# How to Access Global Memory Efficiently in CUDA C/C++ Kernels

In the previous [two](https://developer.nvidia.com/blog/parallelforall/how-optimize-data-transfers-cuda-cc/) [posts](https://developer.nvidia.com/blog/parallelforall/how-overlap-data-transfers-cuda-cc/) we looked at how to move data efficiently between the host and device. In this sixth post of our CUDA C/C++ series we discuss how to efficiently access device memory, in particular *global memory*, from within kernels.

There are several kinds of memory on a CUDA device, each with different scope, lifetime, and caching behavior. So far in this series we have used *global memory*, which resides in device DRAM, for transfers between the host and device as well as for the data input to and output from kernels. The name *global* here refers to scope, as it can be accessed and modified from both the host and the device. Global memory can be declared in global (variable) scope using the \_\_device\_\_ declaration specifier as in the first line of the following code snippet, or dynamically allocated using [cudaMalloc()](http://docs.nvidia.com/cuda/cuda-runtime-api/index.html#group__CUDART__MEMORY_1g16a37ee003fcd9374ac8e6a5d4dee29e) and assigned to a regular C pointer variable as in line 7. Global memory allocations can persist for the lifetime of the application. Depending on the [compute capability](https://developer.nvidia.com/blog/parallelforall/how-query-device-properties-and-handle-errors-cuda-cc/) of the device, global memory may or may not be cached on the chip.

1. \_\_device\_\_ int globalArray[256];
3. void foo()
4. {
5. ...
6. int \*myDeviceMemory = 0;
7. cudaError\_t result = cudaMalloc(&myDeviceMemory, 256 \* sizeof(int));
8. ...
9. }

Before we go into global memory access performance, we need to refine our understanding of the CUDA execution model. We have discussed how [threads are grouped into thread blocks](https://developer.nvidia.com/blog/parallelforall/easy-introduction-cuda-c-and-c/), which are assigned to multiprocessors on the device. During execution there is a finer grouping of threads into *warps*. Multiprocessors on the GPU execute instructions for each warp in SIMD ([Single Instruction Multiple Data](http://en.wikipedia.org/wiki/SIMD)) fashion. The warp size (effectively the SIMD width) of all current CUDA-capable GPUs is 32 threads.

Global Memory Coalescing

Grouping of threads into warps is not only relevant to computation, but also to global memory accesses. The device *coalesces*global memory loads and stores issued by threads of a warp into as few transactions as possible to minimize DRAM bandwidth (on older hardware of compute capability less than 2.0, transactions are coalesced within half warps of 16 threads rather than whole warps). To make clear the conditions under which coalescing occurs across CUDA device architectures we run some simple experiments on three Tesla cards: a Tesla C870 (compute capability 1.0), a Tesla C1060 (compute capability 1.3), and a Tesla C2050 (compute capability 2.0).

We run two experiments that use variants of an increment kernel shown in the following code ([also available on GitHub](https://github.com/parallel-forall/code-samples/blob/master/series/cuda-cpp/coalescing-global/coalescing.cu)), one with an array offset that can cause misaligned accesses to the input array, and the other with strided accesses to the input array.

#include

#include

// 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: %sn", cudaGetErrorString(result));

assert(result == cudaSuccess);

}

#endif

return result;

}

template

\_\_global\_\_ void offset(T\* a, int s)

{

int i = blockDim.x \* blockIdx.x + threadIdx.x + s;

a[i] = a[i] + 1;

}

template

\_\_global\_\_ void stride(T\* a, int s)

{

int i = (blockDim.x \* blockIdx.x + threadIdx.x) \* s;

a[i] = a[i] + 1;

}

template

void runTest(int deviceId, int nMB)

{

int blockSize = 256;

float ms;

T \*d\_a;

cudaEvent\_t startEvent, stopEvent;

int n = nMB\*1024\*1024/sizeof(T);

// NB: d\_a(33\*nMB) for stride case

checkCuda( cudaMalloc(&d\_a, n \* 33 \* sizeof(T)) );

checkCuda( cudaEventCreate(&startEvent) );

checkCuda( cudaEventCreate(&stopEvent) );

printf("Offset, Bandwidth (GB/s):n");

offset<<>>(d\_a, 0); // warm up

for (int i = 0; i <= 32; i++) {

checkCuda( cudaMemset(d\_a, 0.0, n \* sizeof(T)) );

checkCuda( cudaEventRecord(startEvent,0) );

offset<<>>(d\_a, i);

checkCuda( cudaEventRecord(stopEvent,0) );

checkCuda( cudaEventSynchronize(stopEvent) );

checkCuda( cudaEventElapsedTime(&ms, startEvent, stopEvent) );

printf("%d, %fn", i, 2\*nMB/ms);

}

printf("n");

printf("Stride, Bandwidth (GB/s):n");

stride<<>>(d\_a, 1); // warm up

for (int i = 1; i <= 32; i++) {

checkCuda( cudaMemset(d\_a, 0.0, n \* sizeof(T)) );

checkCuda( cudaEventRecord(startEvent,0) );

stride<<>>(d\_a, i);

checkCuda( cudaEventRecord(stopEvent,0) );

checkCuda( cudaEventSynchronize(stopEvent) );

checkCuda( cudaEventElapsedTime(&ms, startEvent, stopEvent) );

printf("%d, %fn", i, 2\*nMB/ms);

}

checkCuda( cudaEventDestroy(startEvent) );

checkCuda( cudaEventDestroy(stopEvent) );

cudaFree(d\_a);

}

int main(int argc, char \*\*argv)

{

int nMB = 4;

int deviceId = 0;

bool bFp64 = false;

for (int i = 1; i < argc; i++) {

if (!strncmp(argv[i], "dev=", 4))

deviceId = atoi((char\*)(&argv[i][4]));

else if (!strcmp(argv[i], "fp64"))

bFp64 = true;

}

cudaDeviceProp prop;

checkCuda( cudaSetDevice(deviceId) )

;

checkCuda( cudaGetDeviceProperties(&prop, deviceId) );

printf("Device: %sn", prop.name);

printf("Transfer size (MB): %dn", nMB);

printf("%s Precisionn", bFp64 ? "Double" : "Single");

if (bFp64) runTest(deviceId, nMB);

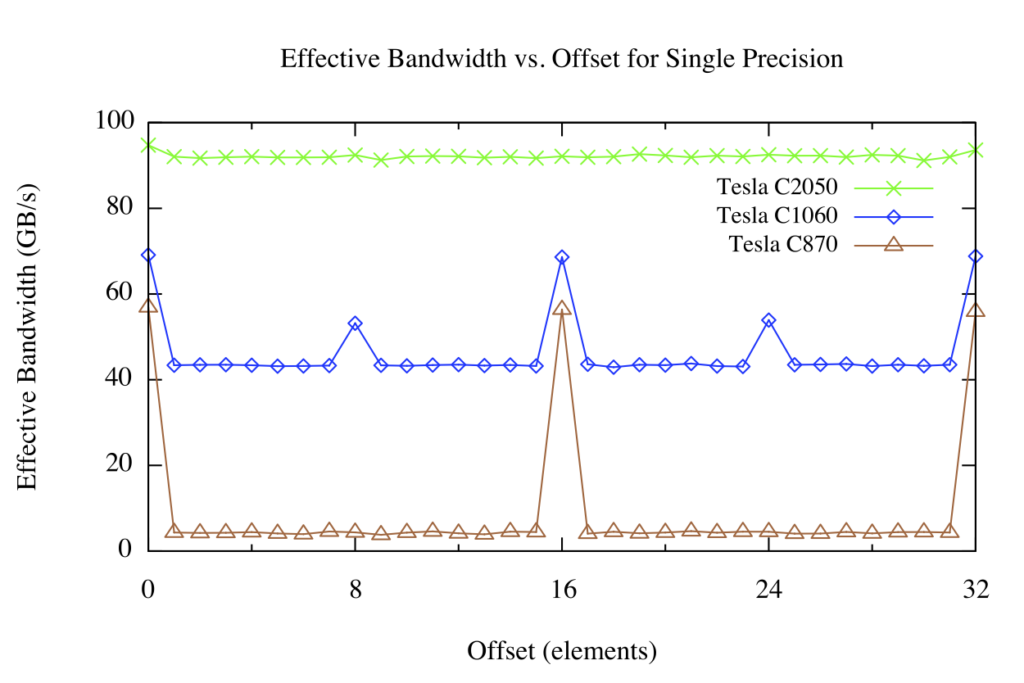
else runTest(deviceId, nMB);

}

This code can run both offset and stride kernels in either single (default) or double precision by passing the “fp64” command line option. Each kernel takes two arguments, an input array and an integer representing the offset or stride used to access the elements of the array. The kernels are called in loops over a range of offsets and strides.

Misaligned Data Accesses

The results for the offset kernel on the Tesla C870, C1060, and C2050 appear in the following figure.

[](https://developer.nvidia.com/blog/parallelforall/wp-content/uploads/2012/10/offset.png)

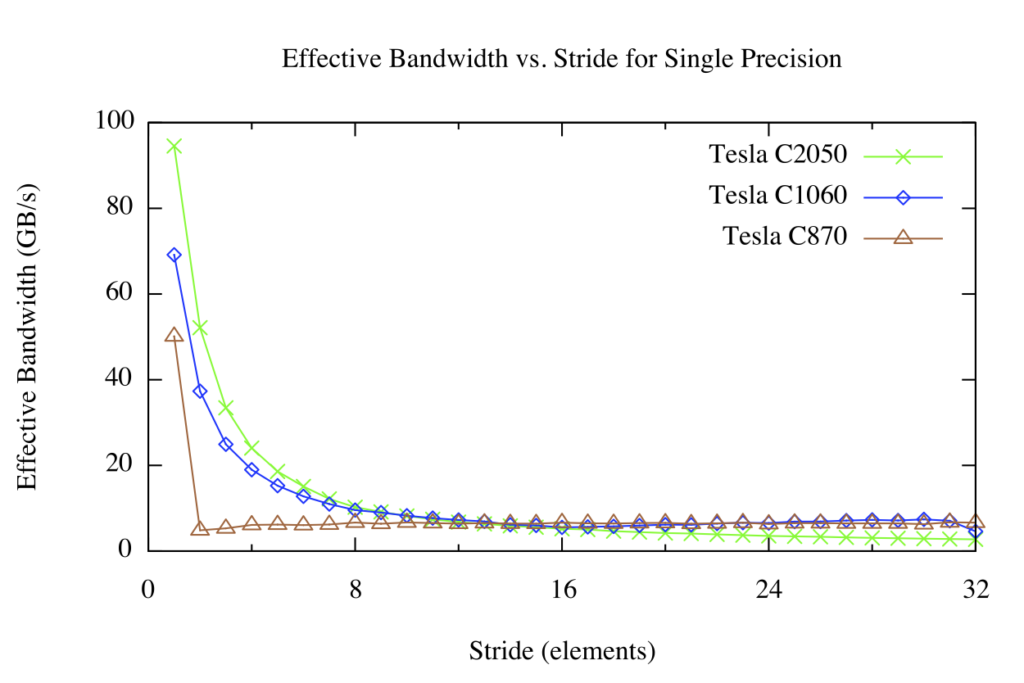
Arrays allocated in device memory are aligned to 256-byte memory segments by the CUDA driver. The device can access global memory via 32-, 64-, or 128-byte transactions that are aligned to their size. For the C870 or any other device with a compute capability of 1.0, any misaligned access by a half warp of threads (or aligned access where the threads of the half warp do not access memory in sequence) results in 16 separate 32-byte transactions. Since only 4 bytes are requested per 32-byte transaction, one would expect the effective bandwidth to be reduced by a factor of eight, which is roughly what we see in the figure above (brown line) for offsets that are not a multiple of 16 elements, corresponding to one half warp of threads.

For the Tesla C1060 or other devices with compute capability of 1.2 or 1.3, misaligned accesses are less problematic. Basically, the misaligned accesses of contiguous data by a half warp of threads are serviced in a few transactions that “cover” the requested data. There is still a performance penalty relative to the aligned case due both to unrequested data being transferred and to some overlap of data requested by different half-warps, but the penalty is far less than for the C870.

Devices of compute capability 2.0, such as the Tesla C2050, have an L1 cache in each multiprocessor with a 128-byte line size. The device coalesces accesses by threads in a warp into as few cache lines as possible, resulting in negligible effect of alignment on throughput for sequential memory accesses across threads.

Strided Memory Access

The results of the stride kernel appear in the following figure.

[](https://developer.nvidia.com/blog/parallelforall/wp-content/uploads/2012/10/stride.png)

For strided global memory access we have a different picture. For large strides, the effective bandwidth is poor regardless of architecture version. This should not be surprising: when concurrent threads simultaneously access memory addresses that are very far apart in physical memory, then there is no chance for the hardware to combine the accesses. You can see in the figure above that on the Tesla C870 any stride other than 1 results in drastically reduced effective bandwidth. This is because compute capability 1.0 and 1.1 hardware requires linear, aligned accesses across threads for coalescing, so we see the familiar 1/8 bandwidth that we also saw in the offset kernel. Compute capability 1.2 and higher hardware can coalesce accesses that fall into aligned segments (32, 64, or 128 byte segments on CC 1.2/1.3, and 128-byte cache lines on CC 2.0 and higher), so this hardware results in a smooth bandwidth curve.

When accessing multidimensional arrays it is often necessary for threads to index the higher dimensions of the array, so strided access is simply unavoidable. We can handle these cases by using a type of CUDA memory called *shared memory*. Shared memory is an on-chip memory shared by all threads in a thread block. One use of shared memory is to extract a 2D tile of a multidimensional array from global memory in a coalesced fashion into shared memory, and then have contiguous threads stride through the shared memory tile. Unlike global memory, there is no penalty for strided access of shared memory. We will cover shared memory in detail in the next post.

 Summary

In this post we discussed some aspects of how to efficiently access global memory from within CUDA kernel code. Global memory access on the device shares performance characteristics with data access on the host; namely, that data locality is very important. In early CUDA hardware, memory access alignment was as important as locality across threads, but on recent hardware alignment is not much of a concern. On the other hand, strided memory access can hurt performance, which can be alleviated using on-chip shared memory. In the [next post](https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/) we will explore shared memory in detail, and in the post after that we will show how to use shared memory to avoid strided global memory accesses during a [matrix transpose](https://developer.nvidia.com/blog/parallelforall/efficient-matrix-transpose-cuda-cc/).