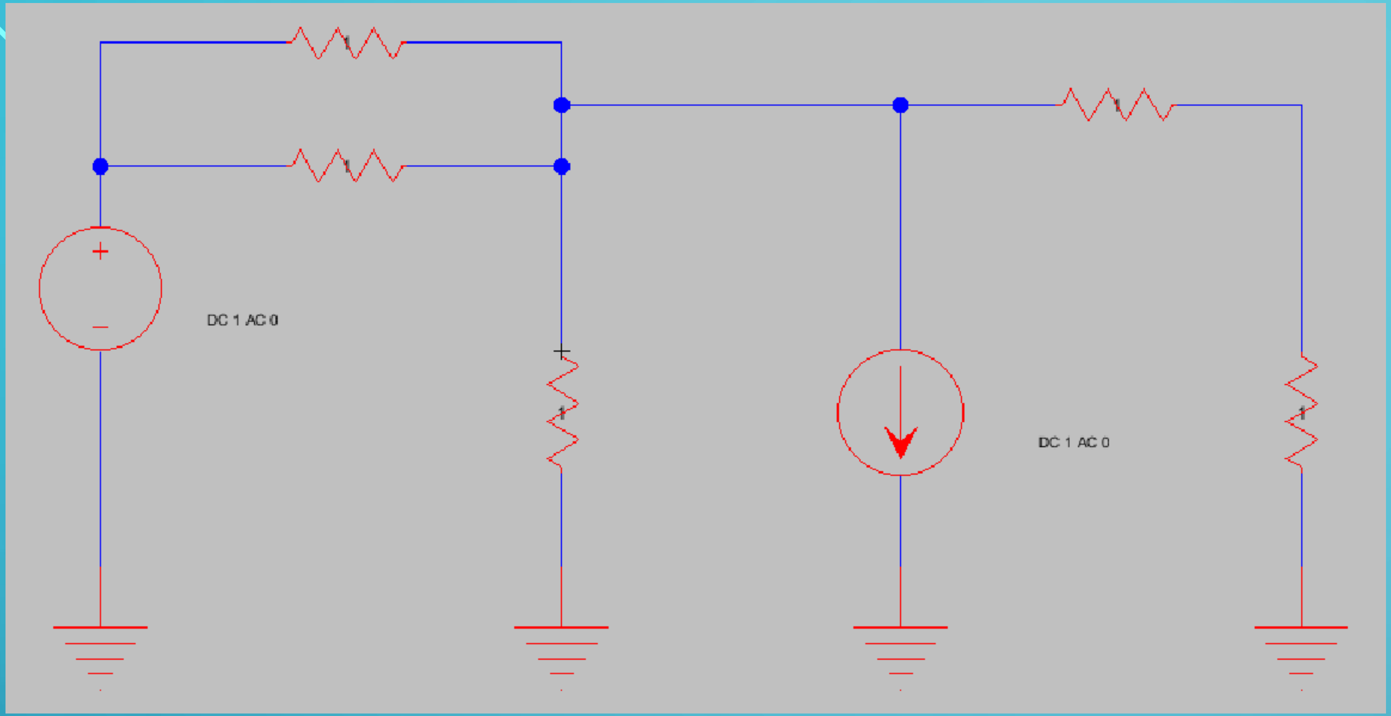
CUDA SPICE CIRCUIT SIMULATOR

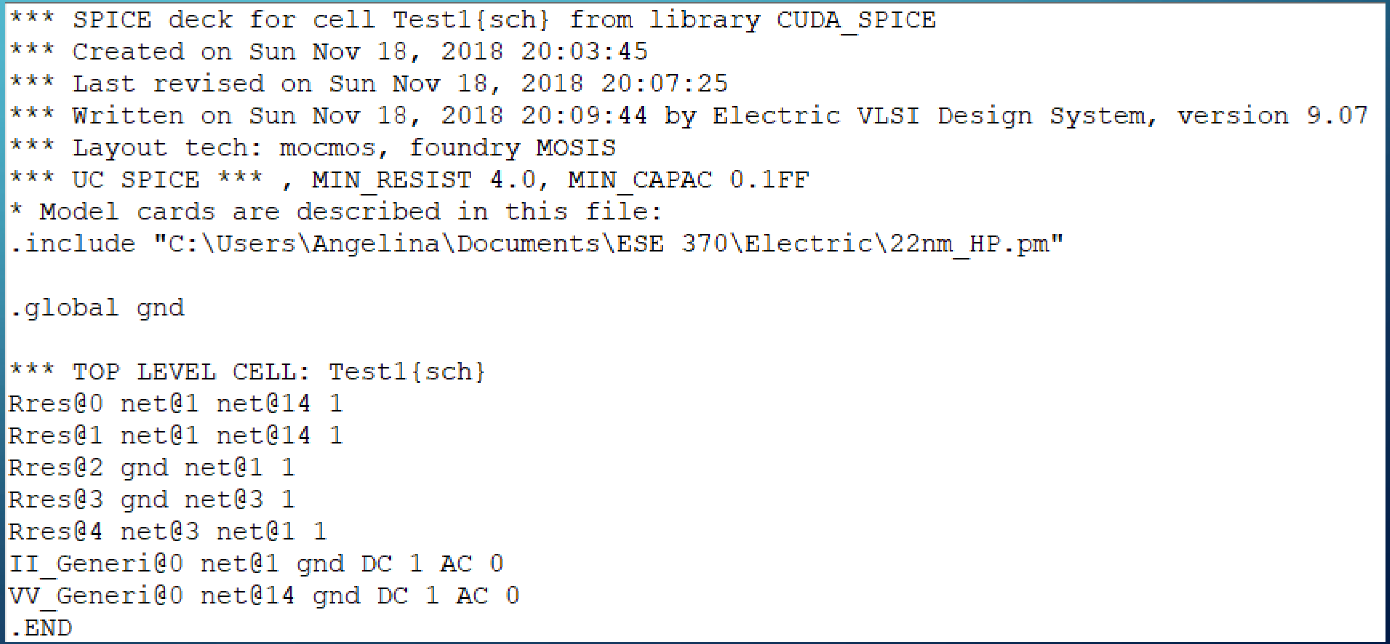
MILESTONE 1

ANGELINA RISI, EE 2018

WHAT IS SPICE?

- Simulation Program with Integrated Circuit Emphasis
- Basis of most modern simulators (ngspice, Cadence (PSpice), etc)
- First version in 1973
- Has fairly well-documented netlist/input file format





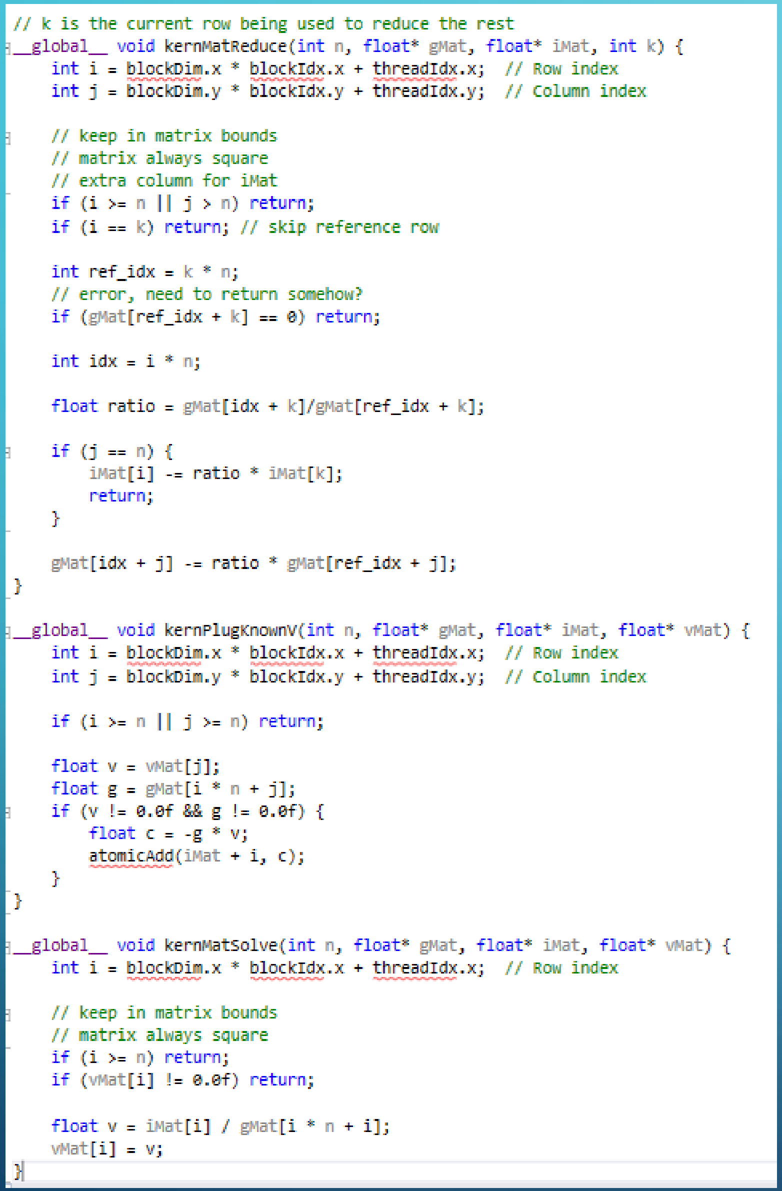
```
*** SPICE deck for cell Test1{sch} from library CUDA_SPICE
*** Created on Sun Nov 18, 2018 20:03:45
*** Last revised on Sun Nov 18, 2018 20:07:25
*** Written on Sun Nov 18, 2018 20:09:44 by Electric VLSI Design System, version 9.07
*** Layout tech: mocmos, foundry MOSIS
*** UC SPICE *** , MIN_RESIST 4.0, MIN_CAPAC 0.1FF
* Model cards are described in this file:
.include "C:\Users\Angelina\Documents\ESE 370\Electric\22nm_HP.pm"

.global gnd

*** TOP LEVEL CELL: Test1{sch}
Rres@0 net@1 net@14 1
Rres@1 net@1 net@14 1
Rres@2 gnd net@1 1
Rres@3 gnd net@3 1
Rres@4 net@3 net@1 1
II_Generi@0 net@1 gnd DC 1 AC 0
VV_Generi@0 net@14 gnd DC 1 AC 0
.END
```

MILESTONE 1 FEATURES

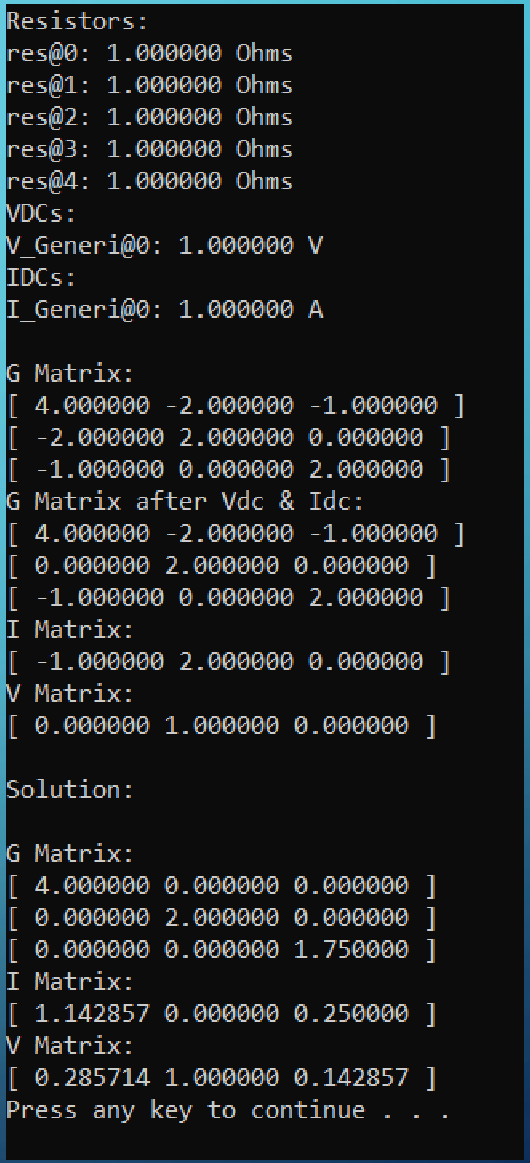
- Partial file parsing – only reads Resistors, VDC, IDC, VCCS elements from input SPICE netlists
- DC Operating Point simulation for simple, (ideally) fully linear circuits
    - Started with CPU implementation
    - Moved matrix solving code to GPU
- Began adding code for transistor (nonlinear) implementation
    - Element structure, current calculation
    - Currently working on framework for non-linear convergence

LINEAR SOLVER

- Circuit elements are converted into Conductance, Current, and Voltage matrices, initialized at 0.0f
- Kirchhoff's Current Law and Ohm's Law
- [G] * [V] = [I], solve for V
  - Matrix reduction
  - Plug in known voltages (e.g. voltage source with one end to GND)
  - Solve



```
// k is the current row being used to reduce the rest
__global__ void kernMatReduce(int n, float* gMat, float* iMat, int k) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;  // Row index
    int j = blockDim.y * blockIdx.y + threadIdx.y;  // Column index

    // keep in matrix bounds
    // matrix always square
    // extra column for iMat
    if (i >= n || j > n) return;
    if (i == k) return; // skip reference row

    int ref_idx = k * n;
    // error, need to return somehow?
    if (gMat[ref_idx + k] == 0) return;

    int idx = i * n;

    float ratio = gMat[idx + k]/gMat[ref_idx + k];

    if (j == n) {
        iMat[i] -= ratio * iMat[k];
        return;
    }

    gMat[idx + j] -= ratio * gMat[ref_idx + j];
}

__global__ void kernPlugKnownV(int n, float* gMat, float* iMat, float* vMat) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;  // Row index
    int j = blockDim.y * blockIdx.y + threadIdx.y;  // Column index

    if (i >= n || j >= n) return;

    float v = vMat[j];
    float g = gMat[i * n + j];
    if (v != 0.0f && g != 0.0f) {
        float c = -g * v;
        atomicAdd(iMat + i, c);
    }
}

__global__ void kernMatSolve(int n, float* gMat, float* iMat, float* vMat) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;  // Row index

    // keep in matrix bounds
    // matrix always square
    if (i >= n) return;
    if (vMat[i] != 0.0f) return;

    float v = iMat[i] / gMat[i * n + i];
    vMat[i] = v;
}
```



```
Resistors:
res@0: 1.000000 Ohms
res@1: 1.000000 Ohms
res@2: 1.000000 Ohms
res@3: 1.000000 Ohms
res@4: 1.000000 Ohms
VDCs:
V_Generi@0: 1.000000 V
IDCs:
I_Generi@0: 1.000000 A

G Matrix:
[ 4.000000 -2.000000 -1.000000 ]
[ -2.000000 2.000000 0.000000 ]
[ -1.000000 0.000000 2.000000 ]
G Matrix after Vdc & Idc:
[ 4.000000 -2.000000 -1.000000 ]
[ 0.000000 2.000000 0.000000 ]
[ -1.000000 0.000000 2.000000 ]
I Matrix:
[ -1.000000 2.000000 0.000000 ]
V Matrix:
[ 0.000000 1.000000 0.000000 ]

Solution:

G Matrix:
[ 4.000000 0.000000 0.000000 ]
[ 0.000000 2.000000 0.000000 ]
[ 0.000000 0.000000 1.750000 ]
I Matrix:
[ 1.142857 0.000000 0.250000 ]
V Matrix:
[ 0.285714 1.000000 0.142857 ]
Press any key to continue . . .
```

PLANS FOR NON-LINEAR CONVERGENCE

- SPICE apparently uses Newton-Raphson Method
  - Successive approximation until error within tolerance
- Start with “guess” voltages for unknown non-linear dependencies
  - Substitute into equations to populate matrices
- Linear solver for new voltage estimate
  - If outside of tolerance, new guess and try again
- Parallelization ideas:
  - Guess voltage and current calculations in parallel for each node
  - First guess multiple values in parallel and choose nearest?

IMPROVEMENTS NEEDED NEXT MILESTONE

- Complete netlist and model file parsing
- DC Sweep simulation
- Full 1st order transistor implementation (which means ignoring 2nd order effects, e.g. body, channel length modulation)
  - Working non-linear solver
- Branch current calculation