AMS 148 Chapter 8: Optimization in CUDA, and Advanced Topics

Steven Reeves

1 Optimizing Data Transfers in CUDA C/C++

This section we will discuss code optimization with how to efficiently transfer data between the host and the device. The peak bandwidth between the device memory and the GPU is much higher (720 GB/s for the Tesla P100 found in Hummingbird) than the peak bandwidth between host memory and device memory (8 GB/s on a PCIex16 Generation 2). This disparity means that if your implementation requires many data transfers from GPU to host or vice-versa it will greatly halter your performance. Let us begin with a few general guidlines for host-device data transfers. We wish to:

- Minimize the ammount of data transferred between host and device when possible, even if that means running kernels on the GPU that get little or no speed-up compared to running them on the host CPU.
- Hihger bandwidth is possible between the host and the device when using page-locked (or 'pinned') memory.
- Batching many small transfers into one larger transfer performs better as 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.

First let us talk about how to measure time spent in data transfers without modifying source code.

1.1 Measuring Data Transfers Times with nvprof

To measure the time spent during each data transfer, we could record a CUDA event before and after each transfer and use `cudaEventElaspsedTime()` as we have in the past. However, we can retrieve the elapsed transfer time without deploying CUDA events by using nvprof, a command-line CUDA profiler included with the CUDA Toolkit. Let's try the following code example:

Listing 1: Example Code for Profiling

```
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 need to compile it using `nvcc`, and the run `nvprof` with the program fileneame as an argument.

Listing 2: Running nvprof

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

Using the Citrisdance (NVIDIA K20) server I recieve the following output:

```
==17821== NVPROF is profiling process 17821, command: ./profile.exe
==17821== Profiling application: ./profile.exe
==17821== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   51.35%  1.5589ms         1  1.5589ms  1.5589ms  1.5589ms  [CUDA memcpy DtoH]
                   48.65%  1.4772ms         1  1.4772ms  1.4772ms  1.4772ms  [CUDA memcpy HtoD]
      API calls:   98.80%  480.32ms         1  480.32ms  480.32ms  480.32ms  cudaMalloc
                    0.93%  4.5109ms         2  2.2554ms  1.7877ms  2.7232ms  cudaMemcpy
                    0.21%  1.0257ms       188  5.4550us     220ns  205.44us  cuDeviceGetAttribute
                    0.04%  197.15us         2  98.574us  66.692us  130.46us  cuDeviceTotalMem
                    0.02%  93.452us         2  46.726us  44.398us  49.054us  cuDeviceGetName
                    0.00%  6.8620us         4  1.7150us     342ns  5.2240us  cuDeviceGet
                    0.00%  3.6260us         3  1.2080us     347ns  2.4570us  cuDeviceGetCount
```

We see that `nvprof` gives a full breakdown of the program, including the CUDA API calls and the GPU activities. We see that the majority of the time is used on the memory allocation, but barring this API call, the next most time consuming are the memory transfers. Memory transfers are much more common than allocation in most applications.

1.2 Minimizing Data Transfers

We should not use only the GPU exectuation time of a kernel relative to CPU functions to decided 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 of the hetrogenous programming model of CUDA (using both the CPU and GPU), code can be ported to the GPU one kernel at a time. In the initial stages of writing CUDA code, data transfers may donimate the overall execution time. It is worth while to monitor time spent on data transfer separately from time spent during computation within a kernel. It is easy to use the command-line profiler for this, as demonstrated above. As more of our code is ported to CUDA, we will remove intermediate transfers and descrease the overall execution time correspondingly.

1.3 Pinned Memory

Pageable memory space means memory contents that can be paged in/ paged out between DRAM and a secondary storage device. Host memory allocations are generally pageable.

The main motivation for using pinned memory is to perform asynchronous transfers of data from the host to the device. This is accomplished by using the CUDA primitive `cudaMemcpyAsync` and related functions. Additionally, certain performance benefits come from using pinned (or *page-locked* memory). In this section we will give a few examples about how to allocate pinned memory and we investigate features of this type of memory.

1.3.1 About pinned memory

The workings of paged memory is best described by the post on http://devblogs.nvidia.com which will be paraphrased here.

Data allocations on the CPU 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 is invoked, the CUDA driver will first allocate a temporary page-locked, or "pinned" host array. Then the data is copied into this pinned array for transfer to the device. An illustration of this process is provided by Nvidia blogpost:

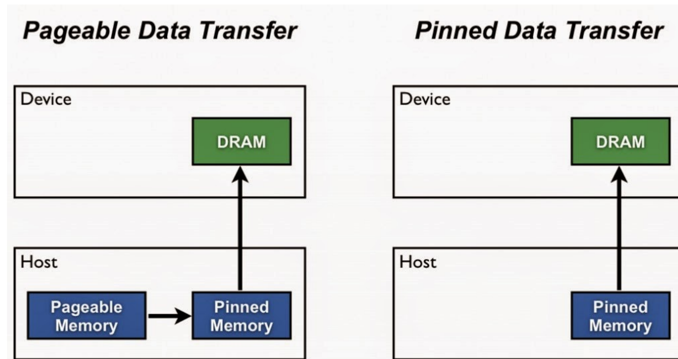


Figure 1: Regular Data Transfer vs Pinned Data Transfer; NVIDIA Developer Blog

As shown in figure 1, pinned memory is used as a staging area for transfers from the device to the hsot. We can avoid the cost of the transfer betweeen pageable and pinned host arrays by directly allocation the host arrays in pinned memory. Allocate pinned host memory in CUDA C/C++ using `cudaMallocHost()` or `cudaHostAlloc()` and free the memory with `cudaFreeHost()`. It is possible for pinned memory allocation to fail, so we should check for errors using the `cudaError_t` class. The following code demonstrates allocation of pinned memory with error chekcing.

```
cudaError_t status = cudaMallocHost((void**)&h_aPinned, bytes);
if (status !=cudaSuccess)
        printf("Error allocating pinned host memory\n");
```

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

Listing 3: Bandwidth Test

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

// Convenience function for hcekcing 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 deifned(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);

  // events for timing
  cudaEvent_t startEvent, stopEvent;

  checkCuda( cudaEventCreate(&startEvent) );
  checkCuda( cudaEventCreate(&stopEvent) );

  checkCuda( cudaEventRecord(startEvent, 0) );
  checkCuda( cudaMemcpy(d, h_a, bytes, cudaMemcpyHostToDevice) );
  checkCuda( cudaEventRecord(stopEvent, 0) );
  checkCuda( cudaEventSynchronize(stopEvent) );
```

```
  float time;
  checkCuda( cudaEventElapsedTime(&time, startEvent, stopEvent) );
  printf("  Host to Device bandwidth (GB/s): %f\n", bytes * 1e-6 / time);

  checkCuda( cudaEventRecord(startEvent, 0) );
  checkCuda( cudaMemcpy(h_b, d, bytes, cudaMemcpyDeviceToHost) );
  checkCuda( cudaEventRecord(stopEvent, 0) );
  checkCuda( cudaEventSynchronize(stopEvent) );

  checkCuda( cudaEventElapsedTime(&time, startEvent, stopEvent) );
  printf("  Device to Host bandwidth (GB/s): %f\n", bytes * 1e-6 / time);

  for (int i = 0; i < n; ++i) {
    if (h_a[i] != h_b[i]) {
      printf("*** %s transfers failed ***\n", desc);
      break;
    }
  }

  // clean up events
  checkCuda( cudaEventDestroy(startEvent) );
  checkCuda( cudaEventDestroy(stopEvent) );
}

int main()
{
  unsigned int nElements = 4*1024*1024;
  const unsigned int bytes = nElements * sizeof(float);

  // host arrays
  float *h_aPageable, *h_bPageable;
  float *h_aPinned, *h_bPinned;

  // device array
  float *d_a;

  // allocate and initialize
  h_aPageable = (float*)malloc(bytes);                    // host pageable
  h_bPageable = (float*)malloc(bytes);                    // host pageable
  checkCuda( cudaMallocHost((void**)&h_aPinned, bytes) ); // host pinned
  checkCuda( cudaMallocHost((void**)&h_bPinned, bytes) ); // host pinned
  checkCuda( cudaMalloc((void**)&d_a, bytes) );           // device

  for (int i = 0; i < nElements; ++i) h_aPageable[i] = i;
  memcpy(h_aPinned, h_aPageable, bytes);
  memset(h_bPageable, 0, bytes);
  memset(h_bPinned, 0, bytes);

  // output device info and transfer size
  cudaDeviceProp prop;
  checkCuda( cudaGetDeviceProperties(&prop, 0) );

  printf("\nDevice: %s\n", prop.name);
  printf("Transfer size (MB): %d\n", bytes / (1024 * 1024));

  // perform copies and report bandwidth
  profileCopies(h_aPageable, h_bPageable, d_a, nElements, "Pageable");
  profileCopies(h_aPinned, h_bPinned, d_a, nElements, "Pinned");

  printf("n");

  // cleanup
  cudaFree(d_a);
  cudaFreeHost(h_aPinned);
  cudaFreeHost(h_bPinned);
  free(h_aPageable);
  free(h_bPageable);

  return 0;
}
```

The data transfer rate can depend on the type of host stysem (motherboard, CPU, and chipset) as well as the GPU. Running this program on HummingBird we have the following output

Listing 4: Output of Bandwidth Test

```
Device: Tesla P100-PCIE-16GB
Transfer size (MB): 16
```

```
Pageable transfers
  Host to Device bandwidth (GB/s): 2.990821
  Device to Host bandwidth (GB/s): 4.364375

Pinned transfers
  Host to Device bandwidth (GB/s): 12.017788
  Device to Host bandwidth (GB/s): 12.726673
```

On Hummingbird the CPU is an AMD Opteron processor and sets a decent pageable transfer rate. However, we find that the pinned transfer rate is much more impressive, offering more than 4 times bandwidth Host to Device and 3 times bandwidth Device to Host. Note that the pageable transfer rate depends on the speed of the CPU. Faster CPUs will offer a better pageable bandwidths, however with modern GPUs pinned bandwidths will exceed this capability.

A word of warning, 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 a priory, so as with most optimizaitons, test your code and the systems they run on for optimal performance parameters.

1.3.2 Batching Small Transfers

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

2 CUDA Streams

In the previous section we discussed how to transfer data efficiently between the host and device. In this section, we discuss how to overlap data transfers with computation on the host, computation on the device, and other data transfers between the hsot and device. Through the use of CUDA streams, overlap between data transfers and other operations can be achieved.

A *stream* in CUDA is a sequence of operations that execute on the device in the order in which they are issued by the host code. While operations within a stream are garaunteed to execute in the prescribed order, operations in different streams can be interleaved and, when possible, they will run concurrently.

2.1 The default stream

All device operations (kernels and data transfers) in CUDA run in a stream. When no stream is specified, the default stream (also referred to the "null stream") is used. The default stream is different from other streams as it is a synchronizing stream with respect to the device: no operation on the default stream will begin until all previous issued operation *in any stream* on the device have completed, and an operation in the default stream must complete before any other operation (in any stream on the device) will begin.

Let us check out a simple code snippet that use the default stream:

Listing 5: Code Snippet Default Stream

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a)
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

In this snippet, from the persepctive of the PGU¡ all three operation are issued to the same (default) stream and will execute in the order that they were issued.

From the perspective of the CPU, the implicit data transfers are blocking or synchronous transfers, while kernel launching is asynchronous. Since the host-to-device transfer on the first line is synchronous, the CPU thread will not reach the kernel call on the second line until the host to device transfer is complete. Once the kernel is issued, the CPU thread moves to the third line, but the transfer on that line will not begin due to the device-side order of execution.

The asynchronous behavior of the kernel launches from the CPU's perspective makes overlapping device and host computations very easy, take the following snippet for example:

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a)
myCpuFunction(b)
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

In this code, as soon as the kernel is launched on the device the CPU thread executes `myCpuFunction()`, overlapping the CPU function with the kernel execution on the GPU. Whether the host function or device kernel completes first doesn't afftect the subsequent device to host memory transfer, which begins only after the kernel is completed. From the perspective of the GPU, nothign has changed from the previous example; the device is completely unaware of `myCpuFunction()`.

2.2 Non-default streams

Streams other than the null CUDA C/C++ are declared, created, and destroyed in the host code, as in this example:

Listing 6: Streams

```
cudaStream_t stream1;
cudaError_t result;
result = cudaStreamCreate(&stream1);
result = cudaStreamDestroy(&stream1);
```

In order to issue a data transfer to a non default CUDA stream, we use the `cudaMemcpyAsync()` function, which is similar to the regular `cudaMemcpy()` function we have been using before, but it takes another argument.

```
result = cudaMemcpyAsyn(d_a, a, N, cudaMemcpyHostToDevice, stream1);
```

This function is asynchronous with the host, so control returns to the host thread immediately after the transfer is issued. There are 2D and 3D extensions of this as well. To issue a kernel to a non-default stream, we specify the stream in question as a fourth argument in the kernel configuration:

```
increment<<<1,N,0,stream1>>>(d_a);
```

2.3 Synchronization with streams

Since all operations in non-default streams are asynchronous with respect to the host code, you will find that there may be situations that will require you to synchronize all streams. There are several ways to do this. The brute force way to do this is to utilize `cudaDeviceSynchronize()` which creates a barrier on the host thread until all previously issued GPU operations are complete. In most cases this is too much, and can hurt performance due to stalling the device and host thread.

The CUDA stream API has multiple less severe methods of synchronization. The function `cudaStreamSynchronize(stream)` can be used to block the host from prorceeding until all previous issued operations on the specified stream have completed. The function `cudaStreamQuery(stream)` tests whether all operations issued to the specified stream have completed, without blocking host execution. The fuctions `cudaEventSyncrhonize(event)` and `cudaEventQuery(event)` do the similar only that their result is based off of whether a particular event has been recorded. You can also synchronize operations within a single stream on a specific event using `cudaStreamWaitEvent(event)`.

2.4 Overlapping Kernel Execution and Data Transfer

Previously we demonstrated overlapping kernel execution in the default stream with execution of operations on the CPU. But our goal in this section is to show how to overlap kernel execution with data transfers. There are several requirements for this to happen:

- The device must be capable of "concurrent copy and exectuion"
- The kernel execution and the data transfer to be overlapped must both occur in different, non default streams
- The host memory involved in the data transfer must be **pinned** memory.

So we're going to modify the simple code from the previous section, the full code available on github. In this modified snippet, we break up the array of size N into chunks of `streamSize` elements. SInce the kernel operates independently on all elements, each of the chunks can be processed indepednently. The number of (non-default) streams used is `nStreams=N/streamSize`. There are multiple ways to implement the domain decompostion of the data and processing, one way is to loop over the operations for each chunk.

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, cudaMemcpyHostToDevice, stream[i]);
  kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
  cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, cudaMemcpyDeviceToHost, stream[i]);
}
```

Another approach is to batch similar operations together, issuing all the host to device transfers first, followed by the kernel lauches, lastly the device to host transfers.

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_a[offset], &a[offset],
                  streamBytes, cudaMemcpyHostToDevice, cudaMemcpyHostToDevice, stream[i]);
}

for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
}

for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&a[offset], &d_a[offset],
                  streamBytes, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToHost, stream[i]);
}
```

Both asyncrhonous methods shown above yield the correct results, and in both cases dependent operations are issued to the same stream in the order in which they need to be executed. The results can vary depending on GPU architectures: On citrisdance a Kepler based server I get the following results:

```
Device : Tesla K20c
Time for sequential transfer and execute (ms): 7.590112
  max error: 1.192093e-07
Time for asynchronous V1 transfer and execute (ms): 3.995456
  max error: 1.192093e-07
Time for asynchronous V2 transfer and execute (ms): 3.975712
  max error: 1.192093e-07
```

However, on Hummingbird the performance is faster:

```
Device : Tesla P100-PCIE-16GB
Time for sequential transfer and execute (ms): 3.154144
  max error: 1.192093e-07
Time for asynchronous V1 transfer and execute (ms): 1.971200
  max error: 1.192093e-07
Time for asynchronous V2 transfer and execute (ms): 1.959584
  max error: 1.192093e-07
```

This is mostly due to hardware advances in the years, but also due to changes in compute combatibility. For devices that have compute cability 3.5 or higher, the Hyper-Q feature eliminates the need to tailor the launch order so either approach yields similar results. However, on older devices, such as the Tesla C1060, a compute 1.5 device the results are different.

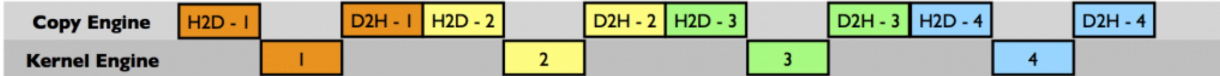
```
Device : Tesla C1060

Time for sequential transfer and execute (ms ): 12.92381
  max error : 2.3841858E -07
Time for asynchronous V1 transfer and execute (ms ): 13.63690
  max error : 2.3841858E -07
Time for asynchronous V2 transfer and execute (ms ): 8.84588
  max error : 2.3841858E -07
```

We see that with this device the version one transfer runs slower than the serially transfered method. A good way to understand this is that the C1060 only has one copy engine and one kernel engine. The following diagram illustrates the method of operation on the C1060

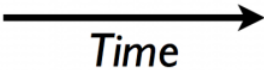
C1060 Execution Time Lines

Sequential Version

Asynchronous Version 1



Asynchronous Version 2



Time

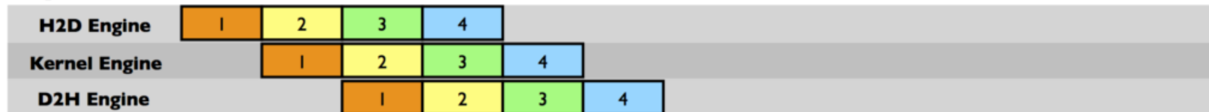
Figure 2: Singe Engine, Nvidia Developer Blogs

whereas a newer device, say a C2050, which contains two copy engines and kernel engines, the timelines are more like this:

C2050 Execution Time Lines

Sequential Version

Asynchronous Version 1



Asynchronous Version 2

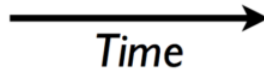


Figure 3: Multiple Engines, Nvidia Developer Blogs

The number of engines in these previous models dictated the nature of asynchronous operations. The Hyper-Q firmware allows for more effective use of the grid management. The following illustrations show the profiling of an applicaiton without and with hyper-Q.

Figure 4: Profiling of older multi-engine GPUs without Hyper-Q

Figure 5: Multi-engine GPUs with Hyper-Q

GPUs with Hyper-Q allow the hardward to compact asynchronous launches using either method. Allowing the developer to not worry (as much) about the hardware implementation.

3 Optimizing Calculations

In the previous sections we looked into to optimizing memory transactions, in this section we'll look into optimizing calculations within kernels. As an example we will follow Mark Harris' *Optimizing Parallel Reduction in CUDA* presentation.

Using this, we will discuss 7 different versions of the reduction kernel. Using these versions we will look at several important optimization strategies. Utilizing these strategies we strive to reach GPU peak performance. We will need to choose the right metric, FLOP/s for compute-bound kernels, and Bandwidth for memory-bound kernels. Reductions have very low arithmetic intensity, 1 flop per element loaded, making this bandwidth optimal. The following code is tested on aa Nvidia Titan XP GPU, which has a theoretical peak bandwidth of 547.6 GB/s. In what follows we will try to design an algorithm that gets close to this theoretical bandwidth. A true measurement is the effiective bandwdith

$$BW_{effective} = (R_B + W_B)/(t \times 10^9)$$

where R_B is the number of bytes read by the kernel, and W_B is the number of bytes written. We multiply t by 10^9 to retrieve GB/s.

The reduction that we have had before utilizes a method called *address interleaving*.

Listing 7: First Reduction

```
template <class T>
__global__ void reduce0(T *g_idata, T *g_odata){
        extern __shared__ T sdata[];
        // each thread loads one element from global to shared mem
        unsigned int tid = threadIdx.x;
        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
        sdata[tid] = g_idata[i];
        __syncthreads();
        // do reduction in shared mem
        for(unsigned int s=1; s < blockDim.x; s *= 2) {
                if (tid % (2*s) == 0)
                {
                        sdata[tid] += sdata[tid + s];
                }
                __syncthreads();
        }
        // write result for this block to global mem
        if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

The if statement within this code is highly divergent, it branches the threads and many launched threads are not active, and can result in very poor performance. We will test this code and the code that follows using 1 million integers.

Upon execution of this first reduce, the elapsed time is 0.132096 ms, with a effective bandwidth of 31.814 GB/s. This is far from the the theoretical bandwidth. Let us change this kernel to remove the divergent branching.

Listing 8: Reduction without divergent branching

```
template <class T>
__global__ void reduce1(T *d_out, const T *d_in)
{
        // sdata is allocated in the kernel call: via dynamic shared memeory
        extern __shared__ T sdata[];

        int myId = threadIdx.x + blockDim.x*blockIdx.x;
        int tid = threadIdx.x;

        //load shared mem from global mem
        sdata[tid] = d_in[myId];
        __syncthreads(); // always sync before using sdata

        //do reduction over shared memory
        for(int s = 1; s<blockDim.x; s *=2)
        {
                int index = 2*s*tid; //Strided indexing!
                if(index< blockDim.x)
                {
                        sdata[index] += sdata[index + s];
                }
                __syncthreads(); //make sure all additions are finished
        }

        //only tid 0 writes out result!
        if(tid == 0)
        {
                d_out[blockIdx.x] = sdata[0];
        }
}
```

All we changed in this code was the divergent inner loop. In this case we moved to a strided indexing scheme to create non-divergent branches. Upon execution of this kernel, we find that the elapsed time is now 0.071264 ms with an effective bandwidth of 58.9709GB/s. This shows that we have a 1.85 times speed up over the original. There's an additional problem here, we have shared memory bank conflicts, induced by the strided index.

We will change our looping scheme to yield sequential addressing, which will alleviate the shared memory bank conflict. To do this we swap the inner loop to be the following

Listing 9: Reduce with sequential addressing

```
        for(unsigned int s = blockDim.x/2; s>0; s >>=1)
        {
                if(tid < s)
                {
                        sdata[tid] += sdata[tid+s];
                }
                __syncthreads(); //make sure all additions are finished
        }
```

all else the same. By executing this kernel, we have the following reports: Elapsed time : 0.062816ms, Effective Bandwidth : 66.9017 GB/s, yielding a 1.13$\times$ speed up from the previous kernel.

The problem with this implementation is that there are **many** idle threads, we only use half of the threads in the thread block upon the first loop of the iteration. In this case, we will half the number of blocks used, and change the way we load into shared memeory. For this purpose we will do two loads, and do the first addition in the reduction kernel upon loading into shared memory. So, we change our kernel to use:

Listing 10: First Add During Load

```
        int myId = threadIdx.x + (blockDim.x*2)*blockIdx.x;
        int tid = threadIdx.x;

        //load shared mem from global mem
        sdata[tid] = d_in[myId] + d_in[myId + blockDim.x];
        __syncthreads(); // always sync before using sdata
```

Upon execution, we reduce the elapsed time to 0.034528ms and increase the bandwidth to 121.713GB/s. Giving a speed up of 1.8192 times.

At 121GB/s we're far from the bandwidth upper bound. Therefore there is likely a bottleneck in the instruction overhead. Ancillary instructions that are not loads, stores, or arithmetic for core computation can become bottlenecks. In this case, address arithmetic and loop overhead. Our strategy for mitigating this will be to unroll loops.

In the next kernel we will unroll the last warp. That is, we will change our loop to end at $s = 32$. Note that this will save useless work in *every* warp, since we will be taking stages out of the for loop. Without the unroll, every warp executes all iterations of the for loop and if statement.

Listing 11: Unrolling the last warp

```
        for(unsigned int s = blockDim.x/2; s>32; s >>=1)
        {
                if(tid < s)
                {
                        sdata[tid] += sdata[tid+s];
                }
                __syncthreads(); //make sure all additions are finished
        }
        if(tid < 32)
        {

           sdata[tid] += sdata[tid+32];
           sdata[tid] += sdata[tid+16];
           sdata[tid] += sdata[tid+8];
           sdata[tid] += sdata[tid+4];
           sdata[tid] += sdata[tid+2];
           sdata[tid] += sdata[tid+1];
        }
```

The effects of unrolling the last warp is fairly noticable. We reduce the kernel time to 0.023936ms with bandwidth being 175.572 GB/s, giving a speed up of 1.442513369 times from the last kernel.

Now if we know the know the number of iterations at compile time, then we could completely unroll the reduction. Luckily, the block size is limiited by the GPU to 1024 threads, and we're sticking to powers of 2 for blocksizes. So we can easily unroll for a fixed block size, which we will use a precompiler derictive to define `blockSize`. Using this value all if statements involving `blockSize` will be evaualted at compile time. Therefore we can change the above for loop to a different statment

Listing 12: Unrolling all the loops

```
        if(blockSize >= 512){
                if(tid < 256)
                {
                        sdata[tid] += sdata[tid + 256];
                }
                __syncthreads();
        }
        if(blockSize >= 256){
                if(tid < 128)
                {
                        sdata[tid] += sdata[tid + 128];
                }
                __syncthreads();
        }
        if(blockSize >= 128){
                if(tid < 64)
                {
                        sdata[tid] += sdata[tid + 64];
                }
                __syncthreads();
        }


        if(tid < 32)
        {
            sdata[tid] += sdata[tid+32];
            sdata[tid] += sdata[tid+16];
            sdata[tid] += sdata[tid+8];
            sdata[tid] += sdata[tid+4];
            sdata[tid] += sdata[tid+2];
            sdata[tid] += sdata[tid+1];
        }
```

Using this method we shave the time down 0.02192 ms, and raise the bandwidth to 191.72GB/s, yielding a moderate speed up just about 1.1.

Before getting to the last optimization, lets consider the complexity of the reduce algorithm. We know that there are $\log_2(N)$ parallel steps, where each step S does $N/2^S$ independent operations. For $N = 2^D$, the reduction algorithm performs

$$\sum_{S\in[1\cdots D]} 2^{D-S} = N - 1$$

operations, making is *work efficient*. With P threads operating in parallel, the time complexity of the reduce is $\mathcal{O}(N/P + \log_2(N))$. Now we need to think about cost, the cost of the parallel algorithm is the number of processors times the time complexity. If we allocate N processors, then the cost is $\mathcal{N}\log(\mathcal{N})$, which is not cost effecient. Brent's theorem suggests that we use $\mathcal{O}(N/\log(N))$threads. Each thread will do $\mathcal{O}(\log(N))$ sequential work. So all $\mathcal{O}(N/\log(N))$ threads will cooperatie for $\mathcal{O}(\log N)$ steps, therefore resulting in $\mathcal{O}(N)$ cost. This is called *algorithm cascading*, and in practice can lead to significant speed ups.

To cascade the reduction, we will combine sequential and parallel reduction methods. Each thread loads and sums multiple elements into shared memory, then we will perform the tree based reduction in shared memory. Brent's theorem suggests that each thread should sum $\mathcal{O}(\log(N))$ elements. It can be sometimes beneficial to push this further, we can hide latency with more work per thread. We will do this using 32 elements per thread. The changes in this last reduction are almost entirely in the load/serial reduce:

Listing 13: Algorithm Cascading

```
    int myId = threadIdx.x + (blockDim.x*2)*blockIdx.x;
        int tid = threadIdx.x;
        int gridSize = blockDim.x*2*gridDim.x;
        sdata[tid] = 0;

        //load shared mem from global mem
        while(myId < n)
        {
                sdata[tid] += d_in[myId] + d_in[myId + blockDim.x];
                myId += gridSize;
        }
        __syncthreads();
```

Then upon kernel launch we have

```
reduce6<<<dimGrid6, dimBlock, 1024*sizeof(int)>>>(reduced, array, 32);
```

Using the algorithm cascading we reduce the elapsed time down to 0.008192 ms, offering 513GB/s bandwidth, and giving a 2.67578 times speed up.

Kernel	Time 2^{20} integers	Bandwidth	Step Speed Up	Cummulative Speed Up
Kernel 1	0.132096 ms	31.814 GB/s	-	-
Kernel 2	0.071264 ms	58.8198 GB/s	1.853x	1.853x
Kernel 3	0.062816 ms	66.9017 GB/s	1.134x	2.102x
Kernel 4	0.034528 ms	131.713 GB/s	1.819x	3.826x
Kernel 5	0.023936 ms	175.572 GB/s	1.442x	5.519x
Kernel 6	0.021920 ms	191.720 GB/s	1.092x	6.026x
Kernel 7	0.008192 ms	513.001 GB/s	2.676x	16.13x

Table 1: Performance of Reduction via optimization

So putting it all together, we can achieve a speed up of over 16 times. Here is an interesting oberservation:

- Algorithmic Optimizations By changing the addressing and using algorithm cascading we achieved 10.23× speed up collectively.
- Coding Optimization By unrolling the loops we only achieved 1.58× speed up collectively.

So a good rule of thumb is to optimizing your *algorithm* first, then optimize your code using unrolling of loops.

In conclusion – to fully optimize CUDA code, you should understand CUDA performance characteristics. Mostly, memory coalescing, divergent branching, bank conflicts, and latency hiding. Consider the algorithm that you are programming, ascertain if it is a compute limited algorithm or bandwidth limited. Using parallel algorithm complexity theory, we found how to cascade the algorithm, allowing for quite a substantial optimization. Identify bottlenecks in your algorithm, like we did with the memory and instruction overhead. Finally, be sure to optimize the your algorithm, and then optimize your code.