OpenCL Sparse Linear Solver for Circuit Simulation

Jason Mak

California Polytechnic State University
San Luis Obispo, CA
jamak@calpoly.edu

Matthew Guthaus

University of California, Santa Cruz
Santa Cruz, CA
mrg@soe.ucsc.edu

Abstract—**Sparse linear systems are found in a variety of scientific and engineering problems. In VLSI CAD tools, DC circuit analysis creates large, sparse systems represented by matrices and vectors. The algorithms designed to solve these systems are known to be quite time consuming and many previous attempts have been made to parallelize them. Graphics cards have evolved from specialized devices into massively parallel, general purpose computing units. With their parallel architecture and SIMD processing units, they are well designed for high-throughput operations on large matrices. Various APIs have been developed to allow users to access the resources of their GPUs. One relatively new API, OpenCL, provides a high level abstraction of GPU architecture. OpenCL, with its open source standard and support for both CPU and GPU compute devices, may become a dominant framework for parallel computing on GPUs in the future. Here, we test an OpenCL implementation of a sparse linear solver for VLSI CAD tools.**

I. Introduction

A circuit is represented by one or more voltage sources and the resistances of its wire connections. Circuit nodal analysis, a major step of DC analysis, tries to determine the voltage value at each node given the resistances and currents of the wires connecting to the node. The basis of nodal analysis relies on two well-known laws of electricity: Kirchhoff's current law and Ohm's law. By applying both laws to a branch in a circuit, a linear equation is generated with the nodal voltages as the unknowns. After combining linear equations for all branches, a linear system is created. A complex circuit may have nodes that number in the millions. Such a circuit may generate a linear system with a correspondingly large size.

Considering the size of these linear systems, circuit nodal analysis can be a very time-intensive task for VLSI CAD tools. In addition, transient analysis requires DC analysis to be performed repeatedly as the circuit changes over time. Various studies have tried taking advantage of parallel hardware to speed up linear solvers. Such parallel environments include networked distributed systems [1, 2], a transmuter array [3], a vector supercomputer [4], and graphics processing units [5], [6], [7], [8], [9], [10], [11]. Since GPUs are rapidly growing in popularity for general purpose computing, our work will also focus on this approach. Several platforms have been developed to enable the use of the GPU as a general purpose compute unit for running parallel code. For circuit analysis, various linear solvers were created with successful results on the CUDA platform. However, the Nvidia CUDA standard, in addition to being exclusive to Nvidia GPUs, is lacking in many features. Therefore, our work will focus on the advantages of a newly released framework, OpenCL. The flexibility of OpenCL and its accessible interfaces may lead it to become a major platform for developing parallel applications, including VLSI desktop applications.

In addition to a development platform, we must also select an algorithm to parallelize. Although GPUs have proven themselves quite effective in achieving speedup with their innate parallel architecture, this parallelism is mostly limited to data parallelism. In OpenCL, this is abstracted as a workgroup of work-items, which must run in lockstep to achieve the best performance results [12]. Sparsity is an important feature of matrices generated from circuit analysis. It presents a unique challenge because, in order to conserve memory, special data structures must be used to represent the matrices. The chosen algorithm must access and manipulate these special structures in an efficient way. LU-Decomposition is a well-known algorithm that was considered for testing. As a slow direct solver, this algorithm appeared to be a good candidate for achieving speedup through parallelization. We have found, for example, that the single-threaded implementation of LU in ngspice [13] would take hours to complete on some of our large circuit benchmarks. However, after attempting to create a parallel version of LU on the GPU and finding confirming statements in Feng's research [5], we determined that the current state of specialized data-level parallelism of GPUs and the irregularity of data structures needed to hold sparse matrices prevents the algorithm from being practically mapped to these devices. Our conclusion led us to explore another popular algorithm, the conjugate gradient method. Broken down, the computational bulk of this algorithm consists of simple matrix and vector operations [14]. In this report, we explore the results of using an existing OpenCL implementation of this algorithm to reach our desired goal of solving sparse linear systems for circuit simulation.

II. OpenCL and ViennaCL

Released in August 2009, OpenCL provides an API for parallel programming on a multitude of platforms [12]. Like CUDA, the OpenCL architecture has interfaces for accessing the computing resources of a GPU. However, unlike its competitor, OpenCL supports both major GPU vendors, AMD and Nvidia, and also supports multi-threaded computing on the CPU. With these features, OpenCL provides a framework that is more inclusive and flexible.

ViennaCL is an open source linear algebra library developed at the Vienna University of Technology in Austria [15]. Taking advantage of OpenCL's flexibility, ViennaCL

detects computing hardware (with preference given to GPUs) and uses it to run its parallel implementations of linear algebra solvers and matrix operations. ViennaCL's routines and data structures are accessed via high level C++ interfaces so that users do not have to worry about the intricate architecture of their computing devices. Many of these interfaces are compatible with the Boost uBLAS interfaces that are used in existing applications [19]. In our work, we were able to integrate ViennaCL's conjugate gradient solver into our existing code with relative ease.

III. Conjugate Gradient

CG is a well known iterative method for solving sparse linear systems. When the algorithm is applied to the linear system $Ax = b$, the vector x is guaranteed to converge to the correct solution only if the A matrix is both symmetric and positive definitive [14]. The circuits we are concerned with generate linear systems that exhibit both properties. As such, the use of parallelized CG for circuit analysis has been studied and many implementations exist for Nvidia's CUDA [7], [8], [9]. To conserve memory and prevent random data accesses, the sparse matrices can be stored in CSR format while the vectors retain their zeroes. The following version of CG, taken from Saad's book [18], is used in ViennaCL.

Algorithm 1 Conjugate Gradient

Initialize: $x_0 = 0$, $r_0 = b$, $p_0 = r_0$, $ip\_rr = inner\_prod(r_0, r_0)$

1: **for all** j such that $1 \leq j \leq N$ **do**
2: $tmp = matrix\_vector\_multiply(A, p_j)$
3: $\alpha = ip\_rr / inner\_prod(tmp, p_j)$
4: $x_{j+1} = x_j + \alpha * p_j$
5: $r_{j+1} = r_j - \alpha * tmp$
6. $new\_ip\_rr = inner\_prod(r_{j+1}, r_{j+1})$
7: $\beta = new\_ip\_rr / ip\_rr$
8. $ip\_rr = new\_ip\_rr$
9: $p_{j+1} = r_{j+1} + \beta * p_j$
10: **end for**

The maximum number of iterations N is specified by the user. Increasing the number of iterations may lead to a more accurate solution at the cost of performance. The primary computations of the algorithm are in lines 2-6 and line 9; they consist of matrix-vector multiplication, vector inner products, and scalar multiplication. Parallelizing each of these operations on a GPU is fairly straightforward. Studies have already been done on optimized matrix-vector multiplication on the GPU [16]. In these operations, accesses and modifications to matrix and vector elements occur independently and can run as separate, parallel threads in a SIMD fashion on the GPU. As indicated, the algorithm is easily dissected into parallelizable routines. This implementation allows for the parallel portions of the code to be decoupled and optimized separately.

IV. Results

A. Performance

A single-threaded implementation of CG and ViennaCL's implementation of CG were both tested on a set of large industry power grid circuits [17]. A few of these benchmarks have well over a million nodes. Single-threaded CG was tested only on an Intel Core 2 Duo processor, while the parallel implementation was also tested on an AMD Athlon II x4, a Nvidia GTS 250 GPU, and a Nvidia Tesla C2050. The results are indicated in Table 1 and Fig. 1. In all benchmarks other than *ibmpg1*, ViennaCL's parallel implementation showed significant improvement over the single-threaded implementation. The GTS 250 showed a good performance gain with a speedup of nearly a factor of 4 on most of the benchmarks. The Core 2 Duo and Athlon II, despite only having 2 and 4 cores, respectively, still performed well. This is due to the higher clocks of the CPUs, with the Athlon II clocked at 2.9 Ghz. It is also important to note that the parallel code in ViennaCL is not yet optimized for the memory layout of GPUs. Although powerful, the Tesla GPU is a non-consumer device. Despite being fully expected to perform well, its results still highlight the potential high performance gains from a GPU. As a final note, a significant portion of the measured times were not actually spent on executing CG. When our GPUs were tested for performance enhancement, the overhead of memory transfers to the devices could not be ignored.

B. Error

Each benchmark was tested with 1500 iterations of CG. Table 2 shows the average and maximum amounts of error for the solutions computed by the singled-threaded and parallel implementations of CG. The parallel implementation on the GPU does not magnify the error by a significant amount. Also, while increasing the number of iterations of CG does reduce the error, we found that the amount of error reduction quickly decreased as iterations were increased.

V. Future work

At the time of this writing, ViennaCL is still in the alpha phase of its release. While the library's interfaces are adequate, many features are missing and the internal matrix algorithms are far from optimal. The current implementation of CG fails to take advantage of shared local memory on the GPU architecture, which has the potential to be as fast as registers [12]. The next release, version 1.1, will likely address these issues.

Extra steps can also be taken to help CG, an iterative method, converge to a solution more quickly. The use of a preconditioner is an effective and commonly known way to do this. Although ViennaCL's current implementation of the CG preconditioner is parallel, it is unreasonably slow. Another factor of convergence time that can be studied is the initial guess of the solution vector x. ViennaCL's current implementation simply zeroes x initially. Setting an initial guess value may be particularly useful in transient analysis where the linear system changes over time and must be solved

TABLE I

DC ANALYSIS RUNTIMES USING CONJUGATE GRADIENT. T_S IS THE TIME MEASURED FOR THE SINGLE THREADED IMPLEMENTATION ON A CORE 2 DUO WHILE THE PARALLEL TIMES WERE MEASURED FOR A VARIETY OF DEVICES. *SPD* IS THE AMOUNT OF SPEEDUP FOR THE GPUS. SPEEDUP IS CALCULATED BY DIVIDING THE SINGLED THREADED TIME BY THE GPU TIME. EACH TIME IS A THE TOTAL OF THE ANALYSIS TIMES FOR GND AND VDD.

Circuit	*Size*	T_S	T_{Core2}	$T_{AthlonII}$	T_{GTS250}	T_{Tesla}	Spd_{GTS250}	Spd_{Tesla}
ibmpg1	30638	4.5	4.3	5.5	4.7	3.7	0.9X	1.2X
ibmpg2	127238	35	15	16	13	8	2.7X	4.4X
ibmpg3	851584	88	35	31	26	16	3.4X	5.5X
ibmpg4	953583	262	111	79	67	42	3.9X	6.2X
ibmpg5	1079310	154	63	48	42	27	3.7X	5.7X
ibmpg6	1670494	241	97	69	63	40	3.8X	6.0X

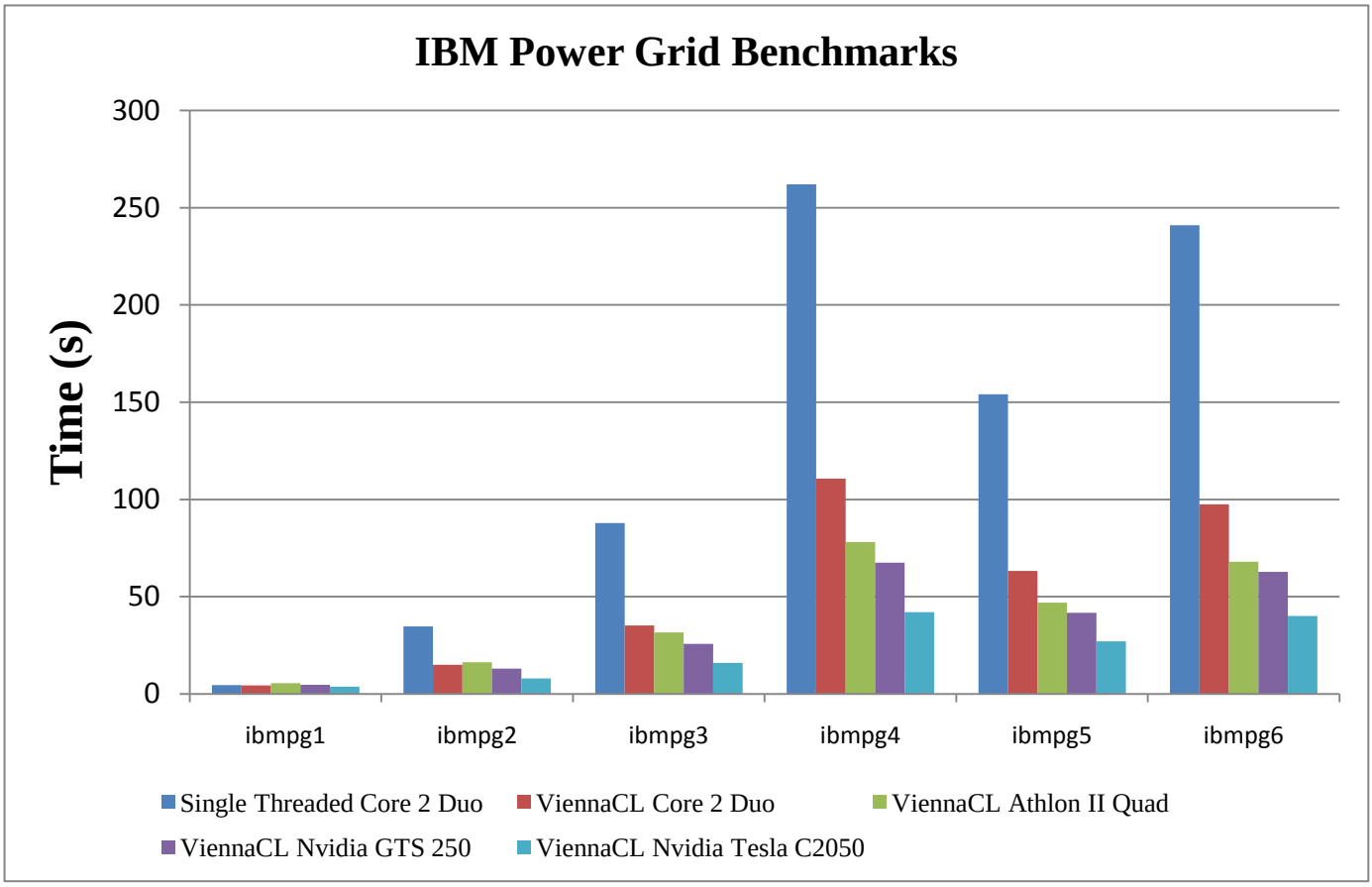

Fig. 1. Computation times of CG

TABLE 2

ERROR OF CONJUGATE GRADIENT. THE MAXIMUM AND AVERAGE ERROR ARE SHOWN FOR BOTH THE SINGLE THREADED AND PARALLEL IMPLEMENTATIONS. THE FORMAT IS GND/VDD. ERROR IS CALCULATED BY SUBTRACTING THE EXPERIMENTAL SOLUTIONS FROM THE EXACT SOLUTIONS OBTAINED FROM A DIRECT SOLVER. A "?" INDICATES THAT THE DIRECT SOLVER RAN OUT OF MEMORY FOR THE BENCHMARK AND WAS UNABLE TO PRODUCE A SOLUTION.

Circuit	Single Thread E_{avg}	Single Thread E_{max}	Parallel E_{avg}	Parallel E_{max}
ibmpg1	0/0	0/0	0/0	0/0
ibmpg2	0/0	0/1	0/0	0/1
ibmpg3	2/4	19/37	3/6	22/36
ibmpg4	3/?	4/?	3/?	4/?
ibmpg5	0/82	1/337	0/101	1/393
ibmpg6	2/?	10/?	2/?	11/?

repeatedly. An initial guess for the current system can be made based on the previous state.

Finally, more algorithms can be added to ViennaCL's linear algebra collection, including ones known to work well for circuits. For instance, the CUDA implementation of the multigrid method has shown great success in quickly solving linear systems generated by power grid circuits [5]. Such algorithms can be added to the internals of ViennaCL without compromising the convenient high level interfaces.

VI. CONCLUSION

In our study, we sought a GPU implementation of a linear solver for use in circuit simulation. We wanted to test the new framework, OpenCL, and the advantages it would have over CUDA. ViennaCL allowed us to take advantage of OpenCL's flexibility by using it as a black-box solver that was easily integrated into our existing interfaces. Despite its generic nature, ViennaCL's implementation of CG provided a significant amount of performance enhancement. In addition, it also highlighted the increasing accessibility of high performance computing without the need for a supercomputer or other special devices. Because of their convenient interfaces and success in improving performance, OpenCL and ViennaCL are viable tools for adding parallelism to existing VLSI CAD tools.

ACKNOWLEDGEMENT

This work was supported by the UCSC SURF-IT 2010 Research Experiences for Undergraduates Site, NSF grant CNS-0852099, <surf-it.soe.ucsc.edu>. We would also like to acknowledge the members of the VLSI-DA group at UCSC for their guidance and support. Last but not least, we would like to thank the staff and faculty at UCSC who contributed to the SURF-IT program.

REFERENCES

[1] K. Shen. Parallel sparse lu factorization on second-class message passing platforms. In ICS '05: Proceedings of the 19th annual international conference on Supercomputing, pages 351–360, New York, NY, USA, 2005.

[2] G. Trivedi, M. P. Desai, H. Narayanan. Parallelization of DC Analysis through multiport decomposition. In *20th International Conference on VLSI Design*, 2007.

[3] A. Mahmood, Y. Chu, and T. Sobh. Parallel sparse-matrix solution for direct circuit simulation on a transputer array. *IEE poc. Circuits Devices Syst.*, vol. 144, pp. 335-342, December 1996.

[4] P. Sadayappan and V. Visvanathan. Efficient sparse matrix factorization for circuit simulation on vector supercomputers. *IEEE Trans. CAD*, vol. 8, no. 12, pp. 1276-1285, Dec. 1989.

[5] Z. Feng and P. Li. Multigrid on GPU: tackling power grid analysis on parallel SIMT platforms. In *Proceedings of the ACM/IEEE ICCAD*, 2008.

[6] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. *Proc. ACM SC*, 22(3):917–924, 2005.

[7] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. on Graphics*, 22(3):917–924, 2003.

[8] W. Rodrigues, F. Guyomarch, Y. Le Menach, J. Dekeyser. Parallel sparse matrix solver on the GPU applied to simulation of electrical machines. Compumag 2009 Florianopolis Bresil, 11-2009.

[9] L. Buatois, G. Caumon, and Bruno Levy. Concurrent number cruncher: An efficient sparse linear solver on the GPU. *HPCC, LNCS*, pages 358–371, 2008.

[10] V. Volkov and J. Demmel, LU, QR and Cholesky factorizations using vector capabilities of GPUs. Tech. Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.

[11] M. Wang, H. Klie, M. Parashar, and H. Sudan. Solving sparse linear systems on NVIDIA Tesla GPUs. In *Proceedings of the 9th international Conference on Computational Science: Part I*, Baton Rouge, LA, May 25 - 27, 2009.

[12] OpenCL Tutorials. http://www.macresearch.org/opencl, 2010.

[13] Ngspice circuit simulator release 21. http://ngspice.sourceforge.net/, 2010.

[14] J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Carnegie Mellon University, Pittsburgh, PA, 1994.

[15] ViennaCL version 1.0.5. http://viennacl.sourceforge.net/, 2010.

[16] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. In *NVIDIA Technical Report NVR-2008-004*, December 2008.

[17] IBM power grid benchmarks. http://dropzone.tamu.edu/pli/pgbench/.

[18] Y. Saad. *Iterative methods for sparse linear systems*. Pacific Grove, California: PWS publishing, 1996.

[19] Boost Linear Algebra C++ Libraries http://www.boost.org/