TÜBİTAK

Yüksek Başarımlı Permanent Hesaplama Algoritmaları

Program Kodu: 1002

Proje No: 119E544

Proje Yürütücüsü:
Dr. Kamer KAYA

Bursiyer(ler):
Fatih TAŞYARAN

Haziran 2021
İSTANBUL

İçindekiler

1 Giriş **1**

1.1 Permanent ve Eşleme 1

2 Literatür Özeti **4**

3 Gereç ve Yöntem **5**

3.1 Yoğun Matrislerin Permanent Değerinin Hesaplanması 5

3.1.1 CPU Üzerinde Yoğun Matrislerin Permanent Değerinin Hesaplanması . . 6

3.1.2 GPU Üzerinde Yoğun Matrislerin Permanent Değerinin Hesaplanması . . 6

3.1.2.1 `Ryser-Gx`: **x** dizisinin GPU ana belleğinde yaratılması 8

3.1.2.2 `Ryser-Rx`: **x** dizisinin GPU yazmaçlarına taşınması 9

3.1.2.3 `Ryser-Sx`: **x** dizisinin paylaşımlı hafızada yaratılması 10

3.1.2.4 `Ryser-SxC`: **x** dizisinin hafıza birleştirme ile paylaşımlı hafıza üzerinde yaratılması. 11

3.1.2.5 `Ryser-SxC-Sm`: **x** ve girdi matrisin paylaşımlı hafızada tutulup hafıza birleştirme ile erişilmesi 11

3.1.3 Birden Fazla GPU Üzerinde Yoğun Matrislerin Permanent Değerinin Hesaplanması 12

3.1.3.1 `Ryser-SxC-Sm-MG`: `Ryser-SxC-Sm` algoritmasının birden fazla GPU üzerinde statik parçalamayla çalıştırılması 12

3.1.4 CPU ve GPU Üzerinde (Hibrit) Yoğun Matrislerin Permanent Değerinin Hesaplanması 12

3.1.4.1 `Ryser-SxC-Sm-MG+`: `Ryser-SxC-Sm` ve `ParallelRyser` algoritmalarının GPU ve CPU üzerinde aynı anda çalıştırılması 12

3.2 Seyrek Matrislerin Permanent Değerinin Hesaplanması 13

3.2.1 CPU Üzerinde Seyrek Matrislerin Permanent Değerinin Hesaplanması . . 14

3.2.1.1 CPU üzerinde `Skipper` algoritmasının gerçeklenmesi 14

3.2.1.2 CPU üzerinde `SpaRyser` algoritmasının gerçeklenmesi 15

3.2.2 GPU Üzerinde Seyrek Matrislerin Permanent Değerinin Hesaplanması . 18

3.2.2.1 `ParallelSkipper`: Skipper algoritmasının GPU üzerinde gerçeklenmesi 18

3.2.2.2 `SkipPer-SxC-Sm`: **x**, `CRS`, `CCS` dizilerinin *paylaşımlı hafıza* üzerinde yaratılıp, *hafıza birleştirme* ile erişilmesi 19

3.2.2.3 `SpaRyser-SxC-Sm`: **x** ve `CCS` gösterimlerinin *paylaşımlı hafıza*da yaratılarak *hafıza birleştirme* ile erişilmesi. 22

3.2.3 Birden Fazla GPU ve hibrit düzende Seyrek Matrislerin Permanent Değerinin Hesaplanması 24

3.2.3.1 `SpaRyser-SxC-Sm-MG+`: `SpaRyser-SxC-Sm` ve `ParallelSpaRyser` algoritmalarının hibrit düzende kullanılması 24

3.2.3.2 SkipPer-SxC-Sm-MG+: SkipPer-SxC-Sm ve ParallelSkipper algoritmalarının hibrit gerçeklenmesi . . . 24
3.3 Permanent Değerinin Yakınsanması . . . 24
3.4 CPU Üzerinde Permanent Değerinin Yakınsanması . . . 28
3.5 Yoğun Matrislerin Permanent Değerlerinin Yakınsanması . . . 28
3.5.1 GPU Üzerinde Permanent Yakınsama . . . 28
3.5.1.1 RasmussenGpu: Rasmussen algoritmasının GPU üzerinde gerçeklenmesi . . . 28
3.5.1.2 Rasmussen+Gpu: Rasmussen+ algoritmasının GPU üzerinde gerçeklenmesi . . . 30
3.5.1.3 ScalingGpu: Scaling algoritmasının GPU üzerinde gerçeklenmesi 30
3.5.2 Birden Fazla GPU ve Hibrit Düzende Permanent Yakınsama . . . 32
3.5.2.1 Rasmussen+MGpu: Rasmussen+ algoritmasının birden fazla GPU ve birden fazla GPU ve CPU için gerçeklenmesi . . . 32

4 Bulgular **33**
4.1 Test Ortamı . . . 33
4.2 Sentetik Matrisler Üzerinde Bulgular . . . 34
4.2.1 Tam Permanent Değerlerinin Hesaplanması . . . 34
4.2.1.1 Yogun Matrisler Üzerinde Yapılan Deneyler . . . 34
4.2.1.2 Seyrek Matrisler Üzerinde Yapılan Deneyler . . . 36
4.3 Single vs. Double Precision . . . 42
4.3.1 Permanent Yakınsama Üzerinde Yapılan Deneyler . . . 43
4.3.1.1 Yoğun Matrisler Üzerinde Yapılan Deneyler . . . 43
4.3.1.2 Seyrek Matrisler Üzerinde Yapılan Deneyler . . . 44
4.3.1.3 Permanent Yakınsama Algoritmalarının İsabeti Üzerine Deneyler 48

5 Tartışma ve Sonuç **50**

Şekil Listesi

1 8 bitlik bir `Gray kod` dizisinde sıra numarasına bağlı olarak değeri değişen bitler 8
2 **x** dizisinin *paylaşımlı hafıza*ya taşınması . . . 11
3 **x** üzerinde *hafıza birleştirme* . . . 11
4 (a) 5×5 boyutunda bir binary matrisin CRS (solda) ile CCS (sağda) gösterimleri . . . 13
5 (a) çıkan (ya da işlenen) satırların takibi için kullanılan dizi (b) indislerin ikili tabanda gösterimleri . . . 30
6 Yoğunluğu 0.60 olan farklı boyutta matrislerin farklı algoritmalarla permanent değeri hesaplama süreleri (saniye) . . . 36
7 `SpaRyser-SxC-Sm` algoritmasının `SortOrder` kullanan ve kullanmayan versiyonlarının tamsayı matrisi üzerinde permanent hesaplama süreleri (saniye) . . . 37
8 `SkipPer-SxC-Sm` algoritmasının `SkipOrder` kullanan ve kullanmayan versiyonlarının tamsayı matrisi üzerinde permanent hesaplama süreleri (saniye) . . . 38
10 Farklı algoritmaların yoğunluğu 0.20 olan farklı boyutlarda matrislerde bir GPU üzerinde permanent hesaplama süreleri (saniye) . . . 40
9 Farklı algoritmaların yoğunluğu 0.10 olan farklı boyutlarda matrislerde bir GPU üzerinde permanent hesaplama süreleri (saniye) . . . 40
11 Farklı algoritmaların yoğunluğu 0.30 olan farklı boyutlarda matrislerde bir GPU üzerinde permanent hesaplama süreleri (saniye) . . . 41
12 Farklı algoritmaların yoğunluğu 0.10 olan farklı boyutlarda matrislerde birden fazla GPU üzerinde permanent hesaplama süreleri (saniye) . . . 41
13 Farklı algoritmaların yoğunluğu 0.20 olan farklı boyutlarda matrislerde birden fazla GPU üzerinde permanent hesaplama süreleri (saniye) . . . 42
14 Farklı algoritmaların yoğunluğu 0.30 olan farklı boyutlarda matrislerde birden fazla GPU üzerinde permanent hesaplama süreleri (saniye) . . . 42
15 0.60 yoğunluğunda farklı boyut ve veri tipindeki matrislerin permanent hesaplama süreleri . . . 43
16 Permanent yakınsama algoritmalarının 40 x 40 boyutunda yoğun matrisler üzerinde 1 milyon deney sayısı için çalışma süreleri (saniye) . . . 44
17 `Rasmussen` türevi algoritmaların 40 x 40 boyutunda 0.10 yoğunluğunda matrisler üzerinde 1 milyon deney sayısı için permanent yakınsama süreleri (saniye) . . . 46
18 `Rasmussen` türevi algoritmaların 40 x 40 boyutunda 0.20 yoğunluğunda matrisler üzerinde 1 milyon deney sayısı için permanent yakınsama süreleri (saniye) . . . 46
19 `Scaling` türevi algoritmaların 40 x 40 boyutunda, yoğunluğu 0.10 olan matrisler üzerinde, `SInterval` $= 5$ değeriyle çalışma süreleri (saniye) . . . 47
20 `Scaling` türevi algoritmaların 40 x 40 boyutunda, yoğunluğu 0.20 olan matrisler üzerinde, `SInterval` $= 5$ değeriyle çalışma süreleri (saniye) . . . 47
21 `Scaling` türevi algoritmaların 40 x 40 boyutunda, yoğunluğu 0.10 olan matrisler üzerinde, `SInterval` $= 1$ değeriyle çalışma süreleri (saniye) . . . 48
22 `Scaling` türevi algoritmaların 40 x 40 boyutunda, yoğunluğu 0.20 olan matrisler üzerinde, `SInterval` $= 1$ değeriyle çalışma süreleri (saniye) . . . 48

23 Yoğunluğu 0.20 olan farklı boyuttaki matrisler üzerinde yakınsama algoritmalarının isabetlilikleri 49
24 Yoğunluğu 0.30 olan farklı boyuttaki matrisler üzerinde yakınsama algoritmalarının isabetlilikleri 49
25 Yoğunluğu 0.40 olan farklı boyuttaki matrisler üzerinde yakınsama algoritmalarının sonuç kalitesi 50

Tablo Listesi

1 Proje başvurusunda önerilen iş paketleri, proje raporunda bulundukları kısımlar ve tamamlanma oranları . . . 3

2 40 x 40 boyutunda farklı yoğunluklardaki matrislerin tek GPU ve CPU $(\tau = 16)$ üzerinde çalışma süreleri (saniye) . . . 35

3 40 x 40 boyutunda farklı yoğunluklardaki matrislerin birden çok GPU ve GPU+CPU (hibrit) üzerinde çalışma süreleri (saniye) . . . 35

4 `SpaRyser` algoritmasının `SortOrder` kullanan ve kullanmayan versiyonlarının 40 x 40 boyutunda tamsayı matrisi üzerinde permanent hesaplama süreleri (saniye) 36

5 `Skipper` algoritmasının `SkipOrder` kullanan ve kullanmayan versiyonlarının 40 x 40 boyutunda tamsayı matrisi üzerinde permanent hesaplama süreleri (saniye) . 37

6 40 x 40 boyutunda farklı yoğunluklardaki matrislerin CPU üzerinde permanent hesaplama süreleri (saniye) . . . 39

7 40 x 40 boyutunda farklı yoğunluklardaki matrislerin GPU üzerinde permanent hesaplama süreleri (saniye) . . . 39

8 40 x 40 boyutunda farklı yoğunluklardaki matrislerin birden fazla GPU üzerinde permanent hesaplama süreleri (saniye) . . . 39

9 Farklı algoritmaların 40 x 40 boyutunda farklı yoğunluktaki ve farklı veri tiplerindeki matrislerde permanent hesaplama süreleri (saniye) . . . 43

10 `Rasmussen` türevi algoritmaların 40 x 40 boyutunda yoğun matrisler üzerinde 1 milyon deney sayısı için permanent yakınsama süreleri (saniye) . . . 44

11 `Rasmussen` türevi algoritmaların 40 x 40 boyutunda yoğun matrisler üzerinde 1 milyon deney sayısı için permanent yakınsama süreleri (saniye) . . . 44

12 `Rasmussen` türevi algoritmaların 40 x 40 boyutunda seyrek matrisler üzerinde 1 milyon deney sayısı için permanent yakınsama süreleri (saniye) . . . 45

13 `Scaling` türevi algoritmaların 40 x 40 boyutunda yoğun matrisler üzerinde, `SInterval` $= 5$ değeriyle 1 milyon deney sayısı için permanent yakınsama süreleri (saniye) 45

14 `Scaling` türevi algoritmaların 40 x 40 boyutunda yoğun matrisler üzerinde, `SInterval` $= 1$ değeriyle 1 milyon deney sayısı için permanent yakınsama süreleri (saniye) 45

1 Giriş

Permanent, determinant gibi matris karakteristiklerini anlamaya yarayan önemli bir sayısal değerdir ve pratikte birçok uygulaması vardır. Literatürde matris permanentlerini araç olarak kullanan, Matematik, Bilgisayar Bilimleri, İstatistik ve Fizik gibi farklı alanlardan pek çok uygulama bulunmaktadır. Permanentler, Fibonacci ve Lucas sayıları (Kılıç ve Tasçı, 2007) ve sıralama istatistikleri (Balakrishnan, 2007) ile birebir ilişkilidir. Bunun yanında permanentler kuantum hesaplama için benzetim yöntemlerinde (Aaronson ve Arkhipov, 2011; Wu vd., 2018) ve biyoenformatik uygulamalarında (Narahara vd., 2013) kullanılmaktadır. Ayrıca oyun teorisindeki kusursuz bağımsız ayrık kümelerin sayısı ile Gale oyunlarındaki *Nash Dengesi* arasında önemli bir ilişki bulunmaktadır (Merschen, 2011).

Permanentin farklı bir uygulaması da çizge teorisi alanından gelmektedir. Çizgeler (*graphs*) ve seyrek matrisler (*sparse matrices*) yapısal olarak birbirlerine benzediğinden, aynı verinin farklı gösterimi için kullanılabilirler. 0–1 değerlerinden oluşan simetrik bir matrisin permanent değeri, ona karşılık gelen iki parçalı çizge (*bipartite graph*) üzerindeki mükemmel eşlemelerin (*perfect matching*) sayısına eşittir. Bu da çizge noktaları arasındaki bağlantıların yapısı hakkında önemli bir bilgidir. Benzer bir ilişki olarak yönsüz, sıradan çizgeler üzerindeki tüm noktaları kapsayan ayrık döngü kümelerinin sayısı ile, çizgenin eşlenik matrisinin permanent değerinin eşitliği gösterilebilir.

1.1 Permanent ve Eşleme

Verilen bir $n \times n$ **A** matrisinin elemanları $1 \leq i, j \leq n$ için $a_{i,j}$ ise, bu matrisin permanent değeri

$$\texttt{perm}(\mathbf{A}) = \sum_{\sigma \in \mathcal{P}} \prod_{i=1}^{n} a_{i,\sigma(i)} \tag{1}$$

eşitliğiyle hesaplanabilir. Burada $\mathcal{P}$, {1,2, …n } kadar tüm sayıların permütasyonudur. Fakat bu tür bir algoritma tüm permütasyonların üzerinden geçmek zorunda kalacağından $\mathcal{O}(nn!)$ karmaşıklığına sahip olacaktır. Ryser (1963) ekleme ve dışlama (*inclusion-exclusion)* prensipini kullanarak bu karmaşıklığı $\mathcal{O}(2^{n-1}n^2)$'ye düşürmüştür. Ryser'ın algoritması, (1) eşitliğinin yapısını aşağıdaki şekilde değiştirmektedir:

$$\texttt{perm}(\mathbf{A}) = -1^n \sum_{S \subseteq \{1,2,\cdots,n\}} (-1)^{|S|} \prod_{i=1}^{n} \sum_{j \in S} a_{i,j} \tag{2}$$

Bu algoritmayı daha da iyileştirmek için Nijenhuis ve Wilf (1978) işlemde oluşan S kümelerini `Gray kod` sırası ile işleyerek algoritmanın hesaplama karmaşıklığını $\mathcal{O}(2^{n-1}n)$'e düşürmüşlerdir. Bu algoritma genel, yoğun (*dense)* veya bütün değerleri sıfırdan farklı, tam dolu (*full)* matrisler için hala literatürdeki en verimli algoritmadır. Nijenhuis ve Wilf'in önerdiği bu varyant, bu raporda `Ryser` olarak adlandırılmıştır.

Yukarıdaki ilk eşitlikte, matris dolu (bütün değerleri 0'dan farklı) olduğunda bütün olası permütasyonlar, ya da bütün S kümeleri, permanent hesabına sıfırdan farklı bir katkı yapmaktadır: bu tür matrislerde bütün permütasyonların katkısı, hiçbiri aynı satır veya sütundan gelmeyen herhangi n matris elemanı çarpıldığında çıkan değer, sıfırdan farklıdır. Bu nedenle paralel bir algoritma tasarımı, gerekli ve yeterli iş paketlerinin belli olması nedeniyle görece kolay olabilmektedir. Fakat gerçek uygulamalarda sıklıkla karşımıza çıkan seyrek (*sparse)* matrisler için permanenti hesaplamak oldukça karmaşık bir problemdir. Matrisin içerdiği dolu permütasyon (*transversal)* sayısı küçük olduğunda, (1) kullanılıp, sadece dolu permütasyonların katkıları numaralandırma (*enumeration)* yöntemiyle toplanarak permanent hızlıca hesaplanabilir. Fakat bunun anlaşılabilmesi için dolu permütasyon sayısının bilinmesi gerekmektedir. Bu da matris değerleri 0-1 olarak kabul edildiğinde oluşan matrisin permanent değerine eşit olacaktır. Dolayısıyla, bu sayıyı hesaplamanın kolay bir yolu yoktur. Eşitlik (2) kullanıldığında da paralel algoritma daha da zor olacaktır: bunun nedeni çoğu S alt kümesinin sıfır katkı sağlaması ve bu kümelerin bulunmasının daha zor bir problem olmasıdır. Bu konuda literatürdeki en güncel çalışma proje ekibinden Kaya (2019) tarafından önerilmiştir. Bu çalışmada, `Gray kod` sırasına göre bir sonraki pozitif katkı veren S kümesinin bulunması için bir yöntem önerilmiş, bu yöntemi kullanarak geliştirilen `Ryser` türevi `SpaRyser` CPU'lar üzerinde paralel hale getirilmiş ve özellikle seyrek matrislerde $1000\times$'den fazla hızlanma elde edilmiştir. Fakat literatürde, GPU üzerinde bu tür bir yaklaşımı kullanan çalışma yoktur.

Markov-Chain Monte Carlo (MCMC) türevi algoritmalar permanent yakınsama için sıklıkla kullanılmaktadır (Dufossé vd., 2020; Jerrum vd., 2004; Karmarkar vd., 1993) MCMC türevi rassal algoritmalar için en büyük zorluk, matris üzerindeki sıfır katkı sağlamayan permütasyonların oluşturduğu tekdüze bir dağılımdan rasgele bir permütasyon seçmektir. Daha önce tek bir *neredeyse* dolu permütasyon (matrise eşlenik çizgede neredeyse mükemmel bir eşleme) yaratmak için matris ölçekleme yöntemlerini (Dufossé vd., 2015, 2018) başarıyla kullanmıştık. Matris ölçekleme, bir matrisin satır ve sütunları için bir katsayı bulup matris içerisindeki değerleri ilgili katsayılar ile çarparak ölçekler. Bu sayede ölçeklenen matrisin satır ve sütun toplamları bir olur. Bu projede de bu yöntemi, kullanılan dağılımdan daha doğru ve etkin örneklemler seçmek için kullandık. Hem CPU hem de GPU üzerinde gerçeklediğimiz paralel MCMC sayesinde, çok daha büyük matrislerin permanent değerlerine daha yakın değerler, daha kısa süreler içerisinde hesaplanabilmiştir.

Yukarıda belirtildiği gibi, CPU üzerinde permanent hesaplayan paralel algoritmalar literatürde mevcuttur (Kaya, 2019). Bildiğimiz kadarıyla GPU üzerinde çalışan bir permanent hesaplama algoritması literatürde yoktur. Benzer bir şekilde, özellikle GPU üzerinde çalışan bir paralelleştirmenin, permanent yakınsama oranını ne kadar etkileyebileceği literatürde detaylı bir şekilde ölçülmemiştir. Bu projede tam ve yaklaşık permanent hesaplama için çok çekirdekli işlemcilerde ve eğer bilgisayar üzerinde varsa aynı anda birden fazla GPU üzerinde de çalışabilen, yüksek başarımlı paralel algoritmalar kullanan, MATLAB ve Python gibi diller üzerinden arayüzler ile çağırılabilen, açık kaynak **SUPerman** aracı gerçeklenmiştir ve açık kaynak bir araç

olarak `https://github.com/kamerkaya/SUPerman` adresinde kullanıcıların erişimine açılmıştır. Şu anda %50 doluluğa sahip bir 40×40 bir matris için sıralı bir gerçekleme tek çekirdek üzerinde 32497.7 saniye sürmektedir. Bu süre 16 çekirdek üzerinde 2702.92, tek bir GPU ile 259.53, heterojen bir konfigürasyona sahip dört GPU ile 96.49 saniyeye indirilmiştir. Bunun yanında gerçeklenen yakınsama algoritmaları ile benzer boyutlarda bir matris için yaklaşık 0.999971 yakınsama oranı 13.79 saniyede elde edilmiştir.

İş paketlerinin tamamlanması ile alakalı son durum ve her iş paketinde tanımlanan işin anlatıldığı rapor bölümleri aşağıdaki tabloda özetlenmiştir:

İP No	İP Adı ve Hedefleri	Raporda Bulunduğu Kısım	Başarı Ölçütü	Elde Edilen Başarı	Tamamlanma Oranı
1	Yoğun ve tam dolu matrislerin tam ve yaklaşık permanent değerlerinin hesaplanması için yüksek performanslı algoritmaların CPU, GPU ve hibrit gerçeklenmesi.		Bu iş paketi 3,4 ve 5 nolu iş paketlerinin hayata geçirilebilmesi için kritik önem taşımaktadır. İş paketi başarılı olduğu takdirde SUPerman yazılımının yoğun ve tam dolu matrisler için yüksek performanslı kod tabanı tamamlanmış olacaktır.	Bu iş paketi tamamen bitirilmiş, gerçeklenen algoritmaların performans sonuçları final raporunda paylaşılmıştır.	
1.1	Yüksekperformanslı CPU algoritmalarının gerçeklenmesi	3.1.1, 3.3.1	Yüksek performanslı algoritmaların CPU üzerinde %90 paralel verimlilikle gerçeklenmesi.	Algoritmalar gerçeklenmiş, ∼%75 paralel verimlilik sağlanmıştır.	100%
1.2	Yüksekperformanslı GPU algoritmalarının gerçeklenmesi	3.1.2, 3.1.3, 3.3.2	Yüksek performanslı algoritmaların GPU üzerinde bir CPU çekirdeğinden 100x hızlı çalışan paralel gerçeklenmesi.	GPU üzerinde bir CPU çekirdeğinden ∼120x hızlı çalışan algoritmalar geliştirilmiştir.	100%
1.3	Yüksek performanslı algoritmaların hibrit hale getirilmesi	3.1.4.1	Yüksek performanslı algoritmaların hem CPU hem de birden fazla GPU üzerinde paralel gerçeklenmesi. Eklenen her bir işlem birimi ile elde edilen işlem gücünün %90 verimlilikle kullanılabilmesi.	Birden fazla GPU üzerinde çalışan ve hibrit algoritmalar geliştirilmiştir.	100%
2	Seyrek matrislerin tam ve yaklaşık permanent değerlerinin hesaplanması için yüksek performanslı algoritmaların CPU, GPU ve hibrit gerçeklenmesi.		Bu iş paketi 3,4 ve 5 nolu iş paketlerinin hayata geçirilebilmesi için kritik önem taşımaktadır. İş paketi başarılı olduğu takdirde SUPerman yazılımının seyrek matrisler için yüksek performanslı kod tabanı tamamlanmış olacaktır.	Bu iş paketi tamamen bitirilmiş, gerçeklenen algoritmaların performans sonuçları final raporunda paylaşılmıştır.	
2.1	Yüksek performanslı CPU algoritmalarının gerçeklenmesi	3.2.1, 3.3.2	Yüksek performanslı algoritmaların CPU üzerinde %90 paralel verimlilikle gerçeklenmesi.	Algoritmalar gerçeklenmiş, ∼%75 paralel verimlilik sağlanmıştır.	100%
2.2	Yüksek performanslı GPU algoritmalarının gerçeklenmesi	3.2.2, 3.2.3	Yüksek performanslı algoritmaların GPU üzerinde bir CPU çekirdeğinden 100x hızlı çalışan paralel gerçeklenmesi.	GPU üzerinde bir CPU çekirdeğinden ∼40x hızlı çalışan algoritmalar geliştirilmiştir.	100%
2.3	Yüksek performanslı algoritmaların hibrit hale getirilmesi	3.2.3.1	Yüksek performanslı algoritmaların hem CPU hem de birden fazla GPU üzerinde paralel gerçeklenmesi. Eklenen her bir işlem birimi ile elde edilen işlem gücünün %90 verimlilikle kullanılabilmesi.	Birden fazla GPU üzerinde çalışan ve hibrit algoritmalar geliştirilmiştir.	100%
3	SUPerman yazılımının açık kaynak hale getirilmesi	1.1	SUPerman yazılımının dökümantasyonu ile birlikte GitHub'a eklenmesi. Projenin bilimsel ve akademik çıktılarının, diğer araştırmacılar tarafından kullanılabilmesi için önemlidir.	SuPerman yazılımı dökümantasyonu ile birlikte GitHub'a eklenmiş, adresi final raporunda paylaşılmıştır.	100%
4	MATLAB arayüzünün gerçeklenmesi	Kaynak kodu	SUPerman toolbox'ının MATLAB'a eklenmesi	MATLAB üzerinden SUPerman fonksiyonlarını çağırmayı mümkün kılan arayüz gerçeklenmiştir.	100%
5	Python arayüzünün gerçeklenemesi	Kaynak kodu	SUPerman package'ının Python pip'e eklenmesi	Python üzerinden SUPerman fonksiyonlarını çağırmayı mümkün kılan arayüz gerçeklenmiştir.	100%

Tablo 1: Proje başvurusunda önerilen iş paketleri, proje raporunda bulundukları kısımlar ve tamamlanma oranları

2 Literatür Özeti

Matris permanentinin tam değerini hesaplama #P-tam (*#P-complete*) bir problemdir (Valiant, 1979). Bu nedenle problemin polinom zamanda (*polynomial time*) çalışan bir algoritması bulunmamaktadır. Literatürdeki en hızlı, $n \times n$ bir matrisin permanent değerini tam olarak hesaplayan `Ryser` algoritmasının karmaşıklığı $\mathcal{O}(2^{n-1}n)$'dır. Bu da boyutu $n \geq 40$ olan matrislerde bile hesaplamayı oldukça yavaşlatmaktadır. Literatürde tam permanent değerini paralel bir şekilde tek bilgisayarda ya da süperbilgisayarlar üzerinde hesaplama üzerine çeşitli çalışmalar bulunmaktadır (Yue vd., 2013; Wu vd., 2018; Björklund vd., 2019).

Seyrek matrisler için permanent hesaplanması konusunda az sayıda çalışma vardır: Mittal ve Al-Kurdi (2001), CRS ve CCS yapılarının verimliliğinden yararlanarak tüm permütasyonları numaralandıran bir algoritma önermiştir. Daha önce de bahsedildiği gibi, bu algoritma sadece çok seyrek matrisler için etkilidir. Dolu permütasyonların sayısı büyük olduğunda, algoritmaların çalışma süresi önemli ölçüde artmaktadır.

Forbert ve Marx (2003) seyrek matrisler için bir permanent yaklaşım algoritması önermiştir. Aynı çalışmada, permanent için aşağıdaki eşitlik kanıtlanmıştır:

$$\texttt{perm}\left(\begin{bmatrix} a & b & \mathbf{c} \\ \mathbf{d} & \mathbf{e} & \mathbf{A}' \end{bmatrix}\right) = \texttt{perm}\left(\begin{bmatrix} 0 & 0 & \mathbf{c} \\ \mathbf{d} & \mathbf{e} & \mathbf{A}' \end{bmatrix}\right) + \texttt{perm}\left(\begin{bmatrix} a\mathbf{e} + b\mathbf{d} & \mathbf{A}' \end{bmatrix}\right). \tag{3}$$

Eşitliğin sol tarafındaki $n \times n$ matrisi için, a ve b birer eleman, $\mathbf{c}$ ise $(n-2)$ boyutlu bir satır vektörüdür. Benzer şekilde $\mathbf{d}$ ve $\mathbf{e}$ $(n-1)$ boyutlu sütun vektörleri ve $\mathbf{A}'$ $(n-1) \times (n-2)$ boyutlarında bir matristir. Bu yöntem, Liang vd. (2006) tarafından da kullanılmış ve problemi belli bir büyüklüğe ulaşana kadar alt problemlere indirgeyen paralel çalışabilecek bir algoritma önerilmiştir. Bu projede de, birden fazla GPU üzerinde bu yöntemden de faydalanılacak ve başarımı test edilecektir. Unutulmamalıdır ki, bu yöntem genel bir yöntem değildir: eğer matristeki bütün satırlar ya da sütunlar fazla sayıda eleman içeriyorsa bu tür bir indirgeme yöntemi verimli olmayacak, aksine işlemi çok daha pahalı hale getirecektir. Dolayısıyla permanent değerlerini tam olarak hesaplayan paralel bir algoritma için genel bir yönteme ihtiyaç vardır.

Permanent, hesaplaması zor bir değerdir. Bu sebeple permanent yakınsama algoritmaları üzerinde oldukça fazla sayıda makale bulunmaktadır. İlk olarak Jerrum vd. (2004), içerisinde negatif sayı bulunmayan matrisler için $(1+\varepsilon)$ hata ile çalışan, $\tilde{O}(n^{10})$ karmaşıklığa sahip bir yakınsama algoritması geliştirmiştir. Bu algoritmanın gerçeklenmesi en zor kısımlarından bir tanesi, matrisin eşlenik iki parçalı çizgesi üzerindeki mükemmel eşlemelerin tekdüze bir dağılımının oluşturulup bu dağılımdan rassal bir şekilde bir eşleme seçilmesi problemidir. Jerrum vd. (2004) bu sorun için *neredeyse* tam tekdüze bir dağılım veren MCMC (*Markov Chain Monte Carlo*) yöntemini kullanmıştır. Algoritma iki parçalı çizgeler için tasarlandığından genel çizgeler için kullanılması oldukça zordur. Štefankovič vd. (2017) bu yöntemin zorluklarından bahsetmiş ve iki parçalı olmayan bazı özel çizge sınıfları için mükemmel eşlemelerinin sayılarının nasıl tahmin

edileceğini gösteren bir çalışma yapmıştır. Gurvits ve Samorodnitsky (2014) ve Linial vd. (2000) matris ölçekleme (*scaling*) yöntemine dayanan, belirlenimci fakat üssel (2^n ve e^n) garanti sunabilen yaklaşımlar önermişlerdir. Matris ölçekleme yaklaşımı bu projede de kullanılmıştır. Son yıllarda da artı yarıbelgili (*positive semidefinite*) (Anari vd., 2017) ve içeriği belli bir dağılımdan gelen rassal matrisler (Eldar ve Mehraban, 2018) için permanent yaklaşım yöntemleri önerilmeye devam edilmektedir.

Bu projede hem CPU (*Central Processing Unit*), hem de GPU (*Graphics Processing Unit*) üzerinde aynı anda çalışabilen paralel algoritmalar geliştirilmiştir ve permanent değerlerinin hem tam, hem de yaklaşık değerlerinin hesaplanması üzerine çalışılmıştır. Bilindiği kadarıyla, permanent hesaplaması ve yakınsaması üzerine özellikle GPU'lar üzerinde literatürde bir çalışma bulunmamaktadır.

3 Gereç ve Yöntem

3.1 Yoğun Matrislerin Permanent Değerinin Hesaplanması

Algoritma 1: `Ryser`

Girdi: **A** ($n \times n$ matris)

Çıktı: `perm`(**A**)

1: $rowPerm \leftarrow [.]$
2: **for** $i = 1$ to n **do**
3: $\quad sum = 0$
4: $\quad$ **for** $j = 1$ to n **do**
5: $\quad\quad sum \leftarrow sum + a_{i,j}$
6: $\quad$ **x** $\leftarrow a_{i,n} - \frac{sum}{2}$
7: $p \leftarrow \prod_{i=1}^{n}$ **x**$[i]$
8: **for** $g = 1$ to $2^{n-1} - 1$ **do**
9: $\quad j \leftarrow \log_2($**Gray**$_g \oplus$ **Gray**$_{g-1})$
10: $\quad s \leftarrow 2\times$**Gray**$_g[j] - 1$
11: $\quad prod \leftarrow 1$
12: $\quad$ **for** $i = 1$ to n **do**
13: $\quad\quad$ **x**$[i] \leftarrow$ **x**$[i] + (s \times a_{i,j})$
14: $\quad\quad prod \leftarrow prod \times$ **x**$[i]$
15: $\quad p \leftarrow p + ((-1)^g \times prod)$
16: **return** $p \times (4 \times (n \bmod 2) - 2)$

Algoritma 1'de yoğun matrisler için oldukça verimli bir şekilde çalışan, Ryser (1963)'ın önerdiği ve Nijenhuis ve Wilf (1978) tarafından iyileştirilen `Ryser` algoritması verilmiştir. Algoritma'da $\oplus$'ın XOR operatörünü ve $\gg$ operatörü bitleri sağa kaydırmayı temsil etmektedir. Bu durumda

$$\textbf{Gray}_g = g \oplus (g \gg 1)$$

olmaktadır. **Gray**$_g[j]$ ise algoritmada **Gray**$_g$'nin j'inci bitini göstermektedir. Burada $j = 1$'in en az önemli (*least significant*) biti gösterdiği kabul edilmiştir. `Gray kod` içerisindeki bir bit, matristeki ilgili sütunun hesaplamaya katılıp katılmadığını belirtmektedir. Sütunların permanent üzerindeki etkisini tutmak için bir **x** vektörü kullanılmaktadır. Bu vektörün i'ninci elemanı, matrisin i'ninci satırının permanente etkisini içermektedir. Olası her g için algoritma **Gray**$_g$'de **Gray**$_{g-1}$'e göre değişen biti bulup, **x** vektöründe bu bite denk gelen değişiklikleri yapmaktadır. Son aşamada, vektörün elemanlarının çarpımı o ana kadar hesaplanmış olan permanent p'ye eklenmektedir.

3.1.1 CPU Üzerinde Yoğun Matrislerin Permanent Değerinin Hesaplanması

`Ryser` algoritmasında, $O(2^{n-1})$ karmaşıklığa sahip **for** döngüsü, yineleme uzayını parçalara bölünmek suretiyle paralelleştirilerek işlenebilir. Bu şekilde bir paralelleştirme uygulandığında her bir parça, kendi yineleme uzayının başlangıçtaki **x** değerlerini hesaplayabilmek için bir önceki parçanın `Gray kod` değerine ihtiyaç duyacaktır. Bu değer, her bir ipliğin başlangıç indeksi *start* için *start* $\oplus 1$ olacaktır. Başlangıç yineleme noktası *start* değeri elde edildikten sonra **x** değerleri, matrisin `Gray kod` değerinde dahil edilen, ilgili bitleri 1 olan sütunları kullanılarak hesaplanabilir. Bu değerler hesaplandıktan sonra algoritmanın geri kalanı `Ryser` ile aynı mantıkta çalışır.

3.1.2 GPU Üzerinde Yoğun Matrislerin Permanent Değerinin Hesaplanması

CPU üzerinde çalışan algoritmaların kısıtlı belleğe ve hesaplama sürecinde ana bellekten veri aktarımına ihtiyaç duyabilecek GPU'lar üzerinde verimli bir şekilde çalışmak üzere uyarlanması için GPU ile ana bellek arasında ve GPU belleği üzerinde gerçekleştirilecek veri erişimlerinin dikkatlice düzenlenmesi gerekmektedir. Bu kapsamda, yapılan GPU gerçekleştirmelerinde farklı veri erişimi düzenleri için farklı versiyonlar geliştirilmiş ve performansları test edilmiştir. Permanent hesaplama probleminde girdi verisi görece küçük sayılabileceği için *örneğin 40 x 40 bir matris için 1600 değer* algoritmaların girdisi olan matrisler hesaplamanın başında ana bellekten GPU belleğine aktarılıp hesaplama süresince burada tutulmuştur. Bunun yanısıra GPU üzerinde ipliklerin sahip olacağı **x** değerleri de ana bellekte boş olarak yaratılıp GPU belleğine kopyalanmıştır. GPU üzerinde çalışan permanent hesaplama algoritmasının çıktısı, GPU üzerinde çalışan iplik sayısı boyutunda bir dizide tutulmuş, bu diziye her bir GPU ipliği kendi sorumlu olduğu parçanın toplam permanent değerine olan katkısını yazmış, daha sonra bu dizi ana belleğe kopyalanıp içindeki değerler toplanarak matrisin permanent değeri elde edilmiştir.

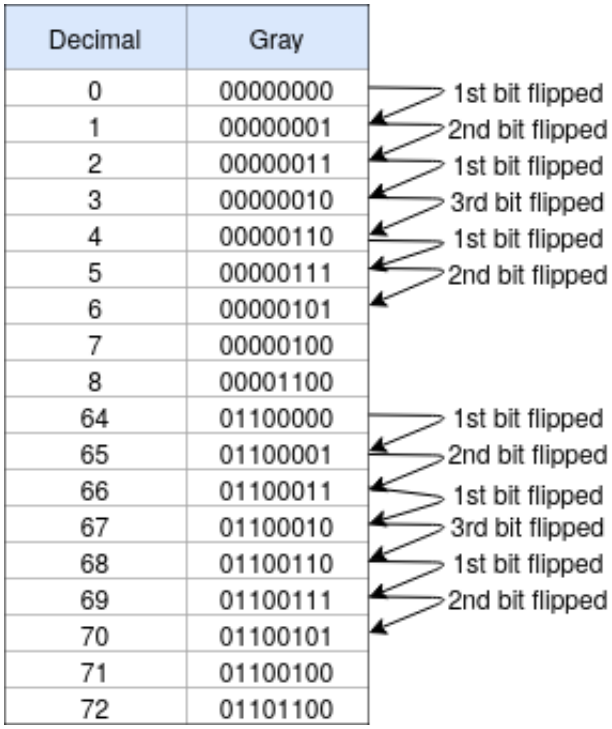
Algoritma 2: `ParallelRyser`

Girdi: **A** ($n \times n$ matris)

Çıktı: `perm`(**A**)

for $i = 1 \ldots N$ **do**
  $rowSum \leftarrow 0$
  for $j = 1 \ldots N$ **do**
    $rowSum \leftarrow rowSum + matrix[i, j]$
  $x[i] \leftarrow mat[i, n] - rowSum/2$
  $p \leftarrow p * x[i]$
$matrix \leftarrow transpose\ of\ the\ matrix$
for $each\ iplik$ **do**
  $myP \leftarrow 0$
  $myStart \leftarrow start + iplikID * chunkSize$
  $myEnd \leftarrow min(start + (iplikID + 1) * chunkSize, end)$
  $calculate\ x\ vector\ using\ the\ previous\ gray\ code$
  for $g = myStart \ldots myEnd - 1$ **do**
    $j \leftarrow \log_2 (Gray_g \oplus Gray_{g-1})$
    $s \leftarrow 2 * Gray_g[j] - 1$
    $prod \leftarrow 1$
    for $i = 1 \ldots N$ **do**
      $x[i] \leftarrow x[i] + (s * matrix[j, i])$
      $prod \leftarrow prod * x[i]$
    $myP \leftarrow p + (-1)^g * prod$
  $atomicAdd(p, myP)$
return $p * (4 * (n) mod 2 - 2)$

GPU üzerinde veriye hızlı erişmek ve manipüle etmek kritik önem taşımaktadır. GPU üzerinde bulunan warp adı verilen 32'li işlemci grupları SIMT, *Single Instruction Multiple Threads*, paradigmasıyla çalışmaktadır. Bu paradigmada bir warp'ın içinde bulunan 32 işlemci aynı anda aynı komutları çalıştırmaktadır. Bu sebepten ötürü if-else koşulları GPU'ların performansı üzerinde olumsuz etkiye sahiptir. Bunun yanısıra, bir warp'ın içinde bulunan işlemciler birbirine yakın bellek lokasyonlarına eriştiklerinde daha hızlı çalışmaktadırlar. Bu raporda sunulan algoritmalarda, `Gray kod` üzerinden erişilecek bellek lokasyonu tespit edilerek bu özellikten faydalanan bir yaklaşım takip edilmiştir. `Ryser` algoritmasının her yinelemesinde **x** değerini güncellemek için kullanılacak matris sütunu `Gray kod` farkı ile hesaplanmaktadır. Bir warp'ın içindeki bütün işlemciler aynı matris sütunu üzerinde işlem yapmayı seçerse, yakın bellek lokasyonlarına veri erişimi yapacaklarından hesaplama daha hızlı gerçekleşecektir. Bu durumdan her bir işlemcinin sorumlu olduğu parça sayısı ve büyüklüğü spesifik bir şekilde seçilerek faydalanılabilir. Bir `Gray kod` dizisinde $(i + 1)$ sıralı `Gray kod` ile i sıralı `Gray kod` arasında her zaman bir bit değer değiştirmektedir. Bunun yanısıra, `Gray kod` dizisi incelendiğinde $(i + 2^k)$ sıralı `Gray kod` ile $(i + 2^k + 1)$ sıralı `Gray kod` arasında neredeyse her zaman aynı sıradaki bitin değiştiği gözlemlenmektedir. Bu durum Şekil 1'de gösterilmiştir. Bu durumda, permanent hesaplamanın



Şekil 1: 8 bitlik bir `Gray kod` dizisinde sıra numarasına bağlı olarak değeri değişen bitler

yineleme uzayı her bir GPU ipliği için 2'nin kuvvetleri büyüklüğünde parçalara ayırılırsa, bir warp içindeki iplikler neredeyse her zaman matrisin aynı sütununa erişim yapacaklardır.

3.1.2.1 `Ryser-Gx`: x dizisinin GPU ana belleğinde yaratılması

`Ryser-Gx` algoritmasında her bir iplik tarafından kullanılacak **x** değerlerinin ana bellekte yaratılıp, GPU ana belleğine kopyalanması yaklaşımı izlenmiştir. Her bir ipliğin GPU ana belleğinde kendi **x** değerine sahip olması için bu değerlerin ayrı ayrı her bir iplik için yaratılması gerekmektedir. Çünkü bu iplikler sorunlu oldukları yineleme uzayı parçası üzerinde hesaplama yaparken kendi **x** değerleri üzerinde çalışacaklardır. Bu sebepten **x** öncelikle GPU ana belleğinde `dim` $\times$ `GridDim` $\times$ `BlockDim` boyutunda yaratılmış ve her bir iplik, kendi eşsiz `iplikId` değerini kullanarak bu büyük dizinin kendilerine ait parçası üzerinde işlem yapmışlardır. Algoritma 3'te `Ryser-SxC` algoritmasının sözde kodu verilmiştir. `Ryser-Gx` algoritmasının bu algoritmadan farkı **x** değerinin GPU ana belleğinden okunuyor olması, bu sayede 5. satırdaki *initialize*'a gerek duymamasıdır. Bunun yanısıra, algoritmaya girdi olarak verilen *mat* matrisinin *devriği* olan *matT* matrisi doğrudan GPU ana belleğinden okunmakta, bu sayede de 8. satırdaki okuma işlemine ihtiyaç duyulmamaktadır.

Algoritma 3: RYSER-SXC-SM($matT, dim, \mathbf{x}, p, start, end$)

1: $\texttt{tid} \leftarrow iplikIdx.x + (blockIdx.x \times blockDim.x)$
2: $\texttt{iplikId} \leftarrow iplikIdx.x$
3: $\texttt{BlockDim} \leftarrow blockDim.x$
4: **for** $k = 1 \ldots dim$ **do**
5: $\quad myX[\texttt{BlockDim} \times k + \texttt{iplikId}] \leftarrow x[k]$
6: **for** $k = 1 \ldots (dim \times dim)/\texttt{BlockDim} + 1$ **do**
7: $\quad$ **if** $\texttt{BlockDim} \times k + \texttt{iplikId} < dim \times dim$ **then**
8: $\quad\quad sharedMatT[\texttt{BlockDim} \times k + \texttt{iplikId}] \leftarrow matT[\texttt{BlockDim} \times k + \texttt{iplikId}]$
9: $syncthreads()$
10: $chunkSize \leftarrow (end - start)/(\texttt{GridDim} \times \texttt{BlockDim}) + 1$
11: $myStart \leftarrow start + \texttt{iplikId} \times chunkSize$
12: $myEnd \leftarrow min(start + (\texttt{iplikId} + 1) \times chunkSize, end)$
13: $myP \leftarrow 0$
14: **for** $k \ldots dim - 1$ **do**
15: $\quad$ **if** $(\texttt{Gray kod}_{myStart-1} >> k)\&1$ **then**
16: $\quad\quad$ **for** $j \ldots dim$ **do**
17: $\quad\quad\quad myX[\texttt{BlockDim} \times j + \texttt{iplikId}] \leftarrow$
$\quad\quad\quad myX[\texttt{BlockDim} \times j + \texttt{iplikId}] + sharedMatT[k \times dim + j]$
18: **for** $g = myStart \ldots myEnd - 1$ **do**
19: $\quad j \leftarrow \log_2(\texttt{Gray kod}_g \oplus \texttt{Gray kod}_{g-1}) + 1$
20: $\quad s \leftarrow 2 * \texttt{Gray kod}_g[j] - 1$
21: $\quad prod \leftarrow 1$
22: $\quad$ **for** $i = 1 \ldots dim$ **do**
23: $\quad\quad myX[\texttt{BlockDim} \times i + \texttt{iplikId}] \leftarrow$
$\quad\quad myX[\texttt{BlockDim} \times i + \texttt{iplikId}] + (s \times sharedMatT[j \times dim + i])$
24: $\quad\quad prod \leftarrow prod \times myX[\texttt{BlockDim} \times i + \texttt{iplikId}]$
25: $\quad myP \leftarrow myP + (-1)^g * prod$
26: $p[\texttt{tid}] \leftarrow myP$

3.1.2.2 `Ryser-Rx`: x dizisinin GPU yazmaçlarına taşınması

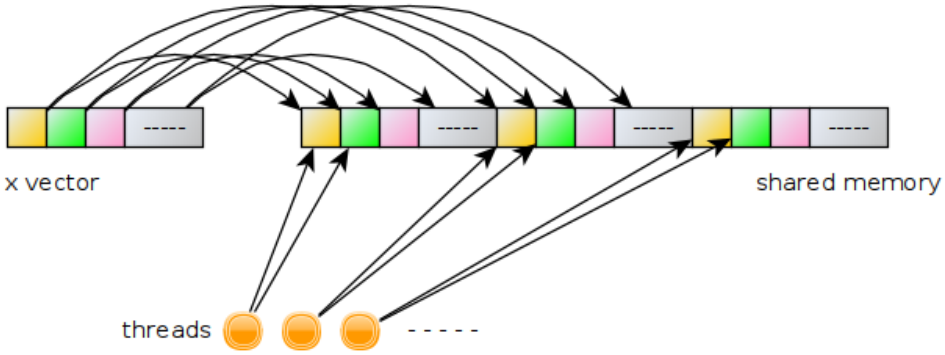
`Ryser-Rx` algoritmasında GPU ipliklerinin başlangıçta kullanacakları **x** değeri GPU yazmaçlarında yaratılmış, vektör elemanlarının değerleri ise Algoritma 3'te **x** girdisi olarak belirtilen diziden, GPU ana belleğinden kopyalanmıştır. Bu algoritmada da Algoritma 3'e benzer olarak 5. satırdaki *myX* değeri kullanılmıştır. Ancak bu algoritmada Algoritma 3'teki gibi x değeri paylaşımlı hafıza üzerinde tutulmamaktadır. Bu sebepten *myX* dizisi, girdi olarak verilen matrisler aynı boyutta olup, *myX* dizisinin i. indeksine *myX[i]* şeklinde ulaşılmaktadır. Benzer şekilde, Algoritma 3'ün tersi bir yaklaşımla matris paylaşımlı hafıza üzerinde değil, GPU ana belleği üzerinde tutulmaktadır ve Algoritma 3'teki 8. ve 9. satırlara olan ihtiyaç ortadan kalkmaktadır. Algoritmanın geri kalanı, her bir ipliğin **x** üzerinde okuma ve güncelleme yaparken, **x** değerini yaz-

maçlar üzerinden okumaları haricinde aynıdır. Register belleği, GPU üzerinde bulunan en hızlı bellek olmasına rağmen, bu versiyon GPU üzerinde bulunan SM (*Streaming Multiprocessor*) başına düşen yazmaç belleğinin kısıtlı olmasından ötürü matris boyutu büyüdükçe **x** uzunluğu da arttığından kısıtlı bir kullanıma sahiptir.

3.1.2.3 `Ryser-Sx`: x dizisinin paylaşımlı hafızada yaratılması

`Ryser-Sx` algoritmasında, GPU üzerinde bulunan paylaşımlı hafıza her bir ipliğin **x** dizisi için ayrılmış ve bu değer algoritmanın başında GPU ana belleğinden paylaşımlı hafızaya kopyalanmıştır. Her bir ipliğin kendi **x** dizisini paylaşımlı hafızada tutabilmesi için, her bir işlemci bloğu için *(block)* paylaşımlı hafıza üzerinde `dim` $\times$ `BlockDim` boyutunda bellek kullanılmıştır. Paylaşımlı hafıza üzerindeki alan ayrıldıktan sonra, her bir iplik Algoritma 3'deki 5. satıra benzer şekilde, kendi **x** dizisini GPU ana belleğinden paylaşımlı hafızada kendilerine ayrılan alana kopyalamışlardır. Burada Algoritma 3'ten ayrışan kısım, Şekil 2'de görülebileceği gibi, her bir ipliğin **x** dizisinin paylaşımlı hafızada bloklar halinde yanyana durmasıdır. Bu yüzden kendisine ait **x** dizisinin $i.$ indeksine erişmek için her bir iplik $myX[iplikIdxdim + i]$ formülünü kullanmaktadır. Girdi matrisi GPU ana belleğinde tutulduğu için, bu algoritmada Algoritma 3'teki 6. satıra ihtiyaç duyulmamakta ve *matT* doğrudan kullanılmaktadır. Algoritmanın geri kalanı Algoritma 3 ile aynıdır.

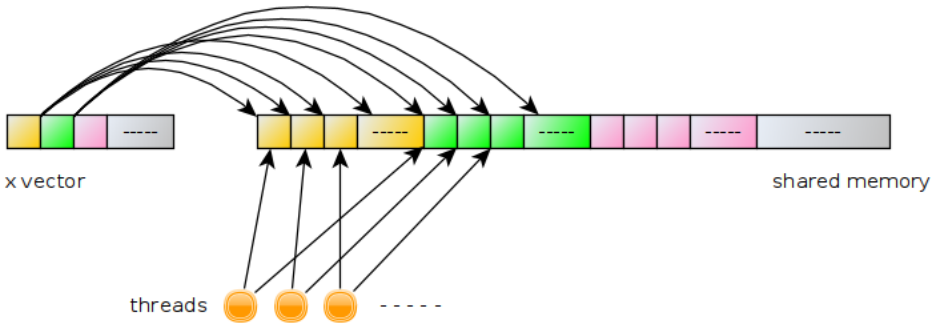
Yukarıda anlatılan yaklaşımla GPU mimarisinin bellek hiyerarşisinden faydalanılarak, **x** elemanlarına GPU ana belleğinden çok daha hızlı olan paylaşımlı hafıza üzerinden ulaşmak amaçlanmıştır. Ancak bu yaklaşım aynı zamanda GPU üzerinde iplik bloğu başına düşen paylaşımlı hafızanın 48KB boyutunda olması sebebiyle GPU doluluk (*occupancy*) oranını kısıtlamaktadır. Bu durum **x** girdi matrisi ile aynı boyuta sahip olduğu için iplik bloğu başına kullanılabilecek iplik sayısını da kısıtlamaktadır. Örneğin, **x** girdileri için `double` veri tipi kullanıldığında her bir iplik bloğu `dim` x `BlockDim` x $8\ byte$ belleğe ihtiyaç duyacaktır ve bu alanın bütün iplikler için gereken alan hesaba katıldığında, toplam paylaşımlı hafıza miktarı olan 48KB'dan küçük olması gerekmektedir. Bu durum girdi matrisinin boyutları 36-40 aralığında olduğunda, her bir iplik bloğunda 150-170 ipliğin, her bir SM'de ise sadece bir bloğun çalıştırılabileceği anlamına gelmektedir. Ancak, her bir SM (GPU mimarisinin versiyonuna bağlı olmak üzere) 2048 iplik çalıştırabilmektedir ve iplikler arasında dağıtılan işlerden oluşacak gecikmeyi gizlemek için bu miktarda iplik çalıştırmak önem arz etmektedir. Bu durum, bu yaklaşımın ölçeklenebilirliği açısından sıkıntı teşkil etmektedir. Yapılan deneylerde, **x** girdileri `float` data tipiyle de değiştirilmiş ve bu şekilde bir SM içine 305-340 iplik koyulabilmiştir.



Şekil 2: **x** dizisinin *paylaşımlı hafıza*ya taşınması

3.1.2.4 `Ryser-SxC`: x dizisinin hafıza birleştirme ile paylaşımlı hafıza üzerinde yaratılması.

`Ryser-SxC` ve `Ryser-Sx` algoritmaları, her iki algoritmanın da **x** değerlerine hızlı bir şekilde ulaşma sağlanması için **x** dizisini paylaşımlı hafıza üzerinde tutma yaklaşımı açısından benzerlik göstermektedir. Aynı zamanda iki algoritma da girdi matrisi *matT*'yi GPU ana belleğinde tutmaktadır. `Ryser-Sx` algoritmasında daha önce bahsedildiği gibi, **x** değerleri spaylaşımlı hafızada yanyana bloklar halinde tutulmaktadır. Bu durum bir warp içindeki ipliklerin daha geniş bellek alanına erişmesine sebep olmaktadır. Bir warp içindeki iplikler daha dar bir bellek alanına erişim yaptıkları zaman veriyi kopyalama, haliyle işlemi gerçekleştirme hızları artmaktadır. Bu teknik literatürde *hafıza birleştirme* olarak adlandırılır. `Ryser-SxC` algoritmasında *hafıza birleştirme* tekniği, paylaşımlı hafızada bloklar halinde yan yana duran her biri bir başka bir ipliğe ait **x** elemanlarının aynı warp'a dahil ipliklerin aynı indekse sahip elemanları yan yana duracak şekilde iç içe geçirilmesiyle uygulanmıştır. Bu dizilim Şekil 3'de görülebilir.



Şekil 3: **x** üzerinde *hafıza birleştirme*

3.1.2.5 `Ryser-SxC-Sm`: x ve girdi matrisin paylaşımlı hafızada tutulup hafıza birleştirme ile erişilmesi

`Ryser-SxC-Sm` algoritmasında, paylaşımlı hafızada **x** için ayrılan alanın yanısıra, girdi matrisi

için de alan ayrılarak, sıklıkla erişilen bellek lokasyonlarının hepsinin paylaşımlı hafızada tutulması amaçlanmıştır. Algoritma 3 bu algoritmanın sözde kodunu göstermektedir. Daha önce bahsedilen `Ryser-Sx` ve `Ryser-SxC` algoritmalarına benzer şekilde `Ryser-SxC-Sm` algoritmasında da kısıtlı büyüklüğe sahip paylaşımlı hafıza kullanımından doğan limitasyonlar mevcuttur. Bu durum, girdi matrisi de paylaşımlı hafıza üzerinde tutulduğundan `Ryser-SxC-Sm` algoritmasında daha yoğun bir şekilde kendini göstermektedir. **x** float değerler aldığı zaman, 36-40 arası matris boyutları için bir iplik bloğunda 265-305 kadar iplik çalıştırılabilmektedir. **x** double değerler aldığı zaman bu sayı 230-270 aralığına kadar düşmektedir.

3.1.3 Birden Fazla GPU Üzerinde Yoğun Matrislerin Permanent Değerinin Hesaplanması

3.1.3.1 `Ryser-SxC-Sm-MG`: `Ryser-SxC-Sm` algoritmasının birden fazla GPU üzerinde statik parçalamayla çalıştırılması

`Ryser-SxC-Sm` algoritması tek bir GPU cihazı üzerinde çalışması için geliştirilmiştir. Ancak, yineleme uzayı, GPU üzerindeki her bir iplik için parçalara bölünebileceği gibi, birden fazla GPU cihazı için de parçalara bölünüp paralel olarak çalıştırılabilir. `Ryser-SxC-Sm-MG` algoritmasında, bu yineleme uzayı, her bir GPU cihazı için eşit parçalara bölünüp her bir parça üzerinde `Ryser-SxC-Sm` algoritması çalıştırılmıştır. Hesaplamanın sonunda, her bir cihazdan elde edilen *kısmi* permanent değeri toplanarak toplam permanent değerine ulaşılmıştır.

3.1.4 CPU ve GPU Üzerinde (Hibrit) Yoğun Matrislerin Permanent Değerinin Hesaplanması

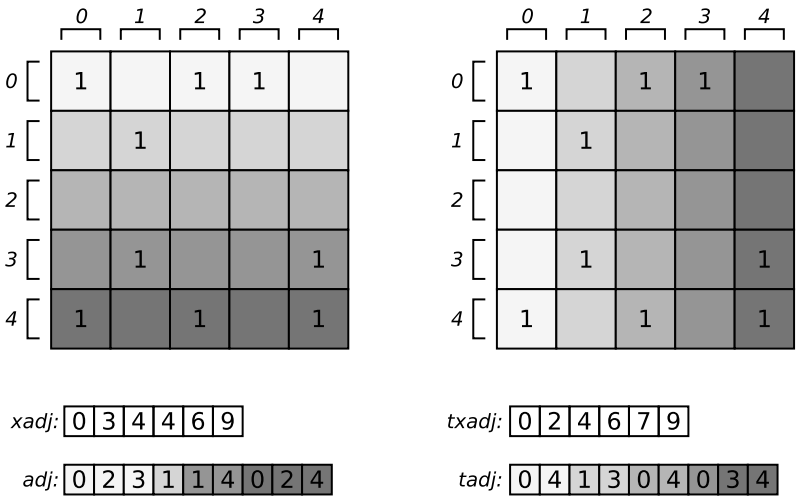
3.1.4.1 `Ryser-SxC-Sm-MG+`: `Ryser-SxC-Sm` ve `ParallelRyser` algoritmalarının GPU ve CPU üzerinde aynı anda çalıştırılması

`Ryser-SxC-Sm-MG` algoritması, yineleme uzayını GPU cihazları üzerinde bölerek çalışmaktadır. Ancak bu durumda, cihazların biri kendi yineleme uzayı üzerindeki hesaplamayı bitirdiğinde diğer cihazın bitirmesini beklemekte ve bu performansta kayba sebep olmaktadır. Bu performans sorunu, cihazlara dağıtılan yineleme uzayı parçalarının küçükltülmesiyle çözülebilir. `Ryser-SxC-Sm-MG+` algoritması bu yaklaşımı izlemektedir. Bu algoritmada, küçük ve çok sayıdaki yineleme uzayı parçalarından birini bitiren bir cihaz, diğerlerini beklemekten sıradaki parçayı alıp işlemeye başlamaktadır. Eldeki parçaların hepsi tamamlanana kadar algoritma parçaları cihazlara dağıtmaya devam etmektedir. Bu yaklaşımda cihazlara dağıtılıcak parçaların boyutlarını doğru seçmek de önemlidir. Eğer yineleme uzayı çok büyük parçalara ayrılırsa, bu durum yeniden hızlı cihazların yavaş cihazları beklemesiyle sonuçlanacaktır. Öte yandan, eğer parçalar çok küçük seçilirse bu da her bir cihazın başlangı. **x** yaratması ve sonuçları geri kopyalaması gerekmesinden ötürü ekstra bir *yaratacaktır.* Bu sebeplerden ideal boyutlarda bir parça boyutu bulmak bu algoritma için önem taşımaktadır. İdeal parça boyutu ampirik bir yaklaşımın sonucu 2^{dim-29} olarak elde edilmiştir. Bu yaklaşımın güçlü yanı cihaz olarak bahsedilen hesaplama noktalarının genel geçer olmasıdır. Bir GPU cihazı yerine, parçalar CPU'ya göndererek algoritma GPU ve CPU üzerinde aynı anda çalıştırılabilir. Bu projede hib-

rit hesaplama `Ryser-SxC-Sm-MG` ve `ParallelRyser` algoritmalarının aynı anda çalıştırılmasıyla gerçeklenmiştir.

3.2 Seyrek Matrislerin Permanent Değerinin Hesaplanması

Matrislerin seyrek olduğu durumlarda Algoritma 1'deki yaklaşım verimsiz olmaya başlamaktadır. Öncelikle, elemanların çoğu sıfıra eşit ve etkisiz olduğundan iki boyutlu bir veri yapısı kullanıldığında gereksiz hafıza erişimleri yapılmakta ve eleman değerlerinin kontrolüne ihtiyaç duyulmaktadır. Bu nedenle, (Kaya, 2019)'da gösterdiği gibi, matrisin seyrek bir veri yapısı ile tutulması ve algoritmanın bu veri yapısına adapte edilmesi elzemdir. Şekil 4'de seyrek bir matris için uygun CRS ve CCS veri yapıları verilmiştir. Bu veri yapıları matris boyutları küçük olduğundan GPU üzerinde kısıtlı ama erişim hızı yüksek ve gecikme süresi az olan paylaşımlı hafıza (*paylaşımlı hafıza*) üzerinde tutulabilmiştir.



Şekil 4: (a) 5×5 boyutunda bir binary matrisin CRS (solda) ile CCS (sağda) gösterimleri

Problemin doğası, ve algoritmaların alt iş paketlerine kolaylıkla ayrılması nedeniyle, problemin ilk bakışta çok çekirdekli işlemciler, dağıtık sistemler ve hızlandırıcılar üzerinde çözülmeye uygun olduğu düşünülebilir. Aslında, yoğun ve tam dolu matrisler için bu çıkarım bir derece doğrudur. Fakat bu tür çözümler geliştirilirken de, yüksek başarımlı hesaplama prensiplerine (örneğin; bellek hiyerarşisi, erişim sırası, zamansal ve bölgesel yerellik vb.) riayet etmek performansın eniyilenmesi için gereklidir. Bu bölümde gösterileceği gibi, seyrek matrisler üzerinde problem paralel bir çözüm geliştirilmesi için oldukça karmaşık bir yapıya sahiptir.

3.2.1 CPU Üzerinde Seyrek Matrislerin Permanent Değerinin Hesaplanması

3.2.1.1 CPU üzerinde `Skipper` algoritmasının gerçeklenmesi

Basit bir paralelleştirme ile tüm matrislerde çalışabilecek paralel bir algoritma için `Gray kod` uzayı, eldeki iplik sayısına dağıtılabilir. Bu tür bir yaklaşımla eğer τ iplik varsa, her iplik $2^{n-1}/\tau$ ardışık `Gray kod` işleyecektir. CPU'lar üzerinde bu yöntem seyrek matrisler için çok çekirdekli işlemciler üzerinde yüksek bir performans vermemiştir (Kaya, 2019). Bunun nedeni `Gray kod` uzayının bütün aralıklarının aynı sayıda katkı yapan kod içermemesidir. Bu nedenle bazı aralıklar boş kalabilmekte, bu da o aralıklara atanan ipliklerin işlemci gücünün boşa gitmesine sebep olmaktadır. Bu nedenle kod uzayını iplik sayısından çok daha fazla aralığa ayıran ve dinamik şekilde bu aralıkları ipliklere atayan görev atama algoritması kullanılmıştır.

Yukarıda bahsedilen `Gray kod` boşlukları alt iş paketlerinde de oluşabileceğinden, bu tür boşlukların atlanması çalışma zamanını azaltacak ve performansı arttıracaktır. Bunun için Kaya (2019) şu gözlemi kullanmıştır. İkili gösterimde n-bit bir $\textbf{Gray}_g$'de, $1 \leq j \leq n$ için j'inci bitin değeri

$$\textbf{Gray}_g[j] = \begin{cases} 0 & \text{eğer } g < 2^{j-1} \text{ ise} \\ \left(\lfloor \frac{(g-2^{j-1})}{2^j} \rfloor \bmod 2 \right) + 1 & \text{eğer } g \geq 2^{j-1} \text{ ise} \end{cases}$$

olacaktır. Bu gözlemde şu şekilde faydalanılabilir: hesaplama sırasında değeri sıfır olan bir $\mathbf{x}[i]$ ile karşılaşıldığında, $\mathbf{x}[i]$'yi sıfırdan farklı yapmak için $\mathbf{A}$ matrisinin sütunlarından en az birinin i'nci vektörün bu elemanına katkı yapması gerekmektedir. $next(g, j)$, şu anda çalıştırılan g'inci koddan sonra $\textbf{Gray}_g[j] \neq \textbf{Gray}_{next(g,j)}[j]$ olduğu ilk kod indisi olsun. Yukarıdaki gözleme göre,

$$next(g, j) = \begin{cases} 2^j & \text{eğer } g < 2^j \text{ ise} \\ g + 2^{j+1} - ((g - 2^j) \bmod (2^{j+1})) & \text{eğer } g \geq 2^j \text{ ise} \end{cases}$$

olacaktır. Dolayısıyla eğer $\mathbf{x}[i] = 0$ ise, $\mathbf{x}[i] \neq 0$ olacağı ilk kodu bulmak için öncelikle

$$g^{(i)} = \min \{next(g, j) : a_{i,j} \neq 0\}$$

değeri hesaplanır. Daha sonra, $\mathbf{x}$'in yineleme g'deki vektör olduğunu varsayarak, sıfırdan farklı bir katkı üreten ilk kod indisi

$$next(g) = \begin{cases} g + 1 & \text{eğer } \mathbf{x}[i] \neq 0, \text{ ve } 1 \leq i \leq n \text{ ise} \\ \max\{g^{(i)} : \mathbf{x}[i] = 0\} & \text{eğer } \mathbf{x}[i] \neq 0, \text{ ve } 1 \leq i \leq n \text{ değilse} \end{cases}$$

olarak hesaplanabilir. Bu projede de bu tür bir yaklaşım performansın arttırılması için kullanılmıştır.

Algoritma 4: `Skipper`

Girdi: $(rptrs, columns, rvals)$ - **A**'nın CRS yapısı
$(cptrs, rows, cvals)$ - **A**'nın CCS yapısı

Çıktı: `perm`(**A**)

1: **for** $i = 1$ to n **do**
2: $sum = 0$
3: **for** $ptr = rptrs[i]$ to $(rptrs[i+1] - 1)$ **do**
4: $sum \leftarrow sum + rvals[ptr]$
5: $\mathbf{x}[i] \leftarrow rvals[rptrs[i+1] - 1] - \frac{sum}{2}$
6: $p \leftarrow \prod_{i=1}^{n} \mathbf{x}[i]$
7: $g' \leftarrow 0$
8: $g \leftarrow 1$
9: **while** $g < 2^{n-1}$ **do**
10: $gr_{diff} = g \oplus g'$
11: **for** $j = 1$ to n **do**
12: **if** $gr_{diff}[j] = 1$ **then**
13: $gr_{diff}[j] \leftarrow 0$
14: $s \leftarrow 2 \times \mathbf{Gray}_g[j] - 1$
15: **for** $ptr = cptrs[j]$ to $cptrs[j+1] - 1$ **do**
16: $row \leftarrow rows[ptr]$
17: $val \leftarrow cvals[ptr]$
18: $\mathbf{x}[row] \leftarrow \mathbf{x}[row] + (s \times val)$
19: $prod \leftarrow 1$
20: **for** $i = 1$ to n **do**
21: $prod \leftarrow prod \times \mathbf{x}[i]$
22: $p \leftarrow p + ((-1)^g \times prod)$
23: $g' \leftarrow g$
24: $g \leftarrow next(g)$
25: **return** $p \times (4 \times (n \bmod 2) - 2)$

Algoritma 4'te gösterildiği gibi `Skipper` algoritmasının içinde `Gray kod` uzayının $2^{n-1} - 1$ olduğu bir döngü bulunmaktadır. $2^{n-1} - 1$ n'den çok daha büyük olduğundan, başvuruda daha önce de belirtildiği gibi permanent hesaplama işlemini dışarıdaki döngüden bölerek paralelleştirmek daha iyi sonuç verecektir. Bu şekilde her iplik farklı ve ardışık kod bloklarını işleyecektir. Herhangi bir kod grubundan işlemeye başlamak zor gibi görünse de, `Skipper`'da mevcut tekniği kullanarak iplikler herhangi bir kod grubunun başladığı yinelemeye sıçrayabilirler.

3.2.1.2 CPU üzerinde `SpaRyser` algoritmasının gerçeklenmesi

`SpaRyser` algoritması, `Ryser` algoritmasını baz aldığı için seyrek matrisler için de yoğun matrislerde olduğu gibi paralelleştirilebilir. Bu algoritmanın sözde kodu Algoritma 5'te verilmiştir. Al-

goritmanın girdileri olarak, matrisin yanısıra CCS gösterimi, matrisin boyutları ve yineleme uzayının başlangıç ve bitiş noktaları da kullanılmıştır. Matrisin kendisi algoritmada yalnız **x** değerlerinin hesaplanmasında kullanılmış, permanent değerinin hesaplanmasında kullanılmamıştır. Her bir iplik bir yinelemede yalnızca bir sütun üzerinde işlem yapıp permanente katkısını eklemek üzere **x** üzerinde güncelleme yapacağı için algoritmanın ana *for* döngüsü için yalnızca CCS gösterimini kullanılmıştır. CCS gösterimini kullanılarak matrisin yalnızca sıfır olmayan elemanlarının hesaba dahil edilmesinden sonra, eğer **x** üzerinde bir sıfır değeri varsa, bu yinelemenin toplam permanent değerine katkısı 0 olacağından yineleme atlanmıştır. **x** üzerinde bir sıfır olup olmadığını verimli bir şekilde anlayabilmek için, her bir yinelemede **x** üzerindeki sıfır sayısı tutulmuş sadece ve değişen değerlerin sıfıra dönüşmesi veya sıfırdan pozitife dönüşmesi sonucu güncellenmiştir. Sıfırların sayısı sıfır ise o yineleme için **x** toplanıp toplam permanent değerine eklenmiştir. Aynı yaklaşımın benzeri Algoritma 5 ve GPU versiyonlarında da kullanılmıştır. **x** üzerindeki bir önceki yinelemeden gelen 0 değerlerinin yanısıra, sıfırdan farklı değerlerin çarpımı da tutulmuştur. Benzer şekilde, sadece CCS gösterimi üzerinde bulunan sıfırdan farklı değerler üzerinde yineleme yapılmış ve **x** üzerindeki sıfırdan farklı değerlerin çarpımı ve sıfır sayısı güncellenmiştir. İtearasyonun sonunda, eğer **x** üzerindeki sıfırdan farklı değerlerin sayısı 0 ise sıfırdan farklı değerlerin çarpımı tutulmuş ve **x** üzerinde tekrar yineleme yapılarak yinelemenin toplam permanent değerine katkısı hesaplanmamıştır. Bu iyileştirmelerin yanısıra, `SortOrder` yaklaşımı da en sık erişilen sütunları eleman olarak en az 0 içeren sütunlardan seçmek için kullanılmıştır. Bunun sebebi `Gray kod`'da ilk bitlerin değeri sık değişirken son bitlerin değerinin sık değişmemesidir. Bu durum ilk bitlere denk gelen sütunlara daha çok erişilmesine yol açmaktadır. Bu sebepten `SortOrder` işlemi yapıldıktan sonra, ilk sütunlar en az 0 içeren sütunlar olacaktır. Paralelleştirme için, $O(2^{n-1})$ içeren ana döngü veri paralel olacak şekilde paralelleştirilmiştir.

Yoğun matrislerin permanent değerinin hesaplanması için de bu bölümde bahsedilen yaklaşımlar kullanılabilir. Bu durumda, rapordaa 3. kısımdan farklı olarak girdi matrisini ana bellekten GPU belleğine kopyalamak yerine CCS gösterimi kullanılmış ve Şekil 4'de gösterilen diziler GPU belleğine kopyalanmıştır. Bu diziler $2 \times nnz + dim$ kadar yer kaplamakta olup, girdi matrisi en çok 0.5 yoğunlukta olduğu zaman matrisin kendisinden daha az yer kaplamaktadır. Ayrıca `SortOrder` yaklaşımı da yoğun matrisler üzerinde permanent hesaplamanın hızlandırılması için kullanılabilir.

Algoritma 5: `ParallelSpaRyser` ($mat, cptrs, rows, cvals, dim, start, end$)

1: $p \leftarrow 1$
2: **for** $i = 1 \ldots N$ **do**
3: $\quad rowSum \leftarrow 0$
4: $\quad$ **for** $j = 1 \ldots N$ **do**
5: $\quad\quad rowSum \leftarrow rowSum + mat[i, j]$
6: $\quad x[i] \leftarrow mat[i, n] - rowSum/2$
7: $\quad p \leftarrow p \times x[i]$
8: **for** *each iplik* **do**
9: $\quad myX \leftarrow x$
10: $\quad myP \leftarrow 0$
11: $\quad myStart \leftarrow start + \texttt{iplikId} \times chunkSize$
12: $\quad myEnd \leftarrow min(start + (\texttt{iplikId} + 1) \times chunkSize, end)$
13: $\quad$ calculate myX using $\texttt{Gray kod}_{myStart-1}$
14: $\quad prod \leftarrow \prod_{n=1}^{\texttt{dim}} myX[i]$ for $myX[i] \neq 0$
15: $\quad zeroNum \leftarrow \sum_{n=1}^{length(myX)} myX[i]$ for $myX[i] = 0$
16: $\quad$ **for** $g = myStart \ldots myEnd - 1$ **do**
17: $\quad\quad j \leftarrow \log_2 (\texttt{Gray kod}_g \oplus \texttt{Gray kod}_{g-1}) + 1$
18: $\quad\quad s \leftarrow 2 * \texttt{Gray kod}_g[j] - 1$
19: $\quad\quad$ **for** $i = cptrs[j] \ldots cptrs[j + 1] - 1$ **do**
20: $\quad\quad\quad$ **if** $myX[rows[i]] = 0$ **then**
21: $\quad\quad\quad\quad zeroNum \leftarrow zeroNum - 1$
22: $\quad\quad\quad\quad myX[rows[i]] \leftarrow myX[rows[i]] + s \times cvals[i]$
23: $\quad\quad\quad\quad prod \leftarrow prod \times myX[rows[i]]$
24: $\quad\quad\quad$ **else**
25: $\quad\quad\quad\quad prod \leftarrow prod/myX[rows[i]]$
26: $\quad\quad\quad\quad myX[rows[i]] \leftarrow myX[rows[i]] + s \times cvals[i]$
27: $\quad\quad\quad\quad$ **if** $myX[rows[i]] = 0$ **then**
28: $\quad\quad\quad\quad\quad zeroNum \leftarrow zeroNum + 1$
29: $\quad\quad\quad\quad$ **else**
30: $\quad\quad\quad\quad\quad prod \leftarrow prod \times myX[rows[i]]$
31: $\quad\quad$ **if** $zeroNum = 0$ **then**
32: $\quad\quad\quad myP \leftarrow myP + (-1)^g * prod$
33: $\quad atomicAdd(p, myP)$
34: **return** $p \times (4 \times (n \bmod 2) - 2)$

3.2.2 GPU Üzerinde Seyrek Matrislerin Permanent Değerinin Hesaplanması

3.2.2.1 `ParallelSkipper`: Skipper algoritmasının GPU üzerinde gerçeklenmesi

Daha önce bahsedildiği gibi, `Skipper` algoritması yineleme uzayını veri paralel bir yapıda ipliklere dağıtmak suretiyle paralelleştirilebilir. GPU üzerinde çalışan `Skipper` algoritmasının sözde kodu Algoritma 6'da verilmiştir. Girdi matrisi *mat*, yalnızca algoritmanın en başında **x** değerlerini hesaplamak için kullanılmıştır. Algoritmanın 2. satırında **x** için başlangıç değerleri hesaplanıp eşitlenmiştir. Bundan sonra, her bir iplik yineleme uzayı içerisinden kendisine düşen parçayı, *dynamic scheduling* ile çok sayıda küçük parçanın arasından seçerek bu parça için hesaplamaya başlamaktadır. `ParallelSkipper` algoritması, `Skipper` ile aynı mantığı takip etmektedir. Yalnızca, matris ve **x** üzerinde olan erişimler değiştirilmiştir.

Algoritma 6: `ParallelSkipper` $(mat, rptrs, cols, cptrs, rows, cvals, dim, start, end)$

```
1:  p ← 1
2:  for i = 1 ... N do
3:      rowSum ← 0
4:      for j = 1 ... N do
5:          rowSum ← rowSum + mat[i, j]
6:      x[i] ← mat[i, n] − rowSum/2
7:      p ← p × x[i]
8:  for each iplik do
9:      myX ← x
10:     myP ← 0
11:     myStart ← start + iplikId × chunkSize
12:     myEnd ← min(start + (iplikId + 1) × chunkSize, end)
13:     prevG ← 0
14:     g ← myStart
15:     while g < myEnd do
16:         grayDiff ← Gray kod_g ⊕ Gray kod_prevG
17:         for each grayDiff[j] = 1 do
18:             if Gray kod_g[j] = 1 then
19:                 for i = cptrs[j] ... cptrs[j + 1] − 1 do
20:                     myX[rows[i]] ← myX[rows[i]] + cvals[rows[i]]
21:             else
22:                 for i = cptrs[j] ... cptrs[j + 1] − 1 do
23:                     myX[rows[i]] ← myX[rows[i]] − cvals[rows[i]]
24:         prod ← 1
25:         for i = 1 ... dim do
26:             prod ← prod × myX[i]
27:         myP ← myP + (−1)^g × prod
28:         prevG ← g
29:         g ← next(g)
30:     atomicAdd(p, myP)
31: return p × (4 × (n mod 2) − 2)
```

3.2.2.2 `SkipPer-SxC-Sm`: **x,** `CRS`**,** `CCS` **dizilerinin *paylaşımlı hafıza* üzerinde yaratılıp, *hafıza birleştirme* ile erişilmesi**

`SkipPer-SxC-Sm` algoritmasında, `ParallelSkipper` algoritmasının performans açısından optimize edilmesi hedeflenmiştir. Bu algoritmanın sözde kodu Algoritma 7'de görülebilir. Algoritmanın girdileri `CRS`, `CCS`, girdi matrisinin boyutu *dim*, matris üzerindeki sıfırdan farklı değerlerin sayısı *nnz*, **x**, her ipliğin kendi *kısmi* sonucunu yazdığı *p* listesi ve yineleme uzayının ayrıldığı parçaların başlangıç ve bitiş noktalarıdır. **x**, `CCS` ve `CRS` dizileri, bu dizilerdeki değerlere daha hı-

zlı erişmek ve güncellemek amacıyla *paylaşımlı hafıza* üzerinde tutulmuştur. Bunun yanısıra **x** üzerinde bu raporda da daha önce bahsedilen, ipliklerin **x** üzerindeki aynı indeksli elemanlarının yan yana tutulduğu, böylece bellek erişiminin hızlandırılmasının hedeflendiği *birleştirme* tekniği uygulanmıştır. `CCS` gösterimi için 3 dizi, *sharedCptrs, sharedRows, sharedCvals paylaşımlı hafıza* üzerinde yaratılmıştır. `CRS` gösterimi için 2 dizi, *sharedRptrs* ve *sharedCols paylaşımlı hafıza* üzerinde yaratılmıştır. `CRS` gösterimi için *rvals* dizisi kullanmaya bu yaklaşımda gerek olmadığı için, `CRS` gösteriminde bu dizi yaratılmamıştır. *Paylaşımlı hafıza* üzerindeki bellek lokasyonları düzenlendikten sonra iplikler yineleme uzayını paylaşıp hesaplamalarına başlamadan çnce senkronize edilmişlerdir. Algoritmanın bu kısmından sonrası `ParallelSkipper` ile aynıdır. Bu algoritmada *paylaşımlı hafıza*da hızlı erişilmek ve güncellenmek istenen diziler için ayrılan alan, *float* ve *int* veri tipleri kullanıldığında **x** için $dim \times BlockDim \times 4$, `CRS` ve `CCS` gösterimleri için $((2 \times nnz + dim + 1) + (nnz + dim + 1)) \times 4$ byte kadar olacaktır. Bu yüzden $48KB$ olan *paylaşımlı hafıza* kısıtlamasının aşılmaması için girdi matrisi boyutlarının

$$(\text{dim} \times \text{BlockDim } \times 4 \text{ bytes }) + ((3 \times nnz + 2 \times dim + 2) \times 4 \text{ bytes }) \leq 48 \times 1024 \text{ bytes}$$

eşitsizliğine uyması gerekmektedir. Bu durumda, 40 x 40 boyutlarında bir girdi matrisi için, $0.20, 0.30, 0.40$ yoğunluklarında `BlockDim` değişkeninin alabileceği değerler sırasıyla $282, 269$ ve 257 olacaktır. Bunun yanısıra, `SkipPer-SxC-Sm` algoritmasına tek GPU üzerinde yineleme uzayı içerisinde kendi parçasını tamamlayan *iplik*'lerin yeni parçaları dinamik olarak aldığı *dynamic scheduling* yaklaşımını entegre etmek `ParallelSkipper` algoritmasının aksine mümkün olmamaktadır.

Algoritma 7: SKIPPER-SXC-SM($cptrs, rows, cvals, rptrs, cols, dim, nnz, x, p, start, end$)

1: $\texttt{tid} \leftarrow iplikIdx.x + (blockIdx.x \times blockDim.x)$
2: $\texttt{iplikId} \leftarrow iplikIdx.x$
3: $\texttt{BlockDim} \leftarrow blockDim.x$
4: **for** $k = 1 \ldots dim$ **do**
5: $\quad myX[\texttt{BlockDim} \times k + \texttt{iplikId}] \leftarrow x[k]$
6: $\quad sharedRptrs[k] \leftarrow rptrs[k]$
7: $\quad sharedCptrs[k] \leftarrow cptrs[k]$
8: $sharedRptrs[\texttt{dim} + 1] \leftarrow rptrs[\texttt{dim} + 1]$
9: $sharedCptrs[\texttt{dim} + 1] \leftarrow cptrs[\texttt{dim} + 1]$
10: **for** $k = 1 \ldots nnz$ **do**
11: $\quad sharedCols[k] \leftarrow cols[k]$
12: $\quad sharedRows[k] \leftarrow rows[k]$
13: $\quad sharedCvals[k] \leftarrow cvals[k]$
14: $syncthreads()$
15: $chunkSize \leftarrow (end - start)/(\texttt{GridDim} \times \texttt{BlockDim}) + 1$
16: $myStart \leftarrow start + \texttt{iplikId} \times chunkSize$
17: $myEnd \leftarrow min(start + (\texttt{iplikId} + 1) \times chunkSize, end)$
18: $myP \leftarrow 0$
19: $prevG \leftarrow 0$
20: $g \leftarrow myStart$
21: **while** $g < myEnd$ **do**
22: $\quad grayDiff \leftarrow \texttt{Gray kod}_g \oplus \texttt{Gray kod}_{prevG}$
23: $\quad$ **for** *each* $grayDiff[j] = 1$ **do**
24: $\quad\quad$ **if** $\texttt{Gray kod}_g[j] = 1$ **then**
25: $\quad\quad\quad$ **for** $i = sharedCptrs[j] \ldots sharedCptrs[j + 1] - 1$ **do**
26: $\quad\quad\quad\quad myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{iplikId}] \leftarrow$
$\quad\quad\quad\quad myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{iplikId}] + sharedCvals[rows[i]]$
27: $\quad\quad$ **else**
28: $\quad\quad\quad$ **for** $i = sharedCptrs[j] \ldots sharedCptrs[j + 1] - 1$ **do**
29: $\quad\quad\quad\quad myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{iplikId}] \leftarrow$
$\quad\quad\quad\quad myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{iplikId}] - sharedCvals[rows[i]]$
30: $\quad prod \leftarrow 1$
31: $\quad$ **for** $i \ldots \texttt{dim}$ **do**
32: $\quad\quad prod \leftarrow prod \times myX[\texttt{BlockDim} \times i + \texttt{iplikId}]$
33: $\quad myP \leftarrow myP + (-1)^g \times prod$
34: $\quad prevG \leftarrow g$
35: $\quad g \leftarrow next(g)$
36: $p[\texttt{tid}] \leftarrow myP$

3.2.2.3 SpaRyser-SxC-Sm: x ve CCS gösterimlerinin *paylaşımlı hafıza*da yaratılarak *hafıza birleştirme* ile erişilmesi.

SpaRyser-SxC-Sm algoritmasında, Ryser-SxC-Sm algoritmasına benzer bir yaklaşım takip edilmiştir. Paylaşımlı hafıza üzerinde **x** ve girdi matrisi için yer ayrılmış, ancak matris yerine CCS gösterimi kullanılmıştır. Bu yüzden, paylaşımlı hafıza üzerinde matrisin boyutu kadar alan kullanmak yerine *cptrs*, *rows* ve *cvals* dizileri için 2 x $nnz + dim$ kadar alan kullanılmıştır. Bu algoritmanın sözde kodu Algoritma 8'de verilmiştir. $4.$ ve $10.$ satırlar arasında **x** ve CCS gösterimleri GPU ana belleğindeki değerler kullanılarak yaratılmaktadır. SpaRyser-SxC-Sm algoritması ayrıca **x** dizisi için *hafıza birleştirme* kullanmaktadır. Ryser-SxC-Sm algoritmasının tersine SpaRyser-SxC-Sm algoritmasında sadece işlenen sütundaki sıfırdan farklı değerler kullanılmakta ve 22-33 satırlar arasında yineleme yapılmaktadır. Sütunlar işlenirken, **x** üzerinde bulunan sıfırların sayısı *zeroNum* ve **x** üzerinde bulunan sıfırdan farklı değerlerin çarpımı güncellenmektedir. Eğer yineleme sonunda **x** üzerinde 0 yoksa, **x** üzerindeki sayılar çarpılarak permanent değerine eklenmektedir. Algoritmanın sonunda, her bir iplik kendi *kısmi* permanent toplamını *tid* değerlerini indeks olarak GPU ana belleğinde bulunan *p* dizisine yazmaktadır. Sonuç elde edilirken *p* dizisi ana belleğe kopyalanıp üzerindeki değerler toplanarak toplam permanent değeri elde edilmektedir. SpaRyser-SxC-Sm algoritmasının önemli bir özelliği, matrisin yarıdan daha az dolu olduğu (*density* $= 0.5$) varsayıldığında, Ryser-SxC-Sm algoritmasının zıttı olarak paylaşımlı hafızada daha az alan kullanması, böylece aynı matris için bir iplik bloğu içinde daha fazla iplik çalıştırabilmesidir.

Algoritma 8: SpaRyser-SxC-Sm($cptrs, rows, cvals, dim, nnz, x, p, start, end$)

1: $\texttt{tid} \leftarrow iplikIdx.x + (blockIdx.x \times blockDim.x)$
2: $\texttt{iplikId} \leftarrow iplikIdx.x$
3: $\texttt{BlockDim} \leftarrow blockDim.x$
4: **for** $k = 1 \ldots dim$ **do**
5: $\quad myX[\texttt{BlockDim} \times k + \texttt{iplikId}] \leftarrow x[k]$
6: $\quad sharedCptrs[k] \leftarrow cptrs[k]$
7: $sharedCptrs[\texttt{dim} + 1] \leftarrow cptrs[\texttt{dim} + 1]$
8: **for** $k = 1 \ldots nnz$ **do**
9: $\quad sharedRows[k] \leftarrow rows[k]$
10: $\quad sharedCvals[k] \leftarrow cvals[k]$
11: $syncthreads()$
12: $chunkSize \leftarrow (end - start)/(\texttt{GridDim} \times \texttt{BlockDim}) + 1$
13: $myStart \leftarrow start + \texttt{iplikId} \times chunkSize$
14: $myEnd \leftarrow min(start + (\texttt{iplikId} + 1) \times chunkSize, end)$
15: $myP \leftarrow 0$
16: calculate myX using $\texttt{Gray kod}_{myStart-1}$
17: $prod \leftarrow \prod_{n=1}^{\texttt{dim}} myX[i]$ for $myX[i] \neq 0$
18: $zeroNum \leftarrow \sum_{n=1}^{length(myX)} myX[i]$ for $myX[i] = 0$
19: **for** $g = myStart \ldots myEnd - 1$ **do**
20: $\quad j \leftarrow \log_2(\texttt{Gray kod}_g \oplus \texttt{Gray kod}_{g-1}) + 1$
21: $\quad s \leftarrow 2 * \texttt{Gray kod}_g[j] - 1$
22: $\quad$ **for** $i = cptrs[j] \ldots cptrs[j+1] - 1$ **do**
23: $\quad\quad$ **if** $myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{iplikId}] = 0$ **then**
24: $\quad\quad\quad zeroNum \leftarrow zeroNum - 1$
25: $\quad\quad\quad myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{iplikId}] \leftarrow$
$\quad\quad\quad myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{iplikId}] + s \times sharedCvals[i]$
26: $\quad\quad\quad prod \leftarrow prod \times myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{iplikId}]$
27: $\quad\quad$ **else**
28: $\quad\quad\quad prod \leftarrow prod/myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{iplikId}]$
29: $\quad\quad\quad myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{iplikId}] \leftarrow$
$\quad\quad\quad myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{iplikId}] + s \times sharedCvals[i]$
30: $\quad\quad\quad$ **if** $myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{iplikId}] = 0$ **then**
31: $\quad\quad\quad\quad zeroNum \leftarrow zeroNum + 1$
32: $\quad\quad\quad$ **else**
33: $\quad\quad\quad\quad prod \leftarrow prod \times myX[\texttt{BlockDim} \times sharedRows[i] + \texttt{iplikId}]$
34: $\quad$ **if** $zeroNum = 0$ **then**
35: $\quad\quad myP \leftarrow p + (-1)^g * prod$
36: $p[\texttt{tid}] \leftarrow myP$

3.2.3 Birden Fazla GPU ve hibrit düzende Seyrek Matrislerin Permanent Değerinin Hesaplanması

3.2.3.1 `SpaRyser-SxC-Sm-MG+`: `SpaRyser-SxC-Sm` ve `ParallelSpaRyser` algoritmalarının hibrit düzende kullanılması

`SpaRyser-SxC-Sm-MG+` algoritmasında `Ryser-SxC-Sm-MG+` algoritmasında kullanılan yaklaşım takip edilmiştir. `Ryser-SxC-Sm-MG+` algoritmasında yineleme uzayı çok sayıda ve küçük parçalara bölünmekte, sonrasında her bir hesaplama noktası (GPUlar veya CPUlar) bu parçalardan birini alıp *kısmi* hesaplamaya başlamaktadır. Hibrit koşturma için, bu GPU cihazlarının yanına istenirse `ParallelSkipper` algoritmasını çalıştıran CPU çekirdekleri de eklenebilmektedir. Bunun yanısıra, iş dağıtımında dinamik model yineleme uzayını küçük parçalara bölüp farklı hesaplama noktalarına dağıtmak suretiyle gerçeklenmiş ve farklı hızlarda çalışan CPU çekirdeklerinin ve GPU'ların yineleme uzayını paylaşmaları sağlanmıştır. `SpaRyser-SxC-Sm-MG+` algoritmasına benzer şekilde, `SkipPer-SxC-Sm-MG+` algoritmasında da parça boyutunun doğru seçilmesi önemlidir. Bu tarz, çok sayıda aynı olmayan hesaplama noktalarını tek bir hesaplama için kullanan hibrit algoritmalarda parça boyutunun çok büyük seçilmesi, hızlı cihazların yavaş cihazları beklemesine neden olacaktır. Bunun aksi olarak parça boyutunun çok küçük seçilmesi ise, fonksiyonların fazla sayıda çalışmasının getireceği ekstra haberleşme ve hesaplama masrafıyla hibrit yaklaşımın birden fazla hesaplama noktasıyla bir hesaplama noktasından daha yavaş çalışma ihtimalini doğuracaktır. Parça boyutunun belirlenmesi için, `SpaRyser-SxC-Sm-MG+` algoritmasında izlenen ampirik yaklaşım takip edilerek parça boyutu 2^{dim-30} olarak seçilmiştir.

3.2.3.2 `SkipPer-SxC-Sm-MG+`: `SkipPer-SxC-Sm` ve `ParallelSkipper` algoritmalarının hibrit gerçeklenmesi

`SkipPer-SxC-Sm-MG+` algoritmasında birden fazla `SkipPer-SxC-Sm` fonksiyonu çalıştıran GPU cihazı, `SpaRyser-SxC-Sm-MG+` algoritmasına benzer şekilde aynı anda çalıştırılmıştır. Bu algoritmada, yineleme uzayı çok sayıda küçük parçaya bölünmüş ve GPU cihazlarına bu parçalardan biri verilip aynı anda kısmi hesaplamaları başlatılmıştır. GPU cihazlarından birisi sorumlu olduğu parçayı bitirip, kısmi sonucunu ana belleğe kopyaladığında, bu GPU cihazından sorumlu CPU *iplik*'i bu sonucu toplam permanent değerine ekleyip, eğer parça kaldıysa GPU cihazı üzerinde yeni bir parçayla tekrar `SkipPer-SxC-Sm` algoritmasını çağırmaktadır.

3.3 Permanent Değerinin Yakınsanması

Ölçülmek istenen belirli bir değer P düşünelim. Bizim uygulamamızda P, permanent olacaktır. Yansız bir kestirici yöntemi X yarattığımızı farz edelim. Bu kestirici X'in beklentisi $\mathbf{E}[X] = P$ olsun. X'i baz alarak P'ye X'in N çıktısı ortalanarak ulaşılabilir. Daha spesifik olarak, X_i'nin kestiricinin i'ninci çıktısını temsil ettiği düşünülürse, $X' = \sum_{i=1}^{N} \frac{X_i}{N}$ değeri P için ortalanmış ölçüm olacaktır. $\Pr(|X' - P| \leq \varepsilon P) \geq 1 - \delta$ olduğu durumda (ε, δ)-yakınsama yapıldığı söylenebilir. Genel olarak, bir (ε, δ) yakınsama $\mathcal{O}\left(\frac{\mathbf{E}[X^2]}{\mathbf{E}[X]^2} \cdot \frac{1}{\varepsilon^2} \cdot \log(\frac{1}{\delta})\right)$ deneme ile elde edilebilir. $\frac{\mathbf{E}[X^2]}{\mathbf{E}[X]^2}$ değeri yakınsamanın polinom zamanında koşturulup koşturulamayacağını belirleyeceği

için kritik oran (*critical ratio*) olarak adlandırılmaktadır. CPU ve GPU'lar üzerinde aşağıda bahsedilecek yöntemlerin paralelleştirilmesi daha fazla deneme ile daha küçük ε ve δ değerlerine ulaşılmasını sağlayacaktır.

Verilen $n \times n$ bir **A** matrisi dolu bir permütasyon (eşlenik iki parçalı çizgesinde mükemmel bir eşleme) barındırıyorsa bu bir *destek* olarak adlandırılır. Eğer **A** matrisinde sıfır olmayan her eleman en az bir dolu permütasyonun içerisindeyse bu tam destek (*full support)* olarak adlandırılır. Tam destek özelliğini barındıran her **A** matrisi iki pozitif köşegen matris **R** ve **C** ile herhangi bir sütun ve satırındaki elemanların toplamı bir eden, çifte rassallık (*doubly stochastic)* yapısına sahip bir **RAC** matrisi oluşturacak şekilde ölçeklendirilebilir. Sinkhorn ve Knopp (1967) algoritması matrisleri çifte rassallık yapısına ölçeklendirmek için sıkça kullanılan bir algoritmadır. Bu algoritma, limiti çifte rassal bir matris olan bir matris dizisi üretir. Bu dizideki matrislerin sütun ve satırları değişimli olarak çifte rassal bir matrise doğru normalize edilir. Eğer **A** matrisi simetrik ve ölçeklenebilir ise, $\mathbf{S} = \mathbf{RAC}$ matrisi çifte rassal olacaktır. Sinkhorn-Knopp algoritması bu simetrik çifte rassal matrisi limit olarak hedeflerken, dizideki diğer matrisler simetrik olmayabilir. Bütün süreç esnasında simetriklik özelliğini koruyan algoritmalar da literatürde bulunmaktadır (Knight ve Ruiz, 2013; Knight vd., 2014).

Eğer $n \times n$ negatif olmayan bir **A** matrisi çifte rasgele $\mathbf{S} = \mathbf{RAC}$ matrisine pozitif köşegen (*diagonal)* **R** ve **C** matriksleriyle ölçekleniyorsa

$$g_{\mathbf{A}}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x}^T \mathbf{A} \boldsymbol{y} - \sum_{i=1}^{n} \ln x_i - \sum_{j=1}^{n} \ln y_j \tag{4}$$

fonksiyonu pozitif x_i ve y_i için **x** = diag(**R**) ve **y** = diag(**C**)'de minimum değerine erişecektir (Kalantari ve Khachiyan, 1996, Önerme 2). Bu önermeden

$$n \leq g_{\mathbf{A}}(\boldsymbol{x}, \boldsymbol{y}) \leq n + n \ln n$$

alt ve üst sınırları çıkarılabilir. Burada alt sınır **A** bir permütasyon matrisi iken, üst sınır da **A**'nın bütün değerleri 1 ilen sağlanır. İki durumda da $\boldsymbol{x}^T \mathbf{A} \boldsymbol{y} = n$ olur.

Idel (2016) çifte rasgele ölçeklemelerin bilinen sonuçlarının detaylı bir özetini sunmuş, yakın zamanda da Chakrabarty ve Khanna (2018) Sinkhorn-Knopp algoritmasının daha detaylı bir analizini sunmuştur. Bunların yanında konveks optimizasyona (*convex optimization)* dayanan farklı ve verimli algoritmalar önerilmiştir (Allen-Zhu vd., 2017; Cohen vd., 2017). Yakın zamanda (Dufossé vd., 2020) çalışmamızda, bir matrisin ölçek çarpanları, alt matrisleri ve bu matrislerin permanent değerleri arasındaki ilişkiyi gösterdik.

Lemma 1. *(Dufossé vd., 2020) **A**, $n \times n$, tam destekli bir 0-1 matrisi olsun.* $\mathbf{S} = \mathbf{RAC}$ *çifte rassallık içeren bir matris olacak şekilde iki köşegen ölçeklendirme matrisi* **R** *ve* **C** *var olduğunu*

kabul edelim. Bu durumda

$$Per(\mathbf{S}) = Per(\boldsymbol{A}) \cdot \prod_{i=1}^{n} r_i \cdot c_i \text{ ve } r_i \cdot c_j = \frac{Per(\boldsymbol{A}_{ij})}{Per(\boldsymbol{A})} \cdot \frac{Per(\mathbf{S})}{Per(\mathbf{S}_{ij})}$$

olacaktır.

Bu önsava dayanan ve permanent yakınsama için kullanılması önerilen yöntem, Algoritma 9'da gösterilmiştir. Bu yöntem permanenti sıfırdan farklı bir 0-1 **A** matrisini girdi olarak alıp, bir $X_{\mathbf{A}}$ rassal değişkeni, ve eğer mümkünse dolu bir permütasyon üretmektedir. Başlangıçta $X_{\mathbf{A}} = 1$'dir. Algoritma n adımda ilerler ve her adımda permütasyona sıfırdan farklı bir eleman eklemek suretiyle dolu bir permütasyon elde eder. Her i adımında, **A**'nın ölçeklenmiş i'inci satırından henüz eşlenmemiş, sıfırdan farklı bir eleman seçilir. Bu seçilen eleman, eşleşen sütunu da permütasyon içerisine ekler. Algoritma bu adımda i için eşleşen sütunu göstermek için $\sigma_1^{(i)}$ notasyonunu kullanmaktadır. Her adımda rasgele değişken $X_{\mathbf{A}}$ seçilen elemanın tersi ile çarpılır. Algoritmanın çalışması için o adımda kullanılan $\mathbf{A}^{(i)}$ matirisindeki desteği olmayan elemanlar elenir. Bu eleme Dulmage ve Mendelsohn (1958) ayrıştırması uygulanarak, ayrıştırma sonucundaki ayrık blok içerisindeki elemanlar silinerek yapılır (Pothen ve Fan, 1990).

Algoritma 9 bir permütasyon yaratmaktadır. Ölçek çarpanları, bu permütasyona olduğundan, i'nci adımda ölçekleme matrislerini belirtmek için $\mathbf{R}^{(\sigma,i)}$ and $\mathbf{C}^{(\sigma,i)}$ notasyonu kullanılmıştır. $\sigma_1^{(i)}$'in i'nci adımda seçilen sütundur. $r_1^{(i)}$ of $\mathbf{R}^{(\sigma,i)}$'ın ölçek çarpanının ilk elemanı olduğundan

$$X_{\mathbf{A}} = \frac{1}{\prod_{i=1}^{n} r_1^{(\sigma,i)} \cdot c_{\sigma_1^{(i)}}^{(\sigma,i)}} \cdot$$

yazılabilir.

Algoritma 9: **P**ermanent yakınsama (tek çalışma)

Girdi: $n \times n$, 0-1 matris **A**

Çıktı: Permanent ölçümü $X_{\mathbf{A}}$

σ: dolu bir permütasyon

1: $X_{\mathbf{A}} \leftarrow 1$

2: $\mathbf{A}^{(1)} \leftarrow \mathbf{A}$

3: **for** $i = 1$ **to** n **do**

4: $\mathbf{A}^{(i)}$'den desteği olmayan elemanları çıkar

5: $[\mathbf{R}^{(\sigma,i)}, \mathbf{C}^{(\sigma,i)}] \leftarrow$ **Ö**lçekle($\mathbf{A}^{(i)}$)

6: $\mathbf{S} = \mathbf{R}^{(\sigma,i)}\mathbf{A}^{(i)}\mathbf{C}^{(\sigma,i)}$ matrisinden, olasılık yoğunluk fonksiyonu sıfırdan farklı değerler için

$$p_j = \frac{s_{1,j}}{\Sigma_{k \in \mathbf{A}^{(i)}(1,:)} s_{1,k}}$$

rassal bir sütun seçilir.

7: $X_{\mathbf{A}} \leftarrow X_{\mathbf{A}}/p_j$

8: $\mathbf{A}^{(i+1)} \leftarrow \mathbf{A}^{(i)}_{1j}$: $\mathbf{A}^{(i)}$'nin ilk satırını ve j'inci sütununu sil.

9: $\sigma(i) \leftarrow j$

Algoritmanın attığı n adımın her biri, bir Dulmage-Mendelsohn ayrışmasını ve bir ölçeklemeyi gerektirmektedir. Bu adımların her biri sıfırdan farklı m elemanı olan bir $n \times n$ matriste $\mathcal{O}(\sqrt{n}m)$ zamanda gerçekleştirilebilir (Pothen ve Fan, 1990). Ölçekleme her adımda, ve farklı permütasyonlar için farklı şekillerde yapılacağından bu kısmın hızının ve verimliliğinin tüm yakınsama işleminin performansı için önemli olduğu düşünülmektedir.

Literatürdeki diğer güncel teknikler ise pratik gerçeklemeleri zor olan konveks optimizasyon teknikleri (Allen-Zhu vd., 2017; Cohen vd., 2017) olup en düşük hesaplama zaman karmaşıklığı $\tilde{\mathcal{O}}(mn + n^{7/3})$ ve $\tilde{\mathcal{O}}(m^3/2)$'dır. Bu teknikler satır/sütun toplamlarında ortalamadan sapmayı kullanıp karmaşıklıktaki $\log n$ teriminden kurtulabilmektedir. Şu ana kadar Sinkhorn-Knopp ile olan iyi tecrübemiz ve algoritmaya iyi derecede hakim olmamız nedeniyle matris ölçekleme için ilk tercihimiz bu algoritma olacaktır. Bu algoritmayı tercih etmemizin bir nedeni de, ölçekleme esnasında her biri $\mathcal{O}(m)$ zaman alan $\mathcal{O}(n^2 \ln n/\varepsilon^2)$ adım kullandığının gösterilmiş olmasıdır. Dolayısıyla, Sinkhorn-Knopp algoritması ölçekleme için kullanıldığı zaman Algoritma 9'ün hesaplama karmaşıklığı $\tilde{\mathcal{O}}(n(\sqrt{n}m + mn^2 \ln n))$ olmaktadır. Gerçeklemeleri daha kolay olan Knight ve Ruiz (2013); Knight vd. (2014) gibi diğer ölçekleme algoritmalarının hiçbirinin polinom zamanda hesaplama karmaşıklığında çalıştığı gösterilmemiştir. Ayrıca, daha önceki çalışmalardaki bulgularımıza göre Sinkhorn-Knopp, 0-1 matrislerin üzerinde çalıştığında 5-10 adımda bile çok iyi bir ölçekleme elde edildiği gözlenmiştir.

3.4 CPU Üzerinde Permanent Değerinin Yakınsanması

3.5 Yoğun Matrislerin Permanent Değerlerinin Yakınsanması

3.5.1 GPU Üzerinde Permanent Yakınsama

Eşlenik bir çizge üzerinde mükemmel eşleşmelerin sayısını hesaplamak, bu eşlenik çizgeyi ifade eden *komşuluk* matrisinin permanent değerini hesaplamakla eşdeğerdir. Bir eşlenik çizgenin *komşuluk matrisi* sadece sütun ve satır indekslerine denk düşen çizge noktaları arasında bir kenar olup olmadığını göstermektedir. Bu sebepten, bir *0-1* matrisin permanenti, bu matrise denk düşen eşlenik çizgenin mükemmel eşleşme sayısı kullanılarak tahmin edilebilir. `Rasmussen` ve `Scaling` algoritmaları, bu yöntemi kullanarak permanent tahmin eden algoritmalardır. Bu yüzden bu tür algoritmalarda olabildiğince çok sayıda kez algoritmayı koşturmak daha isabetli sonuç alabilmek için önemlidir.

3.5.1.1 `RasmussenGpu`: `Rasmussen` algoritmasının GPU üzerinde gerçeklenmesi

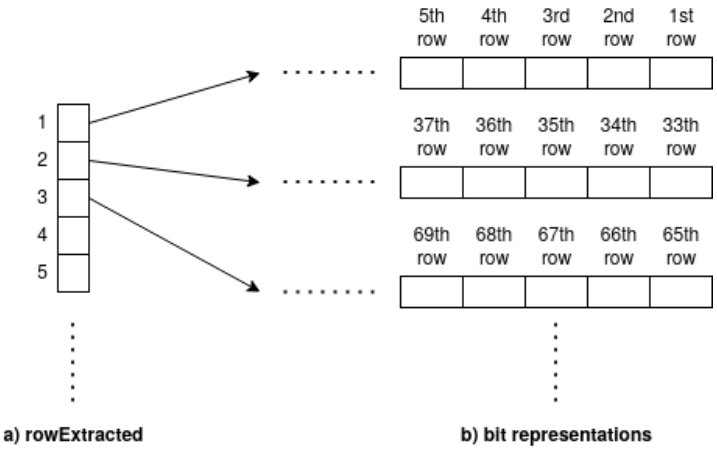
`Rasmussen` algoritması CPU'lar üzerinde deneyleri ipliklere dağırmak suretiyle paralelleştirilebilir. Bu durumda her bir iplik kendi deneylerini yapacak ve her bir deneyin sonucunu *global* bir değişkene atomik operasyon ile ekleyecektir. Bütün deneylerin toplamı elde edildikten sonra bu sayı, toplam deney sayısına bölünerek *0-1* matrisinde permanent tahminini verecek olan ortalama elde edilebilir. Benzer bir yaklaşım GPU'lar üzerinde de izlenebilir. Algoritma 10'da `RasmussenGpu` ile benzer olan `Rasmussen+Gpu` algoritmasının sözde kodu görülebilir. Algoritmanın girdileri, matris *mat*, matrisin boyutu *dim* ve her bir ipliğin sahip olduğu `iplikID` ile çarpılarak rasgele sayı elde etmekte kullanılacak olan `seed` değerleridir. Algoritmanın çıktısı olarak ise, her bir deneyin sonucunun yazılı olduğu *p* dizisi ana belleğe kopyalanıp ortalaması alınmaktadır. `RasmussenGpu` algoritması Algoritma 10'un $5.$ satırında görülebileceği üzere, girdi matrisini GPU ana belleğinden paylaşımlı hafızaya kopyalamaktadır. Daha önce bahsedildiği gibi, paylaşımlı hafıza bu raporda kullanılan ve pek çok modern GPU'da 48 KB boyutundadır. Girdi matrisinin paylaşımlı hafızaya taşınması, matris tamsayı olduğu zaman paylaşımlı hafızada $boyut^2$ x 4 byte alan kullanılması gerektiği anlamına gerekmektedir. Bu sebepten `RasmussenGpu` algoritması kullanıldığında, permanent değeri yakınsanabilecek matrislerin boyutu `int` ve `float` veri tipi için 110 x 110, `double` için 55 x 55 ile sınırlı olmaktadır. Ancak bu durum, bir iplik bloğunda içerisinde kullanılan iplik sayısının paylaşımlı hafızaya etkisi olmadığı için `BlockDim` değişkenini etkilememektedir.

Her bir algoritma adımında çıkarılan sütunları tutmak için bir tamsayı dizisi iplik yazmaçlarında tutulmuştur, ancak söz konusu iplik yazmaçları da sınırlı bir bellek olduğu için bu dizilerin de boyutları kısıtlıdır. Bu sebepten ötürü bir sütunun çıkarıp çıkarılmadığını anlamak için bit operasyonları kullanılmıştır. Thread yazmaçlarında tutulan her biri 32 bit olan tamsayı veri tipindeki değerler, 32 adet sütunun (her sütun için 1 bit) çıkarılıp çıkarılmadığını kontrol etmek için kullanılmıştır. Bahsedilen bit operasyonlarının nasıl işlediği Şekil 5'te gösterilmiştir. Sütunları

kaydetmek için kullanılan bu yaklaşımdan ötürü, her bir ipliğin kendi yazmaçında $dim/32 + 1$ kadar tamsayı veri tipinde değer tutması yeterli olmaktadır. `RasmussenGpu` algoritmasının yoğun matrislerde çalışan versiyonu için, paylaşımlı hafıza kısıtlaması yazmaç kısıtlamasından daha önce geldiğinden yazmaç kısıtlaması aşılmamış olacaktır. Girdi matrisinin boyutu 110 x 110 olduğunda, çıkarılan sütunları tutmak için kullanılan *colExtracted* dizisinin boyutu 4 tamsayı büyüklüğünde olacaktır. Bunun yanısıra sütunları tutmak için bir dizi oluşturulduğu halde, satırlar sırayla işlediği için, ayrı bir bellek kullanımına ihtiyaç yoktur. Bu sebepten Algoritma 10'un 11. satırındaki gibi en az sayıda sıfırdan farklı değeri içeren satırı bulmak yerine `RasmussenGpu` basitçe indeksi yineleme numarasına eşit olan satırı seçmektedir. Bu satır seçildikten sonra, satırdaki çıkarılmayan sütunlarda bulunan sıfırdan farklı değerler toplam permanent değeriyle çarpılır. Satır üzerindeki rasgele sıfırdan farklı sayılardan biri çıkarıldıktan sonra, bu sıfırdan farklı sayının sütunu Algoritma 10'un 16. satırında olduğu gibi çıkarılır. GPU iplikleri tarafından aynı anda alınan deneyler bittikten sonra her bir iplik kendi sonucunu GPU ana belleğinde bulunan *p* dizisine kopyalar. Bütün iplikler deneylerini bitirip fonksiyon tamamlandığında *p* dizisi ana belleğe aktarılır ve dizideki bütün sonuçların ortalaması alınarak nihai permanent yakınsaması değerine ulaşılmış olur.

Algoritma 10: RASMUSSEN+GPU($mat, dim, rand, p$)

1: $tid \leftarrow$ GPUdaki iplik id
2: $iplikId \leftarrow$ bloktaki iplik id
3: **for** $k = 1 \ldots (\texttt{dim} * \texttt{dim})/\texttt{BlockDim} + 1$ **do**
4: **if** $\texttt{BlockDim} * k + iplikId < \texttt{dim} * \texttt{dim}$ **then**
5: $sharedMat[\texttt{BlockDim} * k + iplikId] \leftarrow \texttt{mat}[\texttt{BlockDim} * k + iplikId]$
6: blok içerisindeki ipliklerin eşzamanlaması
7: $perm \leftarrow 1$
8: $rowExtracted[\text{dim}/32 + 1]$
9: $colExtracted[\text{dim}/32 + 1]$
10: **for** $k = 1 \ldots \text{dim}$ **do**
11: $row \leftarrow$ row with the least number of nonzero
12: $nnz \leftarrow$ nonzero number of row
13: $perm \leftarrow perm * nnz$
14: $col \leftarrow$ column of the randomly chosen nonzero in row
15: $rowExtracted[row/32] \leftarrow rowExtracted[col/32] \mid (1 << (row \ \% \ 32))$
16: $colExtracted[col/32] \leftarrow colExtracted[col/32] \mid (1 << (col \ \% \ 32))$
17: $p[tid] \leftarrow perm$



Şekil 5: (a) çıkan (ya da işlenen) satırların takibi için kullanılan dizi (b) indislerin ikili tabanda gösterimleri

3.5.1.2 `Rasmussen+Gpu`: `Rasmussen+` algoritmasının GPU üzerinde gerçeklenmesi

Bu algoritma `Rasmussen` algoritmasının GPU üzerinde gerçeklenmesinde `RasmussenGpu` algoritması ile aynı çizgileri takip etmekte, yalnızca, algoritmanın her bir adımında, sıradaki satırın nasıl seçileceği konusunda farklılaşmaktadır. Bu algoritmada, algoritmanın gelecek adımında sıfırdan farklı değerleri en az sayıda olan satır seçilmektedir. `Rasmussen+Gpu` algoritmasını da girdi matrisi paylaşımlı hafızaya kopyalandığı için `RasmussenGpu` algoritmasını etkileyen bellek kısıtlamaları etkilemektedir. Bunun yanısıra, `Rasmussen+Gpu` algoritması, `RasmussenGpu` algoritmasına ek olarak, iplik yazmaçları üzerinde *colExtracted* dizisinin yanısıra *rowExtracted* algoritması da tutmaktadır. Çünkü `RasmussenGpu` algoritmasında olduğu gibi, satırlar yineleme numarasına göre incelenmediği için hangi satırların çıkarıldığının kaydını da tutmak gerekmektedir. Algoritma 10 11. satırda görülebileceği gibi, bu algoritma çıkarılmamış bütün satırlar üzerinde bir yineleme yaparak, sıfırdan farklı değer sayısı en az olan satırı seçmektedir. Satır çıkarma işlemi Algoritma 10 15. satırda ve çıkarılan satırlan bit operasyonları kullanılarak küçük bir bellek alanında kayıt edilmesi Şekil 5'de görülebilir. Bu algoritmada çıkarılan sütunlar ve satırlar için farklı diziler tutulduğundan iplik başına daha fazla yazmaç belleği kullanılmaktadır. Ancak daha önce bahsedildiği gibi, paylaşımlı hafıza için geçerli olan bellek kısıtlaması algoritmanın permanentini hesaplayabildiği maksimum matris boyutunu yazmaç belleği kısıtlamasından çok daha önce etkilediğinden `RasmussenGpu` ve `Rasmussen+Gpu` algoritmaları arasında permanenti yakınsanabilen maksimum büyüklükteki matris açısından bir fark bulunmamaktadır.

3.5.1.3 `ScalingGpu`: `Scaling` algoritmasının GPU üzerinde gerçeklenmesi

`ScalingGpu` algoritması, GPU üzerinde çalıştırdığı fonksiyonda Algoritma 11 algoritmasındaki mantığı kullanmaktadır. İki algoritmanın girdileri de aynıdır. Bu girdiler, matris, *mat*, bu matrisin boyutu *dim*, her bir ipliğin kendi `iplikID`'si ile çarpılıp rasgele sayı yaratma fonksiyonunda kaynak olarak kullanılacak bir rasgele sayı *rand*, sonuç dizisi *p*, ölçekleme matrislerinin *diagonal* elemanlarını gösteren *R* ve *C* dizileri, ölçekleme aralıklarını tutmak için kullanılan *SInterval* ve

ölçekleme sayısını tutmak için kullanılan *stime* değerleridir.

Algoritma 11: `Scale`($sharedMat$,$rowExtracted$,$colExtracted$, $R, C,$ `STime`)

```
1:  for k = 1 ... STime do
2:    for j = 1 ... dim do
3:      if j^th column is not extracted then
4:        colSum ← 0
5:        for i = 1 ... dim do
6:          if i^th row is not extracted then
7:            colSum ← colSum + R[tid * dim + i] * sharedMat[i * dim + j]
8:        if colSum = 0 then
9:          return
10:       C[tid * dim + j] ← 1/colSum
11:   for i = 1 ... dim do
12:     if i^th row is not extracted then
13:       rowSum ← 0
14:       for j = 1 ... dim do
15:         if j^th column is not extracted then
16:           rowSum ← rowSum + C[tid * dim + j] * sharedMat[i * dim + j]
17:       if rowSum = 0 then
18:         return
19:       R[tid * dim + i] ← 1/rowSum
```

`Scaling+Gpu` algoritması çalışmaya başladığında, öncelikle Algoritma 12 görülebileceği gibi, öncelikle girdi matrisini paylaşımlı hafızaya kopyalamaktadır. *R* ve *C* dizileri, paylaşımlı hafıza üzerinde kısıtlı alan olmasından ötürü GPU ana belleğinde tutulmaktadır. Bunun sebebi her bi ipliğin herhangi bir iterationda çıkarmak için rasgele satır ve sütun seçmesi, bunun her ipliğin tuttuğu dizilerin algoritmanın koşturulması sırasında farklılaşmasına yol açması, ve sonuç olarak her bir ipliğin kendi *R* ve *C* dizilerini tutması gerekmesine yol açmasıdır. Eğer *R* ve *C* matrislerinin her biri paylaşımlı hafıza üzerinde tutulmak istenirse, bu toplam 2 x dim x $BlockDim$ x 4 byte boyutunda hafıza gerektirecek, bu da 40 x 40 boyutunda bir tamsayı matrisi için çalıştırılabilecek toplam blok sayısının 153 olmasına yol açacaktır. Bunun yanısıra, natris büyüdükçe bu ihtiyaç büyüyecek ve büyüyen matris boyutuyla birlikte daha az iplik çalıştırılmasına sebep olacaktır. Bu sebepten, algoritma her bir GPU ipliğinin dim x $BlockDim$ x $GridDim$ boyutundaki dizilerini GPU ana belleğinde tutacak ve ipliklerin id'leri ile erişebilecekleri şekilde gerçeklenmiştir. Algoritma koşturulmaya başlanmadan önce, her bir ipliğin *R* ve *C* matrislerindeki sayılar 1'e sabitlenmiştir. Ayrıca *colExtracted* dizisi iplik yazmaçlarında yaratılmıştır, bu işlem Algoritma 12'nın 12. satırında görülebilir. `ScalingGpu` ile `Scaling+Gpu` algoritmalarının farkı, `ScalingGpu` algoritmasında satırlar en az sıfırdan farklı eleman sayısını seçmek yerine yineleme sırasına göre seçildiği için *rowExtract* dizisine gerek duyulmamasıdır. Bu işlemden sonra, satırdan seçilecek sıfırdan farklı değerin mükemmel eşleme içinde bulunma ihtimalini arttırmak için matris ölçekleme algoritması kullanılarak normalize edilir. Bu ölçekleme algoritması parametre olarak alınan *SInterval* kez çağırılır ve girdi matrisini ölçekleyecek diagonal değerlerini tut-

Algoritma 12: SCALING+GPU($mat, dim, rand, p, R, C$, SINTERVAL, STIME)

```
1: tid ← GPUdaki iplik id
2: iplikId ← bloktaki iplik id
3: for k = 1 ... (dim * dim)/BlockDim + 1 do
4:    if BlockDim * k + iplikId < dim * dim then
5:       sharedMat[BlockDim * k + iplikId] ← mat[BlockDim * k + iplikId]
6: bloktaki ipliklerin eşzamanlanması
7: for k = 1 ... dim do
8:    R[tid × dim + k] ← 1
9:    C[tid × dim + k] ← 1
10: perm ← 1
11: rowExtracted[dim/32 + 1]
12: colExtracted[dim/32 + 1]
13: for k = 1 ... dim do
14:    row ← row with the least number of nonzero
15:    if k % scaleInterval = 0 then
16:       SCALE(sharedMat, dim, rowExtracted,colExtracted, R, C, STime)
17:    col ← column of the randomly chosen unextracted nonzero in row, where each nonzero
       have the probability density function:
18:    perm ← ..
19:    rowExtracted[row/32] ← rowExtracted[col/32] | (1 << (row % 32))
20:    colExtracted[col/32] ← colExtracted[col/32] | (1 << (col % 32))
21: p[tid] ← perm
```

tuğu *R* ve *C* dizilerini günceller. Ölçekleme algoritmasının sözde kodu Algoritma 11'de bulunabilir. Ölçeklemenin her bir adımında, önce sütunlar sonra ise satırlar dengelenir. Dengeleme sırasında eğer herhangi bir satır veya sütunda hiç sıfırdan farklı değer bulunamazsa ölçekleme algoritması koşmayı durdurur ve döner. Çünkü yapılan deney için tahmin 0 çıkacağından artık ölçeklemeye ve yakınsamaya devam eteye gerek kalmamıştır. Ölçeklemeden sonra, $R[i]$ x $sharedM[i * dim + j]$ x $C[j]$ değeri matris üzerinde ölçeklenmiş (i, j) değerini verir. Ölçekleme tamamlandıktan sonra eldeki satırdan bir sütunu daha çıkarılmamış sıfırdan farklı değer seçilir ve permanent değeri bu seçilen değerin ölçeklenmiş karşılığı kadar arttırılır. Daha sonra, bu satırın bulunduğu sütun matristen çıkarılarak *colExtracted* dizisi güncellenir.

3.5.2 Birden Fazla GPU ve Hibrit Düzende Permanent Yakınsama

3.5.2.1 `Rasmussen+MGpu`: `Rasmussen+` algoritmasının birden fazla GPU ve birden fazla GPU ve CPU için gerçeklenmesi

`Rasmussen+Gpu` algoritması, bir eşlenik matris üzerindeki mükemmel eşlemelerin sayısını, ya da başka bir ifadeyle bir 0-1 matrisin permanentini hızlı ve isabetli bir şekilde tahmin edebilmektedir. Algoritma tarafından yapılan deneylerin sayısı arttıkça yapılan tahminin isabeti artacaktır, ancak bununla beraber, algoritmanın çalışma süresi de artacaktır. Doğacak iş yükünü hızlı bir şekilde hesaplayabilmek için, bu iş yükünü birden fazla hesaplama noktası (CPU veya GPU) üzerine dağıtmak mümkündür. `Rasmussen+MGpu` algoritmasında birden fazla CPU çekirdeği GPU üzerinde `Rasmussen+Gpu` algoritmasını çağırmakta, aynı anda birden fazla başka CPU

çekirdekleri de CPU üzerinde `Rasmussen+` algoritmasını çalıştırabilmektedir. `Rasmussen+MGpu` algoritmasında iş yükünü hesaplama noktalarına dağıtmak için eşit parçalara bölmek yerine birden fazla küçük parçaya bölme, ve bu parçalardan birini alıp bitiren cihazın yapılan toplam deney sayısını kontrol etmesi, ve eğer istenen sayıda toplam deney henüz yapılmamışsa parça boyutu miktarınca yeni deney yapması yaklaşımı izlenmiştir. Bu noktada, iş yükü (toplam deney sayısı) parçalara ayırılırken parça büyüklüğünü doğru seçmek önem kazanmaktadır. Eldeki iş bir yakınsama algoritması olduğu için, parçaları hesaplamak tam permanent değerini hesaplamak kadar uzun sürmeyecektir. Bunun yanısıra her bir GPU fonksiyonu çağırılması işlemi ekstra bir iş yükü getirmekte, söz konusu fonksiyon bir yakınsama GPU fonksiyonu olduğunda bu ekstra iş yükü işin kendisinden daha maliyetli olup birden fazla gpu kullanan ve hibrit hesaplamanın bir gpu kullanan hesaplamadan daha uzun sürmesine neden olabilmektedir. Bu sebepten, `Rasmussen+MGpu` algoritması için, her bir kerne 1024 blok ve her bir blokta 1024 iplik çalıştırmak suretiyle 1 milyon deney koşturmaya programlanmıştır.

4 Bulgular

4.1 Test Ortamı

Gereç ve yöntem kısmında anlatılan algoritmaların performansını karşılaştırmak ve analiz etmek için sentetik matrisler üzerinde kapsamlı testler yürütülmüştür. Deneylerde kullanılan bilgisayarın donanım ve ortam özellikleri aşağıda verilmiştir.

Operating System: Ubuntu 20.04.2 LTS (Focal Fossa)
Kernel: Linux version 5.4.0-65-generic
CPU: Intel Xeon E5-2620 v4 @ 2.10GHz x 2 *(16 Cores, 32 Threads)*
GPU0: NVIDIA TITAN X (PASCAL)
GPU1: NVIDIA TITAN X (PASCAL)
GPU2: NVIDIA GEFORCE GTX 980
GPU3: NVIDIA GEFORCE GTX 980
Memory: 189960 MiB DDR3
Compilers and Environment: gcc 9.3.0, Cuda 11.2

Algoritmalar `C++` ve `CUDA` dilleri kullanılarak gerçeklenmiş, `-O3`, `-fopenmp` ve `-Xcompiler` flag'leriyle derlenmiştir. Çok çekirdekli işlemci ortamı için `OpenMP` kullanılmış, birden fazla GPU cihazının aynı anda çalıştırılması için çekirdeklerde paralel olarak GPU fonksiyonları çağırılmıştır.

Kısım 3'te anlatılan `Ryser`, `SpaRyser` ve `Skipper` algoritmalarının GPU'da koşturulan bütün versiyonları için bir iplik bloğu içinde çalıştırılan iplik sayısı 256, blok sayısı 2048 olarak seçilmiştir. Bu değişkenlerden sonuçlarda `BlockDim` ve `GridDim` olarak bahsedilmiştir. Bunun yanısıra, `Ryser` ve `SpaRyser` algoritmalarının CPU üzerinde koşturulan paralel versiyonları için 16 iplik kullanılmıştır. Bu değişkenden tablolarda `Threads` olarak bahsedilmiştir. Birden fazla GPU çalıştıran algoritmalar `Ryser-SxC-Sm-MG+`, `SpaRyser-SxC-Sm-MG+` ve `SkipPer-SxC-Sm-MG+`

2 adet TITAN veya 2 adet TITAN ile 2 adet Gtx üzerinde koşturulmuştur. Buna ek olarak, hibrit düzene sahip algoritmalar için yapılan deneyler 2 adet TITAN ve 2 adet Gtx ile birlikte CPU çekirdekleri üzerinde koşturulmuştur.

Yakınsama algoritmaları için çok sayıda değişken ile testler koşturulmuştur. Bu değişkenler, grafikler üzerinde belirtilmiştir. Yakınsama algoritmalarında, Exp olarak gösterilen değişken, yapılan toplam deney sayısını ifade etmektedir. Her bir yakınsama algoritması için Exp değeri 10 bin'den 1 milyon'a kadar testler koşturulmuştur. Bunun yanısıra, ScalingGpu algoritmasının her ölçeklendirme için yapacağı yineleme sayısı STime ile belirtilmiştir.

4.2 Sentetik Matrisler Üzerinde Bulgular

Sentetik matrisler, $dim \times dim$ boyutunda bir matrisin her bir elemanının birbirinden bağımsız olarak ihtimale dayalı seçilmesiyle yaratılmıştır. Matrislerin sıfırdan farklı değerleri $[0, 5]$ aralığında rasgele olarak seçilmiştir. Koşturulan deneyler için $32, 34, 36, 38, 40$ boyutunda matrisler $0.10, 0.20, \ldots 0.80$ yoğunluklarında *int, double* ve *float* veri tiplerinde yaratılmıştır. Raporda sunulan her bir deney sonucu 5 tekrarın ortalaması olarak alınmıştır.

4.2.1 Tam Permanent Değerlerinin Hesaplanması

4.2.1.1 Yogun Matrisler Üzerinde Yapılan Deneyler

40×40 boyutlarında bir *tamsayı* matrisi için paralel CPU ve tek GPU üzerinde yapılan deneylerin sonucu Tablo 2'de sunulmuştur. Görülebileceği üzere, Ryser-Gx dışındaki bütün GPU algoritmaları 16 çekirdek ile koşturulmuş CPU algoritmasından önemli ölçüde hızlı çalışmaktadır. Ryser-Gx algoritmasının CPU algoritmasından daha yavaş olmasının nedeni, diğer GPU algoritmalarının aksine **x** değerlerini erişimi yüksek maliyetli olan GPU ana belleğinde tutmasıdır. Algoritmaların koşturulması süresince çok sık erişilen **x** değerlerinin Ryser-Rx algoritması ile *iplik yazmaçlarına* taşınmasının hesaplamanın aldığı zamanı önemli ölçüde düşürdüğü görülebilmektedir. Ancak Ryser algoritması için daha fazla optimizasyon da mümkündür. **x** değerlerinin *paylaşımlı hafıza* üzerinde tutulmasının hesaplamanın süresini bir miktar daha düşürdüğü, *hafıza birleştirme* (*hafıza birleştirme*) tekniğinin ise bu bağlamda gözle görülür bir farka sebep olmadığı görülmüştür. Ryser algoritmasının optimizasyonunda **x** değerlerini *paylaşımlı hafızada* tutmaktan sonra en büyük faydayı ise girdi matrisini *paylaşımlı hafıza* üzerinde tutmak göstermiştir. Daha detaylı bir analiz için, söz konusu algoritmaların farklı matris boyutları üzerinde karşılaştırması Şekil 6'da gösterilmiştir. Bahsedilen algoritmalar yoğun matrisler için tasarlanmış algoritmalar olduğundan, hesaplama sürelerinin matrislerin yoğunluğundan etkilenmediği de gözden kaçırılmamalıdır.

Tablo 3'te yoğun algoritmaların birden fazla GPU üzerinde çalışması ile aldıkları performans değerleri gösterilmiştir. Tablonun birinci sütununda karşılaştırmaya bir dayanak sağlamak için Ryser-SxC-Sm algoritmasının tek TITAN GPU üzerinde gösterdiği performans verilmiştir. Görülebileceği üzere, 2 adet TITAN ile çalışan Ryser-SxC-Sm-MG, hesaplama süresini neredeyse

yarıya indirmiştir. Ancak 2 adet TITAN GPU'nun yanına 2 adet Gtx eklendiğinde hesaplama süresinin arttığı görülmektedir. Bunun sebebi Ryser-SxC-Sm algoritmasının birden fazla cihazda çalıştırmak üzere yineleme uzayını statik olarak eşit parçalara bölmesidir. Bu durum Gtx GPU'lara göre ortalama 3 kat daha hızlı olan TITAN GPU'ların beklemesine yol açmakta ve hesaplama süresini uzatmaktadır. Bu durumun önüne geçmek için geliştirilmiş olan, yineleme uzayını çok sayıda küçük parçaya bölen Ryser-SxC-Sm-MG+ algoritmasının 4 GPU'yu verimli bir biçimde çalıştırıp hesaplama süresini önemli ölçüde azalttığı görülmüştür. Ancak, 4 GPU ile aynı anda 8 CPU çekirdeği çalıştırmanın performansa bir katkısı olmadığı, hatta $\%0.5$ daha yavaş olduğu görülmüştür. Bunun sebebinin GPU'lardan çok daha yavaş olan CPU çekirdeklerinin yaptıkları hesaplamanın yineleme uzayının dinamik olarak paylaşılması sonucu toplam hesaplamanın çok küçük bir kısmını teşkil etmekle beraber, haberleşme ve diğer ekstra masrafların bu hesaplamayı da gölgede bıraktığı düşünülmektedir. Birden fazla GPU kullanan algoritmaların farklı matris boyutları için performans değerleri Şekil 6'da görülebilir.

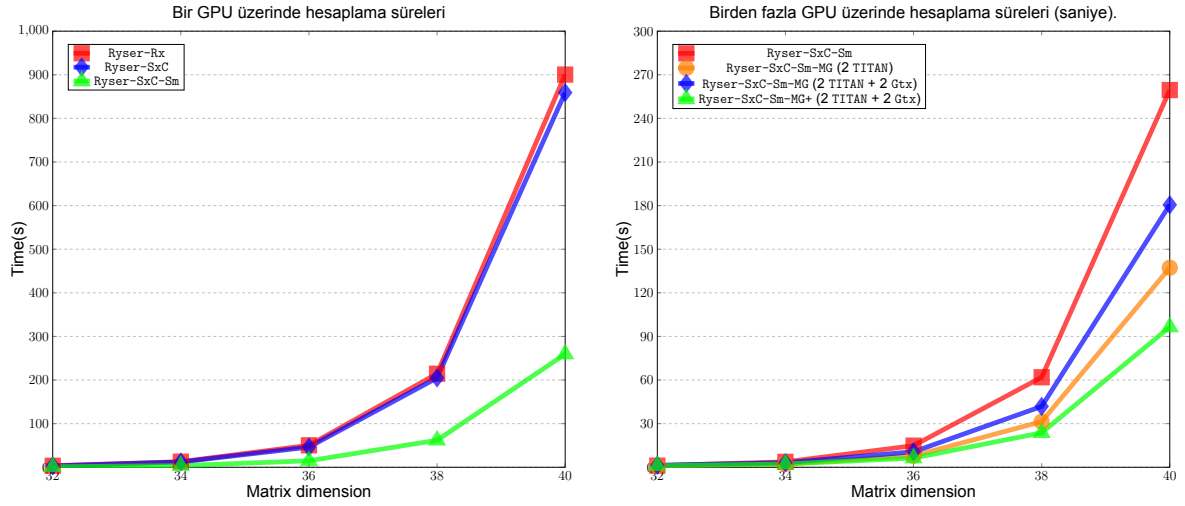
Tablo 2: 40 x 40 boyutunda farklı yoğunluklardaki matrislerin tek GPU ve CPU $(\tau = 16)$ üzerinde çalışma süreleri (saniye)

Density	ParallelRyser Threads = 16	Ryser-Gx BlockDim = 256 GridDim = 2048	Ryser-Rx BlockDim = 256 GridDim = 2048	Ryser-Sx BlockDim = 256 GridDim = 2048	Ryser-SxC BlockDim = 256 GridDim = 2048	Ryser-SxC-Sm BlockDim = 256 GridDim = 2048
0.6	2696.20	12338	900.42	862.42	859.11	259.51
0.7	2703.32	12298	897.43	857.91	854.38	259.44
0.8	2690.99	12312	900.69	858.24	858.65	259.46

Tablo 3: 40 x 40 boyutunda farklı yoğunluklardaki matrislerin birden çok GPU ve GPU+CPU (hibrit) üzerinde çalışma süreleri (saniye)

Density	Ryser-SxC-Sm BlockDim = 256 GridDim = 2048	Ryser-SxC-Sm-MG BlockDim = 256 GridDim = 2048 2 TITAN	Ryser-SxC-Sm-MG BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx	Ryser-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx	Ryser-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx 8 Threads
0.60	259.51	137.25	180.55	96.48	97.09
0.70	259.44	137.26	180.61	96.52	97.00
0.80	259.46	137.27	180.54	96.43	96.93

Şekil 6: Yoğunluğu 0.60 olan farklı boyutta matrislerin farklı algoritmalarla permanent değeri hesaplama süreleri (saniye)



4.2.1.2 Seyrek Matrisler Üzerinde Yapılan Deneyler

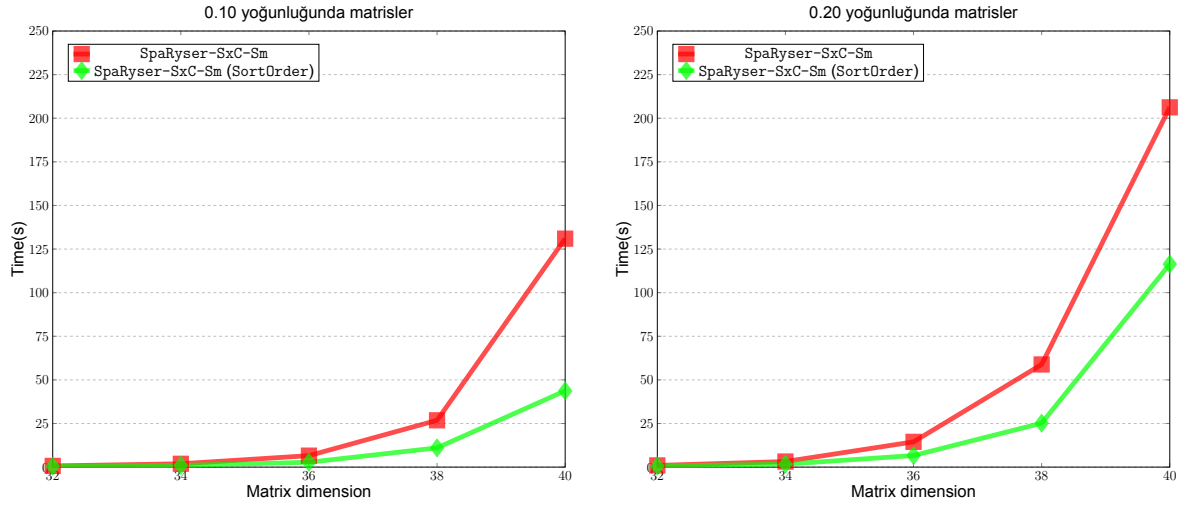
Seyrek matris deneylerinde `SpaRyser` ve `Skipper` algoritmalarının deneyleri CPU, GPU ve birden fazla GPU üzerinde test edilmiştir. Bütün algoritmaların birbiriyle karşılaştırılmasından önce, ön hesaplama basamakları olan `SortOrder` ve `SkipOrder` yöntemlerinin de `SpaRyser` ve `Skipper` algoritmalarının performansı üzerine etkisi gösterilmiştir.

Tablo 4'te 16 çekirdek ile koşturulan `ParallelSpaRyser` ile birlikte tek GPU üzerinde koşturulan `SpaRyser-SxC-Sm` ve 4 GPU üzerinde koşturulan `SpaRyser-SxC-Sm-MG+` algoritmalarının `SortOrder` işlemini yaparak ve yapmadan elde ettikleri performans değerleri gösterilmiştir. Bütün algoritmalar için bütün yoğunluklarda `SortOrder` işlemini uygulamanın performans üzerinde önemli değişime sebep olduğu görülmüştür. `SortOrder` işleminin performansa etkisinin en çok olduğu yoğunluğun 0.10 olduğu görülmüştür. Literatürde kullanılan çizgelerin bu değerden çok daha az kenar yoğunluğuna sahip olduğu düşünüldüğünde bu tür bir sıralama işleminin etkisinin oldukça önemli olduğu görülebilir. `SortOrder` işleminin 0.10 ve 0.20 yoğunluğa sahip matrislerde performansa etksi Şekil 7'de görülebilir.

Tablo 4: `SpaRyser` algoritmasının `SortOrder` kullanan ve kullanmayan versiyonlarının 40 x 40 boyutunda tamsayı matrisi üzerinde permanent hesaplama süreleri (saniye)

	With SortOrder			Without SortOrder		
	ParallelSpaRyser Threads = 16	SpaRyser-SxC-Sm BlockDim = 256 GridDim = 2048	SpaRyser-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx	ParallelSpaRyser Threads = 16	SpaRyser-SxC-Sm BlockDim = 256 GridDim = 2048	SpaRyser-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx
Density	SortOrder	SortOrder	SortOrder			
0.10	360.57	43.71	16.71	981.75	130.97	48.67
0.20	783.37	116.38	43.16	1810.66	206.25	78.15
0.30	1593.18	211.31	77.93	2424.53	355.14	131.33
0.40	2598.11	312.16	114.97	4528.42	456.84	169.49

Şekil 7: `SpaRyser-SxC-Sm` algoritmasının `SortOrder` kullanan ve kullanmayan versiyonlarının tamsayı matrisi üzerinde permanent hesaplama süreleri (saniye)



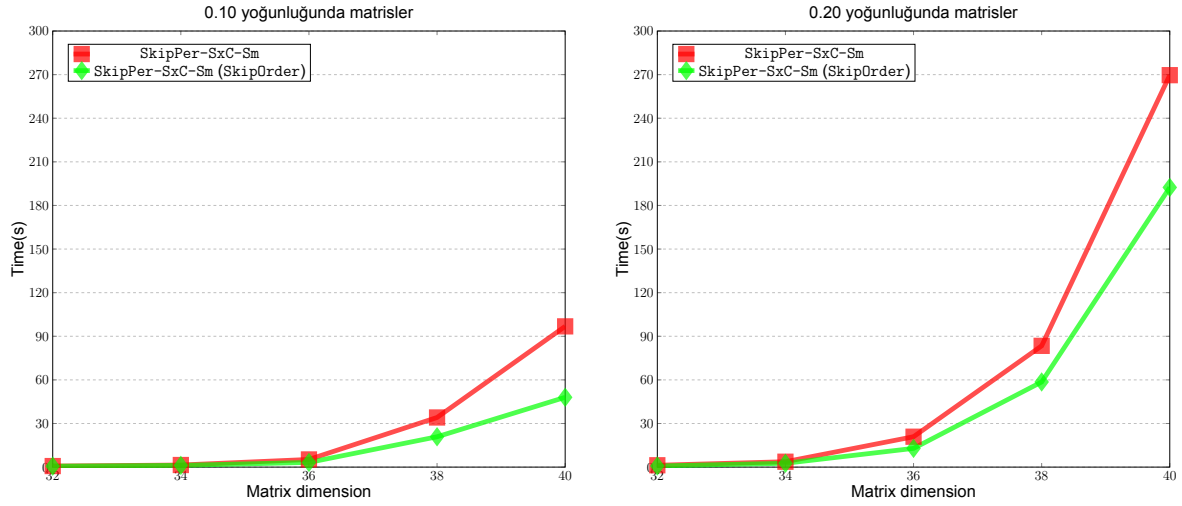
Tablo 5: `Skipper` algoritmasının `SkipOrder` kullanan ve kullanmayan versiyonlarının 40 x 40 boyutunda tamsayı matrisi üzerinde permanent hesaplama süreleri (saniye)

	With SkipOrder			Without SkipOrder		
	ParallelSkipper Threads = 16	SkipPer-SxC-Sm BlockDim = 256 GridDim = 2048	SkipPer-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx	ParallelSkipper Threads = 16	SkipPer-SxC-Sm BlockDim = 256 GridDim = 2048	SkipPer-SxC-Sm-MG+ BlockDim = 256 GridDim = 2048 2 TITAN + 2 Gtx
Density	SkipOrder	SkipOrder	SkipOrder			
0.10	261.06	48.01	22.59	440.99	96.82	28.86
0.20	717.56	192.40	76.48	799.58	269.62	117.15
0.30	1279.66	363.98	167.53	1885.42	516.54	224.65
0.40	2110.20	489.19	213.91	3786.77	603.29	242.03

`SkipOrder` ön işleminin 16 çekirdek ile koşturulan `ParallelSkipper`, bir GPU ile koşturulan `SkipPer-SxC-Sm` ve dört GPU üzerinde koşturulan `SkipPer-SxC-Sm-MG+` algoritmalarında performansa etkisi Tablo 5'te gösterilmiştir. Her üç algoritmada da `SkipOrder` ön işlemi uygulandığında performansın arttığı görülmüştür. `SortOrder` ön işleminin `SpaRyser` algoritmasının performansında yarattığı etkinin aksine, `SkipOrder` ön işlemninin performansa etkisinin algoritma ve kullanan hesaplama noktalarına göre değişebildiği görülmüştür. Örneğin, `SortOrder` ön işlemi uygulandığında `SkipPer-SxC-Sm` algoritmasında iki kata yakın performans artışı gözlenirken `SkipPer-SxC-Sm-MG+` algoritmasında yaklaşık olarak 1.25 kat performans artışı gözlenmiştir. Bunun yanısıra, matrislerin yoğunluğu arttığı zaman `SkipOrder` işleminin performansa etkisinin göreceli olarak azaldığı gözlenmiştir. `SkipOrder` işleminin 0.1 ve 0.2 yoğunluğa sahip matrisler üzerinde performansa etkisi Şekil 8'de görülebilir.

`ParallelRyser`, `ParallelSpaRyser` ve `ParallelSkipper` algoritmalarının 16 çekirdek kullanılarak, genel ve 0-1 matrisler üzerinde karşılaştırmalı performansları Tablo 6'da gösterilmiştir. $0.10, 0.20, 0.30, 0.40$ yoğunluğa sahip matrislerin hepsinde `ParallelSpaRyser` ve `ParallelSkipper` algoritmalarının `ParallelRyser` algoritmasından daha iyi sonuçlar elde ettiği görülmüştür. Bu performans artışı matrislerin yoğunluğu arttıkça azalmaktadır. Bu algoritmalar seyrek matrisler

Şekil 8: `SkipPer-SxC-Sm` algoritmasının `SkipOrder` kullanan ve kullanmayan versiyonlarının tamsayı matrisi üzerinde permanent hesaplama süreleri (saniye)



düşünülerek geliştirildiği ve seyrek veri yapıları kullandıkları için bu anlaşılabilir bir sonuçtur. Yoğunluk 0.4'e geldiği zaman performanslar arasındaki fark da azalmaktadır. Genel olarak, genel matrislerde `ParallelSkipper` algoritmasının `ParallelSpaRyser`'a göre yaklaşık $\%25$ daha iyi performans gösterdiği söylenebilir. İkili matrislerde ise, `ParallelSpaRyser` algoritmasının performansı aynı kalırken `ParallelSkipper` algoritmasının **x** değerlerine yapılacak güncellemelerin her zaman $-1, 0$ veya 1 olmasından ötürü çok daha fazla yineleme atlayarak performansta çok büyük bir geliştirme elde ettiği görülmektedir.

`Ryser-SxC-Sm`, `SpaRyser-SxC-Sm` ve `SkipPer-SxC-Sm` algoritmalarının genel ve 0-1 matrisler üzerinde karşılaştırmalı performansları Tablo 7'de gösterilmiştir. 0.30 matris yoğunluğuna kadar `SpaRyser-SxC-Sm` algoritmasının `Ryser-SxC-Sm` algoritmasından daha iyi performans gösterdiği görülmüştür. `SkipPer-SxC-Sm` algoritmasının ise 0.20 yoğunluğa kadar `Ryser-SxC-Sm` algoritmasından daha iyi performans gösterdiği görülmüştür. Daha önceki CPU sonuçlarıyla karşılaştırıldığında, GPU algoritmalarının CPU'daki eşdeğerleriyle aynı derecede performans artışı getirmediği sonucuna varılabilir. Ancak en seyrek matrislerde $(0.1, 0.2)$ hala performansta çok önemli bir artış söz konusudur. Ayrıca, CPU'da gözlemlenen durumun tam tersi olarak `SkipPer-SxC-Sm` algoritmasının genel matrislerde `SpaRyser-SxC-Sm` algoritmasına göre daha kötü performans gösterdiği görülmüştür. Bunun sebebinin bir *iplik bloğu* içerisindeki bütün *iplik*'lerin farklı sayıda yineleme atlaması ve senkronizasyon problemi yaratması olduğu düşünülmektedir. Yine de, `SkipPer-SxC-Sm` algoritmasının 0-1 matrisler için bütün yoğunluklarda `SpaRyser-SxC-Sm` algoritmasından daha iyi performans gösterdiği gözden kaçırılmamalıdır.

Tablo 6: 40 x 40 boyutunda farklı yoğunluklardaki matrislerin CPU üzerinde permanent hesaplama süreleri (saniye)

	Genel Matris			İkili Matris		
	ParallelRyser	ParallelSpaRyser	ParallelSkipper	ParallelRyser	ParallelSpaRyser	ParallelSkipper
Density	Threads = 16	Threads = 16	Threads = 16	Threads = 16	Threads = 16	Threads = 16
0.1	2685.65	360.57	261.06	2688.95	296.12	1.06
0.2	2709.29	783.37	717.56	2703.33	719.44	19.27
0.3	2703.84	1593.18	1279.66	2705.20	1682.33	155.24
0.4	2756.97	2598.11	2110.20	2741.17	2832.78	1123.64

Tablo 7: 40 x 40 boyutunda farklı yoğunluklardaki matrislerin GPU üzerinde permanent hesaplama süreleri (saniye)

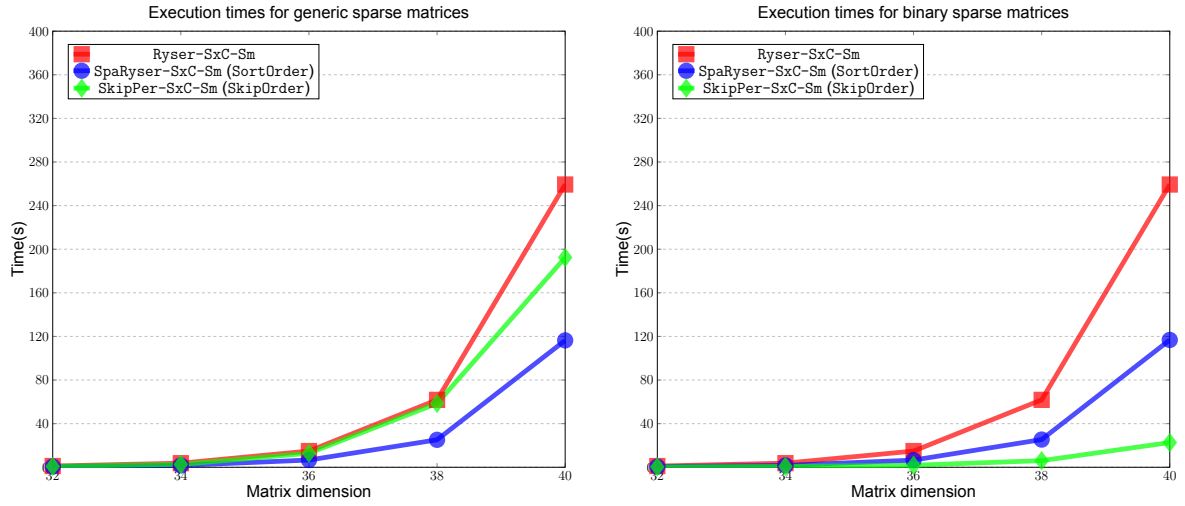
	Genel Matrisler			İkili Matrisler		
	Ryser-SxC-Sm	SpaRyser-SxC-Sm	SkipPer-SxC-Sm	Ryser-SxC-Sm	SpaRyser-SxC-Sm	SkipPer-SxC-Sm
	BlockDim = 256	BlockDim = 256	BlockDim = 256	BlockDim = 256	BlockDim = 256	BlockDim = 256
Density	GridDim = 2048	GridDim = 2048	GridDim = 2048	GridDim = 2048	GridDim = 2048	GridDim = 2048
0.10	259.34	43.71	48.01	259.29	43.05	0.94
0.20	259.35	116.38	192.40	259.33	116.79	22.68
0.30	259.35	211.31	363.98	259.37	215.14	131.23
0.40	260.83	312.16	489.19	259.35	323.94	273.65

Tablo 8'de Ryser-SxC-Sm-MG+, SpaRyser-SxC-Sm-MG+ ve SkipPer-SxC-Sm-MG+ algoritmalarının 4 GPU ile genel ve 0-1 matrisler üzerinde performansı gösterilmiştir. Sonuçlar Tablo 7 ile paralellik göstermektedir.

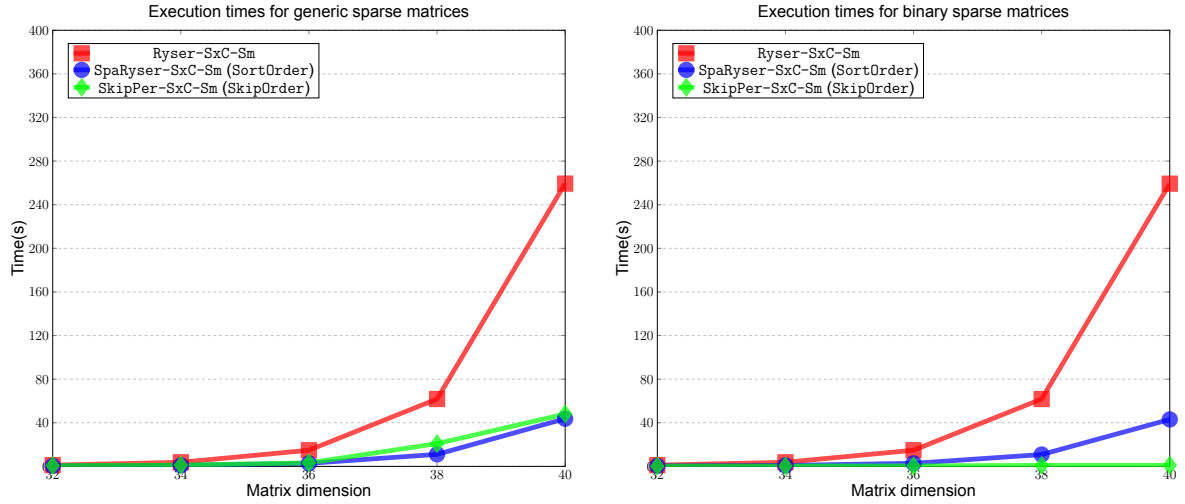
Tablo 8: 40 x 40 boyutunda farklı yoğunluklardaki matrislerin birden fazla GPU üzerinde permanent hesaplama süreleri (saniye)

	Genel Matrisler			İkili Matrisler		
	Ryser-SxC-Sm-MG+	SpaRyser-SxC-Sm-MG+	SkipPer-SxC-Sm-MG+	Ryser-SxC-Sm-MG+	SpaRyser-SxC-Sm-MG+	SkipPer-SxC-Sm-MG+
	BlockDim = 256	BlockDim = 256	BlockDim = 256	BlockDim = 256	BlockDim = 256	BlockDim = 256
	GridDim = 2048	GridDim = 2048	GridDim = 2048	GridDim = 2048	GridDim = 2048	GridDim = 2048
Density		SortOrder	SkipOrder		SortOrder	SkipOrder
0.10	96.54	16.71	22.59	96.55	16.44	1.81
0.20	96.55	43.16	76.48	96.58	43.51	9.67
0.30	96.53	77.93	167.53	96.55	78.47	64.37
0.40	96.70	114.97	213.91	96.65	119.35	106.78

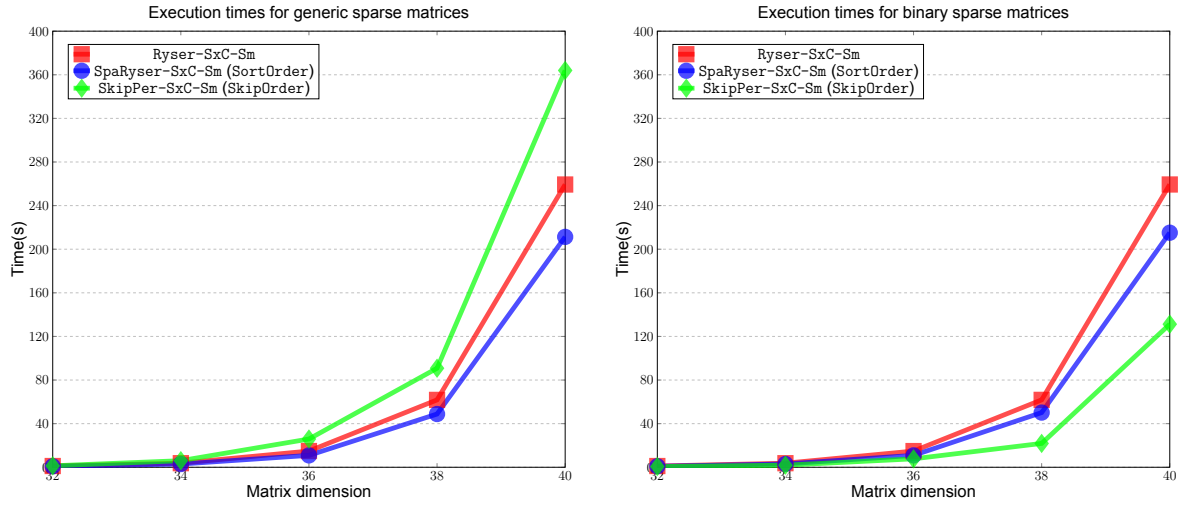
Şekil 10: Farklı algoritmaların yoğunluğu 0.20 olan farklı boyutlarda matrislerde bir GPU üzerinde permanent hesaplama süreleri (saniye)



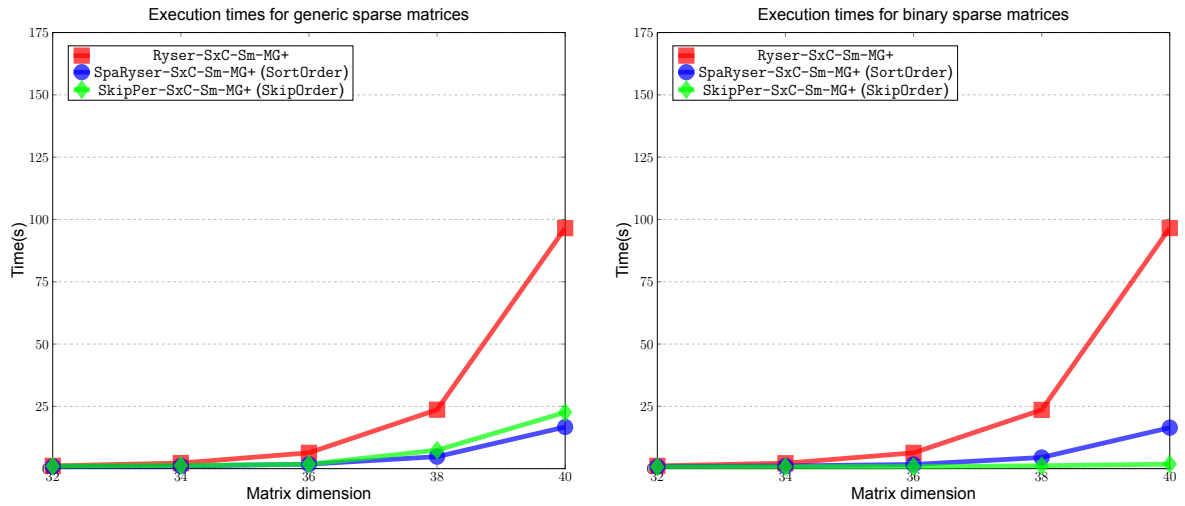
Şekil 9: Farklı algoritmaların yoğunluğu 0.10 olan farklı boyutlarda matrislerde bir GPU üzerinde permanent hesaplama süreleri (saniye)



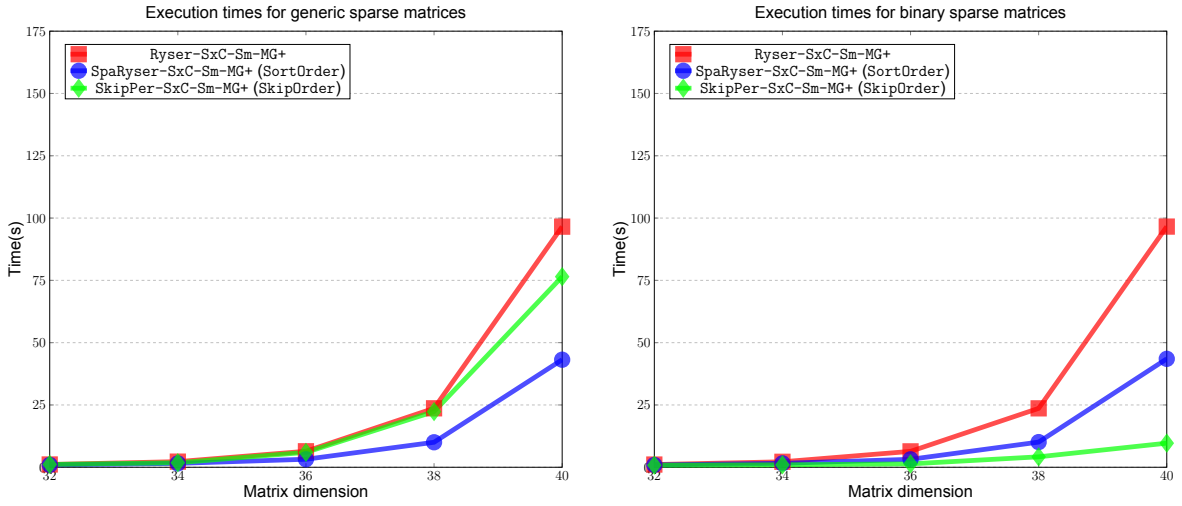
Şekil 11: Farklı algoritmaların yoğunluğu 0.30 olan farklı boyutlarda matrislerde bir GPU üzerinde permanent hesaplama süreleri (saniye)



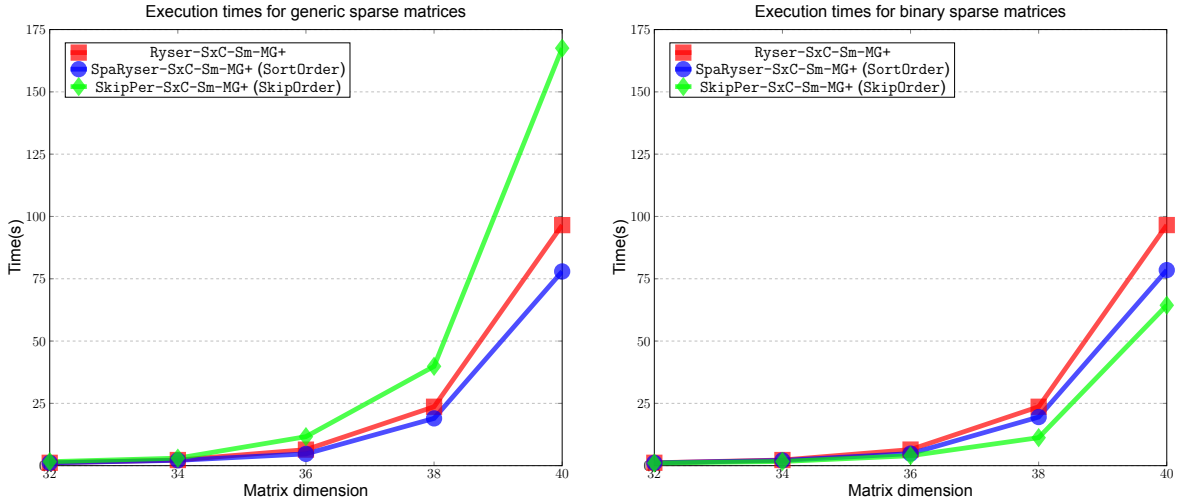
Şekil 12: Farklı algoritmaların yoğunluğu 0.10 olan farklı boyutlarda matrislerde birden fazla GPU üzerinde permanent hesaplama süreleri (saniye)



Şekil 13: Farklı algoritmaların yoğunluğu 0.20 olan farklı boyutlarda matrislerde birden fazla GPU üzerinde permanent hesaplama süreleri (saniye)



Şekil 14: Farklı algoritmaların yoğunluğu 0.30 olan farklı boyutlarda matrislerde birden fazla GPU üzerinde permanent hesaplama süreleri (saniye)



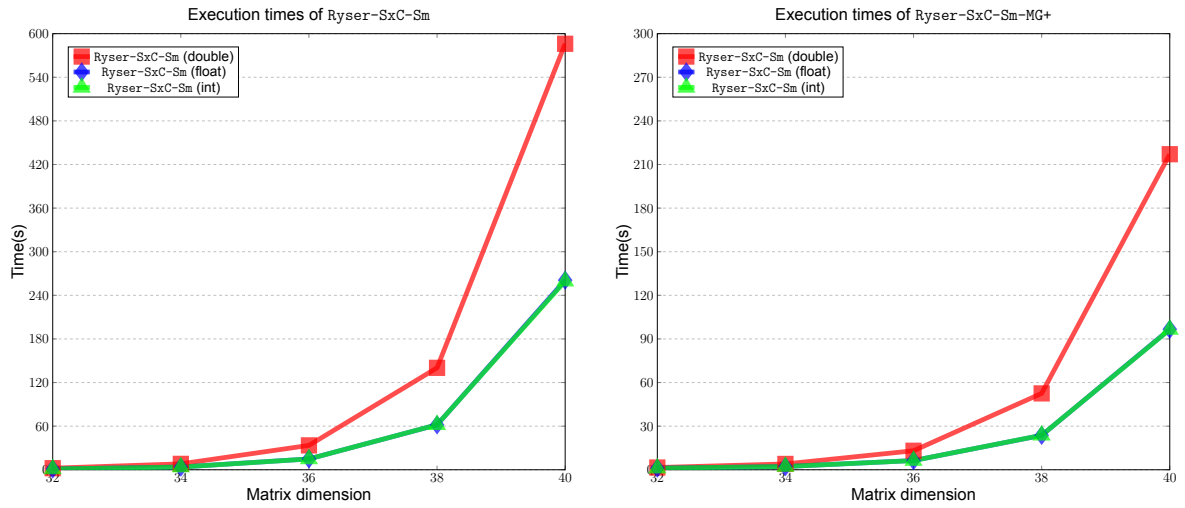
4.3 Single vs. Double Precision

`Ryser` algoritmasının türevlerinin performansları *int, float* ve *double* data tipinde eleman barındıran matrisler üzerinde ölçülmüştür. Sonuçlar Tablo 9'da gösterilmiştir. Algoritmaların hepsi yoğun matrisler için olduğundan matrislerin yoğunluğunun değişime bağlı bir performans farkı görülmemiştir. CPU üzerinde `ParallelRyser` algoritmasının veri tipi *float* olduğunda en hızlı, *double* olduğunda en yavaş süreye sahip olduğu görülmüştür. Bunun yanısıra, `Ryser-SxC-Sm` ve `Ryser-SxC-Sm-MG+` algoritmalarının *double* veri tipinde *int* ve *float* veri tiplerine göre çok daha yavaş çalıştığı görülmüştür. Bu durumun sebebi, veri tipi *double* olduğunda bu algoritmaların *paylaşımlı hafıza* kısıtlamasına takıldıkları için 256 yerine 128 *iplik bloğu* ile çalışmalarıdır.

Tablo 9: Farklı algoritmaların 40 x 40 boyutunda farklı yoğunluktaki ve farklı veri tiplerindeki matrislerde permanent hesaplama süreleri (saniye)

	int			float			double		
		Ryser-SxC-Sm			Ryser-SxC-Sm			Ryser-SxC-Sm	
Density	ParallelRyser		-MG+	ParallelRyser		-MG+	ParallelRyser		-MG+
0.60	2696.21	259.51	96.48	2132.18	260.83	96.68	2843.94	586.10	217.07
0.70	2703.32	259.44	96.52	2126.06	260.92	96.80	2854.14	586.45	217.26
0.80	2690.99	259.46	96.43	2126.69	260.08	96.52	2838.29	582.70	217.07

Şekil 15: 0.60 yoğunluğunda farklı boyut ve veri tipindeki matrislerin permanent hesaplama süreleri



4.3.1 Permanent Yakınsama Üzerinde Yapılan Deneyler

4.3.1.1 Yoğun Matrisler Üzerinde Yapılan Deneyler

Tablo 10'de `Rasmussen` türevi algoritmaların CPU ve GPU üzerinde elde ettikleri performans değerleri karşılaştırmalı olarak verilmiştir. `ParRasmussen+` algoritmasının `ParRasmussen` algoritmasına göre görece düşük performans gösterdiği gözlemlenmiştir. Bunun sebebi, `ParRasmussen+` algoritmasının, en az sıfırdan farklı sayıyı içeren satırı bulmak için matrisin üzerinde fazladan bir yineleme yapmasıdır. Bunun yanısıra, `RasmussenGpu` ve `Rasmussen+Gpu` algoritmalarının CPU eşdeğerlerinden çok daha yüksek performans gösterdikleri görülmüştür. Tablo 11'de `ParScaling`, `ParScaling+`, `ScalingGpu` ve `Scaling+Gpu` algoritmalarının karşılaştırmalı performansları sunulmuştur. GPU üzerinde çalışan algoritmaların, CPU üzerinde çalışan algoritmalara kıyasla $\texttt{SInterval} = 5$ iken yaklaşık 7.5, $\texttt{SInterval} = 1$ iken yaklaşık 2.5 kat hızlı olduğu görülmüştür. GPU algoritmalarında `SInterval` değerine bağlı olarak gözlemlenen büyük performans farkı CPU eşdeğerlerinde görülememiştir. Şekil 16'da `Rasmussen` ve `Scaling` ile, `Rasmussen+` ve `Scaling+` türevi algoritmaların karşılaştırmalı performansları gösterilmiştir.

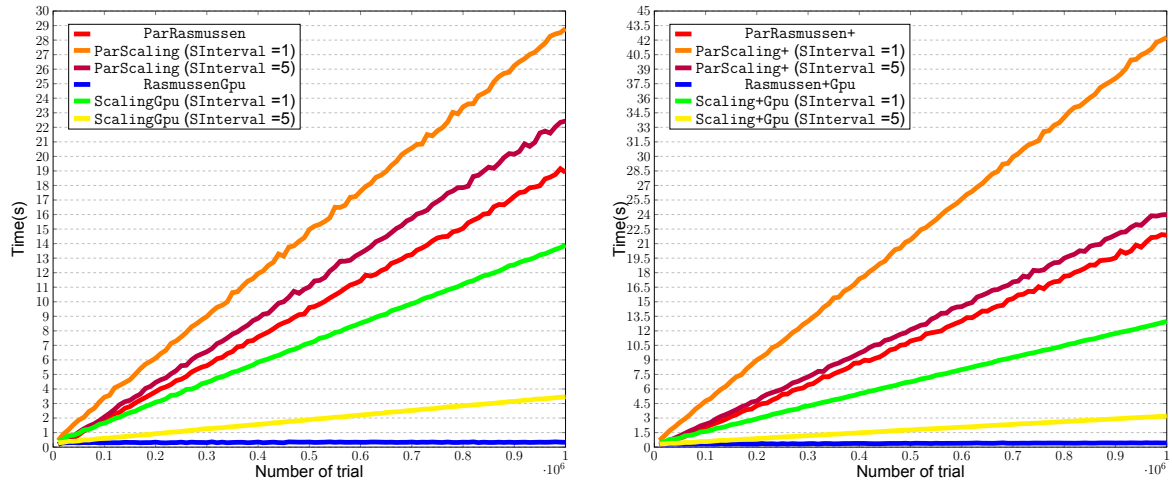
Tablo 10: `Rasmussen` türevi algoritmaların 40 x 40 boyutunda yoğun matrisler üzerinde 1 milyon deney sayısı için permanent yakınsama süreleri (saniye)

Density	ParRasmussen	ParRasmussen+	RasmussenGpu	Rasmussen+Gpu
0.8	18.89	21.86	0.32	0.43

Tablo 11: `Rasmussen` türevi algoritmaların 40 x 40 boyutunda yoğun matrisler üzerinde 1 milyon deney sayısı için permanent yakınsama süreleri (saniye)

Density	ParScaling SInterval = 5 STime = 5	ParScaling+ SInterval = 5 STime = 5	ScalingGpu SInterval = 5 STime = 5	Scaling+Gpu SInterval = 5 STime = 5	ParScaling SInterval = 1 STime = 5	ParScaling+ SInterval = 1 STime = 5	ScalingGpu SInterval = 1 STime = 5	Scaling+Gpu SInterval = 1 STime = 5
0.8	22.43	24.00	3.46	3.20	28.79	42.27	13.89	12.96

Şekil 16: Permanent yakınsama algoritmalarının 40 x 40 boyutunda yoğun matrisler üzerinde 1 milyon deney sayısı için çalışma süreleri (saniye)



4.3.1.2 Seyrek Matrisler Üzerinde Yapılan Deneyler

Tablo 12'de yoğun ve seyrek matrisler için optimize edilmiş algoritmaların farklı yoğunluğa sahip matrisler üzerinde performansları karşılaştırmalı olarak sunulmuştur. Seyrek matrisler için optimize edilmiş, seyrek veri tipi kullanan algoritmalar 'S' ekiyle belirtilmiştir. Bu tabloda bütün yoğun algoritmaların hesaplama süresinin matris yoğunluğu arttığı görülebilir. Bunun sebebi matris seyrek olduğunda deneylerin arasında daha çok 0 permanent değeri bulunması ve permanent değerini 0 bulan deneylerin daha erken sonlandırılmasıdır. Bunun yanısıra, `ParRasmussen` algoritmasında `ParRasmussen+` algoritmasına göre yoğunluk arttıkça performans kaybı daha çok artmaktadır. Bunun sebebi `Rasmussen+` algoritmasının sıfırdan farklı değer sayısı en az satırları bulması ve daha az deneyin 0 ile sonlanmasıdır. Hesaplama sürelerinin 1 saniyenin altında olmasından ötürü aynı çıkarımı `RasmussenGpu` ve `Rasmussen+Gpu` için yapmak doğru olmayacaktır ancak CPU eşdeğerlerinden çok daha hızlı çalıştıkları ve matris yoğunluğu arttıkça performans kaybı yaşamadıkları sonuçlarda gözlemlenebilmektedir.

Seyrek veri yapısı kullanan algoritmalarda, `ParRasmussen+S` algoritmasının `ParRasmussen`

algoritmasından bütün yoğunluklarda daha hızlı çalıştığı ve en büyük farkın 0.1 yoğunlukta olduğu görülebilmektedir. Bunun yanında, performans bakımından `ParRasmussen+S` algoritmasının `ParRasmussen+` algoritmasını 0.1 yopunlukta geride bıraktığı, ancak bu etkinin diğer yoğunluklarda görülmediği gözlemlenebilmektedir. Şekil 17 ve Şekil 18'de `Rasmussen` ve `Rasmussen+` türevi algoritmalarının yoğun ve seyrek versiyonlarının karşılaştırmalı performans değerleri sunulmuştur.

Tablo 12: `Rasmussen` türevi algoritmaların 40 x 40 boyutunda seyrek matrisler üzerinde 1 milyon deney sayısı için permanent yakınsama süreleri (saniye)

Density	ParRasmussen	ParRasmussen+	RasmussenGpu	Rasmussen+Gpu	ParRasmussen+S	ParRasmussen+S	RasmussenGPUS	Rasmussen+Gpu+S
0.10	6.31	20.20	0.34	0.46	3.80	19.92	0.31	0.35
0.20	13.06	21.73	0.33	0.44	7.90	21.44	0.34	0.36
0.30	16.68	22.88	0.32	0.46	10.88	22.56	0.32	0.40
0.40	18.90	22.47	0.34	0.43	12.99	22.63	0.32	0.40

Tablo 13: `Scaling` türevi algoritmaların 40 x 40 boyutunda yoğun matrisler üzerinde, `SInterval` $= 5$ değeriyle 1 milyon deney sayısı için permanent yakınsama süreleri (saniye)

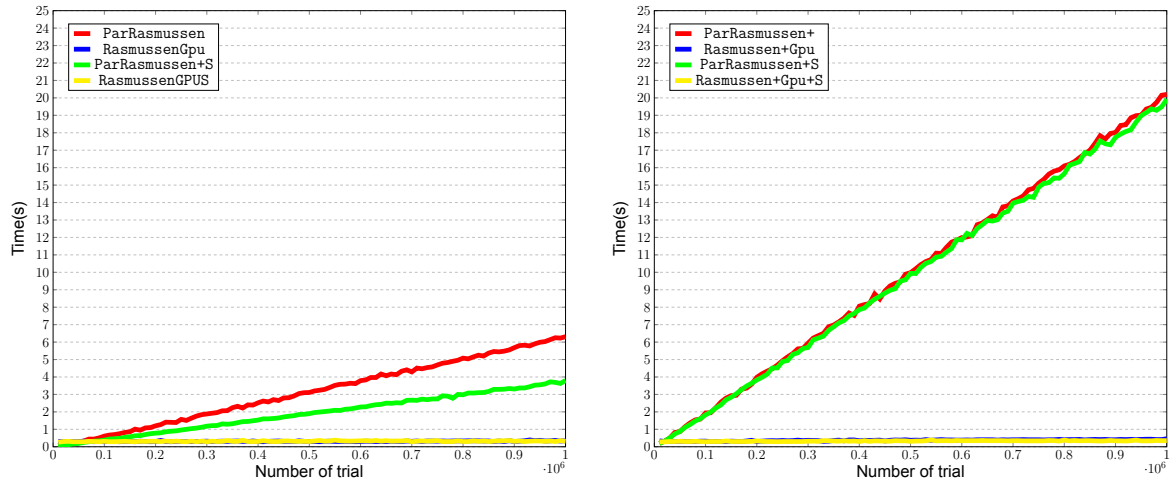
Density	ParScaling SInterval = 5 STime = 5	ParScaling+ SInterval = 5 STime = 5	ScalingGpu SInterval = 5 STime = 5	Scaling+Gpu SInterval = 5 STime = 5	ParScalingS SInterval = 5 STime = 5	ParScaling+S SInterval = 5 STime = 5	ScalingGpuS SInterval = 5 STime = 5	Scaling+GpuS SInterval = 5 STime = 5
0.10	11.56	22.86	2.40	2.99	8.76	22.67	1.36	1.83
0.20	19.16	23.54	3.16	3.08	15.34	23.90	2.08	2.15
0.30	20.94	24.16	3.27	3.12	17.62	24.25	2.30	2.38
0.40	22.53	24.00	3.33	3.14	19.26	24.28	2.52	2.59

Tablo 14: `Scaling` türevi algoritmaların 40 x 40 boyutunda yoğun matrisler üzerinde, `SInterval` $= 1$ değeriyle 1 milyon deney sayısı için permanent yakınsama süreleri (saniye)

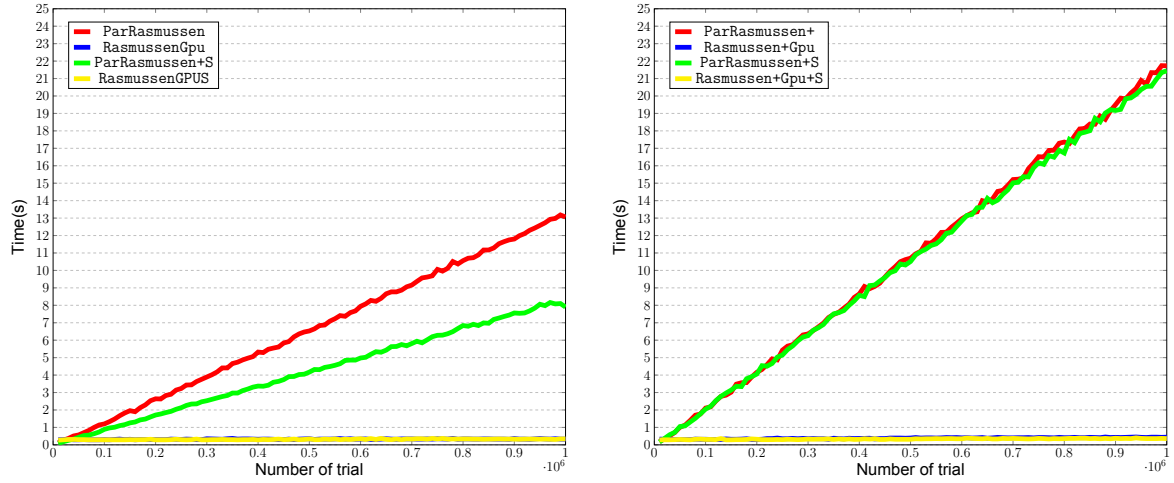
Density	ParScaling SInterval = 1 STime = 5	ParScaling+ SInterval = 1 STime = 5	ScalingGpu SInterval = 1 STime = 5	Scaling+Gpu SInterval = 1 STime = 5	ParScalingS SInterval = 1 STime = 5	ParScaling+S SInterval = 1 STime = 5	ScalingGpuS SInterval = 1 STime = 5	Scaling+GpuS SInterval = 1 STime = 5
0.10	23.36	42.87	11.48	12.50	16.53	23.85	5.89	7.60
0.20	28.00	42.20	13.46	12.76	22.21	25.26	8.76	9.21
0.30	28.43	42.79	13.71	12.89	23.45	30.52	9.44	9.93
0.40	29.17	42.64	13.77	12.88	24.22	37.60	10.21	10.63

Tablo 13 ve Tablo 14'te `Scaling` ve `ScalingGpu` algoritmalarının `SInterval` değişkenine göre performans değerleri sunulmuştur. İki değer için de yoğun ve seyrek algoritmaların kendi içinde benzer performans ortaya koydukları görülmüştür. `ParScaling` ve `ScalingGpu` algoritmalarının yoğunluk arttıkça performans düşüşü gösterdiği, ancak `ParScaling+` ve `Scaling+Gpu` algoritmalarının kullandıkları optimizasyondan ötürü bu düşüşü göstermediği görülebilir. Bunun yanında `SInterval` değeri 5 iken, `ParScalingS` algoritmasının her yoğunlukta ve en fazla 0.10 yoğunluğunda `ParScaling` algoritmasından daha iyi performans gösterdiği görülebilir. Bunun yanısıra seyrek algoritmaların GPU türevleri olan `ScalingGpuS` ve `Scaling+GpuS` algoritmalarının da her yoğunlukta yoğun eşdeğerlerinden daha iyi performans sergilediği görülmüştür. Bu algoritmalar da CPU eşdeğerleri gibi, en büyük performans artışını 0.10 yoğunlukta göstermiştir. Şekil 19, 20, 21 ve 22 yoğun ve seyrek veri yapısı kullanan `Scaling` ve `Scaling+` türevi algoritmaların farklı yoğunluk ve `SInterval` değerleri için performans değerlerini göstermektedir.

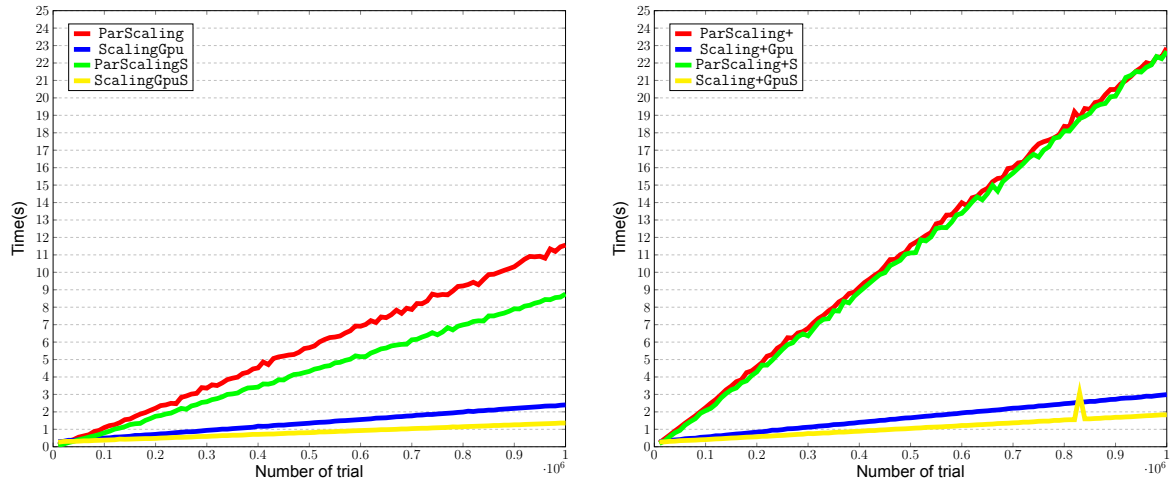
Şekil 17: `Rasmussen` türevi algoritmaların 40 x 40 boyutunda 0.10 yoğunluğunda matrisler üzerinde 1 milyon deney sayısı için permanent yakınsama süreleri (saniye)



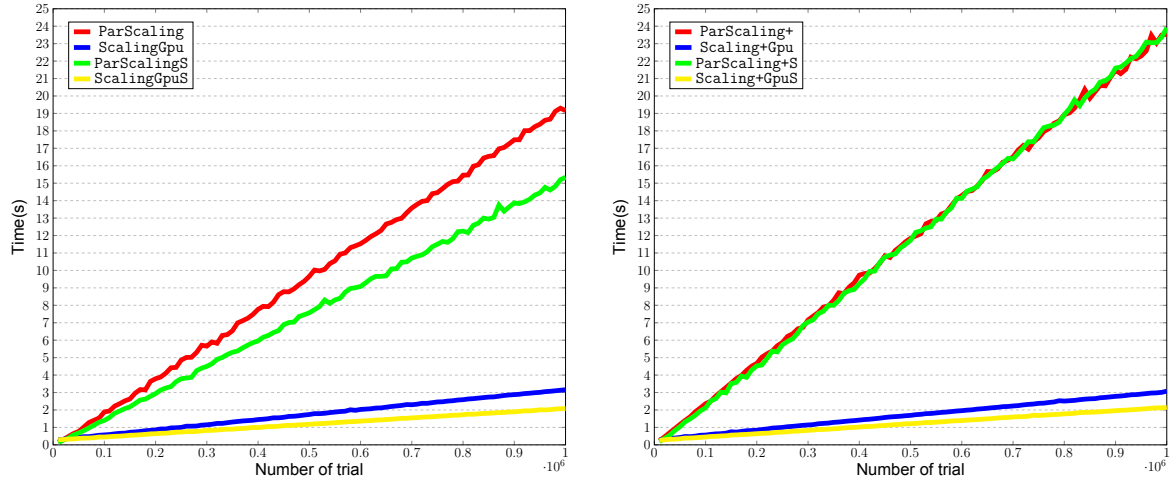
Şekil 18: `Rasmussen` türevi algoritmaların 40 x 40 boyutunda 0.20 yoğunluğunda matrisler üzerinde 1 milyon deney sayısı için permanent yakınsama süreleri (saniye)



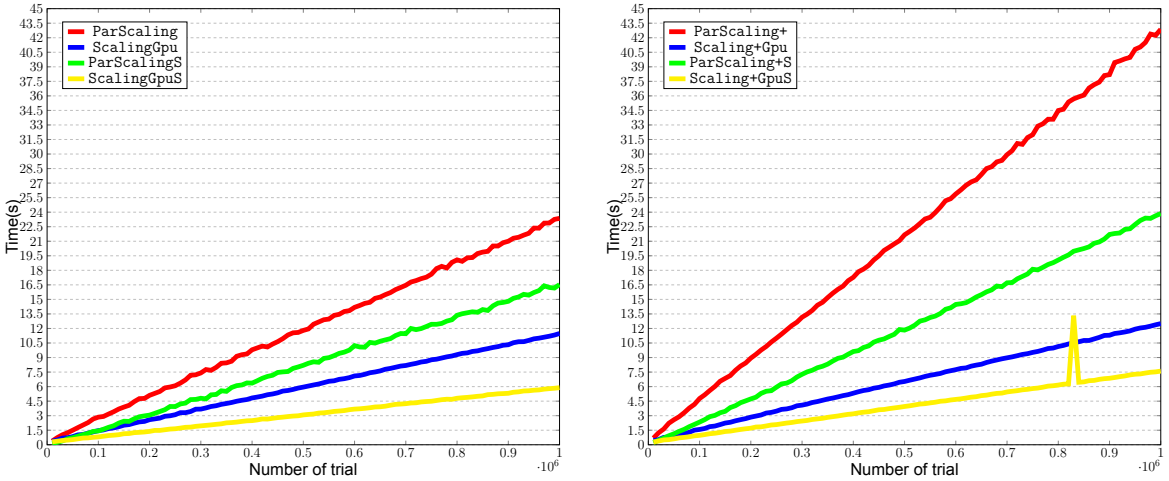
Şekil 19: `Scaling` türevi algoritmaların 40 x 40 boyutunda, yoğunluğu 0.10 olan matrisler üzerinde, `SInterval` $= 5$ değeriyle çalışma süreleri (saniye)



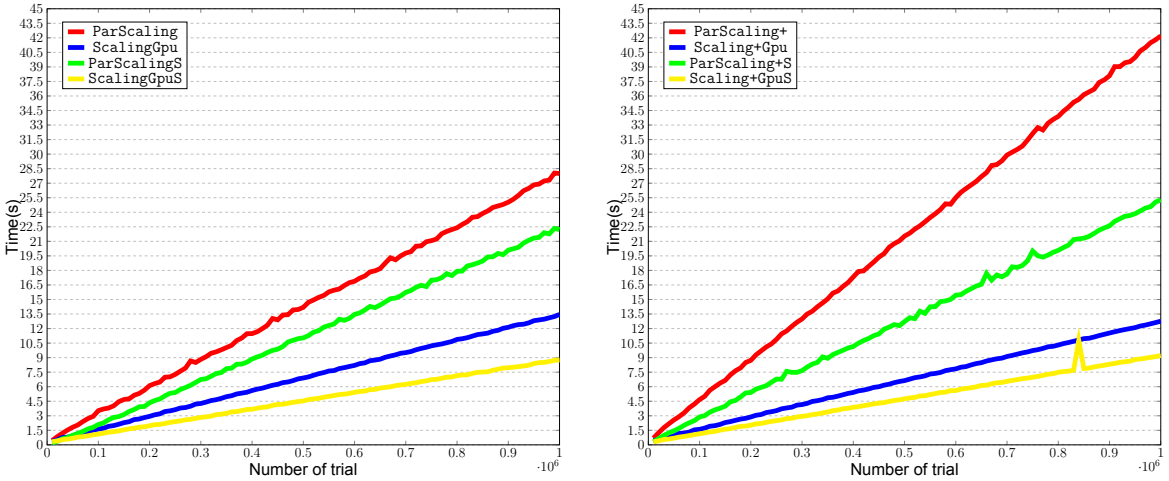
Şekil 20: `Scaling` türevi algoritmaların 40 x 40 boyutunda, yoğunluğu 0.20 olan matrisler üzerinde, `SInterval` $= 5$ değeriyle çalışma süreleri (saniye)



Şekil 21: `Scaling` türevi algoritmaların 40 x 40 boyutunda, yoğunluğu 0.10 olan matrisler üzerinde, `SInterval` $= 1$ değeriyle çalışma süreleri (saniye)



Şekil 22: `Scaling` türevi algoritmaların 40 x 40 boyutunda, yoğunluğu 0.20 olan matrisler üzerinde, `SInterval` $= 1$ değeriyle çalışma süreleri (saniye)

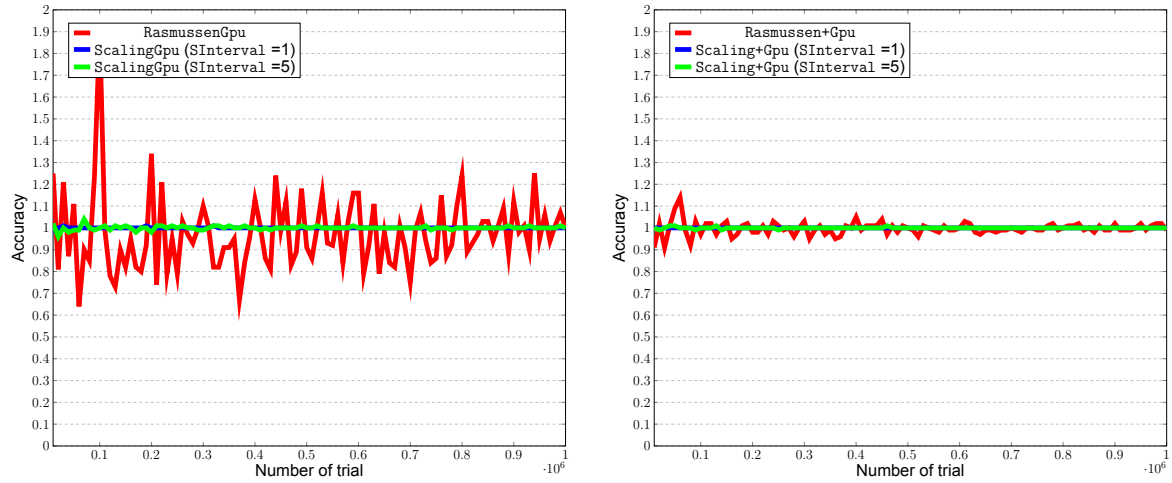


4.3.1.3 Permanent Yakınsama Algoritmalarının İsabeti Üzerine Deneyler

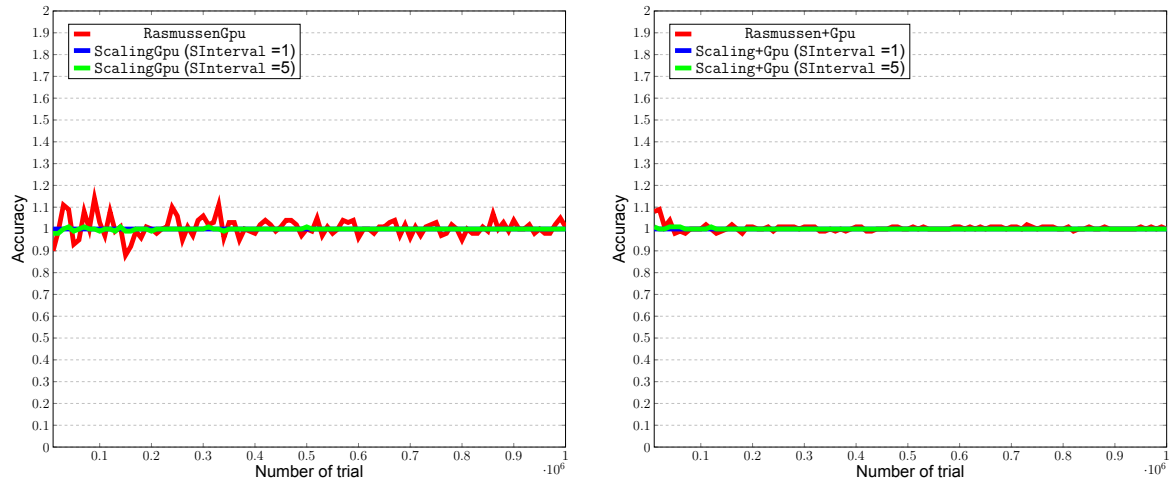
Bu bölümde, önceki bölümde performans sonuçlarından bahsedilen `Rasmussen`, `Rasmussen+`, `Scaling` ve `Scaling+` türevi algoritmaların yakınsamalarının isabet oranları sunulmuştur. İsabet oranı, tahmin edilen permanent ve gerçek permanent oranlanarak bulunmuştur. Şekil 23, 24 ve 25'te 0.20, 0.30 ve 0.40 yoğunluğundaki matrisler için algoritmaların isabetlilik oranları görülebilir. Elde edilen sonuçlar ışığında, bütün yoğunluk değerleri için bütün yakınsama algoritması türevleri içerisinde en iyi sonuçları `SInterval` değeri 1 olan `Scaling` algoritmasının verdiği söylenebilir. Bunun yanısıra `Scaling` türevi algoritmaların `SInterval` değeri 5 olduğunda da `Rasmussen` türevi algoritmalardan daha isabetli tahminler yaptığı görülebilir. Bu sonuçtan, ölçeklemenin tahminin isabetliliğini arttırma konusunda işe yaradığı anlaşılmaktadır. Bunun yanında hem `Scaling+` ve `Rasmussen+` algoritmalarının seleflerine göre daha isabetli yakınsamalar yaptıkları da görülmektedir.

Şekil 23'e bakıldığında, `Rasmussen+Gpu` algoritmasının `RasmussenGpu` algoritmasına göre çok daha az salınım yaptığı görülecektir. Aynı şekilde `ScalingGpu` algoritması da `Scaling` algoritmasına kıyasla daha fazla salınım yapmaktadır ancak bu salınım `RasmussenGpu` kadar büyük değildir. `SInterval` değeri 1'e eşitlendiği zaman ise, salınım neredeyse tamamen bitip grafik düz bir çizgi formunu almaktadır. Ayrıca, yoğunluk arttıkça bütün algoritmaların isabetliliği de artmaktadır.

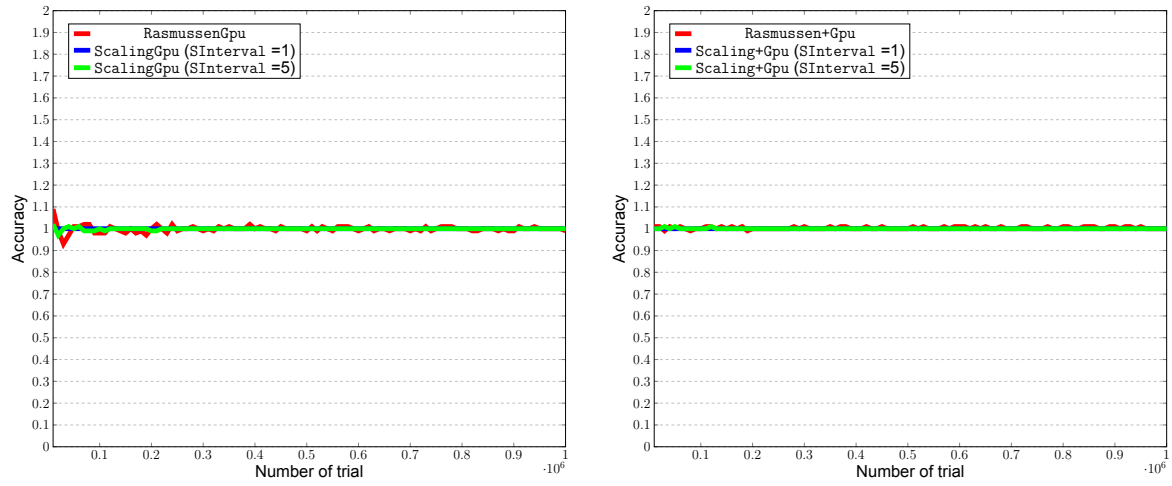
Şekil 23: Yoğunluğu 0.20 olan farklı boyuttaki matrisler üzerinde yakınsama algoritmalarının isabetlilikleri



Şekil 24: Yoğunluğu 0.30 olan farklı boyuttaki matrisler üzerinde yakınsama algoritmalarının isabetlilikleri



Şekil 25: Yoğunluğu 0.40 olan farklı boyuttaki matrisler üzerinde yakınsama algoritmalarının sonuç kalitesi



5 Tartışma ve Sonuç

Bu projede P#-tam bir problem olan permanent hesaplama problemi için çok çekirdekli, donanımsal hızlandırıcı kullanan, birden çok donanımsal hızlandırıcı üzerinde aynı anda çalışabilen, böylece daha önce hesaplanması imkansız veya çok uzun süre gerektiren hesaplamaları kabul edilebilir bir zamana indiren açık kaynaklı **SUPerman** aracı geliştirilmiştir. Problemi donanımsal hızlandırıcılara taşırken ortaya çıkan problemler çözülmüş ve optimizasyon fırsatları araştırılmış, uygulanan yöntemler detaylı olarak sunulmuştur. Donanımsal hızlandırıcıların permanent hesaplama problemini başarıyla hızlandırdığı görülmüştür. Donanımsal hızlandırıcıların performans kazancından istifade ederek yakınsama algoritmalarında performans-kalite takasının mantıklı zaman sınırlamaları içinde yapılabileceği kanıtlanmıştır. Proje kapsamında geliştirilen algoritmalar, daha detaylı optimizasyon teknikleri ve dağıtık hesaplama marifetiyle, bu raporda kullanılandan çok daha büyük matrislerin permanentlerini hesaplamak amacıyla bilimsel hesaplamalarda kullanılabilir. Permanent hesaplama, giriş bölümünde bahsedildiği gibi bilimin pek çok dalında faydalanılan bir yöntem olduğu için sunulan bulguların ve aracın diğer araştırmacılara olumlu katkıda bulunacağı öngörülmektedir.

Kaynakça

Aaronson, S. ve Arkhipov, A. 2011. “The computational complexity of linear optics”. In Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing, STOC ’11, sf. 333–342, New York, NY, USA. ACM.

Allen-Zhu, Z., Li, Y., Oliveira, R., ve Wigderson, A. 2017. “Much faster algorithms for matrix scaling”, arXiv preprint arXiv:1704.02315.

Anari, N., Gurvits, K., Gharan, S. O., ve Saberi, A. 2017. “Simply exponential approximation of the permanent of positive semidefinite matrices”. In 58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017, sf. 914–925.

Balakrishnan, N. 2007. “Permanents, order statistics, outliers, and robustness”, Revista Matematica Complutense, 2007, 20.

Björklund, A., Gupt, B., ve Quesada, N. 2019. “A faster hafnian formula for complex matrices and its benchmarking on a supercomputer”, J. Exp. Algorithmics, 24(1):1.11:1–1.11:17.

Chakrabarty, D. ve Khanna, S. 2018. “Better and simpler error analysis of the Sinkhorn-Knopp algorithm for matrix scaling”, arXiv preprint arXiv:1801.02790.

Cohen, M. B., Madry, A., Tsipras, D., ve Vladu, A. 2017. “Matrix scaling and balancing via box constrained Newton’s method and interior point methods”, arXiv preprint arXiv:1704.02310.

Dufossé, F., Kaya, K., Panagiotas, I., ve Uçar, B. 2018. “Approximation algorithms for maximum matchings in undirected graphs”. In Proceedings of the 8th SIAM Workshop on Combinatorial Scientific Computing.

Dufossé, F., Kaya, K., ve Uçar, B. 2015. “Two approximation algorithms for bipartite matching on multicore architectures”, J. Parallel Distr. Com., 85:62–78.

Dufossé, F., Kaya, K., Panagiotas, I., ve Uçar, B. 2020. “Scaling matrices and counting the perfect matchings in graphs”, Discrete Applied Mathematics.

Dulmage, A. L. ve Mendelsohn, N. S. 1958. “Coverings of bipartite graphs”, Canad. J. Math., 10:517–534.

Eldar, L. ve Mehraban, S. 2018. “Approximating the permanent of a random matrix with vanishing mean”. In 59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018, sf. 23–34.

Forbert, H. ve Marx, D. 2003. “Calculation of the permanent of a sparse positive matrix”, Computer Physics Communications, 150(3):267 – 273.

Gurvits, L. ve Samorodnitsky, A. 2014. “Bounds on the permanent and some applications”, arXiv preprint arXiv:1408.0976.

Idel, M. 2016. “A review of matrix scaling and Sinkhorn’s normal form for matrices and positive maps”, arXiv preprint arXiv:1609.06349.

Jerrum, M., Sinclair, A., ve Vigoda, E. 2004. “A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries”, J. ACM, 51(4):671–697.

Kalantari, B. ve Khachiyan, L. 1996. “On the complexity of nonnegative-matrix scaling”, Linear Algebra Appl., 240:87–103.

Karmarkar, N., Karp, R., Lipton, R., László, L., ve Luby, M. 1993. “A monte-carlo algorithm for estimating the permanent”, SIAM J. Comput., 22:284–293.

Kaya, K. 2019. “Parallel algorithms for computing sparse matrix permanents”, Turkish Journal of Electrical Engineering & Computer Sciences.

Kılıç, E. ve Tasçı, D. 2007. “On the permanents of some tridiagonal matrices with applications to the Fibonacci and Lucas numbers”, Rocky Mountain J. Math., 37:1953–1969.

Knight, P. A. ve Ruiz, D. 2013. “A fast algorithm for matrix balancing”, IMA J. Numer. Anal., 33(3):1029–1047.

Knight, P. A., Ruiz, D., ve Uçar, B. 2014. “A symmetry preserving algorithm for matrix scaling”, SIAM J. Matrix Anal. A., 35(3):931–955.

Liang, H., Huang, S., ve Bai, F. 2006. “A hybrid algorithm for computing permanents of sparse matrices”, Applied Mathematics and Computation, 172(2):708 – 716. Special issue for The Beijing-HK Scientific Computing Meetings.

Linial, N., Samorodnitsky, A., ve Wigderson, A. 2000. “A deterministic strongly polynomial algorithm for matrix scaling and approximate permanents”, Combinatorica, 20:545–568.

Merschen, J. 2011. “Nash equilibria, gale strings, and perfect matchings”. PhD thesis, The London School of Economics and Political Science (LSE).

Mittal, R. ve Al-Kurdi, A. 2001. “Efficient computation of the permanent of a sparse matrix”, International Journal of Computer Mathematics, 77(2):189–199.

Narahara, M., Tamaki, K., ve Yamada, R. 2013. “Application of permanents of square matrices for DNA identification in multiple-fatality cases”, BMC Genetics, 14(1):72.

Nijenhuis, A. ve Wilf, H. S. 1978. “Combinatorial Algorithms”. Academic Press.

Pothen, A. ve Fan, C.-J. 1990. “Computing the block triangular form of a sparse matrix”, ACM T. Math. Software, 16:303–324.

Ryser, H. 1963. “Combinatorial Mathematics”. Mathematical Association of America.

Sinkhorn, R. ve Knopp, P. 1967. “Concerning nonnegative matrices and doubly stochastic matrices”, Pacific J. Math., 21:343–348.

Štefankovič, D., Vigoda, E., ve Wilmes, J. 2017. “On counting perfect matchings in general graphs”, arXiv preprint arXiv:1712.07504.

Valiant, L. G. 1979. “The complexity of computing the permanent”, Theor. Comput. Sci., 8(2):189–201.

Wu, J., Liu, Y., Zhang, B., Jin, X., Wang, Y., Wang, H., ve Yang, X. 2018. “A benchmark test of boson sampling on Tianhe-2 supercomputer”, National Science Review, 5(5):715–720.

Yue, B., Liang, H., ve Bai, F. 2013. “Improved algorithms for permanent and permanental polynomial of sparse graph”, MATCH, 69:831–842.

TÜBİTAK
PROJE ÖZET BİLGİ FORMU

Proje Yürütücüsü:	Dr. Öğr. Üyesi KAMER KAYA
Proje No:	119E544
Proje Başlığı:	Yüksek Başarımlı, Paralel Algoritmalar ile Permanent Hesaplamaları
Proje Türü:	1002 - Hızlı Destek
Proje Süresi:	12
Araştırmacılar:	
Danışmanlar:	
Projenin Yürütüldüğü Kuruluş ve Adresi:	SABANCI Ü.
Projenin Başlangıç ve Bitiş Tarihleri:	01/04/2020 - 01/04/2021
Onaylanan Bütçe:	42000.0
Harcanan Bütçe:	19600.0
Öz:	Permanent, determinant gibi matris karakteristiklerini anlamaya yarayan o?nemli bir sayısal deg?erdir ve pratikte birc?ok uygulaması vardır. O?rneg?in, c?izgeler ve seyrek matrisler yapısal olarak birbirlerine benzedig?in-den, aynı verinin farklı go?sterimi ic?in kullanılabilirler. 0?1 deg?erlerinden olus?an simetrik bir matrisin permanent deg?eri, ona kars?ılık gelen iki parc?alı c?izge u?zerindeki mu?kemmel es?lemelerin sayısına es?ittir. Bu da c?izge noktaları arasındaki bag?lantıların yapısı hakkında o?nemli bir bilgidir. Benzer bir ilis?ki olarak yo?nsu?z, sıradan c?izgeler u?zerindeki tu?m noktaları kapsayan ayrık do?ngu? ku?melerinin sayısı ile, c?izgenin es?lenik matrisinin permanent deg?erinin es?itlig?i go?sterilebilir. Literatu?rde matris permanentlerini arac? olarak kullanan, Matematik, Bilgisayar Bilimleri, I?statistik ve Fizik gibi farklı alanlardan pek c?ok uygulama bulunmaktadır. Permanentler, Fibonacci ve Lucas sayıları ve sıralama istatistikleri ile birebir ilis?kilidir. Bunun yanında kuantum hesaplama ic?in benzetim yo?ntemlerinde ve biyoenformatik uygulamalarında kullanılmaktadırlar. Ayrıca oyun teorisindeki kusursuz bag?ımsız ayrık ku?melerin sayısı ile Gale oyunlarındaki Nash Dengesi arasında o?nemli bir ilis?ki bulunmaktadır. Matris permanentinin tam deg?erini hesaplama #P-tam bir problemdir. Bu nedenle problemin polinom zamanda c?alıs?an bir algoritması bulunmamaktadır. Literatu?rdeki en hızlı, n × n bir matrisin permanent deg?erini tam olarak hesaplayan RYSER algoritmasının karmas?ıklıg?ı üsseldir. Bu da n ? 40 gibi ufak sayılabilecek matrislerde bile hesaplamayı oldukc?a yavas?latmaktadır. Literatu?rde tam permanent deg?erini paralel bir s?ekilde tek bilgisayarda ya da su?perbilgisayarlar u?zerinde hesaplama u?zerine c?es?itli c?alıs?malar bulunmaktadır. Üssel karmaşıklık dolayısıyla permanent yakınsama algoritmaları u?zerinde oldukc?a fazla sayıda makale bulunmaktadır. Fakat özellikle GPU üzerinde paralel algoritma geliştirilmesi konusunda literatürde eksikler vardır. Bu projede hem CPU, hem de GPU üzerinde aynı anda çalıs?abilen paralel algoritmalar gelis?tirilmiş ve permanent deg?erlerinin hem tam, hem de yaklas?ık deg?erlerinin hesaplanması üzerine

TÜBİTAK

Abstract:	Permanent, such as determinant, is an important numerical value for understanding matrix characteristics and has many applications in practice. For example, because graphs and sparse matrices are structurally similar, they can be used for different representations of the same data. The permanent value of a symmetric matrix consisting of 0?1 values is equal to the number of perfect matchings on the corresponding bipartite graph. This information is important to understand the structure of connections among the graph vertices. A similar relation can be shown as the equality of the permanent value of the conjugate matrix of the graph with the number of disjoint sets of cycles covering all points on the undirected, ordinary graphs. There are many applications in the literature that use matrix permanents as tools from different fields such as Mathematics, Computer Science, Statistics and Physics. Permanents are directly related to Fibonacci and Lucas numbers and ranking statistics. In addition, they are used in simulation methods for quantum computing and bioinformatics applications. There is also an important relationship between the number of perfectly independent discrete sets in game theory and the Nash Equilibrium in Gale games. Calculating the exact value of the matrix permanent is a #P-complete problem. Therefore, the problem does not have an algorithm running in polynomial time. The complexity of the RYSER algorithm, which is the fastest algorithm to calculate the exact permanent value of an n × n matrix, is exponential. This makes the computation very slow even for matrices that can be considered as small as n ? 40. In the literature, there are various studies on calculating the full permanent value in parallel on a single computer or on supercomputers. Due to exponential complexity, there are also quite a number of articles on permanent approximation algorithms. However, there are deficiencies in the literature, especially on the development of parallel algorithms on the GPU. In this project, parallel algorithms that can run on both the CPU and the GPU simultaneously were developed for the calculation of both exact and approximate permanent values.
Anahtar Kelimeler:	Permanent, seyrek matrisler, paralel algoritmalar, CPU, GPU, yakınsama algoritmaları
Fikri Ürün Bildirim Formu Sunuldu Mu?:	Hayır

TÜBİTAK