Volume xx (200y), Number z, pp. 1–10

Benchmarking and Implementation of Probability-Based Simulations on Programmable Graphics Cards

Stanimire Tomov, Michael McGuigan, Robert Bennett, Gordon Smith, John Spiletic

Information Technology Division, Brookhaven National Laboratory, Bldg. 515, Upton, NY 11973
tomov@bnl.gov, mcguigan@bnl.gov, robertb@bnl.gov, smith3@bnl.gov, spiletic@bnl.gov

Abstract

The latest Graphics Processing Units (GPUs) are reported to reach up to 200 *billion floating point operations per second (*200 *Gflops* [19]*) and to have price performance of* 0.1 *cents per M flop. These facts raise great interest in the plausibility of extending the GPUs' use to non-graphics applications, in particular numerical simulations on structured grids (lattice). In this paper we (1) review previous work on using GPUs for non-graphics applications, (2) implement probability-based simulations on the GPU, namely the Ising and percolation models, (3) implement vector operation benchmarks for the GPU, and finally (4) compare the CPU's and GPU's performance. Original contribution of this work is implementing Monte Carlo type simulations on the GPU. Such simulations have a wide area of applications. They are computationally intensive and, as we show in the paper, lend themselves naturally to implementation on GPUs, therefore allowing us to better use the GPU's computational power and speedup the computation. A general conclusion from the results obtained is that moving computations from the CPU to the GPU is feasible, yielding good time and price performance, for certain lattice computations. Preliminary results also show that it is feasible to use them in parallel.*

Categories and Subject Descriptors (according to ACM CCS): I.6.3 [Computing Methodologies]: Applications; I.3.1 [Computing Methodologies]: Graphics processors; B.8.2 [Hardware]: Performance Analysis.

1. Introduction

There are several factors that motivated us to investigate the plausibility of using programmable GPUs for numerical simulations on structured grids. These factors are the GPUs'

- high flops count,
- compatible price performance, and
- better than the CPU rate of performance increase over time.

To be specific, nVidia's NV30 graphics card is advertised to have a theoretical operation count of 200 Gflops (see the NV30 preview [19] or Table 1, where we summarize the results of a GPU-CPU comparison in performing graphics related operations). Taking into account the price of the NV30 graphics card we get the low 0.1 cents per M flop. Also, the current development tendency shows a stable doubling of the GPU's performance every six months, compared to doubling the CPU's performance for every 18 months (a fact valid for the last 25 years, known as *Moore's law*).

Review of the literature on using GPUs for non-graphics applications (subsection 1.1) shows that scientists report high speedups of the GPU compared to the CPU for low precision operations. Several of the non-graphics applications that make use of the recently available 32-bit floating point arithmetic report much lower speedups. Often advertisements will report "extremely" high performances and be vague about the precision of the computations. For example, the advertised 200 Gflops in [19] can not be achieved for 32-bit floating point operations (Figure 1, Right gives the NV30 graphics card specifications). To get a better understanding of the GPU capabilities for scientific computations we decided to develop benchmarking software for the GPU's 32-bit floating point operations.

In the rest of the paper we will refer to flins as 32-bit floating point GPU assembly instructions per second. Such instructions perform operations on 4D vectors of floating point numbers, so we will refer to flops as flips times four. It will

be clear from the context if we refer to a particular arithmetic operation.

Problem size	Frames per second using	
	OpenGL (GPU)	Mesa (CPU)
11,540	189	8
47,636	52	1.71
193,556	13	0.44
780,308	3.28	0.12

Table 1. GPU *vs* CPU in rendering polygons. The GPU (Quadro2 Pro) is approximately 30 times faster than the CPU (Pentium III, 1 GHz) in rendering polygonal data of various sizes.

Probability-based models, discussed in section 2, have a wide area of applications. They are computationally intensive and lend themselves naturally to implementation on GPUs, as we show in the paper.

1.1. Literature review

The use of graphics hardware for non-graphics applications is becoming increasingly popular. A few examples of such uses are matrix-matrix multiplication [9], visual simulations of boiling, convection, and reaction-diffusion processes [6], nonlinear diffusion [17], multigrid solver [1, 5], Lattice Boltzmann Method (LBM) [12], fast Fourier transform (FFT) [14], and 3D convolution [7]. For a more complete list and more information on general purpose computations that make use of the GPU see the GPGPU's homepage: `http://www.gpgpu.org/`.

In all the cases the non-graphics computations are expressed in terms of appropriate graphics operations. These graphics operations are executed on the graphics card and the results are used to interpret the results of the original non-graphics computations. Up until recently the output of the graphics operations was constrained to integers, which was a serious obstacle for a meaningful use of many of the non-graphics algorithms developed for graphics hardware. For example, M. Rumpf and R. Strzodka [17] used 12-bit arithmetic (InfiniteReality2 graphics card) for nonlinear diffusion problems, E. Larsen and D. McAllister [9] used 8-bit arithmetic (GeForce3 graphics card) for matrix-matrix multiplications, etc. The low accuracy of the graphics cards computations encouraged research in complicated software techniques to increase the accuracy. See for example the range scaling and range separation techniques developed in [12]. Even when 16-bit floating point precision became available (in GeForce4), it was not uniformly provided throughout the graphics pipeline. This was the reason why C. Thompson's et al. [20] general-purpose vector operations framework was not able to retrieve the computational results without loss of precision.

Currently, graphics cards like the nVidia Quadro FX 1000, known also as NV30, support 32-bit floating point operations throughout the entire graphics pipeline. Features like programmable vertex and fragment shaders (see section 1.2) allow software developers to easily modify the graphics pipeline, which is used in most of the non-graphics applications. The programmability also becomes easier since more alternatives to assembly programming become available. One example is the high level language Cg [15].

By porting non-graphics applications to the GPU, scientists try to achieve computational speedup or upload some of the computations from the CPU by using the GPU as coprocessor. Currently significant speedups of GPU vs CPU are reported if the GPU performs low precision computations. Depending on the configuration and the low precision used (often 8-bit) scientists report speedups in the range of $25-30$ times, and as high as 60 times. Advertisements that make claims such as that GeForce 4 GPUs are capable of 1.2 trillion operations/s or that a supercomputer made out of PlayStations is capable of 0.5 trillion operations/sec [11] are based on certain types of low precision graphics operations. Our experience shows that currently certain general purpose applications that use 32-bit floating point arithmetic on NV30 can be $2-6$ (for the configuration specified) times faster than their equivalent running on Pentium 4, 2.8 GHz. This was confirmed in our applications and benchmarks, the multigrid solver in [1], the shaders' speeds using 32-bit floating point arithmetic from `http://www.cgshaders.org`, etc.

1.2. Programmable graphics cards

Old graphics cards had a fixed function graphics pipeline, a schematic view of which is given on Figure 1, Up. The operations in the dashed rectangle were configured on a very low level and were practically impossible to change by software developers. In August, 1999 nVidia released the GeForce 256 graphics card, which allowed a certain degree of programmability of its pipeline. In February, 2001 nVidia released the GeForce3 GPU, which is considered to be the first fully programmable GPU. Here fully programmable means that developers were able to provide their own transformations and lighting (T & L) operations (vertex shaders) to be performed on the vertices (by the Programmable vertex processor) and their own pixel shaders to determine the final pixels color (executed on the Programmable fragment processor). Both the vertex and pixel shaders are small programs, which when enabled, replace the corresponding fixed function pipeline. The programs get executed automatically for every vertex/pixel and can change their attributes. Originally the vertex and pixel shaders had to be written in assembly. The constantly increasing functionality provided by the graphics cards allows more complex shaders to be written.

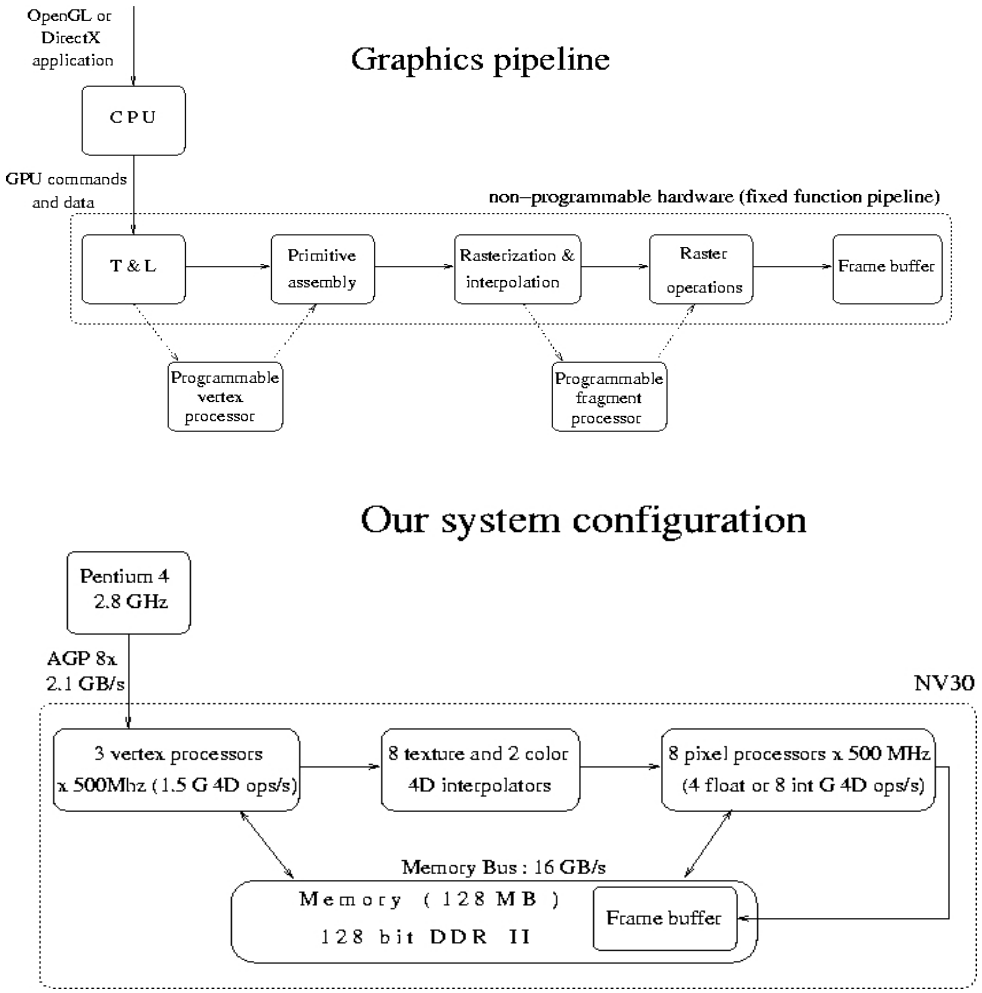

Figure 1: *Up: Graphics pipeline. The dashed rectangle marks the fixed function graphics pipeline (with T&L standing for transformation and lighting). Additional hardware, programmable vertex and fragment processors, available in newer graphics cards provide the developers with opportunity to change the fixed function pipeline. Down: Our system configuration and CPU/GPU performance specifications (Pentium 4 CPU and NV30 GPU).*

To simplify the implementation of such shaders nVidia recently released a high level shader language, called Cg [10, 15]. Cg stands for "C for graphics" since it has syntax similar to C.

2. Probability-based simulations

In this section we briefly describe the models that we implemented on the GPU, namely the Ising (Section 2.2) and the percolation (Section 2.3) models. Both methods are considered to be of Monte Carlo type (Section 2.1).

2.1. Monte Carlo simulations

Monte Carlo (MC) methods are used in the simulation of a variety of phenomena in physics, finance, chemistry, etc. MC simulations are based on probability statistics and use random numbers. The name derives from the famous Monte Carlo resort and is associated with roulette as a simple way of generating random numbers.

A classical example of using MC methods is to compute the area of a circle. First the circle is circumscribed by a square, then random locations within the square are generated, and finally

$$\frac{\text{circle area}}{\text{square area}} \approx \frac{\text{locations within the circle}}{\text{number of generated locations}},$$

where the approximation greatly depends on the number of locations generated and the "quality" of the random locations generator. Our aim is to use graphics cards for the fast generation of extremely large amounts of random numbers, and to model various Monte Carlo type simulations. To be specific, we would like to achieve performance close to the theoretically possible maximal performance, which is 16 Gflops for the fragment processors of the NV30 graphics card.

A problem of general interest is the computation of expected values. For example, assume we have a system that can be in any of its states S_i, $i = 1,...,N$ with known probabilities $P(S_i)$. Also, assume a quantity of interest F is computable for any of the states. Then, the expected value for F, denoted by $E(F)$, is given by

$$E(F) = \sum_{i=1}^{N} F(S_i)P(S_i). \quad (1)$$

The Ising model, described in Section 2.2, is a MC method for computing such expected values. The difficulty in computing $E(F)$ is when N becomes large. If we consider a 2D system of particles, say on a 1024×1024 lattice, and every particle is modeled to have k possible states, then N is of order k^{1024^2}.

The accuracy of the Monte Carlo simulations, as mentioned above, depends on the quality of the random number generator. Computer programs generate pseudo-random numbers. For the applications that we consider we used a linear congruential type generator (LCG). The form is

$$R(n) = (a * R(n-1) + b) \mod N.$$

Fixing a starting value $R(0)$, called seed, uniquely determines the numbers generated. LCGs are fast, easy to compute, and reasonably accurate. Furthermore, they lead to uniformly distributed random numbers and are the most frequently used. The LCGs are well understood and studied. One has to be careful in the choice of the constants and the seed. Undesirable patterns may occur in applications that consider n-tuples of numbers generated by LCGs (see [13]).

2.2. Ising model

The Ising model, a simplified model for magnets, was introduced by Wilhelm Lenz in 1920 and further developed later by his student Ernst Ising. The model is on a lattice, in our case two dimensional. A "spin", pointing up or down, is associated with every cell of the lattice. The spin corresponds to the orientation of electrons in the magnet's atoms. There are two opposing physics principles that are incorporated in the Ising model: (1) minimization of the system's energy, achieved by spins pointing in one direction, and (2) entropy maximization, or randomness, achieved by spins pointing in different directions. The Ising model uses temperature to couple these opposing principles (see an illustration on Figure 2). There are many variations of the Ising model and its implementation (see [3] and the literature cited there). The computational model that we used is described as follows.

We want to be able to compute various expected values (equation 1), such as expected magnetization and expected energy. To compute expected magnetization, $F(S_i)$ from equation (1) is the magnetization of state S_i, defined as the number of spins pointing up minus the number of spins pointing down. To compute expected energy, $F(S_i)$ is the energy of state S_i

$$\mathrm{E}(S_i) = -\sum_{<j,k>} S_i(j)S_i(k),$$

where the sum is over all lattice edges with $< j,k >$ being the edge connecting nearest neighbor sites j and k, $S_i(j)$ is the spin for site j of state S_i with values 1, for spin pointing for example up, or -1, for spin pointing down.

The idea is not to compute the quantity described in equation (1) exactly, since many of the states are of very low probability, but to evolve the system into "more probable" states and get the expected value as the average of several such "more probable" states. The user inputs an absolute temperature of interest T in Kelvin, and probability $p \in [0,1]$ for spins pointing up. Using this probability we generate a random initial state with spins pointing up with probability p. The procedure for evolving from this initial state into "more probable" states is described in the following paragraph. The theoretical justification of methods dealing with similar sequences evolving from state to state, based on certain probability decisions, is related to the so-called *Markov chains* [4].

The lattice is colored in a checkerboard manner. We define a sweep as a pass through all white or all black sites. This is done so that the order in which the sites are processed does not matter. We start consecutive black and white sweeps. At every site we make a decision whether to flip its spin based on the procedure

1. Denote the present state as S, and the state with flipped spin at the current site as S'.
2. Compute $\triangle \mathrm{E} \equiv \mathrm{E}(S') - \mathrm{E}(S)$.
3. If $\triangle \mathrm{E} < 0$ accept S' as the new state.
4. If $\triangle \mathrm{E} \geq 0$, generate a random number $R \in [0,1]$, and accept S' as the new state if

$$R \leq \frac{P(S')}{P(S)} = e^{-\triangle \mathrm{E}/(kT)},$$

otherwise the state remains S. In the last formula the probability $P(S)$ for a state S is given by the Boltzmann probability distribution function

$$P(S) = \frac{e^{-\mathrm{E}(S)/(kT)}}{\sum_{i=1}^{N} e^{-\mathrm{E}(S_i)/(kT)}},$$

where k is a constant, known as the Boltzmann's constant.

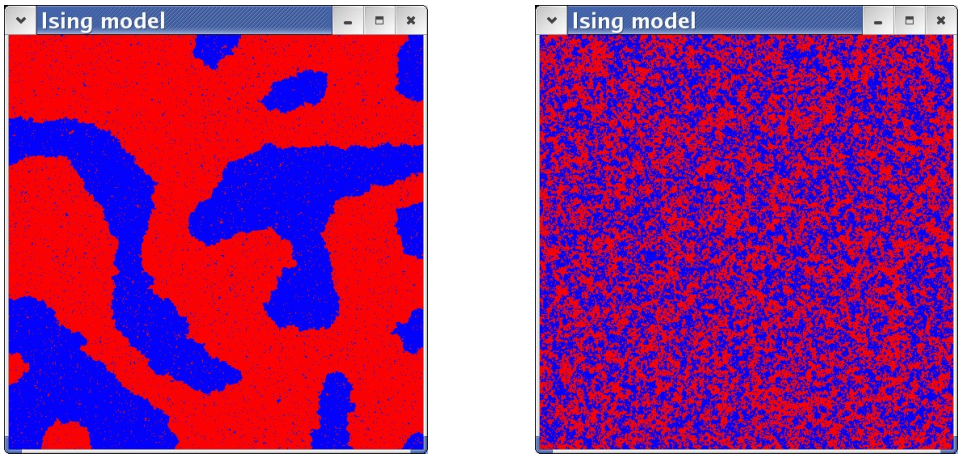

Figure 2: *Ising model visualization on lattices of size* 512×512*. The red/blue regions correspond to spins pointing up/down. Observe the spin clustering (related to energy minimization) for low input temperature (Left) and the randomness (related to entropy maximization) for higher input temperatures (Right).*

A standard Monte Carlo implementation of the Ising model would involve a random traversing through different sites of the lattice, and flipping or not the spin depending on the outcome of the above procedure. The implementation that we describe is influenced by one of M. Creutz's [3] simulations of the Ising model. He uses the checkerboard sweeps in a fully deterministic spin flipping dynamic, which does not require generation of high quality random numbers and yields one order of magnitude faster execution than conventional Monte Carlo implementations. For proof of concept in benchmarking the GPU we use the algorithm described, which involves a richer variety of mathematical operations. The checkerboard sweeps are crucial for the efficient use of the GPU, since the GPU computations are performed on passes over the entire lattice. Simultaneous update on all the sites would not be a simulation of the Ising model, as shown in [21].

2.3. Percolation model

The percolation model is a model for studying disordered media. It was first studied by Broadbent and Hemmercley[2] in 1957. Since then percolation type models have been used in the study of various phenomena, such as the spread of diseases, flow in porous media, forest fire propagation, phase transitions, clustering, etc. The general scenario is when a medium is modeled as an interconnected set of vertices (sites). Values modeling the media and the phenomena of interest are associated with the sites and the connections between them. Usually these values are modeling disordered media with a certain probability distribution, and are obtained from random number generators. A point (or points) of "invasion", such as the starting point of fire, disease, etc, is given, and based on the values in the sites and the connections, a probabilistic action for the invasion is taken. Of particular interest is to determine:

1. Media modeling threshold after which there exists a spanning (often called percolating) cluster. This is an interconnected set of invaded sites that spans from one end of the medium to another.
2. Relations between different media models and time to reach steady state invasion or percolation cluster.

Using the basics of the percolation, as described above, one can derive methods of substantial complexity. For example, R. Saskin et al. [18] presented a novel approach for the clustering of gene expression patterns using percolation: gene expressions are modeled as (1) sites containing m measurements each, and (2) connections between all pairs of sites. The connections have weights which represent the similarity between the gene expressions. Based on the mutual connectivity of the gene expressions, they come up with a probabilistic percolation model to cluster the data.

Here, for proof of concept in benchmarking the GPU, we implemented a simple percolation model with application to diffusion through porous media (see Figure 3). The porous media is modeled on a two dimensional lattice. First, we specify porosity as a probability P of the site being a pore. Then we go through all the sites and at each site (1) generate a random number $R \in [0,1]$, and (2) consider the site "pore" if $R \leq P$, or solid otherwise. Finally, we consider an invasion point, initialize a cluster to be the invasion point, and start the process of spanning the initial cluster through the porous space.

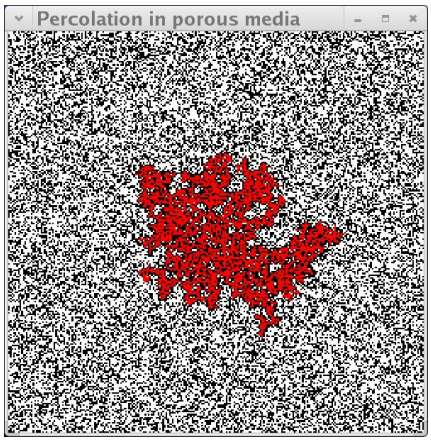

Figure 3: *Percolation model visualization. Seen is a snapshot of fluid spreading in 2D porous media (modeled on a lattice of size* 512×512*, porosity* 0.6*).*

3. Implementation details

The implementations of the models described are Cg fragment programs, which are being invoked (and executed on the GPU) by OpenGL application running on the CPU. To do the OpenGL - fragment programs binding we used the OpenGL Cg run-time functions described in the Cg toolkit user's manual [15]. The programming model is described in the following paragraph.

The OpenGL application repeatedly executes the iterative procedure:

- Prepare the input for the GPU pipeline;
- If the GPU executes a non-fixed function pipeline specify the vertex or fragment programs to be used;
- If needed, read back the graphics pipeline output (or part of it).

This programming model is standard. It is very often related to the so-called *Dynamic texturing*, where on the first step one creates a texture T, then a GPU fragment program uses T in rendering an image in an off-screen buffer (called p-buffer), and finally T is updated from the resulting image.

To be more specific, we have used `GL_TEXTURE_RECTANGLE_NV` textures, an extension to the 2D texture, which allows the texture dimensions not to be a power of 2. To initialize such textures from the main memory we use `glTexSubImage2D(GL_TEXTURE_RECTANGLE_NV, ... )`. To replace (copy) rectangular image results from the p-buffer to `GL_TEXTURE_RECTANGLE_NV` textures we use `glCopyTexSubImage2D(GL_TEXTURE_RECTANGLE_NV, ...)`, which is entirely done on the graphics card. We used the floating point p-buffer that is distributed with the Cg toolkit which is available on the Cg homepage [16]. For our fragment programs we used fp30 profile with options set by the `cgGLSetOptimalOptions` function.

The times in the benchmarking programs were measured with the `gettimeofday` function. Since the OpenGL calls are asynchronously executed we used `glFinish` to enforce OpenGL requests completion before measuring their final execution time. The time measures were used to determine the GPU's performance.

We implemented the Ising model in 2D and 3D. In order to efficiently do the black and white sweeps (see Section 2.2) through a 2D checkerboard lattice we represent the lattice with 2 textures: one for the white and one for the black sites. This is needed for the efficient execution of the Ising model since implementation based on only one lattice will have `if` statements in the fragment program to distinguish black/white sites, and consequently be slow (see Section 5). Figure 4 demonstrates our representation of a 2D vector with two textures. The extension from 2D to 3D is straightforward. The planes in the z axes of the lattice are represented by pairs of textures (for the black and white sites), as in 2D. We have two fragment programs: one to process the white sites and one for the black. As in 2D we do consecutive black and white sweeps. A sweep trough the white sites for example is implemented as a loop over the white textures, which are processed by the corresponding fragment program with arguments the neighboring black textures.

We also note that having the data structures described allows us to easily produce a real-time volume visualization of the 3D simulation results. Since the data (the textures) reside on the GPU its visualization will have very little overhead to be visualized. We simply go through the textures from back to front, based on the viewing direction, and map them to the planes (rectangles ordered in the z axes) that they correspondingly represent (see Figure 5).

The reduction operation over a vector is non-trivial to implement. This operation is used to compute global sums (total energy in our case), dot products, etc. We implemented it by defining textures of decreasing resolution. Then the reduction is done consecutively in a loop from the highest resolution texture to texture of size 1×1. A high resolution texture is mapped to its lower resolution texture. In the mapping process the values in the pixels that get mapped to a single pixel are reduced to a single value by fragment program.

4. Benchmarking the GPU

We wrote several fragment programs to benchmark the performance of basic floating point vector operations on the GPU. The results were compared with those on the CPU. The specifications of the system on which we applied the benchmark programs are given in Figure 1, Down. We perform vector operations, where the vectors are represented by 2D textures. Here we included computational results for tex-

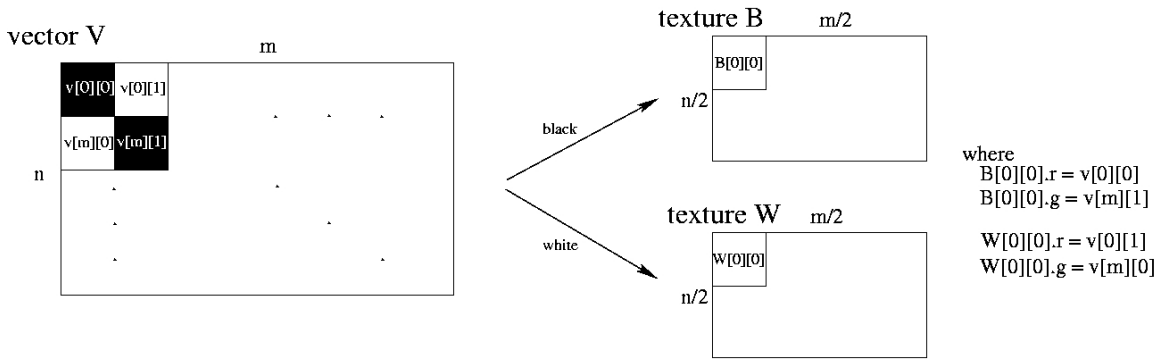

Figure 4: *Representation of a checkerboard lattice, correspondingly a 2D vector v of size $m \times n$, with two textures W and B accounting for the white and black sites.*

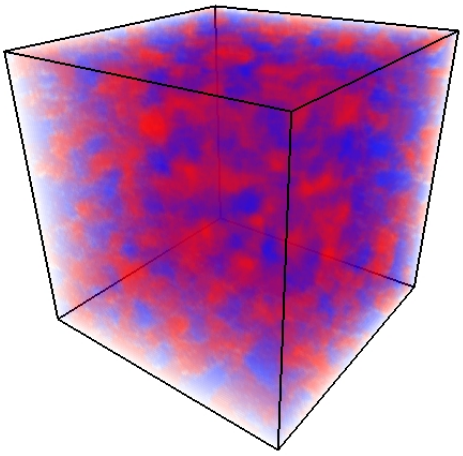

Figure 5: *A snapshot from the real-time volume visualization of the 3D Ising simulation results. Textures representing the computational domain are mapped with prescribed transparency from back to front, based on the viewing direction, to the planes (rectangles ordered in the z axes) that they correspondingly represent.*

tures of sizes 256×256 and 512×512 (with corresponding vector sizes equal to texture sizes times 4).

Table 2 gives the GPU performance for different floating point vector operations. Table 3 is the corresponding CPU performance.

The performance is measured as explained in section 3. We performed our computations and measures on Linux Red Hat 9 operating system. The computations in Tables 2 and 3, and sections 5 and 6 are done using Linux driver display 4363 from April, 2003.

The operations measured are in the following forms $a = c$ (for the $=$ operation), $a+ = c$ (for the basic arithmetic operations), and $a = sin(a)$ (for the *sin*, *cos*, *log*, and *exp* operations), where a is a vector represented with a 2D texture and c is a constant. The $\sum$ operation denotes global sum. The $a = c$ operation (Table 2) is present in all of the operations considered. Its execution time includes overheads related to the graphics pipeline and time for transferring data (reading and writing to the local GPU memory of four 32-bit floating point values for each lattice vertex). Having the time for this operation allows as to exclude it from the other

operation	Lattice size (not necessary power of 2)	
	256×256	512×512
=	0.00063	0.0024
+ , - , *, /	0.00073	0.0027
cos, *sin*	0.00089	0.0034
log, *exp*	0.00109	0.0039
$\sum$	0.00191	0.0086

Table 2. Time in seconds for different floating point vector operations on the GPU (NV30). The vector sizes are lattice size times 4.

operation	Lattice size	
	256×256	512×512
+ , - , *, /, $\sum$	0.0011	0.0046
cos , *sin*	0.0540	0.0650
log, *exp*	0.0609	0.1100

Table 3. Time in seconds for different floating point vector operations on the CPU (2.8 GHz Pentium 4). The vector sizes are lattice size times 4.

	Lattice size (not necessary power of 2)				
	128×128	256×256	512×512	1024×1024	2048×2048
GPU sec/frame	0.0007	0.0027	0.011	0.043	0.19
CPU sec/frame	0.0020	0.0069	0.028	0.116	0.55

Table 4. GPU (NV30) and CPU (2.8 GHz Pentium 4) performances in processing a single step of the 2D Ising model.

operations and get a better idea about their performance. For example, adding a basic arithmetic operation to a fragment program will increase the execution time on a lattice of size 512×512 with 0.0003 seconds. Thus the performance would be $512^2/0.0003 \approx 0.87$ Gflips or approximately 3.5 Gflops. This is the performance to be expected for longer fragment programs, where the overhead time would get significantly small. Indeed, in section 5 we achieve this performance for the 2D Ising model, which has a fragment program of 129 assembly instructions. The overhead time is significantly large compared to the other operations. This shows that creating a library of different basic vector operations on the GPU and using it for general purpose mathematical computations would not be efficient. For example the basic arithmetic operations on a 512×512 lattice would yield a performance of approximately 0.39 Gflops (counting the overhead time) instead of the much higher 3.5 Gflops (when the overhead time becomes insignificant compared to the time to execute a larger fragment program). The performance results for the $\sum$ operation are expected since reduction has twice the overhead of the other operations (the algorithm involves twice larger data transfers).

Overheads related to data traffic are also observed on the CPU, but they are more difficult to measure. For example the basic operations would take less time to compute than $a = c$ (we use full compiler optimization). This suggests that the time of the basic operations is a measure for these overheads. Note that the *cos*, *sin*, *log*, and *exp* are significantly faster on the GPU. Part of the reason is that they are accelerated on the GPU by reducing the precision of their computation.

The results from the other benchmark problems that we developed, namely the Ising and percolation implementations, are given in section 5. Measures for the CPU-GPU communication speed are given in section 6, Table 5. The communications measured use the AGP 8x port, which has a theoretical bandwidth of 2.1 GB/s on the NV30 (Figure 1, Down).

5. Performance results and analysis

We tested our Ising model implementation with results from Michael Creutz's [3] implementation. As expected, for lower input temperatures the minimization of the system's energy physical principle prevails, which is expressed by clustering of spins pointing in one direction (see Figure 2, Left). For higher temperatures the entropy maximization physical principle prevails, which is expressed by randomness in the spin orientations (see Figure 2, Right).

We tested our percolation model implementation by experimentally confirming the well known fact that the percolation threshold, or in our case the porosity threshold for fluid flow, in 2D is 0.592746 (see [8]). Figure 3 gives a snapshot of fluid spreading in 2D porous media.

Comparison between the GPU and CPU performances on the 2D Ising model is given in Table 4. The CPU code is compiled with full optimization. The results show that the GPU implementation runs approximately 3 times faster than the CPU's implementation. We observe the same speedup in our 3D implementation.

One performance drawback in implementing the Ising model on the GPU is that currently GPUs, and in particular the NV30, do not support branching in the fragment programs (see [15], Appendix C, Step 9). This means that a conditional `if/else` statement would get executed for the time as if all the statements were executed, regardless of the condition. Originally we had implemented the checkerboard execution pattern with conditional statement in the fragment program, which had made the running time twice longer, and thus the CPU's and GPU's performances comparable. Another important consideration related to the GPU's performance is that the operations should be organized in terms of 4D vector operations. For a full list of considerations for achieving high performance see [15], Appendix C.

6. Extensions and future work

Our implementation of the Ising and the percolation models were designed mostly to prove a concept and to be used in benchmarking the GPUs floating point performance. Currently we are looking into other applications that would benefit from using the GPU. We are working on Quantum Chromodynamic (QCD) applications and fluid flow simulations. Such applications usually lead to large scale computations. These computations are often impossible to perform on a single GPU (or CPU) due to memory constraints. Therefore, a main direction of our research is on the parallel use of programmable GPUs. Table 5 gives the different communication rates between the CPU and the GPU for lattices of different sizes. Although the results are far from the theoretical maximum, AGP 8x graphics bus with bandwidth 2.1 GB/s,

Operation	Lattice size (not necessary power of 2)				≈ speed (MB/s)
	64×64	128×128	256×256	512×512	
Read bdr	0.00016	0.0002	0.0006	0.0024	14
Read all	0.00040	0.0015	0.0062	0.0250	167
Write bdr	0.00022	0.0003	0.0007	0.0024	14
Write all	0.00020	0.0008	0.0032	0.0120	350

Table 5**.** Communication rates between the CPU and the GPU for lattices of different sizes. The reading is done using the glReadPixels function, writing the boundary is done using the glDrawPixels function, and writing the whole domain is done using the glTexSubImage2D function.

the CPU-GPU communication speed will not be a bottleneck in a commodity-based cluster.

7. Conclusions

Our analysis and benchmarking were mainly concentrated on the NV30 fragment processors' 32-bit floating point operations performance. The results show that it is feasible to use GPUs for numerical simulations. We demonstrated this by banchmarking the GPU's performance on basic vector operations and by implementing two probability-based simulations, namely the Ising and the percolation models. For these two applications, the applications cited in the literature overview, our vector operations benchmarks, and various nVidia distributed shaders we observed that the fragment processors' (nVidia NV30) 32-bit floating point performance can be $2-6$ times faster than the CPU for certain applications. For example, we accelerated the Ising model computation 3 times by implementing it on the GPU. Also, GPUs tend to have a higher rate of performance increase over time than the CPUs, thus making the study of non-graphics applications on the GPU valuable research for the future. Higher than the reported in this paper speedups in favor of the GPU are observed only in certain applications involving lower precision computations. A reason for this difference is the larger traffic involved in 32-bit floating point computations which traffic makes the GPUs' local memory bandwidth a computational bottleneck. Finally, we note that the acceleration graphics ports provide enough bandwidth for the CPU-GPU communications to make the use of parallel GPUs computations feasible.

Acknowledgments

We would like to thank Beverly Tomov from Cold Spring Harbor Laboratory, NY, for her attentive editing and remarks. We thank Michael Creutz from Brookhaven National Laboratory, NY, for the discussions with him, his suggestions, and advice about the models implemented.

References

1. Jeff Bolz, Ian Farmer, Eitan Grinspun, Peter Schröder, *Sparse matrix solvers of the GPU: Conjugate gradients and multigrid*, ACM Transactions on Graphics, 22(3), July 2003.
2. S.R. Broadbent and J.M. Hammersley, *Percolation processes I. Crystals and mazes.* Proc. Cambr. Phil. Soc. 53, 629-641 (1957).
3. Michael Creutz, *Deterministic Ising Dynamics*, Annals of Physics 167 , 1986.
4. W.R. Gilks, S. Richardson, and D. J. Spiegelhalter (Editors) *Markov Chain Monte Carlo in Practice*, Chapman & Hall, 1996.
5. Nolan Goodnight, Gregory Lewin, David Luebke, and Kevin Skadron, *A Multigrid Solver for Boundary-Value Problems Using Programmable Graphics Hardware*, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware (2003), pp. 102-111.
6. Mark J.Harris, Greg Coombe, Thorsten Scheuermann, Anselmo Lastra, *Physically-Based Visual Simulation on Graphics Hardware*, Graphics Hardware (2002), pp. 1-10.
7. M. Hopf and T. Ertl, *Accelerating 3d convolution using graphics hardware*, IEEE Visualization (1999), pp 471-474.
8. N. Jan, *Physica A 266*, 72 (1999).
9. E.Scott Larsen and David McAllister, *Fast Matrix Multiplies using Graphics Hardware*, The International Conference for High Performance Computing and Communications, 2001.
10. William R. Mark, R. Steven Glanville, Kurt Akeley, Mark J. Kilgard, *Cg: A system for programming graphics in a C-like language*, Proceedings of SIGGRAPH (2003).

11. J. Markoff, *From PlayStation to Supercomputer for $50,000*, The New York Times, May 26, 2003.

12. Wei Li, Xiaoming Wei, and Arie Kaufman, *Implementing Lattice Bolzmann Computation on Graphics Hardware*, The Visual Computer (to appear).

13. G.A. Marsaglia, (Editors)*Random numbers fall mainly in the planes*, Proc. Nat. Acad. Sci. 61, 25 (1968).

14. Kenneth Moreland and Edward Angel, *The FFT on a GPU*, Graphics Hardware (2003).

15. NVIDIA Corporation, *Cg Toolkit User's Manual*, Release 1.1, February 2003, Internet address (accessed on 08/2003): http://download.nvidia.com/developer/cg/Cg_Users_Manual.pdf

16. NVIDIA Corporation, *Cg homepage*, Internet address (accessed on 08/2003): http://developer.nvidia.com/object/cg_toolkit.html

17. , M.Rumpf and R. Strzodka, *Nonlinear diffusion in graphics hardware*, Proceedings EG/IEEE TCVG Symposium on Visualization (2001), pp. 75-84.

18. R. Sasik, T. Hwa, N. Iranfar, and W.F. Loomis, *Percolation Clustering: a Novel Approach to the Clustering of Gene expression Patterns in Dictyostelium Development*, Pacific Symposium on Biocomputing 6:335-347 (2001).

19. Spode's Abode, *GeForce FX Preview (NV30)*, Spode, November (2002), Internet address (accessed on 10/2003): http://www.spodesabode.com/content/article/geforcefx

20. C. Thompson, S. Hahn, M. Oskin, *Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis*, 35th Annual IEEE/ACM International Symposium on Microarchitecture (2002).

21. G. Vichniac, *Physica D*, 10 (1984), 96.