Hardware/Software Co-Design

Review so far

Miaoqing Huang
University of Arkansas
Fall 2011
A student mentioned that he was able to multiply two $1,024 \times 1,024$ matrices using a tiled matrix multiplication code with 512 thread blocks on the Fermi GPU. He further mentioned that each thread in a thread block calculates one element of the result matrix. What would be your reaction and why?
Problem – I

- A student mentioned that he was able to multiply two $1,024 \times 1,024$ matrices using a tiled matrix multiplication code with 512 thread blocks on the Fermi GPU. He further mentioned that each thread in a thread block calculates one element of the result matrix. What would be your reaction and why?

- **Solution:** The explanation as given is clearly in error. Elements of a $1,024 \times 1,024$-element output matrix evenly divided among 512 thread blocks results in each block being responsible for 2,048 output elements. The statement that each thread computes a single output element implies that each thread block contains 2,048 threads; however, the programming model limits the thread block size to 1,024 threads. The description is therefore not possible under the given constraints.
The following kernel is executed on a large matrix, which is tiled into submatrices. To manipulate tiles, a new CUDA programmer has written the device kernel to transpose each tile in the matrix. The tiles are of size $\text{BLOCK\_SIZE} \times \text{BLOCK\_SIZE}$, and each of the dimensions of matrix $A$ is known to be a multiple of $\text{BLOCK\_SIZE}$. The kernel invocation and code are shown below. $\text{BLOCK\_SIZE}$ is known at compile time but could be set anywhere from 1 to 20.
dim3 blockDim(BLOCK_SIZE, BLOCK_SIZE);
dim3 gridDim(A_width/blockDim.x, A_height/blockDim.y);

BlockTranspose<<<gridDim, blockDim>>>(A, A_width, A_height);

__global__ void
BlockTranspose(float* A_elements, int A_width, int A_height)
{
    __shared__ float blockA[BLOCK_SIZE][BLOCK_SIZE];

    int baseIdx = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    baseIdx += (blockIdx.y*BLOCK_SIZE + threadIdx.y) * A_width;

    blockA[threadIdx.y][threadIdx.x] = A_elements[baseIdx];
    A_elements[baseIdx] = blockA[threadIdx.x][threadIdx.y];
}

- Out of the possible range of value for BLOCK_SIZE, for what values of BLOCK_SIZE will this kernel function correctly when executing on the device?
**Solution:** In current practice, `BLOCK_SIZE` value less than or equal to 5 will work correctly. A block configured in such a way will only contain one warp of threads, and will execute the entire program in lock-step for all the threads, correctly. A `BLOCK_SIZE` of over 5 will create thread blocks with more than one warp, and it is not guaranteed that the values written by one warp to the shared memory array will precede the reads by another warp from those same memory locations.

Without the implicit warp synchronization guarantee, only a `BLOCK_SIZE` of 1 would function correctly.
Problem – III

If the code does not execute correctly for all BLOCK_SIZE values, suggest a fix to the code to make it work for all BLOCK_SIZE values.
Problem – III

- If the code does not execute correctly for all $\text{BLOCK\_SIZE}$ values, suggest a fix to the code to make it work for all $\text{BLOCK\_SIZE}$ values.

**Solution:** To guarantee ordering of shared memory reads and writes across threads in this kernel, a `__syncthreads()` call should be placed between the lines that read and write the shared memory array.
Problem – IV

Consider the matrix addition where each element of the output matrix is the sum of the corresponding elements of the two input matrices. Can one use the shared memory to reduce the global memory bandwidth consumption? *Hint: Analyze the elements accessed by each thread and see if there is any commonality between threads.*
Consider the matrix addition where each element of the output matrix is the sum of the corresponding elements of the two input matrices. Can one use the shared memory to reduce the global memory bandwidth consumption? *Hint: Analyze the elements accessed by each thread and see if there is any commonality between threads.*

**Solution:** No. In this case, no threads share input data, so read-sharing cannot be exploited through cooperative use of shared memory.
__shared__ float partialSum[];
unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x>>1; stride > 0; stride >>= 1) {
    __syncthreads();
    if (t < stride) {
        partialSum[t] += partialSum[t+stride];
    }
}
The kernel in the previous slide is wasteful in the use of threads; half of the threads in each block never execute. Modify the kernels to eliminate such waste. Give the relevant execute configuration parameter values at the kernel launch. Is there a cost in terms of extra arithmetic operation needed?

```
__shared__ float partialSum[];
unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x; stride > 0; stride >>= 1) {
    __syncthreads();
    if (t < stride) {
        partialSum[t] += partialSum[t+stride];
    }
}
```
The kernel in the previous slide is wasteful in the use of threads; half of the threads in each block never execute. Modify the kernels to eliminate such waste. Give the relevant execute configuration parameter values at the kernel launch. Is there a cost in terms of extra arithmetic operation needed?

**Solution:** The configuration parameters should be modified such that the block size is half of what it was originally in this case. One operation (i.e., load) was added, and another (i.e., right shift) removed. Note that slight variations of equivalent implementations are possible.

```c
__shared__ float partialSum[];
unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x; stride > 0; stride >>= 1) {
    __syncthreads();
    if (t < stride) {
        partialSum[t] += partialSum[t+stride];
    }
}
```