Drexel University
Electrical and Computer Engineering Department

ECEC 622
Introduction to Parallel Computer Architecture

Programming Assignment 2 (CUDA Lab 1)
May 6th, 2013

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

Design

Global Memory

Preparation

The global memory implementation of vector multiplication involved several steps of preparation before moving to actual GPU computation. The first of which was to allocate space on the GPU and copy data from main memory to GPU memory. Once completed, the parameters of block and grid size must be specified. Blocks are memory and computation abstractions that contain a shared memory space for a specified number of thread. Our implementation uses blocks that contain 512 threads organized into a 2D array of width 1 (a 1D structure). The number of blocks is specified by the size of the grid. The grid was also organized into a 2D array of width 1 (1D array). The array length is determined by finding how many blocks can 'fit' into our specific problem size. That is, our grid size is determined by taking the problem size divided by the block size. It is at this point that timers are started and GPU computation begins.

Kernel

The global memory implementation of the vector multiplication kernel is relatively simple when compared to the shared memory implementation discussed later. Nearly all of the kernel is shown in the pseudo code included below:

```
thread_id = block_index * block_size + thread_index;
product = 0;

for( i from 0 to MATRIX_SIZE)
{
       product += A[MATRIX_SIZE*thread_id + i] * X[i];

}
return product;
```

Fig 1. Global memory kernel implementation pseudo code.

As mentioned, the above strategy is less complex than the shared memory version though still contains some notable features. The 'thread_id' is an integer representation of which thread is currently executing and also acts to represent where within the matrix the code is currently at. The 'A' matrix is stored as consecutive values in memory (as opposed to a 2D structure) and the 'thread_id' is used to access appropriate entries within this structure. Each thread computes using a single row of matrix 'A'. This row is accessed by computing the number of elements offset needed to reach this row in the 1-dimensional 'A' structure. This is the 'MATRIX_SIZE*thread_id' portion of the 'A' indexing. Each thread then loops over the entire row using the looping variable 'i'. The 'X' matrix is a one dimensional structure needing only the 'i' variable for memory access.

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

Shared Memory

Preparation

The preparation process is the same as described in the global memory section above, with the distinction that the block dimensions and grid dimensions are different. For the shared memory implementation, a block size of 256 threads is used and is organized into a 2D array of 16 by 16. The grid size used here is 256 by 1. This is found by dividing the matrix width by the block width, so in our case, 4096 / 16 = 256. It is important to note that these blocks won't cover every element themselves; they will start at the left side of the matrix, and be "walked" across in successive iterations. This is to achieve memory coalescing, and is explained more in the section below.

Kernel

When using shared memory, the most important thing is to make sure the memory transfers from global memory are coalesced. If not, each memory access is basically serialized as the device will only send one at a time. The below code excerpt shows how this is done. Since the thread blocks are further broken up into warps, each warp will allow all the global memory accesses to be sent across the bus in one package. This will happen twice an iteration in each warp, for both A and X. The loop then moves to the next 16-wide section of the matrix. The second part shown, the calculation, is only done by the first thread in each row, and the other threads wait before the next iteration. Finally, after the for-loop finishes, the first thread in each row reports its value and stores it in the resulting vector.

```
for ( int i = 0; i < MATRIX_SIZE; i = i + 16) {

              shared_A[ ty ][ tx ] = Ad[ MATRIX_SIZE * row_num + tx  + i ];
              shared_X[ tx ] = Xd[ tx + i ];
              __syncthreads();

              if ( threadIdx.x == 0 ) {
                     for ( int k = 0; k < blockDim.x; k++ ) {
                                   temp += shared_A[ tx ][ k ] * shared_X[k];
                     }
              }
              __syncthreads();
}

if ( threadIdx.x == 0 ) {
       Yd[ row_num ] = temp;
}
```

Fig 2. Shared memory kernel implementation pseudo code.

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

Performance

Theoretical

The GTX 275 GPU can achieve a peak processing rate of about 933 GFLOPs. The memory bandwidth on the device is 141.7 Gb/s. To achieve this peak processing rate we can examine a one second time span. In one second the device can theoretically transfer 141.7 Gb worth of data. In that same time span 933*10^9 floating point operations must be completed to fully utilize the device. This rate evaluates to 933*10^9 flop / (141.7/8) *10^9 bytes = 52.67 floating point operations per byte. Which, using 32 bit (4 byte) floating point values, evaluates to 210.69 floating point operations per floating point value loaded. The smallest load operation can be considered loading a single floating point value. Under this definition, the GPU must complete ~210 floating point operations per load operation to reach peak processing rate.

Actual

The number of floating point operations completed depends on the size and type of computation. For matrix vector multiplication the number of operations depends on the size of the two matrices. In our problem the first matrix contains 4096*4096 elements, while second contains just 4096. Matrix multiplication involves an addition and multiplication (2 operations) for each row and column combination evaluated. That is, there are 4096*4096 row-column combinations and, for each, 2 operations are completed - our problem involves 3.35*10^7 floating point operations (0.0335 Gflop). For any computation to be equivalent to peak GPU rates it must complete in approximate (3.35*10^7)/(933*10^9) = 0.035 ms. However, the mere transfer of (4096*4096 + 4096) elements (~1/2 Gb of data) at 141.7Gb/s is about 3.78 ms itself - one way. In theory the computation should take approximately 0.035 + 3.78 = 3.82 ms (as the transfer of 4096 elements back is negligible.

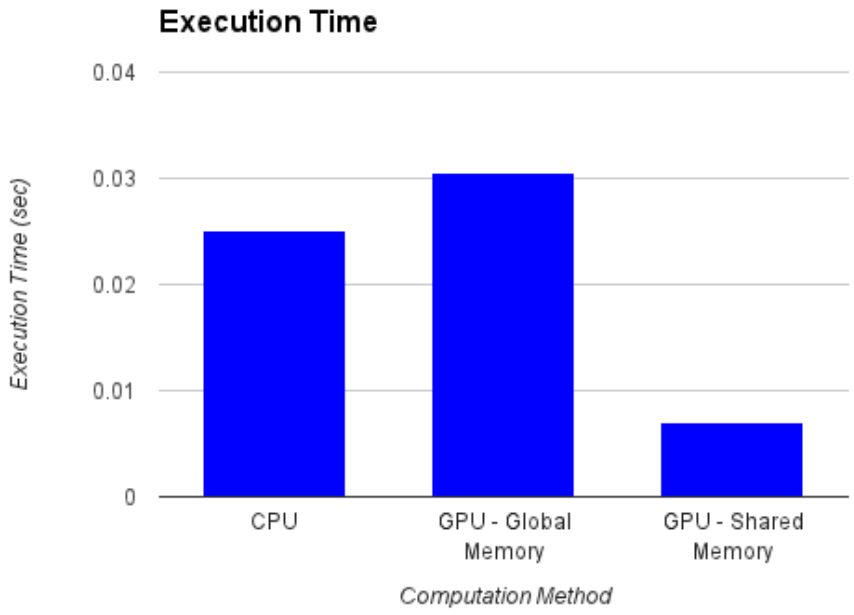

Fig 3. Execution time as a function of computation method.

The above figure highlights the benefits (and downfalls) of GPU computation. It may be surprising to see GPU execution take longer than CPU execution. However, this occurs when memory is not properly shared and cache lines are constantly refreshed - a decline in performance is expected. When memory is properly shared execution time is significantly lower than both GPU (global memory) and CPU execution.

Using global memory on the GPU, the computational utilization of the GPU can be calculated as (0.0335 Gflop) / (0.03 sec) = 1.11 GFLOPS or 0.11%. Using shared memory the computational utilization of the GPU can be calculated as (0.0335 Gflop) / (0.006 sec) = 5.58 GFLOPS or 0.59%.