

# A framework for exploring numerical solutions of advection–reaction–diffusion equations using a GPU-based approach

Allen R. Sanderson · Miriah D. Meyer ·  
Robert M. Kirby · Chris R. Johnson

Received: 17 March 2006 / Accepted: 22 March 2007 / Published online: 4 March 2008  
© Springer-Verlag 2008

**Abstract** In this paper we describe a general purpose, graphics processing unit (GP-GPU)-based approach for solving partial differential equations (PDEs) within advection–reaction–diffusion models. The GP-GPU-based approach provides a platform for solving PDEs in parallel and can thus significantly reduce solution times over traditional CPU implementations. This allows for a more efficient exploration of various advection–reaction–diffusion models, as well as, the parameters that govern them. Although the GPU does impose limitations on the size and accuracy of computations, the PDEs describing the advection–reaction–diffusion models of interest to us fit comfortably within these constraints. Furthermore, the GPU technology continues to rapidly increase in speed, memory, and precision, thus applying these techniques to larger systems should be possible in the future. We chose to solve the PDEs using two numerical approaches: for the diffusion, a first-order explicit forward Euler solution and a semi-implicit second order Crank–Nicholson solution; and, for the advection and reaction, a first-order explicit solution. The goal of this work is to provide motivation and guidance to the application scientist interested in exploring the use of the GP-GPU computational framework in the course of their research. In this

paper, we present a rigorous comparison of our GPU-based advection–reaction–diffusion code model with a CPU-based analog, finding that the GPU model out-performs the CPU implementation in one-to-one comparisons.

## 1 Introduction

Advection–reaction–diffusion has been widely applied to solve transport chemistry problems in scientific disciplines ranging from atmospheric studies [28], through medical science [29], to chemotaxis [12]. Turing’s original paper published in 1952, “The chemical theory of morphogenesis”, is the best-known discussion [27]. In this paper, Turing describes a system that both reacts and diffuses, reaching (under certain circumstances) a dynamic equilibrium where a stable spot-like pattern forms. Turing’s spot pattern has been widely replicated because of its simplicity and ease of implementation. Over the years, this work has been expanded in a variety of fields by Belousov, Prigogine, Zhabotinsky, Mienhart, Gray-Scott, FitzHugh-Nagumo, and many others [9].

Our goal in studying advection–reaction–diffusion models is to create spatio-temporal patterns that can be used for texture synthesis [24] and the visualization of vector fields [23]. We also want to create a system that can be used by chemists in their analysis of reaction-diffusion models, such as those being investigated in [32]. Figure 1 provides examples of some of the patterns formed using reaction-diffusion models that meet these goals. Our research focuses on a class of advection–reaction–diffusion models that can be computed using finite difference techniques, and that can also be solved using relatively simple first and second order numerical integration techniques, such as a forward-Euler or Crank–Nicholson.

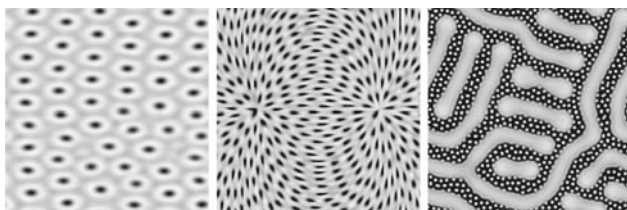
Communicated by M. Rumpf.

A. R. Sanderson (✉) · M. D. Meyer · R. M. Kirby · C. R. Johnson  
Scientific Computing and Imaging Institute,  
University of Utah, Salt Lake City, UT 84112, USA  
e-mail: allen@sci.utah.edu

M. D. Meyer  
e-mail: miriah@sci.utah.edu

R. M. Kirby  
e-mail: kirby@sci.utah.edu

C. R. Johnson  
e-mail: crj@sci.utah.edu



**Fig. 1** Three examples of reaction-diffusion patterns for texture synthesis (*left*), vector field visualization (*center*), and nonlinear chemical dynamics (*right*)

There are numerous characteristics of the advection–reaction–diffusions models that make analysis difficult. One such property is the nonlinearity of the reaction functions, which cause the tuning of the parameter values that drive the models toward stable pattern formation difficult. Another challenging characteristic is the sensitivity of the numerical techniques to these tunable parameters. For instance, what begins as a stable numerical integration may itself become unstable, forcing the researcher to restart the simulation using different integration parameters. Even when a researcher has successfully adjusted all of the necessary parameters, solving the associated PDEs can be time consuming. All of these challenges taken together have led us to seek a system that will allow a researcher to easily change parameters and quickly solve a series of PDEs in order to more effectively ask *what if* questions.

To create an interactive advection–reaction–diffusion system, our research focuses on significantly reducing the PDE solution time. Researchers have traditionally taken advantage of the finite difference discretization to solve PDEs in one of two ways in order to achieve such computational accelerations: either in parallel on multiple processors using the message passing interface (MPI) or parallel compilers, such as F90; or through multi-threaded methods on shared memory architectures.

### 1.1 Graphics processing units for general processing

More recently, researchers seeking computational accelerations have turned to Graphic Processing Units (GPUs). Similar to the math co-processors of yesterday, GPUs are specialized add-on hardware that can be used to accelerate specific graphics-related operations. GPUs are parallelized and currently operate on up to 128 single-instruction-multiple-data (SIMD) instructions (the exact number is proprietary), providing a parallel desktop work environment for users of Linux, OS X, and Windows. Furthermore, the speed increase of each new generation of GPUs is out-pacing Moore’s Law, one standard measure of performance increase of CPUs. These reasons have led GPUs to become popular for general parallel computing [19].

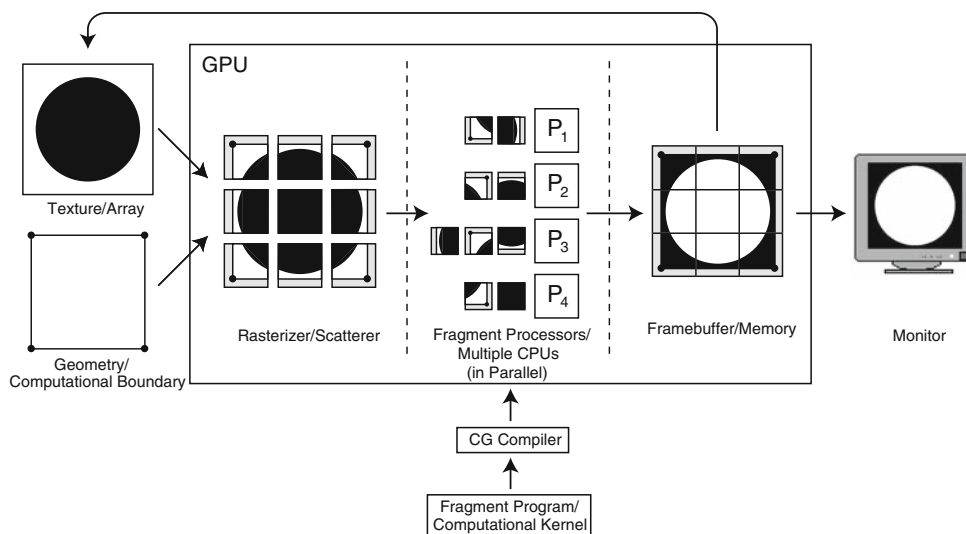
The conveniences and advantages of using a GPU, however, do not come without a cost. GPUs require researchers to think outside their normal paradigms because the hardware is optimized specifically for computer graphics operations rather than for general purpose processing. Thus, researchers using GPUs for general processing (GP-GPU) must understand the graphics processing pipeline (i.e. OpenGL [31]), and how to adapt it for general processing [19].

The GPU further differs from the math co-processor in that it works autonomously from the CPU. As such, the data must be packaged up and specialized computational algorithms written before shipping these pieces off to the GPU for processing. Although there exists high level coding languages that can help in these tasks, such as the OpenGL shading language [4], Cg (C for graphics) [10], and Direct3D’s HLSL [2], a graphics API must still be used for handling the data, such as OpenGL [31] or Direct3D. For our implementation we have chosen to use the OpenGL and Cg APIs because they offer portability to multiple platforms, unlike Direct3D which is available only under Windows. At the same time, there has been considerable research into streaming programming languages for the GPU, such as Brook [1], Scout [21], Sh [20], and Shallows [5] that attempt to hide the graphics pipeline altogether. However, at the time of this writing, these new languages are either not publicly available (Scout), are experimental (Brook and Shallows), or are no longer being supported and being replaced with a commercial product (Sh). There is also NVIDIA’s recently announced compute unified device architecture (CUDA) technology, which provides a “C” like development environment [3].

Another drawback to using the GPU for general purpose processing is the limited memory and the floating point precision currently available on the hardware. Today’s commodity level GPUs have at most 512MB of memory and only support up to 32-bit non-IEEE floating point precision data (64 bit precision has been announced by some manufacturers but is not yet available). While these constraints make the GPU not yet practical for solving large-scale, highly precise engineering simulations, the problems typically investigated by scientists exploring advection–reaction–diffusion models, such as [32], fit well within the current GPU capabilities. These computations provide stable solutions in the Lyapunov-sense [25]—that is, under small amounts of noise the patterns remain the same. Also, advection–reaction–diffusion problems are second-order PDEs, making them prototypical of parabolic (diffusion dominant) and hyperbolic (advection dominant) PDEs.

Thus, advection–reaction–diffusion problems provide an ideal test bed for comparing and contrasting CPU- and GPU-based implementations for real-world scientific problems. The computational experiments described in this paper also provide an indication of the future possibilities for using

**Fig. 2** Layout of the CPU computational pipeline in terms of the GPU graphics pipeline, where data defined in a simple array (e.g. a texture) over a computational boundary (e.g. the geometry) is scattered to the processors (e.g. rasterized) and operated on using a computational kernel (e.g. a fragment program), with the results stored in memory (e.g. the framebuffer). This example shows how a 4 pipe GPU can be used to invert the color of a texture



GPUs for large-scale scientific computing as the evolution of the GPU continues to increase in speed, memory, and precision.

## 1.2 Related work

A review of the literature shows that among the first to explore the use of GP-GPUs for advection were Weiskopf et al. [30], while some of the earliest work computing nonlinear diffusion on a GPU was proposed by Strzodka and Rumpf [26]. Harris et al. [13] presents a method for solving PDEs using GP-GPUs, including a reaction-diffusion system that utilizes an explicit Euler scheme requiring one pass per data dimension. Lefohn et al. [17] builds on this approach for solving PDEs associated with level sets. Kruger et al. [16] and Bolz et al. [7] propose more general matrix solvers, ranging from sparse to conjugate gradient methods, applying the solvers to simple Navier–Stokes simulations and other examples. The closest related work to our own is that of Goodnight et al. [11], who implement a multigrid solver while discussing various implementation strategies, including a cursory comparison of their GPU implementation with a CPU analog. Our work combines aspects of each of these proposed ideas to create PDE solvers for advection–reaction–diffusion systems using GPUs, while at the same time, performing a rigorous one-to-one comparison between the CPU and GPU implementations for scientific applications—these one-to-one comparisons are the main contribution of our work.

## 2 CPU and GPU pipelines

We define the CPU computational pipeline for solving advection–reaction–diffusion problems to be comprised of

five distinct components: *arrays*, *computational boundaries*, *scattering* of data to processors, *computational kernels*, and *memory*. Each of these components can be mapped one-for-one into the GPU graphics pipeline as *textures*, *geometry*, *rasterization*, *fragment programs*, and the *framebuffer*—these components form the basic building blocks for GP-GPU computations. Details on the complete OpenGL graphics pipeline can be found in [31], and further details on GP-GPU basics can be found in [19] and [22].

In the following sections we describe how to implement a simple CPU program on the GPU, where data defined in a simple array (e.g. a texture) with a computational boundary (e.g. the geometry) is scattered to the processors (e.g. rasterized) and operated on using a computational kernel (e.g. a fragment program), with the results stored in memory (e.g. the framebuffer). The pipeline is illustrated in Fig. 2. Included in Fig. 2 is a branch showing where the fragment program is loaded into the pipeline. There are several other minor parts of the graphics pipeline and fragment programs, such the initialization process, that will not be discussed in this paper, but are covered in detail in [10].

### 2.1 Arrays and textures

*Textures* are the main memory data structures for GP-GPU applications, which store the values that would otherwise populate arrays in a CPU implementation. Although traditional graphics applications use textures to store 2D images, these data structures can be generalized as a one, two, or three dimensional array of data. Each element in the array is known as a *texel* (texture element), and each texel can store up to four values (representing the four color elements used when storing an image—red, green, blue, and alpha, also known as RGBA). Each of these values can be stored

as a single byte, or up to a 32-bit floating point value. When accessing texels, *texture coordinates* are used in a similar fashion as array indices in a CPU implementation. Texture coordinates, however, are floating point numbers that have been (typically) normalized from zero to one. As will be discussed, the normalization of the texture coordinates plays an important role in the implementation of the boundary conditions.

For our GP-GPU implementation we choose to use four element (RGBA), 2D floating point textures throughout for reading and writing data, and also for representing the finite difference discretization of the domain.

## 2.2 Computational boundary and geometry

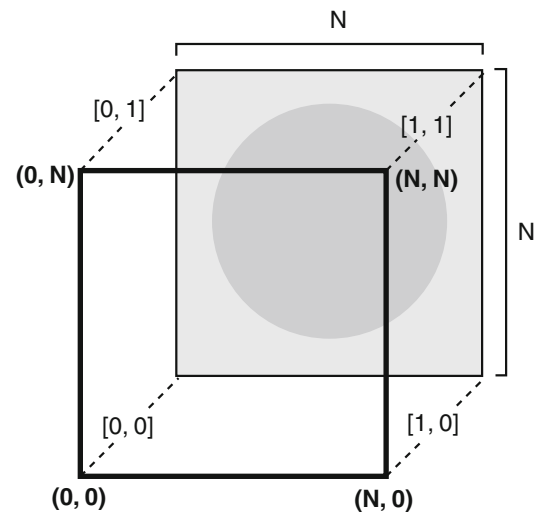
Textures cannot be used alone and must be associated with some *geometry*, just as a painter needs a frame (geometry) on which they can stretch their canvas (texture) for painting. As such, a quadrilateral is analogous to the boundary of the discrete computational domain of a CPU application. Geometry is generated through the specification of *vertices* that define a polygonal object. A mapping that associates parts of the texture with regions of the geometry is then implemented by associating texture coordinates with each vertex. For example, when using all of a 2D texture for computation, the texture can be applied to a quadrilateral of the same size, matching each corner texture coordinate with a vertex of the quadrilateral. Furthermore, a normalized mapping between the quadrilateral and the texture would be from zero to one, as shown in Fig. 3.

This one-to-one mapping is not always used, as it is often desirable to break the computational domain into a series of sub-domains. This approach is useful for applying Neumann boundary conditions when performing relaxation. For example, if the computation was to occur on the lower right quadrant of a texture, the quadrilateral would be one quarter the size with the normalized texture coordinates extending from one half to one, as shown in Fig. 4.

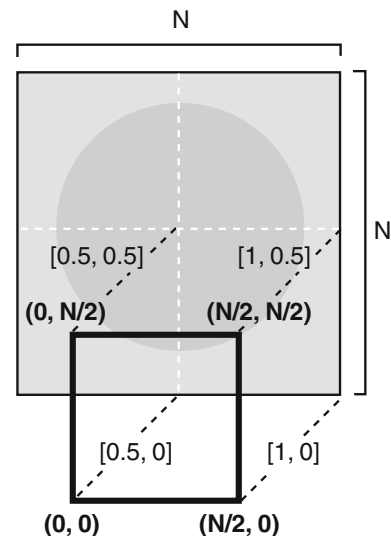
Worth noting in a discussion of the geometry is the use of the GPU's vertex processors, which transform the geometry before associating a texture with it. Although the vertex processors work in parallel, and may in some cases increase the efficiency of a GP-GPU implementation, we have not found their usage to be particularly helpful in our work, and as such only mention them in passing.

## 2.3 Scattering and rasterization

Once the four texture coordinates and geometry have been specified they are next *rasterized* by the GPU. Rasterization interpolates the values across the geometry, producing a set of *fragments*. For example, each fragment is assigned their own set of texture coordinates based on the texture coordi-



**Fig. 3** Example of mapping a texture onto geometry with the same size so that computations occur over the entire texture. The geometry vertices  $(0,0)$ ,  $(N,0)$ ,  $(N,N)$ , and  $(0,N)$  are mapped to the normalized texture coordinates  $[0,0]$ ,  $[1,0]$ ,  $[1,1]$ , and  $[0,1]$ , respectively



**Fig. 4** Example of mapping a texture onto geometry that is one quarter the size so that computations occur only over the lower right quadrant of the texture. The geometry vertices  $(0,0)$ ,  $(N/2,0)$ ,  $(N/2,N/2)$ , and  $(0,N/2)$  are mapped to the normalized texture coordinates  $[0.5,0]$ ,  $[1,0]$ ,  $[1,0.5]$ , and  $[0.5,0.5]$ , respectively

nates initially specified at the four corners. Each fragment represents a single *pixel* (picture element) and includes properties of the geometry such as the 3D position and texture coordinates as above but also the color. The fragments are then scattered to the GPU processors similarly to how data might be scattered to multiple CPUs.

## 2.4 Computational kernels and fragment programs

*Fragment programs* are the algorithmic steps that are performed in parallel on each fragment in a single instruction



multiple data (SIMD) fashion, equivalent to the computational kernels within the inner loops of a CPU program. However, fragment programs have some limitations not inherent to a CPU application. For example, global scope does not exist, thus all data necessary for the computation must be passed to the fragment program in much the same manner one would pass-by-value data to a third party library. Also, fragments have an implicit destination associated with them that cannot be modified. That is, a fragment with array indices  $(i, j)$  will be written to location  $(i, j)$ . This may appear to be a computational limitation, but in practice we have found it is rarely necessary to use different indices for the source and destination.

For our work we utilize NVIDIA's high level shading language Cg (C for graphics) [10]. NVIDIA provides APIs for loading the Cg fragment programs from the main program, as well as a compiler for creating the graphics hardware assembly language. Using a high level language such as Cg greatly aids in the development of GP-GPU applications as the programming model is very similar to the well known C paradigm. An example fragment program, implemented in Cg, can be found in the Appendix.

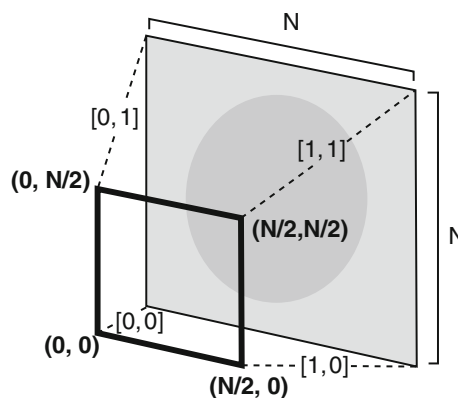
### 2.5 Memory and framebuffer

The output destination of a fragment program is the *framebuffer*. The data stored in the framebuffer can either be displayed on a monitor, or it can be accessed as a texture that has been *attached* to a data structure called a *framebuffer object*—this texture can then be used as the input texture to another fragment program. Unlike CPU memory, however, framebuffer objects do not support *in-place* writing of data (e.g. the source is also the destination). Further, the GPU will not write the results until all fragments have been processed. The implications of this for iterative processes such as explicit solvers and relaxation are discussed later in the paper.

### 2.6 Global scope and reduction

Due to the absence of global scope on the GPU, it is not possible to directly find, for example, the minimum or maximum value of the data stored in a texture—a process that is easily implemented within the computational kernel of a CPU program. This process of reducing the data into a single value, known as *reduction*, can be synthesized on the GPU through a series of steps that gradually contracts the data into a single value. The GPU reduction operation is akin to the restriction operation used in a multigrid implementation.

To obtain different reduction operations—averaging, summation, greatest, least, equal, or Boolean—different filters are specified. On the GPU, these filters are implemented as short fragment programs and used when performing an asymmetrical map between the geometry and a texture. For



**Fig. 5** Example of mapping a  $N \times N$  texture onto smaller geometry, reducing the texture into a result that is half the texture's size. The geometry vertices  $(0,0)$ ,  $(N/2,0)$ ,  $(N/2,N/2)$ , and  $(0,N/2)$  are mapped to the normalized texture coordinates  $[0,0]$ ,  $[1,0]$ ,  $[1,1]$ , and  $[0,1]$ , respectively

instance, if the geometry is half the size of the texture and a  $2 \times 2$  weighted linear average filter is specified in the fragment program, then the output written to the framebuffer will be one quarter the original size (Fig. 5). By repeatedly halving the geometry over multiple passes, the output will eventually be reduced to a single value that represents the filtered value of all the data stored in the original texture. While this description assumes that the domain is square and an integer power of two, this need not be the case. The reduction technique can be used on arbitrarily sized domains with appropriately sized filters.

Boolean operations can also be performed using the GPU's built-in hardware occlusion query. We have chosen to rely on reduction operations instead of this more sophisticated method for performing Boolean operations for two reasons. First, using the occlusion query requires a deeper understanding of the graphics pipeline, including the use of the depth buffer. And second, the occlusion query is single purpose, only doing Boolean operations where as more general reduction operations such as summation and Boolean are needed.

## 3 GPUs for solving PDEs

In the previous section we describe the basic CPU computational pipeline and how it maps to the GPU graphics pipeline. We now discuss our application of the GP-GPU components to solving PDEs used for advection–reaction–diffusion problems.

### 3.1 Problem background

Advection–reaction–diffusion problems are first-order in time and second-order in space PDEs, representative of many

parabolic (diffusion dominant) and hyperbolic (advection dominant) systems. We have chosen to implement on the GPU second-order finite differences in space and two commonly employed time discretization schemes: forward-Euler and a semi-implicit Crank–Nicholson. The reason for choosing these solvers is twofold. First, these discretization methodologies provide adequate results for our advection–reaction–diffusion problems. And second, the properties of each of these solvers are well known. This latter reason allows for a rigorous comparison of the solvers’ implementations on the CPU and the GPU, a task that may be unwieldy with more complicated solvers.

### 3.1.1 Advection–reaction–diffusion basics

Arguably the most popular reaction-diffusion model is the one developed by Turing in 1952 [27]. This model describes the chemical process between two morphogens within a series of cells. Due to instabilities in the system, the morphogens both react and diffuse, changing their concentration within each cell.

Turing described the reaction-diffusion of a two morphogen model as a set of nonlinear PDEs:

$$\frac{\partial u}{\partial t} = F(u, v) + d_u \nabla^2 u \quad (1)$$

$$\frac{\partial v}{\partial t} = G(u, v) + d_v \nabla^2 v. \quad (2)$$

These equations are solved on a two-dimensional domain  $\Omega$  with appropriate initial and boundary conditions, where  $u(x, y, t) : \Omega \times [0, \infty) \rightarrow \mathbb{R}$  and  $v(x, y, t) : \Omega \times [0, \infty) \rightarrow \mathbb{R}$  are the morphogen concentrations;  $F$  and  $G$  are the (non-linear) reaction functions controlling the production rate of  $u$  and  $v$ ; and  $d_u$  and  $d_v$  are the positive diffusion rates.

For the particular chemical problem of interest to Turing [27],  $F$  and  $G$  are defined as:

$$F(u, v) = s(uv - u - \alpha) \quad (3)$$

$$G(u, v) = s(\beta - uv) \quad (4)$$

where  $\alpha$  and  $\beta$  are the decay and growth rate of  $u$  and  $v$  respectively, and  $s$  is the reaction rate. For our applications, we allow the decay and growth rates, as well as the reaction rate, to vary across the domain.

Expanding upon our previous work [23], we consider advection–reaction–diffusion systems of the form:

$$\frac{\partial u}{\partial t} + (\mathbf{a} \cdot \nabla) u = F(u, v) + (\nabla \cdot \sigma_u \nabla) u \quad (5)$$

$$\frac{\partial v}{\partial t} + (\mathbf{a} \cdot \nabla) v = G(u, v) + (\nabla \cdot \sigma_v \nabla) v \quad (6)$$

where  $\mathbf{a} : \Omega \times [0, \infty) \rightarrow \mathbb{R}$  denotes a (possibly) spatially and temporally varying advection velocity, while  $\sigma_u$  and  $\sigma_v$  denote symmetric positive definite, spatially

inhomogeneous, anisotropic, diffusivity tensors for  $u$  and  $v$  respectively. Both  $u$  and  $v$  are clamped to be positive values because of the physical impossibility for the concentration of either morphogen to be negative. For more details on the stability, equilibrium conditions, and other affects related to the particular reaction chosen, see [9].

### 3.1.2 PDE solvers

In what is to follow, we will define everything in terms of the morphogen  $u$  with the tacit assumption that  $v$  will be handled similarly. For the purposes of our discussion, let us assume that our two-dimensional domain of interest,  $\Omega$ , consists of a square with periodic boundary conditions. In this domain, a finite difference grid point,  $x_{ij}$ , lies a distance  $\Delta x$  from neighboring grid points. For many reaction-diffusion models  $\Delta x$  is a dimensionless unit spacing. Let us discretize our morphogens as follows:  $u_{ij}^n = u(x_{ij}, t_n)$  where  $t_{n+1} = t_n + \Delta t$  denotes discretization in time and  $n = 0, \dots, M$  with  $M$  being the total number of time steps, and  $(i, j) = 0, \dots, (N - 1)$  with  $N$  equal to the width of the (square) domain. For notational simplicity, let us define  $\bar{u}^n$  to be an  $N^2 \times 1$  vector containing the values of the morphogens over the grid at time step  $t_n$  where the entries of the vector are given by  $\bar{u}_{i+jN}^n = u_{ij}^n$ .

Let  $A(\bar{u})$  and  $D(\bar{u})$  denote discretized finite difference operators which, when acting upon  $\bar{u}$ , returns a second-order centered approximation of the advection operator and the diffusion operator respectively. For simple uniform isotropic diffusion, the diffusion operator is

$$\begin{aligned} \nabla^2(u)_{i,j} = & (u)_{i-1,j} + (u)_{i+1,j} \\ & + (u)_{i,j-1} + (u)_{i,j+1} - 4(u)_{i,j} \end{aligned} \quad (7)$$

while the advection operator is

$$\nabla(u)_{i,j} = \begin{pmatrix} (u)_{i+1,j} - (u)_{i-1,j} \\ (u)_{i,j+1} - (u)_{i,j-1} \end{pmatrix}. \quad (8)$$

For complete details on diffusion operators using inhomogeneous anisotropic diffusion, see [24].

The (explicit) forward-Euler finite difference system generated by discretizing Eqs. (5) and (6) is defined as:

$$\frac{\bar{u}^{n+1} - \bar{u}^n}{\Delta t} + A(\bar{u}^n) = F(\bar{u}^n, \bar{v}^n) + D(\bar{u}^n) \quad (9)$$

$$\frac{\bar{v}^{n+1} - \bar{v}^n}{\Delta t} + A(\bar{v}^n) = G(\bar{u}^n, \bar{v}^n) + D(\bar{v}^n) \quad (10)$$

where the reaction terms are assumed to be explicit evaluations of the morphogens at the grid points. This scheme is first-order in time.

Equations of this form can be manipulated so that the unknowns at the next time step, namely  $\bar{u}^{n+1}$  and  $\bar{v}^{n+1}$ , can be deduced explicitly from known information at time step

$t_n$ —no system inversion is required. The major advantage of computational systems of this form is the ease of implementation, while their major drawback is that the time step used in the computation is not only dictated by accuracy, but also stability. Depending on the diffusivity of the system, the stability constraint due to the diffusion number may be far more stringent than the accuracy constraint (and is more constraining still than the CFL condition for the advection and reaction terms) [14].

One possible way to alleviate the diffusion number constraint is to use a semi-implicit scheme [15], in which the advection and reaction terms are handled explicitly and the diffusion terms are handled implicitly. Such an approach yields the following system of equations to solve:

$$\frac{\bar{u}^{n+1} - \bar{u}^n}{\Delta t} + A(\bar{u}^n) = F(\bar{u}^n, \bar{v}^n) + (1 - \theta)D(\bar{u}^n) + \theta D(\bar{u}^{n+1}) \tag{11}$$

$$\frac{\bar{v}^{n+1} - \bar{v}^n}{\Delta t} + A(\bar{v}^n) = G(\bar{u}^n, \bar{v}^n) + (1 - \theta)D(\bar{v}^n) + \theta D(\bar{v}^{n+1}). \tag{12}$$

where  $0 \leq \theta \leq 1$ . Given the linearity of the discretized diffusion operator, we can thus re-write the system to be solved as:

$$(\mathbf{I} - \theta \Delta t \mathbf{D})\bar{u}^{n+1} = \bar{u}^n + \Delta t(-A(\bar{u}^n) + F(\bar{u}^n, \bar{v}^n) + (1 - \theta)\mathbf{D}\bar{u}^n) \tag{13}$$

$$(\mathbf{I} - \theta \Delta t \mathbf{D})\bar{v}^{n+1} = \bar{v}^n + \Delta t(-A(\bar{v}^n) + G(\bar{u}^n, \bar{v}^n) + (1 - \theta)\mathbf{D}\bar{v}^n). \tag{14}$$

where  $\mathbf{I}$  denotes the  $N^2 \times N^2$  identity operator and  $\mathbf{D}$  denotes the  $N^2 \times N^2$  linear finite difference diffusion operator. With the choice of  $\theta = 0$ , we regain the explicit forward-Euler scheme. For  $\theta > 0$  we obtain a semi-implicit scheme. Two commonly used values of  $\theta$  are  $\theta = 1$  (corresponding to first-order backward-Euler for viscous terms) and  $\theta = 0.5$  (corresponding to second-order Crank–Nicholson for the viscous terms) [15].

The semi-implicit scheme with  $\theta \geq 0.5$  eliminates the stability constraint due to the diffusion terms at the trade-off of requiring inversion of the linear operator  $(\mathbf{I} - \theta \Delta t \mathbf{D})$ . The GP-GPU is amenable to several types of iterative solution techniques for matrix inversion, such as general relaxation techniques (e.g. Jacobi) and hierarchical techniques (e.g. multigrid).

### 3.2 GPU implementation

Given the previous background we now discuss the specific GP-GPU implementation for solving the PDEs that couple the advection–reaction–diffusion equations.

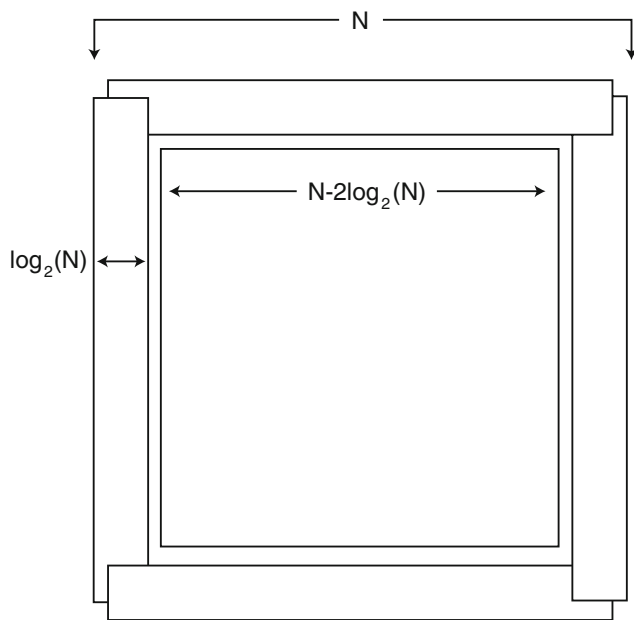
#### 3.2.1 Common components

For the explicit and semi-implicit solvers there are three common components. The first is the use of four element (RGBA) floating point textures that hold the constants associated with advection, reaction, and diffusion for each cell, and are used as input data to the fragment programs. These textures are similar to the arrays one would use in a CPU implementation in that they are used to pass static data to the computational kernel. These textures are referred to as ancillary textures ( $T_{\text{ancil}}$ ).

The second component common to both solvers is the need in the diffusion computation to access not only the current data value, but also the neighboring data values. Additionally, for inhomogeneous anisotropic diffusion it is necessary to also access the neighboring diffusivity terms.

In a CPU implementation, determining the offset positions of the neighbors at each data value can be explicitly calculated, or preferably, retrieved from a precomputed lookup table. This latter approach can be taken with a GPU by using another texture as the lookup table. This texture, however, then must be stored within the limited GPU memory. Computing neighbor offsets could also be done using the GPU’s vertex processors, albeit at the cost of increased programming complexity. An alternative approach that we have taken is to explicitly pass the neighbor offsets into the fragment program as multiple texture coordinates. To pass these coordinates, multiple texture coordinates are specified at each vertex of the geometry, with the rasterizer interpolating these coordinates for each fragment passed into the fragment program. Ideally, the coordinates of all eight surrounding neighbors would be passed, however most GPUs allow at most four sets of texture coordinates to be specified per vertex. To bypass this hardware limitation we use three texture coordinates for specifying the neighbor positions—the lower left neighbor, central coordinate, and the upper right neighbor. These three coordinates can be combined to fully specify all eight neighbors by accessing each component of the texture coordinates individually within a fragment program.

The third common component to both solvers is the implementation of boundary conditions, such as periodic or first order Neumann (zero flux). For a CPU implementation it would be necessary to adjust the boundary neighbor coordinates in the lookup table, or to have conditional branches for the calculations along the boundary. In a GPU implementation, these alterations are avoided by defining a texture to be *repeated* or *clamped*. The repeat definition requires that a modulo operation be performed on the texture coordinates before accessing a texel. Likewise, the clamped definition requires that a floor or ceil operation be performed if the texture coordinates are outside the bounds. Thus, adjusting the texture definition to be repeated or clamped, periodic or



**Fig. 6** Geometry domain using five quadrilaterals to obtain Neumann boundary conditions

Neumann boundary conditions respectively can be performed automatically by the GPU.

For a semi-implicit implementation utilizing Neumann boundary conditions, however, it is well understood that extra relaxation sweeps are required near the boundaries to obtain ideal asymptotic convergence rates [8]. As such, our semi-implicit implementation uses five different computational domains; one for the interior and four for the boundaries, as shown in Fig. 6. The thickness of the Neumann boundaries are dependent on the size of the grid. For the examples in this paper, the thickness is  $\log_2(N)$ , where  $N$  is the grid edge length. Overlap of the boundaries at the corners adds only a negligible cost to the overall computation while aiding in the convergence of the solution.

Solving within a fragment program the PDEs that govern the advection–reaction–diffusion equations for both the explicit and semi-implicit form is very similar to a CPU-based approach with two exceptions—the use of vector arithmetic and swizzling [18] in Cg fragment programs. Vector arithmetic allows for element-wise arithmetic operations to be performed, while swizzling allows the element ordering to be changed during vector operations.

Because the textures and framebuffer are four element (RGBA) texels and pixels respectively, arithmetic operations in the fragment program may be done on a vector basis, greatly simplifying and speeding up the advection and diffusion calculations. Although vectoring is of little use in the reaction calculation where individual components of the texture are used, the swizzle operation can be used to optimize the performance of other operations.

### 3.2.2 Explicit solver

Explicitly solving the advection–reaction–diffusion PDEs [Eqs. (9) and (10)] is straight forward, and requires only one fragment program per iteration. Moreover, because updating the values of  $u$  and  $v$  are independent computations, they may be computed efficiently using the GPU vector arithmetic operations.

The basic loop is as follows: store the required constants (advection, reaction, and diffusion rates) as ancillary texture values ( $T_{\text{ancil}}$ ); store the initial morphogen values in an input texture ( $T_{\text{ping}}$ ) that is attached to the framebuffer; use the fragment program to vector calculate (including clamping to positive values) the advection–reaction–diffusion for both morphogens; store the results in an output texture ( $T_{\text{pong}}$ ) that is also attached to the framebuffer. The output texture is then used as the input texture for the next iteration. The process is repeated for a set number of iterations or until the user determines a stable solution has been reached. A flow diagram of the complete process is shown in Fig. 7, and example code can be found in the Appendix.

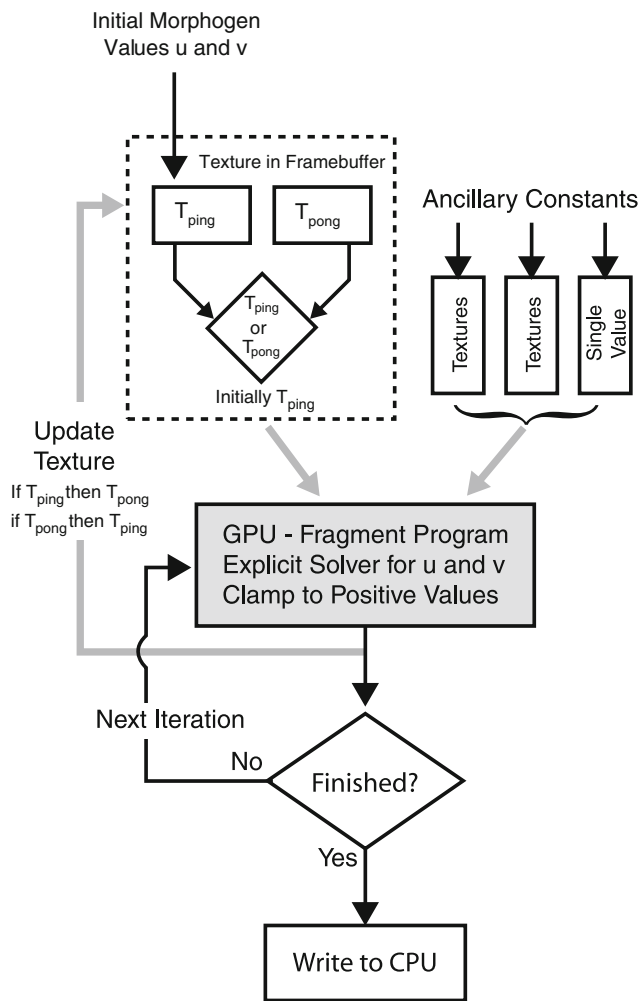
There is one aspect of the loop that must be expanded upon—the use of the input and output textures used to hold the current and next time step morphogen values ( $T_{\text{ping}}$  and  $T_{\text{pong}}$ ). In a CPU implementation, the new morphogen values would be written over the old values once the calculations have been completed, i.e. *in-place* writing. Previously discussed under the definition of Framebuffer Objects, however, an input texture can not also be the output texture. As such, two textures must be used and alternated as the input and the output. This technique is commonly called ping-ponging, and as long as an even number of iterations are performed, the use of these two textures can be hidden from the CPU.

There is no communication between the GPU and the CPU other than to load the initial values and to obtain the final results because the complete advection–reaction–diffusion calculation, including clamping the morphogen values to be positive, can be done in a single fragment program. This lack of interaction greatly speeds up the computations by limiting the communication between the GPU and CPU, which is often a bottleneck due to the limited amount of bandwidth available.

### 3.2.3 Semi-implicit solver

Solving the advection–reaction–diffusion PDEs with an implicit-solver can be, depending on one’s point of view, more interesting, or, more difficult. The GPU is amenable to several types of iterative solution techniques, such as relaxation and hierarchical techniques, all of which require multiple fragment programs. For our implementation we have chosen to use relaxation in combination with a  $L^2_{\text{norm}}$  residual test [6]. As explained in detail below we also utilize a





**Fig. 7** Flow diagram showing the basic steps taken using an explicit solver on the GPU. The process flow is represented with *black arrows* while the inputs are shown in *gray*

$L_{norm}^\infty$  test to limit the number of relaxation steps. Each of these tests require a fragment program for the computation, as well as one for a reduction operation, to obtain the result. All total, five steps are necessary using three fragment programs and two reduction operations per iteration. The three fragment programs are described in the following sections, with a succeeding section that summarizes the semi-implicit computation loop.

*Right hand side*

The first fragment program vector calculates the right-hand-side of Eqs. (13) and (14), storing the results in a texture ( $T_{RHS}$ ) attached to the framebuffer.

*Relaxation*

The second fragment program performs the relaxation to obtain a solution at the next time step. When performing

relaxation, a Gauss-Seidel updating scheme is usually preferred as it will typically converge twice as fast as Jacobi schemes [14]. For a Gauss-Seidel scheme to be implemented efficiently in parallel, it is necessary to have a shared memory architecture. The GPU design, however, does not allow access to the next values until all fragments have been operated on. This limitation can partially be overcome by using a red-black updating scheme, but requires conditional branches in the fragment program to process the red, then black, passes. This situation is further complicated on the GPU because it is not possible to write *in-place*, requiring a ping-pong technique to be integrated. These cumbersome restrictions do not apply to the parallel Jacobi scheme, where intermediate values are not required and an even number of relaxation steps can keep the texture ping-ponging hidden. Thus, the experiments presented in this paper use a Jacobi updating strategy.

Although a preset number of relaxation steps can be performed, it may be stopped early if the high frequency values are damped quickly and do not change significantly with subsequent relaxation. This is based on an  $L_{norm}^\infty$ . When testing for this on a CPU, a global variable would be set if one or more of the values have changed significantly after a relaxation step. The setting of a global variable is not possible with GPUs, requiring the test to be performed through a reduction operation.

To perform the test, an  $L_{norm}^\infty$  is calculated at each texel using the current and previous value, and the difference is then tested against a preset value to see if the value has changed significantly. This is done as a boolean operation in the fragment relaxation program and stored as one of each texel’s four elements. Once a relaxation step is completed, the boolean results are sent to a reduction operation that uses a summation filter to obtain a count of the values that have changed significantly—if the count is non zero the relaxation step is repeated.

The overhead of performing the test after each relaxation step, though, outweighs the benefits of stopping the relaxation early. As such, we have empirically determined that performing the test after every fourth to sixth relaxation step provides a reasonable balance in performance. By not testing after each iteration we are relying upon the behavior of iterative relaxation schemes to quickly remove the high frequency error while taking many iterations to remove the low frequency error.

It should be noted that if the  $L_{norm}^\infty$  passes the residual test is still required insure that a satisfactory solution has been found.

*Residuals and clamping*

The third fragment program calculates the residuals that are used to insure that the relaxation solution, which is an

approximation, is satisfactory. This test is a two step operation. For the first step, the square of the residual of Eqs. (13) and (14) is calculated and stored as one of each texel's four elements in a separate texture ( $T_{resid}$ ). For the second step, the sum of squares of the difference is calculated using the reduction operation.

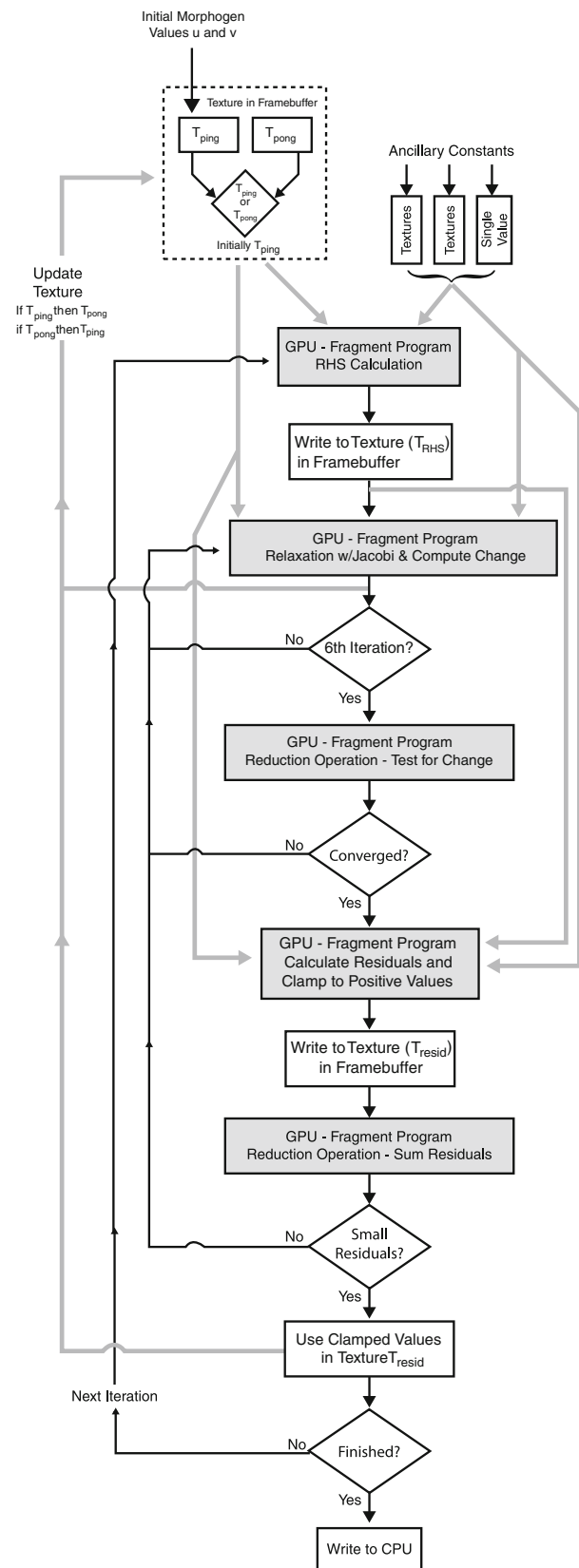
While the residual values are being calculated, the morphogen values are also clamped to positive values and stored as two of each texel's four elements in the residual texture ( $T_{resid}$ ). If the residual test is passed, these *pre-calculated* clamped values can be used in the next iteration. Otherwise, the unclamped values can be further relaxed and re-tested. Though this pre-calculation of the clamped values results in only a slight overall increase in efficiency (<1%), it demonstrates the flexibility of using the GPU for general processing.

The entire process is repeated for a set number of iterations or until the user determines a stable solution has been reached.

### Complete loop

With this general description, the semi-implicit loop is as follows: store the required constants (advection, reaction, and diffusion rates) as ancillary texture values ( $T_{ancil}$ ); store the initial morphogen values in an input texture ( $T_{ping}$ ) that is attached to the framebuffer; vector calculate the RHS for  $u$  and  $v$ , storing the results in a texture ( $T_{RHS}$ ) attached to the framebuffer object. Next, load the current morphogen ( $T_{ping}$ ), RHS ( $T_{RHS}$ ) and the required constants ( $T_{ancil}$ ) textures; use the relaxation fragment program to vector calculate the next value of  $u$  and  $v$  and the  $L_{norm}^{\infty}$  for each; store the results in an output texture ( $T_{pong}$ ) that is also attached to the framebuffer. Check the  $L_{norm}^{\infty}$  after every sixth iteration using the reduction operation. After the relaxation is complete, calculate the residuals and clamped values storing the results in a texture ( $T_{resid}$ ) attached to the framebuffer object. Sum the residuals using a reduction operation. If the residual is too large the relaxation process is repeated until the residual is small enough or is no longer decreasing. If the residual test passes, use the clamped values and repeat the entire process for the next iteration. A flow diagram of the complete process is shown in Fig. 8.

There is one aspect of the process that deserves special attention. In order take advantage of the arithmetic vector processing on a GPU, our implementation requires that both  $u$  and  $v$  be operated on in a lock step manner. That is, if one requires further relaxation or has large residuals it is the same as if both failed. As a result, both morphogens will be processed with the same number of relaxation steps, which in some cases will result in an over relaxation. It is possible, however, to operate on each separately through the use of conditional branches, but the cost of doing so in fragment programs out weighs the cost of the over relaxation



**Fig. 8** Flow diagram showing the basic steps taken using an implicit solver on the GPU. The process flow is represented with *black arrows* while the inputs are shown in *gray*

computations. Conversely, a staggered approach which would be done on a CPU can be used. In this case,  $u$  and  $v$  are stored and operated on separately—in our experience this results in a 25% to 33% increase in computational time without any quantitative difference in the solution. As such, the results presented in this paper utilize the more efficient lock step approach for computing  $u$  and  $v$ .

The above point highlights that when mapping algorithms onto the GPU, implementing the CPU version exactly may not always be the most algorithmically efficient scheme. And for some algorithms, such as a Gauss-Seidel updating scheme, the inefficiencies may preclude its use on the GPU. These examples illustrate how the GPU architecture can often dictate the implementation.

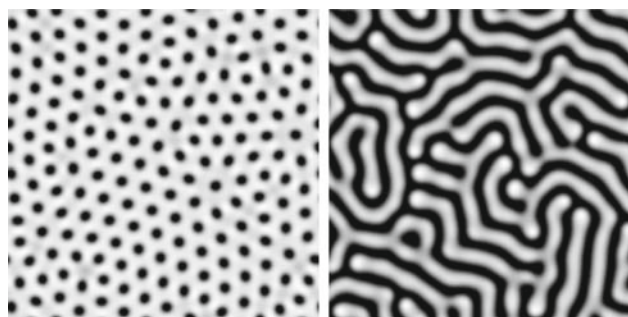
#### 4 Results and discussion

We have used Turing’s reaction-diffusion model, which in its simplest form produces a spot or stripe pattern as shown in Fig. 9, to verify and validate our implementations. The system has been implemented on both the CPU and GPU as explicit and semi-implicit (Crank–Nicholson) Euler solutions with periodic boundary conditions on an Intel Xeon P4 running at 3.4GHz with 2 Gb of RAM and 2 Gb swap, and a NVIDIA GeForce 6800 Ultra graphics card with 256Mb of memory using version 1.0-8178 of the NVIDIA Linux display driver.

For both integration schemes we discretize the grid using a dimensionless unit spacing, which is the norm for reaction-diffusion models. In addition, the largest time step possible with respect to stability limitations was used for each scheme. The diffusion stability bounds the explicit scheme, whereas the reaction stability bounds the semi-implicit scheme.

To facilitate a fair comparison, both the CPU and GPU implementations have the exact same capability. Because the GPU is able to use at most 32-bit floating point precision the CPU version was also limited to machine single precision. We further note that memory organization can effect performance. As such, contiguous memory allocation, which is required for the textures, was used for CPU memory layout.

In order to compare the results it is necessary to ensure that the solutions obtained are the same in all cases, or at least reach a preset stopping criteria typically based upon a difference measure. Quantitatively comparing the solution results, however, is not possible for reaction-diffusion models because the equilibrium reached is dynamic; meaning that although a stable pattern forms, the system continues to change. This problem is further compounded by the differences in the implementation of the floating point standard between the CPU and GPU, which creates slightly different solutions. Thus, we use a criteria based strictly on the number of iterations, such that each solution has the same overall



**Fig. 9** A spot and stripe pattern formed using a Turing reaction–diffusion system

**Table 1** Time, in seconds, for the four different PDE solutions of Turing’s spot pattern, along with the relative speedup for a  $512 \times 512$  grid

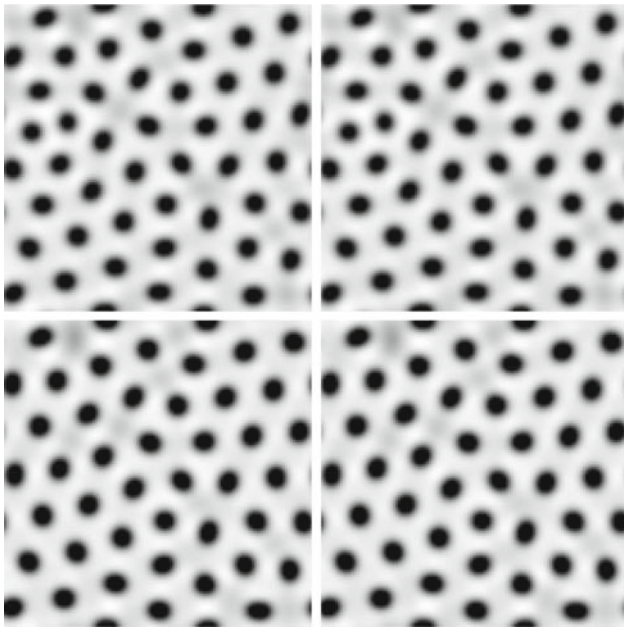
	CPU		GPU
Forward Euler	6,030 s	$\xrightarrow{11.7x}$	516 s
	$\downarrow 3.3x$		$\downarrow 1.4x$
Crank–Nicholson	1,807 s	$\xrightarrow{5.0x}$	362 s

The solutions use a total integration step time of 25,000 s, with an explicit time step of 0.5 s for 50,000 iterations, and an implicit time step of 12.5 s for 2,000 iterations

integration time whether the technique used was explicit or implicit, or performed on a CPU or GPU—see Table 1 for the integration times. In Fig. 10 we show a comparison of the results for the four solution techniques that have each reached a dynamic equilibrium; each of the solutions are equivalent in the Lyapunov-sense. Furthermore, under very close examination is it difficult to visually discern the differences between the CPU and GPU solutions. We thus conclude that each of the solvers have reached an approximately equal solution.

Although the implementations and the compilers are optimized, the times we report are not the absolute best possible. This is because the framework was developed as an application to explore various advection–reaction–diffusion models rather than to specifically compute one model. We believe, however, that even if the computations were streamlined further, the relative speedups would not change significantly. As such, the results give a good indication of the power of using the GPU for general processing on a practical level.

Table 1 shows the relative compute times for both the CPU and GPU implementations using a  $512 \times 512$  grid. For all cases the total integration time is the same—25,000 s—with an explicit time step of 0.5 s for 50,000 iterations, and an implicit time step of 12.5 s for 2,000 iterations. The 0.5 and 12.5 s represent the largest possible time steps for maintaining stability requirements. In the case of the explicit solution, the time step is bound by the diffusion, whereas with the semi-implicit solution, the time step was bound by the reaction.

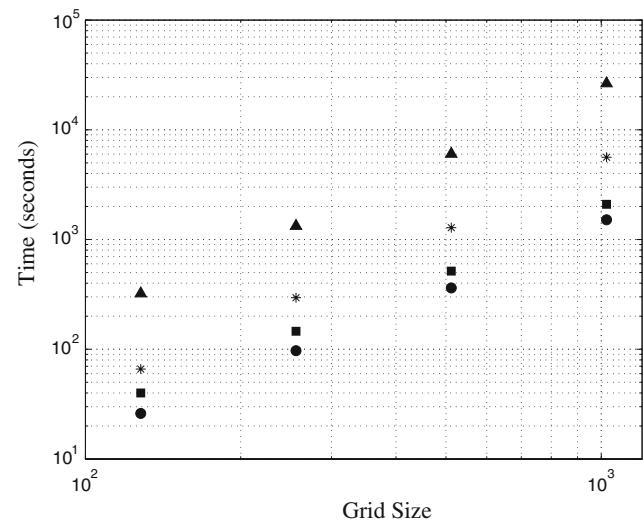


**Fig. 10** A comparison of the solution for Turing's basic spot pattern using (left to right, top to bottom): an explicit CPU technique; an explicit GPU techniques; a semi-implicit CPU technique; and a semi-implicit GPU technique. Qualitatively all of the solutions are approximately equal

As one would expect, the GPU implementations are faster than the CPU implementations, ranging from approximately 5.0–11.7 times faster for semi-implicit and explicit solutions, respectively. This is most evident when comparing the explicit implementations where, unlike the semi-implicit implementations, a single fragment program is used, resulting in less overhead and a greater overall speed up.

When comparing the CPU and GPU explicit and semi-implicit implementations, a greater speed up is realized on the CPU than the GPU, 3.3 versus 1.4, respectively. This difference is attributed to the overhead of using multiple fragment programs and a specialized reduction operation on the GPU. On the CPU there is no need for an explicit reduction operation because there is global scope and such calculations can be done as part of the relaxation and residual operations.

Finally, the speed of the GPU implementations makes it practical to visually study the affects of advection because of the near real time images obtained during the simulation. When visually studying the advection, it is preferable to use the explicit solution because more intermediate views are available for visualization, resulting in smoother motion. For example, when displaying the results after every 50th iteration for a  $256 \times 256$  grid using an explicit GPU implementation, 6.8 frames per second can be obtained. It is possible to get real time frame rates by displaying the results after every 10th iteration, which provides 22 frames per second. The overhead of this smoother visualization process, however, slows



**Fig. 11** Average time in seconds for square grid widths of 128, 256, 512, and 1,024 for four different implementations: forward Euler CPU (triangle); Crank–Nicholson CPU (asterisks); forward Euler GPU (square); Crank–Nicholson GPU (circle)

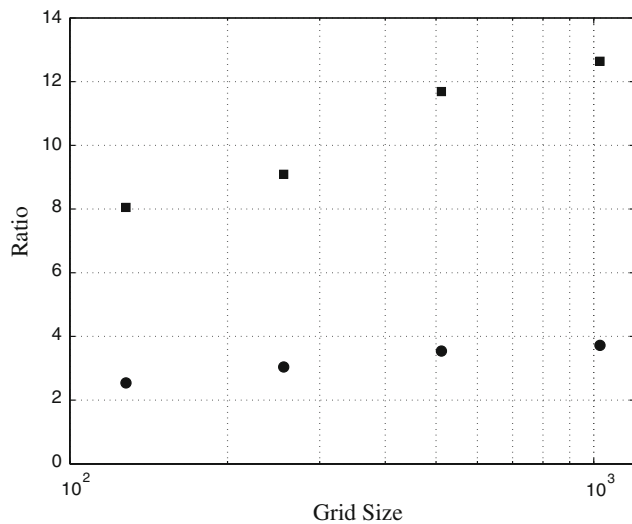
down the system by approximately 1/3 when compared to displaying the results after every 50th iteration.

Another important issue is scalability. GPUs, like other processors, have a limited amount of memory that can be accessed before explicit memory management must be used. Unlike CPUs where there is some flexibility on the amount of memory available, GPUs have only a fixed amount of memory. Thus, for optimal GPU performance it is necessary to work within these strict memory bounds. In Fig. 11 we show the results of the computation time as a function of grid width for both the forward Euler and Crank–Nicholson implementations. The computation time is approximately quadratic with grid width, and large ( $1,024^2$ ) grids are accommodated without a loss in performance.

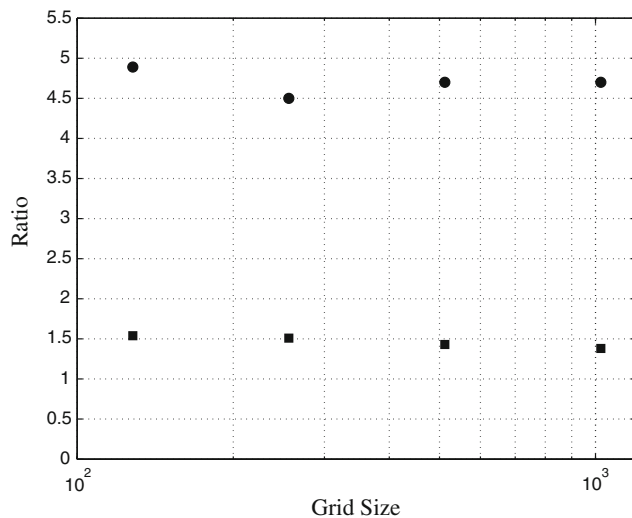
The next comparison is the ratio of the CPU and GPU solution times as a function of the grid width using both explicit and semi-implicit implementations. Shown in Fig. 12, it is readily apparent that there is a consistently greater speed up when using an explicit solution. Furthermore, the speed up ratio of the explicit solution increases with the grid width at a greater rate than the semi-implicit solution due to the increase in the number of relaxations required to smooth the fine detail in the semi-implicit solution over the larger grids. We expect that the use of hierarchical solutions, such as multi-grid, will cause the relative speed ups between the explicit and implicit solutions to remain approximately constant.

Finally, we compare the ratio of the explicit and semi-implicit times as a function of grid width using GPU and CPU implementations as shown in Fig. 13. The relative speed ups are similar across the grid width, with the CPU speed up being greater than the associated GPU speed up. This is again





**Fig. 12** Speed up ratios between CPU and GPU solutions for square grid widths of 128, 256, 512, and 1,024 using an explicit (*square*) and an semi-implicit (*circle*) implementation



**Fig. 13** Speed up ratios between explicit and semi-implicit implementations for square grid widths of 128, 256, 512, and 1,024 using a GPU (*square*) and a CPU (*circle*)

attributed to the need for multiple fragment programs and a separate reduction operation on the GPU when using semi-implicit implementations. In fact, this overhead is so great on the GPU for the semi-implicit solutions that much of the computational benefits are lost. This shows that not only must the problem space *fit* on the GPU, but the implementation must also *align* with the GPU architecture to achieve significant increases in the computation time.

In all of the experiments reported above we have observed similar trends when using Neumann boundary conditions. As such, we conclude that the type of boundary condition adds very little overhead to the overall implementation.

## 5 Conclusions

The use of GPUs for general purpose processing, though still in its infancy, has proven useful for solving the PDEs associated with advection–reaction–diffusion models. We show that both explicit and semi-implicit solutions are possible, and that good speed ups can be achieved, more so with explicit than semi-implicit solutions. Even with their associated programming overhead, GPU implementations have a similar structure to CPU implementations. Though global scope is not directly available with GPU-based implementations, it can be synthesized through other native GPU operations that mimic CPU style restriction operations. Implementations of these types of operations, however, may not be straight forward to researchers without graphics programming knowledge.

Perhaps the most surprising finding is that due to the overhead of multiple fragment programs, the speed-ups most often associated with semi-implicit relaxation schemes are lost when implemented on a GPU. Though faster than CPU based semi-implicit implementations, they are not significantly greater than GPU based explicit solution. As previously noted, this shows that not only must the problem space fit on the GPU, but the implementation must also align with the GPU architecture.

In the work presented here we use basic solvers with 2D finite differences. Finite differences, with its implied neighbor relationships, is straight forward to implement because of the one-to-one mapping to a texture, while the neighbor indexes can be readily pre-calculated. We take advantage of the use of additional texture coordinates to perform this calculation.

We are now left with the question: Is our implementation and associated results indicative of other problems and other solvers? To explore this question, we have begun to investigate a GPU volume solver with finite differences which, when compared to a similar CPU implementation also using an explicit solver, provides a 22–27 times increase in performance. This much larger increase in performance we believe is due the unique memory paging system on the GPU that is not native to, but could be implemented on, the CPU.

It would also be of interest to use a finite element approach that requires textures to store the neighbor indices. If a curved domain was used, additional textures would be needed to store the mapping for each element back to a unit element. Using additional textures is limited only by the number of textures that can be passed, and the available memory on the GPU.

Perhaps not as clear are the results of applying GPU techniques to other, more complicated solvers because of the overhead of using multiple fragment programs and their associated control structures. We envision that GPU implementations of techniques that require global results, such as

an  $L_{\text{norm}}^2$  residual, will generate less speed up over techniques that would require an  $L_{\text{norm}}^\infty$  (which may be able to use the hardware occlusion queries in an asynchronous manner).

Of greater interest is applying this work to other science and engineering problems that can fit within the memory and precision limits of the GPU. While current GPU memory and precision are limited, we expect the continuing evolution of GPU technology (including multiple GPUs per machine and an accumulator that would provide global memory scope as well as the full development of environments like NVIDIA's CUDA) to allow for the application of the techniques presented in this paper to larger computational applications.

Finally we must stress that the technology is changing rapidly. In the course of this research, the authors used three different graphics cards, six versions of the NVIDIA display drivers, three versions of the Cg compiler, and two different off-screen rendering protocols (Pixelbuffers and Framebuffer

Objects). The affect of each change typically increased the speed of the GPU implementations, but not always. This is because the four GPU components presented in the paper are typically not updated in concert. As such, as one component is updated it may take time for the other components to also utilize the newer technology.

**Acknowledgments** This work was supported, in part, by the DOE SciDAC Program and from grants from NSF, NIH, and DOE. The authors wish to thank Martin Berzins and Oren Livne for their insightful discussions on PDEs, Lingfa Yang for his insights on reaction-diffusion systems, and the anonymous reviewers for their many useful suggestions.

## Appendix

Pseudo code that demonstrates the C++ program that is used on the CPU for the explicit solver.

```
// Load the fragment program to be used on the GPU
cgGLBindProgram( _cg_Program );

// Enable the acillary texture
cgGLEnableTextureParameter( _cg_ConstsTex );

// Set the inputs to be the Ping Texture
cgGLSetTextureParameter( _cg_Inputs, _gl_PingTexID );
//Set the drawing (writing) to be the Pong Texture which is in FrambebufferObject1
glDrawBuffer( GL_COLOR_ATTACHMENT1_EXT );
// Do the drawing which invokes the fragment program
glCallList( _gl_dlAll );

// Repeat the last three steps but this time the Pong Texture is the input and
// the Ping Texture is the output.
cgGLSetTextureParameter( _cg_Inputs, _gl_PongTexID );
glDrawBuffer( GL_COLOR_ATTACHMENT0_EXT );
glCallList( _gl_dlAll );
```

A sample fragment program, implemented in Cg, that demonstrates the calculation of the reaction portion of the system on the GPU.

```
void main(float2 texCoord0:TEXCOORD0, // Upper left neighbor texture coordinate
float2 texCoord1 : TEXCOORD1, // Central neighbor texture coordinate
float2 texCoord2 : TEXCOORD2, // Lower right neighbor texture coordinate

uniform sampler2D inputs, // Input texture values
uniform sampler2D consts, // Constant texture values

out float4 oColor : COLOR) // Output value
{
// Get the current value from the input texture.
float4 c = f4tex2D( inputs, texCoord1 );
```

```

// Set the output to the current value.
oColor = c;

// Get constant values for this cell.
float4 c_values = f4tex2D( consts, texCoord1 );

// Calculate the "Reaction" portion using float4 vector operations
// along with swizzling (eg. c.rrbb * c.ggaa).
float4 nonlinear = c.rrbb * c.ggaa * float4(1.0, -1.0, 1.0, -1.0);
float4 linear = c * float4(-1.0, 0.0, -1.0, 0.0);
float4 konst = c_values.rgrg * float4(-1.0, 1.0, -1.0, 1.0);

// Add the reaction to the current value.
oColor += c_values.a * (nonlinear + linear + konst);

// Clamp the values to be positive.
if ( oColor.r < 0.0f ) oColor.r = 0.0f;
if ( oColor.g < 0.0f ) oColor.g = 0.0f;
}

```

## References

1. Brook homepage. <http://graphics.stanford.edu/projects/brookgpu>
2. Microsoft high-level shading language. <http://msdn.microsoft.com/directx/>
3. Nvidia cuda homepage. <http://developer.nvidia.com/object/cuda.html>
4. Opgengl shading language. <http://www.opengl.org/documentation/ogls.html>
5. Shallows homepage. <http://shallows.sourceforge.net/>
6. Barrett, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., Vorst, H.V.der.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM, Philadelphia (1994)
7. Bolz, J., Farmer, I., Grinspun, E., Schroder, P.: Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.* **22**(3), 917–924 (2003)
8. Brandt, A.: Algebraic multigrid theory: the symmetric case. *Appl. Math. Comput.* **19**(1–4), 23–56 (1986)
9. Epstein, I., Pojman, J.: An Introduction to Nonlinear Chemical Dynamics. Oxford University Press, New York (1998)
10. Fernando, R., Kilgard, M.: Cg: The Cg Tutorial. Addison Wesley, New York (2003)
11. Goodnight, N., Wollley, C., Lewin, G., Luebkw, D., Humphreys, G.: A mutligrid solver for boundary value problems using programable graphics hardware. In: *Graphics Hardware 2003*, pp. 1–11 (2003)
12. Gray, P., Scott, S.: Sustained oscillations and other exotic patterns of behaviour in isothermal reactions. *J. Phys. Chem.* **89**(1), 22–32 (1985)
13. Harris, M., Coombe, G., Scheuermann, T., Lastra, A.: Physically-based visual simulation on graphics hardware. In: *Graphics Hardware 2002*, pp. 1–10. ACM Press, New York (2002)
14. Karniadakis, G., Kirby, R.M.: *Parallel Scientific Computing in C++ and MPI*. Cambridge University Press, New York (2003)
15. Karniadakis, G., Sherwin, S.: *Spectral/hp Element Methods for CFD*. Oxford University Press, New York (1999)
16. Kruger, J., Westermann, R.: Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph.* **22**(3), 908–916 (2003)
17. Lefohn, A., Kniss, J., Hansen, C., Whitaker, R.: Interactive deformation and visualization of level set surfaces using graphics hardware. In: *IEEE Visualization*, pp. 75–82 (2003)
18. Mark, W.R., Glanville, R.S., Akeley, K., Kilgard, M.J.: Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.* **22**(3), 896–907 (2003)
19. Mark Pharr, E.: *GPU Gems 2C*. Addison Wesley, New York (2005)
20. McCool, M., Toit, S.D.: *Metaprogramming GPUs with Sh*. A.K. Peters, Natick (2004)
21. McCormick, P.S., Inman, J., ahrens, J.P., Hansen, C., Roh, G.: Scout: a hardware-accelerated system for aunatiatively driven visulization and analysis. In: *Visualization '04: proceedings of the conference on visualization '04*, pp. 1171–1186. IEEE Computer Society, Washington (2004). doi:[10.1109/VIS.2004.25](https://doi.org/10.1109/VIS.2004.25)
22. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krer, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. In: *Eurographics 2005, State of the Art Reports*, pp. 21–51 (2005)
23. Sanderson, A.R., Johnson, C.R., Kirby, R.M.: Display of vector fields using a reaction–diffusion model. In: *Visualization '04: Proceedings of the conference on Visualization '04*, pp. 115–122. IEEE Computer Society, Washington (2004). doi:[10.1109/VIS.2004.25](https://doi.org/10.1109/VIS.2004.25)
24. Sanderson, A.R., Johnson, C.R., Kirby, R.M., Yang, L.: Advanced reaction–diffusion models for texture synthesis. *J. Graph. Tools* **11**(3), 47–71 (2006)
25. Sandri, M.: Numerical calculation of lyapunov exponents. *Math. J.* **6**, 78–84 (1996)
26. Strzodka, R., Rumpf, M.: Nonlinear diffusion in graphics hardware. In: *Proceedings of EG/IEEE TCVG Symposium on Visualization*, pp. 75–84 (2001)

27. Turing, A.: The chemical basis of morphogenesis. *Phil. Trans. R. Soc. Lond.* **B237**, 37–72 (1952)
28. Verwer, J., Hundsdorfer, W., Blom, J.: Numerical time integration for air pollutions. *Surv. Math. Ind.* **10**, 107–174 (2002)
29. Wedge, N., Branicky, M., Cavusoglu, M.: Computationally efficient cardiac bioelectricity models toward whole-heart simulation. In: *Proceedings of International Conference on IEEE Engineering in Medicine and Biology Society*, pp. 3027–3030. IEEE Press, New York (2004)
30. Weiskopf, D., Erlebacher, G., Hopf, M., Ertl, T.: Hardware-accelerated lagrangian-eulerian texture advection for 2d flow visualization. In: *Proceedings of Workshop in Vision, Modeling, and Visualization*, pp. 77–84 (2002)
31. Woo, M., Neider, J., Davis, T., Shreiner, D.: *OpenGL Programming Guide*. Addison Wesley, New York (1999)
32. Yang, L., Dolnik, M., Zhabotinsky, A.M., Epstein, I.R.: Spatial resonances and superposition patterns in a reaction–diffusion model with interacting turing modes. *Phys. Rev. Lett.* **88**(20), 208, 303–1–4 (2002)