GPU Computing to Improve Game Engine Performance

Abu Asaduzzaman* **& Hin Y. Lee**

Department of EECS, Wichita State University
1845 Fairmount St #JB-253, Wichita, Kansas 67260-0083, USA
*Email: Abu.Asaduzzaman@wichita.edu

Abstract. Although the graphics processing unit (GPU) was originally designed to accelerate the image creation for output to display, today's general purpose GPU (GPGPU) computing offers unprecedented performance by offloading computing-intensive portions of the application to the GPGPU, while running the remainder of the code on the central processing unit (CPU). The highly parallel structure of a many core GPGPU can process large blocks of data faster using multithreaded concurrent processing. A game engine has many "components" and multithreading can be used to implement their parallelism. However, effective implementation of multithreading in a multicore processor has challenges, such as data and task parallelism. In this paper, we investigate the impact of using a GPGPU with a CPU to design high-performance game engines. First, we implement a separable convolution filter (heavily used in image processing) with the GPGPU. Then, we implement a multiobject interactive game console in an eight-core workstation using a multithreaded asynchronous model (MAM), a multithreaded synchronous model (MSM), and an MSM with data parallelism (MSMDP). According to the experimental results, speedup of about 61x and 5x is achieved due to GPGPU and MSMDP implementation, respectively. Therefore, GPGPU-assisted parallel computing has the potential to improve multithreaded game engine performance.

Keywords: *Game engine; GPGPU computing; multicore processor; parallel programming; performance improvement; simultaneous multithreading.*

1 Introduction

Intel introduced the first video graphics controller (iSBX 275) in 1983 [1,2]. Then Texas Instruments (TMS34010, 1986), IBM (8514 Graphics System), etc. enhanced GPU applications. Throughout the 1990s, 2D graphical user interface (GUI) acceleration continued to evolve. The NVIDIA Corporation was the first to produce a chip capable of programmable shading (GeForce 3, early 2000s). Today (mid-2014), GPUs are used in embedded systems, mobile phones, personal computers, workstations, and game consoles. Modern GPUs are very efficient in manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing large blocks of data is done in parallel. Since 2006 (with the

Received August 3rd, 2013, 1st Revision January 22nd, 2014, 2nd Revision May 6th, 2014 Accepted for publication May 7th, 2014.
Copyright © 2014 Published by ITB Journal Publisher, ISSN: 2337-5779, DOI: 10.5614/j.eng.technol.sci.2014.46.2.8

introduction of the GeForce 8 series), NVIDIA has produced general purpose GPUs for scientific and engineering computation. For its GPUs, the company has also developed compute unified device architecture (CUDA) [2], a parallel computing platform and programming model. Currently, many companies, including NVIDIA, Intel, and AMD/ATI, produce GPGPUs [3-5]. Although the original GPU, a specialized electronic circuit, was designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display, now GPGPU computing offers exceptional application performance by offloading computation intensive parts of the application to the GPU, while the rest of the code runs on the CPU.

Game engines are traditionally used for developing video games. In addition to standalone and online game machines, game engines are now being used for educational, medical, and military applications as well [6,7]. A simple modern game engine is normally comprised of the following components: input, game logic, artificial intelligence (AI), physics (engine for collision detection/ response), audio (for sound), and graphics. A rendering engine called “renderer” is required for 2D or 3D graphics. Many subcomponents can comprise a component and together they form a complete package. Different levels of parallelism, such as task and data levels, can be used in game programming. When components in a game engine consist of many different types of middleware, the design of the library will most likely dictate which one is more suitable to use. Some middleware, such as the Bullet Physics Library, includes multithreading in its API [8]. Depending on the type of multithreading model used, some level of data redundancy and mechanism to ensure data coherency is required to improve performance. To fulfill the performance requirement, game engines are adopting new hardware like multicore CPUs and software like multithreaded processing.

Multithreading can be implemented in a game engine in many different ways. However, the currently available middleware used in high-level APIs makes the implementation of parallelism very challenging. Therefore, various methods should be evaluated when implementing multithreading in a game engine because a given multithreading technique might not be suitable for a particular component’s API due to the way it is built. It is difficult to objectively calculate and/or predict which implementation is needed to properly optimize a multithreaded application; therefore, optimization of multithreaded game engines requires numerous experiments.

The success of modern game engines significantly depends on new innovation, that is, the shift from single-core to multi-core systems [9,10] and also substantial changes in software design, from sequential programming to parallel programming [11]. Recently introduced CUDA/GPGPU computing has the

potential to increase the speedup factor by many times [12]. However, more research work is needed to explore the challenges and opportunities of multithreaded game engines running on multicore/manycore systems.

This paper is organized as follows: Section 2 reviews a number ofrelated published articles. Section 3 describes task and data parallelism concepts with respect to designing gaming engines. Two experiments are conducted for this work: a GPGPU/CUDA-assisted separable convolution filter implementation is presented in Section 4 and a multithreaded multi-object game engine implementation is introduced in Section 5. The experimental results are discussed in Section 6. Finally, this work is concluded in Section 7.

2 Literature Survey

Considerable research work has been done on GPGPU computing and the multithreaded game engine in recent years. Some articles are presented in this section.

Intel has introduced a method of using a "thread pool" to manage task-level parallelism, as discussed in [13]. In a thread pool, each component has one or more tasks that will be queued and threads that are idle or have finished a task will retrieve a task from the queue to run next. Usually, the number of threads matches the number of cores of a particular system. In [14], Intel's thread building blocks (TBBs) are used to implement the multithreaded engine. This threading middleware consists many algorithms and data structures to help developers implement multithreading.

In [15,16], a task tree with a thread pool system to manage dependencies between tasks is used. In this approach, tasks are arranged in a tree where one task has multiple children and one parent. Each task is given a priority order number. The task within the same parent with the same priority number can run in parallel and the next order can only run when tasks from the previous order are completed. In this scheme, the mixture of data level parallelism and task level parallelism shows performance benefits.

In an asynchronous model of game engines, as introduced in [17], each thread runs a task without a synchronizing step. The task depends on the fact that another task will always secure the latest data available to be processed. This allows each component to update at its own frequency. Occasionally, threads must access shared data. Data sharing could limit the effectiveness of this model, depending on the amount of synchronization required. An asynchronous multithreaded game engine is introduced in [18] to improve game performance.

In [19,20], multicore architecture is integrated to expose a multithreaded engine of game programmers to a different number of cores without recompilation of the code. Games are inherently serial, which makes multithreaded application difficult. The first attempt to parallelize the game engine was by running the client and the server on their individual cores using coarse grained threading. The best-case scenario was twice the performance improvement; however, a 1.2 times improvement was seen in single-player mode, in which case, the server needs 20% of the time it takes the client to complete a clock cycle.

A game company called RedLynx implemented multithreading in its game Trials HD [21]. This game uses the Bullet Physics Engine for simulation. The library is optimized in-house for the Xbox 360 CPU and vector units. Workloads are split among all six of the Xbox 360 hardware threads. Physics are handled in one thread and the graphics setup, graphics rendering, game logic, sound networking, and particle systems are handled in the other threads. One thread is used to handle timing, scheduling, and data synchronization between the other threads. One of the three CPU cores is completely reserved for the physics engine. Physics-heavy levels tend to utilize most of the cores' processing time. The graphics setup thread is bottlenecked in the final stage of this optimization.

Current learning algorithms for unsupervised learning models (such as deep belief networks (DBNs) and sparse coding) are too slow for large-scale applications. Therefore, researchers are forced to focus on smaller-scale models. General principles for massively parallelizing unsupervised learning tasks using a GPGPU are developed in [22]. These are able to reduce the time required to learn a four-layer DBN with 100 million free parameters from several weeks to around a single day. For sparse coding, a simple, inherently parallel algorithm leading to a 5- to 15-fold speedup over previous methods was developed.

The GPGPU has become an integral part of today's mainstream computing systems. In [23], four GPU computing successes in game physics and computational biophysics are presented that deliver order-of-magnitude performance gains over optimized CPU applications. Because games have become increasingly limited by CPU performance, offloading complex CPU tasks to the GPGPU yields better overall performance. For the Havok FX [24] game physics package, experimental results show that a single-core CPU implementation (on an Intel 2.9 GHz Core 2 Duo) achieved 6.2 frames per second, whereas the initial GPGPU implementation on an NVIDIA GeForce 8800 GTX reached 64.5 frames per second.

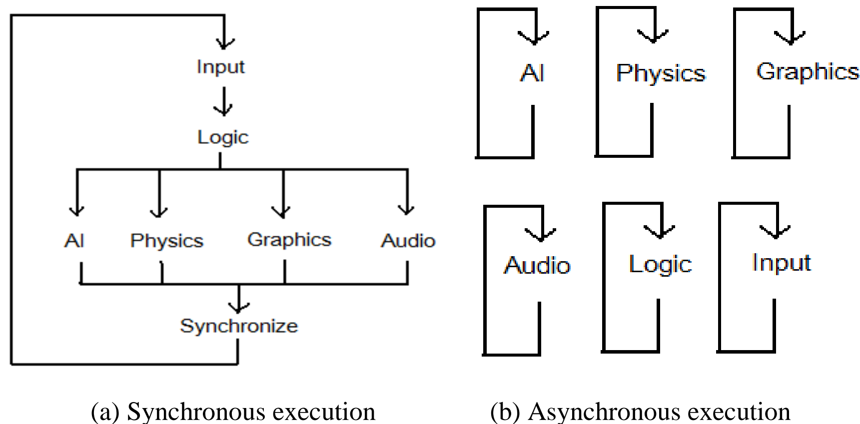
3 Task and Data Parallelism

Task-level and data-level parallelism are important in game engines. They are briefly explained in the following subsections.

3.1 Task Level Parallelism

Task parallelism is the distribution of different tasks across different threads. Task parallelism is used in a game engine by running each component task in its own thread [25-27]. Graphics rendering and physics simulation are good candidates for parallelism because they are usually process-intensive tasks. This model is most likely the simplest and most straightforward way to implement multithreading because the programmer is only required to create and keep the threads running until they are not needed anymore. For every system running in a separate thread, the programmer may need to handle race conditions with mutual exclusions. When using this method of parallelism, there are two models of execution: synchronous and asynchronous.

The synchronous model is where all component tasks must finish in a single clock cycle, as shown in Figure 1(a). At the end of the clock cycle, the application will loop to the beginning to begin the operations again in the same order every time. The components run in parallel after the logic processing stage. The asynchronous model is where component tasks can run and finish on their own time. A component that runs on a thread is independent from the clock cycle of the other threads. This is ideal when there is little communication between components. Figure 1(b) shows the model where all components run in their own loop.



(a) Synchronous execution (b) Asynchronous execution

Figure 1 Task parallelism in a game loop utilizing synchronous and asynchronous techniques.

3.2 Data Level Parallelism

Data parallelism is the distribution of the same type of processing data across different threads. For a game engine, data parallelism is where the same type of data in a component is parallelized in multiple threads [28,29]. As shown in Figure 2, the animation subcomponent in the graphics component is divided into three batches of data for simultaneous processing. The use of this in a game engine is when a component spawns multiple worker threads to process one type of data. If only data parallelism is employed, then the series of different types of operations are sequential and only the data of one type of operation are processed concurrently at one stage. If a data type requires communication among itself, a thread safe communication system must be implemented. This method scales well for a great number of processors because the size of the data for each thread can be divided equally. Communication among the threads can be reduced by grouping the objects that are most likely to interact with each other in the same thread [30].

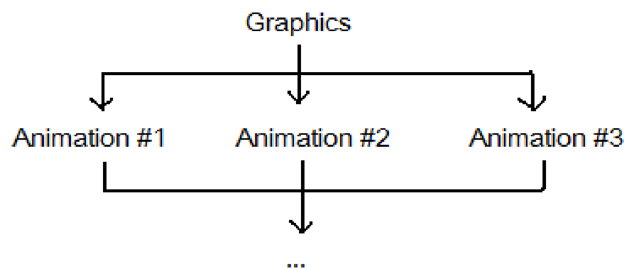


Figure 2 Game engine utilizing data parallelism where graphical objects are divided among three threads.

3.3 Task and Data Parallelism

A combination of task and data parallelism is the optimum approach to exploit multithreading in game engines [15]. Here, each task can run parallel with another task and may spawn several worker threads. A system may have a number of cores, less or more than the number of parallelizable components. In task parallelism, if there are more cores than the number of types of components to be parallelized, then there will be cores that are not used if each type of component runs in a single core. Therefore, to maximize parallelism, data parallelism should be employed to maximize the use of all cores. A system with a high amount of data-parallelism would make it easier to manage tasks that are sequential because there may only be a race condition among the same type of data being parallelized; but it may not fully utilize the concurrency advantage for some components that are decoupled from each other. A system with a high

amount of task-parallelism would cause some cores to be unused because there may be more cores than the number of different types of tasks that can run at the same time, but having a synchronization stage with no mutex (i.e., mutual exclusion) locking can easily be implemented if it is a synchronous model. Mixing task and data parallelism takes advantage of the fact that not all components and data objects of a game engine are completely dependent. In most cases, task parallelism is implemented on different types of components or subcomponents and data parallelism is implemented inside a component or subcomponent.

3.4 Synchronization

Synchronization with respect to multithreading is basically data synchronization and is used to ensure that data are not executed at the same time by two threads. One method for synchronization is by using mutex. Mutex locking in a game engine depends on the multithreading model.

The main drawbacks with mutex locks are overhead, deadlocks, contention, and priority inversion [31]. Acquiring and releasing locks requires some time, thus causing overhead (and decreasing performance). Deadlocks can occur when the order of acquiring a lock leads back to the same lock at the beginning.

There are many synchronization techniques. Another method is to use a message passing system between threads. This avoids the use of mutex locking when passing data. The idea here is to use a common interface among all components and the advantage is a unified model of synchronization, thus avoiding the need to write synchronization code for every component. Other synchronization techniques include reader-writer lock and read-copy-update [32].

4 Separable Convolution Filter

Separable convolution is a technique for fast convolution. It is commonly used in computer vision, image processing, signal processing, etc. Convolution is a mathematical operation on two functions (say, “f” and “g”) that produces a third function (say, “c”). Function “c” is typically viewed as a modified version of one of the original functions (say, “f”) giving an area overlap between the two functions (as illustrated in Figure 3). In the following section, CUDA/GPGPU-assisted separable convolution filter implementation is introduced.

4.1 Separable Filters

A separable filter is a special type of filter that can be expressed as the composition of two 1-D (one dimensional) filters, one on the rows of the image, and one on the columns. For a width n and height m filter kernel, a two-dimensional convolution filter normally requires n*m multiplications for each output pixel. A separable filter can be divided into two consecutive one-dimensional convolution operations on the data and therefore requires only (n + m) multiplications for each output pixel.

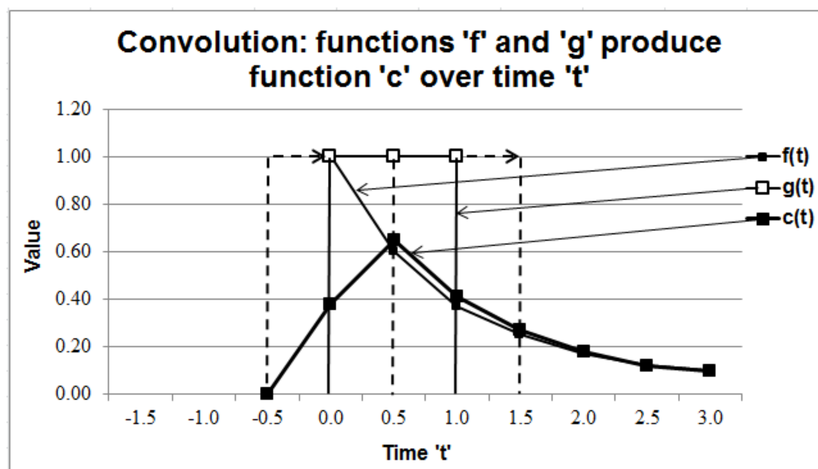


Figure 3 Separable convolution filter–applying function c(t) to some data is the same as applying f(t) followed by g(t).

For example, the 3x3 filter shown below is a separable Sobel [33] edge detection filter. Because applying $\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$ to the data is the same as applying $2\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ followed by $\begin{pmatrix} -1 & 0 & 1 \end{pmatrix}$.

Separable filters offer more flexibility in their implementation, as well as a reduction of the arithmetic complexity and bandwidth usage of the computation for each data point.

4.2 Simple CUDA Implementation

This approach involves the following: (i) a block of the image loaded into a shared memory array; (ii) a point-wise multiplication of a filter-size portion of

the block; and (iii) the sum written into the output image in the device memory. Each thread block processes one block in the image and generates a single output pixel. An illustration of this is shown in Figure 4.To filter the image block, an apron of pixels is required. An apron of pixels occurs around the image block within a thread block for the width of the kernel radius. The apron of one block overlaps with adjacent blocks and, in order to implement properly, requires special attention (for example, threads loading the apron pixels will be idle during the filter computation).The image block apron region is not considered in this work.

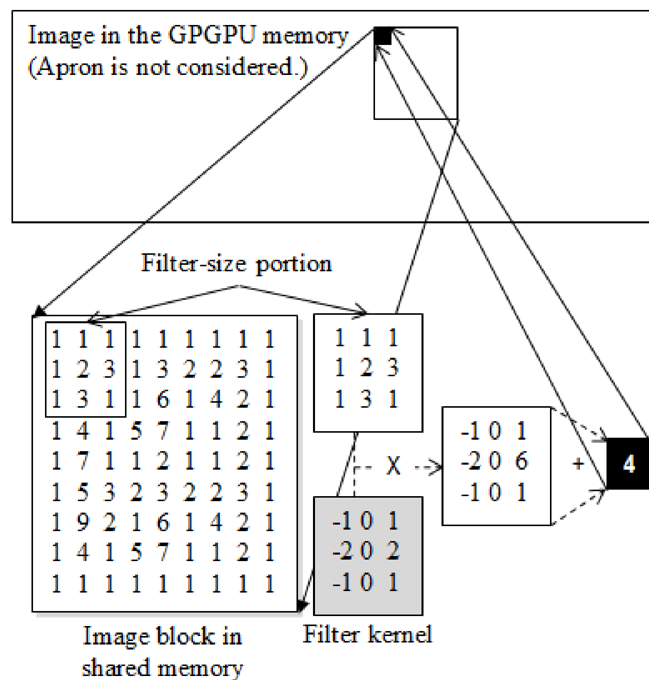


Figure 4 Simple implementation of a separable convolution filter using GPGPU/CUDA technology.

Five major steps are involved in this approach: (i) random input data values are used; (ii) the Gaussian convolution kernel is calculated and copied to a CUDA constant array and because the Gaussian is a symmetric function, the row and column filters are identical; (iii) the CUDA computation grid is configured for the requested image and filter parameters; (iv) row and column filters are applied to the input data; and (v) the resulting image is copied back to the CPU and checked for correctness.

5 MultiobjectInteractive Game Console

A multithreaded multi-object game engine was implemented. The objective of the game is to defend a main base structure against enemy attacks. The player must build defensive structures that will destroy waves of enemies trying to destroy the main base structure. The player will try to survive as many waves as possible. Enemies will become more difficult after every wave. For every enemy destroyed, the player gains credits that can be used to build more defenses. If/when the base is destroyed, the game is over. There may be as many as 20 objects on the screen at one time.

The graphics for this game engine, provided by Ogre3D [34], use a scene graph to represent graphical objects. While graphics rendering is a GPU-intensive operation, the program must call the render operation to send the data to the GPU for rendering. The physics are provided by the Bullet Physics Engine [35], which will run on the CPU only. The multithreading implementation in the Bullet Physics Engine employs data-level parallelism. Kinematic objects, sensors and rigid bodies are used from this engine. AI "pathfinding" is provided by Recast [36]. The pathfinding library will create the navigational mesh in tiles. Because an object is actively residing in the navigational mesh, it becomes an obstacle. Such obstacles must update their position with Recast when they move to ensure that other objects can create a path around other obstacles. Input is provided by Object Oriented Input System (OIS) [37], which has buffered input and un-buffered input. Tinythreads++ [38], a low-level threading library with basic functionalities, is used. This game is implemented using a single-threaded model (STM) and various multithreaded models. The different implementations are explained briefly below.

5.1 Single-Threaded Model

The order of operations in the STM implementation is as follows: (i) capture input (an I/O operation with the operating system); (ii) update input operation (handles input events and queries); (iii) update game logic (which is inherently sequential); (iv) update AI (perform state machine and pathfinding operations); (v) update physics (involves kinematic physics objects); (vi) process navigational mesh updates (updates processing objects); (vii) simulate physics (simulates all physics objects in the world); and (viii) render graphics (from the frame into the screen).

5.2 Multithreaded Asynchronous Model (MAM)

In the MAM implementation, there are two threads, each having an independent clock cycle. Fine-grained mutex locking is used on the data. The order of operations is as follows:

Thread 1:

(i) Capture input; (ii) update game logic; (iii) update AI; (iv) updatephysics; (v) process navigational mesh updates; and (vi) simulate physics.

Thread 2:

(i) Update graphics (sets the transform of the mesh model before rendering); and (ii) render graphics.

5.3 Multithreaded Synchronous Model (MSM)

In this lockless MSM implementation, a synchronization stage occurs between clock cycles. Data synchronization is done in the serial stage only. In the parallel stage, all operations run in parallel, each on a different thread. Threads are created at the beginning of each cycle and destroyed at the end of each cycle. The order of operations of this MSM implementation is as follows:

Serial Stage:

(i) Capture input; (ii) update logic; (iii) update AI; (iv) updatephysics; and (v) update graphics.

Parallel Stage:

(i) Process navigational mesh updates; (ii) simulate physics; and (iii) render graphics.

5.4 MSM with Data Parallelism (MSMDP)

The final MSMDP implementation is a combination of task and data parallelism using the multithreaded synchronous model. This is similar to the synchronous model but the physics involve two worker threads in order to process collision detection. In the physics simulation thread, two more threads are spawned during the collision detection stage. This is considered parallelism within a component. The order of the operations is as follows:

Serial Stage:

(i) Capture input; (ii) update logic; (iii) update AI; (iv) update physics; and (v) update graphics.

Parallel Stage:

(i) Update navigational mesh; (ii) simulate physics; (iii) perform collision detection on object batch 1; (iv) perform collision detection on object batch 2; and (v) render graphics.

6 Results and Discussion

The focus of this work was to explore the impact of GPGPU/CUDA-assisted multithreaded programming on game engine performance. A multiobject interactive game console in an eight-core workstation using an MAM, an MSM, and an MSMDP was implemented. The experimental results are presented in the following subsections.

6.1 Separable Convolution Filter

Three implementations of separable convolution filters were conducted: (i) 8-core CPU only; (ii) 8-core CPU and 144-core GPU without shared memory; and (iii) 8-core CPU and 144-core GPU with shared memory. The execution times as the result of different implementations are shown in Table 1. The kernel filter radius was kept fixed at 8 and the image radius was changed from 128 to 2048. Compared to CPU time, GPGPU times (with and without shared memory) decreased significantly as the image radius increased. It is also observed that the shared memory GPGPU implementation took the least amount of time to solve the problem.

Table 1 CPU and GPU time (kernel radius = 8).

Image Radius	CPU Time (msec)	GPGPU Time (msec)	
		Without Shared Memory	With Shared Memory
128	2.05	0.50	0.20
256	11.03	1.00	0.40
512	34.46	3.30	0.80
1024	140.99	13.10	2.40
2048	568.87	54.70	9.80

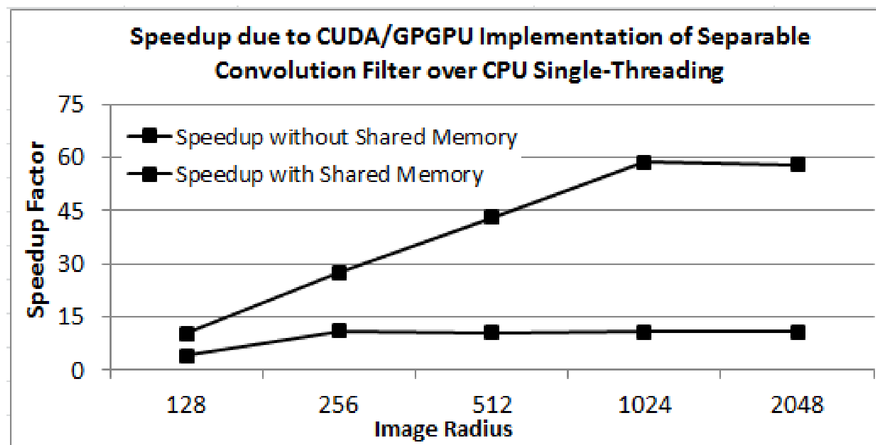


Figure 5 Speedup due to GPGPU/CUDA-based parallel implementation.

According to the experimental results, the speedup (ratio of single-thread time to multithread time) increased as the image radius increased from 128 (see Figure 5). It is observed that the GPGPU without shared memory implementation hit the performance wall at image radius 256, whereas the speedup of GPGPU with shared memory implementation hit the performance wall at image radius 1024.

6.2 Multiobject Interactive Game Console

Consider the number of frames generated due to various implementations for a 30-second simulation of the game. As illustrated in Figure 6, the multithreaded synchronous game engine with data parallelism generated more threads than any other console. Then, consider the time required to process different frames. As shown in Figure 7(a), the single-threaded game console took the highest amount of time to process a frame compared with other multithreaded models.

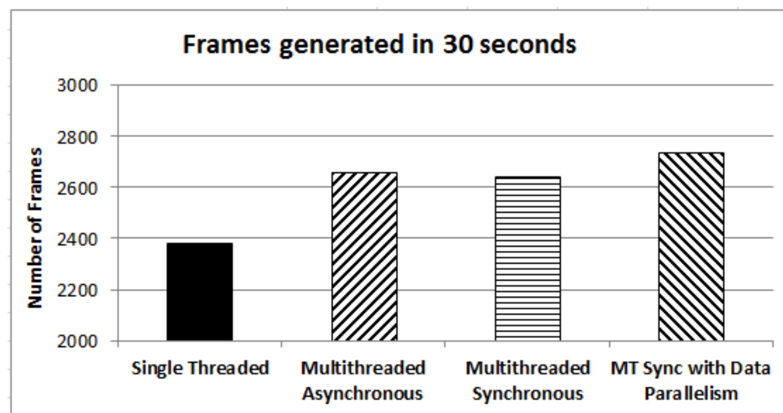
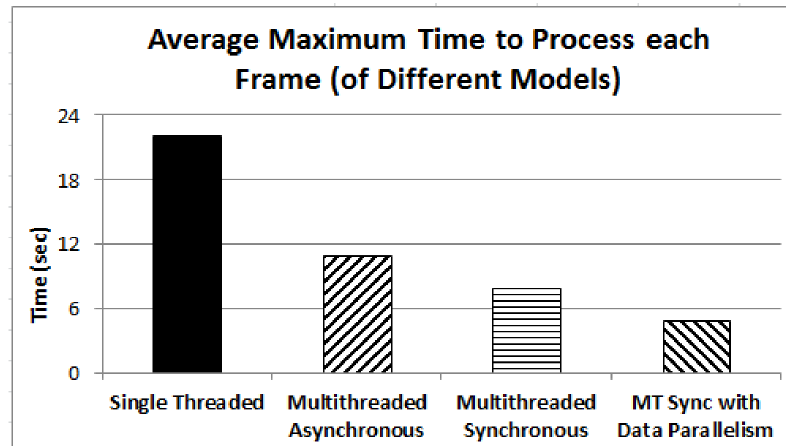


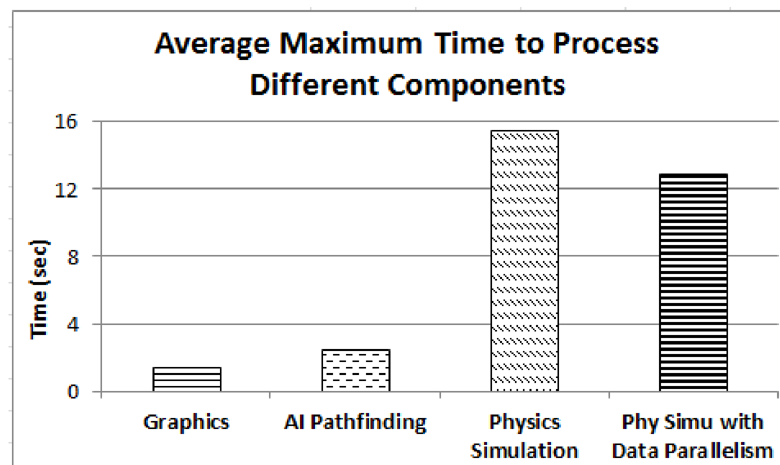
Figure 6 Average number of frames generated for 30-second game execution.

It can also be seen that the multithreaded synchronous game-console with data parallelism took the lowest amount of time to process a frame. This is due to the large amount of mutex locking required to run a clock cycle when a threaded component communicates with many different components in other threads. This occurs when using two sets of copies of the shared data. The asynchronous model of execution will most likely require more memory to implement and as such it is only recommended in cases where user experience can be improved by components running on their own clock cycles. We notice that the time to complete a frame was also very inconsistent in this model, sometimes equal or slightly slower than the single-threaded version.

Now, we consider the time required to process different components. As shown in Figure 7(b), the components were found to have different processing times.



(a) Time for each frame



(b) Time for each component

Figure 7 Average maximum time to process frames and components.

The physics component consumes most of this because the game uses a considerable number of physics objects and operations. In a coarse-grained implementation, such as the synchronous model, the component that takes the longest time becomes the bottleneck because other threads must wait for that thread to complete. Using multithreading within a component itself seems to improve its performance. The physics component with data parallelism for processing collision detection improved the overall performance. The

combination of task and data level parallelism achieved the best time in the ideal hardware platform.

Finally, the speedup due to multithreaded implementation over single-threaded implementation of the game is considered. The maximum speedup factor (about 5) was achieved by the multithreaded synchronous model with data parallelism (see Figure 8). Here, speedup was calculated using the frames generated and time to process the frames as expressed in Eq. (1).

$$\text{Speedup} = \frac{\textit{Frames generated due to MT}}{\textit{Frames generated due to ST}} \times \frac{\textit{Time due to ST}}{\textit{Time due to MT}} \quad (1)$$

where MT means multithreading and ST means single-threading.

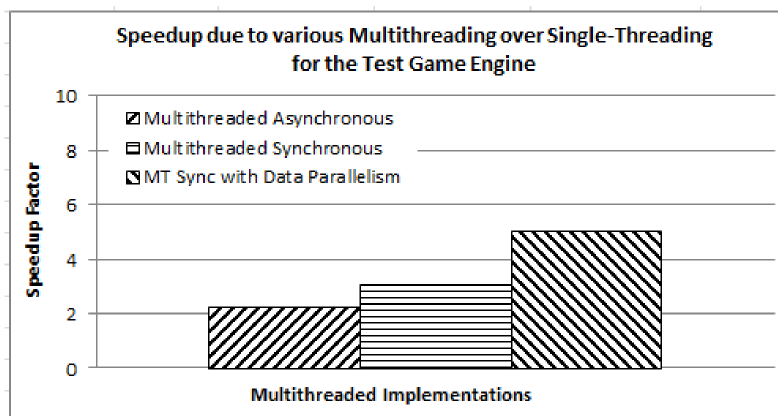


Figure 8 Speedup due to the multithreaded implementations (MAM, MSM, and MSMDP) compared to the single threaded model (STM).

7 Conclusions

Single-processor multithreaded game engines suffer from poor performance due to the lack of hardware support. The multicore/many core CPU/GPGPU platform shows promise for improving the performance of multithreaded game engines. However, multithreading in multicore system architecture introduces challenges, such as data parallelism and synchronization. The work reported here investigated the challenges and rewards of GPGPU-based implementation of a separable convolution filter and a multithreaded test game console. Various single-threaded and multithreaded models with and without data parallelism were implemented.

In most implementations of game engines, data parallelism seems to fit identical types of components, while task parallelism fits different types of components. The synchronous model designed with task and data parallelism takes advantage of concurrency and is highly scalable for any number of cores. Implementing the asynchronous model is more difficult than implementing the synchronous model. The asynchronous model is suitable for certain components that are not used for rendering at every single clock cycle of the render loop, such as networking and resource loading. The results of our experiment of game engine implementation support the fact that multithreaded models outperform the single-threaded model; MSMDP generates more frames and takes less time to process frames (see Figures 6 and 7(a)). Game components normally take different amounts of processing time (see Figure 7(b)). The speedup factor due to MSMDP implementation with respect to the single-threaded implementation was about 5 (see Figure 8). However, according to the experimental results from shared-memory GPGPU implementation of the separable convolution filter, the speedup factor was more than 61. Therefore, GPGPU/CUDA-based multithreaded parallel programming can be used to improve game engine performance.

A plan for a future endeavor is to implement the entire test game engine using the CPU/GPGPU platform and evaluate its performance and power consumption.

References

[1] *Graphics Processing Unit*, Wikipedia, https://en.wikipedia.org/wiki/Graphics_processing_unit (1 August 2013).

[2] Swaine, M., *New Chip from Intel Gives High-Quality Displays*, Intel Press, http://en.wikipedia.org/wiki/Graphics_processing_unit (1 August 2013).

[3] *CUDA*, Nvidia, http://www.nvidia.com (1 August 2013).

[4] Kruger, J. & Westermann, R., *Linear Algebra Operators for GPU Implementation of Numerical Algorithms*, International Conf. on Computer Graphics and Interactive Techniques, 2005.

[5] Liepe, J., Barnes, C., Cule, E., Erguler, K., Kirk, P., Toni, T. & Stumpf, M.P., *ABC-Sysbio – Approximate Bayesian Computation in Python with GPU Support*, Bioinformatics Journal, **26**(14), pp. 1797-1799, 2010.

[6] Berberich, S., *Video Games Starting to Get Serious*, Gazette.net, http://ww2.gazette.net/stories/083107/businew11739_32356.shtml, (1 August 2013).

[7] Waele, R.D., *Gaming: Mobile and Wireless Trends for 2008*, M-trends.org, http://www.m-trends.org/2008/01/mobile-and-wireless-trends-for-2008.html (1 August 2013).

[8] Parberry, I., *Intro to Bullet Physics*, larc.unt.edu, http://larc.unt.edu/ian/classes/fall11/csce4215/notes/bulletphysics.pdf, (1 August 2013).

[9] Brodkin, J*., Shift to Multicore Processors Inevitable, but Enterprises Face Challenges*, Network World, http://www.networkworld.com/news/2008/022708-multicore-processors.html (1 August 2013).

[10] Schauer, B., *Multicore Processors – A Necessity*, ProQuest Discovery Guides, http://www.csa.com/discoveryguides/ multicore/review.pdf (1 August 2013).

[11] Sutter, H., *The Free Lunch Is Over: A Fundamental Turn toward Concurrency in Software*, Dr. Dobb's Journal, **30**(3), pp. 1-7, 2005.

[12] *Designing the Framework of a Parallel Game Engine (PGE)*, Intel, http://www.intel.com (1 August 2013).

[13] *Threading Building Blocks (TBB)*, Intel, http://www.threading buildingblocks.org/ (1 August 2013).

[14] Rhalibi, A.E., England, D. & Costa, S., *Game Engineering for a Multiprocessor Architecture*, School of Computing and Mathematical Sciences, Liverpool John Moores University, 2005.

[15] Harbour, J.S., *Multi-Threaded Game Engine Design*, Course Technology PTR (1st ed.), ISBN-10: 1435454170, ISBN-13: 978-1435454170, 2010.

[16] Lake, A. & Gabb, H., *Threading 3D Game Engine Basics*, Gamasutra, 2005, http://www.gamasutra.com/view/feature/2463/threading_3d_game _engine basics.php (1 August 2013).

[17] Vries, A.D., *Multithreaded Renderloop,* slapware.eu, http://blog.slapware.eu/game-engine/programming/multithreaded-renderloop-part1/ (1 August 2013).

[18] Leonard, T., *Dragged Kicking and Screaming: Source Multicore*, Valve Corporation, GDC 2007, pp. 8-13, 2007.

[19] Gasior, G., *Valve's Source Engine Goes Multi-Core*, techreport, 2006, http://techreport.com/review/11237/valve-source-engine-goes-multi-core (1 August 2013).

[20] Aaltonen, S. & Ilvessuo, A., *Tech Interview: Trials HD*, Richard Leadbetter, Eurogamer, 2009, http://www.eurogamer.net/articles/digital foundry-tech-interview- trials-hd (1 August 2013).

[21] Gadd, K., *Threading and Your Game Loop*, #Alt-DevBlogADay, http://www.altdevblogaday.com/2011/07/03/threading-and-your-game-loop/ (1 August 2013).

[22] NVIDIA *Developer Zone: CUDA Samples*, http://docs.nvidia.com/cuda/cuda-samples/ (1 August 2013).

[23] Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E. & Phillips, J.C., *GPU Computing: Graphics Processing Units--Powerful, Programmable, and Highly Parallel--Are Increasingly Targeting General-Purpose Computing Applications,* in: the Proceedings of the IEEE, **96**(5), pp. 879-899, 2008.

[24] Govindaraju, N.K., Henson, M., Lin, M.C., &Manocha, D., *Interactive Visibility Ordering of Geometric Primitives in Complex Environments*, in: Proc. 2005 Symp. Interact. 3D Graph. Games, pp. 49-56, 2005.
[25] Ooste, J.V., *3D Game Engine Programming: Helping You Build Your Dream Game Engine,* 2011, http://3dgep.com/?p=1821 2011 (1 August 2013).
[26] Guevara, M., Gregg, C., Hazelwood, K., & Skadron, K., *Enabling Task Parallelism in the CUDA Scheduler,* in PEMA. 978-1-4244-6443-2/10/ IEEE, 2009.
[27] Baumstark, L.Jr. & Wills, L., *Exposing Data-Level Parallelism in Sequential Image Processing Algorithms*, in Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02), 1095-1350/02 IEEE, 2002.
[28] *Domino Research: Using Data-Parallel SIMD Architecture in Video Games and Supercomputers*, IBM, http://domino.research.ibm.com/ comm/research.nsf/ pages/r.arch.simd.html (1 August 2013).
[29] Mönkkönen, V., *Multithreaded Game Engine Architectures*, Gamasutra, 2006, http://www.gamasutra.com/view/feature/130247/multithreaded_ game_engine_.php?page=3 (1 August 2013).
[30] Kriemann, R*., Implementation and Usage of a Thread Pool based on POSIX Threads* in MPI MIS Leipzig, Report 2/2003, 2004.
[31] Cronin, E., Kurc, A.R., Filstrup, B., & Jamin, S., *An Efficient Synchronization Mechanism for Mirrored Game Architectures (Extended Version)*, Kluwer Academic Publishers, 2003.
[32] Davies, L., *Examples of Multi-Threading in Games*, Intel, 2006.
[33] *An Introduction to Edge Detection: The Sobel Edge Detector*, Generation5, http://www.generation5.org/content/2002/im01.asp (1 August 2013).
[34] *OGRE – Open Source 3D Graphics Engine,* www.ogre3d.org/ (1 August 2013).
[35] *Game Physics Simulation*, bulletphysics.com/ (1 August 2013).
[36] Recastnavigation: Navigation-Mesh Construction Toolset for Games, code.google.com/p/recastnavigation/ (1 August 2013).
[37] *Object Oriented Input System,* sourceforge.net/projects/wgois/ (1 August 2013).
[38] *TinyThread++ – Portable thread library for C++*, www.tinythreadpp. bitsnbites.eu/ (1 August 2013).