Excessively Technical Tricks

In our relentless and possibly endless pursuit of speed, let's look at a few more highly technical aspects of CUDA. Here we look at some of the warp-level primitives that let us access registers from other threads on the current one.

Recall that we can use three levels of memory on our CUDA devices: global memory, shared memory, and registers. These are listed in order of slowness. We understand that when we load CUDA device memory, our data are going into global memory. We also understand that shared memory is faster, but accessible only within each individual block of a grid. And we know that registers are the fastest memory, but can generally only be used by a single thread of our programs.

When I first started writing CUDA programs I wondered how I might get to this fast register memory. After all, I could allocate global and shared memory explicitly. What was the command to allocate some register memory? The answer is more or less that you can't. On the other hand, whenever you declare an ordinary variable in a kernel or device function, the memory for that is almost certainly a register. So, we understand that local variables in our kernels and device functions are in register memory.

There are a few CUDA primitives that allow one thread of a warp to access the registers of another thread in the same warp. It seems obvious that such functions would be useful, especially in reduction processes. We will use `__shfl_down_sync` to try to make our sum just a little faster.

The idea behind `__shfl_down_sync` is to grab the value of a register in another thread in the current warp that has a higher index. We can then e.g. add it to a register in the current thread, so that the current thread holds the sum from those two threads. Here is a simple device function to sum the registers for a given variable in each thread of a warp.

```
__inline__ __device__ double warpSum(double tot){
  unsigned mask = 0xffff;
  for(int stride=16;stride%lt;0;stride/=2){
    tot += __shfl_down_sync(mask,tot,stride);
    mask /= 2;
  }
  return tot;
}
```

The first thing you might notice is the `__inline__` declaration

- This class will be more in the nature of a seminar than a proper tutorial, chiefly because the instructor is not an expert in the topic. The aim is simply to provide some information that might useful in students' research and future employment. For that reason, we will try to be fairly relaxed about most traditional aspects of a class.

- Assignment 3 is posted.

The first thing you might notice is the `__inline__` declaration. This is probably unnecessary. It tells the compiler to place the entire code for this function at the point where it is called, instead of causing a jump to occur. This makes the compiled code run faster. On the other hand, NVCC knows to make many (maybe most) device functions inline already without asking, so our request is probably redundant. I put it in so I could talk about it. If the function were longer, perhaps it would not be inline by default, but if you had written the function mostly for readability, it might be important to you to make it inline. In that case, you get more readable code, without sacrificing speed to a jump.

The next thing you notice is that the only argument for this function is a single scalar. This will go into a register *for each thread in this warp.* It might take a minute to get your head around this. The point is that every thread in the warp is going to call this function simultaneously, and each such thread is going to feed its call some number. Those numbers may all be different. This function will operate on those 32 numbers, and return 32 new numbers, which will probably be stored in 32 registers on our SM.

The next line sets a value for a mask. The mask supposedly describes which threads are participating in this computation. I suppose that the idea is that if this is thread *n* of a warp, and if the i^{th} bit of the mask variable is one, then this the `__shfl_down_sync()` will run for this thread. If the i^{th} bit is 0, then the shuffle doesn't run. Much is made in the documentation of the importance of the mask, but as far as I can tell, it is not used at all on our hardware. I was able to set the mask to zero, and received the same results. I suppose that there is hardware out there for which the mask might be important. For the moment, I pretended that the mask should be taken seriously, set the first sixteen bits to one and the last 16 to zero. You can see that inside the loop, I cut the number of 1 bits in half at each step. Once again, though, on our devices, this makes no difference.

Finally, inside the loop we call `__shfl_down_sync()`. This function goes to the thread that has an index `stride` greater than the current thread, and grabs the value of the `tot` variable from that thread. You see that it adds that to the `tot` variable from the current thread. It starts with a stride of 16, cutting that in half at each iteration. The result is that at the end of the function, thread 0 has the sum of the input registers from all the threads in the warp. The rest of the threads in the warp have values that can be discarded.

This did a bisection reduction of a single warp. Now we need to use that to reduce an entire block. We have already discussed why our blocks are multiples of 32, so our code is built on the idea that the block is composed of some number of warps that is a power of two.

```
__global__ void blockSum(double *u, int n){
  int lane = threadIdx.x%32;
  int iWarp = threadIdx.x/32;
  unsigned iThr = blockDim.x*blockIdx.x+threadIdx.x;
  double total = u[iThr];
  total = warpSum(total);
  __shared__ double v[32];
  if(lane == 0) v[iWarp] = total;
  __syncthreads();
  // Do the final sum for the block now.
  total = threadIdx.x<blockDim.x/32? v[lane] : 0.;
  if(iWarp==0) total = warpSum(total);
  if(threadIdx.x==0) u[blockIdx.x] = total;
}
```

- This is a kernel, so it has no return type.
- `lane` gives us the index of this thread in the current warp.
- `iWarp` gives us the index of the warp within the block, while `iThr` is the absolute index of the thread among all the threads in the grid.
- The variable `total` lives in a register in the current thread, so we are effectively transferring from the global memory location `u[iThr]` to a register. That lets us call `warpSum()`.
- We throw the result into shared memory according to which warp we are in. Of course, we have to be sure to do that only for thread zero of each warp. Remember that shared memory is accessible to an entire block.
- We sync the block, and then do the operation one more time. This means we are adding the sums of the warps. Since the largest block size is 1024, then there are at most 32 warps, so all the warp totals fit in one warp - warp 0. That's why we only run the last `warpSum` on warp 0.
- Note the slightly sneaky way we used shared memory to load the warp sums into registers in the first warp. Those sums were calculated in registers in the various warps of the block, which are not generally accessible to warp 0. We put them into shared memory, which is accessible to the entire block, and then copied that into registers for the threads in warp 0.
- Note also that the documentation indicates that there are shfl_down etc. functions for cooperative groups, but it appears that our devices did not support these. Thus, we did not use groups here.

The code to call all this stuff, less the commands having to do with timing, looks like this.

```
  initKernel<<<blocksPerGrid, threadsPerBlock>>>(d_u,n);
  while(blocksPerGrid>=threadsPerBlock){
    blockSum<<<blocksPerGrid, threadsPerBlock>>>(d_u,n);
    blocksPerGrid /= threadsPerBlock;
    n /= threadsPerBlock;
  }
  sumKernel<<<blocksPerGrid, threadsPerBlock>>>(d_u,n);
```

The functions `initKernel` and `sumKernel` are not shown because they are uninteresting - they just initialize the global array, and sum whatever is left over after the bisection reductions, respectively. You can see then that we run the bisection reduction repeatedly, reorganizing the global memory in the process.

We include the results from the coordinated group experiments in Table 1 for comparison with what we got from the shuffle version.

Table 1 for comparison with what we get from the shuffle version.

Method	Total Time	Exec. Time
__syncthreads()	188ms	2.28ms
g.sync() managed	220ms	2.02ms
g.sync() unmanaged	189ms	2.67ms
g.sync() managed device array only	55ms	2.67ms
__syncthreads() shfl_down	49ms	2.27ms

Table 1. Times for simple reduction

These results are slightly mysterious. The execution time has increased somewhat, but it is such a small part of the total time for the computation that the overall time has decreased. More to the point, the initialization and copy back for this version of the code was the same as for the version with the device array only, so the copy time should have been the same. In practice, this was significantly faster. Why?

I noticed two things in testing this program. I noticed that there is asynchronicity in the copying from the device to the host. In particular, the values in the device array were not really available to the host until the timing sync after all the computations were done. On the other hand, the values on the card were correct according to debug output. It seems as if doing the managed memory allocation on the device shifts some overhead from the copy phase to the execution phase. I think the point is that using managed memory reduces latency on the device significantly. Apparently that memory management happens during the computation phase, and affects the execution timer, but saves time overall. In any case, it seems clear that forcing most of the computation onto registers within warps did save time overall.