The Challenges of Writing Portable, Correct and High Performance Libraries for GPUs

Miriam Leeser
Department of Electrical and
Computer Engineering
Northeastern University
Boston, MA
mel@coe.neu.edu

Devon Yablonski
Mercury Computer Systems
Chelmsford, MA
dyablons@mc.com

Dana Brooks
Department of Electrical and
Computer Engineering
Northeastern University
Boston, MA
brooks@ece.neu.edu

Laurie Smith King
Department of Mathematics
and Computer Science
College of the Holy Cross
Worcester, MA
lking@holycross.edu

ABSTRACT

Graphics Processing Units (GPUs) are widely used to accelerate scientific applications. Many successes have been reported with speedups of two or three orders of magnitude over serial implementations of the same algorithms. These speedups typically pertain to a specific implementation with fixed parameters mapped to a specific hardware implementation. The implementations are not designed to be easily ported to other GPUs, even from the same manufacturer. When target hardware changes, the application must be re-optimized.

In this paper we address a different problem. We aim to deliver working, efficient GPU code in a library that is downloaded and run by many different users. The issue is to deliver efficiency independent of the individual user parameters and without *a priori* knowledge of the hardware the user will employ. This problem requires a different set of tradeoffs than finding the best runtime for a single solution. Solutions must be adaptable to a range of different parameters both to solve users' problems and to make the best use of the target hardware.

Another issue is the integration of GPUs into a Problem Solving Environment (PSE) where the use of a GPU is almost invisible from the perspective of the user. Ease of use and smooth interactions with the existing user interface are important to our approach. We illustrate our solution with the incorporation of GPU processing into the Scientific Computing Institute (SCI)Run Biomedical PSE developed at the University of Utah. SCIRun allows scientists to interactively construct many different types of biomedical simulations. We use this environment to demonstrate the effectiveness of the GPU by accelerating time consuming algorithms in the scientist's simulations. Specifically we target the linear solver module, including Conjugate Gradient, Jacobi and MinRes solvers for sparse matrices.

1. INTRODUCTION

There has been an explosion of interest in accelerating general purpose applications on Graphics Processing Units (GPGPU), resulting in what we call the age of "heroic programming" for GPGPUs. A scientist chooses both an application to accelerate and a target platform and then, with great effort, maps the application to that platform. If they are a true hero, they achieve two or three orders of magnitude speedup for that application and target hardware pair. The effort required includes a deep understanding of the application, its implementation and the target architecture. When a new, perhaps higher performance architecture becomes available, additional heroic actions are required to achieve speedup for the same application.

Most scientists would prefer to spend their time focused on the application level rather than the details of the implementation. These scientists would like to use GPUs for their applications, but would prefer to ignore such issues as numbers of threads and thread blocks, instruction level parallelism, etc. The research described in this paper aims to help this group of scientists by providing parameterized library components that deliver high performance over a range of input parameters and hardware platforms while requiring no heroics on the part of the user.

At first glance, OpenCL [4] appears to solve the problems of writing portable libraries. However, while there are compilers from OpenCL to several target platforms including GPUs, OpenCL is written at a much lower level than the libraries that we are developing. In addition, OpenCL is not *performance portable*. OpenCL code needs to be rewritten in order to achieve performance for each different target platform.

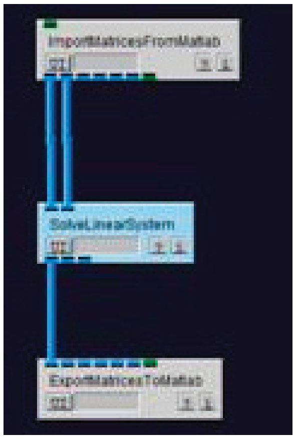
Parameterized libraries are not the only approach to achieving the stated goals. There are several attempts to compile from high level languages such as C and Fortran, including the Accelerator Compilers from the Portland Group [9]. Several researchers are developing auto-tuning approaches to generating high performance code on GPUs [5, 3]. We view these approaches as complementary to parameterized libraries. They can be used to generate the code in the library components.

Since compiling and autotuning may require long run times, the library approach gives the scientist instant access to high performance code that has been developed in advance.

We have been investigating parameterized library components for use with the SCIRun Biomedical Problem Solving Environment from the University of Utah. The goals are to keep support for GPU processing transparent to the user and to support seamless co-existence of CPU and GPU implementations by minimally perturbing both the algorithm structure and the user-interface. A similar approach can be used to extend GPU support to other modules as well as other PSEs.

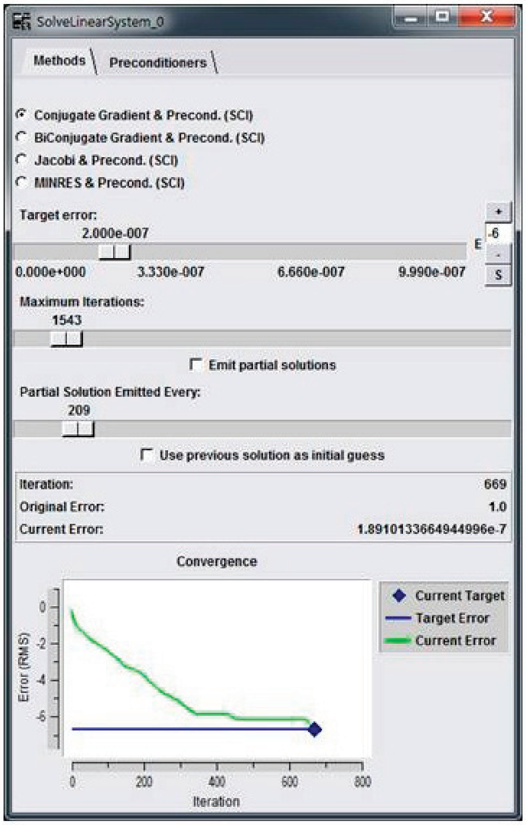
2. SCIRUN

The SCIRun software package and its extension for biomedical imaging problems, BIoPSE, is a product of the Center for Integrative Biomedical Computing (CIBC), a Biotechnology Research Resource supported by NIH since 1999. SCIRun/BioPSE is designed to be an extensible, scalable, scientific problem-solving environment [8]. SCIRun supports interaction among the modeling, computation, and visualization phases of biomedical imaging. SCIRun is supported on the dominant operating systems (Windows, Mac/OSX, and Linux). It has been used in such diverse research areas as cardiac electromechanical simulation, ECG and EEG forward and inverse calculations, modeling of deep brain stimulation, electrical impedance tomography, and determination of the electrical conductivity of anisotropic heart tissue. Between 2005 and 2010 there have been over 11,000 downloads of SCIRun from the CIBC website.



1: A simple SCIRun network

SCIRun provides a "computational workbench"; users can select software modules from a set of categories and connect them to create a network that performs all the necessary steps. An example of a very simple SCIRun network is shown in Figure 1. It consists of three steps: import data, SolveLinearSystem module, and export data. Each module has its own distinct settings adjustable by the user for the specific task at hand. The modular structure allows users, perhaps working with SCIRun developers, to create new modules for a specific task and plug them into the rest of an existing network, as well as to create "meta-modules" from parts of a network for easy re-use.


SolveLinearSystem_0

Methods | Preconditioners

● Conjugate Gradient & Precond. (SCI)
○ BiConjugate Gradient & Precond. (SCI)
○ Jacobi & Precond. (SCI)
○ MINRES & Precond. (SCI)

Target error: 2.000e-007 E -6

Maximum Iterations: 1543

□ Emit partial solutions

Partial Solution Emitted Every: 209

□ Use previous solution as initial guess

Iteration: 669
Original Error: 1.0
Current Error: 1.8910133664944996e-7

2: User Interface for SolveLinearSystems

For our case study, we focus on the linear system module in SCIRun. This module contains the Conjugate Gradient (CG), Biconjugate gradient, Minimal Residual (MinRes) and Jacobi methods. These methods are applied to large, sparse matrices. The scientist has control over the algorithm's parameters through its user interface (UI) as shown in Figure 2. Through the UI, the user specifies the algorithm, and, under another tab, the preconditioner to use, if any. In addition, the user specifies the maximum acceptable error and the maximum number of iterations to reach that error. The interface also gives visualization of the algorithm as it progresses, including the current iteration's error, minimum error and a graphical representation of the convergence. This information is useful for the scientist to decide if the algorithm is proceeding correctly and to diagnose problems when they arise. Our case study aims at accelerating the linear solvers with GPUs while preserving the user experience with SCIRun's interface. We added a checkbox to the UI to allow users to explicitly choose the GPU implementation. The linear solving algorithms are an obvious choice for GPU acceleration because they dominate many simulations' total run time and contain substantial parallelism. Such features are essential to achieve acceleration on GPU architectures. We currently target NVIDIA GPUs and use double precision floating point on GPUs for the linear solvers. Single precision introduces challenges including many more iterations to converge.

3. CASE STUDY: LINEAR SYSTEM MODULE

One approach would be to make use of the most efficient GPU implementation of each of the solvers in the Linear System Module: CG, Biconjugate gradient, MinRes. For example, at the time we began this research, the fastest CG for sparse matrices on a GPU was the Concurrent Number Cruncher (CNC) [1].

However, choosing the best point solution to a solver is not consistent with the aims of this research. First, a complete solution may not be the best solution on newer GPUs as they come out. We want to take advantage of the best solutions available, and not lock ourselves into a particular implementation that is frozen in time. For example, a new version of CNC has not been released since 2009. Second, it is difficult to integrate a complete solution into the SCIRun user interface. We make use of synchronous communication to SCIRun's environment where information about algorithm progress is displayed to the user. This requires the ability to break up the solver into smaller computations to enable the communications. Such fine grain control is often not available with a third party solution such as the CNC.

In keeping with this philosophy, we chose to keep the overall code structure of the SCIRun software, and make use of GPU acceleration of particular calculations, such as matrix-matrix and matrix-vector computations. This has the advantage of making the incorporation of GPUs into the PSE more straightforward, as well as allowing us to easily take advantage of new libraries as they become available. For example, our most recent results make use of NVIDIA's Toolkit 4.0, which includes cuSPARSE, a library of sparse matrix and vector operations developed in CUDA [7]. The calculations we use include sparse-matrix-vector multiply (SpMV) as well as many vector operations including addition, multiplication, scaling and addition, dot product and normalization.

Our experiments also show that, while a complete solver may run more quickly, it has almost no effect on the experience of the user when end-to-end run times are considered, since the bottleneck is the information communicated during the running of the algorithm.

The challenges of parallelizing the calculations on the sparse compressed row storage (CRS) format of the input data and processing with double precision accuracy make substantial increases in performance unattainable. However, reasonable speedups of an order of magnitude and significant reduction in runtimes have been realized on real applications.

4. INCORPORATING GPUS INTO SCIRUN

We have designed a reproducible and adaptable code structure to allow GPU acceleration in SCIRun that is maintainable and replaceable on a modular basis. We used the SCIRun "SolveLinearSystem" module as a case study for formulating a design that integrates well into the SCIRun environment. The algorithms in this module include Conjugate Gradient (CG), Biconjugate gradient and Minimal Residual (MinRes) each with preconditioners available. These iterative solvers account for long run times and are easily parallelized. Due to the sparse matrix format, they also represent challenges for GPU acceleration.

4.1 GPU Code Integration in SCIRun

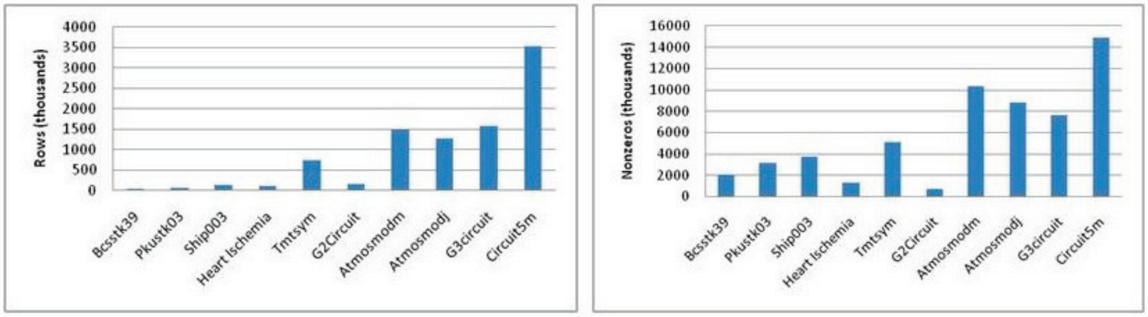
SCIRun abstracts complex calculations in its code by using a "ParallelLinearAlgebra" (PLA) class that contains low-level functions such as matrix vector multiplication and vector-vector addition. This design allows algorithms (e.g. CG) to be written in a readable manner in the module code. SCIRun code is purposefully structured in a layered, modular way to enable easy code modification and contributions by users who may not be programmers. This modular structure also allows for acceleration techniques to be performed on these functions without cluttering the algorithm. We have extended this abstraction by creating a duplicate class, "GPULinearAlgebra", that contains identical low-level functions but performs the calculations on the GPU. We have created sparse matrix vector multiply (SpMV), scale and add, subtraction and other CUDA implementations in this class. For most vector calculations that do not involve manipulation of the sparse matrix A, NVIDIA's CUBLAS library is used.

The GPULinearAlgebra (GLA) class provides a mechanism for any programmer to write their high-level mathematical algorithm and run it on the GPU with little or no GPU programming experience or knowledge. GLA is a duplicate of the ParallelLinearAlgebra class for the GPU and thus requires no code changes for a programmer familiar with the PLA class. It allows existing algorithms to be converted to run on the GPU quickly for immediate speedup. Instead of creating CPU matrices and vectors, GPU matrices and vectors are created and the functions are named and used identically to the CPU functions. Using this structure, the algorithms in the SolveLinearSystem module have been implemented targeting the GPU. For example, the CPU version of the conjugate gradient algorithm was duplicated and a simple replacement of all ParallelLinearAlgebra object types with their GPU counterparts was performed; the result is an accelerated GPU version. Note that data stay on the GPU throughout the computation and are communicated to the host at the end of the algorithm.

4.2 Data and Memory

The data used in these simulations are gathered from the University of Florida's sparse matrix collection [2] along with a heart ischemia data example provided with SCIRun. These are sparse matrices that contain millions of nonzero entries. The largest holds more than 14 million nonzero entries spread over 3 million rows. Characteristics of some of the data we tested are shown in Figure 3. These benchmarks are characteristic of the size of problems we expect users to have with SCIRun and the LinearSystemSolver module.

SCIRun's linear system solving module accepts matrices in a the compressed row storage (CRS) sparse form, a common storage technique for sparse matrix computation. CRS allows for more efficient memory usage as only non-zero entries are stored. All nonzero data elements of the matrix are stored in a single array (henceforth referred to as the matrix data array) with two reference arrays containing indexing informa-



3: Sparse matrix sizes. a) Number of rows. b) Number of non-zero elements.

tion for the elements. CRS is used for the GPU implementation, and is deemed a necessary strategy due to the small amount of memory available on most GPUs compared to CPUs. It also makes it more difficult to achieve acceleration, as several memory accesses are required to fetch each data point. One of the ways that GPUs achieve acceleration is through memory coalescing, and such coalescing is challenging to achieve when fetching sparse data. Note that other sparse formats have been experimented with for use with GPUs that achieve better coalescing than CRS. We did not consider these as they are not supported by SCIRun and would have required additional format conversion which adds substantial overhead. The CRS format was both compatible with SCIRun and enabled us to achieve speedup with large data matrices on a variety of GPUs.

4.3 User Interface

Minor modifications were made to the interface of the linear solvers module to benefit the performance of both the CPU and GPU versions of the algorithm. Most importantly, a check box was added to allow the user to select GPU or CPU. To date, we have not found any examples that failed to run or experienced worse performance using the GPU. Should input data arrive that appears to be performing poorly on the GPU, the original CPU version can be selected by the user without recompiling. If many small systems are being solved, the CPU would be a better choice for best performance. An adjustment to the default (every 20 iterations) for how often the convergence graph is updated was added to speed up both the GPU and CPU versions. It erodes performance in the GPU version particularly because, as the CPU updates the parameters and draws the graph, the GPU waits to begin the next iteration. Experiments have shown that updating the graph once per 200-500 iterations adds negligible time to either implementation, while being frequent enough for the user to observe the behavior of convergence. These, along with the increase in performance exhibited by the GPU version, are the only modifications to the user interface and experience using SCIRun, adhering to the goals of this project.

5. EXPERIMENTS AND RESULTS

5.1 Experimental Setup

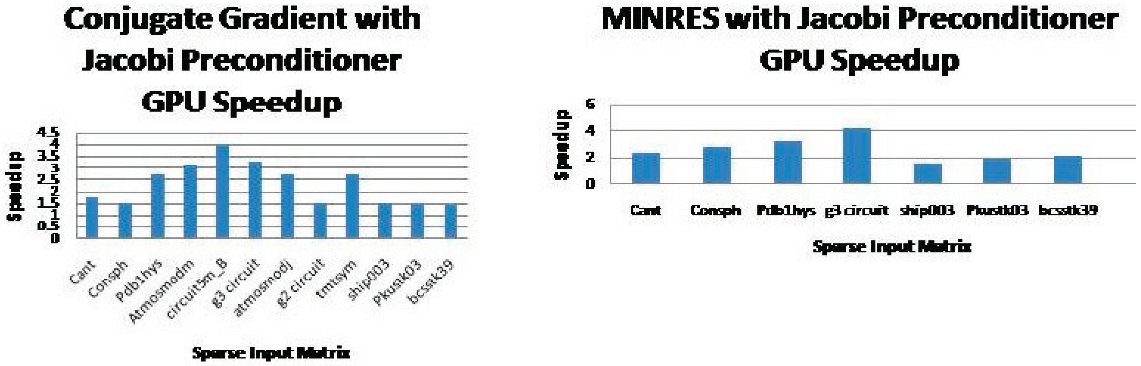
We have demonstrated the performance of our approach using double precision floating point and NVIDIA's CUDA C on a number of systems. The earlier system contained an Intel Core2 CPU running at 1.86GHz with a NVIDIA GeForce GTX 280. Our more recent results are for an Intel i7 CPU running at 3.4GHz with an NVIDIA 560Ti. Note that SCIRun automatically makes use of multiple cores. SCIRun includes platform-specific code to detect the number of processors available and spawn the same number of threads to run the algorithm. Hence our results compare multicore CPU times to GPU times. Also note that the same SCIRun GPU code was simply recompiled for the newer system. No changes were made to the source. The results presented in this section are based on the newer setup.

5.2 Results

The linear solving algorithms, including Conjugate Gradient, Biconjugate Gradient and MINRES were run on CPUs and GPUs. In all cases, a Jacobi preconditon was used. Note that the particular problem we are solving, double precision floating point computations on large sparse matrices, are particularly challenging on GPUs. Despite this, we saw speedups for all examples we ran. These results are end-to-end speedups including all data transfers to and from the GPU that are not required in the CPU implementation. We are also comparing to a multi-threaded, multicore CPU implementation. Figure 4 shows speedups of the linear solving models run on an NVIDIA 560Ti GPU compared to the same algorithm run on the Intel i7 CPU. Speedups were generally in the 1.5 to 5x range. Our best results, discussed in the next section and not shown on the graphs, were on the heart ischemia model, where speedups of 14x were realized. A speedup of 2 or 3 times, while modest, represents a noticeable improvement in the user experience. For example, CG on “g3circuit,” a matrix with 10 million non-zero entries, reduced the end-to-end run time from 46 seconds to 16 seconds. The longest running example was MINRES on “ship0003” which runs for 218 seconds on the CPU and 137 seconds on the GPU.

5.3 The Heart Ischemia Example

The heart ischemia model is provided in a package distributed with SCIRun to demonstrate how to use the problem solving environment. The data is based on observations of an ischemic heart (a heart with damaged tissue due to restricted blood flow) from a dog. The heart was scanned to render a model that makes it possible to measure and predict extracellular cardiac



4: Speedup a) Conjugate Gradient b) MINRES.

potentials. The model along with the final view of the simulation, seen in Figure 5, is represented in a three dimensional interactive image that can be evaluated for its properties. The heart ischemia model makes use of the solve linear system module for Conjugate Gradient with Jacobi preconditioner. The CG solver, which is the bottleneck of the overall processing, runs 14 times faster on the GPU compared to CPU, dropping the time from 28 seconds to 2 seconds. This is faster than the results for other sparse matrices. We attribute this to the memory layout of the data, but plan to investigate further.

6. DISCUSSION

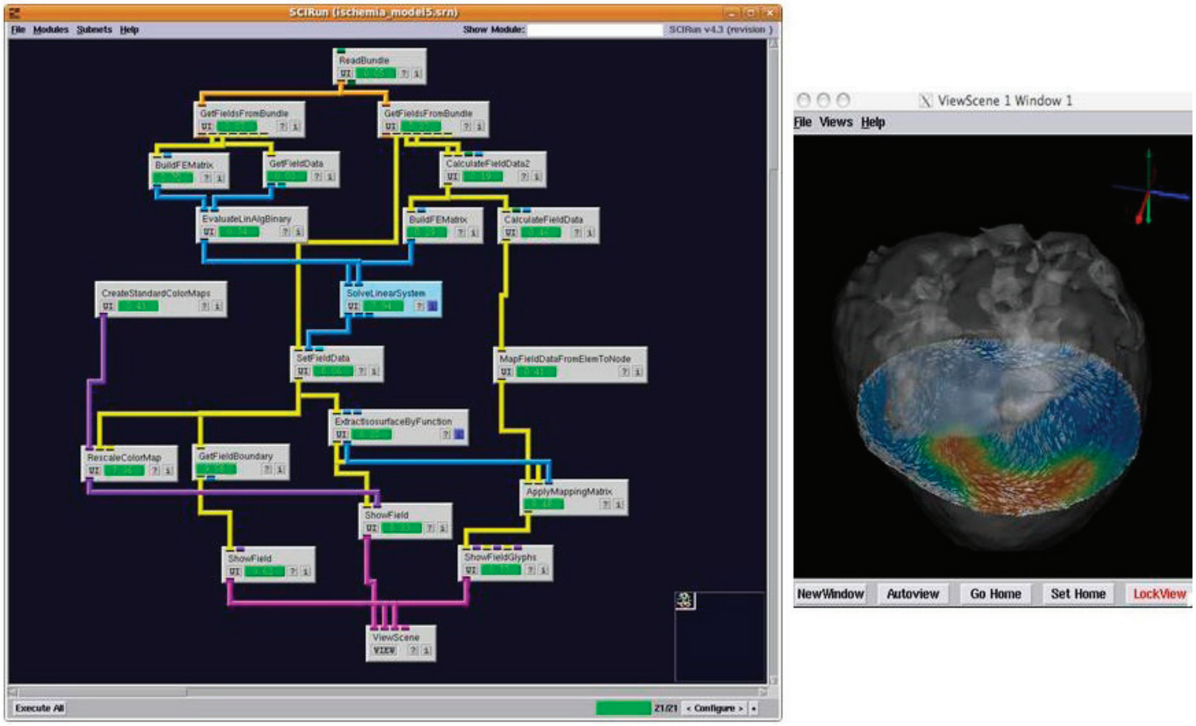
GPU speedup was achieved without modifying the algorithm design from the original SCIRun code and without optimizing the CUDA implementation for a specific problem or dataset. In order to provide the visualizations that SCIRun provides in its environment during algorithm execution, including the convergence graph and the display of the current iteration's error, some sacrifices are made. Ideally, this information could remain on the GPU and no data transfer would occur until the end of the last iteration. To remain transparent to the scientist, this information must instead be transferred from the GPU to the host memory in order to be accessible to the rest of the code. The data transferred is not large (only several single values), but the algorithm waits for a short time while the GUI performs actions on the interface, such as calculating and drawing the convergence graph. Despite the sacrifice for usability, the speedup achieved is a noticeable improvement to the user experience.

We studied the components of the CG algorithm, and saw that GPU accelerated all the main parallel vector computations in the iterative portion of the algorithm. The largest portion of time is spent in the sparse matrix vector multiply (SpMV) calculation. Although data transfer and the preconditioner individually take substantial time, they are only performed once while SpMV, subtraction, scaling addition, dot product and normalization are used multiple times in each of the hundreds or thousands of iterations of the algorithm. The dot product and normalization are the operations that transfer their result to the host for visualization purposes. An advantage of our approach is that the latest, highest performance implementations of components such as SpMV can easily be incorporated. For example, in version 4, CUDA recently included improved sparse matrix and vector routines.

One concern in using our GPU implementations with SCIRun is that our solvers make use of double precision floating point computations on GPUs. Many SCIRun users have Apple computers with older GPUs that only support single precision. This is an issue for a number of reasons. First, the solvers in the linear systems module may converge very slowly in single precision, making the GPU implementation slower than its double precision CPU counterpart. Second, they may converge to a different solution in single versus double precision. Finally, for NVIDIA GPUs with compute capability before 2.0, which includes most Apple platforms, single precision floating point operations are not IEEE compliant *by default* [10, 11]. We are still considering ways to make the GPU accelerated single precision solvers available to users. One option is to give the user a warning when the GPU only supports single precision. Another is to not provide GPU acceleration for systems that have single precision GPUs. Mathworks takes the latter approach and has recently announced increased support for Matlab for GPUs [6]. They have chosen to only support double precision and require the user's GPU to be NVIDIA compute capability 1.3 or higher. This ensures that the double precision is IEEE floating point compliant in hardware and delivers correct results as well as high performance. Note that there are other correctness issues with floating point, including obtaining different solutions when reordering computations. Being IEEE floating point compliant addresses only some issues in floating point correctness.

7. CONCLUSIONS

We present the integration of GPUs into problem solving environments in a manner completely transparent to the user. The requirements may result in choosing a solution other than the fastest GPU implementation available. Considerations other than speed, including interactions with existing code bases, data formats, and user interfaces are also important. These can be achieved while still delivering performance to the PSE



5: Heart Ischemia a) SCIRun image b) Resulting Visualization

user by making use of GPUs.

The source code for the linear solver module will be incorporated and publicly available in future versions of SCIRun.

8. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation Engineering Research Centers Innovations Program (Award Number EEC-0946463) and the NIH Center for Integrative Biomedical Computing, 2P41-RR12553-12, which develops SCIRun and supports the third author. We thank SCI Institute personnel Jeroen Stinstra, Darrell Swenson (for guidance and test problems), and especially Ayla Khan (for help integrating our work with the SCI tools). This research was conducted when Devon Yablonski was an MS student at Northeastern University.

9. REFERENCES

[1] L. Buatois, G. Caumon, and B. Lévy. Concurrent number cruncher - a GPU implementation of a general sparse linear solver. *International Journal of Parallel, Emergent and Distributed Systems*, 24:205–223, 2009.

[2] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 2011. To appear.

[3] M. Hall, J. Chame, C. Chen, et al. Loop transformation recipes for code generation and auto-tuning. In *Languages and Compilers for Parallel Computing*, volume 5898 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 2010.

[4] Khronos. The OpenCL Specification. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, June 2010.

[5] A. Klöckner, N. Pinto, Y. Lee, et al. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. http://arxiv.org/abs/0911.3456v2, March 2011.

[6] Mathworks. MATLAB GPU Computing with NVIDIA CUDA-Enabled GPUs. http://www.mathworks.com/discovery/matlab-gpu.html, 2011.

[7] NVIDIA. CUDA 4.0 Math Libraries Performance Boost. http://developer.nvidia.com/content/cuda-40-math-libraries-performance-boost, 2011.

[8] http://www.scirun.org. SCIRun: A Scientific Computing Problem Solving Environment, Scientific Computing and Imaging Institute (SCI).

[9] The Portland Group. PGI Accelerator Compilers. http://www.pgroup.com/resources/accel.htm, 2010.

[10] N. Whitehead and A. Fit-Florea. Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf, 2011.

[11] D. Yablonski. Numerical Accuracy Differences in CPU and GPGPU Codes. Master's thesis, Northeastern University, Boston MA, 2011. http://www.coe.neu.edu/Research/rcl/publications.php#theses.