acmqueue

Realtime GPU Audio

Finite difference-based sound synthesis using graphics processors

Bill Hsu and Marc Sosnick-Pérez, Department of Computer Science, San Francisco State University

Today's CPUs are capable of supporting realtime audio for many popular applications, but some compute-intensive audio applications require hardware acceleration. This article looks at some realtime sound-synthesis applications and shares the authors' experiences implementing them on GPUs (graphics processing units).

APPROACHES TO SOFTWARE SYNTHESIS

Software synthesizers, which use software to generate audio in realtime, have been around for decades. They allow the use of computers as virtual instruments, to supplement or replace acoustic instruments in performance. Groups of software instruments can be aggregated into virtual orchestras. Similarly, in a video game or virtual environment with multiple sound sources, software synthesizers can generate each of the sounds in an auditory scene. For example, in a racing game, each discrete software synthesizer may generate the sound of a single car, with the resulting sounds combined to construct the auditory scene of the race.

Traditionally, because of limited computing power, approaches to realtime audio synthesis have focused on techniques to compute simple waveforms directly (e.g., additive, FM synthesis), using sampling and playback (e.g., wavetable synthesis) or applying spectral modeling techniques (e.g., modal synthesis) to generate audio waveforms. While these techniques are widely used and understood, they work primarily with a model of the abstract sound produced by an instrument or object, not a model of the instrument or object itself. A more recent approach is physical modeling-based audio synthesis, where the audio waveforms are generated using detailed numerical simulation of physical objects or instruments.

In physical modeling, a detailed numeric model of the behavior of an instrument or sound-producing object is built in software and then virtually “played” as it would be in the real world: the “performer” applies an *excitation* to the modeled object, analogous, for example, to a drumstick striking a drumhead. This triggers the computer to compute detailed simulation steps and generate the vibration waveforms that represent the output sound. By simulating the physical object and parameterizing the physical properties of how it produces sound, the same model can capture the realistic sonic variations that result from changes in the object's geometry, construction materials, and modes of excitation.

Suppose you are simulating a metallic plate to generate gong or cymbal-like sounds. Varying a parameter that corresponds to the stiffness of the material may allow you to produce sounds ranging from a thin, flexible plate to a thicker, stiffer one. By changing the surface area for the same object, you can generate sound corresponding to cymbals or gongs of different sizes. Using the same model, you may also vary the way in which you excite the metallic plate—to generate sounds that result from hitting the plate with a soft mallet, a hard drumstick, or from bowing. By changing these

parameters, you may even simulate nonexistent materials or physically impossible geometries or excitation methods.

There are various approaches to physical modeling sound synthesis. One such approach, studied extensively by Stefan Bilbao,¹ uses the *finite difference approximation* to simulate the vibrations of plates and membranes. The finite difference simulation produces realistic and dynamic sounds (examples can be found at <http://unixlab.sfsu.edu/~whsu/FDGPU>). Realtime finite difference-based simulations of large models are typically too computationally-intensive to run on CPUs. In our work, we have implemented finite difference simulations in realtime on GPUs.

The next section addresses key issues in realtime audio software and looks at how they relate to the computing characteristics of GPUs. The following sections look at how some types of realtime synthesis applications have benefited from GPU acceleration, ending with the challenges encountered while implementing finite difference-based synthesis on GPUs.

REALTIME AUDIO GENERATION IN SOFTWARE

Realtime audio synthesis can be broadly broken down into three steps:

- 1. Excitation.** An excitation event signals the synthesizer that realtime audio generation should begin. To strike a virtual cymbal, for example, a human performer may hit a key on a keyboard, which generates an excitation event.
- 2. Sample generation.** Audio sample data is computed for the desired sounds (for example, the cymbal crash).
- 3. Output.** Data generated in step 2 is sent to system software for playback by the system.

Figure 1 shows two approaches to realtime audio synthesis: in 1a the naïve approach computes

and outputs a single sample at a time, while in 1b the buffered approach computes multiple samples and outputs them as a block.

In figure 1a, after the excitation event, a function is called to generate a single sample. The new sample is sent to a buffer in the audio-output device. These two steps are repeated until a new excitation is received, or until the sound becomes inaudible. At the CD-quality sample rate of 44.1 kHz, one sample has to be computed and ready for output every $1/44,100$ seconds, or every $23\ \mu\text{s}$.

The naïve approach in figure 1a incurs high overhead. Every sample requires a function call and a copy into a system buffer, which may involve a context switch.

Instead, the buffered approach illustrated in figure 1b is usually used. Samples are generated and transferred in blocks of n samples, significantly reducing overhead. Though the buffered approach reduces overhead, it introduces *latency* into the signal path. Latency is the time it takes from the excitation of the instrument to the production of its sound. The longer the latency, the less responsive an instrument feels. For software instruments, latency should be kept to a minimum, on the order of tens of milliseconds.

For the buffered approach, a block of n samples has to be generated in $n \times 23\ \mu\text{s}$. In commonly used, less compute-intensive algorithms such as wavetable synthesis, generating a sample involves a few arithmetic operations and table lookups; this is usually achievable on the CPU. For compute-intensive realtime audio applications, the sample generation needs to be parallelized. The ubiquity of GPUs today makes them an obvious choice for such applications.

GPU BASICS

NVIDIA's GPUs and CUDA platform are popular options for high-performance computing today.

FIGURE 2

Hardware Configuration of an NVIDIA GPU

An NVIDIA GPU (figure 2) is a hierarchical multiprocessor on a chip, usually consisting of a number of SMs (*streaming multiprocessors*); each SM contains a number of SPs (*streaming processors*). An SM can execute large numbers of threads simultaneously, with each thread running the same program. This SIMT (*single instruction multiple thread*) architecture is especially suitable for applications with a high degree of data parallelism, where the same operations are applied to large amounts of data. For example, the NVIDIA GeForce GT 650M in a Mid 2012 MacBook Pro Retina has two SMs, each with 192 SPs (cores), at a clock rate of 900 MHz.

To the CPU, an NVIDIA GPU looks like a separate coprocessor with its own memory system. Jobs are configured on the CPU, or *host*, which then interacts with the GPU *device*. The host copies data and device-specific programs from host memory to device memory, and initiates program execution on the device. The GPU then executes the jobs independently of the host, either synchronously or asynchronously. When the GPU device is done, results are copied from the device to host memory. NVIDIA CUDA systems provide ways of reducing this memory copy latency using such techniques as shared memory pages between the host and device.

A function that is executed in parallel by the GPU is called a *kernel*. Just as the GPU hardware is composed of streaming processors grouped into streaming multiprocessors, a kernel is executed in parallel by *threads* grouped into *blocks* (see figure 3). One or more blocks are assigned to a streaming multiprocessor, guaranteeing that threads in the same block are executed on the same SM. Each thread executes the same kernel function, but usually on different data. Since threads in a block execute on the same SM, they can leverage fast hardware-supported synchronization and share data using shared memory within the same SM. Different blocks cannot be synchronized within a kernel and are not guaranteed execution order by any particular SM.

A hierarchy of memory is available on the GPU device. In addition to registers accessible to a thread, a limited amount of faster *shared memory* can be shared among threads in a block, but

FIGURE 3

Block and Thread Configuration of an NVIDIA GPU

persists only as long as the SM is executing the thread block. Larger amounts of slower *global memory* can be accessed and shared among all threads in any block. Global memory is allocated on the device by the host and persists until deallocated by the host, but access times for global memory are slower than for shared memory. Optimized GPU code leverages these thread and memory characteristics.

In older GPUs, efficient kernel execution usually requires careful management of shared memory in software. More recent GPUs, based on the Fermi and Kepler architectures, support a true hardware cache architecture; explicit software management of shared memory is not as critical on these systems.

GPU-BASED APPLICATIONS WITH MULTIPLE INDEPENDENT AUDIO STREAMS

GPUs have previously been used successfully in realtime audio applications. Many of these applications have involved the simultaneous generation of multiple loosely coupled sound sources or processing streams. In these instances the GPU has been used to ease the load on the CPU, caused by the computational complexity of generating and processing many sounds simultaneously. Examples include rendering and spatializing sound-generating objects in a game or virtual environment, or synthesizing multiple instruments in a virtual ensemble. Each sound source might be a car in a racing game or an instrument in an orchestra.

Recall the buffered approach in figure 1b; at each computation step, for each sound source, a block of n samples is computed and sent to system buffers for playback. Sequential samples in a buffer usually have to be computed in order. Since buffers for two sound sources can be computed independently of each other, assigning each sound source to a thread in the GPU is straightforward. Each thread computes an n -sample buffer and synchronizes; output from all the sources is mixed down and sent to system buffers for playback (see figure 4). Since a typical GPU today efficiently supports thousands of simultaneous threads, this type of application is a good match for GPU acceleration.

For example, Zhang et al.³ describe a typical parallel setup that implements *modal synthesis*; the

FIGURE 4

Multiple Independent Audio Streams Calculated in Parallel

sound of a vibrating object is generated by combining a bank of damped sinusoidal oscillators, each representing a *mode*, or resonant frequency of the sound. Since the n samples for each mode can be calculated independently of other modes, each mode can be assigned to an independent thread in the GPU. After n samples are generated for each mode, the results for all modes are combined for playback.

Realtime finite difference synthesis works somewhat differently and is arguably not an efficient use of the GPU, but we have been able to get useful results despite some severe constraints.

FINITE DIFFERENCE APPROXIMATION

Physical objects are frequently modeled using differential equations. To perform numerical simulations of these objects, finite difference approximations of the differential equations are commonly used. For our work, we use an approximation of the *2D wave equation*, which describes vibrations in two dimensions through an object. Consider exciting a flat rectangular plate to produce sound. The plate is modeled as a horizontal 2D grid of points. When the plate is struck, points in the grid “bounce” up and down very fast, resulting in vibration and sound, as shown in figure 5.

FIGURE 5

Vertical Displacements of Points on Flat Plate after Being Struck

In the simulation, 2D arrays keep track of the vertical displacement of the plate at each point. One array stores the current displacements, while arrays of the two previous time steps are retained. To calculate the displacement of a point at the current time step, previous displacement values around the point being calculated are used. Suppose $x_{i,j}(t)$ contains the vertical displacement at time t of the point at (i,j) . In a previous paper,² we saw that $x_{i,j}(t+1)$ can be calculated from $x_{i,j}(t)$, the four nearest neighbors of $x_{i,j}$ at time t , and $x_{i,j}(t-1)$. A *sample point* at (for example) the center of the grid, marked in red in figure 5, can be monitored to produce the audio samples for the output sound.

In a straightforward finite difference implementation, computing a $W \times W$ grid of data points at time t requires W^2 steps, and depends on the $W \times W$ grid at the previous two time steps. The next section discusses parallelizing this finite difference simulation.

CHALLENGES OF IMPLEMENTING THE FINITE DIFFERENCE TECHNIQUE

Using a finite difference-based simulation to generate a single stream of audio is a compute-intensive endeavor. Compared with the calculation of one output sample from a few sinusoidal oscillators or filters, generating a single sample in a finite difference simulation involves significantly more arithmetic and memory operations. Hence, the computation of a single sample in realtime has to be spread over multiple threads. Figure 6 shows a high-level view of a GPU-based finite difference simulation.

We faced three major challenges implementing a realtime finite difference synthesizer on the GPU. First, *kernel launch overhead*, a delay from the time the host executes the kernel on the device until the device begins execution of the kernel, may be significant. Second, the limit on the number of available threads per block restricted how the simulation grid was mapped onto the GPU. Third, the inability to synchronize or order block execution limited the way blocks of threads could be

FIGURE 6

Realtime Finite Difference Audio Synthesis

configured and executed on the GPU device.

Figure 7 shows two types of parallel audio applications: 7a demonstrates multiple independent audio streams, while 7b shows parallel finite difference simulation. As mentioned previously, with independent-stream audio processing it is usually feasible to configure the system as shown in Figure 7a; each thread freely computes an independent stream of n samples simultaneously, producing

a buffer of n audio samples at the end of a period of computation. A single synchronization event occurs after n time steps, waiting for all threads to complete their calculations. After the synchronization event, these buffers of audio data are then organized before being sent back to the host. For example, multiple sources may be mixed together or organized to maintain temporal coherence.

Suppose $x(t)$ is a two-dimensional array of vertical displacements at time t . With finite difference simulations, recall that to calculate vertical displacement of a point at i, j at time $t+1$, you need to refer to the point and its nearest neighbors at time t and $t-1$; these calculations need to be performed over the entire x array. To generate a buffer of audio samples over n time steps, you capture the vertical displacement of a single sample point at the same location in x over time; to do this it is necessary to calculate the $x(t+1)$ to $x(t+n)$ simulation arrays, while building a buffer of n samples of vertical displacement from the sample point in x . At time $t+1$, it is necessary only to retain the arrays for times t and $t-1$. Pseudocode for a sequential finite difference simulation is shown in figure 8.

As shown in figure 6, you specify $p = W \times W$ threads to execute the GPU kernel. You spread the inner-loop calculation of the $x(t+1)$ array over multiple threads. Each thread computes one point at $(myRow, myColumn)$ of $x(t+1)$, with $myRow$ and $myColumn$ based on the thread's unique ID. The parallelized pseudocode is shown in figure 9.

To calculate a buffer of audio samples over n time steps, you simply make one kernel call. Since the calculations at $x(t+1)$ depend upon the completed calculations of the two previous time steps $x(t)$, $x(t-1)$, you must synchronize after each time step (Figure 7b). The time spent in the n synchronizations in the kernel is critical to the efficiency of this approach. CUDA provides fast

FIGURE 8

Pseudocode for a Sequential Finite Difference Simulation

```
// x1ptr: ptr to x(t+1) array
// x0ptr: ptr to x(t) array
// xmptr: ptr to x(t-1) array
// buffer: n-sample audio output buffer
// W: width of plate
for bufferIndex 0 to n-1

    for row 0 to W-1
        for column 0 to W-1
            Calculate x1ptr[row][column] using x0ptr, xmptr
        End for column
    End for row

    Write sample point to buffer[bufferIndex]
    tempPtr = xmptr;
    xmptr = x0ptr;
    x0ptr = x1ptr;
    x1ptr = tempPtr;

End for bufferIndex
```

hardware synchronization support but only for threads within the same block. Therefore, all calculations must be performed using threads within a *single* block (figure 7b). Using a single block allows you to synchronize using fast mechanisms native to CUDA, but you can no longer leverage the efficiency of allowing the GPU to schedule multiple blocks of threads. Since there is only one block of threads, only one SM can operate on one finite difference approximation at any time. You could, however, simulate more than one finite difference-based instrument simultaneously, up to the number of SMs on the device.

To use multiple blocks of threads and multiple SMs, you can try configuring the kernel to calculate one time step of $x(t)$ at a time, returning control to the host after each time step. The synchronization is taken care of in the return from the kernel. In this solution, the configuration of blocks of threads depends on the locality of the data being accessed and calculated; it is a standard GPU optimization problem. However, this approach has a problem very similar to the naïve audio generation problem described previously in figure 1a; there is an overhead to the host executing a kernel on the device, and the device returning from the kernel. This kernel launch overhead builds linearly and is not insignificant. Our experiments have shown that on an NVIDIA GeForce GT 650M, on average, the minimum time to execute and return from a kernel is around 17 μ s, with some initial delays of 2 ms or longer. Recall that CD-quality audio requires generating one sample per 23 μ s. This means that even with the minimum overhead, there are about 6 μ s to calculate a sample in realtime. This is unrealistic for finite difference calculations.

Hence, we took the approach of using a single block of threads, with n time steps per kernel call, as shown in the previous pseudocode. However, we ran into another constraint: the maximum number of threads in a block is fixed in each GPU implementation and depends further on hardware

FIGURE 9

Pseudocode for a Sequential Parallelized Difference Simulation

```
for bufferIndex 0 to n-1

    Get myRow, myColumn from GPU thread ID
    Calculate x1ptr[myRow][myColumn] using x0ptr, xmptr

    Synchronize threads

    if [myRow][myColumn] is the sample point
        Write x1ptr[myRow][myColumn] to buffer[bufferIndex]

    tempPtr = xmptr;
    xmptr = x0ptr;
    x0ptr = x1ptr;
    x1ptr = tempPtr;

End for bufferIndex
```

resource constraints such as the number of registers and SPs. For larger simulation grids with more points than threads per block, each thread must be able to calculate a rectangular *tile* of several points. For example, the GeForce GT 650M supports up to 1,024, or 32×32 threads per block. To simulate a 64×64 grid, each thread would calculate a tile of 2×2 points.

THE FINITE DIFFERENCE SYNTHESIZER

Our software synthesis package, FDS (Finite Difference Synthesizer), was designed to operate on Mac OSX and Linux. FDS simulates a vibrating plate, similar to a percussion instrument. The system (figure 10) has three primary components: the controller interface (figure 10b), the finite difference engine (figure 10c), and the audio callback handler (figure 10d). Each of these three components runs in its own thread.

The controller interface (figure 10b) is the program's foreground thread. It includes a listener loop that receives control and configuration messages from an external controller, via the OSC (Open Sound Control) protocol (<http://opensoundcontrol.org>).

To use FDS, a performer manipulates an external OSC-capable controller (figure 10a), which may be a keyboard, drum pad, tablet, etc. An OSC message is sent from the controller to FDS's foreground thread. This message may change settings (simulation parameters, strike location, etc.) or trigger an excitation event (strike the plate, damp it, etc.). The thread then initiates the corresponding operations in the finite difference and audio callback threads.

To address the implementation challenges previously described, we created a *finite difference engine* (figure 10c). This engine runs continually in its own thread, executing the finite difference simulation and generating audio data. This engine thread is the only one in FDS that interacts

with the GPU device. It contains a control loop running on the host, keeping the finite difference simulation running on the device (figure 11). The control loop on the host part of the engine waits for control signals from the foreground (control) thread, such as excitation and damping events. When an excitation event is received, the host adds a precalculated 2D Gaussian impulse of vertical displacements into the finite difference grid, maintaining the current waveform while adding the energy from the excitation event. The center of the impulse is determined by the location of the “strike” on the grid (figure 5).

Audio data is transferred from the device to the host using memory shared by both the host and device. This eliminates one of the major bottlenecks formerly associated with GPUs, the device-host memory transfers. The host copies the audio data to a ring buffer shared with the *audio callback thread*, which handles getting audio data to the audio driver.

The audio callback thread communicates with PortAudio (<http://www.portaudio.com>), a cross-platform audio driver that coordinates the interface between FDS and the operating system’s audio layer. When the PortAudio driver is ready for more audio data, it executes a callback function in the audio callback thread. This function copies data placed in the ring buffer by the finite difference thread to the PortAudio output buffer. The output buffer is then sent to the operating system for playback.

RESULTS

To evaluate whether FDS constitutes a useful development in software synthesis, we asked two related questions: (1) Is FDS able to generate realtime audio based on finite difference simulations,

FIGURE 11

Finite Difference Engine Thread

for an interesting range of simulation parameters, at reasonable latencies? (2) How does FDS's performance on a GPU compare with a single-threaded finite difference simulation executed on a CPU, with identical simulation parameters? For the second question, note that GPUs and CPUs have very different architectures; different systems can have very different CPU/GPU combinations in terms of performance. Hence, our CPU-vs.-GPU comparisons should be considered *practical* references for application end users; they are not intended as rigorous comparative performance studies. Even for GPU-to-GPU comparisons, the models vary widely in their capabilities and system implementations, making these comparisons difficult.

For our measurements, we kept the audio buffer size constant and ran finite difference simulations for a number of simulation grid sizes on both CPUs and GPUs. Large grid sizes are important for generating sounds with low pitches and simulations with high spatial resolution. We also monitored the audio output buffer for underruns (i.e., when audio data is not being produced fast enough to keep up with the demands of audio playback). This produces gaps in the audio output data, which are audible as glitches or other unpleasant artifacts as the audio system waits for data to be ready.

The results of these experiments are obviously highly system dependent. We have implemented versions of FDS on GPUs for some years. On the earlier platforms, we were able to execute FDS for up to 21×21 simulation grids in realtime on an NVIDIA GTX285 GPU, with audio buffer size of 4,096

FIGURE 12

Time to Generate Audio Buffer with FDS on a Mid 2012 MacBook Pro Retina

samples, without audio buffer underruns; on the 3-GHz Intel Xeon 5160 CPU on the same system, audio buffer underruns were reported for all but trivially small grid sizes.²

Our latest test platform is a Mid 2012 MacBook Pro Retina, with a 2.7-GHz Intel Core i7 processor and 16 GB of RAM. This system has a built-in 900-MHz NVIDIA GeForce GT 650M GPU, which has two SMs with 192 SPs each. The 650M is an implementation of NVIDIA's latest Kepler architecture, optimized for power efficiency; hence, while it has numerous enhancements over the earlier Tesla and Fermi architectures, the 650M is one of the slower Kepler-based GPUs. The operating system is OS X version 10.8.2, running CUDA 5.0. We timed the finite difference kernel execution within the FDS program infrastructure, taking measurements around the kernel calls.

Figure 12 shows the time needed to generate one 512-sample buffer of audio for FDS running on the CPU and GPU of the MacBook Pro Retina. Execution times above 11 ms produce audio buffer underruns and cannot be used for audio playback. These numbers are included for speed comparison only.

For our tests we kept the audio output buffer size at 512 samples. This means that FDS needed to produce and have ready 512 samples every $23 \mu\text{s} \times 512 = 11.61 \text{ ms}$. Simulations with execution times above 11 ms produce buffer underruns and are unusable. We were able to obtain good results—meaning audio playback with no buffer underruns—for grid sizes as large as 69×69 on the CPU, and

FIGURE 13

Time to Generate Audio Buffer with FDS on a GPU Server

grid sizes as large as 84×84 for the GPU, or a 48 percent improvement in the maximum grid size supported.

As mentioned earlier, performance analysis involving GPUs and CPUs is tricky. At the risk of comparing even more apples and oranges, we introduce another point of reference. Figure 13 shows measurements made on a system with a 2-GHz Intel Xeon E5504 CPU and a 1.15-GHz NVIDIA Tesla C2050 GPU, designed as a GPU-based server for scientific applications. The C2050 is based on NVIDIA's Fermi architecture; this implementation is targeted at the high-performance computing market, with power consumption being a lower priority. For these measurements, we again timed the finite difference kernel execution independent of the FDS program infrastructure, which does not run on the GPU. The audio output buffer size was again 512 samples. The C2050 supports simulation grid sizes up to 81×81 , but the largest grid size that the slower Xeon CPU can support is about 27×27 .

SUMMARY AND FUTURE WORK

Our experiments have shown that it is possible to run finite difference-based simulations on the GPU to generate realtime audio with reasonable latency, and for grid sizes larger than is possible on a CPU. We note that CPU-vs.-GPU comparisons are tricky at best; our latest measurements were made on the GPU and CPU for the Mid 2012 MacBook Pro Retina. We were surprised that the Kepler-based 650M, designed for power efficiency, had comparable performance on FDS to the slightly earlier Fermi-based C2050, which was designed for high-performance computing. (This is partly because a single finite difference simulation can use only one SM at a time; the 650M has two SMs, while the C2050 has 14, so a much larger fraction of the C2050's hardware resources is idle). We were also surprised that current CPUs such as the Intel Core i7 exhibit competitive performance at medium grid sizes; an obvious future direction is to port FDS to multicore CPU systems for comparison with GPUs.

Our current implementation of the finite difference approximation on the GPU is straightforward. In addition, we plan to study approaches to optimize the software for larger grid sizes, for multiple finite difference-based instruments, and to support different simulation geometries. We also plan to port FDS to other GPU computing platforms such as OpenCL for testing on other GPU architectures.

REFERENCES

1. Bilbao, S. 2009. *Numerical Sound Synthesis: Finite Difference Schemes and Simulation in Musical Acoustics*. John Wiley and Sons, Chichester, UK.
2. Sosnick, M., Hsu, W. 2010. Efficient finite difference-based sound synthesis using GPUs. In *Proceedings of the Sound and Music Computing Conference*, Barcelona, Spain, July 2010.
3. Zhang, Q., Ye, L., Pan, Z. 2005. Physically based sound synthesis on GPUs. In *Entertainment Computing-ICEC 2005, Lecture Notes in Computer Science*, Springer, Berlin, 328–333.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

BILL HSU is an associate professor of computer science at San Francisco State University. His current interests include high-performance computing, audio analysis and synthesis, physics-based modeling, and audiovisual performance systems. He has collaborated extensively with artists and musicians and has organized numerous concerts in the San Francisco Bay area.

MARC SOSNICK-PÉREZ is a graduate student, teacher, and researcher at San Francisco State University. His interests include high-performance computing, software engineering education, and computer music. He is part of SFSU's SETAP (Software Engineering Team Assessment and Prediction) project exploring novel means of assessing student learning, and he was part of the CAMPAIGN project exploring the use of GPU-accelerated clustering algorithms.

© 2013 ACM 1542-7730/13/0400 \$10.00