Scout: A Hardware-Accelerated System for Quantitatively Driven Visualization and Analysis

Patrick S. McCormick Jeff Inman James P. Ahrens *
Advanced Computing Lab
Los Alamos National Laboratory

Charles Hansen Greg Roth †
Scientific Computing and Imaging Institute
The University of Utah

ABSTRACT

Quantitative techniques for visualization are critical to the successful analysis of both acquired and simulated scientific data. Many visualization techniques rely on indirect mappings, such as transfer functions, to produce the final imagery. In many situations, it is preferable and more powerful to express these mappings as mathematical expressions, or queries, that can then be directly applied to the data. In this paper, we present a hardware-accelerated system that provides such capabilities and exploits current graphics hardware for portions of the computational tasks that would otherwise be executed on the CPU. In our approach, the direct programming of the graphics processor using a concise data parallel language, gives scientists the capability to efficiently explore and visualize data sets.

CR Categories: I.3.6 [Computing Methodologies]: Computer Graphics—Methodology and Techniques

Keywords: Visualization systems, hardware acceleration, multivariate visualization, volume rendering.

1 INTRODUCTION

The visualization and analysis process involves the investigation of relationships between the numerical and spatial properties of one or more data sets. Many different visualization processes use indirect mappings, such as transfer functions, to assign optical properties like color and transparency to data values. While these techniques can be powerful, they have had limited acceptance in the scientific community because they require scientists to work in a secondary data space (e.g. the transfer function domain). This often makes it difficult to efficiently express queries and mathematical operations that would be natural in the original data space. As the success of software packages like MATLAB and Mathematica have shown, it is beneficial to have the ability to directly perform mathematical operations when exploring and analyzing data. Although this approach is effective and expressive, due to the computational costs involved in evaluating mathematical expressions, it often fails to provide users with an interactive experience. The power and programmability of today's commodity graphics hardware provides a unique opportunity to help reduce the impact of these calculations.

We have developed Scout – a software system that provides expression based queries which are evaluated in the data space. In this case, we consider a query to be a set of relational and conditional expressions based on numerical values. This system reduces the computational bottleneck by utilizing the graphics processor (GPU) not only for rendering, but also as a computational co-processor responsible for offloading portions of the work that would traditionally be executed on the CPU. The language-based interface allows domain scientists to process multivariate data, express derived data, and define the associated mappings to the final image in a more familiar environment than many graphical user interfaces provide. A key feature of our approach has been to keep the interface to the GPU as simple and expressive as possible, while at the same time revealing the underlying parallel capabilities of the hardware. This has been done by using a data-parallel syntax that reflects the underlying single-instruction-multiple-data (SIMD) characteristics of vertex and fragment programs on the GPU.

In addition to reducing the workload on the CPU, Scout also benefits from the computational performance rates of the GPU. In our experiments, the graphics hardware routinely out-performs the main processor. This performance gain is due to the parallel, streaming architecture, and the high-speed local memory systems of the card. The GPU architecture also allows for derived values to be computed as part of the streaming process and thus avoids the need to allocate system memory for intermediate storage. Finally, by storing data sets on the graphics card we can amortize the cost of transferring data across the system bus, which would generally be a requirement if the CPU were responsible for computing derived values. It is important to note that this benefit often increases the amount of data that must be stored on the GPU.

In the following section, we discuss related efforts and present a brief review of the latest GPU related research activities. Section 3 provides an overview of the features and design of the Scout software and programming language. Section 4 presents the results of applying Scout to different visualization tasks. Finally, Section 5 discusses conclusions, improvements, and future research directions.

2 RELATED WORK

2.1 Expression Evaluation for Visualization

The calculator paradigm of deriving data from equations is utilized by the MATLAB and Mathematica systems, allowing scientists to compute and plot the results of various mathematical expressions [17, 26]. These packages provide excellent insight into formal mathematical models, but are inefficient for processing large data sets, and thus not typically used. Data-flow systems such as OpenDX and SCIRun provide calculator interfaces through their data-flow modules [1, 10, 20, 22]. These systems compute derived field data, but employ the CPU and memory of the host system. For large fields, the computation process of the derived field is non-interactive.

Moran and Henze describe a system based on the calculator paradigm, the *Demand Driven Visualizer (DDV)*, for computation and visualization of fields derived from large datasets [18]. This approach employs both sparse traversal and lazy evaluation to avoid computation at each grid location, thereby running faster than eager

*e-mail:pat@lanl.gov, jti@lanl.gov, ahrens@lanl.gov
†e-mail:hansen@cs.utah.edu, roth@cs.utah.edu

IEEE Visualization 2004
October 10-15, Austin, Texas, USA
0-7803-8788-0/04/$20.00 ©2004 IEEE

evaluation systems. This work is similar to Scout since both systems are based on the application of visualization to derived fields which are expressed by mathematical equations. The focus of DDV is to expedite the computation of derived fields by using lazy evaluation. Since this system runs entirely on general purpose processors, the result is a fast but non-interactive visualization system.

Jankun-Kelly and Ma describe a spreadsheet-like system for data exploration [9]. This system provides an interface through a scripting language for performing operations on data. As such, one can view this system as providing a calculator functionality that executes on the main CPU. Interactivity timings were not provided, but using the CPU without sparse traversal and lazy evaluation was shown to be non-interactive by Moran and Henze [18].

2.2 Graphics Hardware

Since the 1990s the power of commodity graphics hardware has seen incredible growth. This has been realized in terms of performance, programmability, and increased arithmetic precision – all at an amazingly low price. Even though these capabilities have been primarily driven by the entertainment industry, many active research efforts are leveraging GPUs for advanced rendering, visualization, and general purpose computation.

As several efforts have shown, the streaming architectures of the latest graphics cards from ATI [2] and NVIDIA [19], are capable of outperforming CPUs on specific computational tasks [8]. In particular, the GPU has been used for several general scientific computing tasks. For example, Krüeger and Westermann, Goodnight et al., and Bolz, et al. have implemented numerical solvers on the GPU [3, 7, 13]. In addition, Lefohn et al. presented an interactive sparse-grid level set solver [14]. The majority of these projects have focused on the acceleration of particular algorithms and have usually been implemented in low-level (assembly) language or with one of the higher-level graphics shading languages such as NVIDIA's Cg [16].

The Brook programming language allows users to program the GPU using a data streaming paradigm [4]. Under this model, a new data type, referred to as a *stream*, is used to represent a collection of data which can be operated on in parallel. These streams of data are then operated on by *kernels* which are functions that execute a specific set of instructions on stream data. In comparison to the work described above, Scout and Brook allow the programmability of the GPU to be exposed directly to the user through a data parallel paradigm. Given the underlying streaming architecture of the GPU, Scout shares similarities with Brook. For example, data sets in Scout are equivalent to Brook's streams, and a Scout program is a kernel with a result that maps data values to colors. Unlike Brook, Scout is based upon a more traditional data parallel paradigm and provides an interactive development environment for the visualization and exploration of data.

3 Implementation

Current graphics processors must be programmed directly using a graphics-specific API, such as OpenGL or DirectX. The clear disadvantage of this approach is that all operations must be expressed in terms of graphics primitives. In this section we present the details of Scout's underlying OpenGL-based architecture, the Scout programming language, and the details of the runtime environment.

3.1 Data Model and Rendering

Scout represents all imported data sets as either one-, two-, or three-dimensional OpenGL texture maps having from one to four channels. The main disadvantage of this texture-based representation is that only regular grid structures are directly supported. One-dimensional textures are most commonly used as lookup tables, while two and three-dimensional textures normally represent scientific data. Multi-channeled textures are most useful for representing vector fields or for packing multiple variables into a single texture. Data sets may be stored as bytes, shorts, integers, or floats. The particular data type and the number of channels in a texture may have an impact on how the data must be represented within a texture, and may also affect performance. For example, basic OpenGL floating point textures (`GL_FLOAT`) must have values in the range $[0, 1]$ and a four-channel texture may align better in memory than a three-channel texture, giving increased performance. The impact of normalized texture values is discussed further in Section 3.2.

After one or more data sets have been stored in texture memory, Scout allows users to write a program that maps data values to a final rendered image. The source code from this program is compiled to either a single fragment shader or a single vertex and fragment program pair. The Scout compiler currently generates low-level code that corresponds to the OpenGL ARB extensions `GL_VERTEX_PROGRAM_ARB` and `GL_FRAGMENT_PROGRAM_ARB` [15]. Once these programs are downloaded to the GPU, geometry is rendered according to the types of data sets that are stored in texture memory. For two-dimensional data, a single quadrilateral (matching the given data set dimensions) is rendered. In the case of three-dimensional data, view-aligned slices are rendered that implement texture based, direct volume rendering [5, 25].

3.2 The Scout Programming Language

In designing Scout we had several goals for the programming language:

1. It should be simple and concise. The primary motivation for this was to make the language easy to learn and use. In addition, we wanted to assure that compilation times would be fast enough to guarantee interactive performance for the user.
2. It should reveal the parallel nature of the underlying hardware without complicating the language. This should be true for both the SIMD and the 4-vector arithmetic parallelism available on the GPU.
3. Where possible, the language should hide any nuances introduced by the OpenGL API or the graphics hardware.
4. It should provide the user with flexible methods for producing both general purpose computations and visualization results.

Of these goals, the most challenging are hiding the details of the graphics API and the specific hardware limitations of the GPU. Contemporary GPUs have many constraints; these include restrictions on the number of instructions a program may contain, the number of textures that may be active, the number of local temporary registers that can be used, and the number of parameters that may be passed in via the OpenGL interface. In addition, OpenGL (1.5) stores textures internally within the hardware as floating point values in the range $[0-1]$. This forces texture data to be normalized, and when used with scientific data can lead to the loss of important information. Recent OpenGL extensions have eliminated this restriction, allowing textures to be stored in an unnormalized 32-bit floating point format. Because current hardware does not

support the blending of these 32-bit values in the frame buffer, they cannot be used when implementing direct volume rendering. When possible, Scout uses 16-bit float point values and blending for three-dimensional texture based volume rendering. [1]

Fortunately, the first two goals for the Scout language are easier to achieve. The instruction set of the GPU is small, and recently added capabilities for flow control (e.g., loops, true branches, and subroutine calls) have not yet been implemented in Scout; therefore, only simple language structures are currently supported. Without considering instruction-level optimizations (e.g., scheduling) and multi-pass rendering, this limited functionality makes compilation relatively straightforward – provided that the language closely models the architecture of the hardware. Furthermore, because the assembly-level instructions of the GPU already use a SIMD and vector parallel paradigm, the task of supporting a data parallel language is greatly simplified. Finally, the constraints that vertex and fragment programs receive inputs via registers to which they cannot write, and produce outputs from which they cannot read, can easily be matched to a limited functional language. Given these advantages we have loosely based the Scout language on the *C** Programming Language [23]. This results in a language structure that is simple and can easily be applied to many areas within the scientific and visualization communities.

All Scout programs achieve their effect by assigning values to pixels in the output image. This assignment is made to the predefined 4-component vector variable, `image`, representing the output (RGBA) register of the fragment engine. The simplest Scout program is:

```
image = c;
```

where `c` is a scalar constant. When assigning a scalar value to a vector value, the scalar value is simply replicated across all four components of the vector. In the case of the variable `image`, the channels are all clamped to the range $[0,1]$. Assigning a floating point value in this range results in a gray-scale image.

The Scout language supports scalar and 4-component vector types, arithmetic operators (`+`, `-`, `*`, `/`), relational operators (`<`, `<=`, `==`, `=>`, `>`, `!=`), and logical operators (`&&`, `||`, `!`), matching the functionality of the equivalent operators in the C programming language. In addition, Scout provides built-in functions for all operators available on the fragment or vertex engines, such as `sin`, `cos`, `pow`, dot-product, cross-product, and a few additional functions that are expanded in-line by the compiler, such as `hsva`, `norm`, `global_min`, and `global_max`.

In situations where 16-bit or 32-bit floating point textures are not available, or when data has been quantized into a non-floating point type, the Scout compiler automatically includes instructions that de-normalize texture values into the range of the original data. In the remainder of this section we will introduce the Scout language by presenting a series of small programs that operate on a two-dimensional data set produced by the POP ocean model [6].

When working with data produced by POP it is common to use a land mask to represent locations where model results have not been computed. This mask contains 1 at locations where land is present and 0 for regions of water. The example presented in Figure 1 introduces this mask (`land`), potential temperature at the ocean surface (`pt`), and a one-dimensional array of RGB colors (`colormap`). We also make use of the Scout functions, `norm`, to normalize values from any dataset into the range $[0,1]$, and `positionsof`, to return the number of elements (colors) in the color map. The simple program shown in Figure 1 displays black for land values and mapped color values for ocean temperatures.

```
// Display color mapped potential ocean surface temperature.
where (land == 1)
  image = 0; // Render land as black.
else
  // Map temperature values into color look-up table.
  image = colormap[positionsof(colormap) * norm(pt)];
```

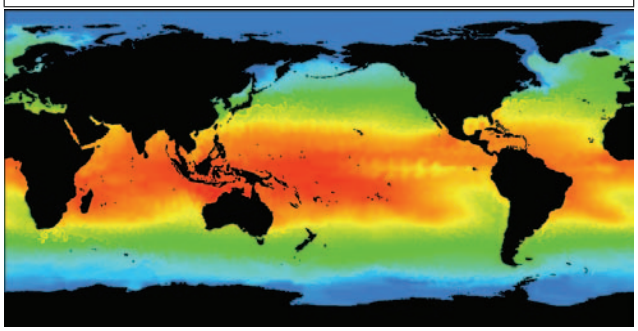

Figure 1: Sample Scout code and the resulting image showing the color mapped potential temperatures and black landmasses.

Input data sets (textures) can be treated as either arrays or as scalars. When treated as scalars, they are implicitly accessed with the texture-coordinates of the current output pixel. This scalar syntax is natural for data-parallel computations, whereas array syntax is useful for referencing neighbors or values in a pre-computed lookup table.

Scout preserves the ability of the GPU to treat floating point values as indices (texture coordinates) of a data array. When such an index value falls between existing array elements, the hardware performs an interpolation between the neighboring values based on the relative position of the index. However, in contrast with assembly-level GPU programming, Scout array indices are based on the dimension of the given variable (data set). This policy matches the syntax used by many other languages for multi-dimensional array operations, and thus makes the language more familiar to scientists. The array-indices of nearest neighbors are simply ± 1 in a given dimension, relative to the current data-parallel position of the pixel. The indices of this location are represented by the predefined variables `i`, `j`, and `k`. We also arrange, where possible, for array index expressions to be computed on the vertex engine. This is often more efficient, because the values need only be computed at the vertices and can then be generated via hardware-supported interpolation at all other points.

In the next example we add a conditional expression that isolates a specific range of temperature values. The sample code uses the `hsva()` function to produce an RGBA color described in terms of the HSV color space with the addition of an alpha channel. In comparison to the RGB color space, the continuous nature of the HSV space makes it more suitable for mapping colors to a range of data values. This approach allows users to define a color mapping dynamically, and thus potentially use many different mappings for different regions in the data. An `rgba()` function is also available, allowing users to define the output color as an additive combination of color channels. Figure 2 shows both the code and the resulting image.

The display can easily be tailored with successive conditions to either emphasize or de-emphasize features, or to present multiple variables from one or more data sets. Furthermore, we can use spa-

[1] NVIDIA's NV4X series of hardware currently supports this feature [19].

```
where (land == 1)
  image = 0; // render land as black.
// select data range(s) of interest and render as color.
else where (pt < 2.375 || (pt >= 21.0 && pt < 29.5))
  image = hsva(240 - (norm(pt) * 240), 1,1,1);// blue to red
else
  image = 0.6; // outside data range colored by gray pixels.
```

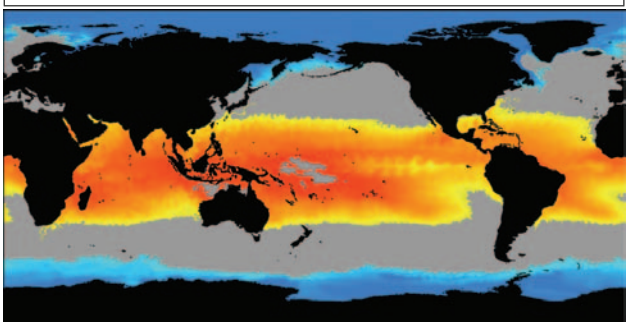

Figure 2: Sample source and the resulting image showing the selected ranges of temperature values.

tial criteria to define regions of interest. Figure 3, illustrates the use of mathematical expressions to define a circular clipping region, with a radius of 600 grid units, centered at the point $(2400, 1000)$. The interior of this circle is rendered with temperature values using the same color mapping applied in Figure 1. The region outside of the circle is colored by a function that displays blue decreasing from full intensity to zero intensity, west to east, and green decreasing in an identical fashion from south to north. The built-in symbols `i` and `j` refer to the x and y coordinates of the current computational grid cell. The example code also introduces a function that returns the dimension of a variable along a given axis index, `dimof`.

The final component of the Scout implementation is the runtime layer that provides users with an interactive environment for the development of code and the exploration of data.

3.3 Runtime Environment

The Scout runtime environment consists of a user interface containing a code editor and a rendering window. The user interface provides support for invoking the Scout compiler in addition to a mouse-based interface for navigation (rotation, translation, and zooming) within the rendered data. A diagnostic window provides feedback when warnings or errors are detected in the source code or when hardware limits have been exceeded. The compilation of a Scout program results in the generation of an assembly level fragment program, and possibly an accompanying vertex program. These two programs are interdependent and are treated as a unit.

The assembly program(s) generated by the compiler depend on the creation of a OpenGL environment by the runtime system. For example, texture lookups depend upon variables being assigned to compiler specified texture-units on the GPU. These references are currently implemented by associating the compiled Scout program with a set of dependencies. The dependencies identify data sets used by the program and values for support of operations, such as normalization, queries of data-range, dimension-size, etc. When a compiled program is being executed, the runtime system first resolves all of the associated dependencies and then binds data to the proper texture units and loads local variables into the proper registers. Once these steps are complete, the associated geometry is generated and sent to the GPU for execution.

```
// Compute the distance from our location (i,j) to the center
// of the circle clip region at (2400, 1000).
float radius = sqrt(pow(abs(2400-i),2) + pow(abs(1000-j),2));
where (land == 1)
  image = 0; // Render land as black.
else where (radius < 600) // Color by pt within the circle.
  image = colormap[positionsof(colormap) * norm(pt)];
else
  // Color by spatial location. dimof() returns the dimension
  // of pt along the given axis index (0: x axis, 1: y axis).
  image = rgba(0, i/dimof(pt, 0), j/dimof(pt, 1), 1);
```

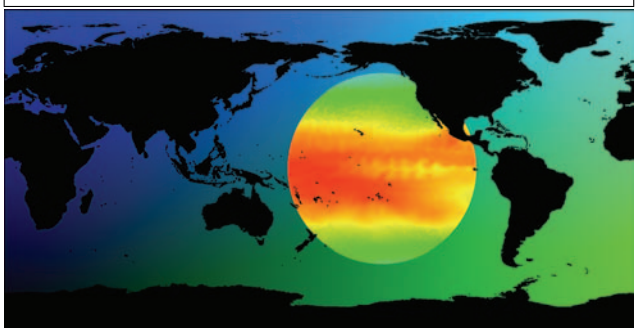

Figure 3: Scout code and rendered image showing temperature within a circular clipping region and spatially-controlled coloring outside of the region.

4 Results

The development of Scout has been motivated by the need to solve advanced visualization and analysis problems in various application areas. In this section we present examples of this, with a particular focus on multi-dimensional transfer functions, multivariate visualization, and the derivation of data using the Scout language.

4.1 Multi-dimensional Transfer Functions

Volume rendering applications commonly encounter data sets in which it is difficult to distinguish between regions due to overlapping boundaries. Multi-dimensional transfer functions are commonly used to address this issue, as demonstrated by Kniss et al. using the Simian volume renderer [11, 12]. The flexibility of Scout's programming language allows the implementation of multi-dimensional transfer functions based on many different variables (limited only by the number of textures supported on the GPU).

To demonstrate the use of multi-dimensional transfer functions in Scout we have duplicated one of the results produced by the triangular widget used in Simian. Using Simian, it is possible to set the transfer function by probing the data and placing widgets interactively with keystrokes. While this is a powerful exploratory technique, these widgets do not support the use of direct quantitative bounds. We have implemented the triangular transfer function in Scout by using the density and the gradient magnitude of the computed tomography generated, tooth data set that was also explored by Kniss et al. [11]. The domain of the transfer function is defined by the range of density values along the horizontal axis, U, and gradient magnitude values along the vertical axis, V. The implemented triangle function is an isosceles triangle positioned within the domain such that it's base is parallel to the U axis and the apex is pointed downward. The edges of this triangle are defined by the half angle at the apex θ, the minimum and maximum allowed gradient magnitudes G_{min} and G_{max}, and the density center point C of

the triangle.

```
// Compute the half width of the triangle.
width = tan(theta) * gm;
// Compute and clamp the distance from the
// center to density.
where (d >= (C - width) && d < (C + width))
  dist = abs(d - C);
else
  dist = width;
// Scale alpha value by distance from the center.
where (gm > G_min && gm < G_max)
  alpha = 1 - dist / width;
else
  alpha = 0.0;
```

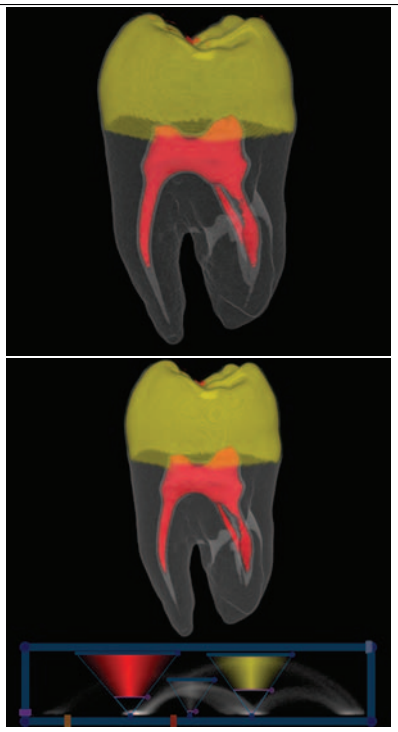

Figure 4: Volume rendered results using triangular transfer functions from Scout (top) and Simian (bottom). Both images are rendered with diffuse lighting. This was implemented directly in Scout, but removed from the sample code for clarity.

To quantitatively calculate the triangle function for a given input value, we first calculate half the width of the triangle at the gradient magnitude gm using θ. The distance from the center, to the current density value d is calculated and clamped to $width$. If gm falls outside of the limits G_{min} and G_{max} alpha is assigned zero, otherwise, the alpha value is scaled according to the distance from the center tapering off to zero as it approaches the edges of the triangle. The Scout code presented in Figure 4 shows the basic details of this implementation. For comparison, the figure shows the results produced by both Simian and Scout using triangular functions.

In Scout the elliptical widget can be succinctly described for a 2D transfer function similar to the 2D spatial clipping circle presented in section 3.2. While this approach is not as flexible for exploring unknown data, the ability to directly control the transfer function using quantitative expressions provides a versatile, and often more accurate interface for scientists.

4.2 Multivariate Visualization

When analyzing computational modeling results, it is often valuable for scientists to study the relationships between several different variables. This can be especially true when exploring anomalies and temporal data. In this section we investigate the multivariate data produced by a simulation of an of an El Niño-Southern Oscillation (ENSO) event generated by the POP ocean model [6].

During normal Pacific Ocean conditions, the trade winds blow from east to west across the tropics. These winds cause warm surface water to pile up in the western portion of the Tropical Pacific. In the east, the water that has been pushed to the west is replaced by the upwelling of cooler water. The ocean state can therefore be summarized as increased sea surface height, increased potential temperature, and an increased thermocline depth in the western Pacific relative to the east. The thermocline is a sharp temperature gradient separating the upper layers of ocean water from those at depth. Under El Niño conditions, the east-west trade winds relax over the central and western Pacific. This results in a decreased sea surface height and an elevated thermocline in the west, and an increased sea surface height and a depressed thermocline in the east.

In order to explore the details of a simulated ENSO event, we have extracted a $414 \times 128 \times 17$ region from a global ocean simulation. The visualization of ocean simulations is often difficult because the horizontal extent of the data is much greater than the depth. To make it easier to see details in the data, the data is resampled along the z axis to produce a $414 \times 128 \times 64$ data set. Using Scout we visualize several key features that show the changes that occurred during the simulation of the 1998 El Niño. Figure 5 shows the results.

The Figure presents land masses as tan colored regions rendered with diffuse lighting; with the western Pacific on the left-hand side of the image. The first condition that we are interested in studying is the difference between the potential ocean temperature during the January of the ENSO event and the mean of all January conditions from the model run. The rendered results show the regions where this difference exceeds 2.5° C as solid magenta (in the east), and locations where the difference is less than -2.5° C as solid blue (in the west). Both regions are clipped as they pass south of the equator. This result shows how the increased depth of the thermocline in the east causes an overall warming in the surface waters, while the shoaling of the thermocline in the west leads to cooler water temperatures.

Figure 5 also includes the position of the thermocline near the equator. The white strip of data shown in the rendered image represents the position of the El Niño thermocline over the temperature range 19.0° C – 21.0° C; which is traditionally used as a proxy for the thermocline in the tropics. The thermocline for the January mean is also shown for the same temperature range; this surface is colored by the difference in sea surface height between the ENSO and the mean data. The differences are color mapped from blue (low) to red (high). Note the relationship between the two different thermoclines: the white, ENSO, surface is higher in the west and lower in the east. This shows, along with the differences in sea surface heights, the see-saw pattern that is consistent with the occurrence of an El Niño.

4.3 Computing Derived Fields

In the process of analyzing data it is often necessary for scientists to compute one or more derived values. This section demonstrates the computation of derived variables using Scout.

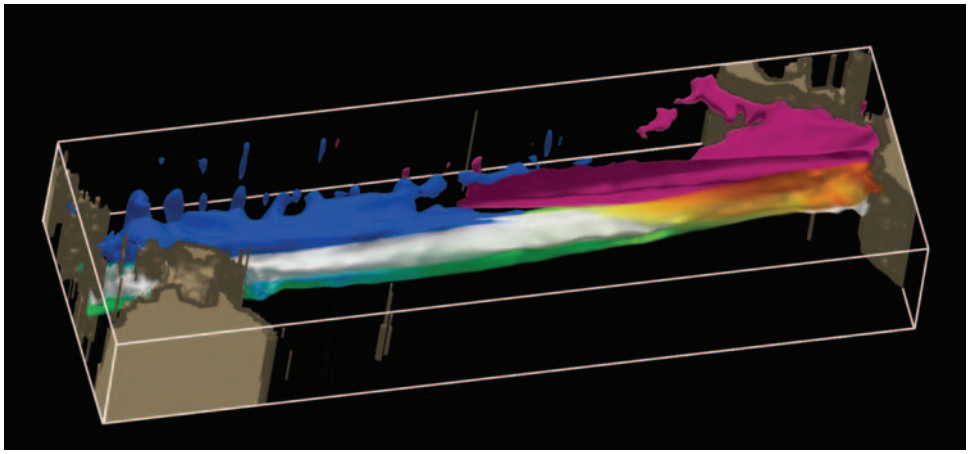

Figure 5: The result of studying several features that are consistent with an El Niño event. Anomalies in water temperature are shown as solid magenta and blue and are clipped south of the equator. The white region represents the ENSO thermocline and the color mapped region shows the mean January thermocline; the colors show the difference in sea surface heights. Land masses are rendered in tan. All variables have been rendered with diffuse lighting.

The Terascale Supernova Initiative (TSI) project is currently studying the mechanisms responsible for driving core collapse supernovae [24]. A core collapse supernovae occurs when a iron core forms in an aging super-giant star. This core is a result of several sequences of fusion reactions which come to a halt once the iron is produced. In this state, the core no longer supports these reactions, and internal gravity pulls inward on the core until it collapses into an incredibly dense region that will eventually become a neutron star. The core becomes so dense that the collapse stops, and a shock wave is sent outward through the gas layers that are rushing in to the fill the void left from the collapse. The expanding shock wave from the core, and the inward collapse of the outer gases result in the explosion of the star.

The goal of the computation presented below is to model the behavior of the core during the time in which the shock wave is expanding. The simulation is a hydrodynamics-only calculation in which the shock wave is modeled after it has moved approximately 200 km from the center of the star. The simulation is computed on a $320 \times 320 \times 320$ grid and produces the following variables: pressure (p), density (ρ), and velocity ($\vec{v}$). From these variables we can define two additional variables of interest: entropy (e) and tangential velocity (v_t). Entropy is given by the equation, $e = p/\rho^{4/3}$.

Tangential velocity is computed as follows. Let $\vec{r}$ represent the radius vector from the center point C of the grid to a given location on the grid. Then for each point P on the computational grid:

$$\vec{r} = (P_x - C_x)\mathbf{i} + (P_y - C_y)\mathbf{j} + (P_z - C_z)\mathbf{k}. \quad (1)$$

Next we compute the square of the radial velocity (v_r):

$$v_r^2 = \frac{\vec{v} \cdot \vec{r}}{\vec{r} \cdot \vec{r}}. \quad (2)$$

This then allows us to compute the tangential velocity:

$$v_t = \sqrt{\vec{v} \cdot \vec{v} - v_r^2}. \quad (3)$$

Using these derived fields we can begin to explore the results produced by the simulation.

Due to the size of the full floating point data sets, it is not possible to fit two variables in the texture memory of most graphics cards, which are commonly limited to 256 MB. This is perhaps the most limiting factor to our approach of leveraging the GPU. In order to compute entropy and tangential velocity we have resampled the data to $256 \times 256 \times 256$. Figure 6 shows the computed entropy. The image was produced by a Scout program that computes entropy, selects two different ranges of entropy using a `where` statement, and finally colors them by the magnitude of the velocity field. The first entropy range is between 0.070 and 0.076 and represents high entropy values. This region is partially clipped away to reveal the structure of the turbulent flow in the interior of the core. The second range (0.01 to 0.04) represents lower entropy values and isolates the details of the shock front.

The computation of v_t gives insight into the structure and behavior of the flow created by the shock wave within the core. Figure 7 shows the results produced by computing v_t and volume rendering the results. The transparent white region represents low entropy values, and is displayed to show the shock front as a frame of reference. The shell is clipped away by two arbitrary clipping planes that were computed using the plane equation. The code for this example follows directly from the equations given above and is not presented.

4.4 GPU Versus CPU

The ability to derive data values as part of the rendering process can reduce the amount of required system memory, and eliminate the travel of recomputed data between the CPU and GPU. It is important to note that these benefits often come at the price of requiring that more texture memory be used on the GPU. As noted earlier, this is a clearly a major disadvantage of using the GPU. In order to complete the comparison between the CPU and GPU this section reviews the performance of the computations carried out in Section 4.3.

Table 1 presents the execution times for the computation of entropy, magnitude of the velocity field, and the tangential velocity. The table compares two different versions of the CPU code. The first version is compiled using GNU GCC 3.3.3 with full optimization (-O3) and the second uses version 8.0 of the Intel C++ compiler

```
// Compute entropy.
float ent = pressure / pow(density, 4.0/3.0);
// Compute |v|.
float vmag = sqrt(dot3(v, v));
where(i > 115 && ent > 0.070 && ent < 0.076)
  image = hsva(...);
else where(ent > 0.01 && ent < 0.04) // shock wave
  image = hsva(...);
else
  image = 0; // transparent black
```

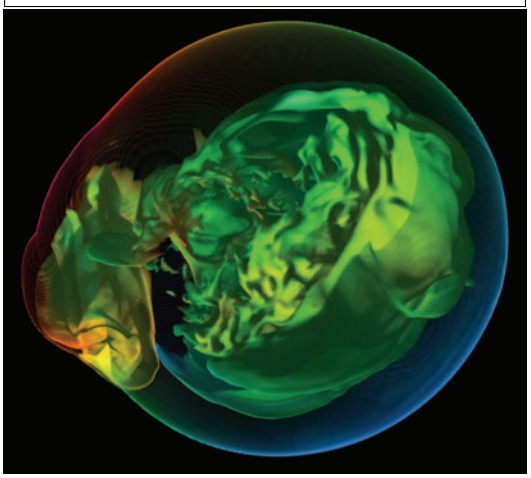

Figure 6: Volume rendered results of two selected entropy ranges colored by corresponding velocity magnitude. Both the entropy and velocity magnitude were computed directly using Scout.

which is capable of producing code that uses the Streaming SIMD Extensions (SSE). The use of the SSE instructions can result in dramatic performance improvements. The CPU timing results are measured using the hardware counters available on the main processor. The benchmarks were run on a 3.0 GHz Intel Xeon EM64T processor with 4 GB RAM and an NVIDIA Quadro 3400 (connected via PCI Express).

The GPU benchmarks are measured by instrumenting Scout with the CPU's hardware counters and then rendering the same image multiple times and taking the average of the resulting execution times. In order to study the performance of the GPU directly, the cost of the first rendered frame is discarded to avoid including the costs of transferring textures and other data to the GPU. It is important to note that this does not present an accurate overall view of Scout's performance but is done intentionally to measure performance characteristics of the graphics hardware.

Code	CPU	CPU w/ SSE	Quadro FX 3400 (NV45)
entropy	3.3	0.42 (7.9×)	0.12 (27.5×)
$\lvert\overrightarrow{v}\rvert$	0.27	0.19 (1.4×)	0.10 (2.7×)
v_t	0.62	0.40 (1.55×)	0.17 (3.65×)

Table 1: Table 1: CPU vs. GPU computation times in seconds. All times reflect computation times only, overhead costs are not included. Values in parentheses show the speed up achieved in relation to the times presented in the CPU column.

All Scout programs incur an overhead cost associated with the process of compiling code, downloading textures, downloading the compiled fragment and vertex programs, sending the necessary geometry to the GPU, and other tasks carried out by the runtime system. Of these operations texture download rates clearly dominate the overhead execution time; the other operations require only a few microseconds each to complete. Large 32-bit floating point textures can be downloaded at approximately 200-300 MB/second. Our benchmarks show that a large data set, such as the TSI velocity field, takes approximately 1 second to download to the graphics card. It is possible to download textures at a faster rate (700-800 MB/second) over PCI Express – these rates require the use of byte-based texture data and therefore were not benchmarked. Overall, the GPU can achieve very fast computational rates, but in order to outperform the CPU it is necessary to amortize the cost of the transfer of data between main memory and the graphics card.

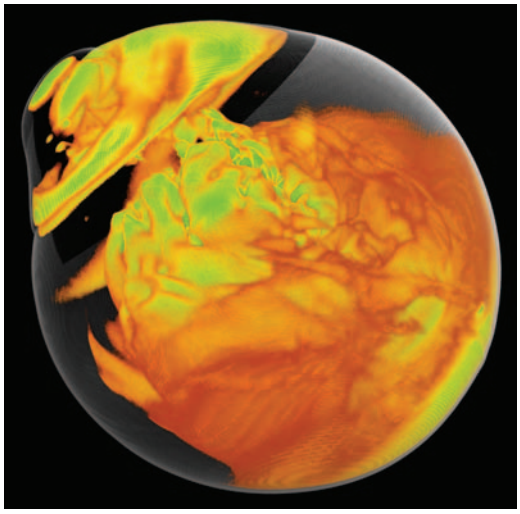

Figure 7: A volume rendered representation of tangential velocity. The shell of low entropy regions is shown as transparent white to emphasize the shock front as a frame of reference.

5 Conclusions and Future Work

This paper has introduced Scout, a hardware-accelerated software system for the visualization and analysis of data using quantitative, query based expressions. This approach allows scientists to work directly in the original data space using a direct programming methodology. By leveraging the GPU as a co-processor we have allowed for interactive response rates, reduced main memory costs, and, in several situations, decreased the amount of data transfered over the system bus.

Although Scout allows the use of quantitative expressions to control the visualization process, it does not yet provide the user with a complete set of general purpose operations or direct quantitative feedback. Supporting these features is critical to providing a complete analysis environment. The task of providing more advanced language features on the GPU can be challenging. In particular, the limits of the GPU (number of supported instructions, number of textures, and number of temporary variables) require the determination and management of multiple rendering passes. We are collaborating with UC Davis to incorporate their multi-pass partitioning technology to limit the impact of these restrictions [21].

Although Scout provides interactive performance, there are many data sets that easily exceed the capabilities of a single GPU. To address this challenge we are exploring the use of Scout in a parallel, cluster-based, environment. In addition, there are likely to be many

situations where a GPU-based co-processor may not be the most advantageous approach. In this situation we are investigating the use of other processor technologies to help improve the data visualization and analysis process.

6 Acknowledgments

This work was sponsored by the Mathematical, Information, and Computational Sciences Program which is part of the DOE Office of Science. A very special thanks to John Blondin and Tony Mezzacappa of the TSI project, and JoAnn Lysne, Matthew Hecht and Mat Maltrud from the COSIM project. Without their help, lively discussions, and enthusiasm, this paper would have been impossible. Thanks to Nick Triantos from NVIDIA and Mark Segal at ATI for their technical help as well as donating hardware to our effort. Gordon Kindlmann's help and advice as well as his *nnrd* software were used for data manipulation. Finally, thanks to the reviewers for their comments and suggestions. This paper has been released under LA-UR-04-2373.

References

[1] G. Abram and L. Treinish. An extended data-flow architecture for data analysis and visualization. In *In Proceedings of Visualization '95*, pages 263–270. IEEE CS, October 1995.

[2] ATI. ATI Technologies Inc. http://www.ati.com, 2004.

[3] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. *ACM Transactions on Graphics*, 22(3):917–924, July 2003.

[4] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics – SIGGRAPH 2004, to appear*, 2004.

[5] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *ACM Symposium On Volume Visualization*, 1994.

[6] J. K. Dukowicz, R. D. Smith, and R. C. Malone. A reformulation and implementation of the Bryan-Cox-Semtner ocean model on the Connection Machine. *J. Atmos. Ocean. Tech.*, 10:195–208, 1993.

[7] Nolan Goodnight, Cliff Woolley, Gregory Lewin, David Luebke, and Greg Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *Graphics Hardware 2003*, pages 102–111, July 2003.

[8] Mark Harris. General-Purpose Computation Using Graphics Hardware. http://www.gpgpu.org, 2004.

[9] T.J. Jankun-Kelly and Kwan-Liu Ma. Visualization exploration and encapsulation via a spreadsheet-like interface. *IEEE Transactions on Visualization and Computer Graphics*, 7(3):275–287, 2001.

[10] C. Johnson, S. Parker, and D. Weinstein. Large-scale computational science applications using the SCIRun problem solving environment. In *In Proceedings of Supercomputing 2000*, pages 263–270. IEEE CS, November 2000.

[11] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *Preceedings of IEEE Visualization 2001*, pages 255–262, 2001.

[12] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.

[13] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3):908–916, July 2003.

[14] A.E. Lefohn, J.M. Kniss, C.D. Hansen, and R. Whitaker. Interactive deformation and visualization of level set surfaces using graphics hardware. In *IEEE Visualization 2003*, pages 75–82, October 2003.

[15] Eric Lengyel. *The OpenGL Extensions Guide*. Charles River Media, Inc., 2003.

[16] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, 22(3):896–907, July 2003.

[17] Mathworks. The Mathworks - Products - MATLAB. http://www.mathworks.com/products/matlab/, 2004.

[18] P. Moran and C. Henze. Large data visualization with demand-driven calculation. In *In Proceedings of Visualization '99*, pages 27–33. IEEE CS, October 1999.

[19] NVIDIA. NVIDIA Home. http://www.nvidia.com, 2004.

[20] OpenDX. Open Visualization Data Explorer. http://www.opendx.org, 2004.

[21] Andrew Riffel, Aaron E. Lefohn, Kiril Vidimce, Mark Leone, and John D. Owens. Mio: Fast multipass partitioning via priority-based instruction scheduling. In *Graphics Hardware 2004*, August 2004.

[22] SCIRun. A Scientific Computing Problem Solving Environment. Scientific Computing and Imaging Institute (SCI), http://software.sci.utah.edu/scirun.html, 2004.

[23] Thinking Machines Corporation. *C* User's Guide*, June 1991.

[24] TSI. Terascale Supernova Initiative. http://www.phy.ornl.gov/tsi/, 2004.

[25] Orion Wilson, Allen Van Gelder, and Jane Wilhelms. Direct Volume Rendering via 3D Textures. Technical Report UCSC-CRL-94-19, University of California at Santa Cruz, June 1994.

[26] Wolfram Research. Mathematica: The Way the World Calculates. http://www.wolfram.com, 2004.