GPU-accelerated 3-D model-based tracking

J Anthony Brown and David W Capson
Department of Electrical and Computer Engineering, McMaster University, Hamilton, Ontario, L8S 4K1 Canada
E-mail: \{brownja5, capson\}@mcmaster.ca

Abstract. Model-based approaches to tracking the pose of a 3-D object in video are effective but computationally demanding. While statistical estimation techniques, such as the particle filter, are often employed to minimize the search space, real-time performance remains unachievable on current generation CPUs. Recent advances in graphics processing units (GPUs) have brought massively parallel computational power to the desktop environment and powerful developer tools, such as NVIDIA Compute Unified Device Architecture (CUDA), have provided programmers with a mechanism to exploit it. NVIDIA GPUs’ single-instruction multiple-thread (SIMT) programming model is well-suited to many computer vision tasks, particularly model-based tracking, which requires several hundred 3-D model poses to be dynamically configured, rendered, and evaluated against each frame in the video sequence. Using 6 degree-of-freedom (DOF) rigid hand tracking as an example application, this work harnesses consumer-grade GPUs to achieve real-time, 3-D model-based, markerless object tracking in monocular video.

1. Introduction and Background

Tracking the pose of a 3-D object in live video, without the use of position encoders, markers, or other measurement tools, has applications across many domains including augmented reality, visual servoing, surveillance, and human-computer interaction (HCI). Approaches to 3-D tracking can be coarsely divided into appearance- and model-based [1]. Appearance-based methods use cues from the video sequence (skin colour, contours, etc.) whereas model-based methods project a 3-D representation of the tracking target onto a plane for comparison with each video frame. In both cases, the goal is to estimate the set of parameters that best describes the tracking target’s pose. Model-based approaches often generate superior results and are algorithmically simpler because the parameters being estimated are the same parameters being manipulated to control the model. However, because such approaches require many 3-D renderings of the tracking target to be produced and evaluated for each frame, they are computationally intensive and consequently less popular than appearance-based techniques.

In this work, it is demonstrated that recent advances in consumer-grade graphics cards and the maturation of tools that exploit their highly parallel computational power has made 3-D model-based approaches to object tracking possible in real-time, at a relatively low cost. Specifically, using hand tracking [2] as an example, NVIDIA’S Compute Unified Device Architecture (CUDA) [3] and Microsoft’s Direct3D [4] are employed to generate several thousand 3-D model renderings, compare each rendering to the current frame of the video stream, and compute an estimate for the pose of the tracking target based on the quality-of-fit of the renderings. The particle filter [5], a sequential monte carlo approach to stochastic state estimation, is applied to ensure that all model renderings are statistically likely to be realistic predictions of the tracking target’s pose.
based on tracking history. The massive parallel processing power of the graphics processing unit’s (GPU) streaming multiprocessors (SMs), aided by the particle filter, facilitates robust 3-D model-based object tracking at 30+ frames per second on consumer-grade hardware.

In recent years, the GPU has emerged as a powerful, affordable, and adaptable parallel computing platform suitable for many high-performance computing (HPC) tasks. Programming models, such as ATI Stream, OpenCL, and NVIDIA CUDA, have allowed programmers to exploit hundreds of SMs for general purpose computing on the GPU (GPGPU) instead of the 3-D graphics rendering for which they were originally designed. Serving as a coprocessor to the central processing unit (CPU), the GPU has been proven capable of accelerating applications 10 to 1000 times over CPU-only implementations [6, 7]. NVIDIA CUDA is a highly scalable platform introduced in 2007 to provide GPU computing on modern NVIDIA GeForce and Quadro products, millions of which are already deployed in PCs and workstations around the world [8]. With a large, collaborative online community, an ever-expanding number of CUDA university courses and textbooks [9], and a massive user-base, NVIDIA GPUs are already well-established in the scientific community. CUDA has been a particularly popular target for object tracking and pose estimation applications, specifically face, body, and hand tracking [10–12].

Pose estimation and tracking using a particle filter is very well-suited to GPU-acceleration, yet there is relatively little literature available describing the integration of the two. Real-time, appearance-based particle filtering approaches to face tracking are described in [13] and [14], with the former using a sparse template matching algorithm and the latter integrating several visual cues, such as color and edges. Both report speedups of at least 10 times over CPU implementations. A hybrid model-based and appearance-based tracker is described by Cabido et al. [15] for 2D articulated object pose estimation. GPU filtering, projection and weighting are also used in [16] for a divide-and-conquer, machine learning approach to 3-D model-based tracking of articulated objects in a surgical setting with both mono and stereo image sequences. Finally, Lenz et al. [12] describe an approach similar to ours, reporting a speedup of up to 10 times over a CPU implementation in face and hand tracking experiments.

The remainder of this paper focuses on the GPU-acceleration aspects of the tracking system, beginning with a review of the NVIDIA GT200 GPU architecture and how it can be exploited with CUDA and Direct3D in Section 2. The 3-D model based tracking system is detailed in Section 3, Section 4 contains the experimental results, and Section 5 concludes the paper with a discussion and summary of future work.

2. GPU Computing
A complete overview of the broad field of GPGPU is beyond the scope of this paper; instead, the focus is limited to material that will clarify and justify this approach to model-based tracking. Specifically, the hardware architecture of the NVIDIA GT200-series GPU and the manner through which it can be exploited using CUDA and Direct3D is explored.

2.1. NVIDIA GT200-Series GPU Architecture
NVIDIA’s highly-scalable GT200-series GPU was released in 2008 and built on the company’s previous G80 architecture. The GT200 is structured as a hierarchy (Fig. 1) [17]. At the lowest level is the streaming processor (SP), a processing element that contains two arithmetic logic units (ALUs) and an instruction pointer, but no hardware for fetching or scheduling instructions. Next is the SM, essentially an independent multiprocessor able to fetch, schedule, and execute instructions. Each SM contains eight SPs, two special function units for advanced operations (sine, square root, etc.), shared memory, a register file (divided equally between SPs), and a read-only cache for constant memory. SMs are organized in groups of three called texture processing clusters (TPCs), each of which contains a read-only L1 texture cache. At the top level of the hierarchy, a GT200 GPU consists of up to ten TPCs, a read-only L2 texture cache, up to 2
GB of GDDR3 device memory (constant, global, local, and texture), and a peripheral control interface express (PCI-E) bus to communicate with the CPU. To give some perspective, high-end consumer-grade GPUs built on this impressive architecture are comprised of approximately 1.4 billion transistors, have a bandwidth up to 159 GB/s, and are theoretically capable of executing over 1 trillion parallel 32-bit floating point instructions per second (1 teraflop) on their 240 SPs, whereas a similarly priced multicore CPU is theoretically capable of approximately 48 gigaflops.

**Figure 1.** The GT200 processing element hierarchy consists of a GT200 GPU, 10 TPCs, 30 SMs, and 240 SPs.

This degree of parallel processing power can only be exploited by highly parallel tasks, such as the high-resolution 3-D graphics rendering for which it is intended. In fact, 1000’s of concurrent threads are recommended to maximize hardware utilization in most applications. Unlike multicore CPU threads, a GPU thread is extremely lightweight and can be created and launched almost instantly. Additionally, general purpose CPUs are designed to maximize single-thread execution through branch prediction and other flow control techniques, whereas the majority of transistors on a GPU are dedicated to achieving maximum arithmetic intensity through massively parallel processing, while sacrificing individual thread performance.

Threads are partitioned by each SM into groups of 32 called warps, a paradigm referred to as **single-instruction multiple-thread** (SIMT), a variation on the single-instruction multiple-data (SIMD) class of parallel computers. Both SIMD and SIMT utilize parallel processing elements to simultaneously execute a common instruction on a vector of data; however, SIMD requires all threads to take the same execution path, whereas SIMT relaxes this constraint by allowing threads within a warp to diverge. **Warp divergence** forces all threads in a warp to execute every instruction branch taken by any thread in the warp, but unnecessary branch instructions are suppressed. As a result, computation time increases for highly-divergent code and is minimized when all warp threads agree on a common path. Additionally, SIMD threads have access to all vector elements whereas each SIMT thread maintains its own registers and must rely on shared memory (Section 2.2) to share data. Depending on resource utilization, up to 32 warps can be active on each SM, totaling 30,720 threads on a GPU. This allows a new warp to be swapped in by the SM scheduler while another is waiting on a high-latency instruction, such as a memory transfer. Because warp packaging and scheduling is handled by low-level hardware, the programmer could choose to completely ignore the concept of warps; however, significant performance gains can be realized if problems, such as warp divergence and other considerations (Section 2.2), are respected.

### 2.2. NVIDIA CUDA

CUDA provides a set of C-language extensions that allow the programmer to create applications that execute serial code on the **host** (CPU) and can asynchronously launch functions called **kernels** to execute on the **device** (GPU), which serves as a coprocessor for data-intensive,
parallelizable code [3, 18]. Scalability is a key tenet of CUDA. A well-written CUDA application is entirely independent of the underlying GPU and will scale to maximally utilize whatever hardware is available.

This is achieved through a thread hierarchy as illustrated in Fig. 2. The kernel code is executed by fine-grained parallel threads, grouped into blocks. Blocks are arranged in a grid, providing second, coarser level of parallelism. A closer look at the GPU thread scheduling paradigm demonstrates the purpose of this hierarchy. One or more blocks are assigned to each SM and consecutive threads within those blocks are partitioned into warps and executed by SPs according to the SM's internal scheduler. However, there is no guarantee of concurrency for blocks within the grid. All blocks may be executed in parallel if enough SMs are available, but generally some will have to wait until other blocks are ready to be swapped out. Consequently, only threads within a block are able to share data and synchronize execution, meaning the programmer has no control over the coordination of warp-level or grid-level scheduling and interaction. The obvious benefit is scalability: as long as an arithmetically intense task can be partitioned into computationally independent blocks that do not need to communicate with other blocks, that task can be efficiently scheduled across all SMs of any capable NVIDIA GPU. Threads within each block can use the on-chip shared memory to efficiently exchange data and can synchronize execution through barrier synchronization, but the only way to synchronize all threads in the grid is to wait for a kernel to finish execution and launch another.

The manner in which a task is partitioned can significantly affect performance. If blocks are too large, there may not be enough to keep all available SMs busy; however, if they are too small, there may be too many and they will end up being serialized. The ratio of the number of active warps to total warp capacity is termed multiprocessor occupancy, which should be kept above 50% (16 active warps) for most applications. As a strong recommendation, block sizes on the GT200 architecture should be multiples of 32 to ensure even warp partitioning and there should be enough blocks active on an SM for the scheduler to hide high-latency instructions through block swapping.

CUDA also exposes various GPU memory spaces, which are described below and summarized in Table 1.

**Global Memory** is the largest, highest-latency memory (400 to 800 clock cycles) available on the GPU and acts as the primary data transfer channel between the host and device. The host and all threads in the grid can read and write global memory; however, it should be accessed sparingly to avoid the access latencies. Global memory is most efficient when consecutive threads access consecutive memory locations. Coalescing memory access in this manner facilitates batch transactions and ultimately increases throughput. A small amount of host memory can be page-locked and mapped into the global memory space to increase bandwidth and support memory transfers that are concurrent with kernel execution.
### Table 1. Summary of the GT200 Memory Model

| Memory    | Access | Scope          | Cached | Latency |
|-----------|--------|----------------|--------|---------|
| Global    | R/W    | Grid+host      | No     | High    |
| Shared    | R/W    | Block          | No     | Low¹    |
| Register  | R/W    | Thread         | No     | Low     |
| Constant  | R      | Grid+host      | Yes    | Low²    |
| Texture   | R      | Grid+host      | Yes    | Low²    |

¹Assuming no bank conflicts
²Cached reads only

**Shared Memory** is an extremely fast, low-latency memory space found on each SM that has two main functions:

- Provide an intermediate storage area to avoid unnecessary global memory usage.
- Facilitate low-latency inter-thread data sharing.

Shared memory can be accessed by all threads within a block, but never by the host. The coalescing requirements of global memory do not apply to shared memory; however, if multiple threads of the same half-warp access the same 32-bit shared memory bank simultaneously, a *bank conflict* will occur and accesses will be serialized.

**Registers** are a fast, low-latency (typically 22 clock cycles), on-chip resource but, unlike shared memory, are only visible to their resident threads. Registers are generally used for intermediate results not shared with other threads.

**Constant Memory** also resides in global memory but data is cached on-chip to greatly reduce latency when multiple threads in a block access the same location. Constant memory is ideal for storing constants, such as filter kernel parameters, and can only be written to by the host and read by the device.

**Texture Memory** is another cached, read-only space that resides in global memory; however, unlike constant memory, it can span the entire global memory space (with some limitations) and has a less-restrictive access pattern. Texture memory requires an area of device memory (linear region or region allocated as a CUDA array) to be bound to a *texture reference* and can be accessed from kernels through *texture fetches*. Texture fetches are optimized for 2-D spatial locality within the memory space, providing a powerful alternative to global memory access patterns in some applications.

Both register and shared memory usage can have a significant effect on multiprocessor occupancy. Because each SM has a finite amount of register and shared memory space that is partitioned across all active warps on the SM, it is possible that there will not be enough to satisfy the memory requirements of all threads in a block. To compensate, the SM will lower the number of active warps to reduce SM occupancy and, consequently, performance. This is one of the many implementation guidelines that must be considered when writing effective code for the GPU. It is trivial to rework CPU code for CUDA but, to receive a tangible speedup, the programmer must be aware of nuances of the architecture and fully utilize the many tools CUDA provides.

### 2.3. Microsoft Direct3D

Direct3D, an API available as part of Microsoft’s DirectX SDK used primarily in video game development, facilitates optimized 3-D graphics rendering on the GPU. For a detailed look, the reader is directed to the programming guide [19] or one of the many textbooks [4, 20] or
online tutorials on the topic. Here, a small subset of its features and terminology relevant to model-based tracking are briefly introduced.

There are a number of Direct3D resources available, such as *vertex buffers*, *index buffers*, *textures* and *surfaces*, all of which can be registered in CUDA’s memory space and subsequently mapped there to be accessed by kernels. Here, the focus is on surfaces, a linear area of GPU memory that can be customized with a desired size (resolution) and pixel format (32-bit ARGB in this work). Surfaces have a number of useful features, such as the ability to be locked by the host to access and modify individual pixel data and efficient resizing and stretching options. They can also serve as render targets and are easily swapped into the video card’s *back buffer*, which displays them to the screen. Textures are similar to surfaces, but can be applied to geometry during rendering. Rendering an image to a texture then applying it to a simple *quad* (square) is an efficient way of creating a modular 2-D projection of a model on a larger surface.

The Direct3D rendering pipeline is highly streamlined to produce realistic 2-D projections of a model with simulated lighting and shading. The projective properties of the virtual camera can be configured in a *projection transform matrix* and the position and direction of the camera are configured in a *view transform matrix*. A variety of virtual lights can be placed around the object to simulate real world lighting and shading; however, this increases computational complexity. Once the camera, lights, render target, transformation matrices, and texture properties are configured, a 3-D model rendering can be produced extremely quickly on the GPU by projecting mesh geometry onto a 2-D plane. Instead of displaying the renderings on the screen as they are intended, they are preserved in GPU memory for further processing using CUDA’s Direct3D interoperability.

3. GPU-Accelerated 3-D Model-Based Tracking
The operation of the GPU-accelerated tracking system is broadly divided into the host (CPU) and device (GPU) subsystems as shown in Fig. 3. The remainder of this section examines these two subsystems in detail.

![Figure 3. GPU-accelerated model-based tracking system block diagram](image)

3.1. Host Subsystem: Particle Filtering and State Estimation
Within the particle filter framework, the position and orientation of the tracking target is represented by a state vector:

\[ \mathbf{x}_t = (O_x, O_y, O_z, r_x, r_y, r_z) \]  

(1)
where \((O_x, O_y, O_z)\) are the coordinates of the tracking target’s origin and \((r_x, r_y, r_z)\) are the rotations around the x-, y-, and z-axis, respectively. The goal of tracking is to estimate the parameters of this vector at discrete time points (e.g., once per frame). To obtain this estimate, the system maintains a set of \(N\) possible states, known as particles, each of which is a statistically likely guess at the true tracking target location. Each particle has an associated weight that provides a measure of how well that particle represents the true state based on the most recent frame of video. Estimating the state that represents the tracking target’s true pose can be obtained based on the:

- state of highest-weighted particle;
- average of \(M\) highest-weighted particle states; or,
- weighted average of \(M\) highest-weighted particle states

where \(M \leq N\).

The value of the particle filter (also known as bootstrap filtering, the condensation algorithm [21], and survival of the fittest) is its ability to intelligently select statistically likely particles for simulation and evaluation based on particle weights and system dynamics. This work uses the sampling importance resampling (SIR) particle filter, which is described in detail by Arulampalam [5] and will only be outlined here. The motion tracking task is framed as a Bayesian state estimation problem wherein the weight of a particle describes the probability that its state represents the true pose of the tracking target. The weights are normalized and used to construct a cumulative density function (CDF). \(N\) particles are then randomly selected from the CDF and the parameters of each are updated based on a system dynamics model, which monitors the velocity and acceleration of every DOF in the state estimates (particle drift), as well as some Gaussian noise (particle diffusion). By selecting particles from a CDF in this manner, the highly-weighted particles can be selected many times, whereas the lower-weighted particles may not get selected at all. The result is a set of particles that approximates the current posterior density function of the state sequence. Of course, reliable particle selection is contingent on meaningful particle weights generated by the GPU subsystem, which is the computationally demanding task described in the proceeding section.

3.2. Device Subsystem: Model Rendering and Weighting

The goal of the device subsystem is to simulate and evaluate the particles selected by the host subsystem and generate a weight for each. In model-based tracking, simulation involves configuring a 3-D model of the tracking target according to the state of each particle, then projecting each configuration onto a 2-D plane, referred to as a particle image. Evaluation involves extracting features from the current video frame and comparing them to corresponding features extracted from each particle image.

To initialize the system, a 3-D model of a tracking target is imported (tracking frame rate is proportional to model polygon count) and Direct3D resources are created based on various system parameters, such as the number of particles and particle image resolution. Additionally, several values are precalculated and stored in CUDA constant memory to minimize unnecessary high-latency instructions within CUDA kernels (e.g., division, modulus). Finally, the image background data is identified based on a short training sequence and transferred into CUDA global memory for use in the background removal stage.

To generate the particle images, the model is oriented with respect to the virtual camera according to each of the \(N\) particles’ parameters and projected onto a texture. The \(N\) textures are applied to quads and tiled onto a single, large Direct3D surface, referred to as the particle grid (Fig. 4). The particle grid must have the same x- and y-dimension, meaning the number of particles \(N\) must be a perfect square. The particle grid is then mapped to a CUDA array in the device’s texture memory along with the current frame from the video stream.
CUDA’s Direct3D interoperability allows individual pixels within the Direct3D surfaces to be accessed through texture fetches and read into the device’s shared memory as floating point data. Because each SM’s shared memory is on-chip, it is extremely low latency and facilitates the fast, scalable, synchronizable, parallel image processing required to efficiently generate a meaningful weight for all particles in the grid. The stages involved in weight generation are:

- Background subtraction
- Edge and silhouette detection
- Weight calculation

The remainder of this section describes how the weight generation task and is efficiently partitioned across SM’s to minimize processing time and maximize GPU utilization.

A background subtraction CUDA kernel is executed on each frame in the video sequence to segment the tracking target from the background. Pixel data for each frame is compared to the background data acquired during initialization. If a pixel falls within a configurable threshold of the corresponding pixel in the background data, it is assumed that pixel represents image background and it is suppressed; otherwise, it is assumed the pixel is part of the tracking target and its value is preserved. While this approach requires a static camera position be maintained, this is not a significant restriction for the majority of target applications.

Edge and silhouette detection (Fig. 5) are performed by CUDA kernels on both the current frame and the particle grid, which reside in texture memory. The kernel processing the frame stores results as floating point data in global memory, whereas the kernel processing the particle grid maintains its results in shared memory for the weight calculation stage. Because the process is essentially identical for both, the algorithm is described in terms of the particle grid only.

A simple thresholding approach to silhouette detection is employed: white pixels are considered background and are set to black, whereas non-white pixels are identified as part of the tracking target and are set to gray. Edge detection is achieved using the Sobel edge detector, which involves the convolution of the pixel data with two 3x3 filters - a highly parallelizable task that is extremely well-suited to the SIMT parallel programming paradigm.

An efficient CUDA implementation partitions the algorithm in a manner that maximizes GPU utilization. In this work, a 32x8 thread block is used, regardless of particle image resolution or number of particles. The number of blocks in the CUDA grid, however, varies with these parameters. Texture fetches in consecutive threads are used to access consecutive pixels and store them as floating point data in shared memory. This adheres to the optimal access patterns for both shared and texture memory described in Section 2.2. To ensure blocks evenly tile the particle grid, the width of particle images are multiples of 32 and the height preserves the aspect ratio of the video source. The number of texture fetches is minimized by loading as much pixel data shared memory as possible, without excessively impacting multiprocessor occupancy. The CUDA occupancy calculator [22] was used to determine that each thread can
safely load three 32-bit pixels into shared memory while maintaining an SM occupancy of 50%. Fig. 6 demonstrates an example pixel access pattern. Once all pixel data has been loaded to shared memory, barrier synchronization is used to ensure all other threads within a block have fetched their data. Finally, each thread performs edge and silhouette detection on the pixels that it fetched, accessing neighbouring pixels from shared memory as needed, and stores the end results in shared memory for the subsequent weight calculation stage.

The goal of weight calculation is to compare the temporary frame, which resides in global memory, to each particle image in the particle grid, which is partitioned across shared memory, and generate a weight for each particle with pixel-level accuracy. A divide-and-conquer approach is used to first generate a weight for each pixel in the particle grid, then sum the pixel weights to generate a weight for each thread block, and finally sum the block weights of all blocks that tile a given particle image to generate a weight for that particle (Fig. 7).

Figure 6. Partitioning of a 2x2 particle grid with particle image resolution of 64x48, block size of 32x8, each thread fetching 3 pixels along the y-direction. Pixel indices shown outside boxes, block IDs and thread IDs shown inside the left and right boxes, respectively.

Figure 7. Particle Weight calculation

Figure 8. Weight summation parallel reduction algorithm
Since shared memory does not persist across kernel launches, weight calculation is done by
the same kernel that performs edge and silhouette detection on the particle grid to circumvent
unnecessary global memory accesses. Each thread loads three registers with the frame pixels
corresponding to the previously fetched particle grid pixels and pixel weights \( w_{\text{pixel}} \) are calculated
according to
\[
  w_{\text{pixel}} = \epsilon_{\text{max}} - |p_{\text{frame}} - p_{\text{particle}}|
\]
where \( p_{\text{frame}} \) is the floating point value of a frame pixel, \( p_{\text{particle}} \) is a particle pixel, and \( \epsilon_{\text{max}} \)
is the maximum possible error (i.e., residual) the pixels can achieve. Pixel weights are written
back to shared memory.

The most computationally complex aspect of weight calculation is the summation of pixel
weights. This step is performed by a parallel reduction algorithm similar to \([23]\). Each row of a
thread block reduces its corresponding shared memory row to a single value. This done in five
iterations, the first three of which are partially shown in Fig. 8. In the first iteration, 16 threads
of a row each sum two adjacent shared memory cells and store the results in shared memory.
The next iteration uses eight threads to sum the results of the previous iteration, and so on.
The algorithm is applied to three shared memory rows by each row of the thread block, yielding
a single column of 24 weights for the block, without any bank conflicts. A single thread then
serially adds the values in this column to produce a single weight for the entire block and stores
it in global memory.

Finally, a new kernel is launched (serving as a global synchronization barrier) to sum the
weights of all blocks that tile each particle and writing the resulting particle weights to page-
locked host memory. A major drawback of GPU computing is the significant latency associated
with data transfers between the GPU and CPU. The described approach circumvents excessive
data transfers by performing model rendering, projection, feature extraction, and particle image
evaluation entirely on the GPU, requiring only one floating point value per particle to be
transfered across the PCI-E bus.

4. Results and Discussion
The system was tested on an Intel Core2 Duo E8400 3.0GHZ PC with 4 GB DDR3 RAM, running
Windows 7, CUDA 3.1, and DirectX 9 and equipped with an Asus ENGTX295 NVIDIA GeForce
GTX295 graphics card (2 GPUs, each with 240 streaming processors and 896MB GDDR3
memory). A rigid (6 DOF) 3-D model of a hand comprised of 389 polygons was created using
Blender \([24]\) and used as a test object. An Allied Vision Technology GC660C 119FPS 659x493
GigE CCD camera was used to record a five second (600 frame) video sequence at a resolution
of 320x240, during which the hand moves extremely quickly and through all degrees of freedom,
with several occurrences of simultaneous translation and rotation. The hand in the sequence
was tracked at 5 possible particle image resolutions with \( N \) ranging from 50 to 4050, and a
weighted average of 25 particles was used for state estimation. A demonstration of tracking is
shown in Fig. 9 and accuracy and frame rate results are shown in Fig. 10. Because ground
truth is not defined for this video sequence, accuracy is measured by comparing the silhouettes
of the tracking target and the 3-D model configured in the estimated pose; specifically, accuracy
is defined as the ratio of correctly classified pixels (tracking target vs background pixels) to
the total number of pixels in the bounding box that contains both silhouettes. Note that some
resolutions were not possible at certain values of \( N \) due to limitations of Direct3D.

As expected, the frame rate of the system decreases as particle image resolution or the
number of particles increase. Conversely, accuracy increases with an increase in either of these
parameters; however, diminishing returns are realized once a resolution of 96x72 and a particle
count of approximately 1000 are reached. These settings can be viewed as the ideal operating
parameters for tracking, simultaneously delivering both speed and accuracy (31 fps with 95.2%
accuracy). For applications that demand a higher frame rate but less accuracy, reducing the
Figure 9. Tracking demonstration for three example frames: from top to bottom, the three rows show the original video frame, the frame with a wire-frame overlay, and a rendering of the 3-D model in the estimated pose of the tracking target, respectively.

Figure 10. Frame rate and accuracy results at 5 particle image resolutions and the number of particles varying between 50 and 4050.

The particle count to 500 nearly doubles the framerate while decreasing accuracy by just 0.2%. A qualitative analysis of the system demonstrates that tracking is both accurate and robust, with a nearly one-to-one mapping between the pose of the tracking target and the estimate. Rotation proves more difficult to estimate than translation as there are often multiple solutions when dealing with monocular video. Additionally, abrupt changes in direction at high velocities can cause the tracking target to momentarily be lost; however, recovery normally takes less than 5 frames, as the system dynamics model used by the particle filter compensates. Additional views would certainly increase accuracy, but at a significant increase in computational complexity.

The results of GPU acceleration are also of interest. Table 2 compares the amount of time
taken for each stage of the tracking system in both GPU and CPU implementations for the ideal operating parameters described above. Timing results were captured by GPU timers, which are accurate within half a microsecond. The feature detection and weight calculation stage is the target of GPU-acceleration and demonstrated a speedup of almost 18 times over an equivalent CPU implementation. It is important to note that both versions use the GPU for model rendering as it would be illogical to attempt to use the CPU for this task in any scenario. This explains why the CPU implementation slightly outperformed the GPU in this stage, as the GPU resources are dedicated exclusively to model rendering.

| Stage                     | GPU Time (ms) | CPU Time (ms) | Speedup |
|---------------------------|---------------|---------------|---------|
| Render Particle Images    | 11.90         | 9.67          | 0.81x   |
| Acquire Frame             | 5.68          | 2.50          | 0.44x   |
| Map Resources             | 2.65          | 3.64          | 1.37x   |
| **Edge Detection & Weight Calc.** | **8.78**      | **157.23**    | **17.90x** |
| Particle Filter           | 1.38          | 1.04          | 0.75x   |
| Other                     | 1.93          | 1.84          | 0.95x   |
| **Total**                 | **32.34**     | **175.92**    | **5.44x** |

The speedup is proportional to both the particle image resolution and number of particles and ranges between 2 and 20 times depending on these parameters. One major limitation of the GPU-acceleration lies in Direct3D, which is not able to independently control multiple GPUs. As a result, while the memory spaces of both GPUs are utilized to increase the maximum size of the particle grid, only one GPU is used to render particle images and execute CUDA kernels. Additionally, some aspects of feature extraction and weight calculation are not ideal candidates for a GPU implementation; for example, padding the edges of particle images for convolution with the Sobel kernels requires significant conditional logic that leads to warp divergence and the parallel reduction of all pixel weights to a single value is inherently difficult to parallelize. Nonetheless, during each second of tracking, the GPU generates, processes, and reduces nearly 1GB of pixel data to less than 40KB of particle weight data. This demonstrates the GPU’s utility as a powerful, massively parallel processor that enables previously-unobtainable, real-time performance of the 3-D model-based approach to tracking.

5. Conclusion
This work has demonstrated that robust and accurate real-time 3-D model-based tracking is feasible using modern consumer-grade GPUs. Despite some limitations in maximally utilizing multiple GPUs and the challenges associated with parallelizing some inherently serial aspects of the particle weighting algorithm, speedups up to 20x over an equivalent CPU implementation were realized. Future work includes adapting the system to support articulated object tracking (e.g., a hand with a flexible finger and thumb for virtual ‘gripping’) and exploring more specific applications, such as augmented reality. With the recent release of NVIDIA’s Fermi architecture [25], designed primarily with the GPU-computing community in mind, the utility of GPU-acceleration in high-performance computing tasks, such as model-based tracking and countless other applications, will certainly continue to grow.

Acknowledgments
The authors would like to thank M Kinsner and T Kirubarajan for their guidance, as well as NSERC Canada and McMaster University for funding this research.
References
[1] Lepetit V and Fua P 2005 Foundations and Trends in Computer Graphics and Vision pp 1–89
[2] Stenger B, Thayananthan A, Torr P H S and Cipolla R 2006 IEEE Trans. Pattern Analysis and Machine Intelligence 28 1372–1384
[3] NVIDIA Corporation 2010 CUDA Programming Guide Santa Clara, CA 3rd ed
[4] Sherrod A 2006 Ultimate Game Programming With DirectX (Boston, MA: Charles River Media)
[5] Arulampalam M S, Maskell S, Gordon N and Clapp T
[6] Tan G, Guo Z, Chen M and Meng D 2009 ICS ’09: Proceedings of the 23rd international conference on Supercomputing pp 380–389
[7] Pock T, Unger M, Cremers D and Bischof H 2008 Computer Vision and Pattern Recognition Workshops, 2008. CVPRW ’08. IEEE Computer Society Conference on pp 1 –8
[8] Halfhill T R 2008 Microprocessor Report
[9] Kirk D B and mei W Hwu W 2010 Programming Massively Parallel Processors: A Hands-on Approach (Burlington, MA: Morgan Kaufmann Publishers)
[10] Breitenstein M, Kuettel D, Weise T, Van Gool L and Pfister H 2008 CVPR’08: IEEE Conference on Computer Vision and Pattern Recognition pp 1–8
[11] Lehment N H, Arsic D, Kaiser M and Rigoll G 2010 Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on pp 87 –92
[12] Lenz C, Panin G and Knoll A 2008 VMV’08: International Workshop on Vision, Modeling and Visualization
[13] Lozano O and Otsuka K 2008 ICASSP ’08: IEEE International Conference on Acoustics, Speech and Signal Processing pp 713 – 716
[14] Liu K Y, Tang L, Li S Q, Wang L and Liu W 2009 ICME’09: Proceedings of the 2009 IEEE international conference on Multimedia and Expo pp 1813–1816
[15] Cabido R, Concha D, Pantrigo J J and Montemayor A S 2009 ISPAN ’09: Proceedings of the 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks (Washington, DC, USA: IEEE Computer Society) pp 757–762
[16] Pezzementi Z, Voros S and Hager G D 2009 ICRA’09: Proceedings of the 2009 IEEE international conference on Robotics and Automation (Piscataway, NJ, USA: IEEE Press) pp 1225–1232 ISBN 978-1-4244-2788-8
[17] Kanter D 2008 NVIDIA’s GT200: Inside a parallel processor URL http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242
[18] NVIDIA Corporation 2010 CUDA Best Practices Guide Santa Clara, CA 3rd ed
[19] Microsoft Corporation 2003 Microsoft DirectX 9.0c SDK Documentation.
[20] Luna F 2003 Introduction to 3D Game Programming with DirectX 9.0 (Plano, TX: Wordware Publishing)
[21] Isard M and Blake A 1998 Inter. Journal of Comp. Vis. 29 5–28
[22] NVIDIA Corporation 2008 CUDA GPU Occupancy Calculator Santa Clara, CA
[23] Harris M 2007 Optimizing parallel reduction in cuda Tech. rep. NVIDIA Corporation
[24] Roosendaal T and Selleri S 2009 The Official Blender 2.3 Guide (Austria: Springer-Verlag/Wien)
[25] NVIDIA Corporation 2009 NVIDIA’s Next Generation CUDA Compute Architecture: Fermi Tech. rep. NVIDIA Corporation