Grafik İşlemcileri Üzerinde Paralel Parçacık Süzgeci Kullanarak Tempo Takibi
Tempo Tracking by Using a Parallel Particle Filter on the GPU

Ertuğ Karamatlı, Ali Taylan Cemgil
Bilgisayar Mühendisliği Bölümü
Boğaziçi Üniversitesi
{ertug.karamatli,taylan.cemgil}@boun.edu.tr

***Özetçe* —Son zamanlarda grafik işlem birimi (GİB) kullanılarak hızlandırılan uygulamalar artıyor. Parçacık süzgeçleri de bu uygulamalardan biri. Tempo takibi ise müzik işlemedeki temel problemlerden biridir. Bu çalışmada, ölçü işaretçisi modelinin parçacık süzgeci kullanarak CUDA üzerinde uygulanışını sunuyoruz. Parçacık süzgecini CUDA mimarisine uyarlamak için paralel algoritmalar kullanıyoruz. Oluşturduğumuz yapay gözlem verisiyle başarılı bir şekilde tempo takibi yapılabildiğini gösteriyoruz ve farklı parçacık sayıları için uygulamanın çalışma sürelerini veriyoruz.**

Anahtar Kelimeler*—*Parçacık Süzgeci, Tempo Takibi, Grafik İşlem Birimi

***Abstract*—Recently, using graphics processing unit (GPU) for accelarating applications is becoming very popular and particle filters are no exception. Tempo tracking is one of the basic problems in music processing. In this paper, we present an implementation of the bar pointer model with a particle filter on CUDA. We describe the algorithms used to implement the parallel particle filter. Then, in order to demonstrate the implementation, we create a simulated observation data and run the filter on it. We also give the running times of the application for different number of particles.**

Keywords*—*Particle Filter, Tempo Tracking, Graphics Processing Unit

I. GİRİŞ

Tempo bir müzik eserinin icra edilme hızıdır. Çoğumuz müzik dinlerken vuruşlara göre ayağıyla ritim tutabilir ve tempoyu hissedebilir. Ancak bunu bilgisayarla yapmak oldukça zordur. Tempo takibi müzik işleme alanındaki temel problemlerden biridir. Bu problem için Whiteley *v.d.* [1] *ölçü işaretçisi* (bar pointer) modelini önermiştir. Bu istatistiksel model sadece nota başlangıçlarını gözlemleyerek tempo çıkarımı yapabiliyor. Bir saklı Markov modeli (SMM) olduğu için ayrık bir uzayda çalışır. Yeterli çözünürlüğün sağlanabilmesi için çok durum gerekir ve uzayın büyüklüğü nedeniyle hesaplama maliyeti yüksektir.

Parçacık süzgeçleri (particle filter), diğer adıyla ardışık Monte Carlo (sequential Monte Carlo) yöntemleri, doğrusal olmayan dinamik sistemlerde kestirim yapmak için kullanılır. Hedef takibi ve bilgisayarla görü gibi birçok alanda uygulanmaktadır. Yine Whiteley *v.d.* [2] ölçü işaretçisi modelinin SMM'deki ayrık uzay yerine sürekli bir uzayda çalışabilmesi için bir parçacık süzgeci yöntemi önermiştir. Paralel parçacık süzgeçlerinde en çok hesaplama gerektiren kısım genellikle yeniden örneklemedir (resampling). Bunun sebebi tüm parçacıkların birbiriyle etkileşmesidir. Çeşitli yeniden örnekleme yöntemlerinin CUDA üzerindeki uygulamaları [3]'de karşılaştırılmıştır.

Bu çalışmada, ölçü işaretçisi modelinin parçacık süzgeci kullanarak CUDA üzerinde uygulanışını sunuyoruz. Parçacık süzgecini CUDA mimarisine uyarlamak için paralel algoritmalar kullanıyoruz. Oluşturduğumuz yapay gözlem verisiyle bir deney yapıyoruz ve farklı parçacık sayıları için uygulamanın çalışma sürelerini veriyoruz.

A. Parçacık Süzgeci

Parçacık süzgeci, doğrusal olmayan durum-uzay modellerinde kestirim yapmak için kullanılan bir yöntemdir. Parçacık süzgeçleri hakkında detaylı bilgi için bakınız [4]. Durum-uzay modelleri aşağıdaki biçimde ifade edilebilir.

$$x_0 \sim \pi(x_0) \tag{1}$$
$$x_k \sim f(x_k|x_{k-1}) \tag{2}$$
$$y_k \sim g(y_k|x_k) \tag{3}$$

Burada x_k saklı değişkendir ve bir Markov zinciri oluşturur, y_k ise gözlemdir. Sistemin ilk durumu $\pi(x_0)$ şeklindeki önsel dağılımla belirlenir. Sistem $f(x_k|x_{k-1})$ şeklindeki *geçiş yoğunluğuna* (transition density) göre evrilir. Gözlemler $g(y_k|x_k)$ şeklindeki *gözlem yoğunluğuna* (observation density) göre oluşur. Parçacık süzgeci algoritmasında *öneri yoğunluğu* (proposal density) olarak geçiş yoğunluğu seçildiğinde *artımlı ağırlıkta* (incremental weight) sadeleşme olur ve sadece gözlem yoğunluğu kalır. Elde edilen algoritmaya *bootstrap parçacık süzgeci* denir. Şekil 1 bu algoritmayı gösteriyor.

B. CUDA

CUDA, NVIDIA tarafından geliştirilen ve GİB'leri genel amaçlı hesaplamaya uygun hale getiren bir platformdur [5].

978-1-4799-4874-1/14/$31.00

for $i = 1, \ldots, N$ **do**
  Örnekle: $\hat{x}_0 \sim p(x_0)$
end for
for $k = 1, \ldots, K$ **do**
  for $i = 1, \ldots, N$ **do**
    Türet: $\hat{x}_k^{(i)} \sim f(\hat{x}_k^{(i)} | x_{k-1}^{(i)})$
    Ağırlık hesapla: $\bar{w}_k^{(i)} = g(y_k | \hat{x}_k^{(i)})$
  end for
  Ağırlıkları normalize et:

$$w_k^{(i)} = \frac{\bar{w}_k^{(i)}}{\sum_{j=1}^{N} \bar{w}_k^{(j)}}, \quad i = 1, \ldots, N$$

  Ağırlıklara göre yeniden örnekle
end for

Şekil 1: Bootstrap parçacık süzgeci algoritması.

NVIDIA GİB'leri duraksız çoklu işlemcilerden (streaming multiprocessors) oluşmaktadır. Tek Komut Çoklu Veri (TKÇV) mimarisine benzer Tek Komut Çoklu İş Parçacığı (TKÇİP) kullanılır. TKÇV'nin aksine TKÇİP'de çoklu işlemciler aynı anda en fazla 32 iş parçacığı çalıştırır ve buna *çözgü* (warp) denir. Bir çoklu işlemciye verilen iş parçacıkları sıralanarak çözgüler halinde çalıştırılır.

Paylaşımlı ve evrensel olmak üzere iki ana bellek çeşidi vardır. Paylaşımlı bellek hızlı olmasına rağmen boyut ve erişebilen iş parçacığı sayısı (en fazla 1024) bakımından kısıtlıdır. Evrensel bellek ise büyük olmasına ve bütün iş parçacıkları tarafından erişilebilmesine rağmen yavaştır. Aynı paylaşımlı bellek ve çoklu işlemci üzerinde çalışan iş parçacığı grubuna iş parçacığı bloğu denir ve en fazla 1024 iş parçacığından oluşabilir. Birden fazla iş parçacığı bloğuna ise *ızgara* (grid) denir. Çalıştırılabilecek ızgara boyutu oldukça büyüktür ve genellikle eldeki verinin boyutuna göre değişir. Böylece, işlemci sayısından bağımsız bir şekilde uygulama geliştirilebilir. Izgara içinde bulunan bloklar çalıştırıldığı GİB'deki işlemcilere otomatik olarak paylaştırılır.

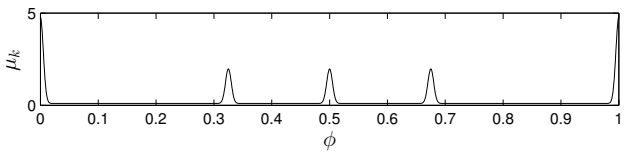
II. ÖLÇÜ İŞARETÇİSİ MODELİ

Ölçü işaretçisi, bir ölçü uzunluğundaki ritmik örüntü içerisinde bulunulan konumu gösterir ve saklıdır [1]. Nota başlangıçları gözlemlenerek ölçü işaretçisinin konumu hakkında çıkarım yapılır. Ölçü işaretçisinin hızı tempo ile orantılıdır. Ritmik örüntü, nota başlangıçlarının ölçü içerisindeki bazı kısımlarda daha yüksek olasılıkla gözlemlenmesi esasına dayanır ve bunu sayısal olarak tanımlar. Şekil 2 örnek bir ritmik örüntü fonksiyonu gösteriyor.

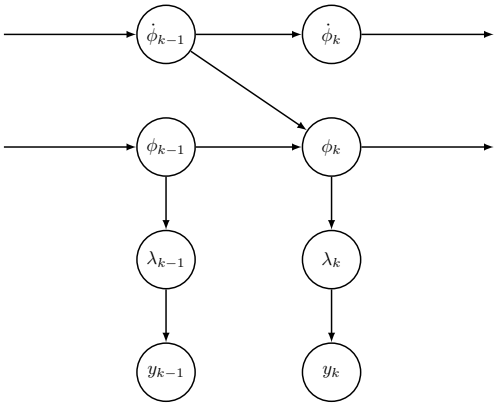
A. Geçiş Modeli

$t_k = k\Delta$ anında $k \in \{1, 2, \ldots, K\}$ ve Δ gözlemler arasındaki zamanı gösteren bir sabittir. Ölçü işaretçisinin konumu $\phi_k \in [0, 1)$ ile gösterilmiştir. Ölçü işaretçisinin hızı ise $\dot{\phi}_k \in [\dot{\phi}_{min}, \dot{\phi}_{max}]$ ile gösterilmiştir ve burada $\dot{\phi}_{min} > 0$. Ölçü işaretçisinin evrimi aşağıda tanımlanmıştır.

$$\phi_{k+1} = (\phi_k + \Delta\dot{\phi}_k) \mod 1 \tag{4}$$



Şekil 2: Örnek bir ritmik örüntü fonksiyonu.



Şekil 3: Ölçü işaretçisinin grafik modeli.

$$p(\dot{\phi}_{k+1} | \dot{\phi}_k) \propto \begin{cases} \mathcal{N}(\dot{\phi}_k, \sigma_\phi^2) & \dot{\phi}_{min} \leq \dot{\phi}_{k+1} \leq \dot{\phi}_{max} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Özet olarak, $\mathbf{x}_k \equiv [\phi_k \ \dot{\phi}_k]^T$ sistemin k anındaki durumunu gösterir. Şekil 3 grafik modeli gösteriyor.

B. Gözlem Modeli

Gözlemlenen nota başlangıçlarının Poisson dağılımına göre gerçekleştiği varsayılmıştır. Bu dağılımın λ parametresi ise ölçü işaretçisinin konumu ve ritmik örüntüye bağlı bir önsel gamma dağılımına sahiptir. Bu şekilde, ritmik örüntü ile tanımlanan nota başlangıcı gözlenme olasılığının yüksek olduğu bölgelerde λ parametresinin değeri artar. λ parametresinin ayrıca çıkarımına gerek olmadığı için üzerinden integral alınmıştır. Detaylar için bakınız [2]. Sonuç analitik olarak bulunabilmektedir ve aşağıda verilmiştir.

$$a_k = \mu(\phi_k)^2 / Q_\lambda \tag{6}$$
$$b_k = \mu(\phi_k) / Q_\lambda \tag{7}$$

$$p(y_k | \phi_k) = \frac{b_k^{a_k} \Gamma(a_k + y_k)}{y_k! \Gamma(a_k)(b_k + \Delta)^{a_k + y_k}} \tag{8}$$

III. PARALEL UYGULAMA

Uygulama, CUDA C programlama dili [5] ve GNU/Linux platformu için bir CUDA öykünücüsü olan GPU Ocelot [6] kullanılarak geliştirilmiştir. Testler ise Windows 7 üzerinde GeForce GT 540M kullanılarak yapılmıştır. Hız ve bellek ihtiyacı göz önüne alınarak tüm kayan noktalı sayılar için tek duyarlık (single precision) kullanılmıştır. Tek duyarlıkta çifte duyarlığa göre fark edebildiğimiz bir başarım kaybı olmamıştır.

A. Parçacık Türetimi

Parçacık türetmek için tekdüze önsel dağılımlar ve geçiş yoğunluğunu kullanıyoruz:

$$\phi_0^{(i)} \sim \mathcal{U}(0,1) \tag{9}$$

$$\dot{\phi}_0^{(i)} \sim \mathcal{U}(\dot{\phi}_{min}, \dot{\phi}_{max}) \tag{10}$$

$$\hat{\mathbf{x}}_k^{(i)} \sim p(\mathbf{x}_k^{(i)} | \mathbf{x}_{k-1}^{(i)}) \tag{11}$$

Yukarıda görülebileceği üzere parçacık türetmek için tekdüze ve normal dağılımdan rasgele sayı üretmek gerekir. Neyse ki, CUDA araç takımındaki CURAND kütüphanesi sayesinde GİB üzerinde sözde rasgele sayılar (pseudorandom number) üretilebiliyor. CURAND, iletişim yükü yaratmadan ve paralel bir şekilde rasgele sayı üretmek için her iş parçacığındaki rasgele sayı üretecini durum sırasında belli aralıklarla başlatır (skip-ahead). Tekdüze, normal, log normal ve Poisson dağılımlarından verimli bir şekilde rasgele sayı üretmek için fonksiyonlar sunar.

İlklendirme (initialization) ve türetim işleri tamamen paraleldir çünkü iş parçacıkları birbirlerinden tamamen bağımsızdır. Yalnız küçük bir *dallanma ıraksaklığı* (branch divergence) vardır. Geçiş yoğunluğundan örnekleme yaparken normal dağılımdan alınan örneğin $[\dot{\phi}_{min}, \dot{\phi}_{max}]$ aralığında olduğundan emin olmak gerekir. Eğer dışarıdaysa tekrar örnek alınır. CUDA mimarisinde çözgü içinde bir iş parçacığı diğer iş parçacıklarından farklı bir yere dallanırsa iş parçacıkları ardışık çalışmaya başlar. Neyse ki, bu nispeten ender olan bir durumdur ve başarıma etkisi düşüktür.

B. Ağırlık Hesaplama

Parçacık ağırlıklarını hesaplamak için gözlem yoğunluğunu kullanıyoruz:

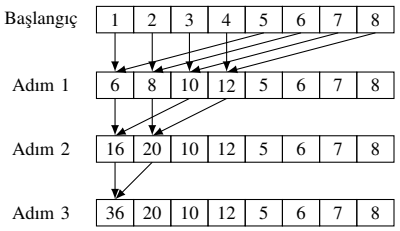
$$\bar{w}_k^{(i)} = p(y_k | \hat{\mathbf{x}}_k^{(i)}) \tag{12}$$

Ağırlıkların toplamının 1 olması gerektiği için normalize ediyoruz:

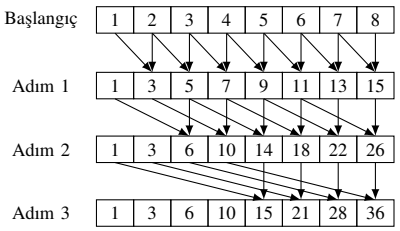
$$w_k^{(i)} = \frac{\bar{w}_k^{(i)}}{\sum_{j=1}^{N} \bar{w}_k^{(j)}} \tag{13}$$

Gözlem yoğunluğunun formülünü kullanarak ağırlıkları bağımsız bir şekilde paralel hesaplamak kolay olmasına rağmen normalize ederken tüm ağırlıkları toplamak gerekir. Bu işlem tüm ağırlıkları içerdiği için paralelleştirmeyi zorlaştırıyor. Bu toplamı verimli bir şekilde hesaplayabilmek için paralel indirgeme yöntemini kullandık.

CUDA'da büyük diziler üzerinde toplama yapmak için paralel indirgeme [7] yöntemi kullanılır. Bu yöntem, her adımda dizinin iki yarısını toplayarak çalışır ve son adımda dizide tek eleman kalır. Şekil 4 bir örnek gösteriyor. Adım sayısı $\mathcal{O}(\log N)$, işlem sayısı $\mathcal{O}(N)$ karmaşıklığa sahiptir. Bu yöntemi CUDA mimarisi üzerinde verimli kullanabilmek için, her iş parçacığı bloğu önce kendi paylaşımlı belleği üzerinde indirgeme yapar ve ara sonuçları evrensel belleğe geri yazar. Daha sonra tek bir iş parçacığı bloğu önceki tüm ara sonuçlar üzerinde tekrar indirgeme yapar ve sonuca ulaşılır. Toplam hesaplandıktan sonra tamamen paralel olarak tüm ağırlıklar toplama bölünür.



Şekil 4: Paralel indirgeme ile toplamaya bir örnek.



Şekil 5: Paralel tarama (Hillis-Steele) ile birikimli toplamaya bir örnek.

C. Yeniden Örnekleme

Monte Carlo varyansını azalttığı ve bir adet tekdüze rasgele sayıya ihtiyaç duyduğu için *düzenli* (systematic) yeniden örnekleme yöntemini kullandık. Yeniden örnekleme iki adımdan oluşur: ağırlıkların birikimli dağılım fonksiyonunu (BDF) hesaplamak ve bu BDF'yi kullanarak tüm parçacıkları yeniden örneklemek.

BDF'yi hesaplamak, ağırlıkları normalize ederken olduğu gibi tüm ağırlıkları içerir ve paralelleştirmek zordur. BDF'yi verimli bir şekilde hesaplayabilmek için paralel tarama yöntemini kullandık. Bu yöntemde hesaplama bir ağaç yapısı şeklinde yapılır. Şekil 5 Hillis-Steele [7] algoritmasına bir örnek gösteriyor. Adım sayısı $\mathcal{O}(\log N)$, işlem sayısı $\mathcal{O}(N \log N)$ karmaşıklığa sahiptir. Biz Blelloch [7] algoritmasını kullandık. Bu algoritmada adım sayısı $\mathcal{O}(2 \log N)$, işlem sayısı $\mathcal{O}(N)$ karmaşıklığa sahiptir.

Düzenli yeniden örnekleme aşağıdaki şekilde yapılır:

$$u \sim \mathcal{U}(0,1) \tag{14}$$

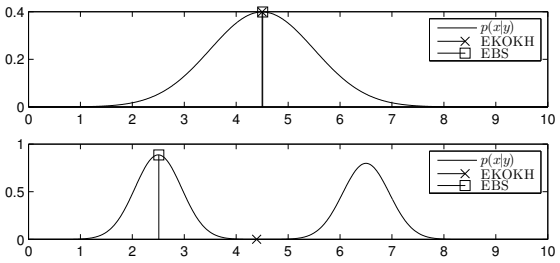
$$u^{(j)} = \frac{u + j}{N} \tag{15}$$

Burada j parçacık indisidir. Ters BDF'de $u^{(j)}$ değeri aranır ve bulunan parçacık j parçacığına kopyalanır. Bu işlemi hızlandırmak için ikili arama (binary search) kullandık. Böylece, Merkezi İşlem Birimi'nde (MİB) tekdüze rasgele sayı u üretildikten sonra her parçacık için paralel olarak bağımsız ikili arama yapılır.

D. Kestirim

Parçacık süzgeciyle kestirim yapmak için genellikle en küçük ortalama karesel hata (EKOKH, MMSE) kestiricisi kullanılır:

$$\hat{x}_{\text{EKOKH}} = \underset{\hat{x}}{\operatorname{argmin}} \, \mathrm{E}[(\hat{x} - x)^2 | y] = \mathrm{E}[x|y] \tag{16}$$



Şekil 6: EKOKH ve EBS kestiricilerini karşılaştıran bir örnek. Tek tepeli simetrik dağılımlarda EKOKH ve EBS kestirimi aynıdır, çok tepeli dağılımlarda ise farklıdır. Bu örnekteki iki tepeli dağılımda EBS iyi bir kestirim yapmasına karşın EKOKH'nin olasılığı çok düşük bir sonuç ürettiği görülüyor.

Fakat ölçü işaretçisi modelinde sonsal dağılım çok tepelidir (multimodal). Bunun sebebi ritmin farklı fazları ve temponun katları da yüksek olasılıklara sahip olabilir. Bu yüzden EKOKH kestiricisi kullanıldığında sonuçlar her zaman doğru olmayabilir. Bu durumda en büyük sonsal (EBS, MAP) kestiricisi daha uygundur:

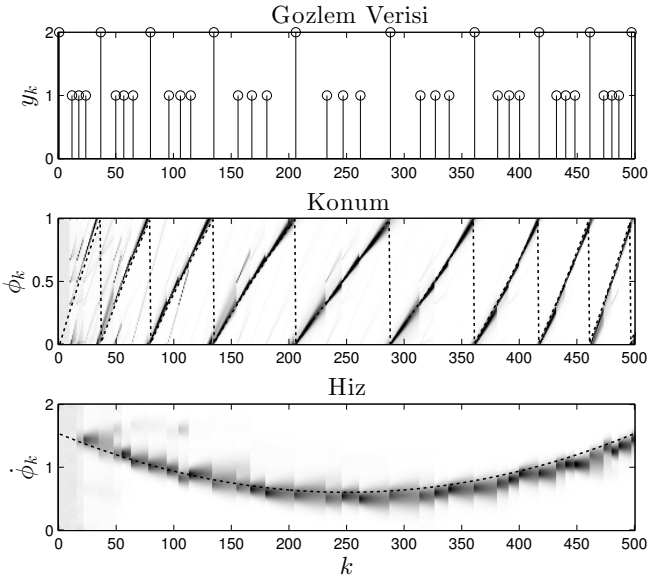
$$\hat{x}_{\text{EBS}} = \underset{x}{\operatorname{argmax}}\, p(y|x)p(x) \tag{17}$$

Şekil 6 bu iki kestiriciyi karşılaştırıyor. EBS kestirimi için Viterbi algoritması Godsill *v.d.* [8] tarafından parçacık süzgeçlerine uyarlanmıştır. Bu algoritma iyi kestirim yapabilmesine rağmen karmaşıklığı $\mathcal{O}(KN^2)$ olduğu için parçacık sayısının yüksek olduğu buradakine benzer uygulamalar için uygun değildir. Parçacık süzgeçleri için paralel EBS kestirimi ayrıca araştırılması gereken bir konudur. Biz bu uygulamada EKOKH kestiricisini kullandık ve yaptığımız deneylerde yeterince iyi çalıştığını gözlemledik. Kestirim yaparken paralel indirgeme ile toplam bulunur ve MİB üzerinde parçacık sayısına bölünür.

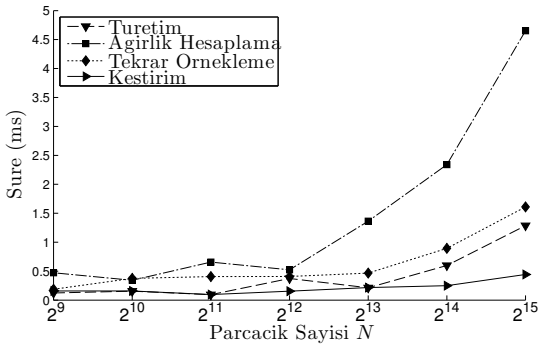
IV. SONUÇLAR VE VARGILAR

Uygulamayı test edebilmek için orijinal makaledekine [2] benzer yapay bir gözlem verisi oluşturduk. Şekil 2 kullandığımız ritmik örüntü fonksiyonunu, Şekil 7'de üstteki grafik ise üretilen gözlem verisini gösteriyor. Kullandığımız parametreler: $\dot{\phi}_{min} = 0.1$, $\dot{\phi}_{max} = 2$, $\Delta = 0.02$s ve $\sigma_\phi^2 = 0.0005$. Şekil 7 yaptığımız deneyin sonuçlarını gösteriyor. Şekil 8 farklı parçacık sayılarıyla yaptığımız deneylerde algoritmadaki işlerin aldığı ortalama süreyi gösteriyor.

Parçacık sayısı $N = 32768$ iken süzgecin bir adımı ortalama 8 ms sürüyor. Kullandığımız gözlem aralığı $\Delta = 0.02$s olduğu için bu süre gerçek zamanlı çalışmaya olanak sağlıyor. Geriye kalan 12 ms içinde ses sinyalinden nota başlangıçlarını bulan bir algoritma kullanıldığında gerçek zamanlı tempo takibi yapılabilir. Parçacıkların etkileşimi azaltılarak paralellik artırılabilir. Örneğin, *Metropolis yeniden örnekleme* [3] etkileşimi azaltıp hızlanma sağlıyor.



Şekil 7: CUDA üzerinde tempo takibi. Konum ve hız grafiklerinde, noktalı çizgi gerçek değerleri, arkasındaki koyu bölgeler ise parçacık yoğunluğunu ifade ediyor. Konum ve hızın doğru takip edildigi görülüyor. Sadelik için EKOKH kestirimi verilmemiştir ama gerçek değerlere çok yakındır. Parçacık sayısı $N = 32768$.



Şekil 8: Süzgecin bir adımdaki ortalama işleme süresinin işlere göre kırılımı. Gözlem yoğunluğundaki işlem miktarının yüksek olması sebebiyle en çok süreyi ağırlık hesaplama alıyor.

KAYNAKÇA

[1] N. Whiteley, A. T. Cemgil, and S. J. Godsill, "Bayesian modelling of temporal structure in musical audio," in *Proceedings of International Conference on Music Information Retrieval*, 2006, pp. 29–34.

[2] N. Whiteley, A. T. Cemgil, and S. Godsill, "Sequential inference of rhythmic structure in musical audio," in *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing ICASSP 2007*, vol. 4, 2007, pp. IV–1321–IV–1324.

[3] L. Murray, A. Lee, and P. E. Jacob. (2013) Rethinking resampling in the particle filter on graphics processing units. arXiv preprint. [Online]. Available: http://arxiv.org/abs/1301.4019v1

[4] O. Cappe, S. Godsill, and E. Moulines, "An overview of existing methods and recent advances in sequential monte carlo," *Proceedings of the IEEE*, vol. 95, no. 5, pp. 899–924, 2007.

[5] CUDA C Programming Guide. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[6] GPU Ocelot. [Online]. Available: http://code.google.com/p/gpuocelot

[7] H. Nguyen, *GPU Gems 3*, 1st ed. Addison-Wesley Professional, 2007.

[8] S. Godsill, A. Doucet, and M. West, "Maximum a posteriori sequence estimation using monte carlo particle filters," *Annals of the Institute of Statistical Mathematics*, vol. 53, no. 1, pp. 82–96, 2001.