Parallel Processing Lab 3: Introduction to CUDA - Matrix-Vector Multiplication

Sarah Peachey & Nathan Schomer

March 4, 2018

Abstract: GPU's can process SIMD code very quickly because it can run many threads on different data with the same instructions. First the CPU must create resources on the GPU and copy the data over, then the kernel code is launched. Once the kernel is launched the CPU can either run more code in parallel with the GPU or it can wait for the GPU kernel to complete. Then the CPU would transfer the output from the GPU back to the CPU and then free the resources on the GPU. The kernel code is ran on every thread in the thread blocks in the grid, which was allocated by CUDA functions in the CPU code.

1 Naive Design

To calculate the matrix vector multiplication of $Ax = y$ where A is a $n \times n$ matrix, x is a $n \times 1$ vector, and y is a $n \times 1$ vector. The naive approach is to create a thread to compute each element in the y vector. That thread then loads all the elements of the x vector as well as all the elements in that corresponding row of A. Then element-wise multiply and sum those values and store it back into global memory in the vector y.

So two values are loaded from global memory for every two floating point operations, so the arithmetic intensity of the algorithm is 1. Which means the code is memory bound, because it only uses 10 of the available 8800 GFLOPs. To achieve the peak performance rate 880 floating point operations must be computed per load operation, since the GTX 1080 GPU has a peak processing rate of 8800 GFLOPs and the memory bandwidth on the device is 320 GB/s. Another algorithm will be applied in the next section that uses shared memory to increase the arithmetic intensity.

Data: Ad, Xd, Yd
Result: kernel to calculate the matrix vector multiplication
initialization;
tid=threadIdx.y+(blockIdx.y*blockDim.y);
yTemp=0;
for *i < MATRIX SIZE* **do**
  yTemp+=A[tid*MATRIX SIZE+i]*X[i];
end
y[tid]=yTemp;

2 Shared Memory Design

The shared memory version of the matrix-vector multiplication algorithm described above takes the same inputs and produces the same result. However, this result is calculated in small pieces known as "tiles". This tiled approach to matrix-vector multiplication leverages spatial locality of the operands. For a tile of size $n \times n$, a thread block of size $n \times n$ will be created and each thread will load an element from the vector into shared memory and a single column of these threads will load elements from the vector into shared memory. The number of thread blocks (block grid height) will be calculated with $\frac{MATRIX\_HEIGHT}{TILE\_SIZE}$. An additional tile will be added to the grid if these are not evenly divisible.

Once in shared memory, the threads can perform matrix-vector multiplication on the current tile and add the partial sum to the resultant vector. Once all tiles are calculated, the result vector (of size $n \times 1$) will contain the result. A snippet of the kernel is included on the following page.

The arithmetic intensity of this is much higher since the number of loads from global memory is reduced but the number of floating point operations remain the same. This can be calculated by $\frac{2 \times n \times n}{n \times n + n}$. When $n = 1024$ the arithmetic intensity is approximately 2... a $2\times$ improvement over the naive method. This results in a performance of 20 FLOPS.

```
// moves tile across matrix
for(k=0; k<MATRIX_SIZE; k+=TILE_SIZE) {
    // check M edge conditions for this tile
    if(k + tileCol < MATRIX_SIZE && row < MATRIX_SIZE)
        M_shared[tileRow][tileCol] = Ad[row*MATRIX_SIZE + k + tileCol];
    else
        M_shared[tileRow][tileCol] = 0.0f;

    if (k + tileCol < MATRIX_SIZE)
        N_shared[tileCol] = Xd[k+tileCol];
    else
        N_shared[tileCol] = 0.0f;

    __syncthreads();

    for(temp = 0; temp < TILE_SIZE; temp++)
        partSum += M_shared[tileRow][temp] * N_shared[temp];

    __syncthreads();
}

if (row < MATRIX_SIZE)
    Yd[row] = (float)partSum;
```

3 Discussion of speed up

Speed-up was calculated for each version of the kernel by dividing the serial run-time by the parallelized run-time as seen in Equation 1. As seen by the speed-up ratios in Table 1, a trend of increased speed-up with increased matrix size was found. This is to be expected since an increased number of FLOPs on the GPU decreases the impact of kernel initialization and related GPGPU overhead.

$$s = \frac{t_{serial}}{t_{parallel}} \tag{1}$$

Size	Global Speed-Up	Shared Speed-Up
512×512	3.85	6.42
1024×1024	3.02	4.78
2048×2048	6.73	5.20

Table 1: Speed-Up calculated on Xunil-05