GPU Accelerated Likelihoods for Stereo-Based Articulated Tracking

Rune Møllegaard Friborg, Søren Hauberg, and Kenny Erleben

{runef, hauberg, kenny}@diku.dk,
The eScience Centre, Dept. of Computer Science, University of Copenhagen

Abstract. For many years articulated tracking has been an active research topic in the computer vision community. While working solutions have been suggested, computational time is still problematic. We present a GPU implementation of a ray-casting based likelihood model that is orders of magnitude faster than a traditional CPU implementation. We explain the non-intuitive steps required to attain an optimized GPU implementation, where the dominant part is to hide the memory latency effectively. Benchmarks show that computations which previously required several minutes, are now performed in few seconds.

Keywords CUDA · GPU Computing · Articulated Tracking · Particle Filtering

Fig. 1. The type of articulated tracking for which we achieve a speed up factor of up to 600 when using a GPU optimization. The images show stereo points with a super imposed illustration of the skin model.

1 The Computational Problem of Articulated Tracking

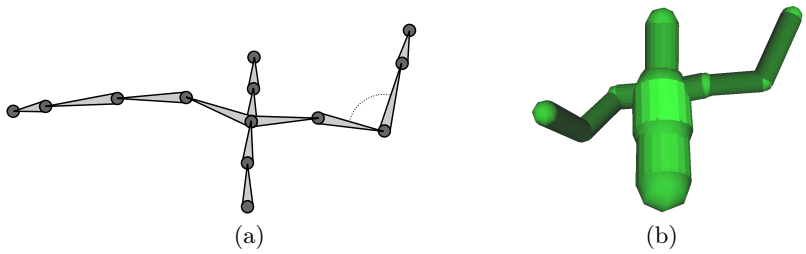
Three dimensional articulated human motion tracking is the process of estimating the configuration of body parts over time from sensor input [1]. One approach to this estimation is to use motion capture equipment where e.g. electromagnetic markers are attached to the body and then tracked in three dimensions. While this approach gives accurate results, it is intrusive and cannot be used outside laboratory settings. Alternatively, computer vision systems can be used for non-intrusive analysis such as the one shown in Figure 1. One standard approach

is to use a particle filter [2] for finding a sequence of poses that match the observed data well. From a practical point of view this means making many random guesses of the current pose and comparing these to the observed data. In terms of performance, the critical part is comparing each guess to the data. In this paper, we present a GPU-based solution to this problem and show a substantial increase in performance compared to a CPU-based implementation. Such performance increases are essential in allowing us to build proper generative likelihood models, that otherwise would be impractical.

Before dwelling into the details of this work, we briefly describe in Section 2 the general particle filter based framework for articulated tracking that forms the foundation for this work. Next we consider related work in Section 3 and in Section 4 we describe the likelihood model for our work. We focus on using the GPU in Section 5 and results can be found in Section 6 before we conclude in Section 7.

2 Particle Filtering for Articulated Tracking

The objective of articulated human tracking is to estimate the position and orientation of each limb in the human body. This, as such, requires a representation of the human body. The most common choice [1] is the *kinematic skeleton* which is a collection of rigid bones organised in a tree structure (see Fig. 2(a)). Each bone can be rotated at the point of connection between the bone and its parent. We will refer to such a connection point as a *joint*.



(a) (b)

Fig. 2. (a) A rendering of the kinematic skeleton. Each bone position is computed by a rotation and a translation relative to its parent. The joints are drawn as circles. (b) A rendering of the skin model.

We model the bones as having known constant length (i.e. rigid), so the direction of each bone constitute the only degrees of freedom in the kinematic skeleton. The direction in each joint can be parametrised with a vector of angles, noticing that different joints may have different number of degrees of freedom. We may collect all joint angle vectors into one large vector $\boldsymbol{\theta}_t$ representing all joint angles in the model. The objective of the tracking system then becomes to estimate this vector at each time step.

At the heart of our articulated tracker is the well-known particle filter [2], which we will briefly describe here. The particle filter is, in general, concerned with estimating an unobserved state of a system from observations. In terms of articulated tracking it is concerned with estimating the pose $\boldsymbol{\theta}_t$ at each frame in a video sequence. In terms of statistics, we seek $p(\boldsymbol{\theta}_t|\mathcal{X}_{1:t})$, where the subscript denotes time and $\mathcal{X}_{1:t} = \{\mathcal{X}_1, \ldots, \mathcal{X}_t\}$ denotes all observations seen at time t. This distribution is crudely represented as a set of samples that are propagated through time by sampling from $p(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1})$. Each sample $\boldsymbol{\theta}_t^{(j)}$ is assigned a weight according to its likelihood $p(\mathcal{X}_t|\boldsymbol{\theta}_t^{(j)})$. Thus, at each time step t we compute

for $j = 1$ to J **do**
  Sample $\boldsymbol{\theta}_t^{(j)}$ from $p(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}^{(j)})$;
  $w_j \leftarrow p(\mathcal{X}_t|\boldsymbol{\theta}_t^{(j)})$;
end for

Usually it is computationally cheap to sample from $p(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}^{(j)})$, whereas it is expensive to evaluate the likelihood $p(\mathcal{X}_t|\boldsymbol{\theta}_t^{(j)})$. It is worth noting that the loop can be executed in parallel as each sample is treated completely independent.

Once we have drawn new samples and assigned them weights, we can estimate the current pose as the mean value of $p(\boldsymbol{\theta}_t|\mathcal{X}_{1:t})$. This can be approximated as

$$\bar{\boldsymbol{\theta}}_t \approx \sum_{j=1}^{J} \frac{w_j}{\sum_{l=1}^{J} w_l} \boldsymbol{\theta}_t^{(j)} \quad . \tag{1}$$

3 Related Work on Computational Tracking

Most work in the articulated tracking literature falls in two categories. Either the focus is on improving the image likelihoods or on improving the predictions. Due to space constraints, we forgo a review of various predictive models as this paper is focused on computational efficient likelihoods. For an overview of predictive models, see the review paper by Poppe [1].

Most publications on likelihood models for articulated tracking are concerned with finding descriptive image features. Sminchisescu and Triggs [3] showed successful tracking using a combination of edge strength and horizontal flow in a monocular setup. This approach is, however, bound to have difficulties due to only having one viewpoint. One solution is to use multiple calibrated cameras as, amongst others, was done by Deutscher et. al. [4] who used a combination of edge strength and background subtraction. Due to the difficulties of calibration, such approaches are, however, hard to use in non-laboratory settings. A possible compromise is to use a pre-calibrated stereo camera as was done by Hauberg et. al. [5]. Their solution did, however, not cope with limbs occluding each other.

While much work has gone into developing functional likelihood models, not much has been published on efficient implementations on GPU hardware. Exceptions include the work of Bandouch et. al. [6] that use a simple colour based appearance model in a multiple camera setup. By representing pixel colours as

bitmasks they are able to make likelihood evaluations using only bitwise operations that can be efficiently implemented on the GPU. Cabido et. al. [7] use a combination of background subtraction along with binary template matching for a planar low-dimensional articulated model. They rephrase the entire optimisation as an application of textures on the GPU and as such get very high frame rates.

4 Our Likelihood Model

In this section we define the likelihood model $p(\mathcal{X}_t|\boldsymbol{\theta}_t)$ used in this paper. We use an off-the-shelf consumer stereo camera[1], which provides us with a set of points in 3D at each time step. We, thus, have $\mathcal{X}_t = \{\boldsymbol{x}_{1,t}, \ldots, \boldsymbol{x}_{I,t}\}$, where I denotes the number of points and each $\boldsymbol{x}_{i,t} \in \mathbb{R}^3$.

We will assume that each point generated by the stereo camera is independent and is normally distributed around the skin of the pose. Thus, we have

$$p(\mathcal{X}_t|\boldsymbol{\theta}_t^{(j)}) \propto \prod_{i=1}^{I} \exp\left(-\frac{d_i^2(\boldsymbol{\theta}_t^{(j)})}{2\sigma^2}\right) \ , \tag{2}$$

where $d_i^2(\boldsymbol{\theta}_t^{(j)})$ denotes the square Euclidean distance between the i^{th} stereo point and the skin of the pose parametrised by $\boldsymbol{\theta}_t^{(j)}$. For numerical stability [2] we implement the particle filter on a logarithmic scale and as such only need to compute

$$\log p(\mathcal{X}_t|\boldsymbol{\theta}_t^{(j)}) = -\frac{1}{2\sigma^2}\sum_{i=1}^{I} d_i^2(\boldsymbol{\theta}_t^{(j)}) + \text{constant} \ , \tag{3}$$

where the constant term can be ignored. For this definition to be complete, we need a definition of the skin model and a suitable metric.

For the skin of the j^{th} sample we will use a collection of capsules $\mathcal{C}^j = \{c_1^j, \ldots, c_K^j\}$. Specifically, we assign a capsule to each bone in the kinematic skeleton, such that the capsule is aligned with the bone. The radius of the capsule depends on the bone. We then define the skin of the skeleton as the union of these capsules. This gives us skins such as the one in Fig. 2(b). This model is very similar to the common model (see e.g. [8, 9]) where a cylinder is assigned to each bone. Here, we use capsules for mathematical convenience.

To compute the distance between a point and the skin, we compute the distance from the point to each capsule and pick the smallest, i.e.

$$d_i^2(\boldsymbol{\theta}_t^{(j)}) = \min_k d^2(\boldsymbol{x}_{i,t}, c_k^j) \ , \tag{4}$$

where $d^2(\boldsymbol{x}_{i,t}, c_k^j)$ denotes the square distance from the i^{th} stereo point to the k^{th} capsule of the j^{th} sample. We will define this distance in terms of ray casting in

[1] `http://www.ptgrey.com/products/bumblebee2/`

the following. To avoid notational clutter, we will omit the time subscript from our notation in the rest of the paper.

Let the capsule c_k^j be defined by the two bone end points $\boldsymbol{a} \in \mathbb{R}^3$ and $\boldsymbol{b} \in \mathbb{R}^3$ and the radius $r \in \mathbb{R}_+$. Consider the stereo point $\boldsymbol{x}_i$. This is a point seen by the camera. Thus, $\boldsymbol{x}_i$ must lie on a ray starting at the camera origin $\boldsymbol{p} \in \mathbb{R}^3$ and casting in the direction of $\boldsymbol{v} = \frac{\boldsymbol{x}_i - \boldsymbol{p}}{\|\boldsymbol{x}_i - \boldsymbol{p}\|}$. We can therefore think of $\boldsymbol{x}_i$ as a function of the ray length parameter Δ. That is, we have the ray definition

$$\boldsymbol{x}_i(\Delta) = \boldsymbol{p} + \boldsymbol{v}\Delta \qquad \forall \Delta \geq 0 \ . \tag{5}$$

From this definition we may define a measure indicating how well a given stereo point $\boldsymbol{x}_i$ fits with a given capsule. Let Δ be the ray length of the stereo point and let $\Delta_{\min}$ be the shortest ray length corresponding to an intersection point between the ray and the capsules then intuitively a distance measure may be taken as $|\Delta - \Delta_{\min}|$. This corresponds to rendering a depth map of the capsules, and computing the absolute difference between this and the depth map from the stereo camera.

Since stereo data contains outliers, both from other objects appearing in the scene and from false matches, we need a robust metric. Here we simply truncate the distance if it exceeds a given threshold

$$\mathbf{d}(\boldsymbol{x}_i, c_k^j) = \begin{cases} |\Delta_{\min} - \Delta| & \text{if } \Delta_{\min} \text{ exists and } |\Delta_{\min} - \Delta| \leq \tau. \\ \tau & \text{otherwise.} \end{cases} \tag{6}$$

For this metric to be computable, we need to be able to determine if a given ray intersects the capsules and if so compute the distance $\Delta_{\min}$. The details of ray capsule intersection can be found in Appendix A. It is worth noting that the basic model works for all skin models, though ray casting details will have to be adapted.

5 Optimizing for the GPU

The algorithm presented in this paper achieves a major speedup when implemented on the GPU. However, it requires careful planning in designing for the massive parallelism in the GPU architecture. The first problem to be addressed is how to block data and computations most efficiently with respect to performance. The task is to minimize data communication and maximize the amount of computations done by one block of threads. Our targeted GPU architectures are the CUDA enabled Nvidia GPUs with compute capability from 1.1 to 1.3.

For our current applications we typically use in the order of $I \approx 50000$ stereo points, $K \approx 40$ capsules and $J \approx 2000$ samples. One simple approach would be to create a 3D float array of dimension $J \times K \times I$ where entry (j, k, i) would hold the value of $\mathbf{d}(\boldsymbol{x}_i, c_k^j)$. This would result in a naive data parallel computation where each thread would compute a single distance measurement. However, such an array would require $2000 \times 40 \times 50000 \times 4$ bytes ≈ 16 Gigabytes of memory.

This clearly exceeds the maximum available device memory, so some tiling must be applied to our problem.

Thus, we create a grid of thread blocks in such a way that each thread block corresponds to one sample and one tile of stereo points and we launch a measure kernel on this grid. During execution the measure kernel will loop over samples in consecutive launches to avoid kernel time-outs. Additionally, support for multiple GPU devices is performed by dividing the samples into one chunk for each GPU. If multiple GPUs are available the same number of CPU worker threads is created and then given a GPU to control. The overhead of launching CPU threads is small and the effect will only be visible for small problem sizes which are not the target for this paper. This orchestration results in the grid setup illustrated in Figure 3. Using this approach we will have an intermediate 2D result array $\mathcal{A}$ consisting of $J \times$ POINTS_TILES computed measurements, where POINT_TILES is set to $\frac{I}{POINTS\_PER\_BLOCK}$. The number of threads in each block is identical to POINTS_PER_BLOCK, thus this value is tuned to achieve the best occupancy for a given GPU.

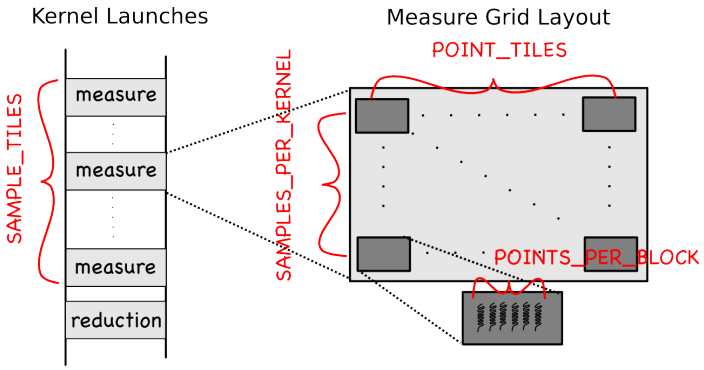


Fig. 3. Illustration of the grid layout and kernel launches for a single GPU. A sequence of measure kernel launches is executed: one for each tile of samples where SAMPLE_TILES = SAMPLES_PER_GPU/SAMPLES_PER_KERNEL. Only a single reduction kernel is launched prior to returning to the CPU thread handling the GPU.

Subsequently we will launch a partial sum reduction kernel. We execute the reduction kernel on a grid where each thread block corresponds with one sample. The kernel performs partial sum reduction on the result array $\mathcal{A}$ to produce the final measurement set $\mathcal{M}$. The j^{th} component in set $\mathcal{M}$ holds the final measurement value of the j^{th} sample. Observe that partial sum reduction is a well studied problem on the GPU and we will therefore not treat it further in this paper. The NVIDIA CUDA SDK version 3.0 contains a sample with code [10] and the next release of CUDPP will also contain sum reduction [11].

To perform the entire computation on the GPU we need to transfer the stereo points $\mathcal{X}$ and the capsules $\{\mathcal{C}^j\}_{j=1}^{J}$ to the GPU device and then read

back the set $\mathcal{M}$ from the GPU device. We also need to setup the intermediate storage $\mathcal{A}$. Since each capsule takes 7 floats to store and each stereo point 3 floats the total memory requirements on device memory is for our typical use: $7JK + 3I + J$ POINT_TILES $+ I \approx 3$ Megabytes. This is far from our upper bound on global memory of 256 Megabytes and means that we can keep all points, capsules and measurements in device memory during execution.

The problem that we have specified is memory bound, since it traverses the set of capsules $\{\mathcal{C}^j\}_{j=1}^{J}$ for every stereo point in $\mathcal{X}$ while the computation does not outweigh the latency of the memory. It is essential that we hide this memory latency. The GPU is perfect for doing exactly this, if enough thread blocks are active and the memory operations are handled with care. For optimal performance it will be necessary to keep data aligned in host memory and ensure coalesced access to host memory by using the 16 Kb shared memory available in each SM (streaming multiprocessor)[2]. Seven threads are used to fetch the data of one capsule (7 floats). In Figure 4 and Listing 1.1 we show how the stereo point data, consisting of the coordinates x, y, and z for a single point, are handled in a similar manner, where every set of three threads is working together to fetch one stereo point (3 floats).

```
/* blockDim.x = POINTS_PER_BLOCK */
__shared__ float Xds[3][blockDim.x];
size_t i = threadIdx.x;
size_t total = blockDim.x*3u;
size_t offset = blockDim.x*blockIdx.y*3u;
for (size_t ii = i; ii < total; ii += blockDim.x)
  Xds[ii%3][ii/3] = Xd[offset+ii];
```

Listing 1.1. All threads in a warp of 32 threads will request data from aligned neighboring addresses in device memory, Xd. This results in two coalesced memory requests of maximum size (64 bytes). The data is then copied to shared memory and organized as illustrated in Figure 4, to avoid bank conflicts.

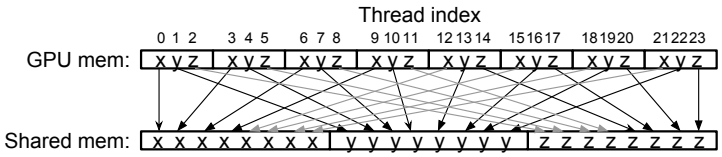


Fig. 4. Ensuring coalesced memory transfers when transferring stereo points from GPU device memory to shared memory. Data is fetched in blocks of 16 and thus aligned in host memory.

[2] The Nvidia Fermi architecture has 64 Kb of cache / shared memory reserved for each SM.

The reason for orchestrating the arrays coordinate-wise in shared memory is to avoid bank conflicts [12]. The GPU is a SIMT (single instruction, multiple thread) architecture and executes in an SM one instruction for a warp of 32 threads. When the 32 threads access a shared memory address, it is crucial that they balance the requests onto all 16 banks. Since the shared memory is organized in a round-robin fashion to the 16 banks, we can make sure that we access neighbouring addresses.

When the GPU executes branch instructions all threads in a warp (32 threads) follow the same branch. This means that if some threads in a warp follow one branch and others follow another branch, all threads must visit both branches and the instruction count goes up. With this in mind we have worked to minimize the number of divergent branches, and where we knew there would be divergent branches, conditional expressions were preferred instead, since both expressions would be evaluated anyway.

The resulting measure kernel uses 24 registers, which means that we can run up to 320 threads on devices with compute capability 1.1 or 1.2 (8192 registers) and 640 threads on devices with compute capability 1.3 (16384 registers). 24 registers is not low enough to completely hide the memory latency, but to go lower would require to split the measure kernel into multiple kernels which could each use less registers. This task would require a huge temporary data set in device memory and thus we concluded that 24 registers is the best we can do. The block size used for the benchmarks in Section 6 is chosen so that the maximum number of active blocks is 8 and can go to either 320 or 640 active threads.

6 Two Orders of Magnitude Speedup

To benchmark the implementation, it was run on the three systems listed in Table 1. For every benchmark, a sequential CPU implementation was also executed and the result values compared for correctness. We varied the number of stereo points and the number of samples to see how well the solution scales for up to 43000 stereo points and 3500 samples. The number of capsules was constant at 48. The current GPU implementation is only limited by the maximum grid sizes and the shared memory, thus it actually supports up to 4.194.240 stereo points and 65535 samples of 64 capsules, which can all fit inside 256Mb device memory.

Table 1. Benchmark systems

System 1	System 2	System 3
Intel Core 2 Quad @ 2.4Ghz	Intel Core 2 Duo @ 2.33Ghz	Intel Core 2 Duo @ 2.4Ghz
4Gb DDR2 800Mhz	4Gb DDR2 800Mhz	2Gb DDR2 667Mhz
Nvidia C1060 Tesla 4Gb	2 * Nvidia 9800GX2 1Gb	Nvidia 8600M GT 256Mb
Compute cap. 1.3	Compute cap. 1.1	Compute cap. 1.1
240 cores @ 1.30Ghz	512 cores @ 1.50Ghz	32 cores @ 0.94Ghz

When comparing the performance of the two 9800GX2 with the C1060, notice that one 9800GX2 actually consists of 2 GPUs with hardware similar to a 8800GTX. This means that we are comparing a system with a total of 4 GPUs with a system with 1 GPU, which gives a disadvantage to the system with 4 GPUs, since the benchmark results include the overhead of handling 4 threads. The plots in Figure 5 clearly shows that the GPU implementation scales linearly with an increasing number of stereo points or samples for both systems. The effect of handling the extra threads can be seen for the smaller problems and we expect that the C1060 will be fastest for small problems. For the largest problem the two 9800GX2 are 2.1 times faster than the C1060, but theoretically two 9800GX2 can actually execute 2.46 times more FLOPS than one C1060. The two 9800GX2 are also more capable at hiding the memory latency, since they can have 4 times 320 active threads, while the C1060 is limited to 512 active threads for our implementation. System 3 was not included in these plots, since the benchmark results was around 20 times slower.

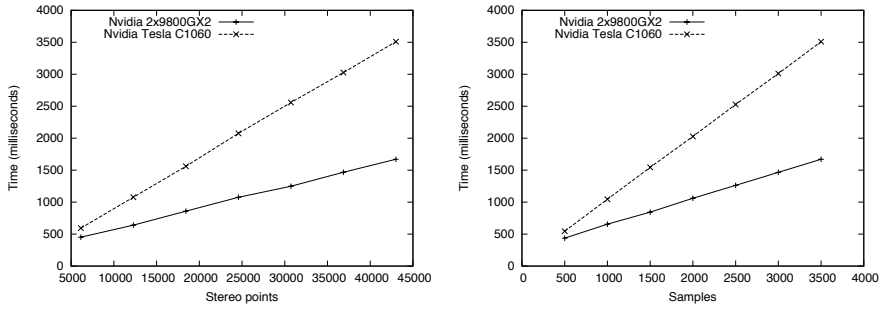


Fig. 5. Plots showing linear scaling for increasing number of stereo points or samples. The number of capsules is kept constant at 48.

The speedup plot in Figure 6 is created using the CPU implementation in Listing 1.2 as the reference. We have used the same input data set for the CPU and the GPU benchmarks. The measurement function used in the CPU implementation (Listing 1.2) is identical to the measurement function used in the GPU implementation (Listing 1.3), but the invocation of the measurement function in listing 1.2 is purely sequential and thus only utilize one core. Since the problem is memory bound, the one thread will have to wait on memory. We expect that an optimized CPU implementation could execute twice as fast, compared to the reference CPU implementation. On the GPU the memory latency has been successfully hidden, which becomes apparent when looking at the speedup numbers in Figure 6.

```
for(size_t j = 0; j < J; ++j)
{
  M[j] = 0.0f;
  for(size_t i = 0; i < I; ++i)
  {
    size_t const ii = i*3u;
    float3 const x_i = make_float3(X[ii],X[ii+1],X[ii+2] );
    float value = MAX_DISTANCE;
    for(size_t k = 0u; k < K; ++k)
    {
      size_t const kk = (j*K + k)*7u;
      float3 const a = make_float3(C[kk],   C[kk+1], C[kk+2] );
      float3 const b = make_float3(C[kk+4],C[kk+5], C[kk+6] );
      float  const r = C[kk+3];
      value = min( measurement(  x_i, r, a, b ), value );
    }
    M[j] += value;
  }
}
```

Listing 1.2. The CPU implementation used for benchmarking. This code is executed in a single thread for the CPU.

```
/* Extracted from the body of the measurement_kernel */
float3 const x_i=make_float3(Xds[0][i], Xds[1][i], Xds[2][i]);
float value = Ads[i];
for(size_t k = 0u; k<K; ++k)
{
  float3 const a = make_float3(Cds[0][k],Cds[1][k],Cds[2][k]);
  float3 const b = make_float3(Cds[4][k],Cds[5][k],Cds[6][k]);
  float  const r = Cds[3][ k ];
  value = min( value,  measurement( x_i, r, a, b ) );
}
Ads[i] = value;
```

Listing 1.3. The GPU implementation, which computes results identical (apart from rounding differences) to the CPU implementation in Listing 1.2. This code is executed in $J * I$ threads for the GPU.

The 8600M GT achieves a stable speedup of ≈ 20, while the others increase in speedup until reaching their maximum stage. The increase in speedup is explained by the overhead of running many kernels. For these benchmarks a kernel was called for every 8 samples, thus the overhead of calling a kernel takes up a larger proportion when the problem size is small and the GPUs are fast.

The fact that we see a correlation in Figure 6 between the speedup of the GPUs and with the GPU hardware specifications, means that we can conclude that the GPU implementation has succeeded to utilize the GPUs efficiently.

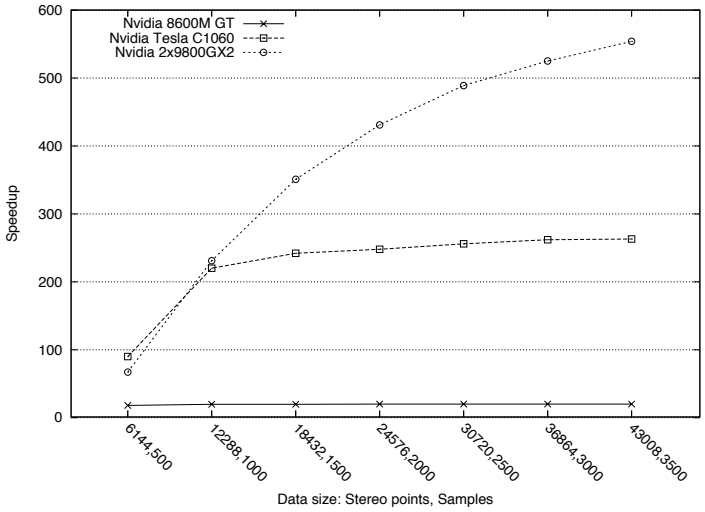


Fig. 6. The speedup achieved when computing a data set of the specified size on a GPU vs. the CPU. The number of capsules is kept constant at 48.

7 Conclusions and Future Work

In this work we have presented a tiling approach that results in a very efficient GPU acceleration of the measurement process for articulated tracking with a particle filter. The main causes to our two orders of magnitude speedup factor lies in careful hiding memory latencies from device memory and avoiding memory bank conflicts in the shared memory. We not only gain from the raw processing power of the GPU, but also from its alternative memory layout.

Our future work involves benchmarking on small scale GPU clusters as this may further interactive markerless computer vision based articulated tracking. Besides this, the sampling process of the particle filter is currently implemented in a naive consumer-producer scheme using a single CPU thread for each sample. This appears to be the next performance bottleneck that we will investigate.

References

1. Poppe, R.: Vision-based human motion analysis: An overview. Computer Vision and Image Understanding **108** (2007) 4–18
2. Cappé, O., Godsill, S.J., Moulines, E.: An overview of existing methods and recent advances in sequential Monte Carlo. Proceedings of the IEEE **95** (2007) 899–924
3. Sminchisescu, C., Triggs, B.: Kinematic Jump Processes for Monocular 3D Human Tracking. In: In IEEE International Conference on Computer Vision and Pattern Recognition. (2003) 69–76
4. Deutscher, J., Blake, A., Reid, I.: Articulated body motion capture by annealed particle filtering. In: cvpr, Published by the IEEE Computer Society (2000) 2126

5. Hauberg, S., Sommer, S., Pedersen, K.S.: Gaussian-like spatial priors for articulated tracking. In: Proceedings of ECCV'10. Lecture Notes in Computer Science, Springer (2010)
6. Bandouch, J., Beetz, M.: Tracking Humans Interacting with the Environment Using Efficient Hierarchical Sampling and Layered Observation Models, IEEE Int. Workshop on Human-Computer Interaction (HCI) (2009)
7. Cabido, R., Concha, D., Pantrigo, J.J., Montemayor, A.S.: High Speed Articulated Object Tracking Using GPUs: A Particle Filter Approach. In: 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks, IEEE (2009) 757–762
8. Rohr, K.: Towards model-based recognition of human movements in image sequences. CVGIP-Image Understanding **59** (1994) 94–115
9. Sidenbladh, H., Black, M.J., Fleet, D.J.: Stochastic tracking of 3d human figures using 2d image motion. In: Proceedings of ECCV'00. Volume II of Lecture Notes in Computer Science 1843., Springer (2000) 702–718
10. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for gpu computing. In: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, Aire-la-Ville, Switzerland, Eurographics Association (2007) 97–106
11. CUDPP: Cuda data parallel primitives library. Accessed Online April (2010) `http://code.google.com/p/cudpp/`.
12. NVIDIA Corporation: NVIDIA CUDA Best Practices Guide. (2010) version 3.0.

A Computing the Ray–Capsule Intersection Point

To find the intersection between the ray and the capsule we first consider the situation with an infinitely long capsule. Here we can find the point of intersection by first finding the point $\boldsymbol{y}_i$ on the line through $\boldsymbol{a}$ and $\boldsymbol{b}$ that is closest to the ray $\boldsymbol{x}_i(\Delta)$. By orthogonal projection we find this as

$$\boldsymbol{y}_i = \boldsymbol{a} + \left((\boldsymbol{x}_i(\Delta) - \boldsymbol{a})^T \boldsymbol{c} \right) \boldsymbol{c} \tag{7}$$

where we have defined $\boldsymbol{c} = \frac{\boldsymbol{b}-\boldsymbol{a}}{\|\boldsymbol{b}-\boldsymbol{a}\|}$.

At the point of intersection between the ray and the infinite capsule we must have

$$\| \boldsymbol{x}_i(\Delta) - \boldsymbol{y}_i \|^2 = r^2 \ . \tag{8}$$

Inserting the ray definition from Eq. 5 gives us

$$r^2 = \| \boldsymbol{p} + \boldsymbol{v}\Delta - \boldsymbol{y}_i \|^2 = \| \boldsymbol{v}_\perp \Delta + \boldsymbol{p}_\perp \|^2 \ , \tag{9}$$

where $\mathbf{P} = \left(\mathbf{I} - \boldsymbol{c}\boldsymbol{c}^T \right)$ and $\boldsymbol{v}_\perp = \mathbf{P}\boldsymbol{v}$ and $\boldsymbol{p}_\perp = \mathbf{P}\left(\boldsymbol{p} - \boldsymbol{a}\right)$. This is readily identified as a second order polynomial in Δ

$$P_c(\Delta) = \boldsymbol{v}_\perp^T \boldsymbol{v}_\perp \Delta^2 + 2\boldsymbol{v}_\perp^T \boldsymbol{p}_\perp \Delta + \boldsymbol{p}_\perp^T \boldsymbol{p}_\perp - r^2 = 0 \ . \tag{10}$$

If no roots to this polynomial exist then the ray does not intersect the infinite long capsule. Otherwise we solve for the minimum positive root Δ_{cap} which will give us the intersection point on the infinite long capsule.

In practice, the skeleton model does not have infinite long limbs and as such we do not have infinite long capsules. The above approach thus needs to be modified to cope with finite capsules. In the case where $0 \leq \boldsymbol{c}^T(\boldsymbol{y}-\boldsymbol{a}) \leq 1$ the above analysis still holds. In all other cases we only need to see if the ray intersects with the spheres of radius r centred in $\boldsymbol{a}$ and $\boldsymbol{b}$. If the ray intersects the sphere centred in $\boldsymbol{a}$, we must have

$$\| \boldsymbol{x}_i(\Delta) - \boldsymbol{a} \|^2 = r^2 \ . \tag{11}$$

Once again, this gives as a second order polynomial

$$P_a(\Delta) = \boldsymbol{v}^T\boldsymbol{v}\Delta^2 + 2\boldsymbol{v}^T(\boldsymbol{p}-\boldsymbol{a})\,\Delta + (\boldsymbol{p}-\boldsymbol{a})^T(\boldsymbol{p}-\boldsymbol{a}) - r^2 = 0 \ . \tag{12}$$

If this polynomial has no roots then the ray does not intersect the sphere centred in $\boldsymbol{a}$. If it does have roots, we find the intersection from the smallest positive root. A similar treatment can be given to the sphere centred in $\boldsymbol{b}$.

Thus, the ray intersection algorithm will solve three second order polynomials $P_a(\Delta)$, $P_b(\Delta)$, and $P_c(\Delta)$ and use some **if**-statements that will determine select the proper smallest positive root as the ray intersection length.