"Hands On" Timing Task

Snippets

```
__global__ void sumMatrixOnGPU2D_n(float *A, float *B, float *C, int NX, int NY, int NUMDATA)
{
    unsigned int ix = NUMDATA * (blockIdx.x * blockDim.x + threadIdx.x);
    unsigned int iy = NUMDATA * (blockIdx.y * blockDim.y + threadIdx.y);
    unsigned int idx = iy * NX + ix;

    if (ix < NX && iy < NY)
    {
        unsigned int n = idx + NUMDATA;
        for (int i = idx; i < n; i++)
        {
            C[idx] = A[idx] + B[idx];
        }
    }
}
```

```
// setup kernel launch parameters
int dimx = 64; // default block size if no runtime parameters given
int dimy = 2;
int data_items = 1;
int singleDim = 0;

if(argc > 2)
{
    dimx = atoi(argv[1]);
    dimy = atoi(argv[2]);
    if (argc > 3)
    {
        data_items = atoi(argv[3]);
        if (argc > 4)
        {
            singleDim = atoi(argv[4]);
        }
    }
}
dim3 block(dimx, dimy);
dim3 grid;
```

```
if (!singleDim)
{
    double grid_x = (nx + block.x - 1) / block.x;
    double grid_y = (ny + block.y - 1) / block.y;

    double _grid_x = grid_x / sqrt(data_items); // Accounting for multiple data items per thread.
    double _grid_y = grid_y / sqrt(data_items);

    grid = dim3 _grid_x, _grid_y);
}
else
    grid = dim3(nxy / (block.x * block.y * data_items), 1);
```

```
if (data_items > 1)
    sumMatrixOnGPU2D_n<<<grid, block>>>(d_MatA, d_MatB, d_MatC, nx, ny, data_items);
else
    sumMatrixOnGPU2D<<<grid, block>>>(d_MatA, d_MatB, d_MatC, nx, ny);
```

Results

2D Block Size	2D Grid Size	Kernel execution time (ms)	1D Grid Size	Kernel execution time (1 datum) (ms)	1D Grid Size	Kernel execution time (16 data items) (ms)
16x16	256x256	2.939	N/A	N/A	4096	0.061
16x32	256x128	2.592	32768	0.529	2048	0.117
32x16	128x256	2.890	32768	0.522	2048	0.064
32x32	128x128	2.569	16384	0.568	1024	0.115
16x64	256x64	2.647	16384	0.590	1024	0.196
64x16	64x256	2.928	16384	0.562	1024	0.073
8x128	32x512	3.816	16384	0.641	1024	0.354

2D Block Size	2D Grid Size	Kernel execution time (ms)	1D Grid Size	Kernel execution time (1 datum) (ms)	1D Grid Size	Kernel execution time (16 data items) (ms)
128x8	512x32	2.568	16384	0.556	1024	0.063
4x256	1024x16	5.630	16384	0.858	1024	0.760
256x4	16x1024	2.561	16384	0.554	1024	0.060
2x512	2048x8	7.080	16384	1.370	1024	1.342
512x2	8x2048	2.574	16384	0.553	1024	0.057
1x1024	4096x4	12.663	16384	3.580	1024	2.646
1024x1	4x4096	2.611	16384	0.554	1024	0.057

Conclusion

When using a 2D grid the trend seems to be that when more weight is placed on the x dimension, the GPU is more efficient. Conversely when more weight is placed on the y dimension, there is an enormous penalty on the efficiency of the task.

When working with a single dimension grid, it is noted that there is a 4-5x speedup when compared to the 2D grid. The trend of speed growing with placing more weight on the x dimension also continues to be exhibited.

When working with multiple data items per thread (16), there is a further speedup of up to 11x. This also includes continued observance of previous trends.

The data suggests that work done on the GPU is most efficient when done in a single dimension and when grouping data items. This is likely due to a decrease in overhead cost as a result of setting up less threads and in less dimensions. It is also likely that the warp scheduler on the GPU itself is biased toward the x dimension.