Parallel Processing Final:

Jacobian Iterative Solver

Sarah Peachey & Nathan Schomer

March 19, 2018

1 Design

A while loop was used on the CPU to launch the GPU kernel and test for convergence. The kernel took some shared memory that is the size of the tile/thread block. Then loaded values from the A matrix and the x vector into those shared memory tiles. The tiles ensured that the memory accesses were coalesced, which increases performance. Each tile of shared memory was then matrix vector multiplied and saved to a partial sum vector. A reduction algorithm that also takes advantage of coalesced memory accessing was then performed on the partial sum vector to calculate $\sum_{m \neq k} a_{m,k} x_m$. Which is then subtracted from the corresponding value in the b vector and is divided by the value on the diagonal. This is the new x value which is then used to calculate the squared error.

1.1 Showing the tiled approach matrix multiplication.

```
1  if(k+tileCol!=row)
2   part_red[tileRow][tileCol]+=(double)A_shared
       [tileRow][tileCol]*(double)x_shared[
       tileCol];
3  else
4   part_red[tileRow][tileCol]+=0; //dont add up
        the diagonal element
```

1.2 Showing the reducation algorithm and difference error calculation.

```
1  int stride;
2  for(stride=TILE_SIZE/2; stride>0; stride/=2){
3   if(tileCol<stride && tileCol+stride <
       TILE_SIZE)
4    part_red[tileRow][tileCol]+=part_red[tileRow
        ][tileCol+stride];
5   __syncthreads();
6  }
7
8     if (col==0){
9   x_new[row] = ((double)Bd[row]-part_red[
       tileRow][0])/(double)Ad[row*MATRIX_SIZE+
       row];
```

```
10   double error =(Xd[row]-x_new[row])*(Xd[row]-
         x_new[row]);
11   atomicAdd(diff, error);
12   //everything is divided by the diagonal
         element
13 }
```

2 Discussion of speed up

Speed-up was calculated for the jacobian iterative solver on the GPU vs the CPU code with, Equation 1. As seen by the speed-up ratios in Table 1, with thread block the size of 32 threads there is considerable speedup, and increased speedup as the size of the matrix increased. This is probably do to the fact that since the matrix was larger more computation had to be performed so the data transfer time had less impact on the overal time. Furthermore, with a thread block the size of 16 threads the speedup was mostly slower except for the matrix with 2048 elements which was actually faster than with 32 sized thread blocks. But since most of the speeds are within an allowable tolerance from 16 to 32 sized thread blocks, I would say the code is not very sensitive to the size of the thread blocks.

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

Thread Block Size	Matrix Size	Speed-Up
16	512	2.131
16	1024	9.539
16	2048	28.503
32	512	2.117
32	1024	9.873
32	2048	24.962

Table 1: Speed-Up calculated on Xunil-05