Large Efficient Table-Top Teraflop Computing

Victor Basili[12], Thiago Craveiro[1], Daniela Cruzes[2], Kate Despain[1],
Bill Dorland[1], Lorin Hochstein[3] , Nico Zazworka[1], Marvin Zelkowitz[1]

1-University of Maryland College Park, Maryland
2-Fraunhofer Center Maryland College Park, Maryland
3-University of Nebraska Lincoln, Nebraska

{basili | thiago | nico | mvz}@cs.umd.edu dcruzes@fc-md.umd.edu
{bdorland | kdespain}@umd.edu lorin@cse.unl.edu

ABSTRACT

Inexpensive graphics processors are being applied to the general computational problem as a way to provide supercomputer capabilities at an affordable price. But little research has gone into understanding the difficulty in developing such programs in terms of productivity and reliability of the resulting code. The University of Maryland is undertaking a project to apply its expertise in understanding the development of high performance computer programs to this domain. In this paper we describe a project where a computational science problem is being developed and its development history captured by our set of development tools and the results are being analyzed.

Categories and Subject Descriptors

D.2.8 [**Metrics**]: Process metrics, Product metrics, Performance metrics

General Terms

Measurement, Performance, Design, Reliability, Experimentation, Languages.

Keywords

Graphical processing units, High-performance computing, Scientific computation.

1. INTRODUCTION

Around 1990, the dominant architecture for scientific computing began to transition from special-purpose vector supercomputers to massively parallel assemblies of commodity CPU's. For a period of years, the actual productivity of the median supercomputer user probably did not improve dramatically, although this is difficult to quantify, as little or no effort was expended during the transition to gather data that would allow one to quantify productivity objectively. Total theoretical peak performance (i.e., its GFLOPS (Giga- [billions of] Floating Point Operations per second) rate and simple benchmarks were used to differentiate new products, with no formal effort undertaken to measure how usable they were. Eventually, MPI emerged as the most widely used communications package for programming among these processors, and the physical networks became fast and wide enough to allow broad advances in computational thinking.

We stand today at the threshold of a new transition, from large machines composed of thousands of nodes of single-to-few, homogeneous, commodity processors, to a new technology of hundreds or thousands of nodes, each composed of heterogeneous, multicore hardware elements, with new and deeper levels of memory hierarchy. For example, a typical node will likely consist of a multicore CPU containing several processor elements, together with hundreds of coprocessors such as can be found in present-day graphics processing units (GPUs). Scientists studying complex systems with large numerical simulations will have to manage new levels of memory and communication and to develop algorithms with millions of concurrent threads rather than thousands to continue to make broad advances in computational thinking. Already, several vendors are competing in this rapidly expanding market, notably NVIDIA, IBM and AMD.

We are investigating the transition to petascale computing with commodity stream processors (*e.g.*, GPUs). Three different communities are involved in this work. (1) A team of scientists working in a particular application area which relies heavily on large-scale numerical simulations (astrophysical turbulence) will move from using the largest parallel supercomputers available today to using new clusters of heterogeneous, multicore nodes (including using GPUs as coprocessors) for leading-edge scientific studies. (2) Applied mathematicians will accelerate this scientific transition by developing scientific middleware packages for GPUs for a collection of widely-used algorithms (including fast multipole, particle-in-cell, discontinuous Galerkin, and pseudo-spectral solvers). (3) A software engineering team with deep experience in measuring total productivity will assess the success of this total effort, providing objective information about programmability, maintainability, portability and raw performance.

GP-GPU Computing. Recently there has been dramatic progress in heterogeneous multicore computing within the GP-GPU[1] paradigm. Examples include computers built around the IBM Cell Broadband Engine, and clusters whose nodes contain multicore CPU's together with one or more GPUs, where the latter are used as special-purpose compute engines. The IBM Cell BE is the graphics chip for the Sony Playstation 3; as a vector-parallel multi-processor, it is capable of running scientific calculations at a sustained rate of a 200-300 GFLOPS.

NVIDIA's GeForce 8800 GTX GPU is a PCI Expressed-based 2D/3D graphics card on which we have achieved comparable performance on scientific problems. Compared to CPU performance, which doubles every 18 months, GPU performance over the last few years has doubled every 6 months. This has occurred as GPUs moved to multithreaded, multiprocessor devices which are designed specifically for data-parallel problems. It is broadly understood that as compute speed consequently greatly outstrips memory access speeds, scientific programmers will be forced to rethink their choice of algorithms and their implementation of parallelism.

[1] General Purpose computation on Graphical Processing Units

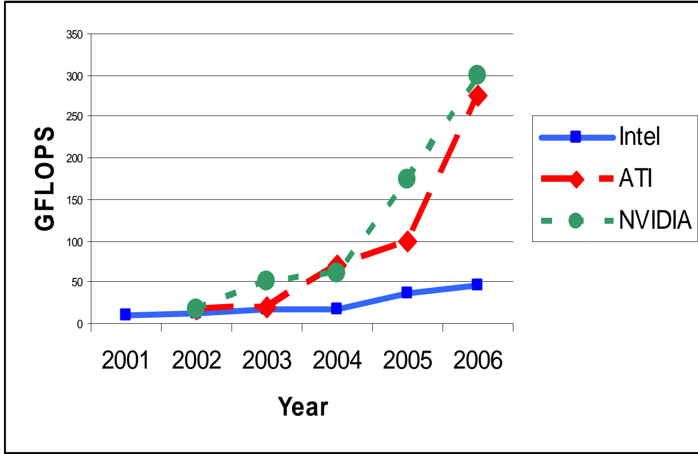

Fig. 1. Over a period of years, GPU performance has improved more rapidly than CPU performance.

In Section 2 of this paper we briefly describe the physics application we are studying, and in Section 3 we summarize the high performance computing development environment we have developed for studying "traditional" high performance computing domains. We will apply this environment to the GP-GPU development problem.

2. PARALLEL ALGORITHMS ON GPUS

Here we briefly summarize the application domain we will study. The goal is simply to provide context for our research and not necessarily to give the full details of high energy physics research.

Plasma physics. Magnetohydrodynamic (MHD) turbulence is encountered in a wide variety of space and astrophysical plasmas, including the solar wind, accretion disks around black holes, and the plasma between stars in galaxies (the interstellar medium). Such turbulence often plays a crucial role in the evolution of astrophysical systems by redistributing heat and momentum, and by influencing the observed radiation via the heating of plasma and the acceleration of highly energetic particles. An understanding of the properties of magnetized plasma turbulence is thus a key problem in space physics and astrophysics.

In MHD theory, all the particles are assumed to move together. At the large scales at which turbulence is driven in most space and astrophysical environments, this is a good approximation. However, at small scales, where turbulent energy is converted to heat, this foundational assumption of MHD theory breaks down. Kinetic theory is required to describe how different particles are affected by the small-scale, electromagnetic fluctuations. We require a theory that simultaneously describes the larger scale electromagnetic turbulent fluctuations in a manner that is consistent with MHD theory and also the self-consistent fluctuations and particle motions at smaller scales, where the various plasma components respond very differently from one another.

This problem is computationally challenging because of the need to solve for both the *real space* dynamics of the turbulence and the *velocity space* dynamics of the plasma. Because petascale level of calculations are needed to solve this problem, a GP-GPU solution allows for an inexpensive route to providing the necessary hardware to achieve this solution.

Parallel algorithms. We have developed a gyrokinetic simulation code for today's terascale computers. We are porting this code (AstroGK) to a CPU-GPU cluster with libraries and algorithms that will scale efficiently to petaflop performance. On present terascale computers, we typically use $O(10^5$-$10^6)$ real-space points and $O(10^3)$ velocity-space points to resolve the kinetic distribution functions.

Programming GPUs. Moving from an MPI-based, conventional parallelization on a large supercomputer to the "relentlessly multithreaded" environment of GPU computing is roughly as complicated as was the move from vector supercomputers to massively parallel supercomputers, and presents a number of challenges to the programmer. NVidia provides a programming interface called CUDA [5], which exposes a SIMD-style data-parallel programming model. CUDA allows programmers to execute performance-critical portions of their code, called *kernels,* on a graphics card containing GPUs. The interface consists of a runtime library and a modest extension of the C programming language, which allows the programmer to specify which functions run on the *device* (i.e., the graphics card) and which variables are located on the *device,* through the use of type qualifiers.

While a GPU may support thousands of threads, it is not possible for all these threads to communicate directly with each other through fast shared memory because of limited memory in each GPU. Instead, only batches of threads called *blocks* are capable of this type fast communication. CUDA requires the programmer to divide up data arrays to be processed efficiently by such blocks.

A further challenge is the lack of maturity of existing development tools. No GPU-specific debuggers exist, and the programmer cannot even invoke print statements in functions that run on the GPU. If a program executing on the GPU crashes, the programmer does not receive any error message: the only hint of such a crash is an execution time much shorter than expected. As the programmer is expecting dramatic reductions in execution time, this incorrect behavior may initially go unnoticed.

Finally, while individual GPUs hold great performance in increasing program performance relative to conventional workstations, if they are to be competitive with the performance modern supercomputers it will require multiple GPUs executing in parallel, most likely in combination with MPI.

This then is the context of this research. We believe that the CPU-GPU environment provides a hardware environment that should allow for inexpensive petascale computing. However, the difficulty of developing such code and the need to measure the ultimate productivity of the development team should help answer the question of whether this approach provides an effective platform for solving complex scientific computations that require massively large computational resources.

3. PRODUCTIVITY MEASUREMENT

We have been involved since 2004 in understanding productivity in the supercomputing domain as part of the DARPA High Productivity Computing Systems (HPCS) program, where DARPA is developing a new class of petascale machines. Because of the similarity of most supercomputing resources and the GP-GPU environment described

previously, we believe that we can apply our tools and knowledge gained from the HPCS program in this new environment in order to address the issues mentioned at the end of the previous section.

The HPCS project has goals of "providing a new generation of economically viable high productivity computing systems for national security and for the industrial user community," and initiating "a fundamental reassessment of how we define and measure performance, programmability, portability, robustness and ultimately, productivity in the HPC domain"[2]. In order to reassess the definitions and measures in a scientific domain it was necessary to study the basis and source of those definitions and measures. However the large amount of tacit information that is merely in people's minds often remains neglected.

As a way to understand these differences, we developed a set of tools and protocols to study programmer productivity in the HPC community. Our concern had been to understand the effort involved and defects made in developing such programs. We also want to develop models of workflows that accurately explain the process that HPC programmers use to build their codes. Issues such as time involved in developing serial and parallel versions of a program, testing and debugging of the code, optimizing the code for a specific parallelization model (e.g., MPI, OpenMP) and tuning for a specific machine architecture are all topics of study. If we have those models, we can then work on the more crucial problems of what tools and techniques better optimize a programmer's performance to produce quality code more efficiently.

We have conducted human-subject experiments at various locations across the U.S. in graduate level HPC courses and with interviews with professional programmer at HPC centers (Fig. 2). Multiple students are routinely given the same assignment to perform, and we conducted experiments to control for the skills of specific programmers (e.g., experimental meta-analysis) in different environments. Due to the relatively low costs, student studies are an excellent environment to debug protocols that might be later used on practicing HPC programmers. An early result needed to validate our process was to verify that students could indeed produce good HPC codes and that we could measure their increased performance.

Table 1 is one set of data that shows that students achieved speedups of approximately 3 to 7 on an 8-processor HPC machine. (CxAy means class number x, assignment number y.) In general, on an n-processor machine, if a program executes on a single processor in k seconds, you would like to be able to execute the program in k/n seconds; hence a speedup of n. However, in practice, speedups of n are approached but rarely achieved since algorithms cannot be parallelized so precisely. Paradoxically, some programmers fail to achieve a speedup of even 1, meaning a program runs faster on a single processor than on a multiprocessing system.

As an example of the research we conducted, we measured productivity in the HPC domain as part of understanding HPC workflows; however, what does productivity mean in this domain [3]? The following is one model that we can derive from the fact that the critical component of HPC programs is the speedup achieved by using a multiprocessor HPC machine over a single processor [6].

Data set	Programming Model	Speedup on 8 processors
Speedup measured relative to serial version:		
C1A1	MPI	mean 4.74, sd 1.97, n=2
C3A3	MPI	mean 2.8, sd 1.9, n=3
C3A3	OpenMP	mean 6.7, sd 9.1, n=2
Speedup measured relative to parallel version run on 1 processor:		
C0A1	MPI	mean 5.0, sd 2.1, n=13
C1A1	MPI	mean 4.8, sd 2.0, n=3
C3A3	MPI	mean 5.6, sd 2.5, n=5
C3A3	OpenMP	mean 5.7, sd 3.0, n=4

Table 1: Mean, standard deviation, and number of subjects for computing speedup on Game of Life program.

In manufacturing productivity is relatively easy to compute. For a pencil manufacturer, counting the pencils produced per day divided by the cost of running the factory gives an easy way to compute productivity. For software, computing the source lines of code (SLOC) written per day gives an approximation for productivity, but has never been a satisfactory measure. In the HPC world, SLOC/day is even less relevant. A serial version of the program does solve the problem, so what "work" is accomplished in getting the program to run on a parallel machine?

Productivity in the HPC domain is often defined as the relative speedup of a program using an HPC machine compared to a single processor divided by the relative effort to produce the HPC version of the program divided by the effort to produce a single processor version of the program:

$$\text{Speedup} = \frac{\text{Reference Execution Time}}{\text{Parallel Execution Time}}$$

$$\text{Productivity} = \frac{\text{Relative Speedup}}{\text{Relative Effort}}$$

$$\text{Relative Effort} = \frac{\text{Parallel Effort}}{\text{Reference Effort}}$$

This computation divides high speedup (a desired trait) by the cost of creating the parallel version (an undesirable trait) and thus tries to balance the benefits of faster execution with the drawbacks of increased development time.

Table 2 presents an example of productivity calculations. In order to normalize the data, we used a standard implementation as our reference implementation in which to compare all implementations. In Table 2 we chose program 2, which has a complete data set and had the least serial execution time. Program 5 did execute faster, but that group did not report the total effort to develop the serial code, so we would be unable to compute a productivity measure for it. In this example, the group who wrote program 3 was the most productive. While its execution time of 12.8 sec. was not the fastest, its total effort of 10 hours was by far the least.

[2] http://www.highproductivity.org

Program →	1	2*	3	4	5
Serial effort (hrs)	3	7	5	15	
Total effort (hrs)	16	29	10	34.5	22
Serial Exec (sec)	123.2	75.2	101.5	80.1	31.1
Parallel Exec (sec)	47.7	15.8	12.8	11.2	8.5
Speedup	1.58	4.76	5.87	6.71	8.90
Relative Effort	2.29	4.14	1.43	4.93	3.14
Productivity	0.69	1.15	4.11	1.36	2.83
*- Reference serial implementation					

Table 2. Productivity experiment: Game of Life

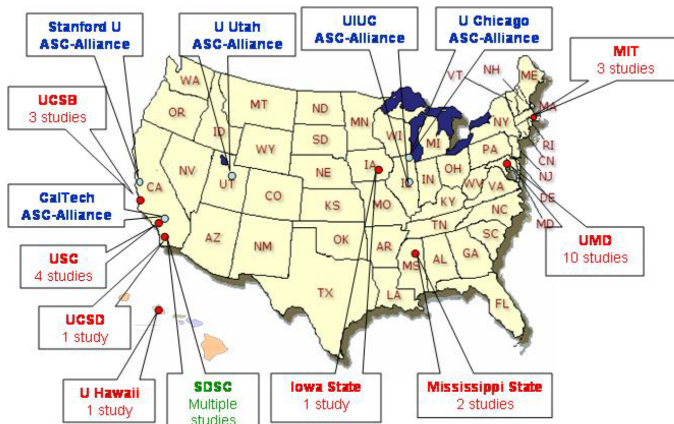

Fig.2. Studies conducted.

3.1 Experimental environment

As part of this research we have developed a series of tools to study the HPC domain. This set of tools includes:

1. *Websites* for knowledge derivation and dissemination
   - http://HPCBugBase.org – A defect database where developers can enter defects found in their code and learn about common problems found in HPC development..
   - http://hpcs.cs.umd.edu – HPCS Development Time website. This is the main website for understanding development time issues for the HPC domain. All of the tools mentioned below are accessible via this website. A general overview of the tool collection is given in Fig. 3.
2. *Data capture* for seamlessly and unobtrusively collecting development data from programmers.
   - UMD instrumentation (UMDINST)– Shell-level timestamps from compilation and execution. This is implemented as a wrapper added to the user's login shell.
   - Experiment Manager – Tools to collect self-reported effort data.
   - Shell logger – Capture all shell commands automatically without the user's involvement.
3. *Data process* for converting the data for use by the analysis programs.
   - Raw data importer – Import UMDINST data to database
   - DB Sanitizer – Remove privacy data from database. This is a critical issue when dealing with student data from universities. It can also be a major problem with proprietary corporate and government data as well.
4. *Visualization and analysis* for understanding the data.

– Automatic Performance Measuring System – Automatically runs scripts of programs to recheck the results of previously submitted programs. This allows us to easily check performance of each program and to correlate it with the development data we collected.

– CodeVizard – View source code evolution. This is somewhat live a visualized form of the *diff* program, but provides additional information. (See Figures 5 and 6.)

– Data Analyzer – Visualization of UMDINST and Experiment Manager data

– Activity graph – View workflow information

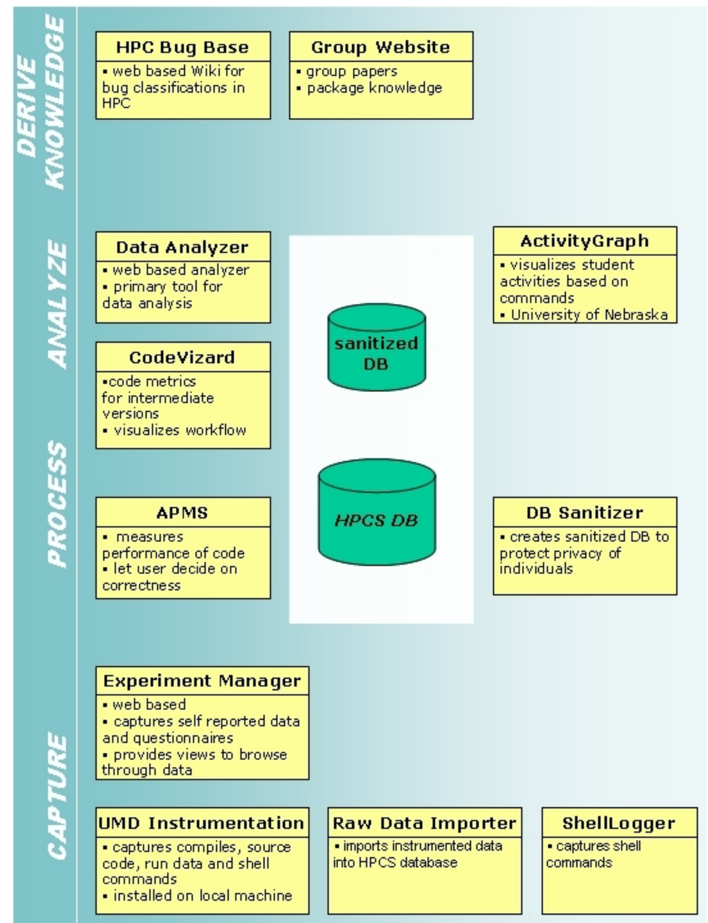

Fig. 3. Structure of HPC development and analysis tools.

In addition, we also use the following tools developed by others:

- Hackystat – Low-level timestamps for many tools. Hackystat was development at the University of Hawaii [4].
- UCSB execution harness – Execute programs under controlled conditions

Our research model is described in Fig. 4. as the interaction of the preceding set of tools. See references [1] and [2].

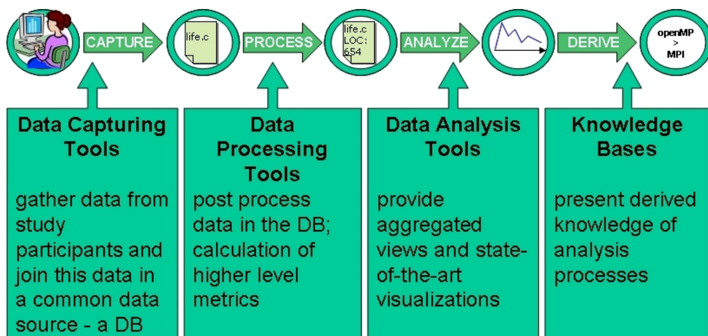

Fig. 4. Research Model.

3.2 Preliminary GPU studies

To answer questions about productivity using the GPU programming model we are currently undertaking studies with the professionals and students in the physics domain described earlier (Table 3). The majority of them have a background in physics (only one of them in computer science) and some worked with parallel languages before, but none of them with GPUs.

Physics	4
Plasma Physics	1
Energy Science	1
Astrophysics	1
Applied Math and Scientific Computation	1
Physics and Astronomy	1
Physics and Math	1
Computer Science	1
Graduate students	5
Undergraduate students	2
Research scientists	4

Table 3. Study participants.

Our initial setup focuses on a broad spectrum of research questions:

- **Effort**: Understand how much effort is required to achieve sufficient performance (speedup) on a GPU for a specific problem or class of problems.
- **Learning curve:** How much time does it take to train novices in the new programming models? What kind of background experience or approach is beneficial?
- **Defects:** What are commonly made mistakes by novices and experts? How do you avoid them? How do these defects compare with defects made in MPI?
- **Workflow:** What is the best way to achieve a correct fast working GPU code as quickly as possible?

In contrast to the single programmer problems that we have been studying in controlled classroom experiments in the HPC domain, we are now applying our toolset to several active development processes with multiple programmers. We capture, visualize, and analyze using following data probes:

- **Program compilation data**: During each compile we capture timestamps of the compile information if the compile has been successful or if it failed any information about the compiled sources including the source code.
- **Program run data**: Each program executed including its timestamp and shell-level command used to run the program is captured.
- **Shell commands**: All shell commands entered during the development are captured. Calls like the invocation of *make*, the performance measurement with *time* and *gprof* give clues to the programmer's current activity.
- **Background Questionnaire**: In the beginning of the study we hand out general questionnaires that provides us with general information about each individual subject. Questions include information about their background and experience in software development and parallel computing. We also asked for their personal goals and reasons to learn this new technology.
- **Daily Diary**: We asked the users to provide us with daily reports of their programming activities.
- **Interviews**: In some cases we invited programmers for interviews to describe their current work and latest experiences.

As an example, we give a sample analysis of code developed by a GPU novice. The problem the subject addressed was a port of an existing component in FORTRAN to a GPU. To understand the process the reader must know that the current programming model implemented for an NIVDIA GPU (i.e., CUDA) is based on the C language. Therefore the subject was forced to either rewrite his code in C or to use an additional third party library to perform GPU calls from the FORTRAN code. He decided for the latter since such a library was already developed at the University of Maryland.

It took the subject six days to finish the task. Surprisingly our analysis shows that most of the time (the first three days) he was dealing with issues of understanding the library calls and getting an already existing FORTRAN template code to run on his machine. After overcoming these issues it took him only three hours on day four to make the necessary changes to his code, with some additional time on the last two days to measure performance. (See Figs. 5 and 6) Our analysis results were later verified by interviewing the participant.

The NVIDIA GPU has its own high-speed shared memory, instruction set, controllers and several processors (with associated small local memory). The graphics processor can be considered as a compute device that is capable of efficiently executing data parallel computations, where the same program is executed on many data elements in parallel. The 8800GTX processor we use has 128 stream processors that are able to access 768 MB of onboard DRAM. The processors are arranged as 16 independent multiprocessors that are composed of 8 processor units, which share 16 kB of local shared memory. Programs executing on the stream processors are called threads and can access the device's DRAM and on-chip memory through the following read-write memory: local 32-bit registers per processor, and a parallel data cache; read-only constant cache that is shared by all the processors; and a read-only texture cache that is shared by all the processors and speeds up reads from the texture memory space. To achieve high performance, one must understand the limitations and capabilities of this architecture.

Our prototype GPU study is the Particle in Cell (PIC) problem. This involves numerical simulation of particles with Coulomb forces and their interaction (movement) in space. Space is divided in cells or boxes. Different abstraction levels and parameters are used for different physics with the same basic algorithm. The common goals are to port existing code to GPU, build new GPU code from scratch, create PIC library to implement code for own physics need as fast as possible (with best performance achievable).

Since the fall of 2007 we have been collecting experience forms from each developer, daily effort forms on activities worked on, and semi-structured interviews. Our experiment manager collects snapshots of all code developed each time the compiler is invoked.

Although similar to HPC programming, GPU programming has its own characteristics. We are developing a separate bug base to reflect the unique characteristics of GPU programming.

4. CONCLUSIONS

At present our GPU work is only beginning, but we believe there are great similarities between HPC programming using traditional supercomputers and this new generation of graphics processors. After we complete our initial “shakedown” test of the PIC port, we intend to tackle the larger MHD problem given in Section 2 of this paper. This work is still preliminary, but we think it will lead to relevant insights in the months ahead.

5. ACKNOWLEDGEMENTS

This research was supported in part by Air Force grant FA8750-05-1-0100 to the University of Maryland. We’d also like to acknowledge the following faculty for allowing us to conduct experiments in their classes: Alan Edelman [MIT], John Gilbert [UCSB], Mary Hall, Aiichiro Nakano, Jackie Chame [USC], Allan Snavely [UCSD], Jeff Hollingsworth, Alan Sussman, Uzi Vishkin, [UMD], Ed Luke [MSU], Henri Casanova [UH], and Glenn Luecke [ISU].

6. REFERENCES

[1] L. Hochstein, T. Nakamura, V. R. Basili, S. Asgari, M. V. Zelkowitz, J. K. Hollingsworth, F. Shull, J. Carver, M. Voelp, N. Zazworka, P. Johnson, Experiments to Understand HPC Time to Development, CTWatch, November 2006.

[2] L. Hochstein, T. Nakamura, F. Shull, N. Zazworka, V. R. Basili, M. V. Zelkowitz, An Environment for Conducting Families of Software Engineering Experiments, Advances in Computers 74, Elsevier, 2008.

[3] The International Journal of High Performance Computing Applications, (18)4, Winter 2004.

[4] P. M. Johnson, H. Kou, J. Agustin, Q. Zhang, A. Kagawa, T. Yamashita, Practical Automated Process and Product Metric Collection and Analysis in a Classroom Setting: Lessons Learned from Hackystat-UH. International Symposium on EMp. Soft. Eng., Los Angeles, 2004, 136-144

[5] NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 1.10. November 29, 2007.

[6] Zelkowitz M., V. Basili, S. Asgari, L. Hochstein, J. Hollingsworth, and T. Nakamura, Productivity measures for high performance computers, Computer Society International Symposium on Software Metrics, Como, Italy, September, 2005.

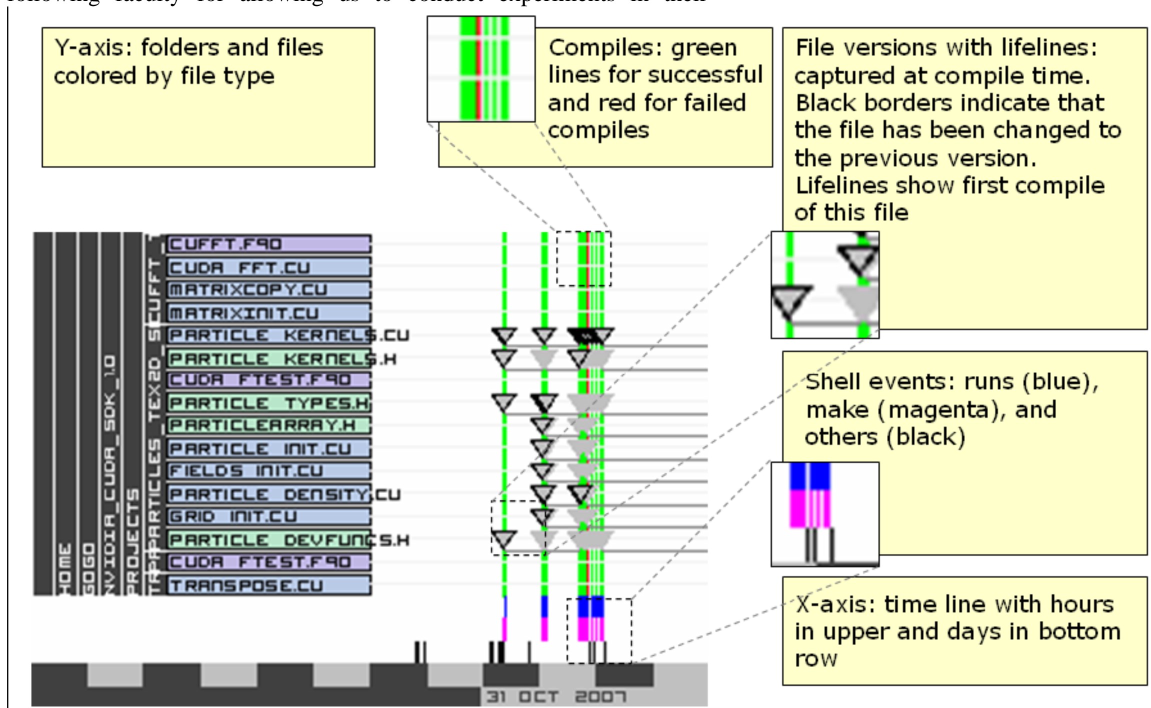

Fig. 5. CodeVizard output. Each vertical line is a compile, and each mark on line represents a component touched by that compile.

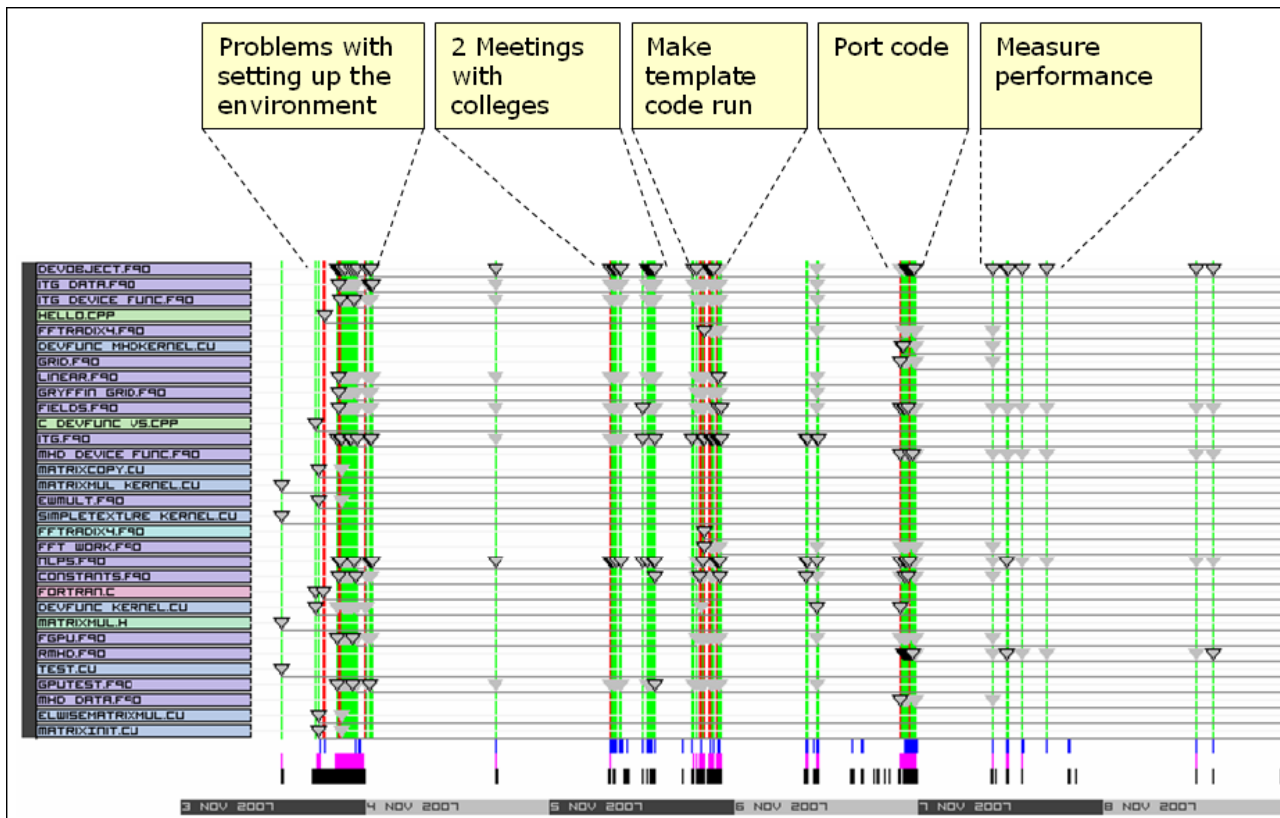

Fig. 6. Six days of development needed to port component. (Each day is represented by alternating gray and black bands on the X-axis.)