LSU EE 7700-2 Homework 1 Solution Due: 28 February 2011

To complete this assignment follow the setup instructions from the course Web page. The setup instructions bring you to the point where you can compile the cpu-only examples but for this assignment that won't be necessary. Next follow the Programming Homework Work Flow instructions to check out this assignment. These instructions describe an assignment from an earlier semester. For this semester, the main routine is in file `stream-2.cc` *and the executable (provided to gdb) is named* `stream-2`*. To solve this assignment both* `stream-2.cc` *and* `stream-2-kernel.cu` *will have to be edited.*

This code runs a simple stream program using CUDA, the same program used in classroom examples. It will run the code for a variety of block and grid sizes, though not necessarily enough to answer all the problems here. (You will have to modify the code to solve the problems.) See the comments in `stream-2.cc` *for details on how the code works.*

Problem 1: Compile and run the code unmodified. When the code is run the available GPUs will be listed. Find a machine with two GPUs and use that for this assignment. One should be of compute capability 1.3 and the other should be $2.x$. Indicate:

- The name of the machine you are running on.

  This solution was run on orion.ece.lsu.edu.

- The names of the GPUs.

  A GeForce GTX 480 and a Quadro FX 3800.

- The manufacturer's claimed memory bandwidth for each GPU.

To answer the last item above look for the manufacturer's specifications. That information is not provided in the program output.

The GTX 480 can transfer 177.4 GB/s and the Quadro FX 3800 can transfer 51.2 GB/s.

Problem 2: As we know, performance will be below its peak potential if there are an insufficient number of threads in a multiprocessor. Call the minimum number of threads needed to reach peak performance $T(n_{\rm B})$, where $n_{\rm B}$ is the number of blocks per multiprocessor. (To clarify, performance will be below peak if the number of threads is less than $T(B)$ and the performance will not be higher than peak if the number of threads is greater than $T(B)$.)

Use the code for this assignment to determine whether $T(n_{\rm B})$ does not depend on $n_{\rm B}$ (the number of blocks), whether $T(n_{\rm B})$ is smaller (a good thing) when $n_{\rm B}$ is smaller or whether $T(n_{\rm B})$ is smaller when $n_{\rm B}$ is larger.

(a) Modify `stream-2.cc` as needed to run the grid and block sizes needed to answer the question.

The strategy is to have an outer loop that varies the number of threads per multiprocessor and an inner loop that varies the block size. The modifications are in the routine `solution.cc:Stream_Demo::Start`, comments there explain the details. The outer loop prints the message "Configurations with X threads ..." while the inner loop shows the usual output. The following is sample output:

```
---- Configurations with 64 threads per multiprocessor:
Grid Dim     48,  Block Dim   32,  BL/MP     2.0,  TH/MP      64  Version 1
CUDA Time  8.739 ms  Throughput   28797.117 MB/s
Grid Dim     24,  Block Dim   64,  BL/MP     1.0,  TH/MP      64  Version 1
CUDA Time  8.844 ms  Throughput   28455.047 MB/s
---- Configurations with 128 threads per multiprocessor:
Grid Dim     96,  Block Dim   32,  BL/MP     4.0,  TH/MP     128  Version 1
```

```
CUDA Time  5.987 ms  Throughput   42036.980 MB/s
Grid Dim     48,  Block Dim   64,  BL/MP     2.0,  TH/MP     128  Version 1
CUDA Time  6.005 ms  Throughput   41904.820 MB/s
Grid Dim     24,  Block Dim  128,  BL/MP     1.0,  TH/MP     128  Version 1
CUDA Time  6.025 ms  Throughput   41766.395 MB/s
```

The output above shows two iterations of the outer loop (64 and 128 threads).

(*b*) Answer the question above about n_{B}. Indicate the configurations you ran and the results and comment on your confidence in the answer given the data collected and experiments performed.

The data rate obtained is nearly the same (varying by less than 1%) for different configurations with the same number of threads per multiprocessor. Therefore for the Quadro FX 3800 it does not matter if threads are in the same or different blocks for the stream code. Some additional data is shown below, a shortened version of the full output:

```
GPU 0: GeForce GTX 480 @ 1.40 GHz WITH 1535 MiB GLOBAL MEM
GPU 0: CAP: 2.0  MP: 15  TH/WP:  32  TH/BL: 1024  BL/GR 65535/65535/1
GPU 0: SHARED: 49152  CONST: 65536  # REGS: 32768
GPU 1: Quadro FX 3800 @ 1.20 GHz WITH 1023 MiB GLOBAL MEM
GPU 1: CAP: 1.3  MP: 24  TH/WP:  32  TH/BL:  512  BL/GR 65535/65535/1
GPU 1: SHARED: 16384  CONST: 65536  # REGS: 16384
Using GPU 1
Array size 4194304 elements,  data size (in and out) 50.332 MB.
---- Configurations with 64 threads per multiprocessor:
Grid Dim     48,  Block Dim   32,  BL/MP     2.0,  TH/MP      64  Version 1
CUDA Time  8.739 ms  Throughput   28797.117 MB/s
Grid Dim     24,  Block Dim   64,  BL/MP     1.0,  TH/MP      64  Version 1
CUDA Time  8.844 ms  Throughput   28455.047 MB/s
---- Configurations with 128 threads per multiprocessor:
Grid Dim     96,  Block Dim   32,  BL/MP     4.0,  TH/MP     128  Version 1
CUDA Time  5.987 ms  Throughput   42036.980 MB/s
Grid Dim     48,  Block Dim   64,  BL/MP     2.0,  TH/MP     128  Version 1
CUDA Time  6.005 ms  Throughput   41904.820 MB/s
Grid Dim     24,  Block Dim  128,  BL/MP     1.0,  TH/MP     128  Version 1
CUDA Time  6.025 ms  Throughput   41766.395 MB/s
---- Configurations with 256 threads per multiprocessor:
Grid Dim    192,  Block Dim   32,  BL/MP     8.0,  TH/MP     256  Version 1
CUDA Time  5.837 ms  Throughput   43114.133 MB/s
Grid Dim     96,  Block Dim   64,  BL/MP     4.0,  TH/MP     256  Version 1
CUDA Time  5.828 ms  Throughput   43181.367 MB/s
Grid Dim     48,  Block Dim  128,  BL/MP     2.0,  TH/MP     256  Version 1
CUDA Time  5.821 ms  Throughput   43235.492 MB/s
Grid Dim     24,  Block Dim  256,  BL/MP     1.0,  TH/MP     256  Version 1
CUDA Time  5.822 ms  Throughput   43225.754 MB/s
---- Configurations with 512 threads per multiprocessor:
Grid Dim    384,  Block Dim   32,  BL/MP    16.0,  TH/MP     512  Version 1
CUDA Time  5.878 ms  Throughput   42816.965 MB/s
```

Problem 3: The code contains three kernels, `dots_loopless`, `dots_stride_large`, and `dots_stride_small`. The original code just launches `dots_stride_large`, in this problem `dots_loopless` will be launched, and `dots_stride_small` is for the next problem.

As its name suggests `dots_loopless` does not contain a loop. It can be run if the total number of threads is equal to the number of array elements (by default 2^{20}). However, the code as written will never use it.

(*a*) Modify routine `dots_launch` so that `dots_loopless` is run if the number of array elements per thread is one, otherwise `dots_stride_large` is run.

The number of threads is simply the product of `blockDim` and `gridDim`. If this number is not greater than the array size the loopless version can be launched. Two versions of the solution is shown, one in `solution-kernel.cu:dots_launch` the other in `solution.cc:Stream_Demo::run`. Both pieces of code set a variable named `version` to `n` if the loopless kernel is to be run, a `switch` statement in `dots_launch` looks at `version`. Excerpts appear below:

```
In solution-kernel.cu:
  int thread_count = dg.x * db.x;
  if ( thread_count >= array_size ) version = 'n'; // Override version.

  switch ( version ) {
  case 'n': dots_loopless<<<dg,db>>>(); break;
  case 'l': dots_stride_large<<<dg,db>>>(); break;
  case 's': dots_stride_small<<<dg,db>>>(); break;
  }


In solution.cc:  version is passed to dots_launch.
    const char version =
      array_size <= db.x * dg.x ? 'n' :
      short_stride ? 's' : 'l';
```

(*b*) Run experiments to determine if performance is any better running `dots_loopless` than it is running `dots_stride_large` when there is one iteration per thread. Describe the experiments (block sizes, etc) and results.

Configurations were run at block sizes 256, 512, and 1024. The best throughput, 43003 was slower, but only by a small amount, than the best iterating kernels.

(*c*) Provide a possible reason for the results in the last part.

The time for initialization done in iterating kernels, though longer per thread, is slower overall because their are fewer threads. For example, a throughput of 43196 was achieved by the stride_large kernel at a block size of 256 and a grid size of 24, for a total of 6144 threads, that's much less than $2^2 2$ threads. Though total time spent initializing is much higher in the loopless kernel, bandwidth is only slightly lower because it is limited by memory bandwidth.

Problem 4: Modify routine `dots_stride_small` so that the array elements accessed by a block are contiguous. For example, if there are 1000 array elements and 10 blocks then block 0 should access 0-99, block 1 should access 100-199, etc. **Be sure that the code still runs efficiently.**

Modify `dots_launch` so that it calls `dots_stride_small`. The `stream-2.cc` routine will be print an error message if the code executes incorrectly, look out for these. CUDA will give an error message if `idx` is out of range.

See the comments in `solution-kernel.cu:dots_stride_small` for details.

```
__global__ void
dots_stride_small()
{
  /// SOLUTION
  //
  //  First, determine how many array elements each block should
  //  access:
  //
```

```
  int elt_per_block = ceilf( float(array_size) / gridDim.x );

  //  Determine the first and last+1 element to be accessed by this
  //  block (blockIdx.x).
  //
  int idx_block_start = elt_per_block * blockIdx.x;
  int idx_block_stop = min(array_size, idx_block_start + elt_per_block);

  //  Determine the first element to be accessed by this thread.
  //
  int idx_start = idx_block_start + threadIdx.x;
  //
  //  Note that because idx_start includes a "+ threadIdx.x" term
  //  consecutive threads will access consecutive array elements, which is
  //  necessary to construct fully occupied memory transactions.

  //  Determine how far ahead to skip each iteration.
  //
  int stride = blockDim.x;
  //
  // Note that the largest value of threadIdx.x is blockDim.x-1.

  for ( int idx = idx_start; idx < idx_block_stop; idx += stride )
    b[idx] = v0 + v1 * a[idx].x + v2 * a[idx].y;
}
```