\n", desc);

  unsigned int bytes = n * sizeof(float);
```

The data transfer rate can depend on the type of host system (motherboard, CPU, and chipset) as well as the GPU. On my laptop which has an Intel Core i7-2620M CPU (2.7GHz, 2 Sandy Bridge cores, 4MB L3 Cache) and an NVIDIA NVS 4200M GPU (1 Fermi SM, Compute Capability 2.1, PCI-e Gen2 x16), running `BandwidthTest` produces the following results. As you can see, pinned transfers are more than twice as fast as pageable transfers.

```
Device: NVS 4200M
Transfer size (MB): 16

Pageable transfers
  Host to Device bandwidth (GB/s): 2.308439
  Device to Host bandwidth (GB/s): 2.316220

Pinned transfers
  Host to Device bandwidth (GB/s): 5.774224
  Device to Host bandwidth (GB/s): 5.958834
```

On my desktop PC with a much faster Intel Core i7-3930K CPU (3.2 GHz, 6 Sandy Bridge cores, 12MB L3 Cache) and an NVIDIA GeForce GTX 680 GPU (8 Kepler SMs, Compute Capability 3.0) we see much faster pageable transfers, as the following output shows. This is presumably because the faster CPU (and chipset) reduces the host-side memory copy cost.

```
Device: GeForce GTX 680
Transfer size (MB): 16

Pageable transfers
  Host to Device bandwidth (GB/s): 5.368503
  Device to Host bandwidth (GB/s): 5.627219

Pinned transfers
  Host to Device bandwidth (GB/s): 6.186581
  Device to Host bandwidth (GB/s): 6.670246
```

You should not over-allocate pinned memory. Doing so can reduce overall system performance because it reduces the amount of physical memory available to the operating system and other programs. How much is too much is difficult to tell in advance, so as with all optimizations, test your applications and the systems they run on for optimal performance parameters.

Batching Small Transfers

Due to the overhead associated with each transfer, it is preferable to batch many small transfers together into a single transfer. This is easy to do by using a temporary array, preferably pinned, and packing it with the data to be transferred.

For two-dimensional array transfers, you can use cudaMemcpy2D().

```
cudaMemcpy2D(dest, dest_pitch, src, src_pitch, w, h, cudaMemcpyHostToDevice)
```

The arguments here are a pointer to the first destination element and the pitch of the destination array, a pointer to the first source element and pitch of the source array, the width and height of the submatrix to transfer, and the memcpy kind. There is also a cudaMemcpy3D() function for transfers of rank three array sections.

Summary

Transfers between the host and device are the slowest link of data movement involved in GPU computing, so you should take care to minimize transfers. Following the guidelines in this post can help you make sure necessary transfers are efficient. When you are porting or writing new CUDA C/C++ code, I recommend that you start with pageable transfers from existing host pointers. As I mentioned earlier, as you write more device code you will eliminate some of the intermediate transfers, so any effort you spend optimizing transfers early in porting may be wasted. Also, rather than instrument code with CUDA events or other timers to measure time spent for each transfer, I recommend that you use `nvprof,` the command-line CUDA profiler, or one of the visual profiling tools such as the NVIDIA Visual Profiler (also included with the CUDA Toolkit).

This post focused on making data transfers efficient. In the next post, we discuss how you can overlap data transfers with computation and with other data transfers.

About the Authors

About Mark Harris
Mark is an NVIDIA Distinguished Engineer working on RAPIDS. Mark has over twenty years of experience developing software for GPUs, ranging from graphics and games, to physically-based simulation, to parallel algorithms and high-performance computing. While a Ph.D. student at The University of North Carolina he recognized a nascent trend and coined a name for it: GPGPU (General-Purpose computing on Graphics Processing Units).
Follow harrism on Twitter
View all posts by Mark Harris »

338
Shares

DEVELOPER

anon18016087

March 25, 2015
Nice article. Do I need to be worried about kernel calls, which operate on a mix of host and device variables ? Would you recommend to use all device variables within a kernel.

anon95180265

March 25, 2015
Kernels are by definition device code. Therefore they must operate only on device memory, or on __managed__ memory (see my post on Unified Memory for more on this:
http://devblogs.nvidia.com/...

anon37663623

July 17, 2015
Hi, Mark
Using Pinned memory I achieved 11GB/s H2D transfer on Titan X (PCI-E 3.0 16x).
Do you think this could be improved?

anon91878944

May 16, 2016
Hi Mark,
Thank you for a very interesting and helpful article. I have a question: I transfer big data to the GPU once, and than create a cut of it using the GPU, transfer the result back to cpu, and than again, new cut, new transfer. Like scrolling of a plane-cut on 3d data. The most time consuming operation is the transfer back of the result (about 5 Mb) which is about 300 msec. Is it reasonable? is there a way to improve it or this not a task that is suitable for GPU?
Thanks
Hadar

anon24328757

June 5, 2017
wonderful post! Very helpful.

Continue the discussion at forums.developer.nvidia.com

4 more replies

Participants

338
Shares