International Journal of Computer Vision and Signal Processing, 4(1), 1-10(2014) ORIGINAL ARTICLE

Investigating the Impact of Data Parallelism and GPU Technology on Computer Gaming

Abu Asaduzzaman*

Assistant Professor in Computer Systems and Architecture and Director of CAPPLab
Wichita State University, 1845 Fairmount St., JB-253, Wichita, Kansas 67260-0083, USA

Deepthi Gummadi

Master's Student in Computer Networking
Wichita State University, 1845 Fairmount St., JB-245, Wichita, Kansas 67260-0083, USA

Abstract

According to the current design trends, multithreaded multicore processors will be ubiquitous in every device. In computer gaming, chip-makers are adding more cores to fulfill the next generation performance requirements. A game engine has many 'tasks' and data parallelism is an important technique for concurrent execution of these tasks. However, effective implementation of multithreaded computer games has challenges including concurrent/parallel processing, data and task level parallelism, and thread synchronization. In this paper, we investigate the impact of data parallelism and graphics processing unit (GPU) technology on multicore game engines. We implement a multi-object interactive game engine in an 8-core workstation using single-threaded model (STM) and various multithreaded models. We also implement a high quality DXT compression (a family of implementations of the S3 texture compression algorithm) using GPU technique. Experimental results show that multithreaded synchronous model with data parallelism (MSMDP) outperforms STM by reducing execution time up to 50%. Results also show that for 448-thread data parallelism, more than 81x speed up can be achieved by applying GPU computing.

Keywords: Computer game, GPU technology, multicore processor, multithreaded programming, speedup.

IJCVSP
International Journal of Computer Vision and Signal Processing

ISSN: 2186-1390 (Online)
http://www.ijcvsp.com

Article History:
Received: 7 January 2014
Revised: 15 April 2014
Accepted: 27 April 2014
Published Online: 28 April 2014

1. INTRODUCTION

Video game engines normally provide software frameworks that developers use to create game consoles running on personal computers. There are many components in a simple modern game engine. In a single threaded game engine, important components (according to the flow of operations) are: Input, Game Logic, Artificial Intelligence (AI), Physics (engine for collision detection/response), Audio (for sound), and 3D Graphics. Figure 1 illustrates the flow of operations in a single threaded game engine. A rendering engine called 'renderer' is required for 2D or 3D graphics. A graphics package may include scene graph, culling and sorting, skeletal animation and rendering. Inside a component, there may be many subcomponents that glue together to form a complete package. Some of these components can be middleware to make programming easier. The operations go around from the beginning to the end every time. An operation from start to finish is known as a clock cycle. It is also known as a render loop as the data processed by the components are used for rendering in one loop. A new frame is drawn after every cycle.

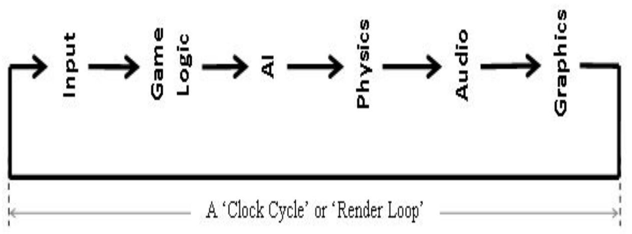

Figure 1: Flow of operations in a single threaded game engine.

*Corresponding author
Email address: `Abu.Asaduzzaman@wichita.edu` (Abu Asaduzzaman)

In addition to standalone game machines, game engines are nowadays being used for educational, engineering, and scientific applications [1]. To fulfill the high performance requirements, game engines are adopting new hardware technologies like multicore CPUs [2] and software technologies like multithreaded parallel programming [3, 4].

One or more from many available parallel programming techniques can be used in game engine programming. When components in a game engine are originated from many different middleware, the design of the library will most likely dictate which one is more suitable to be used. Some middleware such as Bullet Physics library includes multithreading in their application programming interface (API) [5]. Depending on the type of multithreading model used, some level of data redundancy is required to improve performance. Therefore, a mechanism to ensure data/cache coherency is needed in the implementation.

The number of cores in a processor has increased but the speed of the core has not changed much in the recent years. So, multithreading can be very helpful to get as much performance out of a system as possible to the advancement of video game technologies. Currently available middleware used in high-level API, like Open MPI, make the parallel implementation a challenge. Therefore, various methods should be evaluated when implementing multithreading in a game because the components usually never work the same way. One multithreading technique might not be suitable for a particular API of a component because of the way it is built. As it is hard to objectively calculate which implementation is needed to properly optimize a multithreaded application, optimization of multithreaded game engines requires a lot of experimentations.

Finally, the success of modern computer games significantly depends on the new innovation the shift from single core to multi-core systems [6, 7]. Such innovation will also require substantial changes in software design, from sequential programming to parallel programming [8]. Recently introduced compute unified device architecture (CUDA)/general-purpose GPU technology has potential to increase the speedup factor by many times [9, 10, 11]. Therefore, more research work is needed to explore the challenges and opportunities of multithreaded game engines running on multicore/manycore systems.

The rest of the paper is organized as follow: Section 2 reviews some related published articles. Section 3 explains the impact of data/task parallelism, synchronization, and GPU computing. An experiment of data parallelism in a video game engine is presented in Section 4. Section 5 presents another experiment of high quality DXT compression using GPU computing. Section 6 discusses some simulation results. Finally, this paper is concluded in Section 7.

2. LITERATURE SURVEY

In the recent years, gaming industry is rivaling the movie and music industries. The game industry has surpassed the movie and music industry in U.S. in 2005 and 2007, respectively. In 2008, the game industry surpassed the music industry in the U.K. and is expected to surpass DVD sales in the future [12]. The desire for more complex game elements is driving the game industry forward. Multithreaded parallel programming has potential to implement complex game engine. However, multicore CPU (not graphics processing unit GPU) is a relatively new technology, especially in the game development world.

Most vendors are adopting multicore processors to their products. Multilevel cache memories are common in multicore processors. The cache memory hierarchy normally has level-1 cache (CL1), level-2 cache (CL2), and main memory. In most cases, CL1 is split into instruction (I1) and data (D1) caches and CL2 is a unified cache [2]. Performance and power consumption are impacted by cache misses, increased usage of main memory, and poor cache memory arrangement. Using communication that is too fine grained can cause the cache to be underutilized [13]. A thread reading the data can receive the set of multiple data objects first and then process all of them. The effective size of the set of data objects can be calculated by the size of each line and the size of the cache line size of the cores. When communication is too coarse grained, capacity misses could happen when there are large amount of objects being copied that are larger than the cache size. Some processors use shared cache (like shared CL2) that can be accessed by multiple cores. When more cores access the same cache, there will be overhead of managing the use of the cache by multiple processors. Multicore systems are very suitable for multithreaded processing as multiple threads can be executed on multiple cores at the same time [14].

Task level parallelism is a popular method for game engine multithreading, where components run asynchronously in their own loop or synchronously in a single loop with multiple forks and joins. An asynchronous model of game engines has been introduced in [15]. In this model, as soon as a task is done, it will run immediately from the beginning. Data sharing could limit the effectiveness of this model depending on the amount of synchronization required. The multithreaded game engine introduced in [16] is an asynchronous model that uses multiple render states to buffer data. In this implementation, there is one world state and three render states. For a game engine, data parallelism is where the same type of data in a component is parallelized in multiple threads. The use of this in a game engine is when a component spawns multiple worker threads to process one type of data. To properly scale a multicore game engine, task parallelism and data parallelism have to be employed as introduced in [17].

Load balancing is a technique where the tasks are arranged to minimize thread idling by letting a thread process more components. Load balancing can be done either by hardcoding the order or having a system that dynamically manage the tasks [18]. Load balancing may be automated by the use of a thread pool. When the cores are used more evenly, it gives more opportunities for developers to implement more distributed and parallel gameplay elements. When a task can vary in time for processing, an automated load balancing mechanism might be required to properly balance the load in most situations.

Intel Corporation has used a thread pool mechanism to manage tasks as discussed in [19]. 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. This ensures that there will be a maximum amount of threads that can run at the same time. In [20], a multicore architecture is integrated to expose multithreaded concept to game programmers to different number of cores without recompilation of code. They use coarse grained threading for this experiment. The benefit of this is they are forced to face systems that are not thread safe or not thread efficient. Shared data access is introduced as there are problems in multithreading with global data, static data, and singleton objects. Mutex locking is found to be slow, error prone, and failed to scale; however, it establishes a slow but stable baseline of the architecture.

RedLynx has implemented multithreading in their game Trials HD [21]. It uses the Bullet Physics Engine for physics simulation. The library is optimized in-house for the Xbox 360 CPU and the vector units. The workload is split into all six of the Xbox 360 hardware threads. Physics are handled in one thread. Graphics, game logic, etc. are handled in the other threads. The graphics rendering API, DirectX 11 has also been designed to take advantage of multithreaded processing in some elements. One of the new features is the threaded asynchronous resource loading [22]. Developers can load rendering resources in a thread-safe way and use them concurrently with the rendering operation. Multiple threads can start loading resources when they are required. This technique shows performance improvement.

The cluster fit algorithm is presented in [23]. The quality of the resulting images due to traditional real-time implementation of DXT compression is very poor. More efficient compression algorithm is needed to produce high quality results, which is computationally extensive. With the increasing number of assets and texture size in recent games, the time required to process those assets is growing dramatically. CPU-only implementation of such massive algorithm may not be feasible. The nature of these compression algorithms makes them suitable to be parallelized and adapted to the GPU. The cluster fit algorithm is used as a reference to compare the performance of our GPU implementation of a high quality DXT compression algorithm. The implementation detail can be found in [24].

In this work, we use thread pooling technique and graphics rendering API to develop a multithreaded game engine; and we evaluate the impact of data parallelism, synchronization, and GPU computing on performance of multicore computer games.

3. IMPORTANT GAME ENGINE TECHNIQUES

Important techniques used in game engines to improve performance include task level parallelism, data level parallelism, task and data level parallelism, synchronization, and load balancing. We briefly explain these concepts in the following subsections.

3.1. Data Level Parallelism

Data parallelism is the distribution of the same type of data to process across different threads. For a game engine, data parallelism is where the same type of data in a component is parallelized in multiple threads [25, 26]. As shown in Figure 2, the animation subcomponent in the graphics component is divided into 3 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.

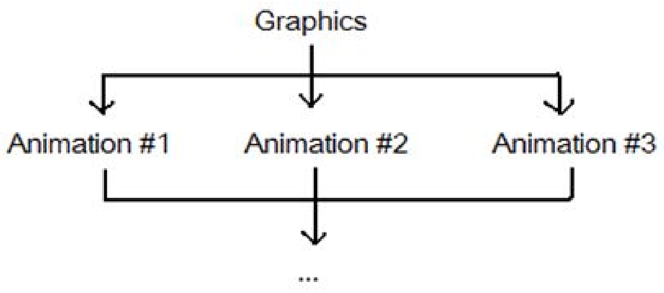

Figure 2: Game engine utilizing data parallelism (a graphical object is divided among three threads).

If only data parallelism is employed, the series of different types of operations are sequential, only the data of a type of operation are processed concurrently at one stage. If the type of data requires communication among themselves, a thread safe communication system has to be implemented. This method scales well for many number of processors because the size of the data for each thread can be divided equally. It may also be easy to balance the load among multiple cores because there is only one type of object being processed concurrently. When the data type does not share data with each other, this method of parallelism can easily be implemented to scale on any number of threads. Communication among the threads can be reduced by grouping the objects that are most likely to interact with each other in the same thread [27].

3.2. Task Level Parallelism

Task parallelism is the distribution of different task across different threads. The use of task parallelism in game engine is by running each component task in its own thread [28, 29]. Graphics rendering and physics simulation are good candidates for parallelism as 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 or synchronization stage. When using this method of parallelism, there are two model of execution, the synchronous model and asynchronous model.

The synchronized model is where all the tasks of the components must finish in a single clock cycle. At the end of the clock cycle, the application will loop to the beginning to start the operations in the same order every time. The components run in parallel after the logic processing stage. To share data among the threads, a synchronization stage can be used in between the clock cycle in this model. The management of shared data should be easier to manage as all there is only one synchronization stage every clock cycle. Mutual exclusion is not required for synchronization using this method.

The asynchronous model is where the tasks of the components can run and finish at their own time. A component that runs on a thread is independent from when the clock cycle of the other threads. This is ideal when there is little communication between components. In an asynchronous model all the components run in their own loop. Some components that do not always have new data for a single frame are usually implemented asynchronously such as resource loading, player input and networking.

3.3. Task and Data Parallelism

Both task parallelism and data parallelism have to be employed in order to properly implement a game engine that will scale properly and fully utilize parallelism for various numbers of cores. A mixture of task and data parallelism is an optimum approach to exploit multithreading in game engines [30].

In this combination, each task can run parallel with another task and may spawn several worker threads. A system may have 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 component to be parallelized, then if each of the types of component runs in a single core, there will be cores that are not used. Therefore, to maximize parallelism, data parallelism should also be employed to maximize the use of all cores. A highly data-parallelism design would make it easier to manage tasks that are sequential as there may only be race condition among the same type of data being parallelized but may not fully utilize the concurrency advantage for some components that are decoupled from each other. A highly task-parallelism design would cause some cores to be unused as 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 locking can be easily implemented if it is the 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 the different types of components or subcomponents and data parallelism is implemented inside a component or subcomponent.

When using a combination of these models, creation of too many threads can occur when there are limited amount of cores in the platform. Too many threads in a system can cause thread switching. The use of a thread pool can help limit the amount of threads being created by turning each operation into a task and by queuing up the tasks into a list.

3.4. Synchronization

Synchronization with respect to multithreading is basically data synchronization. Synchronization is used to make sure that data are not executed at the same time by two threads. One method for synchronization is with the use of mutex (i.e., mutual exclusion). Use of a mutex locking in a game engine depends on the multithreading model. When designing a multithreaded game engine with many objects, fine grained locking and coarse grained locking must be considered. Fine grained locking is locking performed on small amount of data at one period while coarse grained locking is locking performed on large amount of data at one period. Fine-grained locking increases the overhead compared to coarse-grained locking. Coarse-grained locking increases lock contention where a thread requests for a lock in a mutex that is still being locked.

The main drawbacks with mutex locks are overhead, deadlocks, contention, and priority inversion [31]. Acquiring and releasing locks will require some time causing overhead. Deadlocks can happen when the order of acquiring a lock leads back to the same lock at the beginning. Contention is cause by having to waiting for another component to release a lock so that a lock can be acquired. Priority inversion happens when a thread with higher priority than another thread is preempted by a medium priority task.

In some cases, lockless algorithm can be used. In those cases, a game engine is designed so that mutex locking is entirely avoided. The easiest method is to have a synchronization stage where all processes must run in sequence. It should be noted that the lockless algorithm may not be possible in every scenario depending on the architecture of the game engine.

Another method is to use a message passing system between threads. This avoids the use of mutex locking when passing data. The idea is to use a common interface between all components. The advantage of this is a unified model of synchronization avoiding the need to write synchronization code for every component. Some other synchronization techniques include reader-writer lock and read-copy-update [18].

3.5. Threadpool

Thread pool is a system where tasks are queued in a list and assigned to any available threads that are idle [32].

In a thread pool, threads are created and reused until the application ends. It also makes managing the maximum number of threads (something that can be configured) easier by separating the tasks from the threads. By setting the thread affinity, a thread can be locked into a core or more to make sure that all the threads will run in parallel. Creating more threads than the number of cores available will cause thread switching between multiple threads. Using the thread pool, tasks will be queued in the pool and threads that have just completed a task can grab a new task from the pool. Figure 3 displays the basic operation of a thread pool. A thread that has just completed a task will request for a new task in queue. The maximum number of threads assigned to the pool reduces or avoid thread switching by letting the task wait for an idle thread.

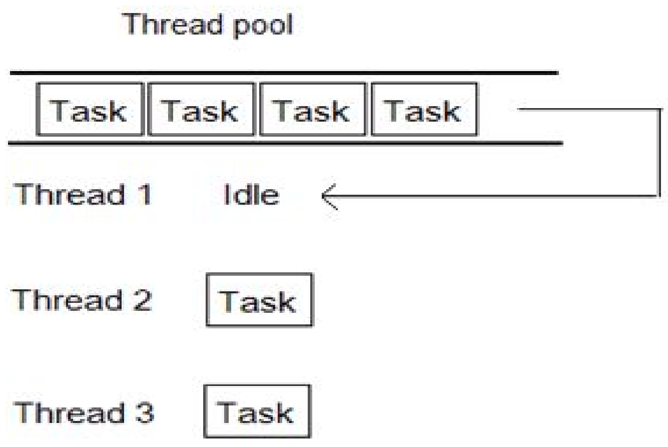

Figure 3: Basic operations of a thread pool.

Thread pool ensures that there will always be a maximum amount of threads that can run at the same time. Usually the number of threads should match the number of cores (aka, hardware threads) of a particular system and the affinity of each thread can be set in the manner of one thread for each core. Task level parallelism and data level parallelism can be employed using the thread pool. For task parallelism, it is as straightforward as queuing a task. For data parallelism, the data have to be decomposed into sub-task batches of data. Using a thread pool is a way to avoid idle threads so the application can take full advantage of concurrency. Thread pool should be able to run in synchronous mode when synchronization stage is required to be flexible for different models of multithreading.

3.6. Load Balancing

Load balancing is where the tasks are arranged to minimize thread idling by letting a thread process more components. This may be required when more threads are being created then the available cores or some tasks take more time to be completed than the other tasks.

Some threads may complete faster than other threads. The cores that have completed their tasks faster (say, Core 1 and Core 2) are idle until the next frame or synchronization stage. This happens when using a coarse grained multithreading solution without load balancing. If the tasks/threads of Cores 1 and 2 are combined together and assigned to Core 1, other core (i.e., Core 2) becomes available to execute another thread. This makes efficient use of processing resources even when using a coarse grained solution in game engine.

Load balancing can be implemented either by hardcoding the order or having a system that dynamically manage the tasks [13]. Load balancing may be automated by the use of a thread pool. When the cores are used more efficiently, it gives more opportunities for developers to implement more complex gameplay elements. When a task can vary in time while being processed, an automated load balancing mechanism might be required/helpful to properly balance the load and improve performance.

4. EXPERIMENT 1: Data Parallelism

In this section, we introduce a multi-object game engine, named Tower Defense Game (TDG), that we implemented in our laboratory to investigate the impact of multicore processors on computer game performance. The initial version of TDG is developed using middleware from various sources. Then, various parallelism techniques are applied to improve performance. We discuss the game policy and various implementations of the game below.

4.1. TDG Policy

The objective of the game is to defend a main base structure. A screenshot of the game is shown in Figure 4. The player has to 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 get harder after every wave. For every enemy destroyed, the player gains credits which could be used to build more defenses. Currently, there are three possibilities to terminate the game: (i) the player defends the main base structure for 3 minutes (the player wins the game), (ii) the main base structure is destroyed (the player loses the game), and (iii) abnormal termination.

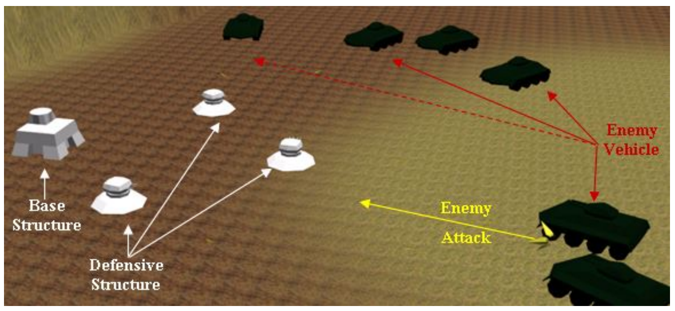

Figure 4: Screenshot of the Tower Defense Game in simulation; enemy vehicles are attacking the base defenses.

4.2. Components for TDG

For the initial version of TDG, Graphics is provided by Ogre3D. It uses a scene graph to represent graphical objects. While graphics rendering is a GPU intensive operation, a program has to call the render operation to send the data to the GPU for rendering. Physics is provided by Bullet Physics Engine that will run on the CPU only. The multithreading implementation in Bullet Physics employs data level parallelism. Kinematic objects, sensors and rigid bodies were used from this engine. AI pathfinding is provided by Recast. The pathfinding library creates the navigational mesh in tiles. As an object is actively residing on the navigational mesh, it becomes an obstacle. These obstacles must update their position with Recast when it moved to ensure that other objects can create a path around other obstacles. Input is provided by Object Oriented Input System. It has buffered input and un-buffered input. The threading library used is tinythreads++. It is a low level threading library with basic functionalities.

4.3. Different Implementations

The video game engine is implemented using single-threaded model and various multithreaded models (with and without synchronization). We briefly explain different implementations below.

- Single Threaded Model (STM): We start with a single threaded implementation. The first operation is to capture input events and put it in a buffer; this is an I/O operation with the operating system. The last operation is to render graphics frame into the screen; it has to read from the list of data objects that it keeps. The order of operations in the single threaded implementation is:

  a) Capture input (an I/O operation with the operating system)
  b) Update input operation (handles input events and queries)
  c) Update game logic (this part is inherently sequential)
  d) Update AI (perform state machine and pathfinding operations)
  e) Update physics (Kinematic physics objects are used)
  f) Process navigational mesh updates (updates processes objects)
  g) Simulate physics (simulates all the physics objects in the world)
  h) Render graphics (from the frame into the screen)

- Multithreaded Asynchronous Model (MAM): In the multithreaded asynchronous implementation, there are two threads; each thread has independent clock cycle. Fine-grained mutex locking is used on the data. The update graphics stage in this model reads the data buffered from the updates of the other thread. During the rendering stage, no internal data can be modified. The logic, AI, and physics are to set the graphics data many times. The order of operations is:

  Thread 1:

  a) Capture input
  b) Update game logic
  c) Update AI
  d) Update physics
  e) Process navigational mesh updates
  f) Simulate physics

  Thread 2:

  a) Update graphics (sets the transform of the mesh model before rendering)
  b) Render graphics

- Multithreaded Synchronous Model (MSM): In this lockless implementation, there is a synchronization stage in between clock cycles. Data synchronization is done in the serial stage only. In the parallel stage, all the operations run in parallel, each on a different thread. Threads are created at the beginning and destroyed at the end of each cycle. The order in which they run is not important because they keep their own private copy of the shared data. The order of operations is:

  Serial Stage:

  a) Capture input
  b) Update logic
  c) Update AI
  d) Update physics
  e) Update graphics

  Parallel stage: (One possible order)

  a) Process navigational mesh updates
  b) Simulate physics
  c) Render graphics

- Multithreaded Synchronous Model with Data Parallelism (MSMDP): The final implementation is a combination of task and data parallelism using the multithreaded synchronous model. It is similar to the synchronous model but the physics will have 2 worker threads 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:

  Serial stage:

  a) Capture input
  b) Update logic

c) Update AI
d) Update physics
e) Update graphics

Parallel stage: (One possible order)

a) Update navigational mesh
b) Simulate physics
c) Perform collision detection on object batch 1
d) Perform collision detection on object batch 2
e) Render graphics

4.4. Important Parameters

In this work, we develop a multi-object game engine using C++ in a multicore computer. We briefly discuss some important parameters in this subsection. The system configuration parameters for the workstation are summarized in Table 1. The workstation is an 8-core (dual-processor, quad-core per processor) system from Intel, runs at 2.13 GHz, and has 6 GB of RAM. The operating system used is the Linux Debian 6.0.

Table 1: System Parameters.

Parameter	Description
CPU	2x Quad-Core Intel Xeon E5506
Cores on CPU	8
CPU Speed	2.13GHz
RAM	8GB DDR3 (3x1GB 1333MHz)
Mainboard	Dell 0CRH6C
Chipset	Intel X58 I/O + ICH10R
GPU	Nvidia, Tesla C2075 448-core, 1.15GHz
Operating System	Linux (Debian)

Output parameters include: the number of frames generated per minute, processing time for each frame, processing time for each component, and speedup factor, S(P) = T(S)/T(P). Where, T(S) is the best sequential time and T(P) is the run time due to the parallel implementation.

5. EXPERIMENT 2: GPU Computing

In this section, we introduce a GPU implementation of DXT1 (a high quality DXT compression algorithm). In GPU-accelerated computing, GPU is used together with a CPU to accelerate massive/complex applications. Multiple cores in a CPU are optimized for sequential serial processing while a GPU consists of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously. The flow of the application codes to CPU and GPU is illustrated in Figure 5. GPU-accelerated computing offers unprecedented application performance by offloading compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU. Today, GPUs are being used in many power energy-efficient datacenters around the world.

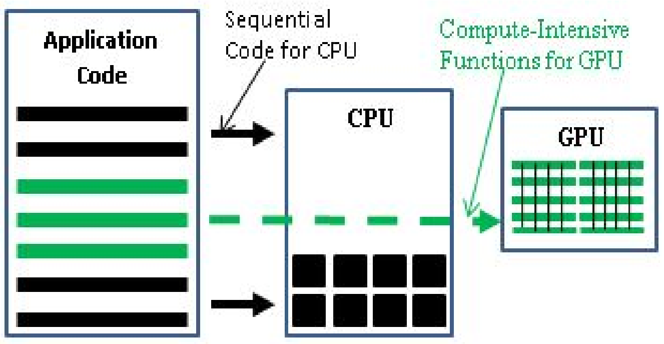

Figure 5: Code-flow in GPU acceleration.

DXT1 is a fixed ratio compression scheme that partitions the image into 4x4 blocks. A DXT1 block layout is shown in Figure 6. One of such block has two 16-bit colors in RGB 5-6-5 format and a 4x4 bitmap with 2 bits per pixel. The block colors are reconstructed by interpolating one or two additional colors depending on whether the value of Color 0 is lower or greater than Color 1. The 512512 pixel standard test image (in color) named Lena is used in this experiment [33].

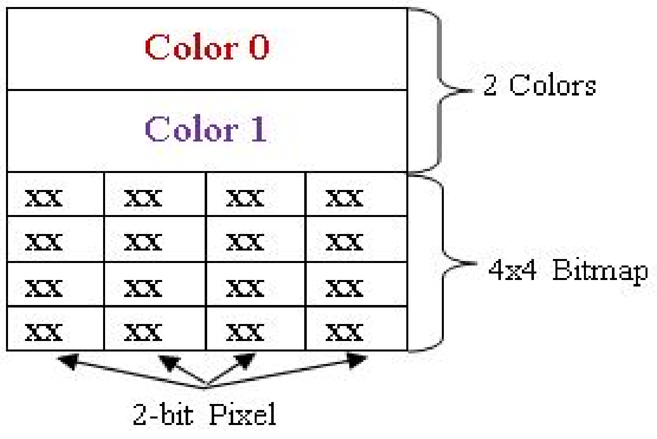

Figure 6: A DXT1 block layout.

A single thread block is used to compress each 4x4 color block. As DXT blocks are independent, no synchronization is needed. We start with grid size equal to the number of blocks in the image is used. The problem is parameterized so that the number of threads per block can be changed to determine the best configuration. For a DXT encoded block, finding the best two points that minimize the error is usually a highly discontinuous optimization problem. If the indices of the block are known, the problem becomes a linear optimization problem. In this case, the indices are not known in advance. So, we need to check them all for the best solution. Using the technique suggested by Simon Brown [35], we consider only the indices that preserve the order of the points along the least squares line. The colors of the block form a cloud of points in 3D space. The direction of the line that best approximates a set of points can be found by computing the largest eigenvector of the covariance matrix. Each element of the covariance matrix is just the sum of the products of different color components. We implement these sums using parallel reductions. Once we have the direction of the best fit line we project the colors onto it and sort them along the line using brute force parallel sort.

6. RESULTS AND DISCUSSION

The primary focus of this work is to explore the challenges and opportunities of data parallelism and GPU computing for future computer games. In this section, we present experimental results that illustrate the impact of multithreaded implementation of game engines on multicore processors. We, first study a single-threaded multi-object game engine, then we study various multithreaded models of the game engine and the GPU implementation of DXT1. We compare the speedup due to various implementations: STM (single threaded), multithreaded asynchronous model (MAM), multithreaded synchronous model (MSM), and MSMDP (MSM with data parallelism).

First, we consider the number of frames generated due to various implementations for a 20-second simulation of the game. As illustrated in Figure 7.

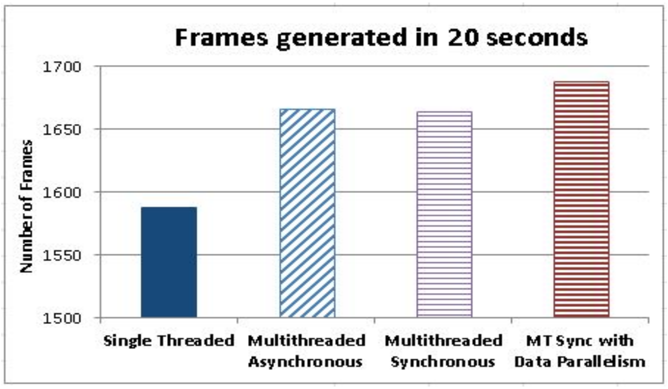

Figure 7: Total number of frames generated in a 20-second simulation of the Tower Defense Game.

The multithreaded synchronous model with data parallelism game console generates more frames per minute than any other consoles. Single-threaded game console generates about 1,587 frames per minute, multithreaded asynchronous and synchronous both generate about 1,665 frames per minute, and multithreaded synchronous with data parallelism generates about 1,688 frames per minute.

Second, we consider the maximum time required to process different frames. As shown in Figure 8, the single-threaded game console takes the maximum amount of time (22 sec) to process a frame when compared with other multithreaded models. MAM shows improvement over STM. Building a proper asynchronous multithreaded engine requires a lot of time and it will be more complex than a synchronized model in most cases. 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 at their own clock cycle. It is also observed that multithreaded synchronous model with data parallelism console takes the minimum amount of time (11 sec) to process a frame.

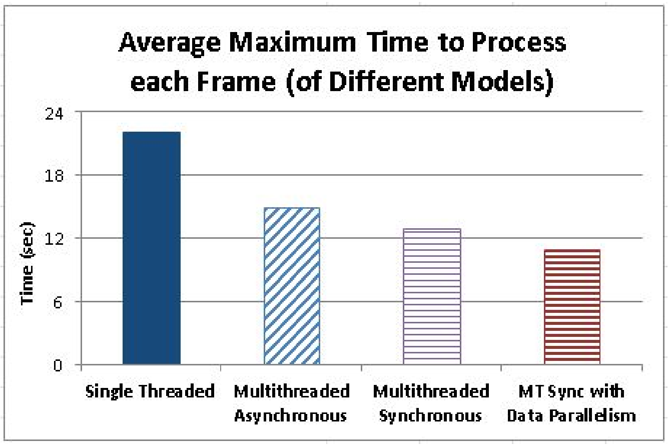

Figure 8: Time to process different frames.

Third, we consider the maximum time required to process different components. As shown in Figure 9, the components are found to have different amount of time for processing. The physics component takes up most of the processing time (15.5 sec); this is because the game uses a lot of physics objects and operations. The Graphics takes the minimum time (1.5 sec) to process.

Different tasks may take different amount of time. In a coarse grained implementation such as the synchronized model, the component that takes the longest time becomes the bottleneck (as other threads have to wait for that thread to be completed). Using multithreading within a component itself seems to improve its performance. The physics component with data parallelism for processing collision detection improves overall performance. The combination of task and data level parallelism achieve the best time in the ideal hardware platform. When a hardware platform has more cores than the number of parallel tasks in the program, employing data level parallelism is the easiest way to scale the number of threads up. This also has some benefits in lower number of cores as some tasks may be completed earlier than other tasks and therefore the idle core can run other processes.

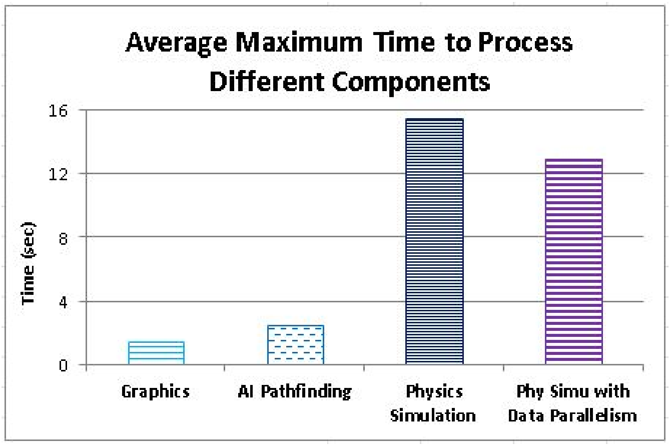

Figure 9: Time to process different components.

Fourth and finally, we consider the speedup due to multithreaded implementations over the single-threaded implementation of the test game engine. From Figure 10, more than $9X$ speedup is achieved due to the multithreaded synchronous model with data parallelism for 512 threads. This result (i.e., speedup) is impressive but not enough for future computer games.

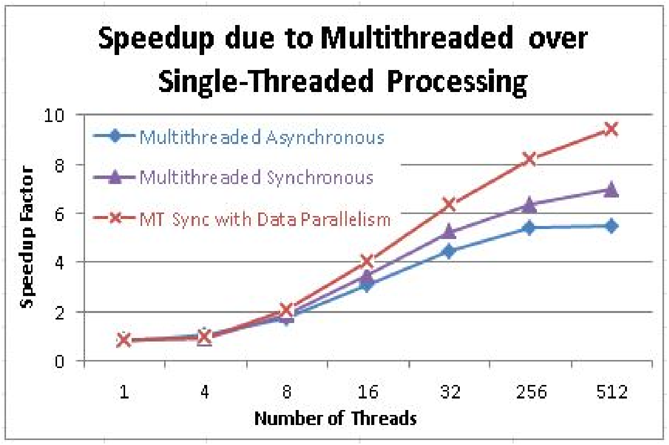

Figure 10: Performance improvement due to multithreaded parallel processing.

Future computer games are expected to be much more complex than what we have today; therefore, speedup needs to be increased to implement and support them. One effective option is to generate thousands of threads and run them concurrently in parallel (as possible) on multicore CPU/manycore GPU systems. Experimental results from a 448-core GPU implementation of DXT1 indicates that a speedup greater than $80X$ is possible (see Figure 11).

7. CONCLUSION

Single-processor multithreaded game engines struggle to improve image and video processing performance due to the lack of hardware support.

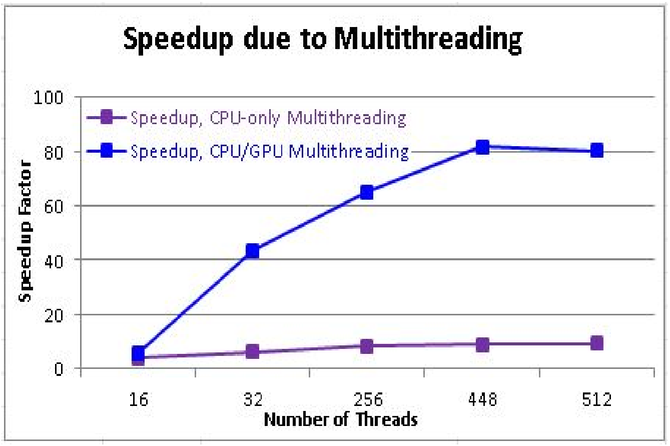

Figure 11: Speedup due to CPU-only multithreading and CPU/GPU multithreading.

Recently introduced multicore CPU/manycore GPU computing platforms have the potential to improve the performance of multithreaded computer games. However, multithreading in multicore game engines introduces various challenges including data parallelism, task parallelism, and thread synchronization. Studies suggest that multicore GPU programming has potential to improve computer game performance. In this work, we explore the challenges due to data parallelism by developing a multicore video game console (called Tower Defense Game). We also explore the impact of GPU computing of multithreaded applications on performance. We implement single-threaded and several multithreaded models and a DXT1 algorithm using GPU/CUDA technology. According to the experimental results, there are indications that the multithreaded model outperform the single threaded model; the multithreaded synchronous models with data parallelism generates more frames and takes less amount of time to process frames.

On an 8-core system, the speedup due to multithreaded synchronous model with data parallelism implementation is about 10 with respect to the single-threaded implementation. However, on an 8-core CPU and 448-core GPU system, the speedup due to multithreaded parallel implementation is more than 80 with respect to the single-threaded implementation. Therefore, GPU technology has potential to improve the performance to power consumption ratio of computer games.

In a platform with multiple processing cores, load balancing becomes important to maximize the use of system resources by avoiding idle threads. Data coupling between components is important in the design of the multithreaded game engine. The mixture of data parallelism and task parallelism is required to take advantage of parallelism in modern systems. In most implementations of game engines, data parallelism seems to fit within the same types of components while task parallelism is between different types of components. The synchronized model with task and data parallelism takes good advantage of concurrency and is highly scalable for any number of cores.

As an extension of this work, we plan to implement the entire test game engine using GPU computing and evaluate the performance and power consumption.

References

[1] S. Berberich, Video games starting to get serious, http://ww2.gazette.net/stories/083107/businew11739_32356.shtml (2007).

[2] A. Asaduzzaman, I. Mahgoub, Cache modeling and optimization for portable devices running MPEG-4 video decoder (2006).

[3] F. Feinbube, P. Troger, A. Polze, Joint forces: From multithreaded programming to GPU computing, IEEE Software Journal 28 (1) (2011) 51.

[4] J. Tulip, J. Bekkema, K. Nesbitt, Multi-threaded game engine design, IE '06 Proceedings of the 3rd Australasian conference on Interactive entertainment (2006) 9–14.

[5] I. Parberry, Introduction to bullet physics, http://larc.unt.edu/ian/classes/fall11/csce4215/notes/bulletphysics.pdf (2011).

[6] J. Brodkin, Shift to multicore processors inevitable, but enterprises face challenges, http://www.networkworld.com/news/2008/022708-multicore-processors.html, network World (2008).

[7] B. Schauer, Multicore processors - a necessity, http://www.csa.com/discoveryguides/multicore/review.pdf, proQuest Discovery Guides (2008).

[8] H. Sutter, The free lunch is over: A fundamental turn toward concurrency in software (2005).

[9] Designing the framework of a parallel game engine (PGE), http://www.intel.com, intel Corporation (2013).

[10] A. Asaduzzaman, C. M. Yip, R. Asmatulu, M. Rahman, CUDA/C based 'green' technology for very fast analysis of nanocomposite properties, SAMPE Tech 2013 Conference, Wichita, Kansas.

[11] A. Asaduzzaman, C. M. Yip, S. Kumar, R. Asmatulu, Fast, effective, and adaptable computer modeling and simulation of lightning strike protection on composite materials, IEEE SouthEastCon Conference 2013, Jacksonville, Florida.

[12] J. Strube, S. Schade, P. Schmidt, P. Buxmann, Simulating indirect network effects in the video game market, 40th Annual Hawaii International Conference on System Sciences (2007) 160.

[13] Multithreading problems, http://www.roguewave.com/portals/0/products/threadspotter/docs/2011.2/manual_html_linux/manual_html/multithreading_problems.html, rogue Wave Software (2011).

[14] A. Asaduzzaman, F. N. Sibai, H. Elsayed, Performance and power comparisons of MPI vs Pthread implementations on multicore systems, 9th International Conference on Innovations in Information Technology (IIT).

[15] J. Lorenzon, E. W. G. Clua, A novel multithreaded rendering system based on a deferred approach, 8th Brazilian Symposium on Games and Digital Entertainment (SBGAMES) (2009) 168–174.

[16] T. Leonard, Dragged kicking and screaming: Source multicore 8 (13).

[17] A. Rhalibi, D. England, S. Costa, Game engineering for a multiprocessor architecture (2005).

[18] L. Davies, Practical examples of multi-threading in games, intel (2006).

[19] Multicore desktop programming with intel threading building blocks, IEEE software 28 (1) (2011) 23–31.

[20] G. Gasior, Valves source engine goes multi-core, http://techreport.com/review/11237/valve-source-engine-goes-multi-core (2006).

[21] K. Gadd, Threading and your game loop, http://www.altdevblogaday.com/2011/07/03/threading-and-your-game-loop/ (2011).

[22] D. Andrade, B. B. Fraguela, J. Brodman, D. Padua, Task-parallel versus data-parallel library-based programming in multicore systems, International Conference on Parallel, Distributed and Network-based Processing (2009) 101–110.

[23] I. Castano, High quality dxt compression using cuda, http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/dxtc/doc/cuda_dxtc.pdf, nvidia Corporation (2007).

[24] A centerfold photo shrunk to width of human hair, http://www.bbc.co.uk/news/technology-19260550), bBC News Online (2012).

[25] R. Hord, Parallel supercomputing in SIMD architectures, CRC Press, 1990.

[26] V. Monkkonen, Multithreaded game engine architectures, http://www.gamasutra.com/view/feature/130247/multithreaded_game_engine_.php?page=3) (2006).

[27] R. Kriemann, Implementation and usage of a thread pool based on POSIX threads, In Max-Planck-Institute for Mathematics in the Sciences (2004) 22–26.

[28] M. Guevara, C. Gregg, K. Hazelwood, K. Skadron, Enabling task parallelism in the CUDA scheduler, In Proceedings of the Workshop on Programming Models for Emerging Architectures (PMEA) (2009) 69–76.

[29] L. Baumstark, L. Wills, Exposing data-level parallelism in sequential image processing algorithms, in: In Proceedings of the Ninth Working Conference on Reverse Engineering, 2002, pp. 1095–1350.

[30] J. Harbour, Multi-Threaded Game Engine Design, Course Technology PTR (1st ed.), 2010.

[31] E. Cronin, A. R. Kurc, B. Filstrup, S.Jamin, An Efficient Synchronization Mechanism for Mirrored Game Architectures, Kluwer Academic Publishers, 2003, extended Version.

[32] B. Dawson, Lockless programming in games, microsoft, GDC (2009).

[33] GPU/CUDA Technology, http://www.nvidia.com, NVIDIA Corporation (2013).