AJIT-e: Online Academic Journal of Information Technology
2013 Spring/Bahar – Cilt/Vol: 4 - Sayı/Num: 12
DOI: 10.5824/1309-1581.2013.2.002.x
http://www.ajit-e.org/?menu=pages&p=details_of_article&id=74

GPU Hızlandırmalı Veri Demetleme Algoritmalarının İncelenmesi

Survey of GPU Accelerated Data Clustering Algorithms

Nazire Merve ÇETİN, Gazi Üniversitesi, Bilgisayar Mühendisliği Anabilim Dalı, merve_cetin_89@hotmail.com

Dr.Murat HACIÖMEROĞLU, Gazi Üniversitesi, Bilgisayar Mühendisliği Bölümü, murath@gazi.edu.tr

ÖZET *Veri demetleme algoritmaları, arama; spam, saldırı tespiti; hücre, gen, doküman analizi; moleküler dinamik simülasyonlarının biçimlerinin analizi gibi uygulamalar için oldukça önemlidirler. Veri demetleme algoritmaları için birçok araç geliştirilmiştir; ancak günümüzde teknolojinin hızla gelişmesiyle toplanan veri miktarı git gide artmaktadır. Veri miktarının artması, analizin neticesini olumlu etkilese de mevcut veri demetleme araçları, büyük-ölçekli veri kümeleriyle çalışan uygulamaların gereksinimlerini hız bakımından karşılayamaz hale gelmişlerdir. Veri demetlemede hızın rolü, veri madenciliği araştırma topluluğunun bir süredir ilgi alanındadır. Araştırmacılar, çeşitli optimizasyon tekniklerinden, veri yapısı tasarımlarından, CPU'da paralelleştirme tekniklerinden ve PC küme sistemi kullanımı gibi yöntemlerden yararlanmaktadırlar. Fakat son zamanlarda düşük maliyet ile yüksek performans sunan yeni bir yaklaşım tüm ilgiyi üzerine çekmiştir: Genel Amaçlı GPU Programlama (GPGPU). GPU'ların yüksek paralel hesaplama gücü ve grafik kartlarındaki gelişimin CPU'ya oranla daha hızlı hızlanması, aslında grafik canlandırma ve oyunlar için yoğun matematiksel hesaplamalar yapmak üzere tasarlanan grafik kartlarından genel amaçlı programlar için de yararlanmayı söz konusu hale getirmiştir. Bu makalede, GPGPU yaklaşımıyla veri demetleme algoritmalarının performansını artıran çalışmalar incelenmiş, özetlenmiş, avantajlarından ve eksik yanlarından bahsedilmiştir. Sonuç olarak, bu yaklaşımının üstünlüğü göz önünde bulundurularak konuyla ilgili bilime katkı sağlanabilecek açık alanlar verilmiş ve incelenen çalışmalardan elde edilen GPGPU yaklaşımıyla uygulama geliştirirken dikkat edilmesi gereken hususlar ortaya konulmuştur.*

Anahtar Kelimeler: *CUDA; hızlandırma için GPU; genel amaçlı gpu programlama (GPGPU); grafik işlemci birimi (GPU); paralel hesaplama; veri demetleme*

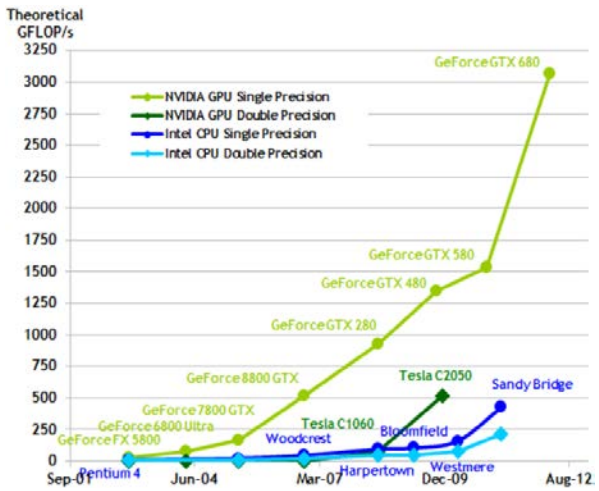
AJIT-e: Online Academic Journal of Information Technology
2013 Spring/Bahar – Cilt/Vol: 4 - Sayı/Num: 12
DOI: 10.5824/1309-1581.2013.2.002.x

ABSTRACT *Data clustering algorithms are quite important for applications such as search; spam, attack detection; cell, gene, document analysis; analysis of conformations of molecular dynamics simulations. Many tools are developed for data clustering algorithms. However, today technology is improving rapidly so that collected data amount grows more and more. Although increased data amount affects the result of analysis positively, when current data clustering tools work with large scale datasets, they don't meet the requirements of such that applications in terms of speed. Data mining research community is interested in the rol of speed on data clustering for a while. Researchers take advantage of methods such as various optimization techniques, data structure designs, parallel techniques on CPU, using PC cluster systems. However, recently a new approach which offers low cost and high performance, attracts all attention: General Purpose GPU Programming: (GPGPU). Through high parallel computing power of GPUs and more rapid development of graphics carts than CPUs, it has become to benefit graphics carts, which design to do intensive mathematical computations, for general purpose programs. In this paper, we investigate works that increase performance of data clustering algorithms with GPGPU approach, summarize them, mention advantages and disadvantages of these works. In conclusion, considering the advantages of this approach, prospected areas in this matter that could contribute to the science are given and particular points in developing the application by GPGPU approach were exhibited from the outcomes of verified practices.*

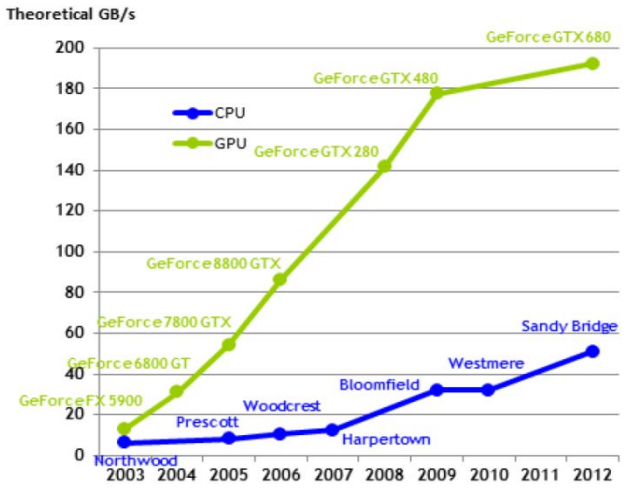
Keywords: *CUDA; GPU for acceleration; general purpose programming (GPGPU); graphic processor unit (GPU); parallel computing; data clustering*

1. GİRİŞ

1980'li yıllarda GPU (grafik işlemci birimi) geliştirme çabaları IBM ve Intel'in elindeydi. 1990'lı yıllarda ise S3 Grafik, NVIDIA ve ATI gibi sadece grafik kartı geliştirmeye yönelen firmalar doğdu. Bu yıllarda 2 boyutlu donanımsal hızlandırma yaygınlaştı ve OpenGL (OpenGL (Open Graphics Library), 2012) grafik programlama kütüphanesi kullanıma sunuldu. Böylece grafik işlemede yeni bir döneme girildi. 2000'li yıllara gelindiğinde ise GPU'nun hesaplama gücü çok artmıştı. Gerçek zamanlı, yüksek tanımlı 3 boyutlu grafikler için doymak bilmeyen piyasa talebi sebebiyle GPU'lar bir evrim geçirmiştir. Günümüzde GPU'lar yüksek hesapsal güce, çok çekirdekli işlemciye ve çok yüksek bellek bant genişliğine sahip; programlanabilir; yüksek derece paralel; çok thread'li hale gelmişlerdir. Şekil 1.1 ve Şekil 1.2 GPU'ların yıllara göre gelişen durumunu CPU (merkezi işlemci birimi) ile karşılaştırarak göstermektedir (NVIDIA, 2012).



Şekil 1.1 CPU ve GPU için saniyedeki ondalıklı sayı işlemleri karşılaştırması (NVIDIA, 2012)



Şekil 1.2 CPU ve GPU bellek bant genişliği karşılaştırması (NVIDIA, 2012)

Görüldüğü gibi grafik kartlarındaki gelişim CPU'ya oranla çok daha hızlı hızlanmaktadır. Aslında grafik canlandırma ve oyunlar için yoğun matematiksel hesaplamalar yapmak üzere tasarlanan grafik kartlarının yüksek paralel hesaplama gücü onlardan genel amaçlı programlar için de yararlanmayı söz konusu hale getirmiştir.

DOI: 10.5824/1309-1581.2013.2.002.x

Genel amaçlı bilimsel ve mühendislik uygulamalarını hızlandırmak için GPU'nun CPU ile birlikte kullanılması yaklaşımına *"Genel Amaçlı GPU Programlama veya Hesaplama"*, kısaca *"GPGPU"* denir. GPGPU, uygulamanın hesapsal yoğun kısımlarını paralel bir biçimde GPU'da, kalan kısımları ise CPU üzerinde çalıştırmayı önerir. CPU'lar seri işleme için optimize edilmiş birkaç çekirdekten oluşurken GPU'lar yüksek derecede paralelizm için tasarlanmış küçük ve etkili binlerce çekirdekten meydana gelir. Bu nedenle, CPU ve GPU güçlü ve verimli bir kombinasyon olmaktadır (Nvidia, 2012).

GPU'lardan genel amaçlı programlar için yararlanmak için Nvidia Cg (Corporation, 2012) ve OpenGL gölgelendirici (*İng. shader*) dili gibi grafik API'leriyle (Uygulama Programlama Arayüzü) genel amaçlı uygulamalar gerçekleştirilmeye çalışılmıştır. Ancak, GPU mimarilerinin ve programlama modellerinin CPU'dan çok farklı olması nedeniyle grafik API'lerinde genel bir problemi ifade etmek ve CPU için yazılan bir kodu uyarlamak oldukça zordur. Gelişmelerin artışıyla birlikte daha kullanışlı arayüzler tasarlanmıştır: Lib Sh (Lib Sh - Embedded Metaprogramming Language, 2012), Close to Metal (AMD "Close To Metal" Technology, 2012), BrookGPU (BrookGPU, 2012), DirectCompute (The Compute Shader Technology (DirectCompute), 2012), OpenCL (OpenCL™ (Open Computing Language) Zone, 2012), C++ AMP (C++ AMP (C++ Accelerated Massive Parallelism), 2012), CUDA (CUDA™ (Compute Unified Device Architecture) Zone, 2012). Bunlardan en önde gideni Nvidia'nın CUDA çözümüdür. Lib Sh, Close to Metal, BrookGPU arayüzleri uzun süredir aktif geliştirmede değildirler. DirectCompute, Windows Vista ve 7 altında GPU'da genel amaçlı hesaplamayı destekleyen bir API'dir. OpenCL ise CPU, GPU, DSP (sayısal sinyal işlemci) ve diğer işlemcileri içeren heterojen platformlarda çalıştırmak üzere program yazmak için bir çatı (*İng. framework*) niteliğinde, halen geliştirilen bir açık standarttır. OpenCL'de programlama CUDA'ya göre daha güç ve daha düşük performanslıdır; fakat heterojen ortamlarda veya farklı platformlarda çalışmak için iyi bir çözümdür (Temizel, 2011). C++ AMP, veri paralel uygulamaları donanım hızlandırıcılara devrederek performansı artırmaya yardımcı olmak için tasarlanmıştır.

Makalede, ikinci bölümde GPU mimarisi ve CUDA programlama hakkında bilgi verilecek, üçüncü bölümde konuyla ilgili literatürdeki çalışmalara değinilecek, son bölümde ise elde edilen sonuçlar sıralanarak konuyla ilgili açık alanlar belirtilecektir.

2. GPU MİMARİSİ ve CUDA

Kasım 2006'da Nvidia, yeni bir birleştirilmiş boruhattı (İng. pipeline) ve gölgelendiririci mimarisine sahip CUDA destekli ilk kartını (GeForce 8800 GTX) tanıttı. CUDA'nın diğer teknolojilerden daha fazla dokümantasyona sahip olması, NVIDIA'nın hızlı performanslı ürünleri, C-benzeri sentaks, kolay kullanım ve düşük maliyet gibi sebepler araştırmacıların uygulamalarını CUDA ile geliştirmeyi tercih etmelerine sebep olmuştur. Günümüzde Adobe Creative Suite ve Mathematica gibi yoğun veri hesaplama içeren uygulamaların CUDA ile donanım hızlandırmalı versiyonları geliştirilmiştir.

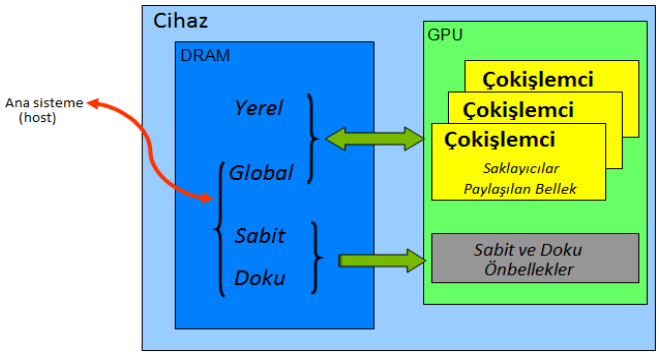
CUDA destekli kartlar donanımsal olarak her biri 8 SP (İng. Stream Processor - akış işlemci) içeren bir dizi SM'den (İng. Stream Multiprocessor - akış çokişlemci) meydana gelir. Örneğin; GeForce GT630M her biri 8 çekirdekli 12 SM'ye sahiptir (toplamda 96 eşzamanlı çalıştırma çekirdeği vardır). SM'ler tek komut çok thread (SIMT) mimarisine sahiptir. Herhangi bir saat döngüsünde her SP aynı komutu farklı veriye işleyerek çalıştırır. Her SM, Şekil 1.3'de görüldüğü gibi 4 farklı tür yerleşik belleğe sahiptir:

Saklayıcı (İng. Register): Her SP için 32-bit yüksek hızlı saklayıcılar dizisi vardır.

Paylaşılan bellek: Her SM, tüm SP'leri tarafından paylaşılan, hızlı, küçük bir miktar (SM başına 16 KB) paylaşılan bellek içerir.

Sabit ve doku önbellek : Her SM, GPU üzerindeki tüm SP'ler tarafından paylaşılan, sadece okunabilir sabit ve doku önbelleğe sahiptir.

DOI: 10.5824/1309-1581.2013.2.002.x



Şekil 1.3 Bir CUDA destekli cihazdaki bellek alanları

Yerel bellek ve *global bellek* gibi yerleşik olmayan bellekler genelde 400'den 600 saat döngüsüne kadar varan uzun erişim gecikmelerine sahiptir. En büyük bellek alanına global bellek sahiptir. SM'ler global bellek aracılığıyla iletişim kurabilirler.

CPU'larda yerleşik kaynakların önemli bir kısmı, genel amaçlı hesaplama, dallanma, senkronizasyon, pipelining ve önbellekleme için büyük komut kümelerini çözmeye tahsis edilmiştir. Bu yüzden ondalıklı sayı hesaplamaları için kullanılabilir kaynaklar sınırlıdır. GPU'larda ise kaynakların çoğunluğu veri işlemeye tahsis edilmiştir. GPU mimarisi çok thread'li yapıdadır. Çok çekirdekli büyük paralel mimarinin üst seviyesinde, thread'ler bir *grid'e* organize olurlar. Grid, 2 boyutlu bloklar kümesi içerir. Bir blok tek bir SM üzerinde çalışır ve global olarak diğer bloklarla senkronize edilemez. Blok içindeki thread'ler, bir SM içindeki farklı çekirdeklere (SP) tahsis edilirler. Bir bloktaki tüm thread'lerin kendi saklayıcıları vardır ve az yük getiren bir bariyerle aynı işlevi gören thread senkronizasyon fonksiyonu vardır. Thread'ler *warp'lara* organize olurlar. Warp, 32 paralel thread'den oluşan ve her SM'nin zamanlanma birimi olan yapıdır. Bir warp bir anda bir komut çalıştırır, bu yüzden sadece warp'taki 32 thread'in hepsi aynı çalıştırma yoluna sahipken tam verimliliğe ulaşılabilir. Bu durum iki önemli sonuç doğurur. Birincisi, bir warp'taki thread'ler koşullu dallanma nedeniyle farklı çalıştırma yollarına sahipse, warp her dalı seri bir şekilde çalıştıracaktır. Bu duruma *"thread ayrılma" (İng. thread divergence)* denir ve warp için çalışan komutların toplam zamanını arttırır. İkincisi, bir bloktaki thread'lerin sayısı warp boyutunun bir katı olmazsa, kalan komut döngüleri boşa gidecektir. Bir thread bloğu içindeki thread'ler aynı anda global bellekten peş peşe elemanlara erişirlerse, tek bir bellek işleminde birçok eleman kullanılır. Buna *"bellek bütünleştirme" (İng. memory coalescing)* denir. Paylaşılan bellek

yığın (İng. bank) yapılarında organize olur. Aynı anda birçok thread'in aynı yığına erişim talebi *"yığın çatışmasına" (İng. bank conflict)* neden olur. CUDA bu durumu, thread'lere seri erişim vererek çözer, bu da çalışma süresini arttırır.

3. LİTERATÜRE BAKIŞ

Teknolojinin gelişimiyle birlikte veri toplama yeteneği de artmış, analiz edilecek çok boyutlu ve geniş ölçekli veri kümeleri doğurmuştur. Veri demetleme uygulamalarında veri boyutu arttıkça makul bir sürede programın sonlanmasına yönelik çalışmalara ilgi artmıştır. Son zamanlarda araştırmacılar düşük maliyetle yüksek performans sağlayan GPGPU yaklaşımını tercih etmeye başlamıştır.

Veri demetleme algoritmalarının performansını GPU ile güçlendiren çalışmalar, bu başlık altında önce algoritmanın kısa bir açıklaması ardından tarihsel sırayla çalışmaların özeti şeklinde anlatılacaktır.

3.1. K-means Algoritması

K-means algoritması, 1957 yılında Cox tarafından ortaya atılmıştır; 1967 yılında MacQueen (MacQueen, 1967) tarafından k-means adı verilmiştir. Algoritmanın adımları şunlardır (Farivar, Rebolledo, Chan, & Campbell, 2008):

1. Demet sayısı k'yi belirle.
2. Rastgele k kadar demet üret.
3. Her noktayı en yakın demet merkezine ata.
4. Yeni demet merkezlerini yeniden hesapla.
5. Herhangi bir yakınsama kriteri karşılanana dek önceki iki adımı tekrarla.

K-means basitliği, uygulama alanının geniş olması ve paralelleştirmeye çok uygun yapısı ile bu zamana kadar literatürde GPU ile performansının geliştirilmesi için üzerinde en fazla çalışılan veri demetleme algoritmasıdır.

DOI: 10.5824/1309-1581.2013.2.002.x

Hall & Hart (2004) (Hall & Hart, 2004) bildiğimiz kadarı ile bir veri demetleme algoritmasını GPU ile hızlandırmaya çalışan ilk çalışmayı gerçekleştirmişlerdir. Cg grafik programlama API'sini kullanarak k-means'i GPU'da uygulamışlardır. GPU programlamanın daha zor olduğu o dönemde veri doku ünitelerinde saklanmakta idi. Bu durum aynı anda kısıtlı sayıda verinin hesaplanmasına izin veriyordu. Dolayısıyla veri ve boyut sayısında kısıtlamalar çok önemli bir sorundu. Yazarlar, bu probleme, çok geçişli etiketleme ve doku yapıları içinde farklı bir veri düzeni kullanarak çözüm getirmeye çalışmışlardır. Herhangi bir zamanda, sadece bir demet merkezi, parça gölgelendirici sabitlerinde saklanabilmiştir. Bu yapı, demet sayısı arttıkça hesaplama geçişleri arttığından performansın sınırlanmasına neden olmaktadır. Hesaplanan uzaklıkların derinlik tamponuna yazılması her defasında değerlerin 0 ile 1 aralığına ölçeklenmesini gerektirmiştir. Deneylere göre, CPU uygulamasına göre sadece 1.5 ila 3 kat hızlanma sağlanabilmiştir. Benzer olarak, Cao ve ark.(2006) (Cao, Tung, & Zhou, 2006) çok geçişli hesaplama metodunu kullanmışlar; ek olarak grafik işlemcisinin donanım hızlandırmalı vektör operasyonlarını şablon tamponu kullanarak parça işlemcideki işlem sayısını azaltmışlardır. Deneylere göre, CPU uygulamasına göre 3 ila 8 kat hızlanma sağlayabilmişlerdir. Takizawa & Kobayashi (2006) (Takizawa & Kobayashi, 2006) de aynı sorunu ele almışlar; fakat daha önceki çalışmaların probleme çok geçişli bir mekanizmayla çözüm getirmelerinden farklı olarak, bir PC küme sistemi kullanarak 3 seviyeli hiyerarşik paralel işleme geliştirmişlerdir. Büyük-ölçekli veri demetleme görevini, küçük alt kümeleri demetleme görevine bölmüşler. Alt görevleri GPU ile donatılmış PC'lere dağıtmışlar. Alt görevlerde GPU'yu en yakın komşu aramayı hızlandırmak için kullanmışlar. Uygulamayı MPI ve OpenGL ile geliştirmişler. CPU ile GPU uygulamalarının aynı sonuçları verip vermediğini ele almamışlar. Bu çalışmanın önemi, işi birçok GPU'ya büyük parçalar (coarse-grained) seviyesinde dağıtmanın mümkün olduğunu göstermesidir. İlerleyen yıllarda araştırmacıların ilgisi bu yöne kaymaya başlayacaktır. Deneylere göre, GPU olmaksızın yalnız CPU kümesine göre, yazarların GPU ile donatılmış CPU kümesi 4 kata kadar hızlanma sağlamış.

2006 yılının sonlarına geldiğimizde Nvidia'nın CUDA arayüzü ortaya çıkmış, GPU ile güçlendirilmiş veri demetleme uygulamaları geliştirmek isteyen fakat mevcut grafik API'lerinde genel bir problem ifade etmenin zorluğuyla karşılaşan araştırmacılara, C-benzeri

sentaks, yüzlerce çekirdekli yüksek derecede paralel bir mimari ve çok yüksek bellek bant genişliği sağlanmıştır. CUDA'nın getirdiği kolaylıklarla birlikte, GPU ile veri demetleme algoritmalarının performansını artırma çalışmaları hem çoğalmış hem de -*CUDA'nın da halen geliştirilmeye devam etmesiyle*- performans iyileştirmeleri yüzlerce katı bulmuştur. CUDA arayüzünü kullanarak k-means demetleme algoritmasının GPU tabanlı uygulaması ilk kez Che ve ark. (2007) (Che, Meng, Sheaffer, & Skadron, 2007)'nın çalışmasında görülmüştür. Yazarlar, önceki çalışmalardaki gibi GPU'da sadece en yakın demet merkezlerinin bulunduğu aşamayı her thread'e 1 veri noktasının hesabını vererek paralelleştirmişlerdir. Deneylerde, k-means'in CUDA versiyonu, MineBench CPU versiyonuna göre 8 kata kadar hızlanma göstermiştir. Yazarlar, CUDA'nın sunduğu bir çok imkandan yararlanmadıkları ve herhangi bir tasarım yapmadıkları halde, geçmiş çalışmalara göre oldukça iyi bir hızlanma elde etmişlerdir. Bu durum GPGPU'nun çok büyük umut vaat ettiğinin açık bir göstergesidir. Yazarların çalışmasına çok benzerlik gösteren Farivar ve ark. (2008) (Farivar, Rebolledo, Chan, & Campbell, 2008) çalışmasında ise k-merkezlere çok sık erişim olması nedeniyle bellek erişim gecikmesinin önüne geçmek için demet merkezleri GPU'nun global belleği yerine GPU'nun sabit (constant) belleğinde saklanmıştır. Bir boyutlu veri noktaları kullandıkları deneylerde, temel CPU uygulamasına göre, geliştirdikleri uygulama 13 kat hızlanma sunmuştur. Bu çalışma, sık erişilen veriler için sabit bellek kullanımının performanstaki katkısını açıkça göstermiştir. Öte yandan bu durum, demet ve boyut sayısını sınırlamıştır. Ayrıca, deneylerde çok boyutlu bir veri kümesi kullansalardı, uzaklık hesabı tek bir çıkarma işlemi ile hesaplanamayacak, eleman başına düşen hesapsal yük artacak ve gerçek veri kümeleri kullanıldığında uygulamanın nasıl çalıştığıyla ilgili daha doğru sonuçlar elde edilecekti. Global bellek erişimi gecikmesini önlemeye çalışan bir başka çalışma Che ve ark. (2008) (Che, Boyer, Meng, Sheaffer, & Skadron, 2008)'in uygulamasıdır. Bu amaçla yazarlar tüm veri kümesini GPU'nun doku belleğine, demet merkezlerini ise GPU'nun sabit belleğine saklamışlar. Deneylerde tek thread'li CPU referans uygulamasına göre 72 kat; 4 thread'liye göre 35 kat hızlanma sağlamışlardır. Yazarların performans kazançlarının temelinde yatan etken, verimli okuma için önbellek mekanizmasına sahip GPU'nun doku ve sabit belleğinden yararlanmaları olmuştur. Bu uygulamadaki dezavantaj, tüm veri kümesinin doku belleğe sığmasını gerektirmesidir.

DOI: 10.5824/1309-1581.2013.2.002.x

Bu zamana kadarki çalışmalarda k-means'in sadece 1 adımını paralelleştiren uygulamaların aksine, Shalom ve ark. (2008) (Shalom, Dash, & Tue, Efficient K- Means Clustering Using Accelerated Graphics Processors, 2008) algoritmanın tamamını GPU'da uygulamaya çalışmışlardır. Ancak uygulamayı OpenGL ve gölgelendirici (shader) programlar aracılığıyla kernelleri uyandırmak için GLSL kullanarak gerçekleştirmişlerdir. GPU'nun çok-geçişli canlandırma ve çok-gölgelendirici yeteneklerini kullanmışlardır. Tüm verileri dokularda saklayarak doku kullanımını maksimize edip gölgelendirici program sabitleri kullanımını minimize etmişlerdir. Bu sayede, CPU ve GPU arasındaki veri işlemlerini azaltmışlardır. K-means'in tüm adımlarını GPU'da uyguladıkları için iterasyonlar boyunca verinin CPU'ya kopyasını tutuyormuş, böylece yarış durumu önlenmiş. Sonunda, azaltma nesneleri ana (host) belleğe geri kopyalanıyormuş ve gerekirse global bir birleştirme yapılıyormuş. Makalede deneysel çalışmalarda işlemlerin ne kadar süre aldığına yer verilse de, elde edilen hızlanmaya dair bir bilgi belirtilmemiş. Ayrıca, yazarların çoğaltma yaklaşımı büyük ölçekli veri kümeleri kullanıldığında ek yük getirebilmektedir. GPU'nun bellek yapılarından verimli yararlanmak isteyen Böhm ve ark. (2009) (Böhm, Noll, Plant, Wackersreuther, & Zherdin, Transactions on Large-Scale Data- and Knowledge-Centered Systems I, 2009)'nın çalışmasında ise saklayıcılar da kullanılmıştır. Veri kümesi ve demet merkezleri GPU'nun global belleğinde saklanmış.Bir veri noktasının demet atamasından sorumlu olan her thread, verisini kendi saklayıcısına yüklüyormuş, sonra tüm merkezleri tek tek saklayıcıya yükleyip verinin yüklendiği merkez ile arasındaki uzaklığı hesaplıyormuş. Bu uzaklık, minimum uzaklıktan küçükse noktayı o demete atıyormuş. Yazarların bu yaklaşımı, çok hızlı saklayıcı yapılarından faydalandıkları için performansa büyük katkı getirmiştir. Deneylerde CPU uygulamasına göre küçük demet sayıları için 100 kata kadar ve 256 demet için 1000 kat hızlanma sağlanmış. Bu çalışma, CUDA'nın kullanıma imkan sağladığı küçük bellek alanına sahip fakat çok hızlı saklayıcıların doğru yerde, doğru şekilde kullanıldığında ne kadar kazanç getireceğini göstermektedir.

Wu ve ark. (Mart 2009) (Wu, Zhang, & Hsu, GPU-Accelerated Large Scale Analytics, 2009) ise veri kümesini global belleğe yükledikten sonra satır-tabanlı düzenden sütun tabanlı düzene çevirerek farklı bir yaklaşım denemişlerdir. Bu sayede, global bellek kullanımından dolayı olası gecikmeleri minimize edecek verimli bütünleşik bellek okumalarına imkan

tanınmıştır. Bu düzen, k-means'in GPU uygulaması için ilk defa bu çalışmada görülmektedir. Yazarlar k-means'in sadece her veri noktasına bir merkez atanan prosedürünü GPU üzerinde paralelleştirmişler, diğer adımlar CPU'ya bırakılmıştır. Makalede, kerneller yürütülürken değeri sabit kalacak verilerin doku ve sabit bellekte; çok sık erişilecek verilerin ise paylaşılan bellekte tutularak verim alındığı vurgulanmıştır. Deneylerde, tek çekirdek üzerinde çalışan MineBench'e göre ortalama 190 kat hızlanma elde edilmiştir. Bir önceki çalışmada büyük demet sayısı için elde edilen verimin çok daha yüksek olması, saklayıcılardan yararlanmanın ne kadar önemli olduğunu bir kez daha göstermektedir. Aynı şekilde sütun tabanlı düzeni kullanan Zechner & Granitzer (2009) (Zechner & Granitzer, 2009) çalışmasında veri, boyut ve demet sayısına getirilen sınırlamaları kaldırmak için veriyi parça parça işleme yaklaşımı geliştirmiştir. CPU'da veri noktaları thread sayısına bölünerek işlenecek parça sayısı bulunmuş. Her blok, 1 veya daha çok parçanın işlenmesinden sorumluymuş. Bir thread, bloktaki diğer thread'lerden önce bitirirse, bütünleşik bellek erişimi için diğer thread'leri bekliyormuş. Bloktaki thread'lerden her biri, aynı merkezin bir bileşenini paylaşılan belleğe yüklüyormuş. Thread'ler, kendi veri noktasının bileşenini global bellekten bütünleşik alıyormuş. Herhangi bir anda, bir bloktaki tüm thread'ler, aynı merkeze uzaklığı hesaplıyorlarmış. Bir merkezi tüm boyutlarıyla paylaşılan belleğe yüklemek, boyut sayısını kısıtlayacağından, yüklemeyi ve uzaklık hesaplama işlemini parça parça yapmışlar. Her iterasyonda bir bloktaki thread sayısı kadar merkezin bileşeni, paylaşılan belleğe yüklenmiş. Her bileşen için kısmi uzaklık hesaplanmış. Tüm thread'ler, veri noktasına en yakın merkezi bulduğunda, merkezin etiketi global belleğe yazılmış. Deneylerde CPU uygulamasına göre 43 kata kadar hızlanma sağlanmış. K-means'in GPU uygulaması için ilk defa burada uygulanan parça işleme tekniği sınırları kaldıran oldukça kullanışlı, önemli bir yöntem olsa da getirdiği ek yükler nedeniyle hızlanma diğer çalışmalardan daha düşük olmuştur. Wu ve ark. (Mayıs 2009) (Wu, Zhang, & Hsu, Clustering Billions of Data Points Using GPUs, 2009) da veri kümesine getirilen sınırı ortadan kaldırmaya çalışmışlardır. GPU'nun belleğine sığmayacak kadar büyük veri kümesiyle çalışan uygulamalar için önceki uygulamalarını (Wu, Zhang, & Hsu, GPU-Accelerated Large Scale Analytics, 2009) genişletmişlerdir. Kullanılan akış tabanlı yaklaşım da benzer şekilde veri kümesini parçalara bölmeye dayanıyormuş. Her iterasyonda sırayla

DOI: 10.5824/1309-1581.2013.2.002.x

büyük bloklar işleniyormuş. Bir bloğun işlenmesi, bloğun GPU'ya transfer edilmesini; sütun-tabanlı biçime dönüştürülmesini; her verinin demet üyeliğinin bulunmasını, bulunan sonuçların CPU'ya gönderilmesini içeriyormuş. CUDA akışları, her blok üzerindeki ilerlemeyi izlemek için kullanılıyormuş. Tüm çağrılar asenkronmuş. k merkezler sabit bellekte tutulmuş.1 milyar verili bir veri kümesi kullandıkları deneylerde, 2 akışlı seçenek en iyi çalışmış. Akış etkin işlem, tüm veri kümesinin GPU'nun belleğine sığdığı uygulamaya göre ekstra transpose işlemi, kernel çalıştırma ve senkronizasyonun getirdiği ek yükten dolayı 1.1 ila 2.5 kat arasında bir düşüş göstermiş. (Wu, Zhang, & Hsu, GPU-Accelerated Large Scale Analytics, 2009)'teki uygulamalarında 1 kez transpose yeterliyken, burada her parçanın her seferinde transpose edilmesi gerekmiş. Sabit belleğin yetmeyeceği durumlarda doku bellek de kullanıldığında, program 2.2 kat yavaşlıyormuş. Merkezler sabit belleğe sığdığında, 8 çekirdekli CPU versiyonuna göre 10 kattan çok hızlanma; doku bellek de kullanıldığında 3 kat hızlanma sağlanmış. Ayrıca, (Che, Boyer, Meng, Sheaffer, & Skadron, 2008)'den 2-4 kat; (Fang, et al., 2008)'den 20-70 kat daha hızlıymış.

Bai ve ark. (2009) (Bai, He, Ouyang, Li, & Li, 2009) ise demet merkezlerini güncelleme adımını da GPU'da yapmışlar. Bu adıma geçmeden önce CPU'da demet etiketlerini sıralatıp her demetin kaç veri noktası içerdiğini hesaplatmışlar. GPU'ya yüklenen bu verilere göre her thread bir demetin yeni merkezini hesaplamakla sorumlu tutulmuş ve kendi demetinin veri nesnelerini sürekli okuyarak işlemini gerçekleştirmiş. Bu sayede, her thread için her veri noktasının sorumlu olduğu demete ait olup olmadığına tek tek bakması gerekmemiş. Bu da bir warp'taki thread'ler arasında SIMD yapısının bozulmasıyla verimi düşüren thread ayrılma durumu oluşmasını önlemiş. Bu sayede güncelleme işlemi çok hızlı bir şekilde gerçekleşmiş; ancak CPU-GPU arası veri transferi yükü oluşmuştur. Deneylerde, CPU tabanlı k-means'e göre yazarların GPU tabanlı uygulaması, sadece demetleme çalışmasına bakıldığında 27 ila 56 kat; toplam çalışma zamanına göre 8 ila 14 kat hızlanma sağlamıştır. Bu çalışma k-means'in GPU uygulaması için demet güncelleme adımında paralelizmi artıran sıralama yaklaşımını ilk kez kullanması bakımından önemlidir. Yazarın thread ayrılmayı engelleme yaklaşımını oldukça verimli olsa da ek yükler, tüm verinin global bellekte tutularak GPU'nun çeşitli hızlı bellek yapılarından faydalanılmaması, global belleğe erişimlerin sık olması ve bütünleşik olmaması performansın önceki çalışmalardan daha

düşük olmasıyla sonuçlanmıştır. Etiketleme adımının yanında merkez güncelleme adımını da GPU ile paralelleştiren ve CPU'da sıralama yaklaşımını kullanan bir diğer çalışma Karch (2010) (Karch, 2010)'ın yüksek lisans tez çalışmasıdır. GPU'da görüntü demetleme üzerine odaklanan yazar, önceki çalışmadan farklı olarak etiketleme adımında her thread'in senkronizasyonla bir merkezi paylaşılan belleğe yüklemesini ve bu merkezlerle sorumlu olduğu piksel arasındaki uzaklığı hesaplamasını sağlamış. Ayrıca, merkez güncelleme adımında da farklı olarak, her thread bloğuna bir demetin merkezini hesaplatmış. Global bellekten blok boyutunca piksel okunuyormuş. Her thread,1 pikselin koordinatlarını (R,G,B) paylaşılan bellekteki koordinat dizisine ekliyormuş. Her thread, kendi dizi elemanındaki değerleri toplamak zorundaymış; çünkü yarış koşulları nedeniyle tüm thread'ler için bir değişken kullanmak mümkün olmuyormuş. Bu ihtiyaçtan dolayı paralel azaltma fonksiyonu, piksel koordinatlarını topluyormuş. Deneylerde CPU uygulamasına göre 50 kata kadar hızlanma sağlanmış. Bu çalışma GPU'nun paylaşılan bellek yapısından yararlandığı önceki çalışmadan daha iyi performans kazancı elde etmiştir.

Wu & Hong (2011) (Wu & Hong, 2011) ise k-means'in GPU uygulaması için ilk defa üçgen eşitsizliği ilkesinden faydalanarak gereksiz uzaklık hesaplamalarını önleyen bir yaklaşım izlemişlerdir. Sadece etiketleme adımının GPU ile hızlandırıldığı uygulamada, geliştirilen CUDA-tabanlı hibrit algoritma, genel bir veri seti için üçgen eşitsizliğinin kullanılıp kullanılmamasına duruma göre kendi belirliyormuş ve yük dengeleme yapıyormuş. Ayrıca, yük dengeleme ve bellek bütünleştirme arasındaki ödünleşimi incelemek için veri düzenlemesini yeniden ayarlayan bir teknik sunmuşlar. Deneylere göre, hibrit CUDA algoritması, tek thread'li CPU-tabanlı versiyona göre k'nın küçük değerleri için 75 kat hızlıymış. Hibritleşme ek yük getirmesine rağmen, k'nın büyük değerleri için performansı hibritleşmenin olmadığı yaklaşımla aynı oluyormuş, ölçeklenebilirlikte ise hibrit yöntem daha iyiymiş. Bu çalışma, üçgen eşitsizliği kullanarak hesaplama sayısını azaltması bakımından önemlidir, bu yöntem uzaklık hesabı gerektiren diğer demetleme algoritmalarında da oldukça kullanışlı olacaktır.

Veri demetleme algoritmalarının GPU kullanarak paralel bir şekilde uygulanmasını sağlayan CUDA için C ile açık-kaynak kodlu bir kütüphane yazan Kohlhoff ve ark. (2011)

DOI: 10.5824/1309-1581.2013.2.002.x

(Kohlhoff, M.H.Sosnick, Hsu, Pande, & Altman, 2011) çalışmalarına CAMPAIGN (Clustering Algorithms for Massively Parallel Architectures Including GPU Nodes) ismini vermişler. CAMPAIGN'de k-means, k-medoids, k-centers, hiyerarşik demetleme ve kendini-düzenleyen harita olmak üzere 5 algoritmanın seri bir CPU referans versiyonu ve bir GPU-hızlandırılmış versiyonu bulunuyormuş. Uzaklık ölçütü için öklid, manhattan ve chebyshev seçeneklerini sunmuşlar. Makalede açık kaynak kodlu uygulamalarını isteyenlerin indirebilecekleri bir link vermişler; bunun dışında kullandıkları yöntem ve tasarımlarıyla ilgili hiçbir bilgi vermemişler. Deneylerde, CPU referans uygulamasına göre, k-means 69 kat, k-medoids 102 kat, k-centers 178 kat, hiyerarşik demetleme 5 kat, kendini düzenleyen harita ise 2 kat civarında bir hızlanma sunmuş. K-means'in neredeyse tüm adımlarını (ilklendirme adımı hariç) GPU'da uygulayan (Shalom, Dash, & Tue, Efficient K- Means Clustering Using Accelerated Graphics Processors, 2008), (Fang, et al., 2008)], (Bai, He, Ouyang, Li, & Li, 2009) ve (Karch, 2010) çalışmalardaki gibi Jian ve ark. (2011) (Jian, et al., 2011) da k-means'i CUDA ile paralelleştirmiştir; bu amaçla demet etiketi güncelleme, merkez güncellemeye merkez hareketlenmesi bulma olmak üzere 3 kernel yazmışlar. Demet etiketi güncelleme kernelinde, her thread1 veri noktasından sorumluymuş. Veri noktalarının boyut bölümleri ve k merkezler bütünleşik şekilde paylaşılan belleğe yüklenmiş. Merkez güncelleme kernelinde, geliştirdikleri paralel yüksek boyut azaltma planını uygulamışlar. Bu planda, bir verideki farklı boyutlar bağımsızsa tüm verilerdeki aynı boyut, ayrı bir vektör olarak ele alınıyormuş ve her thread bloğuna bir boyut vektörü veriliyormuş. 1 boyutlu azaltma için CUDA SDK' daki ardışık adresleme azaltma seçilmiş. Son iterasyonda, yeni merkezler eski merkezlerden çok uzaktaysa, hareketlenme olmuş demekmiş. Bunun için merkez hareketlenmesi bulma kernelinde, öncelikle eski merkezler ile yenilerin arasındaki fark hesaplanıp, fark matrisine atılıyormuş ve paralel yüksek boyut azaltma planı uygulanıyormuş. Deneylere göre CU-K-means, küçük bir veri seti üzerinde (Fang, et al., 2008)'den 5 kat daha hızlıymış. Bu çalışmada bütünleşik bellek okumalarına dikkat edilmiş; boyut azaltma tekniğiyle thread paralelleştirme maksimize edilmiş ve paylaşılan belleği değiştirme maliyet önlenerek performans kazancı elde edilmiştir. Benzer biçimde, Kohlhoff ve ark. (2012) (Kohlhoff, Pande, & Altman, K-means for parallel architectures using all-prefix-sum sorting and updating steps, 2012) da k-means'i tamamen GPU'da uygulamıştır. (Bai, He, Ouyang, Li, &

Li, 2009) ve (Karch, 2010) çalışmalarındaki gibi demet güncelleme adımından önce sıralama yaklaşımı kullanılmıştır; fakat onlardan farklı olarak bu işlem GPU'da hesaplatılmıştır. Çalışmada paralel-önek-toplamı tabanlı sıralama algoritması uygulanmıştır. (Wu, Zhang, & Hsu, GPU-Accelerated Large Scale Analytics, 2009), (Zechner & Granitzer, 2009) ve (Wu, Zhang, & Hsu, Clustering Billions of Data Points Using GPUs, 2009) çalışmalarındaki gibi veri noktaları sütun-tabanlı; demet vektörleri satır-tabanlı düzende saklanmıştır. Demet vektörleri, peş peşe tek tek işlenirken, veri vektörleri, niteliklere bölünmüştür. 4 yardımcı kernel yazılmıştır. *(1)Paralel veri azaltma kerneli*, belleğin bir bölümünde saklanan kısmi değerleri alıp tek bir sonuç değerine birleştirmektedir. *(2)Tüm-önek-toplamı kerneli*, bir çok thread ile paylaşılan bellek üzerinde çalışan toplam bu kernel ile yerinde hesaplanmaktadır. Bu kernel, thread'lere dağıtılmış veriden bir kısmının bir dizide toplanmasına karar verildiğinde, istenen verileri tutan thread'lere, dizinin indislerini bildirmek için kullanılmıştır. *(3)Veri sıkıştırma kerneli*, sıralanmamış veri noktaları içinden belirli bir demete atanan veri noktalarını seçme gibi bir kriteri karşılayan alt kümeyi çıkarmada kullanılmıştır. Burada komşu indisleri hesaplamak için tüm-önek-toplam kerneli kullanılmıştır. *(4)Uzaklık ölçütü kerneli*, ölçütler arasında kolayca geçiş yapmak için ayrı bir kernel olarak yazılmıştır. Vektörler bileşenlerine bölünerek sabit-uzunlukta segmentlerde işlenmiştir. Böylece paylaşılan belleğin sabit-uzunluğu kullanılmıştır. Veri noktaları demetlere atanmalarına göre sıralanırken, tüm noktaların yarısından fazlası sıralanmamışsa tam bir sıralama; aksi takdirde önceki sıralamayı güncelleme uygulanmıştır. Sıralama sayesinde, zaman karmaşıklığını azaltılmış ve bir warp içindeki thread'lerin hepsinin doluluğunu sağlanmış; fakat sıralama yaparken tampon kullanıldığından alan gereksinimi neredeyse iki katına çıkmıştır. Deneylerde, CPU uygulamasına göre 200 kata kadar hızlanma sağlanmıştır.

K-means'in en yakın demet arama adımı için bir PC küme sistemi üzerinde 3 seviyeli hiyerarşik paralel işleme öneren (Takizawa & Kobayashi, 2006)'deki çalışma gibi Vaitheeshwaran ve ark. (2012) (Vaitheeshwaran, Nagwanshi, & Rao, 2012) da onlardan farklı olarak uygulamayı OpenCL kullanarak geliştirmeyi önermiştir. Yazarlar deneylere göre bu yaklaşımın oldukça performans kazancı sağladığını söylemişler; fakat makalede uygulamalarının sözde koduna, deneysel çalışma detaylarına ve sonuçlarına ilişkin herhangi bir bilgiye yer vermemiş, sadece önerilerini sunmuşlardır.

DOI: 10.5824/1309-1581.2013.2.002.x

Mevcut uygulamalardan farklı olarak veri boyutuna göre 2 strateji geliştiren Li ve ark. (2013) (Li, Zhao, Chu, & Liu, 2013)'nın uygulamalarında veri kümesi küçükken saklayıcılardan yararlanılırken, veri kümesi çok büyükken hesapsal yükün bellek erişimine oranını yüksek tutmak için paylaşılan bellekten de yararlanılmıştır. Veri kümesi küçükken, veri noktaları global bellekten saklayıcılara yüklenip o veri noktasına en yakın merkezi bulma işlemi boyunca saklayıcılardan erişilmiştir. Global belleğe bütünleşik erişimler yapılarak okuma gecikmesi azaltılmıştır. Yöntemin dezavantajı saklayıcı sayısıyla sınırlanmasıdır. Veri kümesi büyükken, veri paylaşılan belleğe kare kare bölünerek yüklenmiştir. Her veri noktası global bellekten 1 kez okunmuştur. Bu yöntemde, yığın çatışmasını önlemek için bir yarı-warp'daki thread'ler için 16 sürekli adrese erişen bütünleşik okuma benimsenmiştir. Yazarlar, uzaklık hesaplama işleminin matris çarpımı ile aynı akışı paylaştıklarını fark etmişler. Veri noktalarını data[n][d], merkezleri centroid[d][k] ve uzaklık sonuçlarını Result[n][k] matrisi olarak ifade etmişler ve 3 matrisi, 16x16 kare matrislere bölmüşler. Her blok Result matrisindeki 2 kareyi hesaplıyormuş: SR[16][16x2]. SR global bellekte uzaklıkları saklıyormuş. Her thread, SR'nin bir sütununu hesaplıyormuş. Global bellekten verinin bir karesi, paylaşılan belleğe yükleniyormuş ve geçici bir uzaklık hesaplanıyormuş, sonuç saklayıcıda tutuluyormuş. Merkezler sabit bellekte saklanıyormuş. Yeni demet merkezinin bulunduğu işlemi, "böl ve yönet" stratejisiyle gerçekleştirmişler. Bu yöntemde veriyi SM sayısına bağlı olarak gruplara ayırıp, her grubu azaltarak geçici merkezleri alıyorlarmış, son demet merkezlerini CPU'da hesaplıyorlarmış. Büyük veri kümeleriyle çalışırken, "böl ve birleştir" stratejisini kullanmışlar. Veri kümesini grup grup yükleyip, sonra geçici sonuçları hesaplatıp, birleştirerek sonucu elde ediyorlarmış. Deneylere göre, düşük boyutlu veri kümelerinde, [25]'den 3 ila 8 kat; [20]'dan 10 ila 20 kat; [22]'den 100 ila 300 kat daha hızlıymış. Yüksek boyutlu veri kümelerinde ise [20]'dan 4 ila 8 kat; [22]'den 10 ila 40 kat daha hızlıymış.

K-means veri demetleme algoritması için GPU ile paralelleştirilmiş literatürdeki uygulamalar burada sona ermektedir. Literatürdeki çalışmaları birbiriyle karşılaştırarak yorum yapmak bizim açımızdan zor; çünkü elde edilen hızlanmalar bir çok faktöre bağlıdır. Örneğin; bazı çalışmalar, sadece hızlandırdıkları adımın karşılaştırmasını sunarken, bir kısmı uygulamanın tamamı için karşılaştırmasını sunmuştur. Ayrıca, her çalışmada

kullanılan veri kümesi, boyut sayısı, demet sayısı, toplam iterasyon sayısı farklılık göstermektedir. Bunun dışında, karşılaştırma için temel aldıkları CPU uygulaması hepsinde farklıdır. Dahası, yazarların çoğunluğu çalışmalarında sadece hızlanma katsayısına yer verdiği, geçen süreyi belirtmediği için bir çalışmanın diğer bir çalışmayla karşılaştırılması sağlıklı olmayacaktır. Fakat, bazı yazarlar çalışmalarını internette mevcut ve indirilebilir kıldıkları için bir çok araştırmacının kendi uygulamasını bunlarla karşılaştırmalarına imkan tanımıştır.

3.2. Hiyerarşik Birleştirici Demetleme (HAC) algoritması (Hierarchical agglomerative Clustering Algorithm)

Hiyerarşik birleştirici demetleme (kısaca HAC) algoritması, AGNES *(İng. AGglomerative NEsting)* algoritması olarak da bilinir. AGNES algoritması, 1990 yılında Kaufman ve Rousseeuw (Kaufman & Rousseeuw, Finding Groups in Data: An Introduction to Cluster Analysis, 1990) tarafından sunulmuştur. HAC veya AGNES algoritmasında (Voorhees, 1986) her nokta ayrı bir demet olarak başlar. Demetlerin birleştirilmesi "*dendogram*" denilen ağaç-benzeri bir yapıda sonuçlanır.

Genellikle hiyerarşik demetleme 4 temel adımda gerçekleştirilir (Zhang & Zhang, 2006):

(1) Tüm verilerin arasındaki uzaklığı hesapla ve benzerlik uzaklık matrisini oluştur.

(2) Birbirine en az uzaklıktaki r ve s demetlerini bul.

(3) r ve s demetlerini birleştir. Birleşmeden etkilenen tüm uzaklıkları yeniden hesapla.

(4) Adım 2 ve 3'ü, toplam demet sayısı 1 olana kadar tekrarla.

Zhang & Zhang (2006) (Zhang & Zhang, 2006) hiyerarşik demetlemenin GPU-tabanlı bir uygulamasını gen ekspresyonu veri analizini hızlandırmak amacıyla gerçekleştirmişlerdir. Nvidia CG grafik programlama dilini kullanan yazarlar, genelde mikro dizi veri kümelerinin az boyutlu olmasını temel almışlar ve dokuların kısıtlı sayısı bu uygulama için yeterli olmuş. Tek bir veriden tüm değerlerini doku adresleme mantığıyla 1

DOI: 10.5824/1309-1581.2013.2.002.x

seferde alabilmişler. Uzaklık hesabında ise uzaklık matrisinin aktif kısmını kapsayan üçgen bir pencere canlandırmışlar. Üçgen canlandırma, karşılık gelen her uzaklık hesabı için bir parça (*fragment*) üretilmesini gerektirmiş. Deneylerde, CPU uygulamasına göre 2 ila 4 kat hızlanma elde edilmiş. Bu çalışma veri boyutunun daha fazla olduğu veri kümelerine de uyacak şekilde tasarlanmamıştır. Ancak bildiğimiz kadarıyla ilk defa hiyerarşik demetlemeyi GPU ile hızlandırdığı için önemlidir. O dönemki kısıtlar altında performans kazancı fazla elde edilememiştir. CUDA'nın ortaya çıkışıyla birlikte, yüksek boyutlu vektörleri de demetlemek için Wilson ve ark. (2007) (Wilson, Dai, Jakupovic, & Meng, 2007) CUDA tabanlı HAC algoritması gerçekleştirmişlerdir. Performansı, CPU üzerinde çalışan ticari biyoinformatik demetleme uygulamalarıyla karşılaştırmışlar ve 10'dan 14 kata kadar hızlanma elde etmişlerdir. Böylece, CUDA kullanmanın verimliliğini göstermişlerdir.

Hiyerarşik birleştirici demetlemede (HAC) temel işlemlerden biri olan çift yönlü (*pairwise*) uzaklık hesaplamayı Chang ve ark. (2008) (Chang, Jones, Li, Ouyang, & Ragade, 2008) CUDA aracılığıyla GPU'yu kullanarak hızlandırmaya çalışmışlardır. Çift yönlü uzaklıklar için 2 CUDA algoritması sunulmuş. İlk CUDA kodunda, uzaklık hesabı için yarı-matris kullanılmış. Her thread uzaklık matrisinin 1 satırından sorumluymuş ve bu satırı paylaşılan belleğe yükleyip bloktaki diğer satırlara uzaklığını hesaplıyormuş. İkinci CUDA kodunda, her thread uzaklık matrisinde 1 elemandan sorumluymuş. Bir thread uzaklık matrisinin (i,j) elemanını hesaplarken, öncelikle i.veriyi paylaşılan belleğe yüklüyormuş. (i,j+1), (i,j+2)… (i,j+15) elemanlarını hesaplayacak thread'lerin hepsi; paylaşılan belleğe yüklenen i.verinin aynı kopyasını kullanıyorlarmış. Aynı şekilde, j. veri paylaşılan belleğe yüklendiğinde de, (i+1,j), (i+2,j)… (i+15,j) elemanlarını hesaplayacak thread'lerin hepsi; paylaşılan belleğe yüklenen j. verinin aynı kopyasını kullanıyorlarmış. Bu durum, paylaşılan bellekte veri paylaşan thread bloğunun kare kullanılmasının avantajını ortaya koyuyormuş. Bir bloktaki tüm thread'ler alt matrisleri paylaşılan belleğe yükledikten sonra, her thread kendi kısmi öklid uzaklığını hesaplıyormuş. Sonra thread'ler senkronize edilip sonraki alt matrise geçiliyormuş. Burada önemli bir nokta yazarların yığın çatışmasını azaltmak için veriyi sütun tabanlı düzene çevirmeleridir. Bu sayede 16 kat çatışma azaltılmış. Deneylere göre, CPU uygulamasına göre yazarların CUDA-1 uygulaması 4 ila 8 kat; CUDA-2 ise 20 ila 44 kat hızlanma sağlamış. Bu çalışmadaki ikinci yöntem, paylaşılan belleği verimli

kullanması, kare thread bloğu kullanması, yığın çatışmasını önlemesi etkili olmuş ve performansa önemli katkılar getirmiştir.

HAC algoritmasının bir kısmını GPU'da paralelleştiren çalışmalardan farklı olarak Shalom ve ark. (2009) (Shalom, Dash, Tue, & Wilson, Hierarchical Agglomerative Clustering Using Graphics Processor with Compute Unified Device Architecture, 2009) algoritmanın tamamını tek bağlantı (*İng. single link*) kullanarak CUDA ile uygulamışlardır. Global bellek daha az bellek yönetimi gerektirdiği gerekçesiyle paylaşılan bellekten çok global belleği kullanmışlardır. GPU'da benzerlik yarı-matrisini hesaplatmışlardır. Benzerlik matrisi global bellekte saklanmıştır. Her blok, benzerlik matrisinin bir kare alt matrisini hesaplamış; her thread ise alt matristeki bir elemanı hesaplamıştır. Minimum uzaklık çiftlerini *"cublasIsamin"* fonksiyonuyla sanal olarak bir geçişte belirlemişlerdir. Demetleri birleştirip yeni demet vektörünü hesaplatmışlar, benzerlik yarı matrisini ve minimum uzaklık dizisini güncellemişlerdir. Sonra demeti CPU'ya transfer etmişlerdir. Benzerlikleri güncelleme ve demetleri birleştirme işlemi tek bir demet olana dek tekrarlanmıştır. Global bellekte 1 boyutlu dizi kullanmak hesapsal performansı önemli derecede arttırmış, sürekli bellek adreslerine yazma ve okuma, işlemlerini daha verimli yapmıştır. Deneylere göre, CPU versiyonuna göre 30 ile 65 kat hızlanma sağlanmış. Global bellek kullanmaları sebebiyle nitelik sayısı 100'ü aşınca hızlanma çok hızlı bir şekilde düşmüştür. Bu çalışmaya çok benzer olarak, Chang ve ark. (2009) (Chang, Kantardzic, & Ouyang, Hierarchical clustering with CUDA/GPU, 2009) de HAC'i tamamen CUDA kullanarak uygulamışlardır. Çift yönlü uzaklık matrisini hesaplamada önceki çalışmalarındaki (Chang, Jones, Li, Ouyang, & Ragade, 2008)'deki CUDA-2 yaklaşımını izlenmiş; fakat burada sadece öklid uzaklığı kullanılmamıştır. Tek bağlantı işlemini seri bir şekilde yürütüp her iterasyonu mümkün olduğunca paralelleştirmişlerdir. Her thread çift yönlü uzaklık matrisinin 1 satırı için minimumu bulmuştur. *CUDA paralel azaltma* (*reduction*) kullanılarak minimumlar matrisinin minimumu bulunmuştur. En yakın demetler birleştirilip *SANN* özelliği kullanılarak diğer demetlerin en yakın komşuları güncellenmiştir. Bu adımdaki kernelde, her thread bir demetten sorumluymuş. Matrisin birleştirilen demetin uzaklık matrisindeki yeni minimumu CUDA paralel azaltma kullanılarak bulunmuştur. Deneylere göre, CPU versiyonuna göre öklid ölçütü için 33 kata kadar; manhattan ölçütü için 48 kata kadar; pearson korelasyon

DOI: 10.5824/1309-1581.2013.2.002.x

katsayısı için 28 kata kadar hızlanma sağlanmıştır. Bu çalışma, daha dikkatli tasarımıyla önceki çalışmaya göre daha ölçeklenebilirdir, yüksek veri boyutlarına sahip veri kümelerine de uygundur. GPU'da HAC algoritmasının tamamen uygulayan bir başka çalışma da Shalom & Dash (2011) (Shalom & Dash, Efficient Hierarchical Agglomerative Clustering Algorithms on GPU Using Data Partitioning, 2011)'nın uygulamasıdır. Çalışmada HAC'in gerektirdiği zaman ve bellek karmaşıklığını azaltan Kısmen Örtüşen Bölme (PoP) yöntemi de kullanılmıştır. PoP yönteminde, veri p sayıda örtüşen hücreye bölünür. Kesişime "δ-bölgesi" denilir. δ, bölme mesafesi veya deltasıdır. Her hücre, çekirdek bölgesini ve komşu δ-bölgeleri içerir. Merkez ölçütü için, her demet, tek bir temsilci nokta ile gösterilir. Bir demetin temsilci noktası, bir δ-bölgesine düşerse, etkilenen her hücre onu tutar; düşmezse sadece çekirdek bölge onu tutar. PoP'un temel konsepti, iterasyonlarda en yakın çiftin diğer tüm hücrelerden bağımsız her hücre için bulunması ve bunlardan genelde en yakın çiftin bulunmasıdır. Genelde en yakın çift mesafesi δ'dan azsa, o zaman çift birleştirilir ve sadece kapsayıcı hücrenin benzerlik matrisi güncellenir. En yakın çift veya birleştirilen demet bir δ-bölgesindeyse, o zaman etkilenen hücrelerin de benzerlik matrisi güncellenir. Başlangıçta δ'ya çok küçük bir değer, p'ye çok büyük bir değer verilir. Git gide δ %x arttırılıp, p %y azaltılır. PoP klasik HAC'in gerektirdiği zaman ve bellek karmaşıklığını azaltır. Yazarlar, PoP yapısını "90-10 ilişki" biçimine dayanarak, 2-boyutlu veriler için uygulamışlardır. HAC algoritmasını GPU'da uygularken karşılaşılan kısıtlar şunlar olmuştur: hesaplamalarda thread'lerin yetersiz kalması ve senkronizasyon gerektirmesi; geniş veri kümeleri için bellek yetersizliği; küçük fakat hızlı paylaşılan belleğe karşı büyük fakat yavaş global bellek için programlanabilirlik. PoP-destekli HAC uygulamasında, her PoP hücresi bir bloğa atanmıştır. Her blok, bloktaki veri noktalarının birbirlerine uzaklıklarını hesaplamaktadır. Her thread, verilen veri çiftinin aralarındaki mesafeyi bulmaktadır. Bloklarda her fonksiyonun eşzamanlı başlatılması ve birleştirme operasyonu paraleldir. Thread'ler sırasıyla, uzaklık hesabı, en yakın çiftin belirlenmesi, minimum uzaklık demet çiftinin güncellenmesi ve birleştirilmesi fonksiyonlarını çalıştırmışlardır. Deneylerde, CPU versiyonuna göre, GPU'da çalışan PoP-destekli HAC 2331 kat hızlanma sağlamıştır. 100000 verili veri kümesi kullanıldığında GPU'da PoP-destekli HAC, GPU'da klasik HAC'e göre 400 kat daha az bellek gerektirmiştir. Literatürdeki HAC'in en verimli GPU versiyonu budur.

Önceki bölümde bahsi geçen CAMPAIGN isimli çalışmayı gerçekleştiren Kohlhoff ve ark. (2011) (Kohlhoff, M.H.Sosnick, Hsu, Pande, & Altman, 2011) makalede hiyerarşik demetleme için kullanılan yöntemi raporlamamışlar fakat uygulamanın indirilebileceği link vermişler ve deneylere göre 5 kat hızlı olduğunuraporlanmıştır. Bu hızlanma mevcut çalışmalara göre çok düşüktür.

3.3. Bulanık C-Means (FCM) demetleme algoritması (Fuzzy C-Means Clustering Algorithm)

Bulanık c-means (FCM) 1973 yılında Dunn (Dunn, 1973) tarafından sunulmuş ve 1981 yılında Bezdek (Bezdek, 1981) tarafından geliştirilmiştir. Bulanık demetlemede, veri elemanları birden fazla demete ait olabilir. Her veri elemanı, bir üye değerleri kümesiyle ilişkilidir.

FCM algoritması 4 basit adım ile özetlenebilir (Pangborn, 2010):

(1) Rastgele M adet noktayı demet merkezi olarak seç.

(2) Her veri noktasının her demet için üyeliğini hesapla.

(3) Her demet için, üyeliği bu demete ağırlıklanmış noktalarını topla.

(4) Her demet merkezini demetin toplam üyeliğine bölerek yeniden hesapla.

(5) Durdurma kriterini kontrol et, sağlamadıysa adım 2'ye git.

Harris ve ark. (2005) (Harris & Haines, 2005) GPU tabanlı FCM algoritmasının uygulamasını OpenGL ve Nvidia'nın Cg gölgelendirici (*İng. shader*) dilini kullanarak gerçekleştirmişlerdir. Demet üyeliklerinin üretilmesi ve güncellenmesi adımları, GPU'da bir vektör güncelleme parça (*İng. fragment*) programı kullanılarak yapılmıştır. Deneylere göre, CPU versiyonuna göre 2 kat hızlanma sunmuştur. Yazarların uygulamayı geliştirdiği dönemde, donanım, parça programlar tarafından alınabilen doku sayısını kısıtlamaktaydı. Bu durum verilerin sayısına sınır getirmiştir, uygulama boyut ve demet sayısı bakımından ölçeklenebilir değildir. Anderson ve ark. (2007) (Anderson, Luke, & Keller, Analysis and Design of Intelligent Systems using Soft Computing Techniques, 2007) benzer bir uygulama

DOI: 10.5824/1309-1581.2013.2.002.x

geliştirmişlerdir; ek olarak bellek kısıtlarından kaçınmak için demet başına minimum doku kullanmaya ve portatifliği artırmak için gölgelendirici (*shader*) programlarının yeniden kullanımını maksimize etmeye çalışmışlardır. Ayrıca, mahalanobis ve GK uzaklık ölçütleri kullanmışlardır; bu ölçütler kovaryans matrisi gerektirmiştir. Deneylere göre CPU tabanlı uygulamaya göre 8'den 83 kata kadar hızlanma elde edilmiştir. Yazarlar bir sonraki çalışmalarında Anderson ve ark. (2008) (Anderson, Luke, & Keller, Speedup of Fuzzy Clustering Through Stream Processing on Graphics Processing Units, 2008) uygulamayı öklid uzaklık ölçütünü kullanarak gerçekleştirmişlerdir. Burada C-means 6 aşamalı bir geçişle uygulanmıştır. GPU P1 programı, her veri noktasının demet merkezlerine uzaklığını hesaplamıştır. P2, bu uzaklıkları alıp yeni üyelik değerlerini hesaplamıştır. P3, üyelik değerleri ile veri kümesinin nokta çarpımını gerçekleştirmiştir. P4, paylara azaltma; P5, paydalara azaltma uygulamıştır. P6, P4 değerlerini P5'te hesaplananlara bölerek demet merkezlerini güncellemiştir. Deneylerde CPU FCM versiyonuna göre 4 ila 88 kat arasında hızlanma sağlanmıştır. Yine benzer bir çalışmayı Shalom ve ark. (2008) (Shalom, Dash, & Tue, Graphics Hardware based Efficient and Scalable Fuzzy C-Means Clustering, 2008) büyük sayıda boyut ve demete ölçekleyebilme amacıyla gerçekleştirmişlerdir. Fakat, bu durum CPU'dan GPU'ya veri transferinin çok sık ve büyük miktarda olmasıyla performansı kısıtlamasıyla sonuçlanmıştır. Yazarlar algoritmanın uzaklık hesaplamaları, üyelik hesaplamaları, demet merkezlerinin hesabı gibi yinelemeli kısımlarını gölgelendirici programlar kullanarak parça işlemcide çalıştırmışlardır. Uzaklık ve üyelik matrislerini dokularda saklamışlardır. Deneylere göre, uygulama CPU uygulamasından 20'den 94 kata kadar daha hızlıymış.

Bu zamana kadar GPU tabanlı FCM uygulamalarında OpenGL, Cg, GLSL kullanılırken CUDA'nın popülerliğinin artmasıyla birlikte Espenshade ve ark. (2009) (Espenshade, Pangborn, Laszewski, & Roberts, 2009) çalışmalarında akış sitometri için minimum tanımlayıcı uzunluk çıkarsama (MDL) ile bulanık c-means (FCM) demetleme algoritmasının GPU tabanlı uygulamasını CUDA kullanarak geliştirmişlerdir. Uygulamada tüm paralel işlemler tek kerneldedir. Bu sayede global bellek erişimlerinin sayısı minimize edilmiştir ve bellek gereksinimi azaltılmıştır. Başlangıçta demet sayısı seçiminin sonucu çok etkilemesine çözüm olarak FCM'e MDL framework'ünü entegre etmişlerdir. FCM

algoritmasındaki eşitlikler, her demet için tamamen bağımsız olduğu için CUDA ile bu işlemler paralelleştirilmiştir. Uygulamada, FCM bir demet merkezi kümesine yakınsadıktan sonra, MDL Q matrisi üretilmiş ve devreye Tabu Arama girmiştir. Hangi demetin dahil edilmesi hangi demetin yok sayılması gerektiği tanımlanmıştır. Uygulamanın birkaç problemi vardır. Birincisi bloklar veri paylaşamadıkları için üyelik hesabı paydası için uzaklıklar yeniden hesaplanmaktadır. İkincisi, tek kernel olması thread başına çok fazla saklayıcı kullanımı gerektirmiştir. Bu durum doluluğu düşürmüştür. Üçüncüsü, bütünleşik olmayan çok fazla bellek okuması vardır. Dördüncüsü, geçici sonuçları tutmak için boyut sayısıyla orantılı paylaşılan bellek gerekmiştir. Bu durum, boyut sayısı arttıkça blok başına thread sayısının azaltılmasını gerektirerek doluluk azalmıştır. Deneylere göre, her FCM iterasyonu için gözlenen en yüksek hızlanma 84.34 kat olmuştur.

CUDA kullanarak akış sitometri için C-means veri demetleme algoritmasının birçok GPU ile hızlandırılmasını Pangborn (2010) (Pangborn, 2010) yüksek lisans tez çalışmasında gerçekleştirmiştir. Yazar, tek bir makine (düğüm) üzerinde birçok GPU kullanmıştır. Birçok düğüm arasında iletişim için MPI; her düğümdeki birçok thread için OpenMP; GPU için CUDA olmak üzere hibrit bir paralel ortam kullanmıştır. Yazar hesaplamayı 4 farklı CUDA kerneline ayırmıştır. Tüm kernellerde GPU kaynaklarından tamamen yararlanmak için demet sayısı en az GPU'daki çokişlemci (SM) sayısı kadar olmalıymış. Veri kümesi, demet merkezleri ve uzaklık matrisi global bellekte saklanmıştır. Veri kümesi sütun tabanlı düzene çevrilmiştir. Bu sayede, bütünleşik bellek okumaları sağlanmıştır. Host üzerinde veri kümesi, tüm OpenMP host thread'leri arasında paylaşılan bellek olarak işaretlenmiştir. Birçok cihazdan kısmi demet merkezlerini birleştirmek için host, her GPU'dan bir paylaşılan bellek alanı ayırmıştır. *UzaklıkMatrisi Kerneli*: Her thread bloğu, öklid uzaklık matrisinin 512 elemanını hesaplamıştır (thread başına 1 eleman). Demet merkezleri GPU'nun paylaşılan belleğinde önbelleklenmiştir. Tüm global bellek okumaları bütünleşik yapılmıştır. Üyelik hesaplamalarında 0'a bölme hatasını (veri noktasının demet merkezine eşit olduğu durum) önlemede koşullu dallanma koymamak için uzaklığa küçük bir 10^{-30} hata eklenmiştir. *Üyelik Kerneli*: İkinci kernel, ilk kernelde üretilen uzaklık değerlerini kullanarak sütun tabanlı üyelik değerleri matrisini üretmiştir. Üyelik denklemlerinin toplamındaki payda (bölen), tüm demetler için aynı olduğundan nesne başına sadece bir kez hesaplanmıştır. Her blok üyelik

DOI: 10.5824/1309-1581.2013.2.002.x

matrisinin 256 sütununu hesaplamıştır. Her thread, sorumlu olduğu veri noktasının demetlere uzaklıklarını toplamıştır. Sonra her uzaklığı tekrar yükleyerek üyelik değerini hesaplamıştır. Son üyelik değerleri, orijinal uzaklık matrisinde saklanmıştır. Böylece, bellek gereksinimi yarıya düşürülmüştür. *MerkezleriGüncelle Kerneli*: Kernelde, demetSayısı/4 x veriBoyutSayisi bloklu grid kullanılmıştır. Blok başına 4 demet merkezi hesaplanmıştır. Bu grid, veri kümesinin global bellekten yüklenmesini 4 kat azaltmıştır. Bir veri elemanı her thread için bir saklayıcıya yüklenmiştir. Her thread, 4 üyelik değeri üzerinde çalışıp yaptığı hesabı paylaşılan bellekteki her demet merkezine eklemiştir. Tüm verilerin işlenmesinden sonra, kısmi toplamlar tek bir değere azaltılmıştır. Sonraki küçük kernel, üyelik matrisini demet sayısı uzunluklu vektöre azaltarak demet boyutlarını hesaplamıştır. Her GPU denklemin hem paylarını hem paydalarını döndürmüş ve host, yeni demet merkezlerini hesaplamak için tüm GPU'lardan sonuçları birleştirmiştir. Deneylerde, CPU referans uygulamasına göre elde ettikleri hızlanma 106 kat olmuştur. Tesla-güçlendirilmiş bir süper bilgisayar kullanıldığında 32 GPU ile tek bir CPU referans uygulamasına göre, yazarın GPU uygulaması %85 paralel hızlanma verimliliği ve 2368 kat hızlanma göstermiştir.

3.4. K-medoids (PAM) demetleme algoritması (K-medoids Clustering Algorithm)

K-medoids (PAM) algoritması, 1987 yılında Kaufman ve Rousseeuw (Kaufman & Rousseeuw, Clustering by Means of Medoids in Statistical Data Analysis Based on the L1–Norm and Related Methods, 1987) tarafından önerilmiştir. K-medoids algoritmasının adımları şunlardır:

1. Demetleri temsil eden k tane rastgele nokta (medoid) seç.
2. Her noktayı en yakın demete ata.
3. Medoid olmayan rastgele bir nokta seç.
4. Seçilen nokta medoid olması durumunda toplam karesel hatadaki değişimi hesapla.
5. Hesaplanan değişim 0'dan küçükse seçilen noktayı medoid yap.
6. Herhangi bir yakınsama kriteri karşılanana dek 4. adıma git.

Demet sayısı k önceden belirlenir. PAM, sapan verilerden k-means algoritmasına göre daha az etkilenir. PAM'da her iterasyon için karmaşıklık n veri noktası sayısı olmak üzere $O(k(n-k)^2)$ olmaktadır.

Espenshade ve ark. (2009) (Espenshade, Pangborn, Laszewski, & Roberts, 2009) çalışmalarında bayes bilgi kriteri (BIC) ile k-medoids demetleme algoritmasının GPU ile hızlandırılmış bir uygulamasını CUDA ve Tesla mimarisinden yararlanarak sunmuşlardır. K-medoids algoritmasında demetleri bulanıklaştırmışlardır. Bayes bilgi kriteri (BIC) belirli bir veri seti için en iyi demet sayısını belirlemek amaçlı k-medoids algoritmasına entegre edilmiştir. Uzaklık ölçütü öklid kullanılmıştır. Deneylere göre, BIC ile k-medoids tabanlı GPU uygulaması, CPU versiyonuna göre 7 kata kadar hızlanma sağlamıştır.

Kohlhoff ve ark. (2011) (Kohlhoff, M.H.Sosnick, Hsu, Pande, & Altman, 2011) çalışmalarında k-medoids için kullandıkları yöntemi raporlamasalar da uygulamanın indirilebileceği bir link verip deneylere göre 102 kat hızlanma elde edildiği raporlamışlardır.

3.5. CAST demetleme algoritması (CAST clustering algorithm)

CAST (*İng. Clustering Affinity Search Technique*) 1999 yılında Ben-Dor ve ark. (Ben-Dor, Shamir, & Yakhini, 1999) tarafından geliştirilmiştir. CAST, biyolojik veri demetlemede yaygın bir şekilde kullanılan hem bölünmeli hem yoğunluk tabanlı bir demetleme algoritmasıdır.

CAST, nesne benzerliğini saklamak için bir benzerlik matrisi hesaplamaya ihtiyaç duyar. CAST, tüm nesnelerin uzaklıklarını saklamak için bir benzerlik matrisi ve bir birleşme eğilimi (*İng. affinity*) eşik değeri giriş parametresine sahiptir. Algoritma şu şekilde çalışır: İlk önce demetler için bir C kümesini ve demetlenmemiş nesneleri içeren bir U kümesini ilklendirir. C'deki her demet için her nesnenin birleşme eğilimi hesaplanır. U'daki her nesne için, nesne ile hedef nesne arasındaki benzerlik hesaplanır ve benzerlik değerleri, hedef nesnenin birleşme eğilimi olarak toplanır. Eğer birleşme eğilimi, birleşme eğilimi eşik değerine eşit veya daha büyükse, bu nesne, aktif demete eklenebilir ve U'da "*demetlendi*" olarak işaretlenebilir. Aynı zamanda, demetteki her nesnenin birleşme eğilimi, nesne ile yeni eklenen nesne arasındaki benzerliği ekleyerek güncelleyebilir. Eğer birleşme eğilimi,

DOI: 10.5824/1309-1581.2013.2.002.x

birleşme eğilimi eşik değerinden küçükse, en düşük birleşme eğilimli nesne, bu demetten kaldırılacaktır ve bu nesne U'da "*demetlenmedi*" olarak işaretlenir. Demetteki her nesnenin birleşme eğilimi nesne ile kaldırılan nesne arasındaki benzerliği çıkartarak güncellenebilir. Ekle ve çıkar işlemleri, değişiklik olmayana kadar tekrarlanır. Aktif demet için demetleme de sonlanır. Algoritma, adımları tekrarlayarak yeni demet bulma için diğer demetlemeye başlar. U nesnelerinin hepsi demetlendi olarak işaretlenince sonlanır (Lin & Lin, 2011).

Lin ve Lin (2011) (Lin & Lin, 2011) makalede CAST demetleme algoritmasının iki versiyonunu önermişlerdir: talep üzerine hesaplama CAST (kısaca COD-CAST) ve GPU ile talep üzerine hesaplama CAST (kısaca COD-CAST-GPU). COD-CAST algoritması, büyük miktarda nesneyi, çalışma zamanı bakımından daha verimli işleyebilen bir CAST algoritmasıymış. COD-CAST-GPU algoritması ise COD-CAST'i hızlandırmak için GPU'dan yararlanıyormuş. Yazarlar, CAST için benzerlik matrisi elde etme aşamasının çok zaman tükettiğini belirterek, özellikle nesne sayısı çok iken performans üzerinde bir darboğaz oluşturduğunu vurgulamışlar. Önerilen COD-CAST algoritmasının girişi, n nesneymiş (nxn benzerlik matrisi değilmiş). Çok geniş bir veri kümesini demetlemek için CAST kullanırken, genellikle tüm matrisi belleğe yüklenemiyormuş. Bu nedenle önceden benzerlik matrisini hesaplamıyorlarmış gerektiğinde hesaplıyorlarmış. Öklid uzaklık ölçütü kullanılmış. Önerilen COD-CAST-GPU algoritması ise COD-CAST için önerdikleri birleşme eğilimi eşik değerini güncelleme fonksiyonunun GPU ile paralelleştirilmesini içeriyormuş. Deneylere göre, veri sayısı az olduğunda önerilen algoritmaların hızlandırma kat sayısı çok az iken, örneğin 128.000 veri olduğunda COD-CAST-GPU, CAST'ın çalışma zamanının sadece %7.7'si kadar süre gerektiriyormuş.

3.6. K-centers demetleme algoritması (K-centers clustering algorithm)

K-centers algoritmasında başlangıçta merkez 0 olarak gösterilen, rastgele bir nokta başlangıç merkezi olarak seçilir. Sonra, aktif merkeze tüm N noktanın uzaklıkları hesaplanır ve merkez 0'dan en uzaktaki nokta Y bulunur. Y noktası, bir sonraki iterasyon için yeni merkez haline gelir. Sonra yeni merkeze tüm N noktanın uzaklıkları hesaplanır ve en yakın merkezden daha yakın olup olmadığına bakılır. Bu durumda, noktaya bu yeni merkez atanmış olur. Daha sonra, önceden bulunan tüm merkezlerden en uzak nokta bulunur. Bu

işlem k iterasyon veya bir minimum demet yarıçapı kriteri karşılanana dek tekrarlanır. Demet merkezlerinin sayısı önceden belirtilir (Zhao, Sheong, Sun, Sander, & Huang, 2013).

Kohlhoff ve ark. (2011) (Kohlhoff, M.H.Sosnick, Hsu, Pande, & Altman, 2011) çalışmalarında k-centers için kullandıkları yöntemi raporlamasalar da uygulamanın indirilebileceği bir link verilerek deneylere göre 178 kat hızlanma elde edildiği raporlanmıştır.

Zhao ve ark. (2013) (Zhao, Sheong, Sun, Sander, & Huang, 2013) moleküler dinamik (MD) simülasyonlarının protein biçimleri üzerinde k-centers algoritmasının CUDA ile hızlandırılan paralel uygulamasını sunmuşlardır. Uygulamada, bir merkezle başlamışlar ve iteratif olarak önceden bulunan demet merkezlerinin hepsinden en uzaktaki noktayı bularak k-demet merkezlerinin bir toplamına yaklaşmışlardır. İki protein biçimi arasındaki uzaklığı hesaplamak için Theobald'ın dördey karakteristik polinom (QCP) yöntemini uygulamışlardır. QCP yönteminin GPU uygulaması, hesaplama başına karışık tek ve çift hassasiyetli ondalıklı sayı operasyonlarından yüz binlerce gerektiriyormuş. Uzaklık hesaplamalarının toplam sayısını azaltmak amacıyla üçgen eşitsizliği ilkesinden yararlanmışlardır. Fakat bu yol her iterasyonun başında merkezden merkeze uzaklıkları hesaplama maliyeti getirerek ekstra bellek ve zaman almıştır. Üçgen eşitsizliği ek yükten getirse de MD simülasyonlarında demetlemede üçgen eşitsizliğini sağlayan noktalar çok olduğu için performansa katkısı çok olmuştur. Veri kümesini sütun-tabanlı düzende global bellekte saklamışlardır; bu sayede thread'ler peş peşe bellekten okuma ve belleğe yazma yapabilmişlerdir. Demet merkezlerini global bellekte saklamışlardır. Tüm merkezlerden en uzaktaki noktayı bulmak için paralel bir azaltma kerneli uygulamışlardır. GPU'da thread'lerin boş kalmasını en aza indirmek için tarama-tabanlı sıkıştırma kerneli yazmışlardır. Bu kernelde, üçgen eşitsizliğini sağlamayan noktalar belirlenip bir diziye sıkıştırılmıştır. Bu sayede thread ayrılma da önlenmiştir. Üçgen eşitsizliği tek başına veri kümesinin demetlenmesini 2 kattan 17 kata kadar hızlandırmıştır. Üçgen eşitsizliği, seyrek bölgelerle ayrılmış yoğun bölgeleri olan veri kümeleri üzerinde (yoğun veri kümesi) benzer tarzda dağılmış veri kümelerinden (tekdüze veri kümesi) daha iyi performans sergileme eğilimindeymiş. Yazarlar, makalede üçgen eşitsizliğinin yüksek boyutlu veri kümeleri

DOI: 10.5824/1309-1581.2013.2.002.x

kullanılırken daha az verimli olduğunu gözlemlemiştir. Deneylere göre CPU uygulamasına göre 42 kattan 100 kata kadar hızlanma sunmuştur. Yazarlar, özetle bellek erişim düzenlerini optimize etmişler, GPU belleğinden bütünleşik okuma ve yazmalar yapmaya dikkat etmişler, paralel azaltma uygulamışlar, uzaklık fonksiyonunu paralelleştirmişler, sıkıştırma ile üçgen eşitsizliği uygulamışlardır.

3.7. DBSCAN demetleme algoritması (DBSCAN clustering algorithm)

DBSCAN algoritması, Ester ve ark. (Ester, Kriegel, Sander, & Xu, 1996) tarafından 1996 yılında sunulmuştur. Algoritma *Eps* ve *MinPts* olmak üzere iki parametre gerektirir. Eps, en büyük komşuluk yarıçapı, MinPts ise bir veri noktasının *"çekirdek nokta"* olması için komşuluk bölgesinde bulunması gereken en az veri noktası sayısıdır. DBSCAN algoritması öncelikle birtakım tanımlar verilerek anlatılmıştır.

Tanım 1 **: *(Bir noktanın Eps komşuluğu)*** Bir p noktasının Eps komşuluğu $N_{Eps}(p) = \{ q \in D \mid dist(p,q) \leq Eps \}$ ile tanımlanır. Tanımdaki $dist(p,q)$ fonksiyonu p ve q noktaları arasındaki uzaklığı ifade eder.

Tanım 2 **: *(Doğrudan yoğunluk erişilebilir)*** Eps ve MinPts koşulları altında, aşağıdaki şartlar sağlanıyorsa p noktası q noktasından doğrudan yoğunluk erişilebilirdir:

1. $p \in N_{Eps}(q)$
2. $|N_{Eps}(q)| \geq MinPts$ (q'nun çekirdek nokta olması koşulu)

Tanım 3 **: *(Yoğunluk erişilebilir)*** Eps ve MinPts koşulları altında, $p_1 = q$ *ve* $p_n = p$ olmak üzere $p_1, p_2, \ldots, p_n$ noktalar zinciri varsa ve bu zincirde p_{i+1} noktası p_i'den doğrudan yoğunluk erişilebilirse) p, q'dan yoğunluk erişilebilirdir.

Tanım 4 **: *(Yoğunluk bağlantılı)*** Eps ve MinPts koşulları altında, hem p hem de q bir o noktasından yoğunluk erişilebilirse p noktası q noktasına yoğunluk bağlantılıdır.

Tanım 5 **: *(Demet)*** D veri kümesi ve C bir demet olmak üzere, Eps ve MinPts koşulları altında C, D'nin boş olmayan ve aşağıdaki koşulları sağlayan bir alt kümesidir:

1. $\forall p, q$: Eğer $p \in C$ ve *(Eps ve MinPts koşulları altında)* p, q'dan yoğunluk erişilebilir ise, $q \in C$.

2. $\forall p, q \in C$: *(Eps ve MinPts koşulları altında)* p, q'ya yoğunluk bağlantılıdır.

Tanım 6 : (Gürültü) Eps_i ve $MinPts_i$ i=1,2,…,k koşulları altında $C_1, C_2, \ldots, C_k$ D veri kümesinin demetleri olsun. D veri kümesinde herhangi bir C_i demetine ait olmayan noktalar kümesi "*gürültü*" olarak tanımlanır, $gürültü = \{ p \in D \mid \forall i : p \notin C_i \}$.

DBSCAN algoritmasında rastgele bir noktadan başlanarak tüm noktalar kontrol edilir. Eğer nokta, önceden bir demete eklendiyse işlem yapılmadan sonraki noktaya geçilir. Aksi takdirde, noktanın komşuluğundaki noktalar bulunur. Komşu sayısı MinPts'den küçükse gürültü olarak işaretlenir ve sonraki noktaya geçilir. Komşu sayısı MinPts'den büyük veya MinPts'ye eşitse bir demet oluşturulur ve demete bu nokta ve komşuları eklenir. Sonra önceden bir demete eklenmemiş her bir komşu için komşuluğu araştırılarak onun komşuları bulunur. Komşuluğu araştırılan noktaların komşu sayıları MinPts'den büyük veya MinPts'ye eşitse demete eklenir. Bu işlemler eklenecek nokta kalmayana dek devam eder. Sonra veri kümesinden başka bir nokta seçilerek döngü tekrarlanır.

Böhm ve ark. (2009) (Böhm, Noll, Plant, & Wackersreuther, Density-based clustering using graphics processors, 2009) DBSCAN demetleme algoritmasının GPU tabanlı paralel bir uygulaması CUDA-DClust'ı ve benzerlik bulmada bir indeks yapısı kullanan daha da hızlandırılmış CUDA-DClust* versiyonunu sunmuşlardır. Uygulama, zincir kavramıyla desteklenen paralel demet genişletmeye; hiyerarşik bir indeks yapısı kullanımıyla hızlandırılabilen paralel en yakın komşu bulmaya; mikroişlemciler arasında verimli yük dengelemeye dayanmaktadır. Doğrudan yoğunluk erişilebilirlik ilişkisinin geçişli kapalılık hesaplaması zincir kavramıyla uygulanmıştır. Her SM bir demetin genişletilmesi için ayrılmıştır. Birçok demet genişletilmesi, farklı başlangıç noktalarından farklı zincirler aracılığıyla aynı anda başlatılmıştır. Genişletilecek tohum nokta paylaşılan belleğe yüklenmiştir. Bir tohum noktası, çekirdek nesne özelliğini belirlemek ve onun komşularını yeni potansiyel noktalar olarak işaretlemek için ele alındığında, tohum noktanın potansiyel komşularını eşzamanlı işlemek için birçok thread üretilmiştir. Koordinatlar saklayıcılara

DOI: 10.5824/1309-1581.2013.2.002.x

yüklenmiştir. Ana program, işlenmeyen hiç bir nokta kalmayana dek GPU üzerinde 3 kernel başlatan bir döngüyü içermektedir: (1) CPU'dan GPU'nun global belleğine veri kümesi transfer edilir. (2) Veri kümesinden rastgele seçilen noktalardan yeni zincirler oluşturulur. (3) Demet Genişletme kerneli başlatılır. (4) Zincirlerin durumları GPU'nun global belleğinden CPU'ya transfer edilir. (5) Gerekiyorsa YeniTohumlar kerneli başlatılır. (6) Gerekiyorsa TohumYenidenDoldur kerneli başlatılır. (7) İşlenmemiş nesneler varsa 3. adımdan devam edilir. (8) Demet-ID'leri GPU'nun global belleğinden CPU'ya transfer edilir. *Demet genişletme kerneli*, CUDA-DClust'ın temel kernel metodudur ve çekirdek nokta özelliğinin belirlenmesini ve demetlerin geçişli genişletmesini uygulamaktadır. Tohum listesinin amacı, aktif demetin bir çekirdek nesnesi olduğu onaylanmış herhangi bir nesnesinden doğrudan yoğunluk erişilebilir nesneler için bir bekleme kuyruğu sağlamaktır. CUDA-DClust eşzamanlı çok sayıda zincirin demet genişletmesini uyguladığı için çoklu tohum listelerine sahiptir. Alan problemi yaşanmaması için bir tohum listesinin sahip olabileceği alan 1024 nokta ile kısıtlanmıştır. Zincirlerden biri bittiğinde, işlenmemiş bir nesneden yeni bir zincir başlatılmıştır. İşlenmemiş nesne yoksa bir zincir bölünmüştür. Bu sayede, sistemde çalışan neredeyse sabit sayıda thread'in hepsi aynı iş yüküne sahip olmuş ve yük dengesi sağlanmıştır. CUDA-DClust'ın performansını geliştirmek için, nesnelerin komşularını bulmayı sağlayan çoklu bir indeks yapısı önerilmiştir. CUDA-DClust* için demetleme yöntemini başlatmadan önce verinin parçalara ayrılması ve sıralanması, önerilen indeks yapısının kurulması gerekmiştir. İndeks kurulumu CPU'da uygulanmıştır. Veri kümesinin yanında dizin de GPU'nun global belleğine transfer edilmiştir. Bir nesnenin komşuları belirlenirken, thread kümesinin her biri veri kümesinin farklı bir parçası üzerinde çalışmıştır. Deneylere göre, CUDA-DClust, indeks desteği olmadan CPU uygulamasına göre 10-15 kat; CUDA-DClust* indeks destekli CPU uygulamasına göre 3.5-15 kat hızlanma sunmuştur. Sonraki makalelerinde Böhm ve ark. (2009) (Böhm, Noll, Plant, Wackersreuther, & Zherdin, Transactions on Large-Scale Data- and Knowledge-Centered Systems I, 2009) yine DBSCAN için GPU programlama modeline uyan bir dizin yapısı ve benzerlik birleştirme yöntemi tanımlamışlar; fakat daha farklı bir şekilde uygulamayı geliştirmişlerdir. DBSCAN'i paralel benzerlik birleştirme ile desteklemişlerdir. Çok boyutlu dizin yapısı, geniş veri kümelerinde verilen bir sorgu nesnesine benzeyen nesneleri buluyormuş. Benzerlik

birleştirme yöntemi ise benzerlik bulmada kullanılıyormuş. CUDA kullanarak geliştirdikleri GPU-destekli DBSCAN uygulamasında algoritmanın çekirdek nesne özelliğinin belirlenmesi aşamasını ve doğrudan yoğunluk erişilebilirlik ilişkisinin geçişli kapalılığını hesaplayarak demet genişletme aşamasını paralelleştirmişler. İlk aşamayı benzerlik birleştirme ile desteklemişler. Çekirdek nesne özelliğini kontrol etmek için, her noktanın € komşuluğu içindeki nesnelerin sayısı hesaplanıyormuş. Bir (x,q) nesne çifti birleşme koşulunu sağladığında, nesnelerin sayısını arttırmada sadece q noktasının sayacını arttırmışlar. q noktası, thread'le ilişkilendirilen nokta olduğu için sayıcı[threadID] senkronize edilmemiş sıradan *inc()* operasyonu ile güvenli bir şekilde arttırabiliyormuş. Bu yöntem, sadece 1 sayacı arttırdığı için tüm nesneleri kontrol gerektirmiş. Uygulamada, veri kümesini ve veri noktası adedi büyüklüğündeki sayıcı matrisi global bellekte saklanmış. Her noktadan bir thread sorumluymuş. Her thread sorumlu olduğu noktayı global bellekten kendi saklayıcısına kopyalıyormuş. İteratif bir şekilde global bellekten bir veri noktası x, paylaşılan belleğe yükleniyormuş. Senkronize edildikten sonra bloktaki thread'ler x veri noktasıyla, sorumlu oldukları veri noktasının arasındaki mesafeyi hesaplıyorlarmış ve epsilon'dan küçük eşit olup olmadığını kontrol ediyorlarmış. Şart sağlanırsa, thread'in sorumlu olduğu noktanın sayıcısı inc() ile 1 arttırılıyormuş. Demet genişletme için doğrudan yoğunluk erişilebilirlik ilişkisinin geçişli kapalılığını hesaplamak gerekiyormuş. Geçişli kapalılığı hesaplamada Floyd-Warshall'ın GPU'da yüksek derece paralel versiyonu kullanılmış. Deneylere göre, GPU destekli DBSCAN, CPU versiyonuna göre 80 katı aşkın hızlanma sağlamış.

Çok büyük veritabanlarında DBSCAN kullanırken daha iyi bellek ölçeklenebilirliğine imkân tanıyan bir iyileştirme öneren Thapa ve ark. (2010) (Thapa, Trefftz, & Wolffe, 2010) ise GPU tabanlı DBSCAN programını CUDA kullanarak uygulamışlardır. Uygulamada, çekirdek nokta tespit edildiğinde Eps komşuluk noktalarının her biri bir kuyruğa atılıyormuş ve sırasıyla onların komşularının Eps uzaklığı içine düşüp düşmediği kontrol ediliyormuş. Kuyruktaki her nokta boyunca ilerleyip, demetteki her çekirdek noktadan yoğunluk erişilebilir olan tüm noktaları bularak algoritma bir demeti kurmaya başlıyormuş. Herhangi bir noktanın Eps komşuluğundaki komşularını bulmak için calcRow() kernelini yazmışlar. calcRow(), noktalar arasındaki uzaklıkları belirlemek için GPU üzerinde processPoint() kernelini çağırıyormuş. calcRow() fonksiyonu tarafından hesaplanan noktalar,

DOI: 10.5824/1309-1581.2013.2.002.x

bitişiklik matrisi ile temsil edilmiş. Bitişiklik matrisi i ve j noktaları birbirine bitişikse (Eps veya daha az uzaklıktaysa) '1'; değilse (aralarındaki uzaklık Eps'den büyükse) '0' içeriyormuş. Yazarlar, bitişiklik matrisini bir kerede hesaplamak için herhangi bir anda birden çok veri noktasının diğer noktalarla karşılaştırılması paralelleştirmişler. Makalede iki GPU uygulaması yer almış: ilk GPU uygulaması ve yeni GPU uygulaması (M-GPU). İlk GPU yaklaşımında, her thread'e bir veri noktası veriliyormuş. Bu veri noktasının, veri kümesindeki diğer tüm noktalara karşı karşılaştırması hesaplanıyormuş. n adet eşzamanlı çalışan karşılaştırmaların her birinde nokta sırayla veri kümesindeki diğer noktalara karşı karşılaştırılıyormuş. Bu yaklaşımının bellek gereksinimi $O(n^2)$ mertebesindeymiş. M-GPU'da, bir noktanın veri kümesindeki diğer noktalarla karşılaştırılması eş zamanlı yapılıyormuş; sonraki her nokta kendi karşılaştırmalar kümesini uygulamak için sırasını seri bir şekilde bekliyormuş. Bu yaklaşımın avantajı bitişiklik matrisinin sadece bir satırına ihtiyaç duyması ve böylece bellek talebini O(n) mertebesine azaltmasıymış. calcRow() bir kez tamamlandığında, sonuçlanan dizi satırı ve komşuların sayısının toplamı, cihaz belleğinden host belleğe kopyalanıyormuş. Bu bilgi, expandCluster() tarafından demeti genişletmek için kullanılıyormuş. İki GPU uygulaması da, orijinal seri versiyona göre 2-3 kat daha hızlı çalışmış. İlk GPU yaklaşımı, büyük veri setleri için M-GPU'dan biraz daha uzun sürüyormuş.

3.8. EM (Expectation Maximization) demetleme algoritması (Expectation Maximization clustering algorithm)

EM algoritması her demeti bir olasılık dağılımı ile ifade eder. EM algoritması, E ve M adımlarından oluşur. E-adımında önceki iterasyondan Gauss model parametreleri kullanılarak her veri noktası için demet üyelikleri hesaplanır. Başlangıçta tahmini model parametreleri kullanılır. Her veri belli bir olasılık dağılımına göre demete atanır. M adımında yeni üyelikler kullanılarak parametreler güncellenir. Her EM aşamasından sonra sadece tek bir demet kalana dek veya belli bir demet sayısına ulaşana dek en benzer iki demet birleştirilir (Pangborn, 2010).

Kumar ve ark. (2009) (Kumar, Satoor, & Buck, 2009) EM demetleme algoritmasının GPU ile paralelleştirilmiş hızlı bir uygulamasını CUDA kullanarak sunmuşlardır. Yazarlar,

önce algoritmayı basit bir seviyeye ayrıştırıp her seviyedeki veri paralelliğini belirleyip hızlandırmışlardır. Son aşamada güncelleme denklemlerini matris dönüşümleri olarak ifade etmişlerdir. Her iterasyon için gereken matrislerin hesabını yapan 6 kernel yazmışlardır. Yazarlar kernellerde 6-11 arasında saklayıcı kullanmışlardır. Tüm matrisler global bellekte sütun tabanlı düzende saklanmıştır. Bu sayede, veriler bütünleşik bellek okumalarıyla paylaşılan belleğe yüklenmiştir. Deneylere göre, uygulama CPU uygulamasına göre 164 kat daha hızlı çalışmıştır. Çalışmada bloklara eşit iş dağıtımı yapılarak yük dengelemeye dikkat edilmiştir. Böylece bazı çekirdeklerin boşta kalıp performansı kötü etkilemesine engel olunmuştur. Yazarlar, kovaryans matrislerinin diyagonal olduklarını varsaymış; ancak bu durum boyutların istatistiksel olarak birbirinden bağımsız olmalarına imkan tanımamaktadır. Gauss karışım modelleriyle EM veri demetleme algoritmasının uygulamasını yapan bir başka çalışmada Pangborn (2010) (Pangborn, 2010) farklı olarak GPU'larla donatılmış bir PC küme sistemi kullanmıştır. Uygulamada, giriş verileri global bellekte sütun-tabanlı düzende saklanmıştır. Bu sayede, bütünleşik bellek okumalarına imkan sağlanmıştır. Veri kümesi GPU'lara eşit olarak bölünerek iş yükü dağıtılmıştır. Uygulama kök düğümdeki ana sistemin (*İng. host*) model parametrelerini ilklendirmesiyle başlamıştır. Her host thread, veri kümesinin kendi bölümünü ve ilgili GPU'ya başlangıç gauss model parametrelerini kopyalamıştır. E-adımı 2 kernele bölünmüştür. E-Adım1, bir demete ait bir veri noktasının log-olasılığını hesaplamakta ve ağırlıklandırmaktadır, parametreleri paylaşılan bellekte önbelleklemektedir. E-Adım2, tüm demetlerde üyelik için ağırlıklandırılmış her olasılığı bulanık bir olasılığa çevirmektedir. M adım, her denklem için bir kernel olmak üzere 3 CUDA kerneline bölünmüştür. Çok GPU'lu uygulamada kernel kovaryans yerine her eleman için toplam varyansı hesaplamaktadır. Host her GPU'dan kısmi sonuçları topladıktan sonra varyansı demet boyutuna bölmekle sorumludur. Üç M-adım kernelinden sonra kovaryans matrisine çeviren, demet olasılıklarını hesaplayan, E-adımındaki tüm olasılık hesaplamaları tarafından paylaşılan bir sabiti hesaplayan başka bir kernel başlatılmaktadır. EM aşaması tamamlandıktan sonra karışım modelindeki en benzer iki demet, kök düğümün master thread'i tarafından birleştirilmektedir. Deneylerde, CPU referans uygulamasına göre 73 kat hızlanma elde edilmiştir. 128 GPU'lu Tesla-güçlendirilmiş süper bilgisayar kullanıldığında %72 verimlilik ve 6286 kat hızlanma elde edilmiştir.

DOI: 10.5824/1309-1581.2013.2.002.x

3.9. Sürü doküman demetleme algoritması (Flocking document clustering algorithm)

Doküman demetleme problemi tabanlı sürü (İng. *flocking*) yönteminde zaman karmaşıklığı n^2'dir. Sürü modeli, bir varlık sürüsünün hareketini taklit etme için biyolojik olarak esinlenmiş hesapsal bir modeldir. Kuş ve balık sürülerinde görüldüğü gibi grup hareketini temsil eder. Her birey, diğerleriyle herhangi bir iletişim olmaksızın sadece sürüdeki komşu üyelere ve çevresel engellere bağlı olarak az sayıda basit kurala göre davranırlar. Craig Reynolds'ın sürü modelinde 3 basit yönetim kuralı vardır (Cui, Charles, & Potok, 2012):

1) ***Ayrılma:*** Komşularla çarpışmayı önlemek için yönetim.

2) ***Hizalanma****:* Ortalama rota ve komşuların süratiyle eşleşmeye göre yönetim.

3) ***Birleşme****:* Komşuların ortalama pozisyonuna göre yönetim.

Zhang ve ark. (2011) (Zhang, Mueller, Cui, & Potok, 2011) doküman demetleme problemi tabanlı sürü uygulamasını CUDA-destekli GPU'larla donatılmış bir küme bilgisayar üzerinde gerçekleştirmişlerdir. Yazarlar, iki temel probleme odaklanmışlar. İlk olarak, doküman demetleme algoritmalarında doküman benzerliğini saptamada yararlanılan dokümanların TF-IDF vektörlerini hesaplamayı, ikinci adımda ise TF-IDF-benzeri benzerlik ölçütüne dayanarak bir seferde en az 1 milyon dokümanı demetlemeyi hedeflemişlerdir. Çoklu-Tür Sürü (MSF) simülasyonu uygulamışlardır. TF-IDF kavramı herhangi iki doküman arasındaki benzerliği ölçmek için kullanılmaktadır. Yazarlar, küme uygulamalarına daha uygun frekans-ters esas frekans (TF-ICF) denilen yeni bir terim ağırlıklandırma planı önermişlerdir. TF-ICF, işlenilen doküman koleksiyonları içindeki diğer dokümanlardan terim frekans bilgisi gerektirmiyormuş ve örneklemeyle ICF tablosunu önceden oluşturuyormuş. Sürü simülasyonunun temeli komşuluk bulmaymış. Komşuluk bulma için sanal simülasyon alanını dilimlere bölmüşler. Her düğüm, sadece aktif dilimde bulunan dokümanları ele almış. Düğümden düğüme mesajlarla doküman pozisyonları iletilmiş. Her iterasyonda doküman pozisyonlarının güncellenmesinden sonra tüm dokümanlar *geçiş yapan, komşu ve iç doküman olmak üzere* sınıflandırılmış. Yazarlar, doküman vektörlerini, TF-ICF tablosunda her kelimenin indeksine göre sıralı bir dizide saklamışlar. Bu

veri yapısı minimum bellek kullanımı sağlamış. İki dokümanın benzerliğini hesaplamak için bir kernel yazılmış. Her thread bloğu, 1 doküman çifti almış. Doküman vektörleri eşit bir şekilde bloktaki thread'lere bölünmüş. Her thread, atanan her TF-ICF değeri için diğer doküman vektörünün aynı indeksli girdi içerip içermediğini belirliyormuş. İkili (İng. *binary*) arama uygulanıyormuş. Her iterasyonun başlangıcında, her thread, komşuluk yapan, geçiş yapan dokümanların pozisyonlarını ve vektörlerini elde etmek için komşularına iki mesaj veriyormuş. Sonra, her iç doküman ve geçiş yapan doküman için belirli bir aralık içindeki komşu dokümanlarını arayan bir komşu bulma fonksiyonu devreye giriyormuş. Tüm komşular bulunduğunda, aktif thread'e ait ve onların saptanan komşularına ait dokümanların benzerlikleri hesaplanıyormuş. Sürü kurallarının uygulandığı sonraki adımda doküman pozisyonlarını güncellemek için benzerlik ölçümleri kullanılmış. Deneylerde, GPU kümeleri CPU kümelerinden 30'dan 50 kata kadar üstün performans sergilemiş.

Cui ve ark. (2012) (Cui, Charles, & Potok, 2012) doküman demetleme problemi tabanlı sürü uygulamasını CUDA ile hızlandırmışlardır. Reynold'un sürü modelindeki 3 kural tüm bireylerin tek bir sürüye dönüşmesiyle sonuçlandığı ve yazarlar doğadaki iki veya daha fazla farklı tür sürülerine dönüşmeyi taklit için yeni bir Çoklu Tür Sürü (MSF) modeli sunmuşlar. Bu amaçla, sürü modelindeki kurallara 4. bir kural *-nitelik benzerlik kuralı-* eklemişler. Buna göre bir birey aynı niteliklere sahip bireylere yakın kalmaya çalışırken farklı niteliklere sahip bireylerden uzak duruyormuş. Dokümanlara bireyler gibi davranılıyormuş ve demetlerken MSF modeli kullanılıyormuş. *GPU tabanlı doküman sürü algoritması* için 2 kernel yazmışlar. *İlk kernel*, her doküman çiftine bir thread veriyormuş (toplamda n^2 thread) ve onlar arasındaki uzaklığın komşuluk içinde olup olmadığını belirliyormuş. Aralarındaki uzaklık eşikten küçükse global bellekteki kosinüs benzerlik matrisine göre doküman karşılaştırma yapılıyormuş. Uzaklık değeri yeterince küçükse dokümanlar benzer farz edilip dokümanlar sürü arkadaşı kabul ediliyormuş. Benzer dokümanlar ayrılma, birleşme ve hizalanma kurallarını kullanarak; benzemeyen dokümanlar sadece ayrılma kuralı kullanarak rotalarını belirliyorlarmış. Popülasyon üzerinde etkisi olan her doküman hesaplanınca, ikinci kernel çalışıyormuş. *İkinci kernel*, n thread üretiyormuş, her thread bir dokümanın rotasını ve pozisyonunu güncelliyormuş. Burada, simülasyona rastgelelik eklenmiş, hareket hesaplamalarının %15'i rastgeleymiş. Sisteme rastgele eleman ekleme, dokümanların diğer

DOI: 10.5824/1309-1581.2013.2.002.x

sürü arkadaşlarının araştırmasındaki çözüm uzayını uygun bir şekilde araştırmasını sağlıyormuş. İkinci kernel çalışmayı bitirdiğinde, bir nesil sonlanarak sıradaki başlıyormuş. Uygulamada, GPU'nun global belleğinden sık okumalar yapılmış. Hızlı erişim için bazı doküman terimleri paylaşılan bellekte tutulmuş. Deneylerde, 1000 dokümanlı GPU uygulaması CPU versiyonundan yaklaşık 60 kat hızlı çalışmış. 1000 dokümandan sonra performans neredeyse doğrusal bir şekilde düşmüş. Bu durum popülasyon büyüdükçe global bellek erişim gecikmesinin artışından kaynaklanmıştır.

4. Sonuç (Conclusion)

Algoritmaların GPU ile güçlendirilmiş versiyonlarını uygulamak için araştırmacılar, ele aldıkları algoritmaların çok zaman tüketen adımlarını belirleyip paralelleştirilebilir olup olmadıklarını incelemişlerdir. Paralelleştirilebilir bölümler, GPGPU arayüzünün kısıtlarına, sağladığı esnekliklerine, bellek yapılarına, işlemler için sunduğu avantaj ve dezavantajlara bağlı tasarlanmış ve performansı artıracak yaklaşımlar uygulanmıştır.

Literatürdeki çalışmalara bakarak veri demetleme algoritmalarının performansını CUDA kullanıldığında maksimize etmek için dikkat edilmesi gerekenleri sıralarsak:

- ☞ Veriyi parçalayarak, boyutlarından bölerek hızlı, küçük belleklerden yararlanmak.
- ☞ Global bellek kullanılırken bütünleşik bellek okumaları yaparak gecikmeyi azaltmak.
- ☞ Thread ayrılmayı engellemek için dallanmaları azaltmak, gerekirse GPU'da sıralama yaparak, bir warp'taki tüm thread'lerin aynı komutu çalıştırmalarını sağlamak.
- ☞ Thread bloklarına eşit iş dağıtarak yük dengelemeyi sağlamak.
- ☞ Çok sık erişilecek küçük verileri saklayıcılarda tutmak.
- ☞ Thread başına kullanılacak saklayıcı sayısını dikkatli belirlemek.
- ☞ Sadece okuma amaçlı kullanılacak verileri sabit bellekte, yetmiyorsa doku bellekte tutarak bunların hızlı önbelleklenebilir mekanizmasından faydalanmak.

- ☞ Veri, boyut, demet sayısını sınırlandırmamak için parçalara bölerek işleme; boyut azaltma gibi teknikler kullanmak.
- ☞ Hesaplama sayısını azaltmayı sağlayan üçgen eşitsizliği gibi ilkelerden faydalanmak.
- ☞ Verileri GPU'da bütünleşik okumaya uyan sütun-tabanlı biçime dönüştürmek.
- ☞ CPU'nun verileri transfer edebileceği sistemlerin bant genişliği sınırlı olduğu için çok gecikmeli olan CPU'dan GPU'ya veri transferlerini en aza indirmek.
- ☞ Doluluğu sağlamak için kernellerde blok boyutunu 32'nin katı olarak seçmek.
- ☞ Paylaşılan belleğin kullanımında yığın çatışması oluşmamasını sağlamak.
- ☞ Yarış koşullarını (yazmadan-önce-okuma, okumadan-önce-yazma gibi) önlemek için thread senkronizasyonu yapmak ve çağrıları dikkatlice yerleştirmek.
- ☞ Kernellerin verimli çalışması için grid, blok ve thread sayısını dikkatli belirlemek.

Veri demetleme algoritmalarının GPU ile hızlandırılmış versiyonlarının geliştirilmesi son 5-6 yılda ağırlık kazanmıştır. Konuyla ilgili GPU versiyonu geliştirilmemiş demetleme algoritmalarına örnek CLARA-CLARANS, BIRCH, CURE, ROCK, CHAMELEON, OPTICS, DENCLUE, TURN, STING, CLIQUE verilebilir. Ayrıca DBSCAN, EM ve CAST GPU versiyonu mevcut fakat daha fazla geliştirmeye açıktır.

KAYNAKLAR

AMD "Close To Metal" Technology. (2012, Aralık). *AMD "Close To Metal" Technology*. http://www.amd.com/us/press-releases/Pages/Press_Release_114147.aspx adresinden alınmıştır

Anderson, D., Luke, R., & Keller, J. (2007). Analysis and Design of Intelligent Systems using Soft Computing Techniques. P. Melin, O. Castillo, E.G.Ramírez, J. Kacprzyk, & W.Pedrycz (Dü). içinde Springer Berlin Heidelberg.

Anderson, D., Luke, R., & Keller, J. (2008). Speedup of Fuzzy Clustering Through Stream Processing on Graphics Processing Units. *IEEE Transactions on Fuzzy Systems, 16*(4), 1101-1106.

DOI: 10.5824/1309-1581.2013.2.002.x

Bai, H., He, L., Ouyang, D., Li, Z., & Li, H. (2009). K-Means on commodity GPUs with CUDA. *2009 World Congress on Computer Science and Information Engineering (CSIE 2009)*, (s. 651-655). Los Angeles, California USA.

Beckmann, N., Kriegel, H., Schneider, R., & Seeger, B. (1990). The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, (s. 322-331). Atlantic City, New Jersey, USA.

Ben-Dor, A., Shamir, R., & Yakhini, Z. (1999, Ekim). Clustering Gene Expression Patterns. *Journal of Computational Biology, 6*(3/4), 281-297.

Bezdek, J. (1981). *Pattern Recognition with Fuzzy Objective Function Algoritms.* Kluwer Academic Publishers Norwell, MA, USA.

Böhm, C., Noll, R., Plant, C., & Wackersreuther, B. (2009). Density-based clustering using graphics processors. *CIKM '09: 18th ACM conference on Information and knowledge management*, (s. 661-670). Hong Kong, China.

Böhm, C., Noll, R., Plant, C., Wackersreuther, B., & Zherdin, A. (2009). Transactions on Large-Scale Data- and Knowledge-Centered Systems I. A. H. and J. Küng, & R. Wagner (Dü). içinde Springer Berlin Heidelberg.

BrookGPU. (2012, Aralık). *BrookGPU*. http://graphics.stanford.edu/projects/brookgpu/index.html adresinden alınmıştır

C++ AMP (C++ Accelerated Massive Parallelism). (2012, Aralık). *C++ AMP (C++ Accelerated Massive Parallelism)*. http://msdn.microsoft.com/en-us/library/vstudio/hh265137.aspx adresinden alınmıştır

Cao, F., Tung, A., & Zhou, A. (2006). Scalable Clustering Using Graphics Processors. *WAIM 2006: 7th International Conference on Advances in Web-Age Information Management*, (s. 372-384). Hong Kong, China.

Chang, D., Jones, N., Li, D., Ouyang, M., & Ragade, R. (2008). Compute pairwise euclidean distances of data points with GPUs. *Proceedings of the IASTED International Symposium on Computational Biology and Bioinformatics (CBB 2008).* Orlando, Florida, USA.

Chang, D., Kantardzic, M., & Ouyang, M. (2009). Hierarchical clustering with CUDA/GPU. *PDCCS'09: Proceedings of the ISCA 22nd International Conference on Parallel and Distributed Computing and Communication Systems*, (s. 7-12). Cambridge, Massachusetts, USA.

Che, S., Boyer, M., Meng, J., Sheaffer, D. T., & Skadron, K. (2008, Temmuz). A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing, 68*(10), 1370-1380.

Che, S., Meng, J., Sheaffer, J., & Skadron, K. (2007). A Performance Study of General Purpose Applications on Graphics Processors. *GPGPU'07: First workshop on general purpose processing on graphics processing units.* Northeastern University, Boston, MA.

Corporation, N. (2012). Nvidia Cg. *Nvidia Cg*. http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html adresinden alınmıştır

CUDA™ (Compute Unified Device Architecture) Zone. (2012, Aralık). *CUDA™ (Compute Unified Device Architecture) Zone*. https://developer.nvidia.com/category/zone/cuda-zone adresinden alınmıştır

Cui, X., Charles, J., & Potok, T. (2012, Eylül). The GPU Enhanced Parallel Computing for Large Scale Data Clustering. *Future Generation Computer Systems, In Press, Corrected Proof*.

Dunn, J. (1973). A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters. *Journal of Cybernetics, 3*, 32-57.

Espenshade, J., Pangborn, A., Laszewski, G., & Roberts, D. (2009). Accelerating Partitional Algorithms for Flow Cytometry on GPUs. *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*, (s. 226-233). Chengdu, Sichuan, China.

Ester, M., Kriegel, H., Sander, J., & Xu, X. (1996). A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. *2nd International Conference on Knowledge Discovery and Data Mining (KDD-96).*

Fang, W., Lau, K., Lu, M., Xiao, X., Lam, C., Yang, P., . . . Yang, K. (2008). *Parallel Data Mining on Graphics Processors.* Tech. rep., HKUST-CS08-07.

Farivar, R., Rebolledo, D., Chan, E., & Campbell, R. (2008). A Parallel Implementation of K-Means Clustering on GPUs. *PDPTA'08: International Conference on Parallel and Distributed Processing Techniques and Applications*, (s. 340-345). Las Vegas, Nevada.

Hall, J., & Hart, J. (2004). GPU Acceleration of Iterative Clustering. A. Lastra, M. Lin, & D. Manocha (Dü.), *2004 ACM Workshop on General Purpose Computing on Graphics Processors.* içinde Wilshire Grand Hotel, Los Angeles, California.

Harris, C., & Haines, K. (2005). Iterative Solutions using Programmable Graphics Processing Units. *FUZZ '05: The 14th IEEE International Conference on Fuzzy Systems*, (s. 12-18). Reno, NV.

Jian, L., Wang, C., Liu, Y., Liang, S., Yi, W., & Shi, Y. (2011, Ağustos). Parallel data mining techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA). *Journal of Supercomputing*, 1-26.

Karch, G. (2010). *GPU-based acceleration of selected clustering techniques.* Master's thesis, Silesian University of Technology, Faculty of Automatic Control, Electronics and Computer Science, Gliwice, Poland.

Kaufman, L., & Rousseeuw, P. (1987). *Clustering by Means of Medoids in Statistical Data Analysis Based on the L1–Norm and Related Methods.* (Y. Dodge, Dü.) Reports of the Faculty of Mathematics and Informatics. Delft University of Technology.

Kaufman, L., & Rousseeuw, P. (1990). Finding Groups in Data: An Introduction to Cluster Analysis. Hoboken, New Jersey: John Wiley \& Sons, Inc.

Kohlhoff, K., M.H.Sosnick, Hsu, W., Pande, V., & Altman, R. (2011, Haziran). CAMPAIGN: an open-source library of GPU-accelerated data clustering algorithms. *Bioinformatics, 27*(16), 2321-2322.

Kohlhoff, K., Pande, V., & Altman, R. (2012, Ağustos). K-means for parallel architectures using all-prefix-sum sorting and updating steps. *IEEE Transactions on Parallel and Distributed Systems, IEEE Early Access Articles*.

DOI: 10.5824/1309-1581.2013.2.002.x

Kumar, N., Satoor, S., & Buck, I. (2009). Fast Parallel Expectation Maximization for Gaussian Mixture Models on GPUs Using CUDA. *HPCC'09: 11th IEEE International Conference on High Performance Computing and Communications*, (s. 103-109). Seoul, Korea (South).

Li, Y., Zhao, K., Chu, X., & Liu, J. (2013, Mart). Speeding up k-Means algorithm by GPUs. *Journal of Computer and System Sciences, 79*(2), 216–229.

Lib Sh - Embedded Metaprogramming Language. (2012, Aralık). *Lib Sh - Embedded Metaprogramming Language*. http://libsh.org/ adresinden alınmıştır

Lin, K., & Lin, C. (2011). A fast CAST-based clustering algorithm for very large database. *Proceedings of 2011 International Conference on System Science and Engineering (ICSSE)*, (s. 420-424). Macau, China.

Liu, Z., & Ma, W. (2008). Exploiting Computing Power on Graphics Processing Unit. *CSSE'08: International Conference on Computer Science and Software Engineering*, *02*, s. 1062-1065. Wuhan, China.

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, (s. 281-297). the Statistical Laboratory, University of California, Berkeley, Calif.

Microsoft DirectX. (2012, Aralık). *Microsoft DirectX*. http://msdn.microsoft.com/en-gb/library/windows/apps/hh309467.aspx adresinden alınmıştır

Nvidia. (2012). GPU Hesaplama Nedir? *GPU Hesaplama Nedir?* http://www.nvidia.com.tr/object/gpu-computing-tr.html adresinden alınmıştır

NVIDIA. (2012, Kasım). NVIDIA CUDA Programming Guide | PG-02829-001_v5.0. *NVIDIA CUDA Programming Guide | PG-02829-001_v5.0*. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf adresinden alınmıştır

OpenCL™ (Open Computing Language) Zone. (2012, Aralık). *OpenCL™ (Open Computing Language) Zone*. http://developer.amd.com/resources/heterogeneous-computing/opencl-zone/ adresinden alınmıştır

OpenGL (Open Graphics Library). (2012, Aralık). *OpenGL (Open Graphics Library)*. http://www.opengl.org/ adresinden alınmıştır

Pangborn, A. (2010). *Scalable Data Clustering for Flow Cytometry using GPUs.* Master's thesis, The Kate Gleason College of Engineering at Rochester Institute of Technology, Rochester, New York.

Shalom, S., & Dash, M. (2011, Ekim). Efficient Hierarchical Agglomerative Clustering Algorithms on GPU Using Data Partitioning. *12th International Conference on Parallel and Distributed Computing, Applications and Technologies*, (s. 134-139).

Shalom, S., Dash, M., & Tue, M. (2008). Efficient K- Means Clustering Using Accelerated Graphics Processors. *DaWaK'08: 10th International Conference on Data Warehousing and Knowledge Discovery*, (s. 166-175). Turin, Italy.

Shalom, S., Dash, M., & Tue, M. (2008). Graphics Hardware based Efficient and Scalable Fuzzy C-Means Clustering. *AusDM 2008: The Australasian Data Mining Conference*, (s. 179-186). Glenelg, South Australia.

Shalom, S., Dash, M., Tue, M., & Wilson, N. (2009). Hierarchical Agglomerative Clustering Using Graphics Processor with Compute Unified Device Architecture. *2009 International Conference on Signal Processing Systems (ICSPS 2009)*, (s. 556-561). Singapore.

Takizawa, H., & Kobayashi, H. (2006). Hierarchical parallel processing of large scale data clustering on a PC cluster with GPU co-processing. *Journal of Supercomputing, 36*(3), 219-234.

Temizel, A. (2011). CUDA ve OpenCL Temelleri. *Hesaplamalı Bilimlerde GPU Teknolojileri ve Genel Amaçlı GPU Programlama Semineri*, (s. 59). ODTÜ Enformatik Enstitüsü Ural Akbulut Amfisi, Ankara.

Thapa, R., Trefftz, C., & Wolffe, G. (2010). Memory-efficient implementation of a graphics processor-based cluster detection algorithm for large spatial databases. *2010 IEEE International Conference on Electro/Information Technology (EIT).* Normal, IL, USA.

The Compute Shader Technology (DirectCompute). (2012, Aralık). *The Compute Shader Technology (DirectCompute)*. http://msdn.microsoft.com/en-us/library/ff476331.aspx adresinden alınmıştır

Vaitheeshwaran, V., Nagwanshi, K., & Rao, T. (2012, Mart). Multicore Processing for Classification and Clustering Algorithms. *IJCA Proceedings on National Conference on Innovative Paradigms in Engineering and Technology (NCIPET 2012), ncipet*, s. 20-24.

Voorhees, E. (1986). Implementing agglomerative hierarchical clustering algorithms for use in document retrieval. *Information Processing and Management, 22*(6), 465-476.

Wilson, J., Dai, M., Jakupovic, E., & Meng, F. (2007). Supercomputing with toys: Harnessing the power of NVDIA 8800GTX and Playstation 3 for bioinformatics problems. *CSB 2007: Computational Systems Bioinformatics Conference*, (s. 387-390). University of California, San Diego, USA.

Wu, J., & Hong, B. (2011). An Efficient k-Means Algorithm on CUDA. *IEEE International Parallel \& Distributed Processing Symposium* (s. 1740-1749). IEEE.

Wu, R., Zhang, B., & Hsu, M. (2009). Clustering Billions of Data Points Using GPUs. *UCHPC-MAW'09: Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, (s. 1-6). Ischia, Italy.

Wu, R., Zhang, B., & Hsu, M. (2009, Mart). *GPU-Accelerated Large Scale Analytics.* Tech. rep., HP Laboratories Technical Report.

Zechner, M., & Granitzer, M. (2009). Accelerating K-Means on the Graphics Processor via CUDA. *INTENSIVE 2009: The First International Conference on Intensive Applications and Services*, (s. 7-15). Valencia, Spain.

Zhang, Q., & Zhang, Y. (2006, Nisan). Hierarchical clustering of gene expression profiles with graphics hardware acceleration. *Pattern Recognition Letters, 27*(6), 676–681.

Zhang, Y., Mueller, F., Cui, X., & Potok, T. (2011, Şubat). Data-intensive document clustering on graphics processing unit (GPU) clusters. *Journal of Parallel and Distributed Computing, 71*(2), 211-224.

Zhao, Y., Sheong, F., Sun, J., Sander, P., & Huang, X. (2013, Ocak). A Fast Parallel Clustering Algorithm for Molecular Simulation Trajectories. *Journal of Computational Chemistry, 34*(2), 95-104.