**OpenCL Sparse Linear Solver for Circuit Simulation**

Jason Mak  
*University of California, Santa Cruz*  
Santa Cruz, 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 many common scientific and engineering problems. In VLSI CAD tools, performing DC circuit analysis can create large, sparse systems represented by huge matrices. Solving such systems can take orders of magnitude of time to compute. Many attempts have been made to parallelize algorithms to solve these matrices. Graphics cards, with their innate parallel architecture and SIMD processing units, can be used as general purpose computing units to operate on matrices when solving huge linear systems. Various APIs have been developed to allow users to access the resources of their GPUs. OpenCL is a newer API that was developed to allow high level access to GPU resources. OpenCL, with its open source standard, and support for both CPU and GPU compute devices may become a dominating framework for parallel computing on GPUs in the future. In our work, we test a sparse linear solver written in OpenCL on power grid circuits.

I. INTRODUCTION

A circuit is represented by a set a voltage sources and the resistances of the wires that connect them. Circuit nodal analysis, which is a major step of DC analysis, tries to find the voltage value at each node given the resistances and currents of the wires that connect to the node. The basis of nodal analysis relies on two well-known laws of electricity: Kirchoff’s current law and Ohm’s law. By applying both these laws to a branch in a circuit, a linear equation is generated with the nodal voltages as the unknowns. Combining all the linear equations, a linear system is generated. Complex circuits may have nodes that number in the millions. A linear system would be generated with a correspondingly large size. Another interesting feature of systems generated by circuits is their sparsity. With such large linear systems, circuit nodal analysis can be quite time consuming. In transient analysis, new systems are generated and DC analysis is performed repeatedly. Many attempts have been made to try and parallelize the algorithms that operate on the matrices that represent the linear system. Such parallel environments include 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, we have decided to take this approach. Several platforms have been developed to allow the use of the GPU as a general purpose compute unit to run code in parallel. For circuit analysis, various linear solvers have been developed with successful results on the CUDA platform. However, the Nvidia CUDA standard, in addition to being exclusive to Nvidia GPUs, is lacking in other features as well. Therefore, we choose to focus on a newly released framework, OpenCL. This framework has advantages over CUDA in that it is an open standard, supports CPUs as well as GPUs, and is supported on both major GPU vendors, ATI and Nvidia [12]. The flexibility of OpenCL and its accessible interfaces may drive it to become a major platform for developing parallel applications, including VLSI desktop applications.

In addition to selecting a platform, we must also choose an algorithm to parallelize. Although GPUs have proven themselves quite effective in achieving speedup with its innate parallel architecture, this parallelism is still limited to data parallelism. In OpenCL, this is abstracted as a workgroup of work-items, which must run in lockstep [12]. As mentioned previously, an important feature of matrices generated from circuit analysis is their sparsity. This presents a unique challenge because special data structures must be used to represent the matrices in order to conserve memory. Any algorithm we choose must be able efficiently manipulate these matrices in their special structures. One well-known algorithm we considered is LU-Decomposition. As a slow direct solver, this algorithm seems to be a good candidate to achieve speedup through parallelization. We have found, for example, that the single-threaded implementation of LU in ngslice [13] would take hours to complete on some of our large circuit benchmarks. However, in addition to studies by Feng [5] and our own attempts to create a parallel version of LU on the GPU, we determined that with the current state of specialized data-level parallelism of GPUs and the irregular data structures that must hold the sparse matrices, the algorithm cannot be practically mapped to these devices. Our discovery led us to explore another popular algorithm, the conjugate gradient method. At the simplest level, the computational bulk of this algorithm can be reduced to matrix-vector multiplication and vector inner products [14]. In this report, we explore the results of using this algorithm implemented on our chosen platform, OpenCL, to reach our desired goal of solving circuit analysis linear systems.

II. OPENCL AND VIENNACL

OpenCL is a relatively new framework for writing parallel programs on the GPU. ViennaCL is a linear algebra library developed at the Vienna University of Technology in Austria. It features various linear algebra solvers and matrix operations that are implemented on the GPU through OpenCL. ViennaCL seeks to make itself easy to integrate into existing applications through its high level C++ interfaces. The algorithms and operations are hidden to the programmers so that they do not have to worry about the various intricacies of GPU programming.
Algorithm 1 Conjugate Gradient

1. Compute: \( r_0 := b - Ax_0, p_0 = r_0 \)

2. for \( j=1 \) to (number of iterations), do

   3. \( \alpha_j := (r_j, r_j) / (Ap_j, p_j) \)
   4. \( x_{j+1} := x_j + \alpha_j p_j \)
   5. \( r_{j+1} := r_j - \alpha_j Ap_j \)
   6. \( \beta_j := (r_{j+1}, r_{j+1}) / (r_j, r_j) \)
   7. \( p_{j+1} := r_{j+1} + \beta_j p_j \)

CG is a well known algorithm for solving linear systems and has seen many implementations on CUDA. The heaviest computation resides in lines 3 and 4 with the matrix-vector multiplication and vector inner product.

IV. RESULTS

As shown through our performance graphs, ViennaCL’s implementation of CG yielded a speedup on the GPU by up to a factor of 4. Parallelism on the CPUs also yielded gains expected from having additional cores, although it was less than the that of the GPU. With smaller inputs, the difference was negligible but as the size of the matrices increased, so did the performance gain. It is also notable that a parallel implementation of CG had similar error as the single-threaded implementation. This error was calculated by taking the difference between our results and that of the true solutions for the matrices.

V. CONCLUSION

In our study, we sought to find a GPU implementation of a linear solver for use on our systems generated by large circuits. We sought to test out 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 good amount of performance enhancement. Because of its convenient interfaces and success in improving performance, OpenCL and ViennaCL should be viable tools for adding parallelism in existing VLSI CAD tools.

VI. 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, 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 can potentially be as fast as registers [12]. Future releases will likely address this issue.

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 still unreasonably slow and can be improved. Another factor of convergence time to study and improve is the initial guess of the solution vector \( x \). ViennaCL’s current implementation simply zeroes out \( x \) initially. Setting an initial guess value would allow for faster convergence not only in a single analysis, but can be applied in transient analysis where the linear system changes over time and must be solved repeatedly. An initial guess for the current system can be made based on the state of the previous.

Finally, ViennaCL can add more algorithms to its collection including ones that have been shown to be favorable for circuits. The multigrid method implemented with CUDA on a GPU, for instance, has shown great success in quickly solving linear system generated by circuits [5]. Such algorithms can be added to the internals of ViennaCL without compromising the convenient high level interfaces.
### IBM Power Grid Benchmarks

![IBM Power Grid Benchmarks](image)

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Size</th>
<th>Single Threaded Core 2 Duo</th>
<th>Parallel Core 2 Duo</th>
<th>Athlon II X4</th>
<th>GTS 250</th>
<th>Speedup on GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibmpg1</td>
<td>30638</td>
<td>4.5/4.7</td>
<td>4.3/4.8</td>
<td>5.5/5.8</td>
<td>4.7/5.0</td>
<td>0.9X</td>
</tr>
<tr>
<td>ibmpg2</td>
<td>127238</td>
<td>35/37</td>
<td>15/19</td>
<td>16/19</td>
<td>13/15</td>
<td>2.7X</td>
</tr>
<tr>
<td>ibmpg3</td>
<td>851584</td>
<td>88/93</td>
<td>35/46</td>
<td>31/38</td>
<td>26/32</td>
<td>3.4X</td>
</tr>
<tr>
<td>ibmpg4</td>
<td>953583</td>
<td>262/277</td>
<td>111/138</td>
<td>79/96</td>
<td>67/83</td>
<td>3.9X</td>
</tr>
<tr>
<td>ibmpg5</td>
<td>1079310</td>
<td>154/165</td>
<td>63/81</td>
<td>48/60</td>
<td>42/53</td>
<td>3.7X</td>
</tr>
<tr>
<td>ibmpg6</td>
<td>1670494</td>
<td>241/255</td>
<td>97/124</td>
<td>69/86</td>
<td>63/80</td>
<td>3.8X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Single Threaded E_{avg}</th>
<th>Single Threaded E_{max}</th>
<th>Parallel E_{avg} (ViennaCL)</th>
<th>Parallel E_{max} (ViennaCL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibmpg1</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
</tr>
<tr>
<td>ibmpg2</td>
<td>0/0</td>
<td>0/1</td>
<td>0/0</td>
<td>0/1</td>
</tr>
<tr>
<td>ibmpg3</td>
<td>2/4</td>
<td>19/37</td>
<td>3/6</td>
<td>22/36</td>
</tr>
<tr>
<td>ibmpg4</td>
<td>3/?</td>
<td>4/?</td>
<td>3/?</td>
<td>4/?</td>
</tr>
<tr>
<td>ibmpg5</td>
<td>0/82</td>
<td>1/337</td>
<td>0/101</td>
<td>1/393</td>
</tr>
<tr>
<td>ibmpg6</td>
<td>2/?</td>
<td>10/?</td>
<td>2/?</td>
<td>11/?</td>
</tr>
</tbody>
</table>

### ACKNOWLEDGMENT

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


