Dağınık Çok Çekirdekli CPU ve Çoklu GPU Sistemleri İçin Heterojen Programlama Kütüphanesi

Program Kodu: 1001

Proje No: 112E191

Proje Yürütücüsü:

Doç. Dr. Zeki Bozkuş

KASIM 2015
İSTANBUL

Önsöz

Bu projemiz, TÜBİTAK'ın desteği ile 1 Mart 2013 tarihinde başlayan "Dağınık Çok Çekirdekli Cpu Ve Çoklu Gpu Sistemleri İçin Heterojen Programlama Kütüphanesi" başlıklı ve "112E191" numaralı bilimsel araştırma projesi olup, 01 Kasım 2015 tarihinde sonuçlandırılmıştır. Projede heterojen platformlara program yazmayı kolaylaştıran yeni bir programlama modeli geliştirilmiştir. Bu programlama modeli, hem tekli CPU-GPU sistemlerinde, hem de ortak bellekli çoklu CPU-GPU sistemlerinde ve hem de dağınık bellekli heterojen (çoklu CPU ve çoklu GPU içeren) ortamlar için, kullanım kolaylığı olan, taşınabilir ve verimlilikten olabildiğince az ödün veren, yeni bir programlama modelidir.

Bu proje TÜBİTAK 1001 tarafından desteklenmiştir.

TÜBİTAK

İçindekiler

1 Giriş **1**

2 Literatür Özeti **3**

3 Genel Bilgiler **5**

3.1 Donanım Mimarisi *5*

3.2 HPL Programlama Modeli *6*

3.3 Modelin Tek GPU Ara Yüzü *8*

3.4 Kütüphanenin Tek GPU Gerçekleştirilmesi *9*

4 Gereç ve Yöntem **11**

4.1 DistHPL Programlama Modeli *11*

4.2 DistHPL Yazılım Mimarisi *12*

4.1 DistHPL Matrix Multiplication Örneği *12*

5 Bulgular **16**

6 Genetik Algoritma Tabanlı Moleküler Docking Algoritmalına Çoklu GPU Desteği Kazandırma **19**

6.1 Moleküler Docking Performans Sonuçları *21*

7 Global Birebir Ağ Hizalama Problemine HPL Tabanlı Çözüm **23**

7.1 SPINAL Algoritması *24*

7.2 G-SPINAL: SPINAL Algoritmasının GPU Versiyonu *27*

7.3 Karşılaştırmalı Deneysel Sonuçlar *29*

8 Sonuçlar **31**

9 Kaynakça **32**

10 Ekler **34**

10.1 EK-1 Projeden Üretilen ve Proje Numarasına Atıf Verilerek Yapılan Yayınlar *34*

11 Ekler **35**

11.1 EK-1 Projeden Üretilen ve Fakat Yayınlanmayan Çalışmalar: *35*

12 Ekler **36**

12.1 İş Paketlerin Tamamlanma Oranları *36*

TÜBİTAK

Şekil Listesi

Şekil 1: Heterojen(CPU/GPU) yapı 5

Şekil 2: Dağınık Bellek bilgisayar mimarisi 5

Şekil 3: Global ve local thread domain leri 7

Şekil 4: *Kütüphane yazılım mimarisi* 10

Şekil 5: *Kernel çağırma için kullanılan iş akış diyagramı* 10

Şekil 6: distHPL programlama modeli 11

Şekil 7: distHPL yazilim mimarisi 12

Şekil 8: distHPL yazılmıs MM algoritması 14

Şekil 9: Değişen büyüklükte matrix multiplication. MPI ve distHPL ayni makinada çalışmaktadır. MPI tüm 8 CPU kullanmakta, distHPl ise 2 GPU kullanmaktadır. 16

Şekil 10: MM (N=3000) seri MM nin hızı MPI ve distHPL farlı konfigürasyonlarda karşılaştırılması... 17

Şekil 11: NAS EP testinin 2 GPU çalışan distHPL yazılmışının seri tek CPU çalışan C++ yazılmışının karşılaştırılması. 17

Şekil 12: Matrix Multiplication(N=5000) on Pluto platform. (4CPU, 4GPU) 18

Şekil 13: G-SPINAL'de thread tasarımı 28

Tablo Listesi

Tablo 1: Molekuler Docking Hızlanma - Tek GPU 20

Tablo 2: Molekuler Docking Hızlanma - Çoklu GPU 20

Tablo 3: Spinal, Arrays ve G-Spinal zaman ve sonuç karşılaştırması 30

Özet

Son yıllarda, yüksek hesaplama performansına ihtiyaç duyan uygulamaların en çok tercih ettiği bilgisayar mimarisi, çok çekirdekli CPU'lara eklenmiş çoklu GPUlardan oluşan heterojen sistemlerdir. Fakat bu tür sistemlerin programlanması alışagelmiş olduğumuz tek işlemci ve hatta çok işlemci programlamasından çok daha karmaşıktır.

Bu projede, dağınık heterojen sistemler için, programcının verimliliğini artıran ve taşınabilme özelliği olan bir paralel yazılım kütüphanesi geliştirilmiştir. Proje sıradan bir kütüphaneden çok, C++ dilinin içinde yer alan küçük, yeni bir programlama dilidir. Öyle ki programcı yazdığı herhangi bir C++ programı içinde bu küçük dilin çekirdek fonksiyon ve veri tiplerini de kullanıp donanımda yer alan bütün paralel işlem cihazlarından (CPU/GPU) faydalanılarak paralel programları kolaylıkla yazabilmektedir.

Her karmaşık yazılımda olduğu gibi Heterogeneous Programming Library (HPL) çeşitli katmanlardan oluşmaktadır. Ilk katmanı tekli CPU-GPU ortamında çalışmaktadır. İkinci katmandaki HPL ortak bellekli tek-CPU bağlı, çoklu GPU sistemlerini kullanma katmanıdır. Son olarak da dağınık bellekli çoklu CPU-GPU sistemlerini kullanana distHPL katmanıdır. İlk iki katman için dergi yayını yapmış bulunmaktayız. Son adim için ise teknik raporumuzu hazırladık, yayın yapmaya çalışıyoruz. Geliştirdiğimiz HPL kütüphanesi taşınırlık, kolay programlama ve performans metriklerinde başarılı sonuçlar elde edildi. Örneğin OpenCL ile karşılaştırıldığında, HPL ile yazılan uygulamalarda %70-%90 oranlarda yazım kolaylığı gözlemledik. Son aşamada, iki biyoinformatik algoritmasını, geliştirdiğimiz programlama modeliyle yazarak, yüksek hesaplamalı heterojen platformlarda çalıştırdık.

Anahtar Kelimeler:
Heterogenic Programlama, Parallel Programlama Dilleri, GPU Programlama

Abstract

The distributed heterogeneous system, which consists of shared memory nodes with several multi-core CPUs attached with multi-GPU accelerators and connected to a high speed network to comprise a distributed memory system, has become the best performing modern hardware architecture for the high-performance computing community in the past few years. Unfortunately, however, distributed heterogeneous systems require much more effort to be programmed than traditional single or even multi-core computers that most programmers are familiar with.

In this project, we developed a novel library-based approach for programming of distributed heterogeneous systems that focuses on delivering high programmer productivity and portability. The project rather than an ordinary library, located in the small C ++ language is a new programming language. So that any programmer wrote a program in C ++, with the help of our language of small kernel functions and data types where these kernel can run at all parallel processing device (CPU / GPU). Our library makes it easy to write parallel programs on these heterogeneous systems.

Like every complex software system, Heterogeneous Programming Library (HPL) is made up of several layers. The first layer of HPL runs at single-CPU-GPU the environment. The second layer of HPL runs on shared memory in a single-CPU connected to multi-GPU systems. Final layer of HPL run on the distributed memory heterogeneous CPU-GPU systems. We have done two journal publications of the first two layers. For the last layer, we prepared a technical report. HPL has proven to be very successful for portability, programmability (ease of use) and performance metrics. For example, our publications present that HPL performs 70%-90% betters in terms of programmability metric than OpenCL. Finally, we developed two bioinformatics algorithms with our programming model to heterogeneous platforms.

Keywords:

Heterogeneous Programming, Parallel Programming Languages, GPU Programming.

1 Giriş

Bu projede çok çekirdekli CPU ve çoklu GPU dağınık heterojen sistemlerde hesapsal paralelliği sağlayacak bir yazılım kütüphanesi geliştirdik. Bu amaca yönelik literatürde tanımlanmış sistemlerin önemli eksiklikleri ve yetersizlikleri söz konusudur. Bazıları taşınabilirlikten mustaripken (CUDA tabanlı sistemler), bazıları aşağı seviye doğalarından ötürü kod karmaşıklığına yol açarlar (OpenCL tabanlı sistemler).

Yüksek hesap potansiyeli sunan ölçeklenebilir sistemler dağınık bellekli mimari ile oluşturulurlar. Var olan sistemleri dağınık bellekli modele uyarlamak, bütün alt sistemlerin gerektirdiği programlama modellerinin (MPI+OpenMP/thread+CUDA/OpenCL) uygun bir şekilde birleştirilmesini gerektirir. Bu projede geliştirdiğimiz sistem, dağınık bellekli mimarideki her alt sistem kaynaklarını kullanan, alt sistemler arasındaki iletişimi sağlayan ve kullanıcıya (programcı) olabildiğince alt seviye detaylardan soyutlanmış deyim ve komutlar sunan bütüncül bir programlama modelidir.

Bu modeli geliştirirken bunun aşama aşama yapılacağını öngördük. İlk yaptığımız aşama tekli CPU-GPU ortamında çalişması gerekiyordu. Bunun kullanarak, sistemin ortak bellekli CPU tarafında kontrol edilen çoklu GPU sistemlerinde çalıştırdık. Ve yine bunu kullanarak sistemin son platformu olarak dağınık sistemlerdeki çoklu CPU-GPU kümelerini tamamladık.

HPL sisteminde ilk yaptığımız iş, kernel yazmayı kolaylaştıran komutlar koymak oldu. Bu komutlar C++ makro ve overloading özelliğini kullanarak geliştirildi. Sonra sistemde kullanılan bir Array data tipi geliştirildi. HPL bu arrayleri CPU belleği ve GPU belleği arasında otomatik olarak, ihtiyaç duyuldukça yaptık.

İkinci adımda, HPL kütüphanesini ortak bellekli çoklu GPU larda çaliştirdik. Bunun için HPL Array'ını farkı GPU'lara kopyalama ve orada kernel çalıştırma yeteneği ekledik. Bunun için sistem sadece CPU belleği kontrol etmedi, sistemde var olan çoklu GPU belleklerini kontrol etti. Bunlar için tembel kopyalama (lazzy copy) teknik geliştirdi.

Son olarak sistemin heterojen dağınık sistemlerde çalışması için yukarıdaki özelliklerinin yanında haberleşme özelliğini eklemek zorunda kaldık. Yani özel distributed HPL arraylar kullanan kütüphane,

bu arrayleri blok, blok sistemdeki dağınık CPU hostlarina dağıtma özelliği ekledik. Ve bu dağıtılan Distributed Arrayleri ihtiyaca göre haberleşme yeteneği ekledik. Distribued Arrayler lokal kısımları "owner computes rule" mantığı kullanarak, lokal HPL array yerleştirip buradan GPU kernel çalıştırdık.

Bu yaptığımız kütüphanenin basarisini incelemek için çeşitli benchmark testleri çeliştirdik. Bunların hızlarını ve program satrilarini, taşınabildiklerini test ettik. Bu testlerin en başsında NAS paralel benckmarklari kullandık. Ayrıca kütüphane kullanarak iki farklı bioinformatik uygulaması geliştirdik.

Bu sonuç raporunda geri kalan kısımlarını şu şekilde düzenledik. Bölüm 2 literatür özeti vereceğiz. Bölüm 3 genel bilgiler verilecektir. Bölüm 4 Gereç ve yöntemler hakkında bilgi vereceğiz. Bölüm 5 bulgular ve performans sonuçları anlatacağız. Bolum 6 ise kütüphaneyle geliştirilmiş bir bioinformatik uygulaması olan moleküler docking sunulacak. Bolum 7 is yine proje kütüphanesiyle geliştirilmiş başka bir bioinformatik çalışmaları verilecektir. Bolum 8 iş sonuç kısmı verilecek.

2 Literatür Özeti

GPU sistemlerinin (GPGPU 2011) en yaygın programlama yöntemi C, bazen C++ dillerinden türetilen ve gerekli kütüphanelerle zenginleştirilmiş programlama modellerine dayanır. Bunlara Cell BE hızlandırıcı (IBM 2006.) için C/C++ uzantıları (IBM 2006) ve GPU için ise Brook+ (AMD 2008) güzel örneklerdir. Fakat bugüne kadar en başarılı olan hızlandırıcı dili CUDA (Nvidia 2008) olmuştur. Tüm bunlar, düşük seviyede soyutlama yapan ve çoğunlukla bir tek aygıtta çalışan, taşınabilirlik özelliği olmayan programlama dilleridir. Fakat son yıllarda geliştirilen açık kaynak OpenCL (Nvidia 2008) dili, hem CPU ve hem de her türlü GPUlar için paralel programlama geliştirme amaçlı tasarlanmış bir programlama dilidir. Bu yüzden biz de proje konusu heterojen paralel programlama kütüphanesinin ön çalışması niteliğindeki ilk sürümün arka planında OpenCL kullandık. OpenCL de olsa CUDA da olsa bu dillerin hepsi programcının işini zorlaştıran, çok düşük seviyeli ve donanıma yakın dillerdir. Proje çerçevesinde geliştirilmesi planlanan kütüphane ise paralel program yazmak için yüksek seviyede soyutlanmış deyim ve komutlarıyla dağınık heterojen hesaplama modeline sahip sistemlerdeki programcının üretkenliğini artıran bir dil olarak tasarlanmaktadır.

Tasarladığımız çerçeveye yakın bir yazılım olan Thrust (Hoberock 2011) CUDAyı arka planda kullanan ve bazı algoritmaların paralel yazılmasını sağlayan sistemdir. Bu yazılım da projemizde planlandığı gibi programcının üretkenliğini artırmayı hedeflemesine rağmen önerdiğimiz model kadar genel bir yazılım içermez. Örneğin, sadece tek boyutlu dizilimlere izin vererek programcıyı sınırlar. Ayrıca Thrust'ın çekirdek (kernel) yazılımı çok kısıtlı olduğundan ve programcının farklı bellek seçeneklerini kullanmasına izin vermediğinden üretilen kodun performansı düşük olmaktadır. Thrust ile ilgili bir diğer olumsuzluk ise arka planda CUDA kullanıldığından sadece CUDA'nın geliştiricisi NVIDIA GPU larında çalışır ve taşınırlık özelliği yoktur. Konuyla ilgili belirtilebilecek diğer kütüphaneler arasında PyCUDA ve PyOpenCL (A. Klockner 2009) gelir. Bu kütüphaneler Python dili ile GPU dilleri arasında aracı görevi yaparak, önceden tanımlanan hesaplamaları gerçekleştirmek için kullanılır. Bunlarda da çekirdek, CUDA ile yazılmış dizilimi (string) girdi olarak alma mantığına dayanır. Projede önerilen modelde ise çekirdek kendi anahtar kelimeleri ve makrolar kullandırılarak yazdırılır. Bu, tasarlanan kütüphanenin çekirdek fonksiyon argümanlarının nasıl kullanıldığını analiz edip CPU ve GPU arası gereksiz veri kopyalamasından kurtulmasını sağlar. Intel Array Building Blocks (ABB) (C. J. Newburn 2011) altyapısı ise projemizde önerilen kütüphane gibi kendi veri dizilimleri ve makrolarını kullanarak C++ içinde geliştirilmiş bir kütüphanedir. Buradaki amaç CPUlardaki çoklu çekirdekleri kullanmaktır. Projede geliştirilecek kütüphane ise GPUları kullanabildiği gibi, görev boyunu ayarlama ve farklı bellekleri kullanma özellikleri ile ABB'den farklıdır. Ayrıca ABB donanım ağırlıklı bir programlama modeli olarak

karşımıza çıkar (A. L. Varbanescu 2011). Heterojen sistemleri programlamanın diğer bir alternatifi de, OpenMP'de olduğu gibi derleyici direktifleri kullanarak, derleyiciyi bu sistemler için uygun kod üretmeye yönlendirmektir (Eigenmann). Fakat derleyici direktiflerinin sınırlayıcı özelliği iyi bilinen bir konudur. Bu şekilde çalışan sistemlerin kullanıcıya gerektiği kadar detaylı bir şekilde problemi tanımlama fırsatı vermediğinden, kullanıcı derleyiciden gelen ve performansı hiç de iyi olmayan bir kodu kabullenmek zorunda bırakılır.

Bahsi geçen programlama kütüphanelerinin hepsi dağınık olmayan heterojen sistemler için geliştirilmiş yazılımlardır. Dağınık bellekli heterojen sistemleri destekleyen çalışmalar ise çok sınırlı sayıda olup ancak son yıllarda konuyla ilgili çalışmalar yapılmaya başlanmıştır. Bu sistemlerden biri olan SnuCL (J. Kim 2011) yazılımı dağınık sistemlerde MPIyi kullanarak OpenCL APIsini geliştirmiştir. Sistemin taşınırlık özelliği yoktur ve sınırlı sayıda aygıtlarda kullanılır. SnuCL ile ilgili bir diğer olumsuzluk da heterojen sistemde yer alan her alt sistemin tek bir GPU gibi modellenmesi mantığına dayalı olmasıdır. Bu da dolaylı olarak farklı alt sistemlerin özelliklerinin verimli kullanımına engeldir ve sistemin performansını olumsuz etkiler. CLuMPI ise yine OpenCL API sini MPI kullanarak geliştirmeye bir sistemdir. Bu sistem GPU kullanmaz, sadece dağınık CPUları GPU gibi kullanmayı hedefler. Yani kullanıcı sistemin bir dağınık bellekli sistem olduğundan bile habersiz sadece OpenCL kodunu yazar. CluMPI da bunu dağınık sistemlerin CPUlarında çalıştırır. SnuCL için bahsedilen olumsuzluk burada da kendini göstermekte ve hatta sistemde var olan GPUları bile kullanmadığından performans oldukça kötüdür. Dağınık heterojen sistemlerin programlaması için sadece OpenCL tabanlı programlama modelleri önerilmemiş, bunun yanında CUDA tabanlı modeller de çalışılmıştır. CUDASA (M. Strengert) çoklu GPUları kullanmak için CUDA uzantısını geliştirmiştir. Bu sistemde de GPUlar arasında gerekli veri transferi için MPI kullanılır. rCUDA [(J. Duato 2010)18] da CUDA APIsini dağınık heterojen sistemde geliştirmeyi hedefler. CUDA sadece NVIDIA GPUlarında çalıştığı için taşınırlık özelliği yoktur. OpenCL APIsi geliştirme tabanlı sistemler için belirtilen performans sorunları, tüm sistem yine tek bir GPU şeklinde modellendiğinden burada da aynen geçerlidir.
Global Array ise dağınık sistemlerde global address sunmakda, kullanımı MPI daha kolay fakat heterojen sistemler uzerinde calişmamaktadir (Nieplocha, Harrison, and Littlefield 1994).

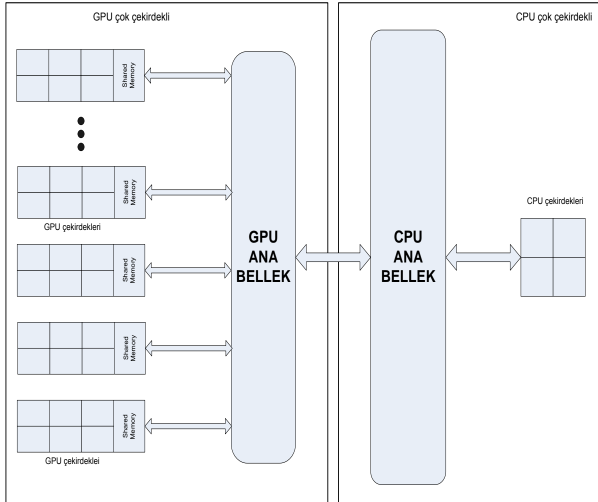
Bu çalişmada (Altilar 2012) OpenCL kernellarini heterojen sistemlerde remote procedure call ile uzakdaki host yollayabiliyor, fakat burada yine OpenCL kodu yazman bekleniyor. Bizimkinde ise OpenCL çok daha kolay bir programlama modeli sunuyor.

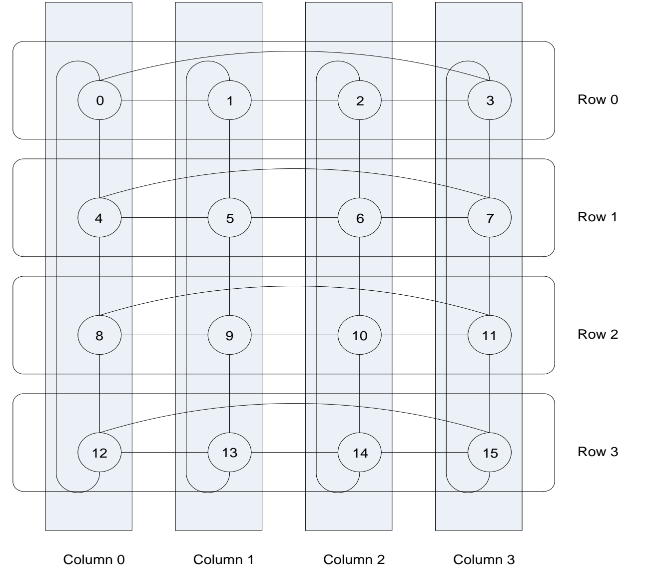
3 Genel Bilgiler

Proje kapsamında, dağınık heterojen sistemlerde kullanılmak üzere C++ tabanlı yeni bir programlama modeli geliştirdik. Bu programlama modelinin bütüncül bir şekilde heterojen sistemin altında bulunan tüm alt sistemleri en verimli bir şekilde kullanmasını ve bu sistemlerin programlanmasını kolaylaştırmasını hedefledik. Bu programlama modelini, C++ ın expression template, operator overloading, makro ve nesneye dayalı programlama yönünü ile Just-in-Time (JIT) compilation teknolojisini birleştirerek, bir yazılım kütüphanesinden çok bir derleyici ve bir programlama dili olarak geliştirdik.

3.1 Donanım Mimarisi

Şekil 1 ortak bellekli çok çekirdekli CPU ve GPU dan oluşan, heterojen yapıda olan donanım mimarisinin genel bir bakışı verilmektedir. Günümüzde bir çok masa üstü ve diz üstü sistemler bu mimariyi kullanmaktadır. Bu mimaride CPU ve GPU nun kendi ana bellekleri bulunmaktadır. Programcı bu iki belleği yapacağı işe göre kullanmak zorundadır. GPU sadece kendi belleğini görür. Aynı şekilde CPU da sadece kendi belleğini görür. GPU nun yüzlerce çekirdeği vardır. CPU nun ise onlarca çekirdeği vardır. Programcının görevi, verileri bellekler arasında taşımak ve çekirdeklere iş vermek; diğer bir ifadeyle, küçük bir kaç satırlık tek çekirdekli sistem için yazılmış bir programı, bir kaç sayfalık GPU programına dönüştürmektedir.





Şekil 1: Heterojen(CPU/GPU) yapı

Şekil 2: Dağınık Bellek bilgisayar mimarisi

Şekil 2 ise örnek olarak, küçük sayıda (16 tane) boğumların birbirlerine mesh ağlarıyla bağlanmış (Toros Cartesian topology) dağınık bellekli bilgisayar mimarisini göstermektedir. Boğumlarında heterojen (CPU/GPU) yapıya sahip oluşan bu sistemi, dağınık bellekli heterojen sistemi olarak adlandırılmaktayız. Biz de geliştirmek istediğimiz programlama modelini bu tür bir mimari platformunda çalıştırmayı hedeflemekteyiz. Dağınık bellekteki mimarilerin alternatiflerine göre ölçeklenmesi daha iyidir. Bu tür makinalarda bellek dağılmış olduğundan, bellek darboğazı oluşmaz. Alternatifi olan tek ortak bellekli sistemlerde ise, tüm işlemciler aynı belleğe ulaşmak istediklerinde, bellek darboğazı oluşur. Bu nedenden dolayı, yüksek performanslı süper bilgisayarlar, dağınık bellek mimarisini tercih ederler.

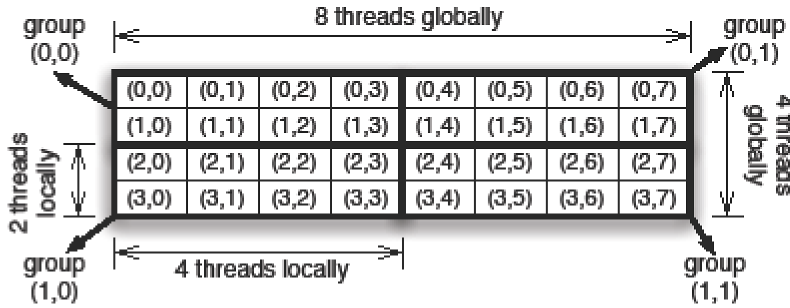
Dağınık bellek yapısına sahip makinalarda, programlama modeli olarak Message Passing Interface (MPI) kullanılır. Bu yazılım bir kütüphanedir ve boğumlar arasında veri transferi yapar, fakat sonuç olarak çok karmaşık bir program ortaya çıkar. Bu programlama modelinde hataları bulmak çok zordur. Bir CPU'nun veriyi yollaması, diğer CPU nun bu veriyi karşılaması mantığına dayanır. Bunların olmaması durumunda ise tüm sistem kilitlenebilir. MPI kullanmayan programcılar ise daha düşük seviyede Linux socket programlama yöntemini kullanarak verileri birbirlerine yollama şansına sahiptirler. Dağınık bellek makinaların her bir boğumunda heterojen (CPU/GPU) bir yapı olduğu zaman, bu programlamayı daha da zor bir hale getirir. Bu donanımda var olan heterojen yapı, yazılımını da da heterojen bir yapıya dönüşmeyi gerektirir. Bu sistemlerde programlamada üç farklı programlama modelinin birleşmesinden oluşan karma bir yöntem kullanılır. Boğumda bulunan çok çekirdekli CPU alt sistemi için OpenMP yada thread modellerinden biri; boğumda bulunan çoklu GPU lar için OpenCL yada CUDA modellerinden biri; dağınık bellek alt sistemi için ise MPI yada Linux socket programlama modellerinden biri kullanılmaktadır.

3.2 HPL Programlama Modeli

Bizim programlama modelini kullanan programcı, üç farklı programlama modelini kullanarak program yazmak zorunda kalmayacak, sadece projenin önerdiği programlama modelini kullanarak program yazılacaktır. Bizim metodumuz, dağınık bellekli heterojen sistemlerin programlama zorluklarından ve detaylarından programcıyı kurtarmaktadır. Geliştirdimiz model, programcının çözmeye çalıştığı problemi küçük parçalara ayrılması, bunların farklı işlemcilere gönderilmesi, eşzamanlı çalışmalarının sağlanması, işlemcilerin birbirleriyle koordinasyonlarının yapılması ve sonuçların birleştirmesi gibi paralel programlamanın detaylarının hepsini üstlenecektir.

Modelde, çekirdek (kernel) fonksiyonları adını verdiğimiz fonksiyonlar, binlerce thread in paralel olarak aynı anda çalıştırılması mantığına dayalıdır. Bu modelde, kernel fonksiyonları seçilen herhangi bir cihazda (CPU yada GPU) binlerce thread kullanarak, her bir thread farklı bir data üzerinde, aynı programı (kernel fonksiyonu) çalıştırarak, veri paralelizm (data paralelizm) yapmaya olanak sağlıyoruz. Bunu yapmak için, ilk olarak modelimiz kullanıcıya binlerce thread sahibi olmayı ve bunlara paralel kernel fonksiyonunu çağırma fırsatı veriyor.

Modelimizde threadler kullanıcının çözmeye çalıştığı problemin içeriğine göre çok basit bir ara yüz sayesinde anında hazırlanır. Örnek vermek istersek *eval(f).global(4, 8).local(2, 4)(a).* Buradaki global ve local komutları, şekil 3 te gösterildiği gibi iki boyutlu bir global domain ve local domainleri oluşturur. Bu threadlerin her biri paralel olarak *f* kernel fonksiyonunu çağırırlar. Modelin sağladığı evaluation metodu *eval* bu işlemleri tetikler yaptırır.



Şekil 3: Global ve local thread domain leri

Kernel fonksiyonlarını çalıştıran her bir thread i belirleyen bir kimlik numarası vardır. Bu thread numaraları sayesinde, işler belli threadlere dağıtılabilir. Threadleri üç boyutlu olabilen adına global domain dediğimiz, tam sayılardan oluşan küme belirler. Global domain deki her bir tam sayı, threadi belirleyen kimlik numarasıdır. Global domain nin büyüklüğü parallel çalışacak olan toplam thread leri gösterir. Seçmeli olarak kullanıcı local domain dediğimiz alt grupları da oluşturma olanağı vardır. Aynı local domainde bulunan threadlerin, ana bellekten çok hızlı, fakat daha küçük bir ortak bellek kullanma şansı vardır. Şekil 3 de gösterilen global domain iki boyutlu olup, toplam 4x8 thread vardır. Thread kimlikleride parantez için de (x, y) gösterilmiş, x boyutunda thread numarası, y boyutunda thread numarası olarak. Ayrıca kalın çizgi ilede local domain gösterilmiştir.

3.3 Modelin Tek GPU Ara Yüzü

Projemizde geliştirdiğimiz C++ kütüphanesi, programcıya verdiği deyim ve komutlarla herhangi bir problemi kernel fonksiyonu olarak yazma olanağı veriyor. Ve yazılan bu kernel fonksiyonu sistemdeki hesaplama yeteneği olan, seçilen herhangi bir cihazda çalıştırılabilir. Kernel sıradan C++ komutlarıyla yazılamaz, nedeni ise C++ derleyicisi kernel fonksiyonunu CPU için derler, sistemdeki herhangi bir cihaz için çalıştırmaz. Kernel sadece geliştirilen modelin deyim ve komutları ile yazılacaktır. Bu komutlar standard C++ veri tipleri (data types), expression template, fonksiyonlar ve makrolar kullanarak dizayn edilmiştir. Bunlar sayesinde kernel kodu yürütüm (run-time) süresinde dinamik olarak istenilen cihaza göre derlenir. Ayrıca kernel içerisinde sadece bizim geliştirdiğimiz veri tipleri (data types) kullanılır. Bu veri tipleri sayesinde, geliştirilen model, veri dizilerini paketleyip, büyüklüklerini belirleyip, hangi bellekte yerleştirileceğine dair bilgiler ile birlikte sarmalayan ve bunu heterojen cihazlara taşıyabilme özelliği olan yapıları programcısına sunar.

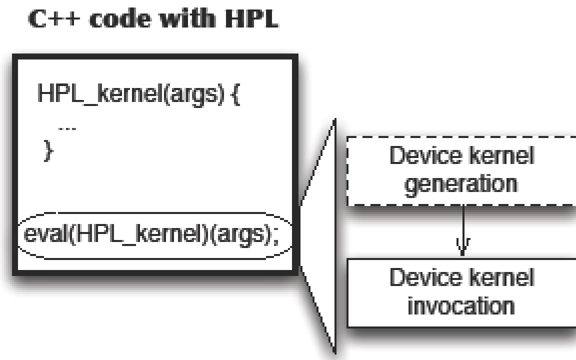
Veri Tipleri: Kernell fonksiyonlarında kullanılması için, model kendi özel veri tipini tanımlamıştır. **Array**< *type, ndim, [memoryFlag]>.* Bu veri tipi C++ template özelliği kullanılarak hazırlanmıştır. Buradaki *type* C++ standard tiplerinden herhangi biri olabilir ve Array elemanının veri tipini gösterir. Seçmeli olan üçüncü değişken (*memoryFlag*) ise Arrayin kayıt edileceği cihazdaki bellek çeşidini belirtmek için kullanılır. Eğer bu seçenek yoksa, Array cihazın ana belleğine yerleştirilir. Örneğin *Local* flag kullanılırsa, Array cihazin daha hızlı cevap veren bölgesel belleğine yerleştirilir. Array lerin büyüklüğü belirtilerek, otomatik olarak bu arrayler gerekli yerlere yerleştirilir. Skalar değişkenlerini göstermek için ise baş harfi büyük olan (Int, Uint Double) C++ veri tiplerini kullanır. Kernel içindeki Array elemanlarına kare parantez kullar anarak erişim sağlanır ve bu erişim paralel dir. Fakat host cpu da ise yuvarlak parantez kullanarak ulaşılabilir. Bunun nedeni kernel kodun runtime dynamic olarak derlenip optimize edilmesi, host kodun ise statik olarak compile time derlenmesidir.

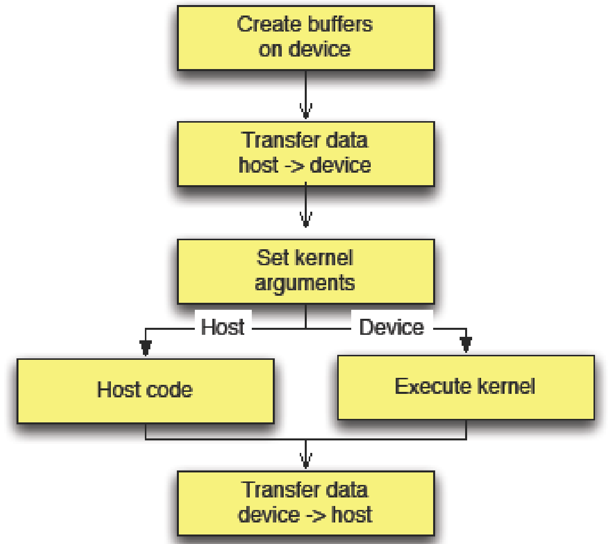
Kernel Sözdizilimi (Syntax): Daha önceden söylediğimiz gibi, kernel fonksiyonları geliştirilen modelin deyim ve komutlarıyla yazılması gerekmektedir. Kütüphane bu komutları yakalayıp, kernel fonksiyonu için gerekli kodu, istenilen cihaza göre üretmesi gerekecektir. Model sunduğu komutlarla standart C++ benzer bir şekilde program yazacaktır. Örneğin C++ kontrol komutları *(if, for, ...)* iki şekilde değiştirerek kernel fonksiyonunda kullanılacaktır. İlk olarak anahtar kelimesinin sonuna underscore (_) eklenerek *(if_, for_, ...)* kendini C++ tan ayırt edecektir. İkinci olarak kontrol komutlarının bittiğini *(endif_, endfor, ...)* gibi komutlarla gösterecektir. Bu yeni komutları kullanarak kernel fonksiyonları, C++ fonksiyonlarına benzer bir şekilde yazılabilir.

Buna ek olarak, modelimiz kullanıcıya önceden tanımladığı özel değişkenleri kernel fonksiyonunda kullanıma sunacaktır. Örneğin*, idx, idy, idz* değişkenleri, çalışan kernel de birinci, ikinci ve üçüncü boyutta threadin kimliğinin belirlenmesi (thread id) için kullanılır. Benzer olarak *lidx, lidy, lidz* değişkenleri de, kernelde çalışan threadin local domainde yerlerinin belirlenmesi için kullanılır. Kernel içinde kullanılan barrier fonksiyonu, local group da olan kernel lerin ayni yerde buluşmasını sağlayan bir komuttur.

3.4 Kütüphanenin Tek GPU Gerçekleştirilmesi

Modelimizin sadece tekCPU/tekGPU çalışan kısmı için geliştirdiğimiz kütüphanenin yazılım mimarisi Şekil 4 de gösterilmektedir. Buradaki iş akışının iki önemli adımı vardır. İlk adım, kernel kodunun üretilmesi ve seçilen cihaza göre derlenmesini içerir. Kütüphanemiz, OpenCL dilini taşınabilirlik özelliğinden dolayı arka planda (back-end) kullanıyor. İkinci adım ise kernel kodunun çağrılmasında oluşur. Şekil 5 ise kütüphanenin bu iş için yapması gereken işlemleri gösteriliyor. Burada sistemimiz kernel fonksiyonun analizinden elde ettiği bilgileri kullanarak, gereksiz CPU ve GPU bellekleri arası veri kopmamalarını engelleyen bir optimizasyon algoritması geliştirilmiştir.





Şekil 4*: Kütüphane yazılım mimarisi*

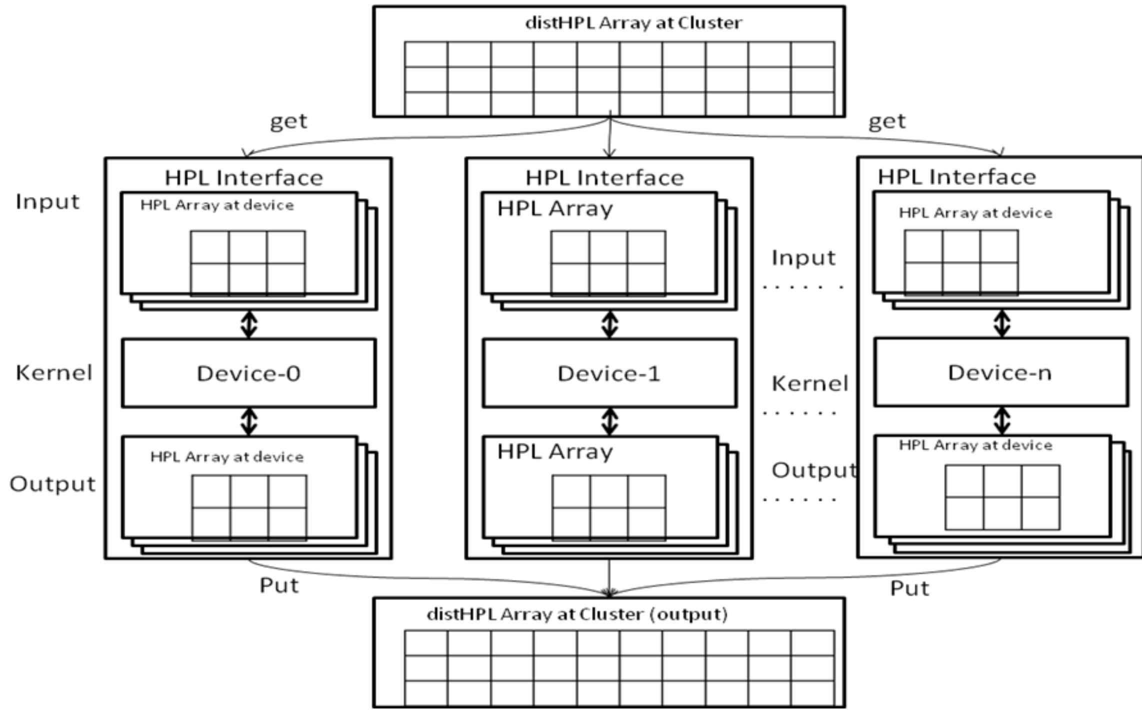
Şekil 5: *Kernel çağırma için kullanılan iş akış diyagramı*

4 Gereç ve Yöntem

(İş Paketi 1.1:
Projede Önerilen Kütüphanenin Dağınık Bellekli Heterojen Sistemlerde Gerçekleştirilmesi Tamamlanması.)

4.1 DistHPL Programlama Modeli

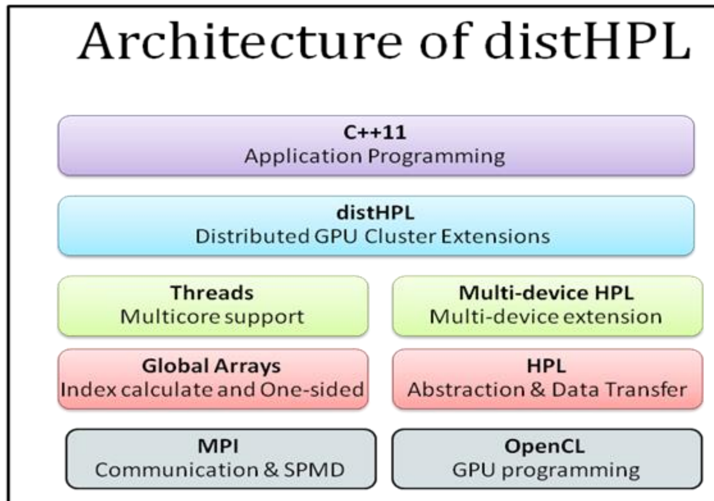
DistHPL kullanıcısına distArray isminde bir yapı tanımlattırıyor. Bu distArray dağınık clusterda blok modeli ile dağıtıyor, her bir processor bu array bir parçasına (tile) sahip olmakta. Sistemin sağladı metotlar ile bu arrayın belli parçaları okuyarak, GPU beslemekte. GPU gelen bu array parçalarını hızlı bir şekilde işlenmekte, hesaplar yapılmakta ve bunun kendi CPU'sunun sahip olduğu distArray parçasına geri yazmaktadır.Şekil 6 bunu görsel olarak açıkladık.



Şekil 6: distHPL programlama modeli.

4.2 DistHPL Yazılım Mimarisi

Şekil 7 DistHPL yazılım mimarisini göstermektedir. Burada distHPL kütüphanesinin Message Passing Interface(MPI) ve Global Arrays (GA) isimli iki kütüphaneyi dağınık sistemlerde tek yönlü haberleşme(put/get) kütüphanesi olarak kullandık. Ayrıca heterojen sistemler içinde OpenCL kullananmak yerine bizim geliştirdiğimiz HPL kullandık. Bunun üzerinde oturan çoklu GPU kullanmaya yardımcı olan HPL extensinini kullandırmakta distHPL bir parçasıydı. Kütüphane distArray isminde dağınık sistemler üzerinde bulunan bir array yapısı oluşturdu. Bu array yapısında global indexler kullanarak array bölümlerini HPL Arraylerine taşiyip, burada bulunan GPUlar üzerinde zaman alıcı hesaplamaları yaptırılmasını sağlayan modeli oluşturduk.



Şekil 7: distHPL yazilim mimarisi

4.1 DistHPL Matrix Multiplication Örneği

Şekil 8 ise bir matrix multiplication (MM) çarpımı yapan distHPL ile yazılmış bir program vermektedir. Eğer bu program MPI+OpenCL ile yazılsa çok daha karmaşık olacaktır. Bu programda sistem üç tane distributed array (distArray) tanımlamakta ve bu arraylerin gerekli kısımları (Array Section) global index kullanarak okunmakta ve yazmakta, Dağınık sistemde arrayleri lokale getirince, GPUlarda lokal çarpma yapmaktadır. Ve bu çarpma işlemleri sisteme bağlı tüm GPU aynı anda paralel olarak yapılmakta, buda çok büyük hız kazanımlarına neden olmaktadır.

distHPL içinde Range isminde bir yapi belirledik. Bu yapının iki tane elemanı vardır. Birincisi en düşük index verir, ikincisi ise en büyük index verir. Örnegin, r1 = new Range(0,99). Bunu her bir dimesion içindeki array bölümünü belirlemek için kullanıyoruz.

Haberleşme primitifleri distHPL tarafından otomatik olarak üretilir; bunlar tek yönlü okuma ve yazma rutinleridir. Örneğin, av = dA(Arange1,Arange2) statement distHPL array olan dA belli parçalarını, Range ile gösterilen parçalarını her bir processorun local av arrayina kopyalar. Bu kopyalama işi tek yönlü receive olarak belirtebiliriz. Eğer yukarıdaki assignment stament tersi ise, yani distHPL array sol tarafta ise, bu durumda distHPL otomatik olarak tek yönlü haberleşme kodu üretir.

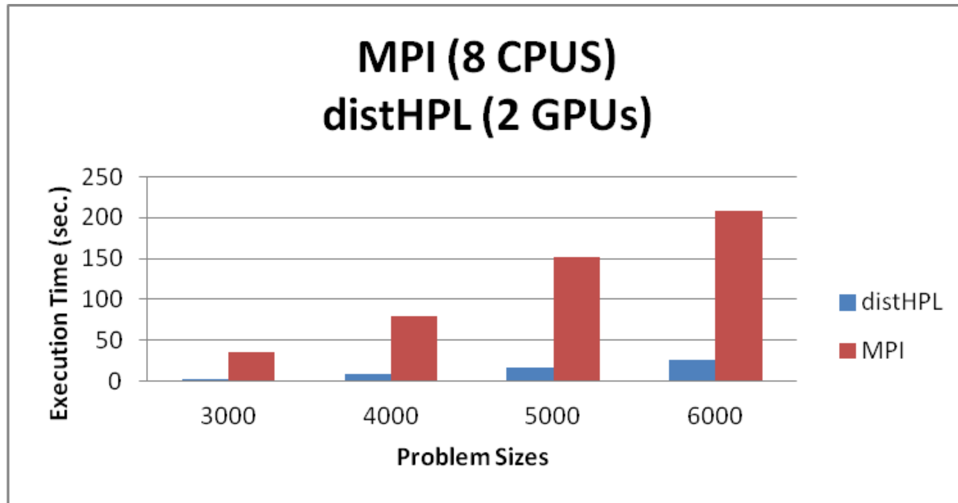
```
void matmulGPU(Array<double, 2> a, Array<double, 2> b, Array<double, 2> c)
{ // Kernel

       Size_t k;

       c[idx][idy] = 0.0;

       for_(k =0, k < N, k++) {

          c[idx][idy] += a[idx][k] * b[idy][k];

       } endfor_

}

void matrix_multiply() {
       // create a distHPL array dA, dB and dC
       auto dims = make_tuple(N,N);
       distArray dA(type, 2, dims),dB(type, 2, dims),dC(type, 2, dims);
       /* initialize dA and dB */
       …..
       // STEP 1: Get the blocks from dA to av
       Range Arange1 = new Range(dA.lo[0],dA.hi[0]);
       Range Arange2 = new Range(0,dA.dims[0]-1);
       Array<double, 2> av(Arange1.local_sz(), Arange2.local_sz());
       av = dA(Arange1,Arange2); //one way communication to read
       // Step 2: Get the blocks from dB to bv
       Range Brange1 = new Range(0,dB.dims[1]-1);
       Range Brange2 = new Range(dB.lo[1],dB.hi[1]);
       Array<double, 2> bv(Brange1.local_sz(), Brange2.local_sz());
       bv = dB(Brange1, Brange2); //one way communication to read
      // Step 3: Transpose bv.
       Array<double, 2> bv_trns(bv.getDimension(1), bv.getDimension(0));
       for(i=0; i < bv.getDimension(0); i++)
         for(j=0; j < bv.getDimension(1); j++)
            bv_trns(j,i) = bv(i,j);
       // Step 4: local GPU matrix multiple
       Array<double, 2> *cv = d_c.localArray;
       int M = dC.local_dims[0];
       int P = dC.local_dims[1];
       eval(matmulGPU).device(Device(GPU,0).global(M,P).local(5,5)(av,
   bv_trns, cv );
`
```

Şekil 8: distHPL yazılmıs MM algoritması .

5 Bulgular

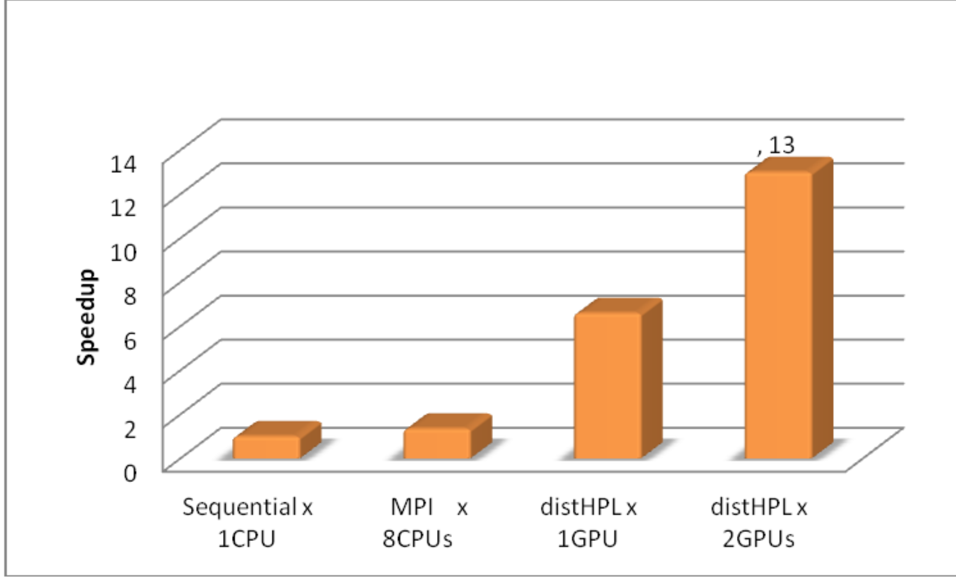
(İş Paketi 1.2:
Projede Önerilen Programla Modelin Dağınık Bellekli Heterojen Sistemlerde Değerlendirilmesi)

Geliştirilen distHPL iki farklı sistemde değerlendirdik. Birinci sistemde Kadir Has Üniversitesinde olan 8 çekirdekli ve 2 GPUs (Tesla C2050/C2070 GPU modeli) bir sistem. Bu sistem fiziksel olarak dağınık bir sistem değil fakat kullandığımız MPI kütüphanesi bunu dağınıkmış gibi yazilimsal olarak görmektedir. Şekil 9 biraz önce örneğini verdiğimiz MM algoritmasının farklı boyutlarda performansı verilmekte. Burada MPI sistemdeki tüm 8 CPU kullanmakta, distHPL ise 2 GPU kullanmakta, buna rağmen distHPL yaklaşık 10 kat hızlı çalışmaktadır. Bunuda distHPL sistemde var olan GPUlari kullanılmasıdır.



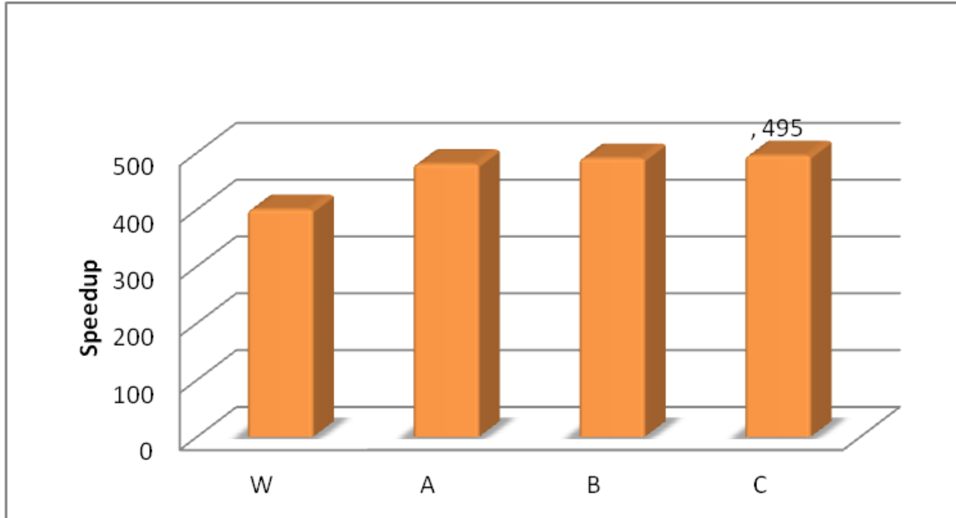
Şekil 9: Değişen büyüklükte matrix multiplication. MPI ve distHPL ayni makinada çalışmaktadır. MPI tüm 8 CPU kullanmakta, distHPl ise 2 GPU kullanmaktadır.

Şekil 10 ise serial CPU yazılan MM algoritmasını 8 CPU çalışan MPI ve 1 GPU ve 2 GPU çalışan distHPL karşılaştırmakta. Burada seri olarak tek CPU çalışana göre, distHPL 2 GPU yaklaşık 13 kez hızlı bitirmektedir. Fakat MPI ise data boyutunu küçük bulduğundan sadece 2 kat hızlı çalışmaktadır.



Şekil 10: MM (N=3000) seri MM nin hızı MPI ve distHPL farlı konfigürasyonlarda karşılaştırılması.

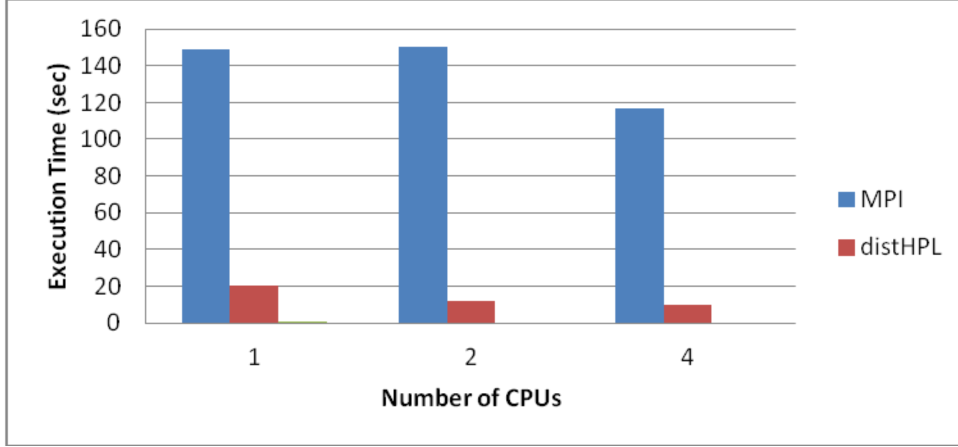
Önerilen Programlama Modeli ile NAS Paralel Mihenk Taşları Programınınlarindan olan EP yazılıp bununla ilgili performansı **Şekil 11** vermektedir. EP ideal şekilde parallelştirildigi için 2 GPU yaklaşık 500 kat hızlı çalışmaktadır. Bu karşılaştırma seri CPU yazılımına göre yapılmıştır.



Şekil 11: NAS EP testinin 2 GPU çalışan distHPL yazılmışının seri tek CPU çalışan C++ yazılmışının karşılaştırılması.

TÜBİTAK

Ayrıca Coruna Üniversitesinde fiziksel olarak dağınık GPU clusterinda distHPL ve MPI karşılaştırttık. Sistem Nvidia Tesla Kepler K20m GPU larını bulunmaktadır. Bu sistemin CPU'ları arası haberleşme için Infiniband network kullanılıyor. Burada ayni makinada MM(N=5000) benckmarkını, MPI sadece 4 CPU kullanıyor, distHPL ise 4 GPU kullanıyor. Her bir CPU nun sadece bir GPU var).Burada distHPL yaklaşık olarak 12 kat hız gösteriyor.



Şekil 12: Matrix Multiplication(N=5000) on Pluto platform. (4CPU, 4GPU)

6 Genetik Algoritma Tabanlı Moleküler Docking Algoritmalına Çoklu GPU Desteği Kazandırma

(İş Paketi 2.1:
Kütüphaneyi kullanarak çoklu GPUlarda moleküler docking tamamlanması:)

Daha önce hazırladığımız Moleküler Docking yazılımına çoklu GPU desteği kazandırmak için island-based GA ile daha ileri düzey bir paralel eştirme uygun görülmüştür. Bu tarz bir paralel eştirmede hiç bir device bir diğerinin verilerine ihtiyaç duymayacağı gibi hesaplama sürelerinin birbirleri ile senkronize olma zorunluluğu da ortadan kaldırılmıştır. Bunun için farklı GPU sayısı kadar CPU süreci oluşturulmuş ve bu süreçler GA tarafından kullanılacak rastgele sayı testlerini oluşturmanın yanında GPU fonksiyonların yönetecek olan host sistemini de içermesi sağlanmıştır.

Bu aşamada öncelikli olarak rastgele sayı setlerini hazırlayan CPU süreci GPU fonksiyonlarını işleme alır. GPU fonksiyonu çağrıları çağrıldıkları yönetici süreci bloklamadıkları için bu süreç GPU hesaplaması sürerken bir sonraki GA için gerekli olacak olan rastgele sayı setini hazırlar. Böylece farklı hesaplama üniteleri yoğun bir şekilde meşgul edilmiş olunur.

Eksik kalan tek kısım GPU sayısından daha fazla sayıda olan CPU çekirdeklerinin atıl kalmasıdır. Bu çekirdeklerin meşgul edilmesini sağlayan herhangi bir iş yükü mevcut değildir. Daha sonra CPU'nun da bir OpenCL device olarak tanıtıldığı bir optimizasyon bu konudaki açığı da kapatabilir. Bu konuda çalışma henüz yapılmamıştır.

6.1 Moleküler Docking Performans Sonuçları

Seri algoritmaya karşılık paralel algoritma ile elde edilmiş hızlanma aşağıda belirtildiği gibidir. Hız testlerinin yapıldığı bakine 6GB 384-bit GDDR5 VRAM içeren 2048 süreç işleme yeteneğindeki her biri 2 adet 850Mhz AMD FirePro D700 GPU'su ile birlikte 8x Intel 3Ghz Xeon E5 CPU kullanır. Derleyici olarak LLVM 6.1 tercih edilmiştir. Her bir docking deneyi birden fazla GA operasyonuna ihtiyaç duyduğu için (25, 50 ve 100'er GA operasyonu olmak üzere) üç farklı şekilde ölçüm alınmıştır.

Tablo 1: Molekuler Docking Hızlanma - Tek GPU

	Hızlanma	
	50 GA operasyon	100 GA operasyon
Bileşik A	22	23
Bileşik B	23	23
Bileşik C	21	20

GPU üzerinde çalışan algoritmayı incelediğimizde GPU'nun bellek bant genişliğinin tam olarak kullanılmadığını fark ettik. Buna dayalı olarak bellek optimizasyonuna gidilmesi gerektiğine karar verdik. *Tiling* diye adlandırılan verilerin GPU akış işlemcilerinin daha hızlı erişebileceği bir bellek biriminde tutulması bu konuda yapılması gereken optimizasyondur.

Tablo 2: Moleküler Docking Hızlanma - Çoklu GPU

	Hızlanma	
	1 GPU (100 GA)	2 GPU (100 GA)
Bileşik A	23	42
Bileşik B	23	43
Bileşik C	20	41

Tablo 2’de görüldüğü gibi birden fazla GPU kullanmak ciddi bir ek zaman maliyeti üretmez. Tek GPU kullanımına göre nerdeyse lineer bir hızlanma görülmüştür.

Geliştirilmiş algoritmanın en verimli şekilde performans farkı gösterebileceği eşit olmayan birden fazla GPU kullanımında nasıl bir performans sergileyeceği test edilmelidir. GA operasyonlarının çoklu GPU’ya dağıtılış şekli tamamen asenkron olduğundan bu konuda performans testleri ile aydınlatılmalıdır. Bu konuda ek bir optimizasyon gerekeceği sanılmamaktadır.

7 Global Birebir Ağ Hizalama Problemine HPL Tabanlı Çözüm

(İş Paketi 3.1:

Kütüphaneyi kullanarak proteinler arası alignment yapan bioinformatik algoritmaların GPU tamamlanması)

Hesapsal biyoloji ve biyoinformatiğin temel problemlerinden biri hizalamadır. Kaynaklık eden deneysel tekniklerin gelişimiyle beraber çoğalan dizi verilerinin (DNA, protein) karşılaştırılması gereksinimi, 1970ler, 80ler ve özellikle de 90ların başında dizi hizalama (sequence alignment) probleminin ve çok çeşitli versiyonlarının oldukça yoğun çalışılmasına yol açmıştır. Başarımı yüksek hizalamalar, homolog (evrimsel olarak ortak ata diziden gelen) diziler hakkında bilgi verdiği kadar, aynı ya da benzer fonksiyona sahip olabilecek diziler hakkında ve de bir grup molekülün paylaştıkları ortak motifler hakkında faydalı ipuçları sunar. NCBI'nin sunduğu HomoloGene sistemi ve Inparanoid (Remm et al., 2001), dizi hizalaması kullanarak türler arası homolog grupları oluşturan sistemlerden bilinen örneklerdir. Diğer taraftan, son on yılık süreç içinde gelişen tekniklerle beraber, sadece tek tek moleküllerin dizi verileri değil, ek olarak moleküllerin birbirleriyle olan etkileşimlerini de içeren biyokimyasal veriler elde edilmiştir. Aralarında maya iki-hibrit sistemi (Finley ve Brent, 1994; Ito et al. 2000), koimmünopresipitasyonlu kütle spektromisinin de (Aebersold ve Mann, 2003) bulunduğu yüksek veri çıkışlı deneysel tekniklerin gelişimi, birçok organizmaya ait geniş ıskalalı protein-protein etkileşim (PPE) ağlarının çıkarsanmasına olanak sağlamıştır. Ayrıca tüm genom üzeri gen füzyon analizi, metabolik yeniden yapılandırma ve gen eş-ifade tabanlı hesapsal algoritmaların (Goh ve Cohen, 2002; Marcotte et al. 1999; Skrabanek et al. 2008) tasarımı ise, bahsi geçen çıkarsamaların hem nicel artışına, hem de nitel güvenilirlik belirlenimine büyük katkı sunmuştur. Elde edilen geniş ölçekli etkileşim verileri, hizalama problemine yepyeni bir boyut kazandırmıştır. Bir veya birden fazla organizmaya ait moleküllerin (bu durumda proteinlerin) hizalanmasında (eşleştirilmesinde) artık sadece moleküllere kaynaklık eden dizilerle yetinilmeyecek, moleküllerin aralarındaki fiziksel etkileşimlerden oluşturulan ağdan da faydalanılabilecektir. Ağ hizalaması (network alignment) terimiyle ifadesini bulan problem, formel olmayan genel bir tanımla, hem hizalanan moleküllerin dizisel benzerliğini hem de hizalama sonucunda elde edilen topolojik benzerliği (hizalanan moleküller arasında korunan etkileşimlerin sayısı) maksimize edecek molekül eşleşmelerini bulma problemidir. Ağ hizalama problem versiyonlarından en yaygını *global birebir ağ hizalaması*dır. Global birebir ağ hizalama için literatürün bilinen hizalama algoritmalarından IsoRank, GRAAL, H-GRAAL, MI-GRAAL, GA, PATH ve PISwap algoritmalarının kullandığı problem tanımı aynıdır. $G1=(V1, E1)$, $G2=(V2,E2)$ verili PPE ağlarına karşılık

gelen yönsüz çizgeler olsun. Burada, $1 \leq i \leq 2$ için, Vi düğümler (proteinler) kümesini, Ei ise ayrıtlar (proteinler arasi etkileşimler) kümesini ifade eder. Düğüm kümesi V1 ve V2'deki düğüm çiftlerinden oluşan bir *hizalama çizgesi, A*=(V, E) tanımlanabilir. <ui, vj> ε V için ui ε V1, vj ε V2 ve de herhangi bir düğüm çifti <ui, vj> ε V, <ui', vj'> ε V için ui ≠ui' ve vj ≠vj' koşulu sağlanmalıdır. Ayrıt kümesi E öyle tanımlanır ki, her *korunmuş etkileşim*, hizalama çizgesinde bir ayrıta karşılık gelsin. Yani, V'de yer alan herhangi bir çift düğüm, <ui, vj> ve <ui', vj'> için (<ui, vj>, <ui', vj'>) ε E, ancak ve ancak (ui, ui') ε E ve de (vj, vj') ε E. Bu durumda bir hizalama çizgesi A için, hizalama skoru GNAS (Global Network Alignment Score) şöyle tanımlanabilir:

$$\mathbf{GNAS(A) = \lambda \times |E| + (1-\lambda) \times \sum seq(ui, vj)}$$

Topolojik benzerliğin hizalama skoruna katkısı formüldeki ilk terimle sağlanırken, dizisel benzerliğin katkısı ikinci terimle ifade edilir. Sabit λ ε [0,1] topolojik benzerlik ve dizisel benzerliğin göreceli önemlerini değiştirmek için kullanılan bir dengeleme parametresidir. İkinci terimde yer alan toplam, hizalama düğüm kümesi V'de yer alan bütün düğümler üzerinde tanımlıdır. <ui,vj> ε V için *seq(ui, vj)*, ui ve vj düğümlerine karşılık gelen proteinlerin dizisel benzerliklerinin ölçüsü olarak protein çiftinin BLAST bit skorudur. Bu tanımlar doğrultusunda global birebir ağ hizalama problemi formel olarak, verili iki çizge G1, G2 ve dizisel benzerlik fonksiyonu *seq* için GNAS skorunu en iyileştiren hizalama çizgesi A'yı bulma problemi olarak tanımlanabilir.

Projenin bu bölümünde global birebir ağ hizalama problemi için önerilen popüler bir hizalama algoritması olan SPINAL algoritmasının (Aladağ ve Erten, 2013) projenin genel çerçevesi içerisinde önerilen HPL kullanılarak GPU üzerinde koşum sağlanacak şekilde yeniden tasarımı yapılmış, bu tasarımın kod gerçekleştirimi yapılmış ve deneysel karşılaştırmalar gerçekleştirilmiştir. Takip eden altbölümde öncelikle SPINAL algoritmasının kısa bir özeti sunulacaktır. Sonraki altbölümde tasarladığımız GPU üzerinde koşumlanabilir G-SPINAL algoritması tanıtılacak ardından deneysel sonuçlar sunulup tartışılacaktır. Burada özeti sunulan çalışmanın ilk yazım versiyonu hazırlanmıştır, yazım iyileştirmelerinin ardından hesapsal biyoloji alanında tanınmış bir dergiye makale olarak gönderilecektir.

7.1 SPINAL Algoritması

SPINAL algoritması pseudocode olarak *Algoritma 1*'de sunulmuştur. Algoritma temel olarak iki aşamadan oluşur: Kaba-ayarlı eşleşme skoru tahmin aşaması ve de ince-ayarlı çelişki çözümü ve hizalama. 3-14 arası satırlar ilk aşamayı ifade ederken kodun geriye kalanı ikinci aşamaya aittir.

Kaba-ayarlı birinci aşama, P(ui,vj), ui, V1'den bir düğüm, vj, V2'den bir düğüm olmak üzere, ui ile vj'nin tahmini eşleşme skoru, yani ui, vj hizalamasının GNAS skoruna tahmini katkısını bulmaya yöneliktir. N(ui), N(vj) sırasıyla ui ve vj'nin G1 ve G2'deki komşulukları olsun. NBG=(N(ui), N(vj), E_BG), katmanları, N(ui) ve N(vj) olan tam (complete) bir çift katmanlı çizge olsun. E_BG'deki her ayrıt (xi, yj) için P(xi, yj) ağırlık olarak atanır. M ayrıt kümesi, NBG çizgesinin maksimum ağırlıklı eşleştirme sonucu olsun. Verili bir düğümün, bulunduğu çizgedeki derecesini *deg* fonksiyonu ile ifade edelim. Bu durumda P(ui,vj) özyineli olarak M içinde yer alan komşuluk çiftlerinin tahmini skorları ve dizisel benzerlikleri üzerinden şöyle tanımlanır:

$$\mathbf{P(ui,vj) = \lambda \times \frac{\sum \frac{P(xi,yj)}{deg_{G1}(xi) x deg_{G2}(yj)}}{\sqrt{|M|}} + (1-\lambda) \times seq(ui, vj)} \quad \textbf{[Eq2]}$$

1: **Input:** $G_1 = (V_1, E_1), G_2 = (V_2, E_2), seq, \alpha$
2: **Output:** Node set V_{12} of the global alignment network A_{12}
3: // Coarse-grained
4: **for all** $u_i \in V_1, v_j \in V_2$ **do**
5: $P(u_i, v_j) = \alpha \times DegDiff(u_i, v_j) + (1-\alpha) \times seq(u_i, v_j)$
6: **end for**
7: **repeat**
8: $P' = P$
9: **for all** $u_i \in V_1, v_j \in V_2$ **do**
10: construct $\mathcal{NBG}(\{\prec u_i, v_j \succ\}, P')$
11: construct contributors set C of $\mathcal{NBG}$
12: compute $P(u_i, v_j)$ as in Equation (2)
13: **end for**
14: **until** enough iterations
15: // Fine-grained
16: $SP =$ List of $\prec u_i, v_j \succ$ sorted w.r.t P, for $u_i \in V_1, v_j \in V_2$
17: **repeat**
18: // Find new connected component in A_{12}
19: pop unaligned $\prec u_i, v_j \succ$ from SP, insert into V_{12}
20: **repeat**
21: construct $\mathcal{NBG}(V_{12}, P)$
22: construct contributors set C of $\mathcal{NBG}$
23: swap improvements for each $\mathcal{NBG}$ edge not in C
24: insert $\prec x_i, y_j \succ$ into V_{12}, for each $(x_i, y_j) \in C$
25: **until** no contributors
26: **until** no unaligned pair in SP

Algoritma1: SPINAL algoritması.

SPINAL algoritmasında tahmini skorlar matrisi P'yi oluşturmak için, enerji minimizasyonunda yaygın kullanılan (Höltje et al., 1997) basit bir tekrarlı bayır yöntemi kullanılır. Her çiftin skorunu 1-boyutlu koordinat gibi ele alırsak, her tekrarda her bir nokta için komşuluk noktalarının (bu durumda M içinde yer alan komşuluk çiftleri) uyguladığı “çekme” kuvveti ile yeni bir koordinat belirlenir. Tekrarlar, sistem lokal bir minimum enerji seviyesine eriştiğinde, yani her bir çiftin skorunun bir önceki tekrardaki skorla aynı olduğu durumda, sonlanır. Benzer iteratif yöntemlerde olduğu gibi hem lokal minimuma erişim için gerekli tekrar sayısının azlığı bakımından hem de daha kaliteli sonuçlar elde etmek için, iyi bir başlangıç konfigürasyonu ile tekrarları başlatmak önemlidir. Derece farkları (0-1 arasında normalize edilmiş şekliyle) ve dizisel benzerliğin, λ bazlı konveks kombinasyonu iyi bir alternatiftir. İki ağın hizalanması söz konusu olduğunda, gereken tekrar sayısı k, ağlardaki düğüm sayıları sırasıyla n_1, n_2, maksimum derece de d_1, d_2 olmak üzere, SPINAL algoritmasının bu aşamasının zaman kompleksitesi $O(kn_1n_2 d_1d_2 \log (d_1d_2))$ olur.

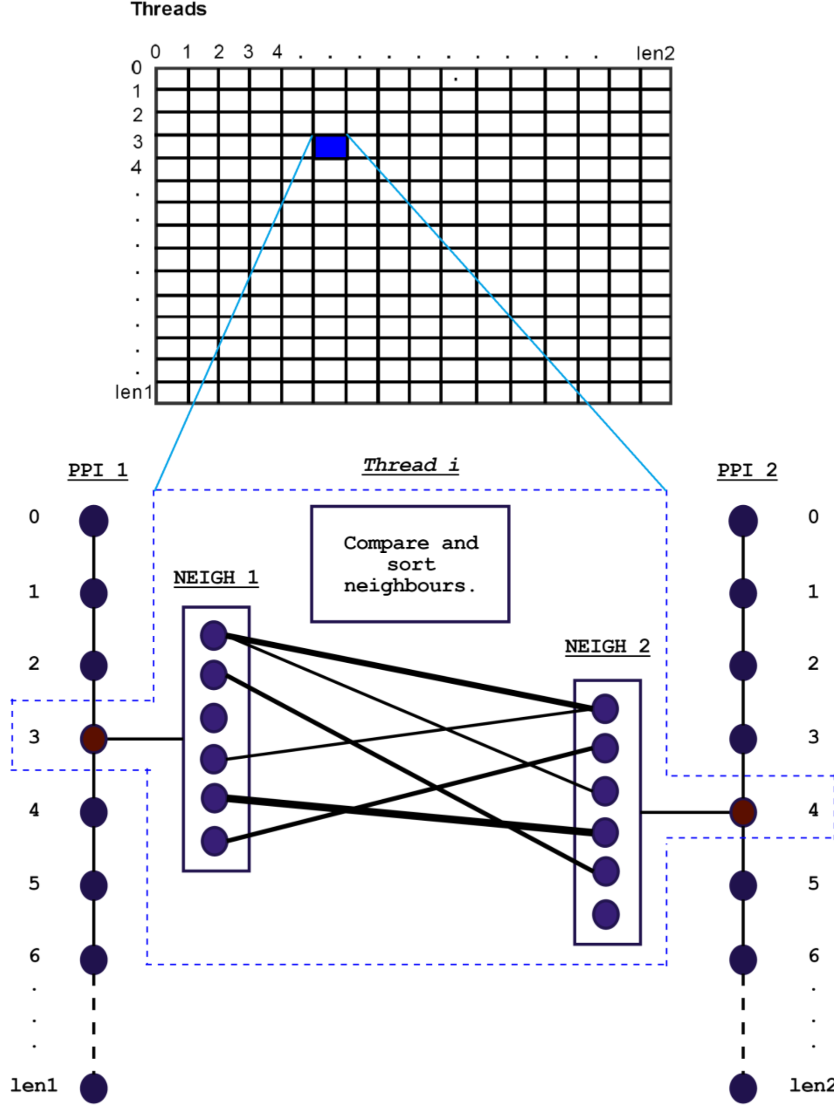
SPINAL'in ince-ayarlı çelişki çözümü ve hizalama ile ilgili ikinci aşamasının temel fikri 'tohumla-ve-genişlet'e (seed-and-extend) dayalıdır. Bu aşama, her tekrarında hizalama çizgesi A'nın bağlı bir parçasını (connected component) bulmaya odaklı tekrarlı bir buluşsaldan oluşur; bakınız Algoritma1 kodu, 16-26 satırlar. Her tekrar, bağlı parçanın oluşumuna, elemanları henüz eşleşmemiş ve önceki aşamada üretilen P skor matrisinde en yüksek tahmini skora sahip çifti bağlı parçada hizalayarak başlar. Her bağlı parçanın kendisi de tekrarlı bir oluşturma süreci boyunca büyütülerek oluşturulur. Bağlı parça *nerdeyse* sığ öncelikli arama (breadth first search) tarzında büyür. Herbir tekrarda, parçanın yeni katmanı (o andaki parçadan tek ayrıtla ulaşılabilen) oluşturulur. PL1, G1'in önceki tekrarda elde edilen katmanı olsun. Yeni katman L1, PL1'de eşleşmemiş düğümlerden ve de PL1 komşuluğunda yer alan eşleşmemiş düğümlerden oluşur. Benzer tanımlar PL2 ve L2 için de geçerlidir. Katman kümeleri L1 ve L2 olan bir çift-katmanlı çizge NBG oluşturulur. Biri L1'de biri L2'de yer alan bir düğüm çifti arasına, eğer çift şimdiye kadar oluşmuş hizalamalarla bir korunmuş ayrıt üretiyorsa, NBG'de bir ayrıt eklenir. Böyelikle NBG'deki her ayrıt, oluşturulacak sığ öncelikli aramada hizalama çizgesi parçasına eklenecek yeni katmanda yer alabilecek olası bir düğüme karşılık gelir. Her NBG ayrıtının ağırlığı yine tahmini skor matrisi P'den elde edilir. NBG üzerinde yapılan maksimum ağırlıklı eşleştirme halihazırdaki parçaya eklenecek yeni katman için adaylardan oluşur. Son olarak aday kümesindeki her eşleşme, kümede yer almayan çelişen eşleşmelerle (en az bir düğümleri kesişen) GNAS skoruna katkı bağlamında kazanç karşılaştırması yapılır. Çelişen eşleşmeler daha fazla kazanç sağlarsa aday kümesindeki söz konusu eşleşme ile yer değiştirirler. Ayrıntılı bir analize girmeden, n_1 büyük ağdaki düğüm sayısı olmak üzere, önerilen sığ öncelikli büyüme algoritmasının zaman kompleksitesinin $O(n_1 n_2 d_1 d_2 + n_1^2 \log n_1)$ olduğunu belirtelim.

7.2 G-SPINAL: SPINAL Algoritmasının GPU Versiyonu

HPL (Heterogeneous Programming Language) OpenCL tabanlı olduğundan altyapı mimarisi benzerdir. Temel fark HPL'nin kolay programlanır olmasıdır. Önceki bölümde tanıtılan global birebir ağ hizalama algoritması SPINAL'in orjinal versiyonu LEDA yazılım kütüphanesi kullanılarak CPU'da çalışacak şekilde tasarlanmıştır. LEDA kitaplığında çizge temsili kitaplığın kendi veri yapısı olarak bağlı listeler tabanlıdır. Burda temel sorun komuşuluk tespitinin yavaş olmasıdır; V düğüm kümsi olmak üzere tek bir komşuluk tespiti $\Theta(|V|)$ kadar zaman gerektirebilir.

Proje çalışmalarında bu bağlamda yapılan ilk iş bahsedilen hem bahsedilen bu zaman gereksimi kısıtını aşmak hem de HPL kitaplığı basit C/C++ dizilerinde çalıştığından SPINAL'in kullandığı çizgeleri dizi

tabanlı yapılarla temsil etmek oldu. Bu gerçekleştirim bile başlıbaşına koşum zamanı açısından oldukça yüksek performans sağladı. Ardından, normal C/C++ dizilerinden aha farklı notasyonlu HPL dizileri ve gerekli HPL değişkenleri tanımandı. Şekil 13'de her bir paralel iş yükünün (thread) nasıl çalıştığı özetlenmiştir.



Şekil 13: G-SPINAL'de thread tasarımı.

Len1, len2 boyutlarında iki dizi tanımlıdır. İlki birinci prtotein etkileşim ağı PPI1'in düğüm listesini tutarken ikincisi PPI2'nin düğüm listesine karşılık gelir. Her bir düğümün kendi ağında bir komşuluğu vardır. Şekilde PPI1'den 3 numaralı ve PPI2'den 4 numaralı düğümlerin komşulukları temsili olarak gösterilmiştir. Bu traz verili bir komşuluklar çiftinde ayrıtların ağırlıkları düğüm benzerliklerine

(Algoritma 1'de P matrısinde tanımlı skorlar) karşılık gelen bir çiftkatmanlı (bipartite) çizge oluşur. Bu komşuluk çiftleri üzerinde tanımlı benzerlik skoru hesaplama, yani P matrisi skoru tanımlama işlerinin her biri için bir thread tanımlanmıştır. Dolayısıyla toplamda len1*len2 thread söz konusudur. Verili bir tekrarda (iterasyon) her bir P skoru hesaplama işi temel olarak sıralama ve karşılaştırma gerektirir ve aynı tekrarda her bir P skoru ötekilerden tamamen bağımsızdır. Bu da SPINAL'de yapılan teme hesapsal işlerin doğru bölümlenmesini sağlar.

Kısaca gerçekleştirim ayrıntılarından bahsetmek gerekir. OpenCL kullanımında temel zorluklardan biri kernel kodunda GPU'da çalışan bir ”malloc()” rutininin olmayışıdır. Öte yanan geliştirdiğimiz G-SPINAL bağlı listeler, sözlükler değişken boyutlu diziler gibi pek çok dinmik veri yapısı gerektirmektedir. Bu problemi çözmek için iki seviyeli bir strateji geliştirdik. Öncelikle kernelde statik veri yapıları kullandık. Eğer düğüm komşuluk boyutu küçükse bu strateji değerli hafıza kaybına neden olabilir. Ancak bu yaklaşımı ancak komşuluk boyutu küçükse tercih ettiğimizden bir sorun oluşturmaz. İkinci olarak da, araç hafızasında kernel koşumlarından önce sabit sayılı veri yapıları oluşturduk ve bu yapıları karşılıklı dıştalama (mutual exclusion, mutex) kilitlerine dayanarak pek çok kernel arasında paylaştırdık. Şu anki gerçekleştirimde 100 adet kernelin paralel koşumu için 100 veri yapısını kontrol eden 100 tane mutex kilidi kullandık. Geliştirdiğimiz G-SPINAL potansiyel olarak len1*len2 lik paralelizme sahiptir. Eşzamanlı hesaplar için GPU'nun işleme elemanları açısından herhangi bir kısıtlama söz konusu değildir.

7.3 Karşılaştırmalı Deneysel Sonuçlar

G-SPINAL gerçekleştirimi C++ ve proje çerçevesinde geliştirilen HPL yazılım kitaplığı kullanılarak yapılmıştır. Karşılaştırmalı deneysel sonuçlar iki amaçla yapılmıştır. Birincisi tasarlanan G-SPINAL'in SPINAL koşumuyla aynı hesapları gerçekleştirip aynı sonuçları verdiğini sınamak amaçlıdır. İkinci olarak da tasarlanan G-SPINAL'in SPINAL'e oranla koşum zamanı bağlamında performansıdır. Deneyler, dört tür çiftine ait protein-protein etkileşim (PPE) ve BLAST benzerlik skorları verileri kullanılarak gerçekleştirildi. Bu türler *Saccharomyces cerevisiae, Drosophila melanogaster, Caenorhabditis elegans* ve *Homo sapiens*'ten oluşur. Bütün veriler IsoBase (Park et al., 2011) veri tabanından elde edilmiştir. Bu veriler aynı zamanda IsoRank ve IsoRankN gibi hizalama yöntemlerinde de kullanılmıştır. PPE ağ büyüklükleri sırasıyla şöyledir: S.cerevisiae ağında 5499 protein ve 31261 etkileşim, D.melanogaster ağında 7518 protein ve 25635 etkileşim, C.elegans ağında 2805 protein ve 4495 etkileşim ve son olarak H.sapiens ağında da 9633 protein ve 34327 etkileşim. Deneylerde sunulanlar, formel problem tanımında optimizasyon amacı olarak belirlenen GNAS skoru bağlamında elde ettiğimiz sonuçları

kapsar. G-SPINAL ve SPINAL algoritmalarının herbir tür çifti verisi için, bu fonksiyondan elde ettikleri skorlar ve koşum zamanları Tablo-1'de sunulmuştur.

Tablo 3: Spinal, Arrays ve G-Spinal zaman ve sonuç karşılaştırması.

Dataset	SPINAL		ARRAYS		G-SPINAL	
	(time)	*(score)*	*(time)*	*(score)*	*(time)*	*(score)*
ce-dm	8m 15s	2310	5m 33s	2369	1m 52s	2015
ce-hs	11m 45s	2277	8m 38s	2318	2m 30s	2057
ce-sc	9m 34s	2288	8m 1s	2341	1m 40s	1918
dm-hs	40m 59s	5825	22m 6s	5894	8m 54s	0
dm-sc	31m 5s	5229	17m 39s	5322	6m 54s	5078
hs-sc	45m 15s	error	31m 46s	error	8m 58s	error

Tablo 3 de ilk çoklu kolon SPINAL sonuçlarını ortadaki çoklu kolon SPINAL'in çizgelerin dizi temsili ile gerçekleştiriminin yapıldığı versiyonu son çoklu kolon ise G-PINAL sonuçlarını verir. Görüleceği üzere G-SPINAL'in SPINAL'e oranla koşum zamanı performansı oldukça iyidir; bazı durumlarda 10 katı kadar hızlı çalışmaktadır. Verilen skorlar GNAS skorlarıdır. GNAS skorlarının farklı olması G-SPINAL'in SPINAL'den farklı hesaplamlar yaptığı anlamına gelmemektedir. Her iki algoritma birebir aynı hesapları yapmaktadır. Skor farklılıkları, SPINAL algoritmasının sıralamada Quicksort algoritmasını kullanmasından ve bu algoritmanın kendi içinde rastgelelik (randomization) kullanmasından kaynaklanmaktadır. SPINAL'in farklı koşumları da tablodaki skor farkları kadar farklar içeren sonuçlar üretebilmektedir.

TÜBİTAK

8 Sonuçlar

Bu çalışmada, dağınık heterojen CPU-GPU sistemlerinden çalışan bir programlama modeli geliştirilmiştir. Bu model C++ makro ve overloading özelliğini kullanarak geliştirildi. Geliştirdiğimiz sistemin ismi distHPL. DistHPL kendi özel array yapısını dağınık bellek üzerinde dağıdir. Ve hesaplamanin ihtiyaci duyulan haberleşme komutlarini üretir.

DistHPL geliştirilmesi adim adim olmuştur. İlk olarak tek CPU-GPU çalışan HPL kütüphanesini geliştirilmesi tamamlanmış. Sonrada ortak bellekli heterojen sistemlerde çoklu GPU kapsayacak şekilde altyapı genişlettirilmiştir. DistHPL yu yapıları kullanarak en üst katmanıtır.

Geliştirdiğimiz kütüphanenin çeşitli platformlarda taşına bilirliğini gösterdik. Kütüphaneyle heterojen sistemlere çok kolay program yazılacağını göstererek ve ayrıca üretilen kodun diğer programlama modelleriyle karşılaştırdığımızda çok iyi sonuçlar aldık. Örneğin dağınık heterojen 2 CPU-GPU platformda NAS paralel mihenktaşlarindan EP testinde yaklaşık 495 kat hızlı kazandık, tek CPU karşılaştırdığımızda. Diger yanadan distHPL yazılan Matrix Multiplication algoritmasi yakalşık MPI karşilaştırdgımızda 12 kat daha hızlı sonuç vermekdetir. Bu karşılaştırmada MPI 4 CPU kullanmada, distHPL is 2 tane CPU-GPU kulnamaktatır.

Ayrıca kütüphaneyi kullanarak iki farklı bioinformatik algoritması geliştirdik ve bu algoritmalar dada bir tanesi ilaç tasarımından kullanılan moleküler docking çözümüdür. Diğer algoritma ise global gene alignment yaptık. Bu iki algoritmada kütüphaneyle GPU kullanilarak katlanarak hızlanması göstermektedir.

Sonuç olarak projedeki hedeflerimizi gerçekleştirmek için geliştirdiğimiz, distHPL kütüphanesi heterojen sistemlerde programlamayı kolaylaştırmıştır.

9 Kaynakça

A. Klockner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih,. 2009. "APyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation." In.

A. L. Varbanescu, P. Hijma, R. van Nieuwpoort, and H. E. Bal. 2011. "Towards an Effective Unified Programming Model for Many-Cores." In *IPDPS Workshops 2011*, 681-92.

Altilar, B. Eskikaya and Turgay D. 2012. 'Distributed OpenCL Distributing OpenCL Platform on Network Scale', *IJCA Special Issue on Advanced Computing and Communication Technologies for HPC Applications*: 26-30.

AMD. 2008. "Stream computing user guide." In.

C. J. Newburn, B. So, Z. Liu, M. D. McCool, A. M. Ghuloum, S. D. Toit, Z.-G. Wang, Z. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang,. 2011. "Intel's array building blocks: A retargetable, dynamic compiler and embedded language." In *9th Annual IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO 2011)*, 224-35.

Eigenmann, S. Lee and R. "OpenMPC: Extended OpenMP Programming and Tuning for GPUs." In *Proc. of 2010 Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC), ,* 1-11.

GPGPU. 2011. "General Purpose Computation on Graphics Pro¬cessing Units." In.

Hoberock, N. Bell and J. 2011. 'Thrust.' in, *GPU Computing Gems Jade Edition. ch 26* (Morgan Kaufmann).

IBM. 2006. "C/C++ Language Extensions for Cell Broadband En¬gine Architecture." In.

IBM, Sony, and Toshiba. 2006. "Cell Broadband Engine Architecture." In.: IBM.

J. Duato, J. Pena, A. F. Silla, R. Mayo, and S. Quintana-Orti, E. . 2010. "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters." In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, 224–31. . Caen, France.

J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "OpenCL as a programming model for GPU clusters," in LCPC'11: Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing, September 2011. 2011. 'OpenCL as a programming model for GPU clusters', *LCPC'11: Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing*.

M. Strengert, C. M¨uller, C. Dachsbacher, and T. Ertl, . "CUDASA: Compute Unified Device and Systems Architecture." In *Eurographics Symposium on Parallel Graphics and Visualization EGPGV08. Eurographics Association*, 49–56.

Nieplocha, Jaroslaw, Robert J. Harrison, and Richard J. Littlefield. 1994. "Global arrays: a portable "shared-memory" programming model for distributed memory computers." In *Proceedings*

of the 1994 ACM/IEEE conference on Supercomputing, 340-49. Washington, D.C.: IEEE Computer Society Press.

Nvidia. 2008. 'CUDA Compute Unified Device Architecture', 2008.

10 Ekler

10.1 EK-1 Projeden Üretilen ve Proje Numarasına Atıf Verilerek Yapılan Yayınlar

- Moisés Viñas, Zeki Bozkus, Basilio B. Fraguela, Diego Andrade, Ramon Doallo: Developing adaptive multi-device applications with the Heterogeneous Programming Library. The Journal of Supercomputing 71(6):2204-2220 (2015)

- Moiśes Viñas, Basilio B. Fraguela, Zeki Bozkus, Diego Andrade, Improving OpenCL Programmability with the Heterogeneous Programming Library, Procedia Computer Science, Volume 51, 2015, Pages 110-119, ISSN 1877-0509

- M. Viñas, Z. Bozkus, B.B. Fraguela. Exploiting heterogeneous parallelism with the Heterogeneous Programming Library. Journal of Parallel and Distributed Computing, 73(12), pp. 1627-1638. December 2013.

- Viñas M., Bozkus Z., Fraguela B.B., Andrade D., Doallo R. Exploting multi-GPU systems using the Heterogeneous Programming Library, Proceedings of the 14th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2014 3-7July, 2014. Costa Ballena, Rota, Cádiz (Spain). ISBN: 978-84-616-9216-3. Pages: 1280-1292.

- Serkan Altuntaş, Zeki Bozkus,BAŞARIM: Ulusal Yüksek Başarımlı Hesaplama Konferanslar 1-2 Ekim, 2015, ODTÜ, ANKARA

11 Ekler

11.1 EK-1 Projeden Üretilen ve Fakat Yayınlanmayan Çalışmalar:

- Z. Bozkus, M. Viñas, B.B. Fraguela. **“distHPL: An Extension of HPL Programming Model for the Distributed GPU Clusters” (Draft)**
- Serkan Altuntaş, Zeki Bozkus, Basilio B. Fraguela “Small Molecule Docking with Multi-GPU Acceleration“ (Hakem değerlendirmesinde)
- Mohammad Sohaib, Zeki Bozkus and Cesim Erten. “PPI Network Alignment Using GPUs” (Draft)

12 Ekler

12.1 İş Paketlerin Tamamlanma Oranları

Bu projede yapmayi söyledigimiz her şeyi yapmaya çaliştik. Broje uzatmadan onceki iş paketi listemiz ve yorumlarımız.

Original İŞ-ZAMAN ÇİZELGESİ

İş Paketi Ad/Tanım	Yorumlarimiz
İş Paketi 1.1: **1. Dağınık Bellekli Heterojen Sistemlerde Veri Parallelizminin Geliştirilmesi** (DATA PARALLELISM)	Tamamlandik. distHPL yazilan Matrix Multiplication MPI 13 kat hızlı, GPU kullandigimiz için
İş Paketi 1.2 **2. Dağınık Bellekli Heterojen Sistemlere Görev Parallelisminin Gelişletirilmesi** (TASK PARALLELISM)	Kısmı olarak tamamlandik. Sisteme direk olarak Task Parallel koymadik, fakat lazzy Kopy teknığini kullandigimiz için aynı anda iki farkli GPU kernel çalişa bilmekde.
İş Paketi 2.1: **Projede Önerilen Kütüphanenin Dağınık Bellekli Heterojen Sistemlerde Gerçekleştirilmesi** (IMPLIMENTATION of THE LIBRARY)	Tamamladik. (Bkz: Bölüm 4)
İş Paketi 3.1: **Projede Önerilen Programla Modelin Dağınık Bellekli Heterojen Sistemlerde Değerlendirilmesi** (EVALUATION)	Tamamladık. (Bkz: Bölüm 5)
İş Paketi 4.1: **Önerilen Programlama Modeli ile NAS Parallel Mihenk Taşları Programının Yazılması**	Tamamladik. NAS EP, NAS FT (Bkz: Bölüm 5, Journal of Supercomputing makalesi)
İş Paketi 5.1: **Dağınık Belekli Heterojen Paralel Programlama Kütüphanesinin Gen İfade Analizi Uygulaması** (DEVELOPING GEN EXPRESION APPLICATION)	Tamamladik. (Bkz: Bölüm 6, 7)

Uzatma İŞ-ZAMAN ÇİZELGESİ

İş Paketi Ad/Tanım	Yorumlarimiz
İş Paketi 1.1: **Projede Önerilen Kütüphanenin Dağınık Bellekli Heterojen Sistemlerde Gerçekleştirilmesi Tamamlanmasi** (IMPLIMENTATION of THE LIBRARY)	Tamamladik. (Bkz Bölüm 4, distHPL draft ve Journal of Supercomputing makalesi)
İş Paketi 1.2: **Projede Önerilen Programla Modelin Dağınık Bellekli Heterojen Sistemlerde Değerlendirilmesi** (EVALUATION)	Tamamladik. (Bkz Bölüm 5 ve distHPL draft)
İş Paketi 2.1: **Kütüphaneyi kullanarak çoklu GPUlarda molekuler docking tamamlanmasi** (DEVELOPING MOLECULER DOCKING with THE LIBRARY)	Tamamladik. (Bkz Bölüm 6 ve BAŞARIM makalesi)
İş Paketi 3.1: **Kutuphaneyi kullanarak proteinler arasi alignment yapan bioinformatik algoritmalarin GPU tamamlamaksi** (DEVELOPING Protein to Protein Interaction APPLICATION)	Tamamladik.(Bkz Bölum 7)

TÜBİTAK
PROJE ÖZET BİLGİ FORMU

Proje Yürütücüsü:	Doç. Dr. ZEKİ BOZKUŞ
Proje No:	112E191
Proje Başlığı:	Dağınık Çok Çekirdekli Cpu Ve Çoklu Gpu Sistemleri İçin Heterojen Programlama Kütüphanesi
Proje Türü:	1001 - Araştırma
Proje Süresi:	24
Araştırmacılar:	CESİM ERTEN
Danışmanlar:	
Projenin Yürütüldüğü Kuruluş ve Adresi:	KADİR HAS Ü.
Projenin Başlangıç ve Bitiş Tarihleri:	01/03/2013 - 01/09/2015
Onaylanan Bütçe:	135470.0
Harcanan Bütçe:	0.0
Öz:	Bu dönemin yapilan diger bir iş ise projenin son hedefi olan disHPL kütüphanesinin tamamlanması olmuştur. Bu kütüphane dağınık sistemlerdeki GPU clusterlarinin programlamasini kolaylaştıran bir kütüphanetir. Bu kütüphane sayesinde programcı birden fazla MPI, OpenMP ve OpenCL/CUDA gibi çoklu programlama modelininin hepsini bir program içinde kullanip, programları içinden çıkılmaz hale getirilmesini önlüyor. Geliştirilen sistem hem dağınık CPUlari ve ayrica GPUlari kullandıklari için çok verimli olmaktatir. Bununla ilgili çalişmamiz ?distHPL: An Extension of HPL Programming Model for the Distributed GPU Clusters? dergi yayini yapma çalişmasındayız. DistHPL kütüphanesi dağınık sistemlerde çalişa bilmesi için gerekli adımların sıralı bir yöntem halinde yapilmasi gerekmedeydi. Çünkü distHPL tam çalişması için, multi-device support HPL çalişmasi gerekiyordu, örnegin bizim khas olan parallel makinamizda sisteme iki device baglı, bunlardan hangisini sececegimi: device(Device(GPU,mpirank) bilmesi gerrekiyor, eger bu yapılmaz ise sistem herzaman device 0 kullanarak, parallel çalişmayi engelliyordu. Ayrica altda yatan HPL kütüphanesininde tam çalişmasi gerekiyordu. Bizde yayınlarımızı bu sırada yaptık. ?İlk olarak tek GPU çalişmasi gerekiyordu. Bu bir journal yayini oldu: M. Viñas, Z. Bozkus, B.B. Fraguela. Exploiting heterogeneous parallelism with the Heterogeneous Programming Library. Journal of Parallel and Distributed Computing, 73(12), pp. 1627-1638. December 2013. ?İkici adimda çoklu GPU fakat bir CPU baglanmiş olmasi gerekiyordu. Buda bir dergi yayini oldu: Moisés Viñas, Zeki Bozkus, Basilio B. Fraguela, Diego Andrade, Ramon Doallo: Developing adaptive multi-device applications with the Heterogeneous Programming Library. The Journal of Supercomputing 71(6):2204-2220 (2015) ?Son olarak distHPL yukaridaki iki yayin için geliştirilen kodları kullanarak ve üzerine insa ederek geliştirilmiştir. Ve ümitimiz bunuda bir dergi yayini yapmakti. ?distHPL: An Extension of HPL Programming Model for the Distributed GPU Clusters? başlıklı çalışmamızıda bir dergi yayını yapmaya çalişiyoruz. Ekdeki draft halini örnekler ile zenginleştirerek bir dergiye yollanaçaktir. Geliştirilen distHPL kütüphanesiyle yazilan program benzer makinada MPI ile yazilandan 10 kat daha hızlı çalişmakda ve çok daha kisa sürede distHPl programı geliştirilmektedir. Tabi MPI GPU kullandırmıyor. distHPL GPU kullanılıyor.
Anahtar Kelimeler:	Heterogeneous Programming, Parallel Programming Languages, GPU Programming
Fikri Ürün Bildirim Formu Sunuldu Mu?:	Hayır

TÜBİTAK

Projeden Yapılan Yayınlar:	1- Exploiting heterogeneous parallelism with the Heterogeneous Programming Library (Makale - Diğer Hakemli Makale), 2- Heterogeneous Programming Library: A Framework for Quick Development of Heterogeneous Applications. (Bildiri - Uluslararası Bildiri - Poster Sunum), 3- Exploting multi-GPU systems using the Heterogeneous Programming Library (Bildiri - Uluslararası Bildiri - Sözlü Sunum), 4- Heterogeneous Programming Library: A Framework for Facilitating the Exploitation of Heterogeneous Systems (Bildiri - Uluslararası Bildiri - Sözlü Sunum),

TÜBİTAK