LSU EE 7722 **Homework 4** **Due: 24 April 2014**

Read Stratton, J.A., Anssari, N., Rodrigues, C., I-Jui Sung, Obeid, N., Liwen Chang, Liu, G.D., and Hwu, W., "Optimization and architecture effects on GPU computing workload performance," Innovative Parallel Computing (InPar), May 2012, pp. 1-10. The paper describes how the evolution of NVIDIA GPUs affect programs written, and re-written, for them.

Problem 1: The Granularity Coarsening pattern was defined under the assumption that in an unoptimized kernel there would be one data element per thread. (The one-element-per-thread style is referred to as an *elemental style* of programming in the paper.)

(*a*) The discussion of granularity coarsening appearing in section 3.8 does not mention loop unrolling. Clearly granularity coarsening is a necessary condition for loop unrolling (when the starting point is something written in the elemental style). Pretend that you are a collaborator on the paper. Add a few sentences to Section 3.8 describing how granularity coarsening would enable loop unrolling and the additional benefits that loop unrolling would provide. In your answer show where your new material should be inserted.

(*b*) How should the number of iterations in a coarsened kernel be chosen? (Do not take in to account loop unrolling in your answer. That is, it should not matter whether or not loop unrolling is employed.) In your answer use N for the number of threads in the unoptimized kernel and use M for the number of multiprocessors. Use additional symbols as needed to describe characteristics of the GPU. (If in your answer you say "Let T denote the optimal number of threads per MP" then please explain why T threads is optimal.)

Problem 2: In the solution to Homework 3 shared memory was used to avoid scattered access to the data array. (The repository has the Homework 3 solution, and part of the solution is at the end of this assignment.)

(*a*) Describe which optimization pattern(s) most closely matches the approach used in the solution to Homework 3.

(*b*) The solution to Homework 3 is effective when the size of the data array is no more than a small multiple of the size of shared memory. (Recall that the size of the index array is much much larger than shared memory.) Describe which optimization patterns can be applied when the size of the data array more than a small multiple of shared memory size.

Problem 3: In the first paragraph of Section 4 the paper notes that "GPU workloads are in a state of perpetual hardware-software co-design and optimization." (The paper uses the term hardware-software co-design to refer to software tuning.) The paper takes into account NVIDIA devices up to CC 2.0. For this question consider CC 3.5 (the expensive Kepler) devices.

For a summary of the features in each hardware generation look in the NVIDIA CUDA C Programming Guide Appendix G (and elsewhere in the guide). On a system with CUDA installed look under `/usr/local/cuda/doc`, or online at
`http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

(*a*) Explain how you think the results in the paper would be changed for CC 3.5 (the expensive Kepler) devices due to the changes in the caching of global memory. In your answer discuss both the impact on optimized and unoptimized codes.

(*b*) Explain how you think the results in the paper would be changed for CC 3.5 (the expensive Kepler) devices due to the changes in the number of CUDA cores and schedulers (from CC 2.0 to CC 3.5) memory. In your answer discuss both the impact on optimized and unoptimized codes.

Part of Solution to Homework 3

```
extern "C" __global__ void sums_2()
{
  /// PROBLEM 3 SOLUTION
  //  Reduce noncontiguous access to dapp.d_data_array using shared memory.

  const int thread_count = blockDim.x * gridDim.x;
  const int tid = threadIdx.x + blockIdx.x * blockDim.x;

  const int dcache_elts = ( 1 << 15 ) >> 2;
  const int nrounds = ( dapp.data_array_elts + dcache_elts -1 ) / dcache_elts;

  for ( int round = 0; round < nrounds; round++ )
    {
      __shared__ float dcache[dcache_elts];
      const int chunk_start = round * dcache_elts;
      if ( round != 0 ) __syncthreads();
      for ( int sdidx = threadIdx.x; sdidx < dcache_elts; sdidx += blockDim.x )
        dcache[sdidx] = dapp.d_data_array[chunk_start + sdidx];
      __syncthreads();

      for ( int piece = tid; piece < dapp.num_pieces; piece += thread_count )
        {
          const int idx_piece_start = piece * clength;
          float sum = 0;
          for ( int i=0; i<clength; i++ )
            {
              const int didx = dapp.d_idx_array[idx_piece_start+i];
              const unsigned int sidx = didx - chunk_start;
              if ( sidx < dcache_elts ) sum += dcache[sidx];
            }
          if ( round == 0 )
            dapp.d_sum_array[piece] = sum;
          else
            dapp.d_sum_array[piece] += sum;
        }
    }
}
```