Discrete-event Execution Alternatives on General Purpose Graphical Processing Units (GPGPUs)

Kalyan S. Perumalla
perumallaks@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA

Abstract

Graphics cards, traditionally designed as accelerators for computer graphics, have evolved to support more general-purpose computation. General Purpose Graphical Processing Units (GPGPUs) are now being used as highly efficient, cost-effective platforms for executing certain simulation applications. While most of these applications belong to the category of time-stepped simulations, little is known about the applicability of GPGPUs to discrete event simulation (DES). Here, we identify some of the issues & challenges that the GPGPU stream-based interface raises for DES, and present some possible approaches to moving DES to GPGPUs. Initial performance results on simulation of a diffusion process show that DES-style execution on GPGPU runs faster than DES on CPU and also significantly faster than time-stepped simulations on either CPU or GPGPU.

1. Introduction

Traditionally, graphics cards for workstations and personal computers have been designed to handle intensive graphics operations to enable high speed rendering of complex objects and scenes. More recently, the graphics cards of the past have been evolving to support more programmable interfaces for graphics operations. These interfaces eventually became sufficiently general to be able to map non-graphics computation in terms of graphics elements and achieve non-graphics computation on graphics processors. Evolution in hardware programmability together with software development platforms has transformed graphics cards into General Purpose Graphical Processing Units (GPGPUs). Their programmability has reached a point to make them suitable for more general-purpose computation[1, 2]. Computing that is generally targeted towards execution on CPUs could now be re-targeted for execution on GPUs. An application can use a GPGPU as either co-processor or core processor.

General-purpose computation using GPUs is, however, a relatively recent area of research. Several new applications of GPGPUs are being considered, and new algorithms are being developed that are demonstrated to be highly efficient for execution on GPGPUs. Certain applications have been shown to execute much faster on GPGPUs than on CPUs[2]. Generally speaking, applications that have more "arithmetic intensity" are more suitable for GPGPUs. GPUs have also been touted as low-cost alternative platforms for supercomputing, due to their high peak execution rates relative to comparable CPUs.

Among simulation applications attempted on GPGPUs, the majority use time-stepped execution, and have been shown to execute must faster on the GPGPUs than on CPUs[3]. This is mainly because time-stepped approaches tend to map relatively easily to the *streaming* paradigm of GPGPUs. However, little is known with respect to the applicability of GPGPUs to discrete event simulation (DES). It is unknown whether DES is relevant, practical and/or better on GPGPUs relative to CPUs. At the outset, it seems unclear as to how discrete event models could be mapped to the streaming model of execution of GPGPUs. For example, is a traditional event loop implementation applicable to GPGPUs? Is there a more suitable alternative DES implementation approach for GPGPUs? How much faster can GPGPU-based DES perform compared to an equivalent CPU-based DES? What types of DES applications are better suited for execution on GPGPUs? In this paper, we attempt to address and answer some of these questions, and highlight areas where additional research is needed for better understanding.

In section 2 we provide relevant background information on GPGPUs. In section 3, we present a case study in using GPGPUs for simulating a phenomenon (diffusion process) that has both TS and DES models, and show that a DES algorithm specially adapted for the GPGPU can outperform both a traditional DES algorithm on the CPU as well as time-stepped algorithms on CPU and GPGPU. Following the case study, in section 4, we sketch possible alternative algorithms and data structures for executing more general DES applications on the GPGPU platforms. We also outline the challenges and limitations that GPGPU platforms impose that constrain the types of DES applications that can be effectively realized. Finally, we conclude and discuss future work in section 5.

2. Motivation and Background

A considerable body of literature now exists on hardware, software and algorithmic details of graphics processors that are suitable for general purpose computation. The related literature and bibliography is now too large to be cited comprehensively here. Hence, we outline the main GPGPU features that are relevant in our immediate context of DES applications. The interested reader is referred to [1, 2, 4, 5] for literature surveys and detailed background on GPGPUs, and to some applications such as efficient line-of-sight calculations for battle-field simulations[6], and efficient tracking of pheromone diffusion effects in unmanned vehicle control[7].

2.1. GPGPU Architectures

A highly simplified functional view of a GPGPU is shown in Figure 1. A set of textures is fed as input to a bank of "fragment processors". User-specified code, called "kernel," can be loaded into fragment processors (FPs), which is executed for each and every element of the input textures. As and when computed values are generated by the fragment processors, the computed output is stored at appropriate locations in target textures. The generated output textures can then be fed back as input for additional processing, and so on. For performance reasons, it is desirable to perform as many operations as possible with the texture memory before the values in texture memory are transferred to/from the host CPU's main memory.

Many other details, such as the vertex processors and the rasterizer, are not discussed here for simplicity. The interested reader is referred to [8] for additional detail on GeForce 6800, which is a good representative of modern GPGPUs. Current GPGPUs even support conditional and looping constructs in FPs, as well as a few registers local to each FP. Parallelism is realized by processing multiple elements of input textures concurrently among all available FPs. Asynchronous memory fetches are supported by FPs, allowing computation on some elements to proceed even while other elements are being fetched from texture memory.

The pipelined flow of vertices and fragments through the graphics pipeline is the main reason why general purpose computation on GPUs is cast in the form of *stream computation*. Streaming applications map very well to GPGPU architectures, in which the same operation is applied to all elements in a stream of data elements (e.g., performing transforms on a stream of Cartesian positions & vectors).

In comparison to CPUs, the biggest constraint imposed by GPGPUs is a lack of scatter operations (i.e., "compute & store instructions" of the form `a[i]=b`). This is because the output address of fragment processor is automatically fixed by the hardware for each input element. This makes assignment to arbitrary location of an output texture difficult. But gather operations (i.e., "load & compute" instructions of the form `b=a[i]`) are supported by GPGPUs in the form of "dependent texture fetch" operations. Algorithms for achieving scatter in terms of gather have been proposed[1], but cannot always be applied without regard to increased runtime overheads.

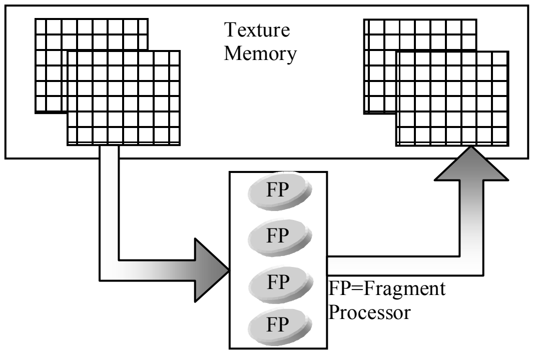

Figure 1: Highly simplified schematic of GPGPU operation. Textures are input to a bank of fragment processors (FPs). FPs "render" the results of their computation to target textures.

Another limitation of GPGPUs is the limit on the maximum size of a texture that can be allocated (e.g., 4096 x 4096 floating point values per 2-D texture). Floating point operations are typically single precision, although very recent GPGPUs are offering support for double (64-bit) precision.

2.2. GPGPU Programming Environments

Beyond traditional tools for graphics applications, special programming languages and environments have been developed for general purpose programming on GPGPUs. The `Cg` language[9] from NVIDIA provides C-like interface to graphics primitives. An optimizing compiler generates shader routines from `Cg` program fragments. Its high-level language interface helps shield the developer from low-level graphics programming, but demands some level of expertise with graphics concepts (e.g., colors, textures, etc).

Similar interfaces are supported by the `fxc` compiler of `DirectX SDK` from Microsoft. Higher-level languages, such as `Brook`[5], provide more generalized abstraction of "streams" and stream programming constructs. The `Brook` compiler generates code that maps all abstractions transparently to graphics primitives, and manages all runtime aspects automatically.

In the experiments reported here, we coded all GPGPU simulations in `Brook`, and we executed using the `DirectX 9 (dx9)` runtime. The CPU is an Intel 2.13GHz Centrino with 2 Gigabytes of memory. The GPGPU is an NVIDIA GeForce 6800 Go[8] with 256MB memory, and contains 16 fragment processors and 4 vertex processors. All CPU programs are compiled with Microsoft Visual C++ v7.

2.3. Parallel Simulation "in the Small"

A significant amount of parallel/distributed discrete event simulation (PDES) literature has been focused on traditional CPU-based execution. Parallel processing on these traditional platforms is enabled by multiple interconnected CPUs connected either by high-speed interconnects or by a network. However, the GPGPUs represent a different type of parallel simulation platforms that are emerging lately.

A way to consider GPUs is to view them as a means to perform parallel simulation "in the small". This is in contrast to traditional parallel simulation "in the large" using networks connecting a large numbers of CPUs. In traditional parallel/distributed simulation efforts, the simulation problem size is typically scaled in order to afford enough parallelism commensurate with the increase in the number CPUs. A modern GPGPU on the other hand contains a small number (e.g., 8 or 16) of fragment processors (parallel processing elements) built into the chip, which can be exploited for parallel simulation with minimal inter-processor communication penalty.

In fact, it is the performance/price ratio that is a key differentiating factor between parallel processors and GPGPUs. While conventional parallel processors (e.g., dual or quad-CPU shared-memory machines) could conceivably deliver performance comparable to GPGPUs on applications of interest, GPGPUs are extremely appealing due to their low cost. For example, the recent GeForce 7 series GPGPU with 24 fragment processors only costs less than $500, which *includes* its 256MB video memory. Thus CPU and GPGPU comparisons are based more on configurations with similar prices, rather than on those that are technically equivalent.

Other non-CPU platforms include Field Programmable Gate Arrays (FPGAs) and network co-processors. These non-conventional platforms also afford opportunities for realizing parts of simulation functionality, such as time synchronization[10], check-pointing[11] and data distribution[12]. The GPGPU approaches considered here, however, are intended to perform an entire simulation, and not just parts of it.

3. DES on GPGPU – Case Study

3.1. Application

We use a diffusion simulation as an application for a case study in initially exploring the DES methodology on GPGPUs. Simulation of the diffusion equation is a well-studied problem, and has many applications (e.g., heat transfer, dye spreading and gas diffusion). We chose this application as it easily affords both time-stepped as well as discrete-event formulations for its solution[13]. Our simulation uses the following two-dimensional version of the diffusion equation:

$$\frac{\partial Q}{\partial t} = \alpha_x \frac{\partial^2 Q}{\partial x^2} + \alpha_y \frac{\partial^2 Q}{\partial y^2} + \beta$$

The space is uniformly and constantly heated from outside, i.e., the temperature is held constant at the boundary, with the initial temperature of the interior being at a value lower than at the boundary. For discretization of the continuous function, the *spatial* dimension is discretized by partitioning the space as a grid in *x* and *y* dimensions. The choice of the method for *temporal* discretization is an important one, leading to different approaches to simulation.

We now outline the traditional time-stepped algorithm, followed by two other algorithms: discrete event and hybrid. For each of these algorithms, we present runtime performance results. The results are normalized against the runtime of the first algorithm (traditional time-stepped) running on a CPU. Thus, all speedup numbers are relative to time-stepped simulation performance on a CPU.

3.2. Time-Stepped Algorithm

In the time-stepped method of simulation (TS), time is discretized into a grid with equi-distant points, with the spacing fixed for all grid elements. Time-stepped simulation is schematically illustrated in Figure 2. The horizontal bars represent timelines of each logical process, while the solid vertical lines represent points in simulation time at which the logical processes are updated. The time step value (simulation time period between successive updates to the state) is determined by model-specific means to ensure stability along with sufficient accuracy.

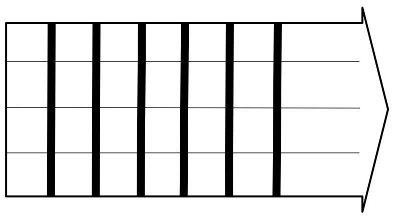

Figure 2: Schematic of timestepped simulation

In the diffusion process simulation, within each time step, the processing per (*i,j*) grid element in the 2-D grid can be performed by one of several known methods. We chose the following simple explicit method, where $q^{n+1}_{i,j}$ is the computed value of q_{ij} at timestep $n+1$:

$$q^{n+1}_{i,j} = q^{n}_{i,j} +$$
$$\alpha_x \frac{q^n_{i,j-1} - 2q^n_{i,j} + q^n_{i,j+1}}{\Delta x^2} + \alpha_y \frac{q^n_{i-1,j} - 2q^n_{i,j} + q^n_{i+1,j}}{\Delta y^2} + \beta$$

The time-stepped algorithm for the diffusion grid is shown in Figure 3. Previous research on GPGPUs has, by and large, implemented simulations using such a time-stepped approach on GPGPUs, and compared their performance against that on CPUs[5]. We implemented this algorithm both on a CPU and on a GPGPU. The runtime performance of time-stepped algorithm is shown in Figure 4. Consistent with the level of speedups published in the literature, our implementation of the diffusion process simulation reflects more than 4-fold speed up of GPGPU over CPU, as shown in Figure 4.

```
1.    While not end of simulation
          /*Advance current simulation time*/
1.1       $t_{now}$ += timestep
          /*Advance all grid elements to current time*/
1.2       For all (i,j): $Q_{ij}$ += $Qdot_{ij}$ * timestep
```

Figure 3: Time-stepped algorithm

It is known from the GPGPU literature that, in general, when an application's working set fits mostly within the cache of CPU, the application executes sufficiently fast to exceed the speed of an equivalent version of the application on a GPGPU. This phenomenon is indeed reflected in the speedup on smaller problem sizes of the diffusion application, which are sufficiently small to fit well within the L2 cache size of 1MB. In problems with grid sizes 50x50 and 100x100, the GPGPU version experiences slowdown relative to the CPU-based time-stepped approach. When the working set of the problem no longer fits in the L2 cache, however, there is significant benefit to using the GPGPU. The fragment processors are kept busy on the GPGPU via asynchronous memory operations, resulting in over 4-fold speedup.

While the relative performance improvement of time-stepped algorithms is as expected, a logical question arises, namely, how GPGPU and CPU performance compares on discrete event algorithms. DES in general entails fewer, infrequent updates to the logical processes and hence performance can be expected on the CPU to equal the speed improvements afforded by GPGPU. We will now consider an equivalent discrete event formulation of the problem, and compare its performance on a CPU against the fast time-stepped performance of GPGPU.

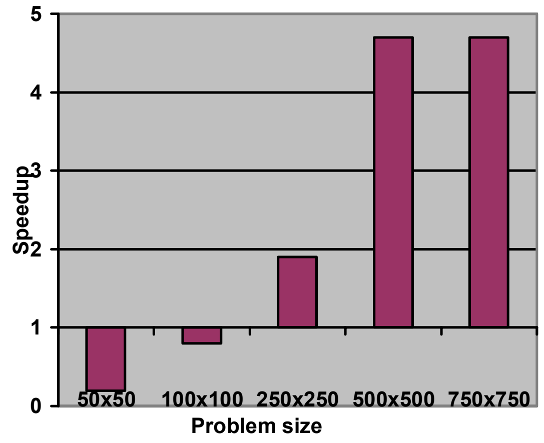

Figure 4: Speedup of GPU-based time-stepped algorithm relative to that on CPU

3.3. *Discrete-Event Algorithm*

In discrete event formulation, time is discretized on an individual basis for each grid element, independently and dynamically during the simulation. The discrete event formulation for the diffusion problem has been studied (see, for example, [13]). A schematic of updates is shown in Figure 5, and the pseudo code for the DES algorithm is shown in Figure 6. We used the ADEVS package[13] for a CPU-based implementation of the DES formulation.

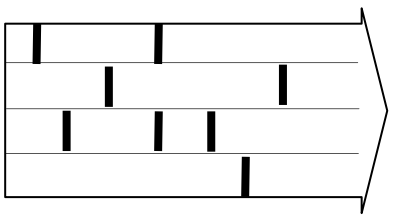

Figure 5: Schematic of DES execution

The main idea is that each grid element *i,j* computes the next latest time at which its state needs to be updated without violating a discretization of the effect of its own state value ($q_{i,j}$) on its neighboring elements.

In this CPU-based formulation, each element schedules an event to itself for its next update time. If nothing changes before the time at which the update occurs, the state gets advanced correctly. Otherwise, if the value of any of its neighboring elements changes "significantly", the update time of this element is recomputed and the self event is rescheduled accordingly. The correction and rescheduling of the update event of an element is accomplished by updating the event in the future/pending event list.

```
1. For all (i,j)
1.1     $Qlast_{ij} = Q_{ij}$
1.2     $dt_{ij}$ = compute_dt(Q,i,j)
1.3     EventList.Insert(i,j,$dt_{ij}$)
2. While not end of simulation
        /*Find grid element with earliest timestamp*/
2.1     ($i_{min}$, $j_{min}$, $dt_{min}$) = EventList.PeekTop()
        /*Advance all neighbor elements to current time*/
2.2     For all neighbors & itself (x,y) of ($i_{min}$,$j_{min}$)
2.2.1           $Q_{xy}$ += $Qdot_{xy}$ * $dt_{min}$
        /*"Move" the grid element to its phase point*/
2.3     $Qlast_{imin,jmin} = Q_{imin,jmin}$
        /*Reschedule events of updated elements*/
2.4     For all neighbors & itself (x,y) of ($i_{min}$,$j_{min}$)
2.4.1           $dt_{xy}$ = compute_dt(Q,x,y)
2.4.2           EventList.Adjust(x,y,$dt_{xy}$)
```

Figure 6: Discrete event algorithm

Clearly, the CPU-based DES algorithm (CPU-DES) outperforms the fast GPGPU-based time-stepped algorithm (GPU-TS), as shown in Figure 7. While GPU-TS is over 4 times faster than CPU-TS, CPU-DES is over 8 times faster than CPU-TS (for the 750x750 grid scenario). The natural question that follows is whether there exists an algorithm analogous to CPU-DES that can make the GPGPU perform better than GPU-TS and CPU-DES. The main challenge in addressing this question is the fact that the DES algorithm cannot be ported to the GPGPU architecture. As we discuss in more detail in section 4, selective individual updates to grid elements are not possible in the GPGPU's streaming paradigm. The challenge then is to find a suitable variant of the DES approach that is better suited and realizable on a GPGPU. We describe one such algorithm next.

3.4. Hybrid Algorithm

The key difference between the (CPU-based) DES algorithm and the traditional time-stepped algorithm is that time advances are permitted to be variable in length at runtime. On a CPU, it is possible to selectively vary this length on an individual element-by-element basis. An equivalent operation on a GPGPU would be to find the minimum timestamp among the next update times of all elements, and advance that particular element to its update time. However, it then becomes difficult to thereafter selectively modify the update times of that element's neighbors.

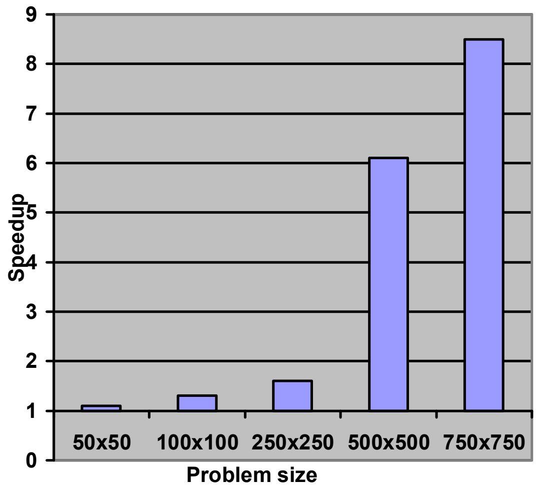

Figure 7: Speedup of CPU-based DES algorithm relative to CPU-based TS algorithm

To accomplish this, it becomes necessary to update *all* elements of the grid, not just the neighbors, since it is necessary to stream *all* the grid elements anyway, even if one wanted to only update the neighbors alone. Reasoning this way, a "hybrid" algorithm emerges naturally that combines the time-stepped nature of synchronous updates to all elements with the variable timestep support of DES. This is realized in the hybrid algorithm shown in Figure 8, which is surprisingly simple. Each individual element computes its next "safe" update timestamp in step 1.2. The minimum among all update times is chosen as the next timestep, and a synchronous update of state is performed across all elements in step 1.5. While the variability of minimum timestep mimics the DES nature of dynamic updates, the synchronous updates mimic the collective nature of state updates of the time-stepped method, thus giving the hybrid scheme. The schematic diagram corresponding to the hybrid algorithm is shown Figure 9. The schematic shows that global synchronous updates are performed at every discrete event point. Note that multiple discrete event points become merged into a single update step during synchronous updates (which turns out to be a key benefit with GPGPU).

Advantages: While the hybrid is surprisingly simple, it provides some non-intuitive advantages when executed on a GPGPU. First, a synchronous update can result in faster time advances globally, leading to faster evolution of state as compared to time-stepped method.

```
1. While not end of simulation
        /*Find next update times for all elements*/
1.2     For all (i,j): $dt_{ij}$ = compute_dt(Q,i,j)
        /*Find minimum among all update times*/
1.3     $dt_{min}$ = min($dt_{ij}$) of all (i,j)
        /*Advance current simulation time*/
1.4     $t_{now}$ += $dt_{min}$
        /*Advance all grid elements to current time*/
1.5     For all (i,j): $Q_{ij}$ += $Qdot_{ij}$ * $dt_{min}$
```

Figure 8: Hybrid algorithm

Secondly, simultaneous events are processed *in parallel* on the fragment processors of GPGPU. In a CPU-DES, simultaneous events are processed one at a time (i.e., grid elements that have the same update time are sequentially processed), where as such simultaneity is naturally processed in a single step (1.5) in the synchronous update in the hybrid algorithm. This effect is pronounced when many grid elements have the same value for their next update timestamps (which is often possible in initial stages of diffusion with large grid sizes). Thirdly, computing the minimum update time is logarithmic in time complexity on the GPGPU due to the availability of hierarchical reduction algorithms on streams[2]. This makes the overhead for computation of the minimum timestep to be comparable to that of event list management on a CPU.

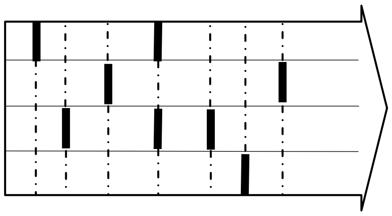

Figure 9: Schematic of hybrid simulation

Note that this algorithm differs from synchronous event processing algorithms such as described in [14]. The operation of the hybrid algorithm is somewhat akin to that of variable timestep algorithms, but differs from them in the fact that all elements are processed in parallel at each timestep, unlike on a single CPU. Moreover, the hybrid algorithm is akin to variable timestep algorithms only operationally, but retains the unique capability of advancing each element independently. In fact, the synchronous global update is a "free" operation for each local update, because of the stream processing nature of GPGPUs. Also, repeatability is not affected by parallel processing because the write operations on the input do not take affect (i.e., input is not affected by output) until all writes are completed.

The runtime performances of hybrid algorithm on a GPGPU (GPU-Hybrid) and on a CPU (CPU-Hybrid) are shown in Figure 10. The 16-fold speedup of GPU-Hybrid over CPU-TS makes it the fastest among all variants. In particular, it is more than twice as fast as the optimized CPU-DES approach which is the fastest sequential implementation on a CPU. The reason is that GPU-Hybrid not only reaps the advantages of leaps in time due to DES-style operation, but also benefits from fast parallel (stream) processing within each synchronous update. That parallel processing on the GPGPU (by fragment processors) contributes significantly to this speedup is reinforced by the low speedup of the same algorithm on a CPU (CPU-Hybrid).

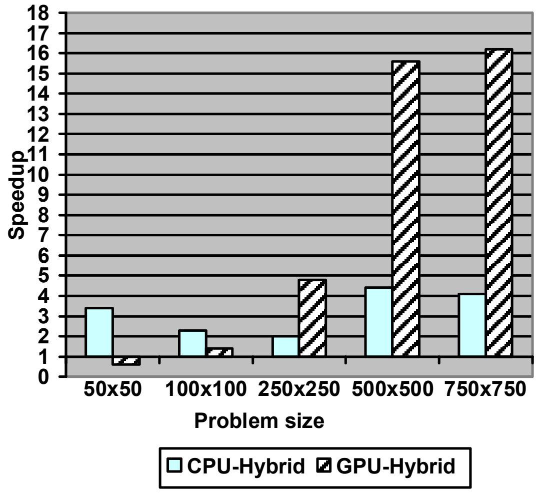

Figure 10: Speedup of CPU- & GPU-based hybrid algorithms relative to CPU-based time-stepped algorithm

3.5. Additional Discussion on Performance

It is clear that the CPU-based simulations are faster than GPGPU-based simulations for smaller problems sizes, presumably due to their containment in the CPU's L2 cache. This is in line with the well-known rule of thumb that the Intel Pentium 4 processor operating mostly off its L2 cache outperforms an ATI or NVIDIA GPU despite the GPGPU's concurrent processing and asynchronous memory operations. This explains the slow down (speed up less than 1.0) for 50x50 and 100x100 sized grids.

The real benefit of GPUs' streaming architecture in fact becomes truly evident on larger problem sizes that exceed the L2 cache size of the CPU (e.g., 500 x 500 x 8 bytes/variable x 3 variables = 6MB, which is more than the 2MB L2 cache of the Intel processor we used). In these scenarios, the GPGPU starts posting gains over CPU, and delivers performance almost equal to or greater than an optimized DES version executing on a

CPU. Brook documentation states that a kernel function can have any number of output streams. However, we found that the runtime is buggy when more than one output is generated by a kernel function. One reason for this discrepancy could be due to the fact that we used a NVIDIA GeForce 6800 Go, which is a portable version of the Brook-tested GeForce 6800 Ultra. Our implementation in Brook hence was constrained by a single output stream, and could be optimized to potentially run faster if multiple output streams were employed.

4. Generalized DES on GPGPU

We now turn our attention to extending these approaches to a more generalized set of DES applications. The GPGPU computation paradigm presents significant challenges to realizing the traditional DES application programming interface in full generality. We describe the typical event/LP model in traditional DES/PDES, and then discuss how such an interface relates to the GPGPU platform constraints. Here we focus on an event-oriented view [14] of DES. Process-oriented views are considerably more complex to implement on the GPGPU (due to lack of stack context support) and not considered here.

In traditional (event-oriented) DES, events are processed in time stamp order. In general, the simulation state is organized into multiple logical processes (LPs). Time-stamped events are sent from one (source) LP to another (destination) LP. During processing of an event, more new events could be generated by the source LP to any subset of LPs (including the source LP itself). The processed event is then noted as "consumed", which removes the event from the event list and makes its memory eligible for reuse. As part of event processing, the LP's state memory is modified. Unfortunately, this classical style of event processing simply does not carry forward to GPGPU platforms for the following reasons:

1. Since processing cannot be performed on events (or LPs) one at a time in von Neumann style, traditional event processing simulation loop does not apply to GPGPUs. Consequently, isolated, individual processing of events is extremely inefficient on GPGPU's streaming architecture.
2. The notion of sending events from one LP to another is straightforward to implement on the CPU. Conceptually if an event `E` is to be sent by LP `i` to LP `j`, it is realized as an assignment similar to: `Event[i][j]=E`. However, this is not easily implemented on a GPGPU either, because of lack of scatter operations as discussed in Section 2.1.
3. Also, the possibility of one event generating multiple events is relatively difficult to realize in the streaming paradigm of GPGPUs, due to significant performance overhead incurred by dynamic variability of stream sizes.

On the other hand, a GPGPU does help perform parallel processing of simultaneous events naturally. As we saw, the hybrid algorithm shown in Figure 8 naturally captures the availability of all simultaneous events and opens them up for parallel processing on the fragment processors of the GPGPU. The GeForce 6800, for example, has 16 fragment processors, which opens up 16-fold parallelism for processing simultaneous events. A CPU on the other hand processes simultaneous events sequentially, one at a time. Overall, it appears as though existing DES (and PDES) conceptual frameworks need to be rethought from scratch. The key to realizing DES on GPUs is to cast traditional events and LPs into a stream processing paradigm supported by GPGPUs.

4.1. GPGPU Usage Alternatives in DES

GPGPUs can be put to use in DES in more than one way. For example, one could use GPGPUs with a traditional CPU-based event scheduler as usual for DES. In this scheme, any intensive computation that is present in intra-event processing can be delegated to be performed on the GPGPU. The GPGPU is essentially used as a co-processor during each individual event processing. Somewhat similar to the line-of-sight calculations used in [6], this style of DES is particularly suitable for medium- to coarse-grained events that entail heavy arithmetic processing (such as linear algebra or transcendental function computations). Also, this approach is relatively easy to adapt an existing application to use GPGPU.

A more challenging approach is to use the GPGPU as a core processor rather than as a co-processor. In this "all-GPU approach," the entire DES event processing is performed on the GPGPU, with little mediation from the CPU. The GPU-Hybrid algorithm in Figure 8 is an example of this type of approach. In general, two streaming alternatives can be envisioned:

1) "Stream of events", in which the event list is stored as a stream of events, and the entire event stream is fed as input to an event processing kernel on the GPGPU. The kernel processes only those events in the input event stream whose timestamps are less than safe/allowed time, and leaves the others marked unprocessed.
2) "Stream of LPs", in which all the logical-processes are stored as a single stream, and fed through an LP-processing kernel, which processes all events of each LP that are eligible for processing. This kernel

processes only those LPs who have events whose event time is less than safe/allowed time.

Although event stream and LP stream schemes appear to be two distinct possibilities, they are in fact necessitated by the GPGPU's streaming and gathering architecture (and lack of scatter) to be one and the same approach. This is because, in order to process an event, the event's LP state must be updated as well. Also, in order to send/receive events, the source and/or destination LP state should be available at the time of an event processing. Thus, the only data structure that seems feasible is one in which an LP has a fixed number of events that it has sent to other LPs. This data structure allows events and LP state to be co-located, allowing them to be read and written in the event processing kernel. The resulting output stream will be an updated LP stream for use in the next iteration of simulation.

5. Conclusions and Future Work

GPGPUs offer a unique set of computational features that should prove useful for efficient execution of DES applications. Also, their mass market enables cost advantages via economies of mass manufacture. GPGPUs thus represent highly promising platforms for efficient (and cost-effective) execution of DES applications, but significant new research is needed to redesign traditional DES/PDES approaches into new stream-based paradigms for GPGPU platforms.

While time-stepped approaches have been studied in simulations using GPGPUs, to the best of our knowledge, this is the first work to explore DES using GPGPUs, and to properly compare the best sequential (DES-based) simulation on the CPU against a comparable technique on GPGPU. The results demonstrate the benefits of GPGPUs, showing two-fold improvement of GPGPU-based DES implementation over a traditional CPU-based DES execution. Discretization is thus helpful to speed up the simulation relative to time-stepped simulation. However, for more general DES applications, the traditional discrete event simulation loop seems inapplicable to GPGPU's stream processing style of computation, and hence a hybrid (discrete plus time-stepped) approach seems more appropriate. The performance results reflect this relative ordering of approach.

Additional research is needed to optimize DES variants on GPGPUs, both in terms of algorithms as well as optimized system implementations. For example, uncovering additional parallelism via lookahead across fragment processors could be explored. Similarly, optimized implementations directly over lower-level frameworks such as `Cg` language (as opposed to using **`Brook`**) can help achieve much higher performance for tuned DES applications. Simulations with greater arithmetic intensity are to be explored for even better suitability to the GPGPU platforms. We are also exploring the performance of physics models such as particle-in-cell models in this light.

References

[1] M. Pharr and R. Fernando, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*: Addison Wesley Professional, 2005.

[2] J. D. Owens, et al., "A Survey of General-Purpose Computation on Graphics Hardware," Eurographics, 2005.

[3] S. Tomov, et al., "Benchmarking and Implementation of Probability-based Simulations on Programmable Graphics Cards," *Computers and Graphics*, vol. 29(1), pp., 2005.

[4] General Purpose Computation Using Graphics Hardware, http://www.gpgpu.org.

[5] I. Buck, et al., "Brook for GPUs: Stream Computing on Graphics Hardware," *ACM Transactions on Graphics*, vol. 23(3), pp. 777-786, 2004.

[6] M. Verdesca, et al., "Using Graphics Processor Units to Accelerate OneSAF: A Case Study in Technology Transition," Interservice/Industry Training, Simulation and Education Conference (IITSEC), 2005.

[7] B. Walter, et al., "UAV Swarm Control: Calculating Digital Phermone Fields with the GPU," Interservice/Industry Training, Simulation and Education Conference (IITSEC), 2005.

[8] J. Montrym and H. Moreton, "The GeForce 6800," *IEEE Micro*, vol. 25(2), pp. 41-51, 2005.

[9] R. Fernando and M. J. Kilgard, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, 1 ed: Addison Wesley Professional, 2003.

[10] M. Rosu, et al., "Supporting Parallel Applications on Clusters of Workstations: The Intelligent Network Interface Approach," IEEE Symposium on High Performance Distributed Computing, 1997.

[11] F. Quaglia and A. Santoro, "Non-blocking Checkpointing for Optimistic Parallel Simulation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14(6), pp. 593-610, 2003.

[12] A. Santoro and R. M. Fujimoto, "Off-Loading Data Distribution Management to Network Processors in HLA-Based Distributed Simulations," Distributed Simulations and Real-Time Applications, 2004.

[13] J. Nutaro, "Parallel Discrete Event Simulation with Application to Continuous Systems," thesis, University of Arizona, 2003.

[14] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*: Wiley Interscience, 2000.