GPU Accelerated Numerical Solutions to Chaotic PDEs

J.R. Seaton[a], J.C Sprott[a]

[a]*Department of Physics*
University of Wisconsin - Madison
1150 University Avenue
Madison, Wisconsin 53706

Abstract

In this study, chaotic partial differential equations (PDEs) were numerically solved using a parallel algorithm on graphics processing units (GPU). This new method will aid in our search for simple examples of chaotic PDEs. Computational time using the GPU was compared to other languages such as Matlab and PowerBASIC. The GPU algorithm was optimized using shared memory and a data padding method was analyzed. We report that we have simulated selected chaotic PDE candidates with greater spatial resolution and speed using the GPU.

Keywords: Chaos, Nonlinear Dynamics, Partial Differential Equations, GPU, CUDA, Spatiotemporal Plot, Lyapunov Exponent

1. Introduction

It is well known that chaos exists in systems of ordinary differential equations (ODEs). The typical 3-dimensional examples include the Lorenz system and the Rössler system. However, while chaos in ODEs is a mature and well studied field, the study of chaos in partial differential equations (PDEs) remains rather new and unexplored [1]. This is in part is due to the complexity of the theory and computational time needed to accurately simulate such systems, since PDEs are an infinite set of coupled ODEs. The existence of chaos in PDEs is relevant in the study of turbulence as well as some quantum phenomena. However, proving a PDE exhibits chaos with specific boundary conditions remains a difficult problem and in most cases nearly impossible analytically. Therefore, computer based simulations currently provide the best environment to study such systems. Since there are currently few proven examples of chaotic PDEs, finding more examples will aid in development of a better mathematical theory as well as providing new models for physical events. A previous study involving a search for the simplest chaotic PDE concluded that the Kuramoto-Sivashinsky equation was the simplest example of a chaotic PDE after a 16 month search on a standard computer [2]. However, this search was limited by the size of the search space and in its ability to simulate possible chaotic PDEs with high spatial resolution.

Graphics processing units (GPUs) are an emerging technology which provide the parallel processing for a fraction of the cost of the equivalent computer network. Physics simulations and differential equations have previously been studied using GPU technology resulting in great acceleration[3, 4, 5, 6, 7]. With application to solving PDEs, the GPU is especially good because the system of ODEs can be integrated over time in parallel. Using the parallel processing ability of GPUs to numerically solve PDEs, it would be possible to find more examples of chaotic PDEs that could have previously been missed. In this paper, we present a method to numerically solve PDEs and implement this method in different ways on a GPU to utilize its parallel processing ability. We then present a few examples of PDEs that may exhibit chaos as well as a simulation of the Kuramoto-Sivashinsky equation, which is known to be chaotic [8]. The results are benchmarked and compared to the same simulation done on an standard desktop computer.

2. The GPU Framework

The GPU has been around since the very first personal computers, but until recently, its primary function was limited to processing data for output on a video monitor. However, with NVIDIA Compute Unified Device Architecture (CUDA), the GPU can be used as a general purpose computational device. Some benefits of GPU programming with CUDA include:

- Eliminating overhead from graphics API for non-graphics based applications
- Reducing global memory bandwidth limitation by utilizing on chip memory
- Allowing a standard and more flexible method of writing to memory
- Automatic scalability for multiple GPU systems using the grid architecture

Computation on the GPU using CUDA involves processing threads in parallel organized in blocks of grids. The CPU can execute a kernel for one grid at a time. The grid is broken into an array of 1, 2, or 3 dimensional blocks each with a unique block identification. Furthermore, each block is split into an array of threads with a unique thread identification. A GPU multiprocessor (MP) can execute a single block of threads at a time. In a grid with blocks of the same dimension, processing can be done using all MPs in parallel. Each MP has a Single Instruction, Multiple Data (SIMD) architecture, which means each thread processor in a MP executes the same operation, but on different data in memory. When a block of threads is executed on a MP, it is first split into SIMD groups of equal size called warps. The MP can then process a warp of threads in 4 clock cycles. The specifications for an NVIDIA Tesla C870, which was used in this study, can be seen in Table 1.

Table 1: NVIDIA Tesla C870 CUDA 1.0 specifications

Multiprocessors	16
Processors per multiprocessor	8
Clock speed	1.35 GHz
Warp Size	32
Maximum active blocks	8
Maximum active warps	24
Maximum active threads	768
Maximum block dimension	512x512x64
Maximum grid dimension	65535x65535x65535

Threads in a block have access to many types of memory including global, textured, constant, shared, and local. However, for simulating PDEs, we will only be concerned with global, shared, and constant memories. Global memory has read/write permissions from both the host computer and the GPU. It is often very large (many gigabytes) depending on the type of GPU, however, its access by each multiprocessor has limited memory bandwidth. Shared and constant memory have much higher bandwidths than global memory because they are located in each multiprocessor. Constant memory is written by the host and read by the GPU through a constant cacher while shared memory can only be read-/written to by the GPU. More information about the GPU architecture can be found in NVIDIA's programming guide [9].

In addition to memory bandwidth limits, to optimize an algorithm, one must be concerned with the number of clock cycles certain floating point operations will use as seen in Table 2.

Table 2: Operations and Clock Cycles

Operation	Number of clock cycles
Floating point add/subtract	4 cycles
24-bit multiplication	4 cycles
32-bit multiplication	16 cycles
Floating point square root	32 cycles
Floating point division	36 cycles
Modulus operator	60 cycles

3. Methods

3.1. The Method of Lines

The GPU PDE solver was generated using the method of lines [10]. First, an approximation is made for the spatial derivatives using a five-point stencil with a central difference. We will utilize the notation $U_x = \frac{\partial U}{\partial x}$, $U_{xx} = \frac{\partial^2 U}{\partial x^2}$, etc. This gives the following approximation [11],

$$U_x = \frac{-U(x+2k,t) + 8U(x+k,t) - 8U(x-k,t) + U(x-2k,t)}{12k} + O(k^5) \quad (1)$$

$$U_{xx} = \frac{-U(x+2k,t) + 16U(x+k,t) - 30U(x,t) + 16U(x-k,t) - U(x-2k,t)}{12k^2} + O(k^4) \quad (2)$$

$$U_{xxx} = \frac{U(x+2k,t) - 2U(x+k,t) + 2U(x-k,t) - U(x-2k,t)}{2k^3} + O(k^2) \quad (3)$$

$$U_{xxxx} = \frac{U(x+2k,t) - 4U(x+k,t) + 6U(x,t) - 4U(x-k,t) + U(x-2k,t)}{k^4} + O(k^2) \quad (4)$$

An example of how the five point stencil utilizes parallel threads and memory accessing can be seen in Figure 1. The result is a set of coupled ODEs for each spatial grid line with spatial step size k. The 4th order Runge-Kutta method can be used to integrate the ODEs,

$$U(x, t+h) = U(x,t) + \frac{1}{6}h(a_1 + 2a_2 + 2a_3 + a_4) \quad (5)$$

where

$$a_1 = \frac{\partial}{\partial t}U(x,t) \quad (6)$$

$$a_2 = \frac{\partial}{\partial t}U(x + 0.5a_1, t + 0.5h) \quad (7)$$

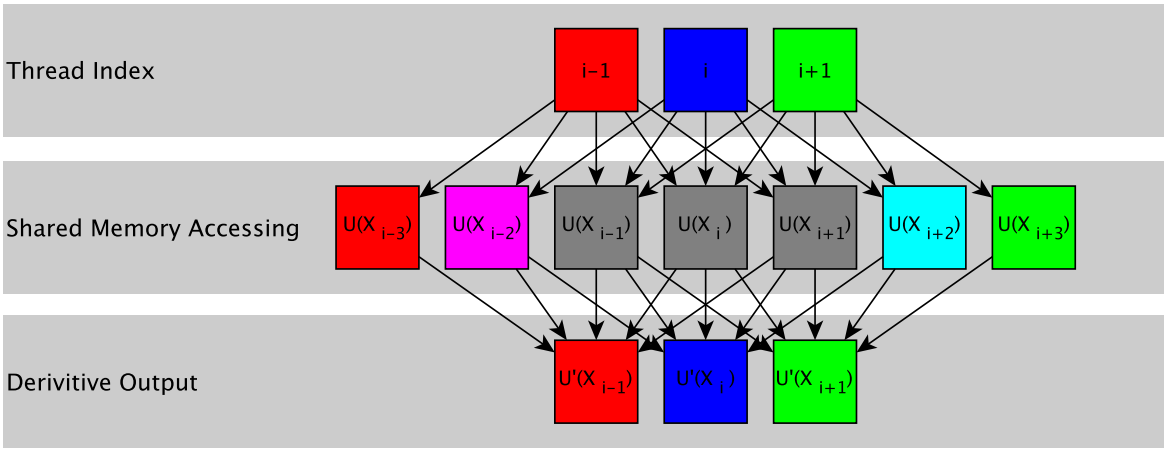

Figure 1: Five-point stencil on the GPU utilizing the gather and scatter memory access methods.

$$a_3 = \frac{\partial}{\partial t} U(x + 0.5a_2, t + 0.5h) \tag{8}$$

$$a_4 = \frac{\partial}{\partial t} U(x + 0.5a_3, t + h) \tag{9}$$

for a temporal step size h.

3.2. Simulated PDEs and Initial Conditions

Candidate chaotic PDEs were selected from a previous work [12] as seen in Table 3. They are organized from PD1 to PD13 based on the algebraic complexity of the system [12]. To generate a comparison of the processing time of the GPU versus the CPU, each PDE was simulated with L=100 and a spatial dimension of 256 coupled ODEs. Periodic boundary condition were used for all cases such that $U(0, t) = U(256, t)$. The calculations were benchmarked by measuring the time for simulating each PDE for one million iterations with a fixed time step $h = 0.001$.

To determine whether a PDE exhibited chaos, the largest Lyapunov exponent (LLE) was calculated using the CPU. This was done by perturbing the initial condition and measuring any exponential growth in the solution for each time iteration. For each step, the perturbed system was scaled by the size of its initial perturbation from the original solution [6]. The average of the log of the maximum expansion was taken in the limit as time approaches infinity. If the LLE was convergent and positive, the system was determined to be chaotic.

3.3. Optimizing the Method of Lines for the GPU

An overview of how the method of lines was implemented on the GPU can be seen in Figure 2. To determine the fastest method for simulating the PDEs, a number of different algorithms were used on the GPU.

Table 3: Selected chaotic PDE candidates and initial conditions [12]

Name	Equation	Initial Condition
PD1	$U_t = UU_x$	$0.1sin\left(\frac{2\pi x}{L}\right) - 0.1$
PD2	$U_t = UU_{xxx}$	$0.1 - 0.1sin\left(\frac{8\pi x}{L}\right)$
PD3	$U_t = (U-1)U_x$	$sin\left(\frac{2\pi x}{L}\right)$
PD4	$U_t = (U+1)U_{xxx}$	$sin\left(\frac{4\pi x}{L}\right)$
PD5	$U_t = (U+1)U_x - 0.1U_{xxx}$	$0.25sin\left(\frac{4\pi x}{L}\right)$
PD6	$U_t = (1 - U_{xx} - 2U)U_x$	$0.04sin\left(\frac{2\pi x}{L}\right) - 0.55$
PD7	$U_t = (0.05U_{xxxx} - U^2)U_x$	$2 + 0.2sin\left(\frac{16\pi x}{L}\right)$
PD8	$U_t = -UU_x - U_{xx} - U_{xxxx}$	$sin\left(\frac{2\pi x}{L}\right)$
PD9	$U_t = (U_{xxx} - U_{xx}^2)U$	$0.1sin\left(\frac{16\pi x}{L}\right)$
PD10	$U_t = -U^2U_x - U_{xx} - U_{xxxx}$	$1 + sin\left(\frac{18\pi x}{L}\right)$
PD11	$U_t = -U^3U_x - U_{xx} - U_{xxxx}$	$sin\left(\frac{2\pi x}{L}\right)$
PD12	$U_t = -UU_x - U_x - U_{xx} - U_{xxxx}$	$sin\left(\frac{2\pi x}{L}\right)$
PD13	$U_t = 1 + (1 - U^3U_{xxx})U_{xxx}$	$sin\left(\frac{16\pi x}{L}\right)$

- Method 1
  In this method, the initial condition is generated by the host CPU and passed to the GPU through global memory. Each ODE is processed by a different thread using one block. Modular operators are used to access data with periodic boundary conditions. Data is written and read using only global memory.

- Method 2
  This method is the same as in method 1, however, data is loaded and stored in shared memory and

```
#define N 256
__constant__ int L = 100;
__constant__ float k = L/N;
__constant__ float dt = 0.001;
__shared__ float Uo[N];
__shared__ float a1[N];

int dim = blockDim.x*gridDim.x;
int i0 = blockDim.x*blockIdx.x + threadIdx.
    x;

//copy from global to shared memory
for(int i=i0; i<N; i+=dim){
  Uo[i] = Initial_Condition[i];
}

//calculate a1
for(int i=i0; i<N; i+=dim){
  float U_m2 = Uo[((i+ N-2)%N)];
  float U_m1 = Uo[((i+N-1)%N)];
  float U_0  = Uo[i];
  float U_p1 = Uo[((i+N+1)%N)];
  float U_p2 = Uo[((i+N+2)%N)];

  a1[i] = Derivative(U_m2, U_m1, U_0, U_p1,
      U_p2, k);
}
```

Figure 3: Example of finding a1 for the Runge-Kutta integration without data padding.

then written back to global memory. A sample of this method can be seen in Figure 3.

- Method 3
  In NVIDIA's Programming Guide it is suggested that, "modulo operations are particularly costly and should be avoided if possible" [9]. Therefore, in this method, the initial condition is padded to eliminate the use of the modulus operator for data accessing. Data is loaded and stored in shared memory and then written back to global memory. The source code for this method is shown in Figure 4 and a diagram of how the data is padded and process through the integration is shown in Figure 5.

These methods were implemented using Matlab, PowerBASIC 6 Console Compiler, and C++ with NVIDIA CUDA and run on a computer with the following specifications: Dual Intel E5420 Xeon CPUs at 2.50 GHz with 4 GB DDR2 RAM at 800 MHz and a NVIDIA Tesla C870.

4. Results and Discussion

4.1. Kuramoto-Sivashinsky

The Kuramoto-Sivashinsky equation (PD8) is a PDE that is often used in numerical simulation of turbulence

```
#define N 256
__constant__ int L = 100;
__constant__ float k = L/N;
__constant__ float dt = 0.001;
__shared__ float Uo[N+4];
__shared__ float a1[N+4];

int dim = blockDim.x*gridDim.x;
int i0 = blockDim.x*blockIdx.x + threadIdx.
    x;

//copy from global to shared memory
for(int i=i0; i<N; i+=dim){
  Uo[i+2] = Initial_Condition[i];
}

//reassign outer array indices
Uo[0] = Uo[N+1];
Uo[1] = Uo[N+2];
Uo[N+2] = Uo[2];
Uo[N+3] = Uo[3];

//calculate a1
for(int i=i0; i<N; i+=dim){
  float U_m2 = Uo[i];
  float U_m1 = Uo[i+1];
  float U_0  = Uo[i+2];
  float U_p1 = Uo[i+3];
  float U_p2 = Uo[i+4];

  a1[i+2] = Derivative(U_m2, U_m1, U_0,
      U_p1, U_p2, k);
}
```

Figure 4: Example of finding a1 for the Runge-Kutta integration using the data padding method.

as well as describing wave processes in active and dissipative systems. This equation was simulated using a time step of $h = 0.015$ and with 128 ODEs. A typical spatiotemporal graph demonstrating chaos for PD8 can be seen in Figure 6. The GPU results agree with those performed on the host CPU. The LLE for the PD8 was determined to be positive based on the calculation done on the CPU, which indicates the system exhibits chaos.

4.2. PD2 and PD13

PDE candidates, PD2 and PDE13, were simulated with a spatial dimension of 128 shown in Figures 7 and 8, respectively. The GPU was used to increase the spatial dimension of these systems without a significant loss in speed. Using more ODEs to simulate PD2 and PD13 indicates that these systems are not chaotic in the spatially continuous limit. This is supported by the strong correlation between the number of ODEs used and the smallest wavelength calculated by a spatial Fourier transform. The smallest wavelengths for PD2 and PD13 decrease inversely with the number of ODEs. If an infinite number of ODEs were used, the small wavelength oscillation would shrink to zero, preventing the onset of chaos. These systems are,

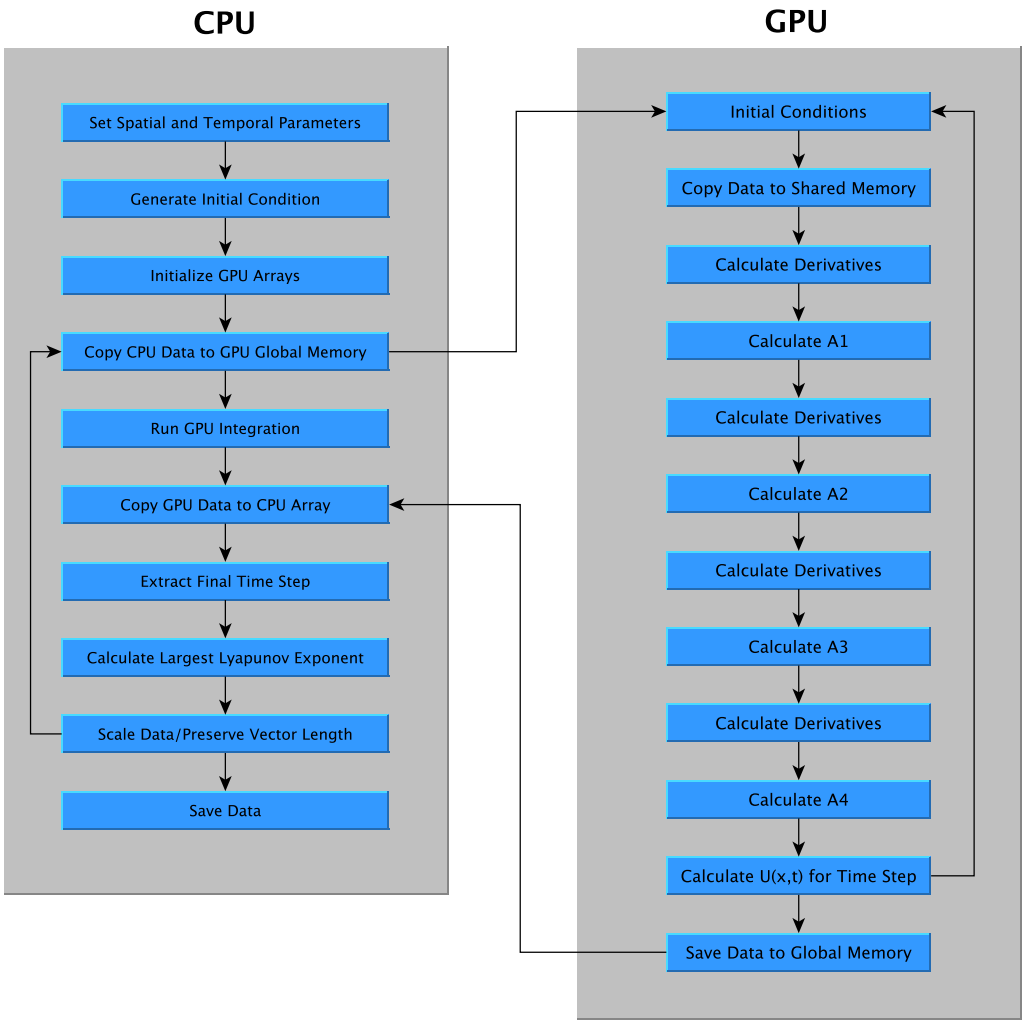

Figure 2: Block diagram of the host CPU and GPU operation

however, examples of numeric chaos as indicated by their positive LLE.

4.3. Lyapunov Exponents

The LLE is important in determining whether a system is chaotic. The LLEs for the systems described were all calculated to be positive, which indicates they are chaotic. In order to accurately determine the LLE, a small perturbation is needed from the initial condition, which can approach the limits of single precision floating point numbers on the GPU. Also, the perturbed system must be routinely scaled such that it maintains the same vector magnitude as the unperturbed system, which requires a serial addition and square root. For these reasons, the LLE could not be efficiently calculated on the GPU. Instead, data was passed to the CPU for the LLE calculation. However, this was subject to global bandwidth limitations which reduced the calculation speed. Since the LLE is only important for determining if a system is chaotic, it is not a necessary calculation each iteration. For implementing a search for new chaotic systems, the LLE could be calculated only at specific times. However, chaos can exist in very small parameter windows of a PDE and if the LLE is not calculated often enough, it is possible to dismiss the system. The answer for the tradeoff between computational time and characterizing chaos is not obvious and highly dependent on the PDEs being analyzed.

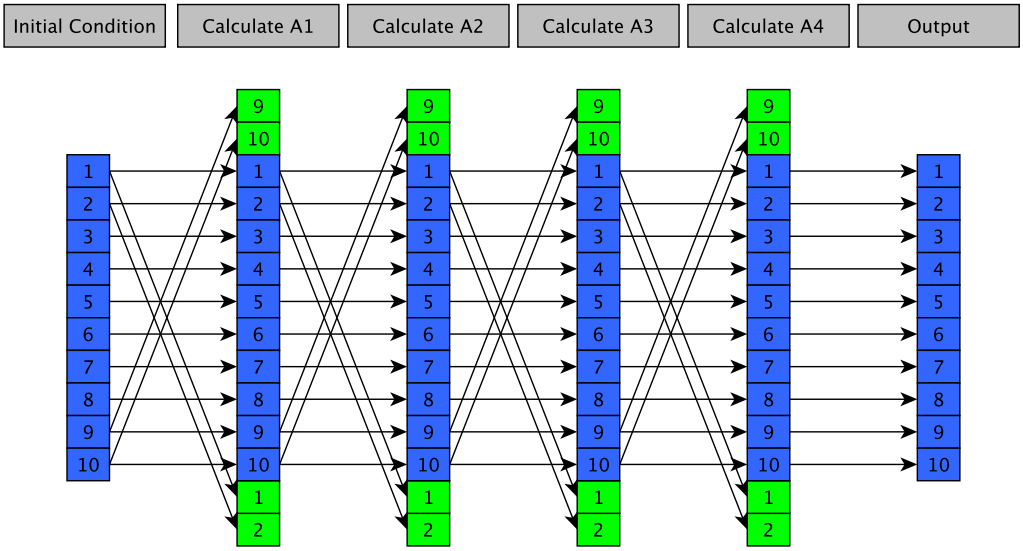

Figure 5: Data padding method

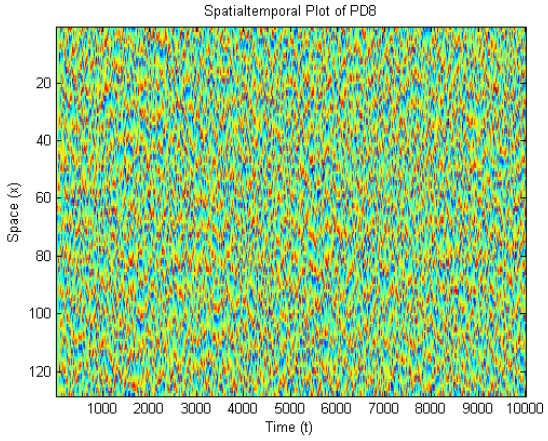

Figure 6: Simulation of PD8 using 128 coupled ODEs

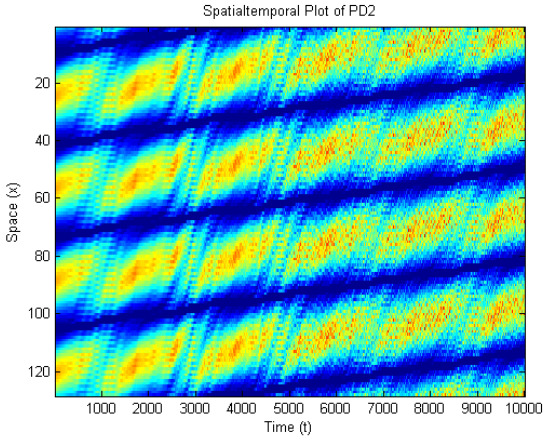

Figure 7: PD2 simulated using 128 ODEs.

4.4. Data Padding and Computational Time

Using the data padding method as shown in Figure 5 resulted in a maximum speedup of 62.42% using 128 ODEs on the CPU shown in Table 4. The data padding speedup was shown to increase with an increasing number of ODEs used in the simulation. This is because the ratio between the latency of the modulus operator and latency of the array reassignment improves with increasing array size. However, when implemented on the GPU, data padding resulted in a loss of speed. As shown in Table 5, method 3 was slower than method 2 despite the elimination of modular operations through data padding. This indicates that the latency in copying data to the edges of an array overpowers any benefit gained from eliminating modular arithmetic on the GPU. Utilizing shared memory in methods 2 and 3 resulted in an increase in speed over method 1 because of increased memory bandwidth. However, the maximum speed of the GPU could not be obtained because the gather and scatter memory accessing inherent in the five-point stencil resulted in bank conflicts between adjacent threads.

Table 4: Effect of data padding on processing time using Matlab.

Num. of ODEs:	L	k	Without Padding	With Padding	Speedup	Iterations/Sec
16	10	0.0001	0.6094	0.5000	21.88%	20000
32	100	0.0001	1.0625	0.8281	28.31%	12075
64	100	0.0001	2.0781	1.3281	56.47%	7529
128	100	0.0001	3.9844	2.4531	62.42%	4076
256	100	0.0001	7.5156	4.7188	59.27%	2119
512	100	0.0001	14.8438	9.3438	58.86%	1070
1024	200	0.0001	29.7031	18.9063	57.11%	528

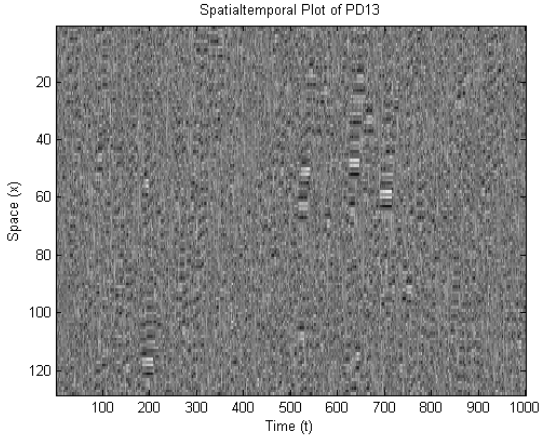

Figure 8: PD13 simulated using 128 ODEs.

Table 5: Benchmark results comparing Methods 1-3 as previously described for PD8 using 256 coupled ODEs, L=100, t=0 to t=10000, h=0.001 measured in iterations per second.

Method:	1	2	3
Itr/sec:	33222	181623	160984

4.5. GPU Benchmarks and Speedup

The GPU based PDE solver could generate spatiotemporal plots of the selected PDEs faster than previous methods using only serial CPU operations. In Table 6, a comparison of iterations per second for the fastest GPU algorithm is compared to two other programming languages, Matlab and PowerBASIC 6 Console Compiler. Using method 2, the GPU was 85.7x faster than Matlab while only 9.5x faster than PowerBASIC. However, our results agree with the results obtained from similar studies using a finite difference method to solve differential equations [7, 13]. Through our study, we were able to identify some important considerations to maximize the utility of the GPU for numerically solving PDEs.

Table 6: Benchmark results for PD8 using 256 coupled ODEs, L=100, t=0 to t=10000, h=0.001. Comparison between Matlab, PowerBASIC, and the optimized GPU simulation in iterations per second.

Method:	Matlab	PowerBASIC	GPU Opt.
Itr/sec:	2119	19104	181623

Important factors for optimizing speed:

- Maximize calculation time on GPU - Transferring data between the GPU and host computer will encounter latency and is limited by the global memory bandwidth. This becomes important when determining when to check the Lyapunov exponent of a system.
- Utilize GPU shared memory - GPU shared memory has much smaller latencies than global memory, but has a limited size, which is a problem with large systems of ODEs [14].
- Choose the number of ODEs as a multiple of the GPU block size - Maximum speedups were obtained using a number of ODEs in multiples of 16, which maximizes the number of calculations performed per warp. Also, increasing the number of ODEs did not proportionally reduce the computational time as was previously shown [7].
- Data padding or data redundancy, while efficient on the CPU, proved to be detrimental to the computational time on the GPU.

5. Conclusion

Our results indicate that some previously suspected chaotic PDEs are examples of numeric chaos and will not hold with decreasing spatial step size. The use of a GPU greatly decreases the processing time to integrate the selected PDEs. Data padding proved to be beneficial on the CPU, but slowed the GPU calculation. Calculating the LLE was more efficient on the CPU than the

GPU. Implementing a search for chaos would require a delicate balance between calculating the LLE and computational speed.

References

[1] Li, Y. C., Proceedings of the 4th International Congress of Chinese Mathematicians **3** (2007) 110.

[2] Brummit, C. D. and Sprott, J. C., Physics Letters A **373** (2009) 2717.

[3] Rostrup, S. and Sterck, H. D., Computer Physics Communications **181** (2010) 2164.

[4] Sainiok, J., Computer Physics Communications **181** (2010) 906.

[5] Januszewski, M. and Kostur, M., Computer Physics Communications **181** (2010) 183.

[6] Sprott, J. C., *Chaos and Time-Series Analysis*, Oxford Univ. Press, 2003.

[7] Alcaraz-Pelegrina, J. and RodrÌguez-GarcÌa, P., Computer Physics Communications **182** (2011) 1414 .

[8] Sivashinsky, G. I., Lettere Al Nuavo Cimento **27** (1980) 504.

[9] *NVIDIA CUDA Compute Unified Device Architecture Programming Guid Version 1.0*, 2007.

[10] Sadiku, M. N. O. and Obiozor, C. N., International Journal of Electrical Engineering Education **37** (2000) 282.

[11] Abramowitz, M. and Stegun, I. A., *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, Dover Publications, tenth edition, 1972, Table 25.2.

[12] Sprott, J. C., *Elegant Chaos*, World Scientific Publishing Limited, 2010.

[13] Eglo, D., High performance finite difference pde solvers on gpus, Technical report, QuantAlea GmbH, 2010.

[14] Klingbeil, G., Erban, R., Giles, M., and Maini., P. K., IEEE Transactions on Parallel and Distributed Systems **X** (2010) 1.