Improving Performance of an NBTI Simulator

A- Gupta

Johnny Yam
jyam@ucla.edu
Advisor: Professor Puneet Gupta

Abstract- The MATLAB version of the Negative Bias Temperature Instability (NBTI) simulator has poor runtime performance. By porting the code to C, optimizing the code with some basic performance increasing algorithms, and utilizing Graphic Processing Units (GPUs) to compute heavy arithmetic in parallel, we see an increase in performance as high as twelve times in the overall runtime compared to the MATLAB version.

I. INTRODUCTION

Negative Bias Temperature Instability has become a reliability issue in modern semi-conductors particularly in PMOS transistors that are almost always negatively biased. Because these transistors operate in this region, the NBTI degradation increases the threshold voltage and consequently decreases the drain current and transconductance of these MOSFETs. The NBTI simulator aims to model this problem numerically through a reaction-diffusion model [1].

The MATLAB version of the NBTI simulator is slow; thus, I aim to increase the performance of this simulator. Since it is widely known that running low level C code can have better runtime performance under certain situations when compared to MATLAB [2], I first port the MATLAB code into C. I perform my own optimizations using dynamic programming and divide and conquer algorithms on certain parts of the code. Then I further increased the performance of the simulator by performing the some of the calculations in parallel by utilizing GPUs with a NVDIA CUDA-enabled machine.

There are two main bottlenecks in the NBTI simulator: calculating the stress and low frequency matrices, which perform numerous matrix-matrix multiplications, and determining the profile vector, which utilizes the stress and low frequency matrices and performs multiple matrix-vector multiplications. The stress and low frequency matrices are both square matrices calculated by raising a base matrix to a certain power. Typical dimensions of the matrices include 4000x4000, where the stress matrix is typically made-up by the base matrix raised to the power of 20, while the low frequency matrix is made-up by the base matrix raised to the power of 50,000.

II. PROCEDURE

A. Matrix-Matrix Multiplication

The first step is to port the MATLAB version of the NBTI simulator to C code. To perform the matrix-matrix multiplications in C, I used the GSL CBLAS library, Level 3 CBLAS Functions [3]. Initially, I naively performed one matrix multiplication at a time in a linear fashion: I would multiply the two base matrices together to get a matrix with a power of 2, then multiple that matrix with the base matrix again to get a matrix with a power of 3, multiple the result with the base matrix again to get matrix with a power of 4, and so on. Doing the matrix multiplication this way was very slow, magnitudes slower than the MATLAB version.

Because of the nature of how the stress and low frequency matrices were generated, I realized that I could construct them through a hybrid of a divide-and-conquer and dynamic programming algorithm. For example, to generate the stress matrix with the base matrix raised to the power of 20, I could generate the matrix by constructing the highest power of 2, which is 16 (2^4), which can be efficiently generated by multiplying two base matrices raised to the power of 8, which can be generated by multiplying the two base matrices raised to the power of 4, which can be generated by multiplying two base matrices raised to the power of 2, which can be generated by multiplying two base matrices together. Then once I have a matrix whose value is equal to the base matrix raised to the power of 16, I can create the stress matrix by multiply that matrix with the matrix whose value is equal to the base matrix raised to the power of 4, which I already calculated previously and kept that value in a look-up table, so I don't have to recalculate it. Through these optimizations, I can generate the stress and low frequency matrices in O(log(n)) versus O(n) time, where n is the power the base matrix is raised to generate the stress matrix. Because the low frequency matrix is also comprised of the base matrix raised to some power, I can create the low frequency matrix in similar fashion.

B. Running Out of Memory

Although this provided a boost in the runtime performance of the NBTI simulator, the algorithm utilized a lot of space. For example, to generate the low frequency matrix with the base matrix raised to the power of 50,000, I would have to allocate memory for at least thirteen 4000x4000 matrices of type double (base matrix^{32768}, base^{16384}, base^{8192}, base^{4096}, et cetera). That along takes up over 1.5 Gigabytes!

To help combat this problem, I first determined which powers of two of the base matrix are need to calculate the stress and low frequency matrices. For the stress matrix (power of 20) for example, I need the power of 16 and the power of 4 matrices. Thus, I could free the memory of the power of 2 and power 8 matrices. Doing this, however, still was not enough to calculate the low frequency matrix when the grid points are large; I was still running out of memory. Thus, I resorted to serializing the needed matrices into files. I initially tried to use the TPL library to serialize the data. However, the C version of the NBTI simulator was still running out of memory. I then looked at the source code of the TPL Library and realized that in process of serializing the data, the TPL Library also tries to allocate memory for itself [4]! Thus, I resorted to just storing the needed matrices as binary data in files, temporarily freeing memory to be used else where, and retrieving the data by reading from the file when needed.

However, this still did not complete solve the problem as there were still situations, particularly when the matrices dimensions are large, when the program would run out of memory. As a result, I set a limit on the grid number to 5000. If a user tries to input a number larger than 5000, I will issue a warning that the program might and crash and prompt the user if he or she wishes to continue.

C. Matrix-Vector Multiplications

The second main bottleneck in the NBTI simulator is in a region of the code where there is a for-loop inside another for-loop. The inner for loop contains two matrix-vector multiplications and typically iterates over 1200 times. Hence, in the inner for-loop alone, there are typically over 2400 matrix-vector multiplications. The number of iterations for the outer for-loop is dependent on the simulation time; the longer a user wants to simulate for, the larger the number of iterations in the outer for-loop. I initially used the Level 2 CBLAS matrix-vector multiplication function from the GSL Library [3]. However, performance using this function was horrible. In fact, using the GSL Library gave poor performance in general; both a single matrix-matrix multiple and a single matrix-vector multiplication in MATLAB are noticeable faster than using their GSL CBLAS counterparts. Therefore, I started looking into different matrix multiplication libraries and functions. Through my initial inspection, I thought that the stress and low frequency matrices were sparse. Thus, I looked into sparse matrix operations and came across the NIST Sparse Blas [5]. To use this library, I converted and stored the stress and low frequency matrices into sparse format, and used the sparse versions in the matrix-vector multiplication in the double-loop area. Despite the fact that matrices are not sparse, utilizing the sparse matrix-vector multiplication function almost always had better runtime performance than the matrix-vector multiplication from the GSL Library.

Another option to improve performance was to use single precision floating point versus double precision. However, the output compared to the MATLAB version from the single precision did not match. Thus, double precision is needed.

D. Incorporating GPUs

To further improve the runtime performance of the NBTI simulator, I used the CUDA CUSPARSE Library to perform both the matrix-matrix multiplication in calculating the stress and low frequency matrices, and in determining the profile vector in the double for-loop area, which requires matrix-vector multiplications. The main issue with using CUDA is that the GPUs are on a separate device; therefore, I have to allocate and copy data back and forth from the host and the device. In addition, the CUSPARSE matrix multiplication functions expect the sparse matrices to be in row-major format. However, for dense matrices, the CUSPARSE functions expect them to be in column-major format. Since the CUSPARSE library only provides sparse-dense and no sparse-sparse or dense-dense matrix multiplication [6], I must format the matrices in both row-major and column-major formats. Despite all this overhead of moving data and formatting the matrices, we see a huge performance increase by utilizing the GPUs (see Fig. 1).

Because of these restrictions, I tried looking into the CUBLAS library. However, the CUBLAS Library only has support for single precision floating point [7], and as I mentioned earlier, the output various too much from the output with double precision.

III. RESULTS

I ran the MATLAB, C, and the CUDA version of the NBTI simulator five times each, varying the grid points and simulation time, and calculated the average runtime for each. I ran the MATLAB and C versions of the program on a machine with eight Intel® Xeon® E5335 CPUs. The CUDA version of the simulator ran on a machine with four Intel® Xeon® E5355 CPUs and an nVIDIA GeForce GTX260 graphics card. The following is a graph of the total runtime comparisons for the grid points (matrix dimension) equal to 4000:

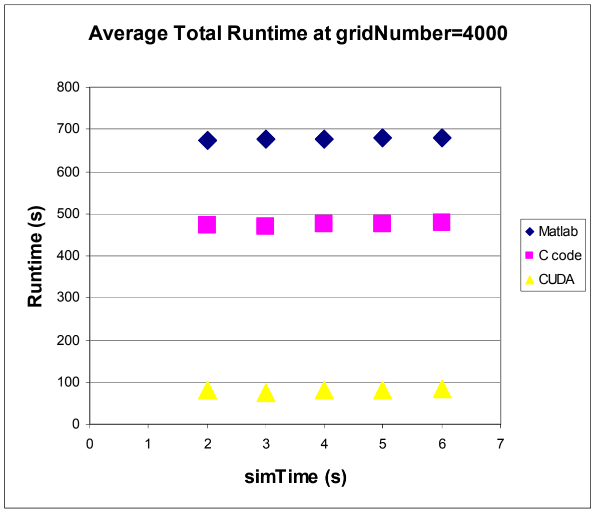

Fig. 1. Average total runtime of the NBTI simulator set at 4000 grid points versus simulator time

We see that when the number of grid points equals 4000, the CUDA version has overall on average of about a ten times increase in performance over the MATLAB version and about a six times increase in performance over the C version.

The following is the average runtime performance for calculating the stress and low frequency matrices versus the number of grid points:

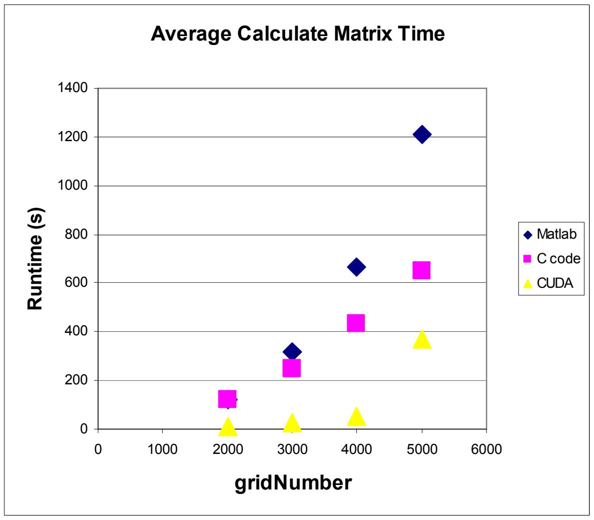

Fig 2. Average runtime to calculate the stress and low frequency matrices versus the number of grid points

Overall, we see that CUDA version performs significantly better than the MATLAB and the C code. At 4000, we see that CUDA has an over a twelve times increase in performance over the MATLAB and over a ten times increase at 3000 and 2000 grid points. However, at 5000, we see that the CUDA only has a three times increase in performance versus the MATLAB code. I theorize that this is because the amount of memory and multiplication operations that can get distributed across on the GPUs get saturated some where between 4000 and 5000 grid points. Thus, some of the operations have to wait for others to finish before it can use a GPU. In addition, because

Next let's look at the average runtime in the double for-loop region:

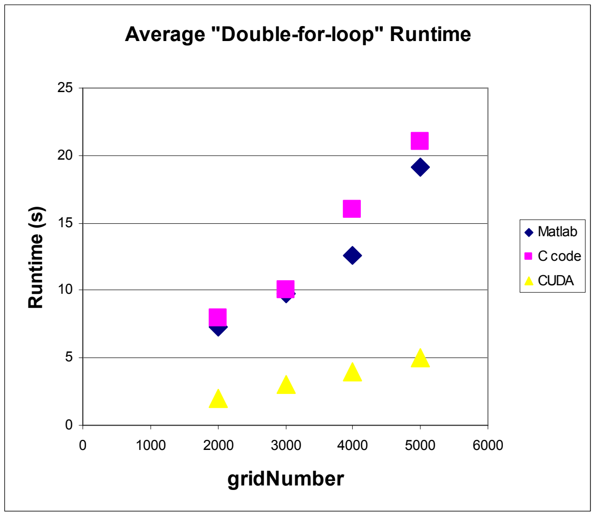

Fig 3. Average runtime in the double for-loop versus grid points

We see that the CUDA version is relatively consistent in performing four times better than the MATLAB version. We also notice that the C code version is slightly slower than the MATLAB code.

IV. CONCLUSION

I aimed to improve the runtime performance of the MATLAB version of the NBTI simulator by first porting the code to C and then to CUDA. Through utilizing more efficient algorithms, such as divide and conquer and dynamic programming, we see that the C version of the simulator overall performs slightly faster than the MATLAB version. I further increased the performance of the simulator by using GPUs to perform parallel calculations. We see that the CUDA version at grid points under 4000 has an overall performance increase of about ten times the MATLAB version and six times increase over the C code version.

V. FUTURE WORK

I was surprised that in the double for-loop region, where numerous matrix-vector multiplications are performed only had on average a four times performance increase when compared to the MATLAB version. Thus, I aim to create my own CUDA kernel function that performs the matrix-vector multiplication to see if I can achieve better performance.

In addition, I hope to make the C and CUDA versions of the simulator more robust, particularly when the number of grid points is large.

VI. ACKNOWLEDGMENTS

I would like to thank Dr. Puneet Gupta, Tuck Chan, Liangzhen Lai, and John Lee, for assisting me in development of the C and CUDA versions of the NBTI simulator.

REFERENCES

[1] Chan, Tuck-Boon, John Sartori, Puneet Gupta, and Rakesh Kumar. "On the efficacy of NBTI mitigation techniques" *IEEE/ACM 2011 Design, Automation and Test in Europe,* Mar 18, 2011

[2] Tiwari, Palak. "MATLAB Vs C." *MathKB: Your Mathematical Knowledge Base* . Advenet LLC, 17 May 2010. Web. 1 Jun 2011. <http://www.mathkb.com/Uwe/Forum.aspx/MATLAB/155333/MATLAB-Vs-C>.

[3] "GSL CBLAS Library- GNU Scientific Library -- Reference Manual." *Appendix D GSL CBLAS Library*. GNU GPL, n.d. Web. 1 Jun 2011. <http://www.gnu.org/software/gsl/manual/html_node/GSL-CBLAS-Library.html>.

[4] Hanson, Troy. "Efficient serialization in C." *TPL easily store and retrieve binary data in C*. N.p., n.d. Web. 1 Jun 2011. <http://tpl.sourceforge.net/>.

[5] Pozo, Roldan. "Sparse Basic Linear Algebra Subprograms (BLAS) Library." N.p., 01 Jan 2006. Web. 1 Jun 2011. <http://math.nist.gov/spblas/>.

[6] "CUDA CUSPARSE LIBRARY." NVIDIACorporation, Aug 2010. Web. 1 Jun 2011. <http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUSPARSE_Library.pdf>.

[7] "CUDA CUBLAS LIBRARY." NVIDIA Corporation, Aug 2010. Web. 1 Jun 2011. <http://developer.download.nvidia.com/compute/cuda/1_0/CUBLAS_Library_1.0.pdf>.