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

Sarah Peachey & Nathan Schomer

March 2, 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 $Ax = y$ when 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 if only uses 10 of the available 8800 GFLOPs. Since the GTX 1080 GPU has a peak processing rate of 8800 GFLOPs and the memory bandwidth on the device is 320 GB/s, to achieve the peak performance rate 880 floating point operations must be computer per load operation.

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

words

Data: start_x, end_x, num_traps, height

Result: estimated integral of function between start_x and end_x

initialization;

for *i < num_threads* **do**

    pack argument struct for thread i;

end

for *i < num_threads* **do**

    create thread i;

    serially calculate sub-trapezoid i;

end

for *i < num_threads* **do**

    join each thread;

    global_sum = global_sum + thread_sum;

end

3 Discussion of speed up

words

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

Thread Count	Global Speed-Up	Shared Speed-Up
2		
4		
8		
16		

Table 1: Average Speed-Up over 10 Iterations