LA-UR-07-2094

Approved for public release;
distribution is unlimited.

Title:

Scout: A Data Parallel Programming Language for Graphics Processors

Author(s):

P. McCormick, J. Inman, J. Ahrens, J. Mohd-Yusof, G. Roth, S. Cummings

Intended for:

Parallel Computing, 33(10-11), pgs. 648-662, 2007

Available online at www.sciencedirect.com

ScienceDirect

Parallel Computing 33 (2007) 648–662

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Scout: a data-parallel programming language for graphics processors

Patrick McCormick [a,*], Jeff Inman [a], James Ahrens [a], Jamaludin Mohd-Yusof [a], Greg Roth [b], Sharen Cummins [a]

[a] *Computer, Computational, and Statistical Sciences Division, Los Alamos National Laboratory, United States*
[b] *Computer Science Department, The University of Utah, United States*

Received 28 March 2007; received in revised form 10 September 2007; accepted 21 September 2007
Available online 29 September 2007

Abstract

Commodity graphics hardware has seen incredible growth in terms of performance, programmability, and arithmetic precision. Even though these trends have been primarily driven by the entertainment industry, the price-to-performance ratio of graphics processors (GPUs) has attracted the attention of many within the high-performance computing community. While the performance of the GPU is well suited for computational science, the programming interface, and several hardware limitations, have prevented their wide adoption. In this paper we present Scout, a data-parallel programming language for graphics processors that hides the nuances of both the underlying hardware and supporting graphics software layers. In addition to general-purpose programming constructs, the language provides extensions for scientific visualization operations that support the exploration of existing or computed data sets.
Published by Elsevier B.V.

Keywords: Graphics processors; Data-parallel programming; Heterogeneous computing; Visualization

1. Introduction

Driven by their wide use within the entertainment industry, GPUs are likely the most cost-effective, high-performance, processors available today. This is primarily due to the highly parallel nature of the graphics pipeline that has allowed the GPU to devote more transistors to arithmetic operations. In comparison, recent general-purpose processors have focused on improving the performance of sequential code by dedicating transistors to features such as branch prediction and out-of-order execution. Fig. 1 provides a summary of the performance trends and current prices of GPUs and CPUs as of August 2007. This section provides a brief review of the parallel nature of the graphics hardware pipeline, discusses the GPU programming model, and presents an overview of Scout.

[*] Corresponding author. Tel.: +1 505 665 0201.
E-mail address: pat@lanl.gov (P. McCormick).

0167-8191/$ - see front matter Published by Elsevier B.V.
doi:10.1016/j.parco.2007.09.001

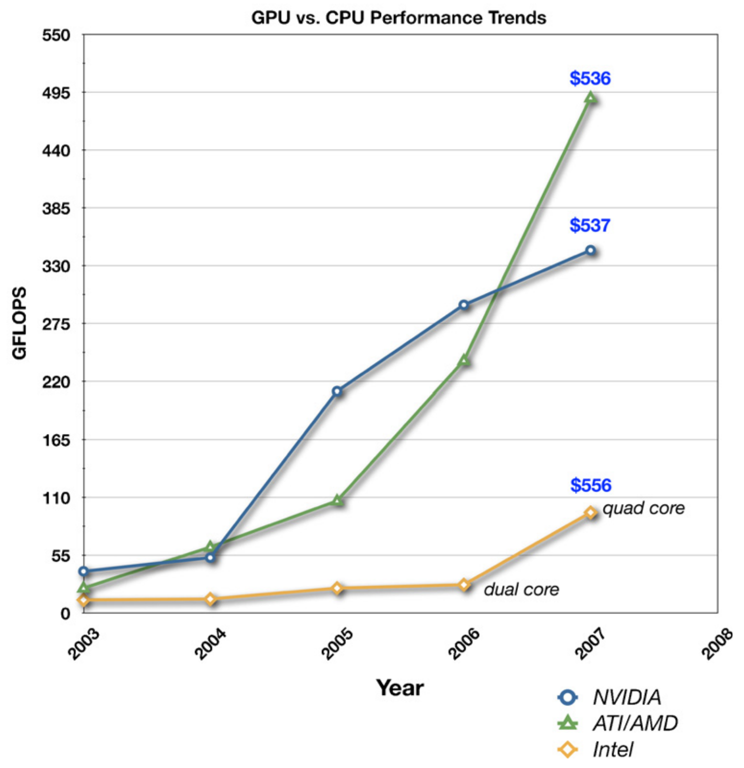

Fig. 1. Historical GPU versus CPU performance trends and current pricing for the most recent processors. Performance numbers courtesy of Mike Houston and John Owens. Pricing data from http://www.pricewatch.com, August 2007.

1.1. The graphics pipeline

The design of graphics processors has traditionally followed a common structure known as the *graphics pipeline*. This architecture breaks the process of rendering polygonal data into several subtasks. These tasks carry out the required steps of transforming the three-dimensional vertices of a polygon into two-dimensional screen coordinates, generating the individual pixels that make up the polygon, and finally processing each resulting pixel and assigning it a color in the frame buffer (video memory). A simplified version of this pipeline is shown in Fig. 2.

This design allows for many levels of parallelism resulting in high computational rates and throughput. Each of the stages are typically implemented as individual hardware components that allow for the overlapping execution of each task (i.e. pipeline parallelism). In addition, the pipeline is composed of multiple vertex and pixel/fragment processors, allowing for many data-parallel operations over the supplied set of input vertices and resulting pixels. For example, common hardware designs use 6–8 vertex processors and 32–48 pixel processors – this difference in processor counts addresses the typical mismatch between the number of vertices and the larger number of generated pixels. Coupled with these processors are memory systems that can reach bandwidths in the approximate range of 50–100 GB/s. To address memory latency issues, the hardware uses a fine-grained, lightweight threading model.

While early implementations of the rendering pipeline were based entirely on a fixed-function design, the last several hardware generations have introduced user-level programability. As shown in Fig. 2, programmers can supply code for both the vertex and fragment processors. In comparison to today's CPUs, the GPU provides a reduced set of instructions that support simple control flow structures and several mathematical operations, many of which are devoted to graphics specific functions. Each program is typically limited to a fixed number of available input and output parameters, registers, constant values, and overall number of instructions. The input parameters to programs can be scalar values, two-, three-, or four-component vectors, or

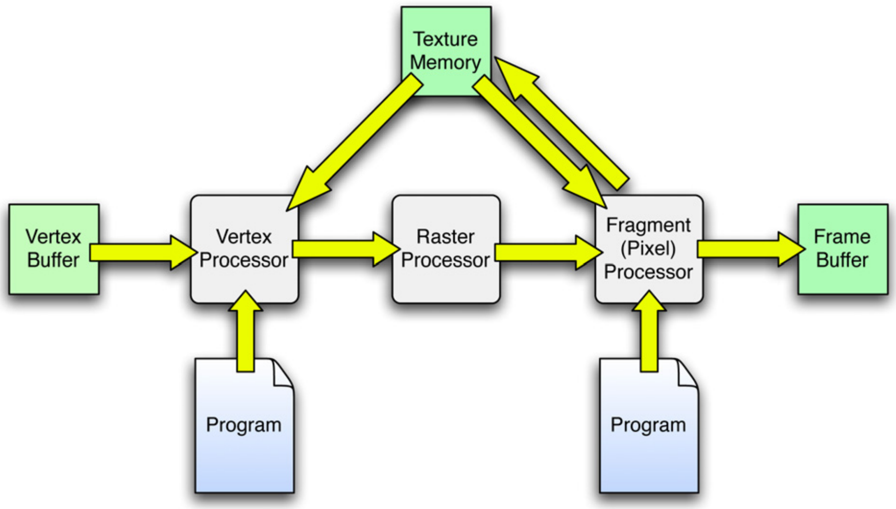

Fig. 2. A simplified graphics pipeline. The *vertex processor* is responsible for transforming the vertices of each polygon into screen space. The *raster processor* takes each transformed polygon and generates the set of pixels contained within the polygon. Finally, the *fragment* (*pixel*) *processor* assigns a color value for each pixel and stores the results in either the frame buffer (video memory) or texture memory.

arrays of values (scalar or vector) that are stored in the texture memory of the graphics card. Addresses into the texture memory are specified by assigning texture coordinates on a per vertex basis. The texture coordinates for each individual pixel are computed as part of the interpolation process carried out by the raster processor. Further details about the texture mapping process are covered in most computer graphics textbooks [30].

The internal flow of data between the pipelined processors is typically represented as four-component vectors, with each component storing a 32-bit floating-point value. Specifically, the output values from the vertex processors are a small number of vectors that represent the location, color, and texture coordinates of each vertex (most GPUs provide additional vectors that can be used to pass other information). The values are typically grouped to form the vertices of a triangle that are interpolated by the raster processor to produce pixels. The fragment processors receive these pixels as input and can output as many as four vectors; where each output vector represents the red, green, and blue components of a color, plus a fourth component representing the opacity level of the pixel. The resulting color(s) are stored in either the frame buffer or texture memory of the graphics card.

Recent hardware announcements have quickly changed the restrictions, capabilities, and the fundamental design of graphics processors. For example, the specialized vertex and pixel units have been replaced with a set of identical processors that use a unified instruction set architecture; thus transforming the GPU into a more general-purpose, multi-core coprocessor. Therefore, current graphics cards support two fundamental programming paradigms: one driven by graphics-centric operations and a second that directly presents the parallel nature of the underlying hardware. Scout's dual nature of supporting both general-purpose computation and visualization operations map directly to both of these models. The programming model for both past and current GPU architectures can present a challenge for the efficient development of general-purpose applications.

1.2. The GPU programming model

Historically, the process of leveraging GPUs for general-purpose computation has provided a programming model that is unfamiliar to most outside of the graphics community. Not only are developers faced with the traditional problems of exposing parallelism in their algorithms, they must also deal with mapping the computation into graphics primitives (polygons) and operations. Given that there are typically many more fragments than vertices, most general-purpose applications gain the most by targeting the pixel processors of the GPU. The details of this method, which is also the approach used by Scout's visualization operations, are covered in depth by Harris [13]. The basic process is summarized below:

(1) The program renders a set of vertices that correspond to the computational domain of the problem. This is typically achieved by rendering a quadrilateral that produces the number of pixels that correspond to the dimensions of the input and output arrays.
(2) To achieve the best possible performance, data stored in texture memory are best represented as two-dimensional arrays. This requires higher dimensional data to be stored in a tiled format. In addition, arrays with dimensions that exceed the limits of the hardware must be processed in multiple passes and care must be taken to address boundary conditions.
(3) Each of the generated pixels are processed by a user-defined program in a data-parallel manner. These programs can read from arbitrary memory locations but are only capable of writing results to a memory location that corresponds to the address of the current pixel in the frame buffer. In other words, the hardware supports gather, but not scatter operations. Recent changes have removed this limitation from some APIs and hardware combinations.
(4) The output of the program is 1–4 vectors per pixel that can be representative of the final results, or can be stored back into the texture memory and used by further computations.

With the introduction of more flexible hardware designs, the use of this programming model is no longer necessary for general-purpose computations. In particular, NVIDIA's CUDA presents a thread-based programming model for graphics hardware that is implemented with extensions to the ANSI C language [26]. While graphics hardware and software are evolving to meet the needs of a broader range of applications, efficiently and effectively programming the GPU still presents a significant challenge. This is especially true for developers, such as scientists, who are unfamiliar with the low-level details of the supporting APIs and hardware features. Many efforts that will be discussed in Section 2, have successfully addressed the issue of hiding the details of the GPU's programming model. The Scout programming language also addresses several of the associated challenges for both general-purpose computation and scientific visualization.

1.3. The Scout programming language

The fundamental objective of Scout is to provide scientists with a high-level, hardware-accelerated programming language for visualization and data analysis [24]. Software development in Scout is supported with an environment that provides tools for interacting with visualization results and simple code development tasks. We had several goals in mind when designing the language:

(1) It should be simple and concise.
(2) The compilation times should be fast enough to guarantee interactive performance for the user – this need is driven by the desired visualization features.
(3) It should reveal the parallel nature of the underlying hardware without complicating the language.
(4) Where possible, the language should hide any nuances introduced by graphics APIs and/or the hardware.
(5) The language should provide the user with flexible methods for producing both general-purpose computations (for data analysis operations) and visualization results.

These goals, and the underlying hardware design, led us to choose a data-parallel programming model for Scout. Building upon this approach, Scout is based on ideas and structures adapted from the C* programming language [34]. It is important to note that our goals were not to study the impact of a new approach to data-parallel programming, but to take advantage of previous work in this area and extend it to support visualization.

The next section covers previous work, including other programming languages for graphics processors and scientific visualization. The details of the Scout language are presented in Section 3 and the compiler technology in Section 4. The Scout runtime system is explained in Section 5. Section 6 presents examples developed using Scout. Finally, Section 7 concludes with a discussion of rapidly advancing trends in hardware, future directions, and further research opportunities.

2. Related work

The growing performance and programmability of graphics processors has fostered many efforts to capture these capabilities for use in general-purpose computations. A broad survey of the techniques and areas that have been explored is presented by Owens et al. [27]. In addition, the General-Purpose Computation Using Graphics Hardware website provides information about the background and current research activities in this area [12]. In this section, we review the supporting efforts in data-parallel algorithms and programming, discuss several languages that have been developed for graphics processors, and provide an overview of other languages targeted for assisting in the visualization of scientific data.

2.1. Data-parallel programming languages

Data-parallel programming languages provide constructs that are explicitly parallel, avoiding the need for compilers to discover parallelism in sequential programs. These constructs execute the same operation on each element of a data structure (e.g. a parallel addition) or move and compute on elements of a data structure (e.g. a prefix sum) [4,6]. Starting with APL in the 1960s [16], data-parallel languages have provided a simple, scalable programming model for parallel computers. A general overview of data-parallel languages is found in Sipelstein [32], and Chamberlain et al. provide a detailed overview of related parallel programming approaches [9]. Data-parallel operations are included as part of Fortran 90 and HPF. Data-parallel versions of FORTRAN, C and Lisp (CM Fortran, C*, and CMLisp) were provided for the Connection Machine from Thinking Machines Corporation. More recent work on data-parallel programming languages includes ZPL and Chapel [8,9].

While data-parallel languages tend to provide a simplified programming model and can be mapped to many different machine architectures, the approach is often not well suited for dealing with irregular data structures (e.g. sparse matrices, graphs, etc.). This issue has been addressed by past efforts with the introduction of nested data-parallel languages. Nested data-parallel languages, such as NESL [5], allow the use of recursive data structures and the invocation of parallel functions on many different sets of data in parallel. Although we are actively exploring the use of nested parallelism, the version of Scout described in this paper does not provide this functionality.

2.2. GPU programming languages

The evolution of graphics hardware from a fixed-function to a programmable pipeline has allowed for the development of multiple approaches for programming them. For presentation, these approaches can be grouped into three categories: shading languages, programming abstractions, and new languages.

Shading languages are directly targeted at the programmable stages of the graphics pipeline. The most common shading languages are Cg [19], the OpenGL Shading Language [18], and Microsoft's High-Level Shading Language [25].

These languages are primarily used in the entertainment industry for adding realism to rendered scenes. While it is possible to develop powerful and complex general-purpose programs using these languages, they require supporting graphics code for configuration, resource management, the creation of graphics primitives, runtime control, and the movement of data between the CPU and GPU. The low-level, graphics-centric nature of the shading languages make them ill-suited for most developers of general-purpose applications. With ATI's "Close to the Metal" (CTM) library, the programmer writes in a low-level assembly language and then uses a set of library functions to manage the execution of the program on the GPU [2]. Thus, the supporting application is no longer graphics-centric but it retains a structure and organization similar to that required for a shading language program.

Programming abstractions leverage the features of an existing language to provide a simplified model for programming the graphics hardware. This is typically achieved by providing a library, or meta-language, and using existing compilers and tools for software development. For example, the RapidMind Platform is a C++ class library that provides new data types and captures a series of operations that are translated into low-level GPU instructions [23]. Glift provides a generic template library that simplifies the design and development of GPU-based data structures [21]. Accelerator provides a C# library of abstractions for data-parallel arrays and operations that allows programmers to combine operations for the CPU and GPU within a single application [33]. To improve efficiency the library uses a graph-based, lazy evaluation of data-parallel operations on the GPU; the evaluation is typically triggered by the explicit conversion of an array from a data-parallel to a serial representation.

New programming languages typically include a compiler that translates a high-level language into the low-level details of programming the graphics hardware. This translation includes both the supporting runtime code as well as one or more shading language programs. The Brook programming language provides a set of streaming extensions to ANSI C and supports the generation of a GPU-targeted executable image [7]. In this model, Brook's streaming operations (kernels) are converted into fragment programs and streaming data values are stored in texture memory. CGiS is similar to Brook in that it provides streaming data types, but it uses an explicit `forall` construct to identify data-parallel operations for the GPU [22].

NVIDIA has recently released the Compute Unified Device Architecture (CUDA) that provides an extended version of ANSI C for developing general-purpose applications on the GPU [26]. These extensions introduce type qualifiers that allow programmers to specify the target platform (CPU or GPU) for functions and variables, and to control the number of threads created on the GPU at runtime. Scout follows the concepts of introducing a new language but also extends the graphics-centric functionality of the shading languages to provide support for directly programming visualization operations.

2.3. Visualization languages

While the generality of the shading and general-purpose GPU programming languages allow them to be used for a broad range of applications, they lack explicit support for visualization operations. There has been very little work done in terms of designing languages with built-in visualization support. Two examples of early languages intended for visualization are Texl[11] and the data shaders described by Corrie and Mackerras[10]. These approaches are closely related to the early development of programmable shaders in computer graphics that eventually led to today's hardware-accelerated shading languages. Both languages maintain the fundamental support for shading operations but they introduce additional features for processing data sets contained within textures – this is the fundamental approach used by the low-level GPU programming model presented in Section 1.2. The Scout language follows the premise of both Texl and data shaders but adds visualization constructs to a general-purpose language. These constructs allow for a much broader range of visualization operations in comparison to extending one of the shading languages.

An important aspect of a language for visualization is the impact it has on the feedback loop with the user. This requires that the overhead needed for compilation and associated tasks be kept to a minimum in order to maintain interactive performance. In general, any approach that requires the full compilation, linking, and loading of an executable image each time a program is modified can seriously impact the response time of the system. Therefore, the runtime compiler support provided by shading languages is appropriate for visualization. However, the graphics-centric nature of the shading languages complicate the programming model. Therefore, Scout leverages the runtime nature of the shading languages, while providing the abstractions and flexibility of the general-purpose languages.

3. The Scout programming language

As discussed in Sections 1 and 2, Scout is a data-parallel programming language that builds upon the ideas used in several languages including C* and CM FORTRAN. This choice was driven by the explicit data-parallel nature of the GPU's fragment processors. In general, this approach is made possible by the underlying use of thousands of lightweight threads that are hidden underneath the low-level programming model. In this section, we present an overview of the basic structure of the language, focusing on the use of explicit parallelism, visualization extensions, and data-parallel operators.

3.1. Explicit parallelism

The majority of Scout's language features are extended versions of those introduced in C* by Thinking Machines [34]. C* provides a set of extensions to the standard C language for explicit parallel programming. Although slightly different in syntactic details, Scout builds directly on the fundamental concepts of C*.

The foundation of a data-parallel program in Scout is the concept of *shapes*. Shapes define a template for declaring a block of data that is operated on in parallel. Each shape is defined by a *rank* and the number of *positions* along each rank. The rank provides the number of dimensions represented by a shape and the positions define the number of cells along each rank. Thus, the total number of cells in a shape is given by the product of the number of positions along each rank. The syntax for declaring a shape follows a familiar array-like syntax. For example, `shape myShape [6,4]`, defines a shape of rank 2 that has $6 \times 4 = 24$ total positions. Once a shape has been defined, it is possible to create one or more data-parallel variables. Parallel variables in Scout are declared by explicitly providing both the type and shape of the variable – `float:myShape myVariable`. In addition to declarations, shapes are used as part of the explicit parallelism structure in programs.

Like C*, the `with` statement in Scout provides the structure for explicit parallelism within a program. The code nested inside a `with` block will be executed in a data-parallel fashion over the individual cells of a specified shape. Scalar or parallel variables may be declared within the block. All parallel variables used for computation within the block must have the same shape as that specified by the argument to the enclosing `with` statement. Any expressions outside the scope of a `with` block are executed in serial on the host CPU. Explicit parallelism maps directly to the hardware of the GPU and has a distinct advantage in reducing the complexity of the compiler. This directly contributes to our goal of supporting interactive visualization.

3.2. Visualization extensions

The basic structure of the `with` statement allows us to identify parallelism within a Scout program, but it does not directly support visualization-centric operations. Scout provides this ability by extending the syntax to support keywords that modify the behavior of `with` statements. For example, a `render with` statement specifies that the instructions within a parallel block of code will produce visual output. As part of this process the programmer has explicit control of how data values are mapped to the final pixels in the resulting imagery. This is accomplished by using predefined functions that represent traditional color spaces in computer graphics (e.g. RGB and HSV) and assigning the final results to the built-in variable `image` that represents the pixels in the frame buffer. Several `with` block modifiers are supported, including volume rendering, ray casting, and point-based rendering. Together these features allow Scout to function as a shading language that is tailored to the modifier applied to the `with` statement.

The ability to directly program the mapping from data values to colors can be very powerful. However, it introduces a fundamental challenge in terms of placing the GPU in its traditional role as a graphics engine, as well as supporting the acceleration of general-purpose computations. This can complicate the details of supporting advanced data-parallel operations.

3.3. Data-parallel operators

Many traditional parallel operations can be difficult to implement given some of the restrictions of the GPU programming model and/or hardware limitations. This is especially true for code that is targeted at the graphics pipeline. For example, reduction operations are performed in several passes through the pipeline, with the size of the input being cut in half at each pass. This results in an algorithm with $\mathcal{O}(\log n)$ passes for a reduction over n elements. More details about implementing reductions on graphics hardware are presented in [7]. Scans, also known as prefix-sums, are useful for a wide range of data-parallel algorithms; they are especially useful for dealing with irregular data structures [6]. As several efforts have shown, the implementation of these algorithms on the graphics processor can be challenging [14,31]. In many cases, the introduction of general-purpose GPU hardware and programming languages can significantly reduce the complexity of implementing these operations. However, as shown by Sengupta et al., efficient algorithms can still require a significant amount of effort [35].

Scout provides several fundamental operators such as shifts, reductions, and scans that are implemented efficiently and supported as high-level operations in the language. The ability to include these higher level operations within both visualization and compute blocks requires that we provide implementations for both the graphics-centric and general-purpose modes of operation. The determination and execution of the appropriate method is handled by both the compiler and the supporting runtime system.

4. Compiler design and implementation

Several issues arise when designing a compiler for the GPU. First, the GPU requires runtime (CPU-side) support to execute programs and move data between the main system and graphics memories. In addition, advanced operations like those described in Section 3.3, may require the management of multiple executions of the rendering pipeline. These issues, and our desire to enable interactive performance, drive the fundamental structure of the Scout compiler. The overall design resembles a typical optimizing compiler, with a parser producing a tree that is used to generate an intermediate representation (IR) of the program [1]. The IR is then subject to control-flow and data-flow analysis. The control-flow stage, like that of a traditional optimizing compiler, produces a directed graph of basic-blocks. These are contiguous lines of IR code which have been identified as having no internal branching. Each branch instruction, and each location that may receive a branch, define the basic-block boundaries.

In most compilers, the control-flow graph is used for analysis and optimization, and then the blocks are reassembled into a sequential stream of optimized instructions. When targeting the combination of the CPU and GPU it is necessary to generate two instruction streams and their associated data dependencies. In this case, our optimization goals are to minimize the number of costly data transfers between the CPU and GPU, as well as increase the computational intensity on the GPU by maximizing the number of basic-blocks that can be merged together. To address this, we mark the blocks with platform affinities that describe both their requirements and restrictions. The compiler then merges the basic-blocks in a prioritized order, using a simple set of analytic and heuristic rules that govern the combination of affinities.[1] For example, the `with` block structure used in Scout can be used to identify explicit tasks. The code found inside a `render with` block will not be merged with code found inside another `render with`, or with code found inside a compute-oriented block.

Inside each `with` statement there may be multiple blocks, depending on the specifics of the affinities involved. Once the compiler has all the information related to the merged blocks it is possible for the back-end to generate different code for compute- and graphics-oriented IR blocks. This allows compute blocks to be transformed into CUDA code, and graphics-centric blocks can be translated into Cg.

While the Scout compiler does very few other traditional code optimizations, it does take steps to improve the utilization of the graphics hardware. For example, one approach to generating code for a shift operation on a data-parallel variable within a `render with` block would be to compute the shifted index (i.e. texture coordinates) within the code targeted for the fragment processors. A more efficient approach is to generate code that computes the shifted texture coordinates using the vertex processors. The fragment processors will then receive coordinates that have been shifted by the interpolation hardware in the raster processor. This approach leverages the pipeline parallelism available within the hardware to provide a modest improvement in performance.

The final output of the compiler is an augmented dependency graph that consists of program nodes that represent the merged code blocks, and data nodes that hold the input and output parameters. The edges between these two classes of nodes represent the path of data movement between the merged code blocks. The program nodes initially represent code in the form of a high-level intermediate representation, which is then finally compiled by the back-end pass. Further challenges can arise during this final compilation stage if the code within a single program node exceeds the limitations of the graphics hardware. In this situation, the code must be split into smaller pieces until the resulting code meets the requirements of the hardware. Rif-

[1] The merging process is simplified by converting the graph into pruned Single Static Assignment (SSA) form [20].

fel et al. provide the details for achieving this partitioning [28]. After the compilation stage has completed the dependency graph is directly executed by the runtime system.

5. Runtime environment

The GPU's role as a coprocessor requires that a supporting runtime environment on the CPU handle the overall details of program execution. This runtime layer is responsible for compiling and loading programs onto the GPU, configuring and binding input and output parameters, starting the execution of the graphics pipeline (i.e. executing programs), and managing the movement of data back and forth between CPU memory and the graphics card. The necessary details for handling these series of operations are explicitly defined in the dependency graph generated by the compiler.

The execution of the program nodes within the dependency graph follows the concepts introduced for data-flow hardware architectures [36]. Although this approach to hardware design had limited success, the model is widely used as a software architecture within many disciplines. In particular, data-flow abstractions have played a significant role within the visualization community since the 1990s [37,39,38]. This technique has several advantages including supporting the ability to process large data sets using data streaming [40], and providing a structure for exploiting further parallelism.

On systems with a single graphics processor, the runtime system defaults to using a simple static scheduling algorithm that sequentially executes all program nodes with GPU affinity according to the dependency information provided by the graph. The CPU is left to executing serial computations and any other tasks required to support the code(s) executing on the graphics card.

The data-flow structure of the dependency graph provides further opportunities on systems that contain multiple graphics cards. Specifically, it provides the runtime system with enough additional information to leverage additional levels of parallelism. Fig. 3 presents three possible approaches to introducing this

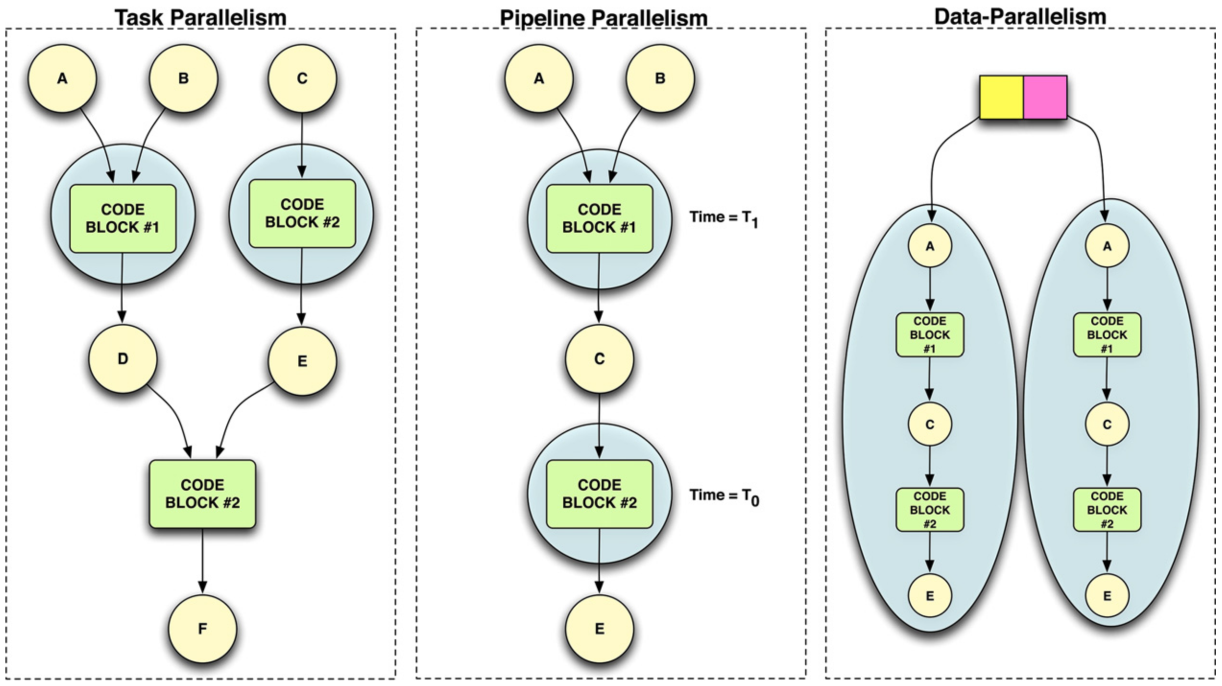

Fig. 3. The structure of the dependency graph supports various forms of additional parallelism beyond that provided by the graphics hardware. In particular, task, pipeline, and data-parallelism are options. Yellow nodes represent input/output parameters, program nodes are shown in green, and the regions highlighted in blue show program nodes that may be executed in parallel. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

parallelism in addition to that provided by the graphics hardware. This parallelism can be handled entirely by the runtime system requiring no further compilation steps.

The determination of task parallelism within the graph is provided directly by finding all program nodes with inputs that are ready to be processed. Once identified, the runtime system can schedule each identified task on the available hardware resources. The `with` block structure of Scout programs naturally leads to program graphs supporting this form of parallelism. Although the approach appears to be beneficial, care must be taken to understand the impact that data movement requirements will have on the overall performance. If several tasks depend upon the same set of input parameters it is likely that the transfer of these parameters to multiple graphics cards will result in a performance bottleneck.

Pipeline parallelism can be leveraged when processing time series, or partitioned data sets. While it has the potential to improve performance, the overall costs of data movement between multiple graphics cards must be carefully considered to avoid degraded run times. For example, it may be much more advantageous to leave the data within a single graphics card than it would be to move the data to CPU memory, and then to a second card for execution of the next pipeline stage.

In the final form of parallelism, the runtime system partitions all input data sets between the available GPUs and the executes multiple copies of the entire dependency graph on multiple CPUs and GPUs. Like the previous methods, the details of the underlying computer system architecture have a direct impact on the performance. In particular, care must be taken to ensure that GPUs do not share a common bus, and ideally that CPUs do not share a memory controller. Finally, the details of boundary conditions must be carefully considered as data will have to be shared between the participating GPUs.

These additional forms of parallelism offer many different opportunities for exploring scheduling algorithms that leverage two or more GPUs. Given the impact that data transfers can have on the overall performance of these methods, we are currently exploring scheduling algorithms that seek to limit the overall amount of data movement. Achieving consistent and beneficial performance improvements with this overall approach is still an area of active study.

6. Results

In this section, we provide an overview of how Scout can be used to explore the results of a pre-computed data set, as well as how it can be used to implement a simple numerical simulation of dendrite growth.

6.1. Visualization results for a cosmological simulation of dark matter

Cosmological simulations track the formation of nonlinear structures in the dark and luminous matter that makes up the universe. The resulting data sets are typically very large, which often makes the process of interactive exploration of the data difficult on most desktop computers. The images in Fig. 4[2] show the results of using Scout to explore the inconsistency between two different particle-based simulation codes that were started from the same initial conditions. These results were created by a single Scout program that first computes the magnitude of the underlying velocity field from the simulation. This result is then used to color the set of particles that occupy cells with a specific density range. Finally, a set of localized clusters of tracer particles (halos) that fall into a certain range of mass values and occupy the lower density regions of the grid are rendered as red points. These computations were executed for each data set and displayed in a side-by-side fashion to allow for easier study. The user interface for this program provides the user with control over the scalar parameters that specify the density and mass values used within the programs.

Using this single program it was possible for scientists to see that there were fewer halos in the lower density regions of one simulation in comparison to the other.[3] More importantly, we were able to use the power of the GPU on a laptop computer to compute, query, and visualize the results of 32 million particles at approximately 24 frames a second. Fig. 5 presents a portion of the code that was used to produce these results.

[2] For interpretation of the references to color in this figure, the reader is referred to the web version of this article.

[3] This result was due to the behavior of the adaptive grid computation used in the corresponding simulation.

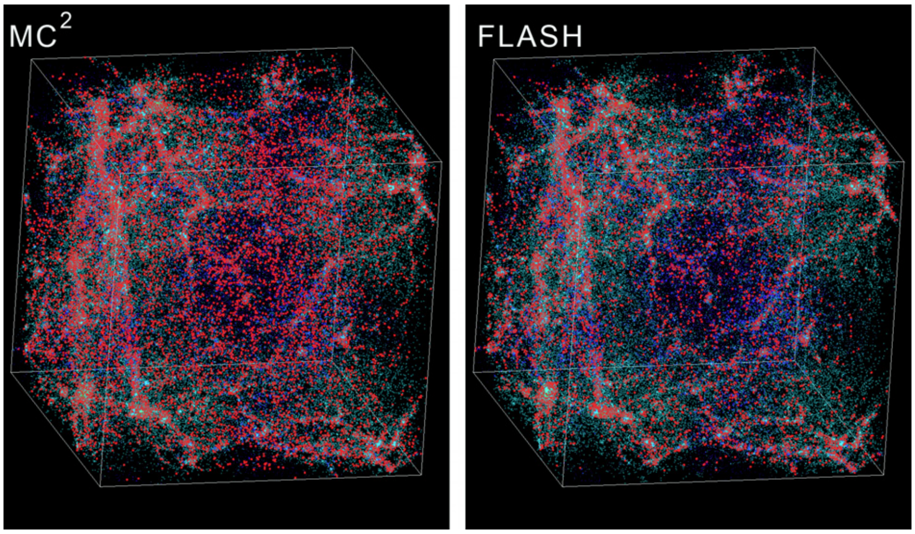

Fig. 4. A qualitative comparative visualization of the results produced by two different application codes. The results show red colored halos with a low mass that also occupy low density regions. Note the fewer number of halos in the right image

```
// Create the left viewport to display the MC^2 results.
viewport "MC2" (0.0, 0.0, 0.5, 0.5)
{ // GridShape is the shape of the particle data sets.
  float:gridShape mag;
 // Compute the magnitude of the velocity field.
  with(gridShape) {
    mag = magnitude(mc2_velocity);
  }

  // Display only those points within the density range and
  // color each by velocity magnitude.
  render points with(pointSet) {
    where(density >= user_set_density)
      image = hsva(240*(max(mag)-mag) /
                   (max(mag)-min(mag)), 1,1,1);
    else
      image = null;
  }
  // Render the halos only where the mass and density
  // fall within given ranges. Halos are colored red.
  render points with(haloSet) {
    where(mass >= user_set_halo_mass && density >= user_set_density)
      image = rgb(1,0,0); // Use the RGB color space to select red.
    else
      image = null;
  }
} // end viewport.
```

Fig. 5. A portion of a Scout program that computes velocity magnitude, selects and renders a user selected density range of specified dark matter particles and halos. The program uses the `viewport` language construct to create a side-by-side comparison of two different simulation results. In addition, the code uses the `hsva` function to select a range of colors from the hue, saturation, and value color space.

6.2. Computation results for a 2D dendrite growth simulation

We have recently started exploring the use of Scout for the implementation of several simple physics codes. At this point in time, we have restricted ourselves to those applications can that be easily represented on the GPU to understand the raw performance of the physics kernels themselves. One example of this work has been to study simple two and three-dimensional dendrite growth using a phase field method with a forward-Euler time stepping scheme. We solve coupled PDEs for phase field and enthalpy variables [17]. In the simulation presented below, the dendrite has four-way symmetry using homogeneous Neumann (symmetry) boundary conditions on all sides.

For reference the equations for the phase field are:

$$[A(n)]^2 \frac{1}{Le}\frac{\partial\phi}{\partial t} = \phi(1-\phi^2) - \lambda(1-\phi^2) + \nabla\dot{(}[A(n)]^2\nabla\phi) \tag{1}$$

$$+\frac{\partial}{\partial x}\left[A(n)A'(n)\frac{\partial\phi}{\partial y}\right] + \frac{\partial}{\partial y}\left[A(n)A'(n)\frac{\partial\phi}{\partial x}\right] \tag{2}$$

The enthalpy is computed using the following equation:

$$\frac{\partial\theta}{\partial t} = (LeD)\nabla^2\theta + \frac{1}{2}\frac{\partial\phi}{\partial t} \tag{3}$$

```
with(shapeof(T)) {
  ...
  ...
  // 2nd derivative of enthalpy
  float:T d2Wdx2 = rdx2*(eoshift(T,0,1) +
                   eoshift(T,0,-1) - 2.0*T);
  float:T d2Wdy2 = rdy2*(eoshift(T,1,1) +
                   eoshift{T,1,-1) - 2.0*T);

   // time derivative of phi
   float source = 0.5*(pnext - P) / beta / dt;

   // enthalpy evolution equations
   enext = T + dt * Le * D * (d2Wdx2 + d2Wdy2) +
           dt * source;
}

viewport "Phase Field" (0.0, 0.0, 0.5, 1.0)
{
  render with (shapeof(P)) {
    float:shapeof(P) Pimage = 240.0 - 240.0 * pnext;
    image = hsva(Pimage, 1, 1, 1);
   }
}

viewport "Temperature" (0.5, 0.0, 0.5, 1.0)
{
  render with (shapeof(P)){
    float:shapeof(P) Timage = 240.0 - 240.0 * (enext - 1.0) / 0.55;
    image = hsva(Timage, 1, 1, 1);
  }
}
```

Fig. 6. A portion of a Scout program that computes the dendrite evolution.

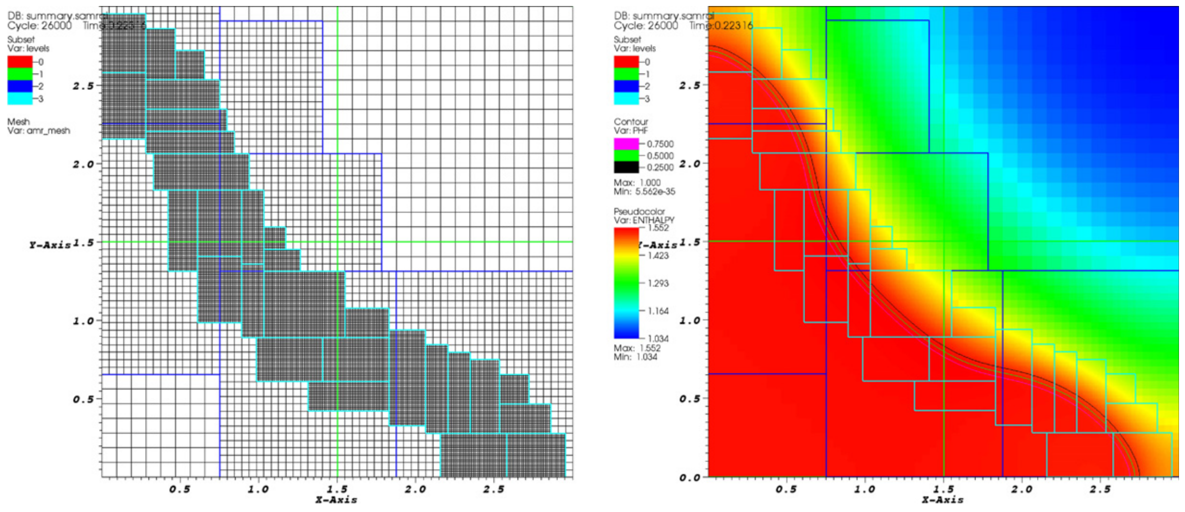

Fig. 7. The AMR mesh (left) and enthalpy (right) from a single time step of the QSC simulation.

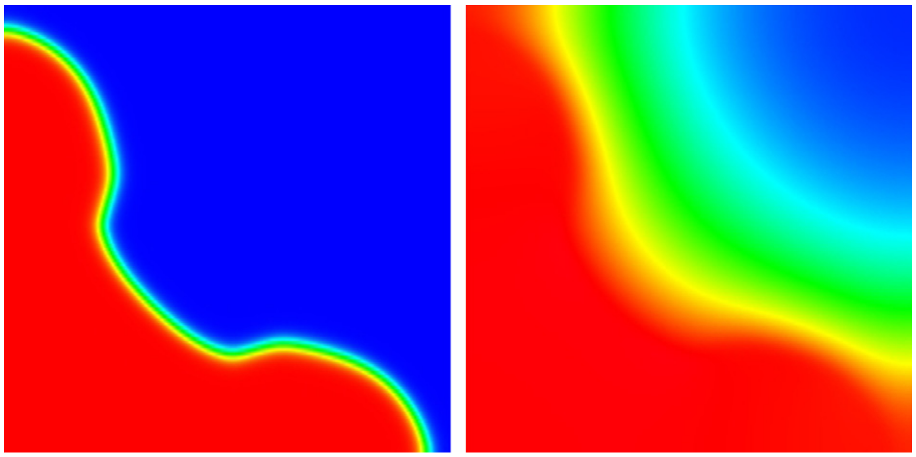
Fig. 8. The phase-field (left) and enthalpy (right) results produced by the GPU simulation.

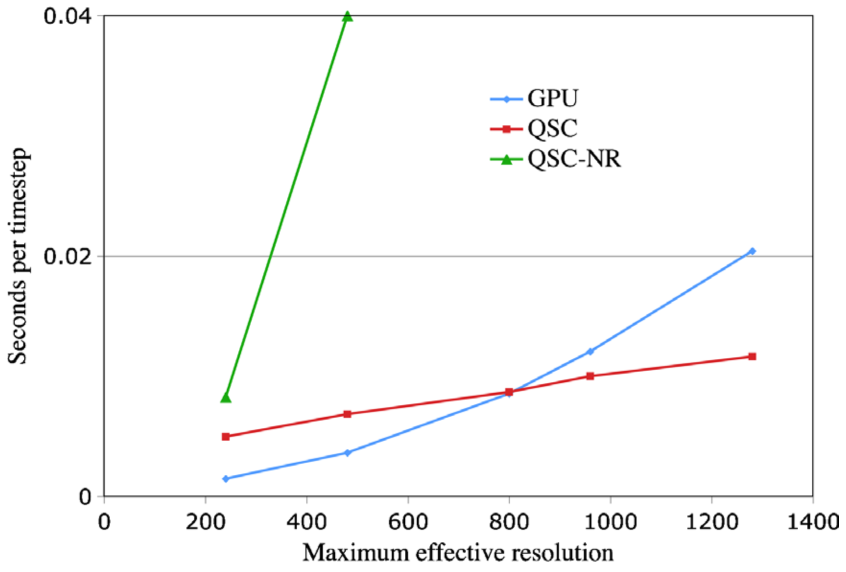

Fig. 9. Timing comparison between the GPU and a single QSC node (4 CPUs).

A portion of the Scout code that shows the update equation for enthalpy is presented in Fig. 6. In general, the code shows the ease of coding basic physics simulations using a data-parallel language. To understand the overall impact in coding time, complexity, and performance we compared our results with simulations performed using SAMRAI, an adaptive mesh refinement software package developed at Lawerence Livermore National Laboratory [29]. The SAMRAI code was run on four Alpha EV6 processors sharing 16 GB of RAM. The GPU simulations were performed on an HP xw8200, using an NVIDIA Quadro FX 4500 with 512 MB of texture memory installed.

For our benchmarks the GPU code was run at the finest resolution throughout the entire domain. Fig. 7 shows the grid refinement used by the CPU that corresponds to the end of a 240×240 run and the corresponding enthalpy result. The phase-field and enthalpy results for the GPU simulation are shown in Fig. 8. The final performance results are presented in Fig. 9[4] with the green line showing the performance of QSC-NR, the four

[4] For interpretation of the references to color in this figure, the reader is referred to the web version of this article.

Alpha processors running without any adaptive refinement. The GPU performance shown by the blue line, highlights the comparison of the native speed of both processors. Finally, the red line shows the performance of the Alpha processors using adaptive mesh refinement (AMR). This clearly shows that the adaptive approach works well in comparison to the non-refining implementation. The GPU initially outperforms the AMR code but the benefits of AMR quickly become apparent at higher resolutions. Note, however, that the amount of work saved by AMR is a function of the interface area, and for a highly convoluted interface (unlike this simple test case), the majority of the domain could be refined. In this case, the overall performance would approach that of the non-AMR implementation, while the GPU case would be unaffected. Thus the crossover point in Fig. 9, where the AMR simulation outperforms the GPU, is strongly problem dependent.

Furthermore, the implementation of the Scout-based code took less than a week of effort, which is substantially faster, and likely much less error prone, than implementing the complexities of an adaptive mesh code. While the GPU approach has obvious disadvantages with regards to total available memory and the associated hardware limitations (e.g. lack of fully compliant IEEE floating point support), this comparison provides a glimpse into what may be possible on future architectures.

7. Conclusions

The Scout data-parallel programming language we have presented provides the ability to leverage the computational power of the graphics processor and simplifies the overall process of developing code for data analysis, visualization, and general-purpose computation. With graphics processors making the transition from fixed-functionality to a more general-purpose processor, it is clear they will likely influence the design of future hardware. While the overall trends are positive, graphics processors still have limitations that make them difficult to adopt within the high-performance computing community. In particular, their lack of double precision is unacceptable for many application domains with high dynamic ranges, as is the lack of fully compliant floating point exceptions and error correcting memories.

With the introduction of multi-core CPUs quickly becoming an industry-wide trend, GPUs can provide us some insight into many of the challenges that we will face in terms of finding parallelism within our applications. In addition, the plans from both AMD and Intel to provide a closer coupling between CPUs and coprocessors [3,15], will also present the challenge of effectively and efficiently programming heterogeneous, massively parallel, computer systems. Our future efforts will include addressing these issues as well as the complexity of large-scale systems built from these emerging commodity components. We will also continue to explore the details of providing additional language structures for directly programming visualization and data analysis operations.

Acknowledgements

This work was supported by the United States Department of Energy Office of Advanced Scientific Computing Research. Special thanks to John Owens and Mike Houston for help with the CPU and GPU performance results presented in Section 1. We would also like to thank the reviewers for their valuable and thoughtful suggestions on improving the paper.

References

[1] A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools, Addison Wesley, 1988.
[2] Advanced Micro Devices Inc., ATI CTM Guide – Technical Reference Manual. <http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf>, 2006.
[3] Advanced Micro Devices Inc., Torrenza Technology. <http://enterprise.amd.com/us-en/AMD-Business/Technology-Home/Torrenza.aspx>, 2006.
[4] G. Blelloch, Vector Models for Data-parallel Computing, MIT Press, Cambridge, MA, 1990.
[5] G. Blelloch, S. Chatterjee, J.C. Hardwick, J. Sipelstein, M. Zagha, Implementation of a portable nested data-parallel language, Journal of Parallel and Distributed Computing 21 (1) (1994) 4–14.
[6] G. Blelloch, Scans as primitive parallel operations, IEEE Transactions on Computers 38 (11) (1989) 1526–1538.

[7] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, Brook for GPUs: stream computing on graphics hardware, ACM Transactions on Graphics 23 (3) (2004) 777–786.
[8] B.L. Chamberlain, The Design and Implementation of a Region-based Parallel Language, PhD Thesis, University of Washington, November 2001.
[9] B.L. Chamberlain, D. Callahan, H.P. Zima, Parallel programmability and the Chapel language, International Journal of High Performance Computing Applications 21 (3) (2007) 291–312.
[10] B. Corrie, P. Mackerras, Data Shaders, in: VIS'93: Proceedings of the 4th Conference on Visualization 1993, 1993, pp. 275–282.
[11] R.A. Crawfis, M.J. Allison, A scientific visualization synthesizer, in: VIS'91: Proceedings of the 2nd Conference on Visualization, 1991, pp. 262–267.
[12] GPGPU, General-purpose Computation Using Graphics Hardware, http://www.gpgpu.org, 2007.
[13] M. Harris, Mapping computational concepts to GPUs, in: M. Pharr (Ed.), GPU Gems 2, Addison Wesley, 2005, pp. 493–508 (Chapter 31).
[14] D. Horn, Stream reduction operations for GPGPU applications, in: M. Pharr (Ed.), GPU Gems 2, Addison Wesley, 2005, pp. 573–589 (Chapter 36).
[15] Intel Inc., Geneseo: PCI Express Technology Advancement, <http://www.intel.com/technology/pciexpress/devnet/innovation.htm>, 2006.
[16] K. Iverson, A Programming Language, Wiley, New York, 1962.
[17] A. Karma, W-J. Rappel, Quantitative phase-field modeling of dendritic growth in two and three dimensions, Physical Review E 57 (1998) 4323–4349.
[18] J. Kessenich, D. Baldwin, R. Rost, The OpenGL Shading Language Version 1.10.59, <http://www.opengl.org/documentation/oglsl.html>, April, 2004.
[19] W. Mark, R. Glanville, K. Akeley, M. Kilgard, Cg: a system for programming graphics hardware in a C-like language, ACM Transactions on Graphics 22 (3) (2003) 896–907.
[20] R. Cytron, J. Ferrante, B. Rosen, M.N. Wegman, F.K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM Transactions on Programming Languages and Systems 13 (4) (1991) 451–490.
[21] A.E. Lefohn, J. Kniss, R. Strzodka, S. Sengupta, J.D. Owens, Glift: an abstraction for generic, efficient GPU data structures, ACM Transactions on Graphics 26 (1) (2006) 60–99.
[22] P. Lucas, N. Fritz, R. Wilhelm, The CGiS compiler, in: Proceedings of the 15th International Conference on Compiler Construction, Lecture Notes in Computer Science, vol. 3923, Springer, March 2006, pp. 105–108.
[23] M. McCool, Data-parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform, GSPx Multicore Applications Conference, Santa Clara, October 31–November 2, 2006, <http://www.rapidmind.net/pdfs/WPdprm.pdf>.
[24] P. McCormick, J. Inman, J. Ahrens, C. Hansen, G. Roth, Scout: a hardware-accelerated system for quantitatively driven visualization and analysis, in: IEEE Visualization 2004, October 2004, pp. 171–178.
[25] Microsoft Corporation, Microsoft High-Level Shading Language, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/hlslreference/hlslreference.asp>, 2005.
[26] NVIDIA Corporation, NVIDIA CUDA Homepage, <http://developer.nvidia.com/object/cuda.html>, 2007.
[27] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, T. Purcell, A survey of general-purpose computation on graphics hardware, in: Computer Graphics Forum, March 2007, pp. 80–113.
[28] A. Riffel, A. LeFohn A., K. Vidimce, M. Leone, J.D. Owens, Mio: fast multipass partitioning via priority-based instruction scheduling, in: Graphics Hardware 2004, August 2004, pp. 35–44.
[29] SAMRAI (Structured Adaptive Mesh Refinement Application Infrastructure) Homepage, www.llnl.gov/CASC/SAMRAI, 2007.
[30] D. Shreiner, M. Woo, J. Neider, T. Davis, OpenGL Programming Guide: the Official Guide to Learning OpenGL, Addison-Wesley, 2006.
[31] S. Sengupta, A.E. Lefohn, J.D. Owens, A work-efficient step-efficient prefix sum algorithm, in: Proceedings of the Workshop on Edge Computing Using New Commodity Architectures, August 2007, pp. D-26-27.
[32] Jay Sipelstein, Guy E. Blelloch, Collection-oriented languages, Proceedings of the IEEE 79 (4) (1991) 504–523.
[33] D. Tarditi, S. Puri, J. Oglesby, Accelerator: using data-parallelism to program GPUS for General Purpose Uses, in: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, October 2006, pp. 325–335.
[34] Thinking Machines Incorporated, C* User's Guide, 1991.
[35] S. Sengupta, M. Harris, Y. Zhang, J.D. Owens, Scan Primitives for GPU, in: Computing Graphics Hardware 2007, August 2007, pp. 97–106.
[36] J.B. Dennis, Data flow supercomputers, Computer 13 (11) (1980) 48–56.
[37] D.S. Dyer, Visualization: a dataflow toolkit for visualization, IEEE Computer Graphics & Applications 10 (4) (1990) 60–69.
[38] W.J. Schroeder, K.M. Martin, W.E. Lorensen, The design and implementation of an object-oriented toolkit for 3D graphics and visualization, in: Proceedings of Visualization'96, 1996, pp. 93–100.
[39] G. Abram, L. Treinish, An extended data-flow architecture for data analysis and visualization, in: Proceedings of Visualization'95, 1995, pp. 363–370.
[40] J. Ahrens, K. Brislwan, K. Martin, B. Geveci, C.C. Law, M. Papka, Large-scale data visualization using parallel data streaming, IEEE Computer Graphics & Applications 21 (4) (2001) 34–41.