Drexel University
Electrical and Computer Engineering Department

ECEC 622
Introduction to Parallel Computer Architecture

CUDA Labs 2 & 3
June 5th, 2013

Team Members:
Julian Kemmerer, jvk@drexel.edu
John Carto, jjc323@drexel.edu

Lab 2 - Vector Dot Product

Design

Computing the vector dot product on the GPU requires a combination of a few different techniques. As a dot product is simply the sum of all the products *A[i] * B[i]* where *i* is the index of the vectors, this part can be parallelized easily. Figure 1 below shows this process. Here, each thread uses a stride along with its thread ID for the starting position to sum the products. Having each thread stride instead of calculating a contiguous portion of the vector allows for memory coalescing. At the end, the thread-local sums are stored in shared memory.

```
/* thread ID and stride lengths (for coalescing memory) */
unsigned int tID = blockDim.x * blockIdx.x + threadIdx.x;
unsigned int stride_length = blockDim.x * gridDim.x;

/* initialize local thread sum and starting location for thread*/
float local_thread_sum = 0.0f;
unsigned int i = tID;

/* calculation loop */
while( i < numElements ) {
       local_thread_sum += A[i] * B[i];
       i += stride_length;
}
thread_sums[threadIdx.x] = local_thread_sum;
__syncthreads();
```

Figure 1. Code showing the main calculation loop with memory coalescing.

The next step is to combine these thread values so there is only a single value per block. This is done with a reduction process, as shown in Figure 2. It begins with half the threads in a block just summing two values, and each iteration the number of threads working decreases by a factor of 2. Eventually, this results in the block-wide sum being in the first index of shared memory for each block.

```
i = BLOCK_SIZE / 2;   // assumes block size is power of 2
while ( i != 0 ) {
       if ( threadIdx.x < i ) {
              thread_sums[threadIdx.x] += thread_sums[ threadIdx.x + i ];
       }
       __syncthreads();
       i = i / 2;
}
```

Figure 2. Code showing reduction process.

The final step is to combine each of the block sums. For this, an atomic function or

some other kind of synchronization is needed to provide mutual exclusive access to the global memory location used for the final sum. Without synchronization, race conditions can occur and cause an incorrect answer. The design here just uses a mutex lock implemented with the atomicCAS() and atomicExch() functions for lock and unlock behaviors.

For more specifics on this technique and implementation, please see the commented source files.

Performance

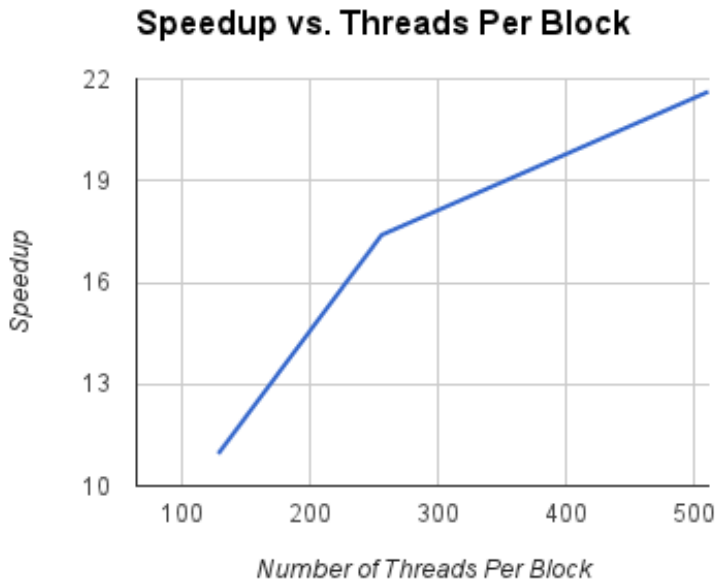

Figure 3. Performance results illustrating kernel sensitivity to changes in thread block size.

The CUDA implementation of the vector dot product was able to achieve noticeable speedup over its CPU counterpart. Grid size was fixed at 240, but block size was was variable. 128, 256, and 512 threads per block were tested. While it was initially thought that 256 threads would present the largest speedup, the data shows that 512 threads per block was the best setup.

Lab 3 - Cholesky Decomposition

Design

Computing the Cholesky Decomposition on the GPU does not leave room for memory-based optimizations as in the Vector Dot Product. However, by coalescing memory accesses for both the division and elimination steps significant performance increases are seen.

The inefficiency of a non-coalesced, single block approach is seen within the *chol_kernel* kernel. This kernel uses only a single block (as to allow the entire decomposition to take place in a single kernel) and 512 threads. The threads distribute outer *i-loop* iterations of the elimination step and that is all. The code is very similar to something implemented using PThreads. However, since this implementation relies on global memory and synchronization within the GPU - it is significantly slower than both the CPU and its optimized counterpart. Compared to CPU execution, *slowdown* averaged 3-4x.

The efficiency of a global memory coalesced access, multi-block, multi-kernel approach is seen within the *chol_kernel_optimized* and *chol_kernel_optimized_div* kernels. Each kernel call represents a single *k-loop* iteration. Synchronization between the division and elimination steps of the decomposition can easily be obtained as global memory is maintained between kernel calls and kernels are easily synchronized (as opposed to thread blocks).

It is desired that threads ‘stride’ across memory such that their accesses can be coalesced. A pseudo code snippet below shows both the division and elimination kernels.

```
elmination_kernel(Matrix U, int k, int stride):
	//Each thread block does a single i iteration
	int i = blockIdx.x + (k+1);
	//Each thread does some part of j
	int offset = i; //From original loop
	int jstart = threadIdx.x + offset;
	int jstep = stride;

	//Top limit on i for whole (original) loop
	int jtop = num_rows-1;
	//Bottom limit on i for whole (original) loop
	int jbottom = i;

	for(j = jstart; (j >= jbottom) && (j <= jtop); j+=jstep)
	{
		U[i * num_rows + j] -= U[k * num_rows + i] * U[k * num_rows + j];
	}
```

Figure 4. Pseudo code snippet showing the optimized elimination kernel.

```
division_kernel(Matrix U, int k, int stride):
	//General thread id
	int tx = blockIdx.x * blockDim.x + threadIdx.x;

	//Only let one thread take the square root of the diagonal element
	if(tx==0)
	{
		U[k * num_rows + k] = sqrt(U[k * num_rows + k]);
	}

	//Each thread does some part of j
	int offset = (k+1); //From original loop
	int jstart = threadIdx.x + offset;
	int jstep = stride;

	//Only continue if in bounds?
	int jtop = num_rows-1;
	//Bottom limit on i for whole (original) loop
	int jbottom = (k + 1);

	//Division only loops on 'j', so allow only a single thread block to complete this
	if(blockIdx.x == 0)
	{
		for(j = jstart; (j >= jbottom) && (j <= jtop); j+=jstep)
		{
			U[k * num_rows + j] /= U[k * num_rows + k]; // Division step
		}
	}
```

Figure 5. Pseudo code snippet showing the optimized division kernel.

As seen above, the elimination step consists of an outer *i-loop* and an inner *j-loop*. The *j-loop* is what moves across the memory space so it is ideal that the *j-loop* be split between threads and memory accesses coalesced. This was completed by allotting one thread block per outer *i-loop* iteration. Then, a predetermined (and optimal) number of threads was then split across the inner *j-loop*. Within the elimination kernel a constant *i* value is calculated based on the *blockIdx* and the *j* range is based on a striding memory access pattern. Similarly, the division kernel has a striding memory access pattern. However, since no *i-loop* is involved, only a single *blockIdx* is allowed to complete the computation.

For more specifics on this technique and implementation, please see the commented source files.

Performance

It was first required to determine the optimal number of threads per block (which is also the size of memory access strides). As the data below shows, a series of possible stride lengths (threads per block) were examined (for a matrix size of 2048) with the optimal speedup result being 256 threads per block.

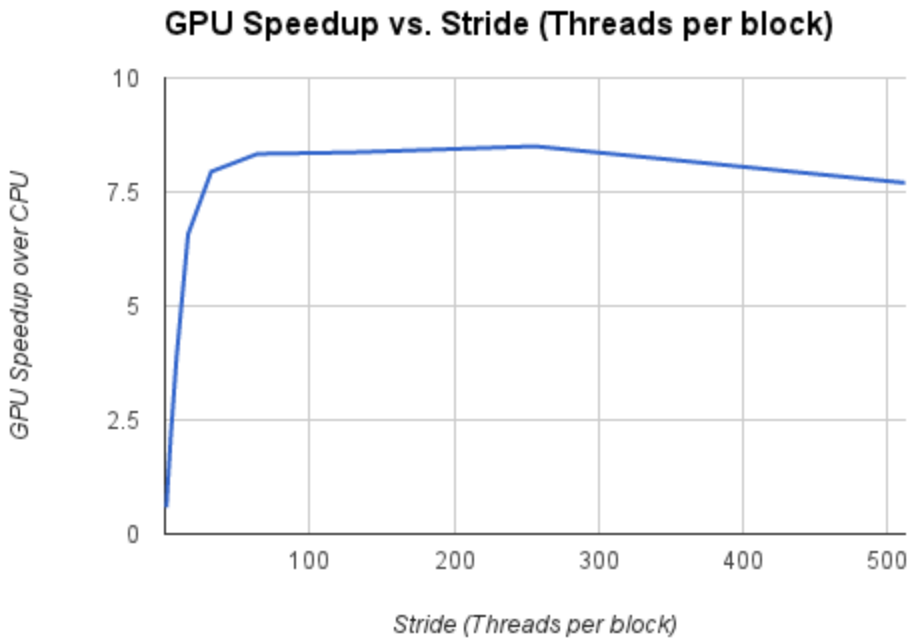

Figure 6. GPU Speedup as a function of threads per block (matrix size = 2048).

Once the threads per block / stride length was determined, the speedup for GPU over CPU execution for multiple matrix sizes was examined. Both the unoptimized and optimized GPU execution times are shown below.

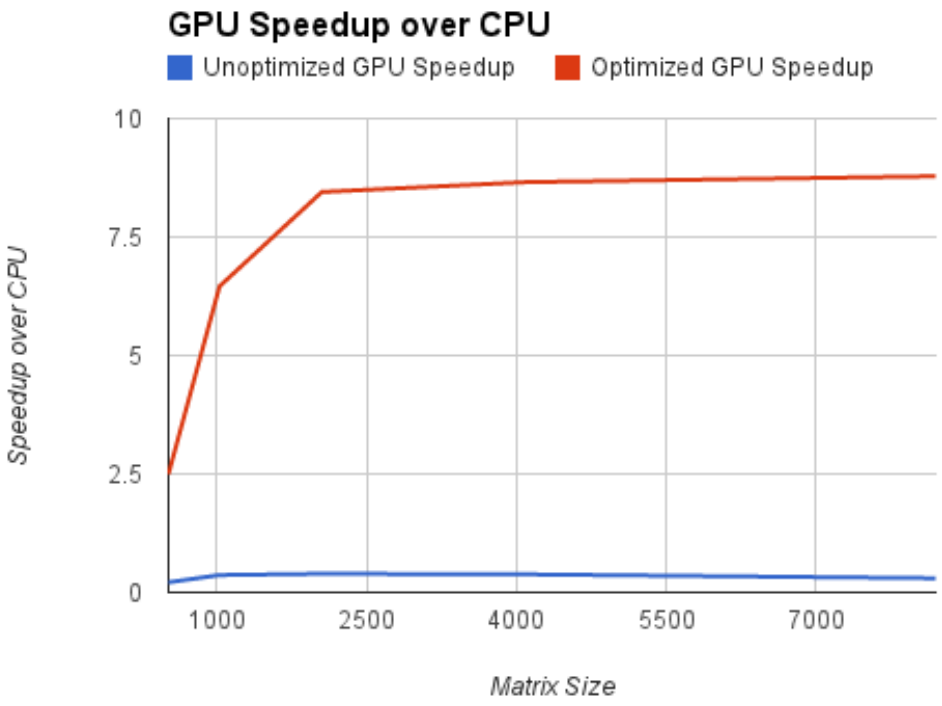

Figure 6. Speedup over CPU for unoptimized and optimized GPU implementations as a function of matrix size.

As mentioned previously, the unoptimized GPU code performs quite poorly while the optimized GPU code executes at near 9x rates.