HMW04 Report

Jacob Engel

April 5, 2019

1 Code Discussion

This lab used CUDA to solve the given differential equation. The necessary vectors were copied onto the GPU, and a Jacobian iteration was completed.

```
float w = 0.5;
int id = block(dx.x*blockDim.x + threadIdx.x;
        if (id>N+2&&id<(N+2)*(N+2)-(N+2)&&id%(N+2)!=N+1 &&id%(N+2)!=0){
        const float Ru =-u[id-(N+2)] -u[id+(N+2)] - u[id-1] - u[id+1];
        const rhs = (1./4)*(f[id]-Ru);
        unew = w*rhs + (1-2)*u[id];
        newu[id] = unew;
        res2v = (unew-u[id])*(unew-u[id]);
```

This is the piece of code that does the Jacobi iterations. Rev2u is the array that stores the residual for each point on the domain, and newu is the array that stores the next iterate of the solution vector.

```
int bid = blockIdx.x;
int I = blockDim.x*blockIdx.x+threadIdx.x;
float extra_sum = 0;
if (bid < Nblocks-1){
for (int s = blockdim*(Nblocks-1)/2; s>Nblocks-1; s/=2){
   if (I<s){
     res2_small[I] += res2[I+s];
     }
     __syncthreads();
    }
}
else{
  extra_sum += res2[I];
```

```
  }
__syncthreads();
res2_small[Nblocks-1] = extra_sum;
```

This is the theoretical reduction for the residual vector. It was unable to be implemented in the code. The for loop loops over half the length of a vector. Each iterate cuts the length in half. If the index was less than the "half" length, then the index plus the half gets summed to the index. The loop goes until it is one less than nblocks. This loop goes over only the filled blocks. At the end, the threads on the last block get summed individually and that sum was put on at the end of the array.

2 Architecture Discussion

The GPU system was set up as a contiguous array of blocks, rather than a matrix of blocks. This did not completely solve the problem of only making contigulus memory reads, but it did insure that every element had at 2 contiguous reads during the stencil. The initial conditions were established on the CPU, and then entered into a while loop. The Jacobian iterations and reisdual reductions were done on the GPU and were then moved back on to the CPU to check the accuracy. The halving forloop was built do allow for contiguous memory at the end, because the only elements that are read are the first Nblocks elements.

3 Nvprof results

3.1 Jacobian Iteration

	write throughput	read throughput	flopcount sp	iters
N = 100, nblocks = 4, threads = 32	102.69 mb/s	10.8232 mb/s	513	10
N = 100 nblocks = 2 threads = 128	244.71 mb/s	104.352 mb/s	1359	10
N = 1000 nblocks = 9 threads = 128	209.61 mb/s	608.98 mb/s	1341	27
N = 1000 nblocks = 32 threads = 32	250.09 mb/s	106.11 mb/s	1359	27

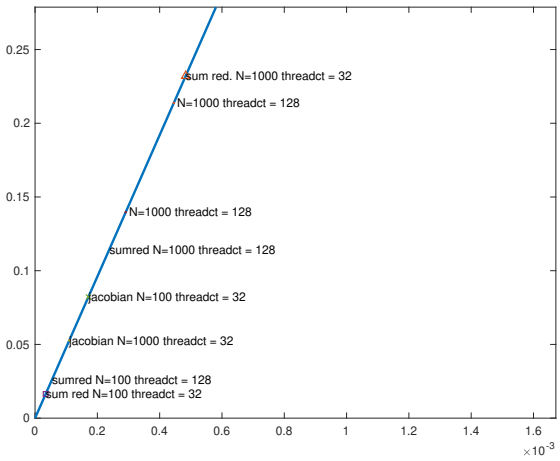
3.2 Vector Sum Reduction

	write throughput	read throughput	flopcount sp
N = 100 nblocks = 4 threads = 32	96.779 mb/s	4.166 mb/s	158
N = 100 nblocks = 4 threads = 128	106.18 mb/s	1.285 mb/s	254
N = 1000 nblocks = 9 threads = 128	209.61 mb/s	608.98 mb/s	1150
N = 1000 nblocks = 32 threads = 32	250.09 mb/s	1.302 mb/s	254

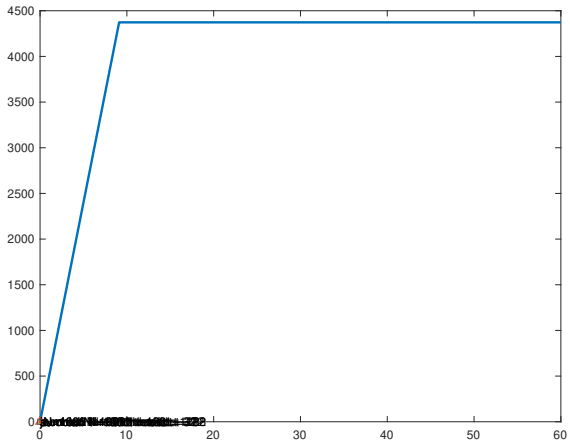
4 Roofline Model

It turned out that the code was not very efficient. The roofline model does not show where all of the various kernels were, so there is also a zoomed in version. The x axis of the model is arithmetic intensity (flops/byte) and the y axis of the model is performance in gflops.

4.1 Zoomed roofline graph



4.2 Unzoomed roofline graph



The computational performance of this code is lacking. Part of it was that in the actual implementation, the kernel that did the vector reduction did not work very well, and that reduction had to be serialized to some degree. The code performed similarly to serial code in terms of iterations for the standard residual (1e-6).