BİR GRAFİK İŞLEMCİ (GPU) İÇİN MPI (MESSAGE PASSING INTERFACE) YAZILIM KİTAPLIĞININ TASARIMI VE GERÇEKLEŞTİRİMİ

A DESIGN AND IMPLEMENTATION OF MPI (MESSAGE PASSING INTERFACE) LIBRARY FOR A GRAPHICAL PROCESSING UNIT (GPU)

İBRAHİM TANRIVERDİ

Hacettepe Üniversitesi
Lisansüstü Eğitim-Öğretim ve Sınav Yönetmeliğinin
BİLGİSAYAR Mühendisliği Anabilim Dalı İçin Öngördüğü
YÜKSEK LİSANS TEZİ
olarak hazırlanmıştır.

2010

Fen Bilimleri Enstitüsü Müdürlüğü'ne,

Bu çalışma jürimiz tarafından **BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI**'nda **YÜKSEK LİSANS TEZİ** olarak kabul edilmiştir.

Başkan :.....
Doç.Dr. Ali Ziya ALKAR

Üye (Danışman) :.....
Yrd.Doç.Dr. Kayhan İMRE

Üye :.....
Yrd.Doç.Dr. Harun ARTUNER

Üye :.....
Yrd.Doç.Dr. Mustafa EGE

Üye :.....
Dr. Ahmet Burak CAN

ONAY

Bu tez ...../...../..... tarihinde Enstitü Yönetim Kurulunca kabul edilmiştir.

...../...../.....

Prof.Dr. Adil DENİZLİ
Fen Bilimleri Enstitüsü Müdürü

Ana'ma ve Baba'ma ...

BİR GRAFİK İŞLEMCİ (GPU) İÇİN MPI (MESSAGE PASSING INTERFACE) YAZILIM KİTAPLIĞININ TASARIMI VE GERÇEKLEŞTİRİMİ

İbrahim TANRIVERDİ

ÖZ

MPI, koşut programlama için tasarlanmış ileti geçirme arayüzü belirtimidir. Taşınabilir, kolay uygulanabilir, hızlı ve esnek koşut uygulamalar geliştirmek için tanımlanmıştır. Birçok sistem sağlayıcı, bilim insanı ve uygulama geliştirici tarafından desteklenen ve bilgisayar bilimi öğrencileri tarafından bilinmesi gereken önemli bir standarttır.

Uygulama geliştirmek için koşut bilgisayar sistemlerinin oluşturulması ve bakımı yüksek maliyet gerektirir. Küçük bir aygıt olan ekran kartında ise grafik işleme için özelleşmiş, yüksek yetenekleri olan birçok grafik işlemci (GPU) bulunmaktadır. Koşut olarak çalışan bu işlemciler, yakın zamanda genel amaçlı uygulama geliştirmeye uygun bir ortama dönüşmüştür. Ayrıca, ekran kartları çoğu kişisel bilgisayardan daha yüksek veri işleme ve belleğe erişim başarımına sahiptir.

Tez çalışmasında, grafik işlemciler için MPI yazılım kitaplığının en çok kullanılan yordamlarının gerçekleştirimi yapılmıştır. Geliştirilen kitaplığın başarımı, bazı koşut uygulamalar ve MPI algoritmaları ile test edilmiştir.

Anahtar Kelimeler: MPI, CUDA, Koşut Programlama, MPI Gerçekleştirimi

Danışman: Yrd. Doç. Dr. Kayhan İMRE, Hacettepe Üniversitesi, Bilgisayar Mühendisliği Bölümü

A DESIGN AND IMPLEMENTATION OF MPI (MESSAGE PASSING INTERFACE) LIBRARY FOR A GRAPHICAL PROCESSING UNIT (GPU)

İbrahim TANRIVERDİ

ABSTRACT

MPI is a message-passing interface specification designed for parallel programming. It has been defined to develop portable, practical, efficient and flexible parallel application. MPI, the important standard, is supported by parallel computing vendors, computer scientists and application developers.

To constitute parallel computer systems to develop application and its maintenance is expensive. However, graphics card which is a minor device has many graphical processing units (GPU) which are specialized for intensive and highly parallel graphic computation. These parallel processors could be used to develop general purpose applications. Besides, graphics cards have better compute and memory access performance than many personal computers.

In this thesis, most used MPI library functions are implemented for a graphical processing unit and, performance of the implemented library are tested with some parallel applications and MPI algorithms.

Keywords: MPI, CUDA, Parallel Programming, MPI Implementation

Advisor: Asst. Professor Kayhan İMRE, Hacettepe University, Department of Computer Engineering

TEŞEKKÜR

Tez konusunun belirlenmesini sağlayan, tez çalışmasının hazırlanmasında ve tez metninin yazılmasında yardımcı olan Sayın Yrd.Doç.Dr. Kayhan İmre'ye;

Tez metnini inceleyerek biçim ve içerik bakımından son halini almasına yardım eden değerli hocalarıma;

Tez çalışmasının ortaya konması için gerekli kaynakları sağlayan Hacettepe Üniversitesi Bilgisayar Mühendisliği Bölümü'ne ve çalışma boyunca desteklerini benden esirgemeyen sevgili arkadaşlarıma;

Hayatım boyunca, bana her konuda destek olan anneme, babama ve ablalarıma içtenlikle teşekkür ederim.

İÇİNDEKİLER DİZİNİ

ŞEKİLLER DİZİNİ

ÇİZELGELER DİZİNİ

SİMGELER ve KISALTMALAR DİZİNİ

API	Application Programming Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GB	Gigabyte
GPU	Graphical Processing Unit
KB	Kilobyte
MIMD	Multi Instruction, Multi Data
MPI	Message Passing Interface
SIMD	Single Instruction, Multi Data
SIMT	Single Instruction Multi Thread
SM	Streaming Multiprocessor
SP	Streaming Processor
TPC	Thread Processing Cluster

1. GİRİŞ

Son yirmi yılda mikroişlemci teknolojisine çağ atlatan gelişmeler olmuştur. İşlemcilerin işlem kabiliyetleri ve hızları artmış, kapasiteleri ve veriye erişim hızları yükselmiştir. *Moore Kanunu*, donanım karmaşıklığının belli bir kuralla arttırılabileceğini, sistem başarımının her 18 ayda iki katına çıkacağını belirtmektedir. Ancak bu artış sonsuz değildir. Günümüzde bu kuralın sınırlarına gelinmiştir[1].

Donanımın fiziksel kapasitesini zorlaması, mevcut iş gücünün de yeterli olmaması koşut mimarileri önemli hale getirmiştir. Mimarideki değişimin yanı sıra ihtiyaçlar da artmıştır. Daha yoğun ve daha büyük veri kümeleri üzerinde, hızlı işlem yapabilen bilgisayarlara gereksinim duyulmaktadır. Bu gereksinimleri karşılamak için iş istasyonları, sunucular artık koşut çalışan işlemcilerle oluşturulmaktadır. Büyük çaplı bilim ve mühendislik uygulamaları, yüzlerce işlemcilerden oluşan sistemlerle çalıştırılmaktadır.

İhtiyaçlar doğrultusunda, koşut sistemler ile programlama yapmak, bir gereklilik haline gelmiştir. MPI (*Message Passing Interface*), koşut uygulama geliştirmek için birçok bilim insanı ve uygulama geliştirici tarafından desteklenen ve kullanılan bir koşut programlama arayüzü standartıdır. Yazılım geliştiricileri donanım sistemlerinden kaynaklanan ve iletişim sırasında oluşabilecek sorunlardan soyutlamayı sağlamaktadır. Taşınabilir, kolay uygulanabilir, hızlı ve esnek koşut uygulamalar geliştirilebilmesini amaçlamaktadır. Günümüzde, MPI yazılım kitaplığıyla geliştirilmiş birçok algoritma ve uygulama bulunmaktadır.

Teknolojideki gelişme, ekran kartlarındaki grafik işlemcilerin gelişmesini de sağlamıştır. Ekran kartlarındaki GPU(*Graphical Processing Unit*)'lar, gerçek zamanlı, üç boyutlu grafikleri işlemede, piyasalardaki artan talep nedeniyle gelişmiş; programlanabilir, koşut, çok iş parçacıklı ve çok işlemcili donanımlar haline gelmişlerdir. Genel amaçlı bilgisayarlardan farklı olarak grafik işlemciler, kayan noktalı ve çok boyutlu değerler üzerinde işlem yapmak için özelleşmiş, CPU(*Central Processing Unit*)'lara göre daha büyük verileri çok daha hızlı işleme kapasitesine erişmişlerdir. Günümüzde, yeni model aygıtlar üzerindeki yüzlerce GPU çekirdeği, genel amaçlı uygulamaları da çalıştırabilen koşut bilgisayarlara

dönüşmüştür. NVIDIA, bu amaçla uygulama geliştirirken GPU'ların genel amaçlı uygulama geliştirmek için kullanılmasına olanak sağlayan CUDA programlama modelini oluşturmuştur.CUDA (*Compute Unified Device Architecture*), koşut uygulama geliştiricilerin yabancı olmadıkları C programlama diline benzer bir tanımlama kümesine sahiptir. Bu özelliği sayesinde de programlamacılar tarafından ilgiyle karşılanmış, kısa sürede geniş destek kitlesine ulaşmıştır.

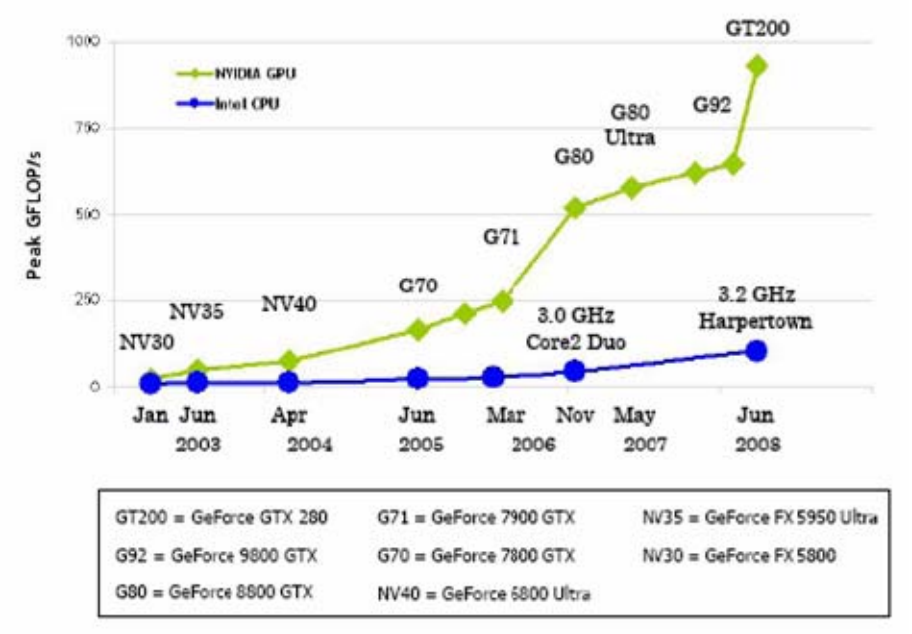
Tez çalışması kapsamında; bir koşut programlama standartı olan MPI yazılım kitaplığının, CUDA programlama modeli kullanılarak, ekran kartları üzerinde çalıştırılabilmesi sağlanmıştır. Öncelikle, koşut programlamayı yeni öğrenen programlamacılara, düşük maliyet ile çok işlemcili bir donanım ortamı sunmak, küçük bir aygıt ile bir tek bilgisayar üzerinde bir laboratuar oluşturulması amaçlanmıştır. Ayrıca, MPI gibi bir standartı bilen kişilere ek bir öğrenme yükü getirmeden, ekran kartı donanımın yüksek başarımını kullanabilmesi hedeflenmiştir. Ek olarak, CUDA modeli ile uygulama geliştiren programlamacıların, MPI algoritmalarını da kullanabilmesi ve daha önceden geliştirilmiş MPI uygulamalarını CUDA ile birleştirerek ekran kartları üzerinde çalıştırabilmeleri sağlanmıştır.

Tez metninde öncelikle NVIDIA CUDA programlama modeli ve koşut mimariler için kullanılan MPI yazılım kitaplığı hakkında bilgi verilecektir. Ekran kartları üzerinde geliştirmek için duyulan gereksinimler, tasarım ve gerçekleştirim yöntemleri açıklanacaktır. İlerleyen kesimde, geliştirilen kitaplığı kullanacak programlamacıların uyması gereken kısıtlardan bahsedilecektir. Daha sonra, önceden geliştirilmiş algoritma ve uygulamaların, tez çalışması kapsamında gerçekleştirimi yapılan MPI kitaplığı ile kullanımı ile başarımı test edilecek ve yorumlanacaktır. Son olarak, çalışmanın sonuçları, gerçekleştirim sırasında karşılaşılan sorunlar ve ileriki aşamalarda yapılabilecek olan geliştirilmelerden bahsedilecektir. Ek olarak, ekran kartı donanımın teknik özellikleri verilecektir.

2. CUDA PROGRAMLAMA MODELİ

CUDA *(Compute Unified Device Architecture*, Bütünleşik Aygıt İşleme Mimarisi); NVIDIA tarafından geliştirilen, ölçeklendirilebilir koşut programlama modelidir. Çok çekirdekli grafik işlemcileri kullanarak genel amaçlı hesaplama işlemlerinin yapılmasını sağlamak için geliştirilmiştir [5].

Ekran kartlarındaki grafik işlemciler (GPU-*Graphical Processing Unit*), gerçek zamanlı, üç boyutlu grafikleri işlemede, piyasalardaki artan talep nedeniyle gelişmiş; programlanabilir, koşut, çok iş parçacıklı ve çok işlemcili donanımlar haline gelmişlerdir. Genel amaçlı bilgisayarlardan farklı olarak GPU'lar, kayan noktalı ve çok boyutlu değerler üzerinde işlem yapmak için özelleşmiş, CPU(*Central Processing Unit*, Ana İşlem Birimi)'lara göre daha büyük verileri çok daha hızlı işleme kapasitesine sahip olmuşlardır. Kişisel bilgisayarlar için, CPU ve GPU'lar arasındaki zamana bağlı olarak sistem yeteneklerinin gelişimi Şekil 2-1'de gösterilmiştir.



Şekil 2-1. CPU ve GPU'ların zamana bağlı gelişimi[4]

Kasım 2006'da, NVIDIA; ekran kartları üzerinde, karmaşık hesaplama problemlerinin CPU'lardan daha etkili çözülebilmesi amacıyla, GPU'ların koşut işlem yeteneklerini kullanan bir genel amaçlı koşut işleme mimarisi (CUDA)

tanımlamıştır. Bu sayede, büyük yatırımlar yapma imkanı olmayan, kişisel bilgisayar kullanıcılarının da koşut uygulama geliştirmesine imkan sağlamıştır.

CUDA'nın komut kümesi ve programlama modeli geliştirilirken, öncelikle yüksek düzeyli programlama dillerinden C dilinin söz dizimi kullanılmış ve C kodları ile uyumlu çalışması sağlanmıştır. İleride FOTRAN, C++, OpenGL, DirectX gibi diğer programlama dilleri ve uygulama programlama arayüzlerinin de desteklenmesi amaçlanmıştır [4].

2.1. Programlama Modeli

CUDA ile uygulama geliştirilirken, kişisel bilgisayarlardan farklı olarak, ekran kartının bazı donanımsal özelliklerini bilmek gerekmektedir. Donanıma göre kabiliyetleri farklılık gösterebilmekte ama temel çalışma prensibi değişmemektedir. Bu kesimde, bu özellikler açıklanmaktadır.

2.1.1. Ana Sistem – Aygıt (*Host-Device*)

Uygulama, öncelikle ana sistemde (bilgisayar) çalıştırılır. İşlenecek verinin aygıta aktarımı ve GPU'ların yönetimi, CPU'nun kontrolündedir. CUDA, uygulamaların CPU ve GPU arasında karışık olarak çalıştırılmasına da olanak verir (Şekil 2-2).

2.1.2. Koşut işlev (*Kernel*)

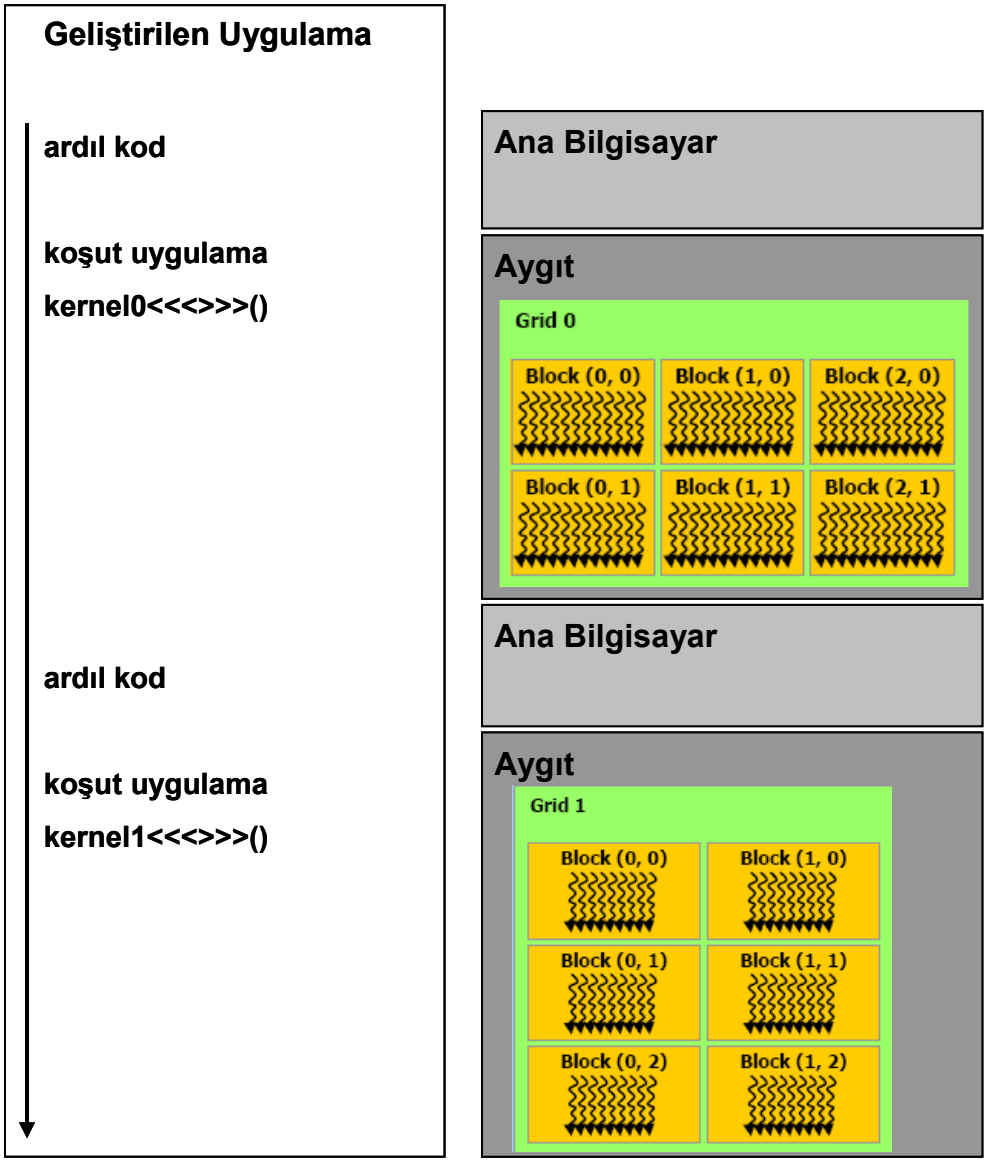
CUDA, uygulama geliştiricilerin, C dili sözdizimini kullanarak yazılım üretmesine olanak sağlamaktadır. Ana sistemde çalıştırılan uygulamanın bir bölümü, koşut işlem yapılması istendiğinde, *kernel* olarak isimlendirdikleri, özel tanımlanmış bir C işlevi ile aygıta bildirilmektedir.

Koşut işlev (*kernel*) ile, uygulamanın ilgili bölümünün, nasıl bir koşut programlama topolojisi ile çalıştırılacağı programlamacı tarafından belirtilebilmektedir. "<<< >>>" niteleyicileri kullanılarak, işlevin çalıştırılacağı *thread*, *block* ve *grid* yapıları bildirilir. Koşut işlev, C işlevleri gibi değiştirge(*parameter*) alabilir.

2.1.3. İş parçacığı – Öbek – Izgara (*Thread – Block – Grid*)

Koşut programlama için işlevler, thread adı verilen iş parçacıkları tarafından gerçekleştirilmektedir. Bir çekirdek işlevi, tek, iki veya üç boyutlu bir topolojide *thread* dizileri olarak işlenir. Bu diziler öbekleri (*block*) oluşturur. Aynı şekilde tek

veya iki boyutlu öbek dizileri, ızgara (*grid*) olarak isimlendirilen bir yapıyı meydana getirir. Bu yapılar ile koşut programlama için daha esnek bir yapı oluşturulması amaçlanmıştır (Şekil 2-2).



Şekil 2-2. CPU ve GPU üzerinde karışık programlama [4]

Koşut çalıştırılacak bir işlev için bir çok iş parçacığı ve öbek kullanılabilir. Ancak her işlev tek ızgaraya sahiptir [4].

2.1.4. Bellek çeşitleri

Ekran kartları, farklı bellek çeşitlerine sahiptir. CUDA, uygulama geliştiricilere bu bellekleri yönetme imkanı verir. Bu bellekler, aşağıdaki başlıklarda incelenmiştir:

2.1.4.1. Yerel bellek ve yazmaçlar

Yerel bellek, iş parçacıklarının kullanılması için ayrılmış bellektir. Yavaş bir bellek çeşididir. Bu sorunun aşılması için GPU'lar çok sayıda yazmaçla donatılmışlardır (Şekil 2-3). Yazmaçlar çok hızlı erişim yapılabilen bellek alanlarıdır. Ancak, her bir öbek için tanımlanmış yazmaç miktarı sabittir ve iş parçacığı sayısı arttıkça, iş başına düşen yazmaç miktarı azalır.

Bir iş parçacığına ait yerel bellekteki veya yazmaçlardaki değerlere, başka bir iş parçacığı tarafından erişim imkanı yoktur.

2.1.4.2. Paylaşımlı bellek

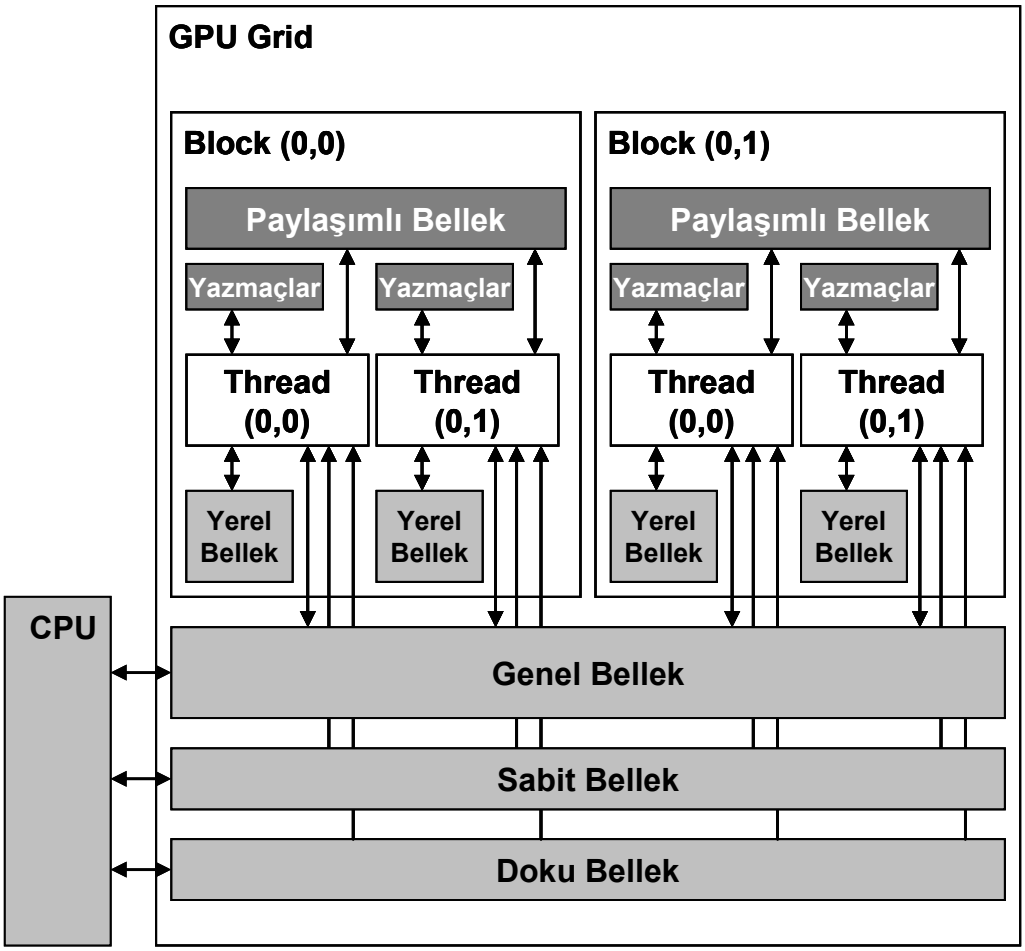
Aynı öbek içinde bulunan iş parçacıklarının birbirlerini ile veri paylaşımına olanak sağlamak için geliştirilmiştir. Yazmaçlar gibi çok hızlı erişim sağlanan belleklerdir ve büyüklüğü kısıtlıdır (Şekil 2-3).

2.1.4.3. Genel bellek

Uygulamanın başından sonuna kadar kullanılabilen, farklı ızgara, öbek ve iş parçacıkları tarafından erişilebilen, paylaşılabilen bellek çeşididir (Şekil 2-3). Büyüklüğü diğer bellek türlerine göre çok daha fazladır.

Genel belleğe, CPU tarafından da erişim imkanı vardır. Koşut programlama için kullanılacak veri, öncelikle bu belleğe iletilir. CPU ve GPU arasındaki veri aktarımı, genel bellek ile sağlanır.

Genel amaçlı programlama için değil ama grafik uygulamalarında kullanılmak üzere ekran kartlarında iki çeşit bellek daha bulunmaktadır. Uygulama süresince, üzerlerinde bulunan komutun veya verinin, sadece okuma amaçlı kullanılabilmesini sağlamaktadır. Bu bellekler **sabit** (*constant*) ve **doku** (*texture*) **bellek** olarak adlandırılmaktadır.



Şekil 2-3. CUDA Bellek Modeli [7]

2.1.5. İşleme yeteneği (*Compute Capability*)

Teknolojideki gelişmelerle koşut olarak, ekran kartlarının işleme yetenekleri de değişmektedir. Ayrıca, ekran kartlarındaki çeşitlilik, donanımlar arasında farklılık yaratmaktadır. Programlamacıların bu durumdan etkilenmeden uygulama geliştirebilmelerini sağlamak için, NVIDIA tarafından, aygıtın genel özelliklerinin belirtildiği iki sayıdan oluşan bir numaralandırma sistemi oluşturulmuştur.

İlk sayı, ana mimari özelliğini göstermekte, ikinci sayı da mimarideki küçük değişiklikleri belirtmek için kullanılmaktadır. Bir aygıtta geliştirilen yazılım, aynı işleme yeteneğine sahip başka bir donanımda da kullanılabilmektedir.

İşleme Yeteneği 1.3 için belirtimler şunlardır:

- Bir öbekte en fazla 512 thread olabilir.
- Öbeğin x, y ve z boyutlarında olabilecek iş parçacığı sayıları sırasıyla 512, 512 ve 64'tür.
- Bir ızgara içinde en fazla 65535 iş parçacığı olabilir.
- Warp'taki iş parçacığı sayısı 32'dir.
- Her çoklu işlemci (*streaming multiprocessor, SM*) için yazmaç sayısı 16384'tür.
- Her çoklu işlemci için paylaşımı bellek boyu 16 KB'tır.
- Sabit belleğin büyüklüğü 64 KB'tır.
- Çoklu işlemcilerde 8 KB önbellek bulunur. Bunu sabit bellek ya da doku belleği için kullanır.
- Her çoklu işlemcinin üzerindeki etkin block sayısı en fazla 8'dir.
- Her çoklu işlemcinin üzerindeki etkin warp sayısı en fazla 32'dir.
- Her çoklu işlemcinin üzerindeki etkin iş parçacığı sayısı en fazla 1024'tür.
- En fazla 2 milyon PTX komutu çalıştırılabilir.
- Her çoklu işlemci 8 çekirdekten (*streaming processor, SP*) oluşmuştur. 32 iş parçacığının bulunduğu bir warp'ın çalıştırılması 4 saat vuruşunda gerçekleştirilir.
- Paylaşımlı bellek üzerinde 32 ve genel bellek üzerinde 64 bitlik atomik işlemler yapılabilir.
- Çift duyarlı kayan noktalı sayılar desteklenmektedir.

2.2. Programlama Arayüzü

CUDA ile uygulama geliştirmek için iki arayüz sunulmuştur;

- C programlama dili uzantısı (*C for CUDA*)
- CUDA sürücü programlama arayüzü (*CUDA driver API*)

CUDA programlama modeli, C diline uygun, bir ifade kümesi içermektedir. Bu şekilde programlamacıların koşut programlama için gerekli işlevleri tanımlaması amaçlanmıştır. Ek olarak, yazılım geliştirirken aygıt üzerindeki belleğin yönetimi, bilgisayar ile ekran kartı arasındaki veri iletişimi, birden fazla aygıtın yönetilmesi gibi gereksinimler için özel işlevler tanımlanmıştır. C dili ile programlama yapılırken bu işlevlerin bulunduğu kütüphane (*CUDA Runtime API*) kullanılabilir.

CUDA sürücü programlama arayüzü alt düzey programlama için kullanılmaktadır. Geliştirilmesi ve denetimi daha zor olmakta, ama başarımının daha yüksek olduğu ileri sürülmektedir [4].

2.3. Niteleyiciler

Geliştirilen kitaplık için kullanılan niteleyiciler hakkında kısa bilgiler aşağıda verilmeye çalışılmıştır.

2.3.1. İşlev niteleyicileri

İşlev niteleyicileri, işlevin ana bilgisayarda ya da aygıtta çalışacağını belirtmek için kullanılır.

2.3.1.1. __device__

Önüne getirilen işlevin;

- Aygıt (*GPU*) üzerinde çalışacağını,
- Sadece aygıtta işlenen yordamlar tarafından çağırılabileceğini,

belirtir.

2.3.1.2. __global__

Önüne getirilen işlevin;

- Aygıt üzerinde çalışacağını,
- Sadece bilgisayar (*CPU*) üzerinde çalışan yordamlar tarafından çağırılabileceğini,

belirtmek için kullanılır.

2.3.1.3. __host__

İşlevin;

- Bilgisayar üzerinde çalışacağını,
- Sadece bilgisayar üzerinde çalışan yordamlar tarafından çağırılabileceğini,

belirtir.

CUDA, özel niteleyici ile tanımlanmayan bütün işlevleri `__host__` niteleyicisine sahip olarak görür ve bilgisayar üzerinde çalıştırır [4]. Hem ana sistemde hem de aygıt üzerinde çalıştırılmak istenen işlevler `__host__` ve `__device__` niteleyicilerinin birlikte kullanılması ile geliştirilebilirler. Ancak bu işlevlerin birbiriyle etkileşimi bulunmayacaktır.

2.3.1.4. Kısıtlar

- Aygıt üzerinden, bilgisayarda çalıştırılmak istenen bir yordamı çağırmak mümkün değildir.
- `__device__` ve `__global__` türündeki işlevler için özyinelemeli programlama desteklenmemektedir.
- `__device__` ve `__global__` türü işlevler içinde `static` değişken tanımlanamaz.
- `__device__` ve `__global__` türü işlevlerin kullanacağı değişkenlerle ilgili bilgiler derleme anında belli olmalıdır. Dizi boyları, değişken sayıları çalışma anında belirlenemez.
- `__device__` türü işlevlerin adresleri, `__global__` türü işlevlerde gösterge olarak kullanılamaz. Tersi durum ise mümkündür.

- `__global__` ve `__host__` niteleyicileri beraber kullanılamaz.
- `__global__` işlevlerin çağırılma şekli, C'de yordam çağırma yönteminden biraz farklıdır. CUDA çağırılacak işlevin, işleneceği ızgara ve öbek yapılarını; öbek ve iş parçacığı sayılarını tanımlamayabileceği özel bir niteleyici kullanmaktadır.
- `__global__` işlevler `void` tipinde geri dönüş yaparlar.
- `__global__` işlevler, zamanuyumsuzdur. Aygıttaki hesaplama tamamlanmadan bilgisayara geri dönüş bildirir.
- `extern` işlevler desteklenmemektedir.

2.3.2. Değişken niteleyicileri

Değişken niteleyicileri, değişkenlerin aygıt üzerinde hangi çeşit bellek alanında olacağını belirtmek için kullanılır.

Bir yordam içinde olmayan ve özel niteleyici ile belirtilmeyen değişkenler bilgisayar belleğinde tutulur.

2.3.2.1. __device__

`__device__` niteleyicisi, değişkenin aygıt üzerinde olacağını belirtir. Diğer niteleyiciler ile beraber kullanılabilir. Eğer ikinci bir niteleyici ile beraber kullanılmamışsa, değişkenin;

- Genel bellekte tutulacağını,
- Uygulama süresince erişilebilir olacağını,
- Bütün iş parçacıkları ile CPU üzerinde çalışan ve *CUDA Runtime API* kullanan işlevler tarafından erişilebilir olacağını,

belirtir.

2.3.2.2. __shared__

Değişkenin;

- Paylaşımlı bellek üzerinde tutulacağını belirtir.
- Öbeğin çalışması süresince erişilebilirdir. Öbekteki iş parçacıkları, görevlerini tamamladıktan sonra tanımsız olacaktır.
- Aynı öbek içindeki iş parçaları tarafından erişilebilirdir.

2.3.2.3. __constant__

Değişkenin;

- Sabit bellek üzerinde tutulacağını,
- Uygulama süresince erişilebilir olacağını,
- Bütün iş parçacıkları ile CPU üzerinde çalışan ve *CUDA Runtime API* kullanan işlevler tarafından erişilebilir olacağını,

belirtir.

2.3.2.4. volatile

Bellek üzerindeki bir veriye erişimde, okuma işlemi yazma işleminden hızlıdır. Genel veya paylaşımlı bellekte bulunan bir değerin üzerinde bir iş parçacığı tarafından değişiklik yapıldığında, diğer iş parçacıkları tarafından bu değişikliğin görünmesi uygulama içinde genelde mümkün olamamaktadır. `volatile` niteleyicisi bunu sağlamak için geliştirilmiştir.

2.3.2.5. Kısıtlar

- Yukarda belirtilen özel niteleyicilerle beraber `struct` ve `union` gibi yapılar kullanılamaz.
- `__shared__` niteleyicisine sahip değişkenlere, tanımlama anında ilk değer ataması yapılamaz.
- Aygıt üzerinde çalışacak şekilde yapılan kodlama içindeki, özel niteleyicisi olmayan değişken tanımlamalarında, değişkenler genellikle yazmaçlar üzerinde tutulur. Buna karşın, bazı durumlarda, derleyici tarafından başarım seviyesini yükseltmek için yerel bellekte oluşturulur.

- Aygıt üzerinde göstergelerin kullanılması, işaret ettiği adresin paylaşımlı bellekte ya da genel bellekte olduğu bilgisinin derleyici tarafından çözümlenebilmesine bağlıdır. Çözümlenemeyen göstergeler, genel bellekteki bir adresi gösteriyor gibi işlenir.
- Paylaşımlı belleği veya genel belleği gösteren bir gösterge ile ana bilgisayar üzerinde işlem yapmaya çalışmak anlamsız sonuçlar üretilmesine neden olacaktır. Büyük bir ihtimalle de uygulamayı sonlandıracaktır. [4]

2.4. Önceden Tanımlanmış Özel Değişkenler

CUDA'da, aygıt üzerinde çalışan iş parçacıklarının, çalışma anında içinde bulunduğu ızgara ve öbek boyutlarının, öbek ve iş parçacığı değerlerin bilgilerini alabilmeleri için tanımlanmış değişkenler bulunmaktadır.

gridDim

Izgaranın boyut bilgilerini belirtmek için kullanılır. Üç boyutlu bir vektördür. Boyut değerleri tamsayı değerlerdir (gridDim.x, gridDim.y ve gridDim.z). Her boyutta kaç adet öbek olduğu bilgisini tutar.

blockDim

Öbeğin hangi boyutunda, kaç adet iş parçacığı bulunduğu bilgisini tutar. *gridDim* gibi üç boyutlu bir vektördür.

blockIdx

Öbeğin ızgara içindeki yerini gösterir. *blockIdx* de üç boyut bir vektördür.

threadIdx

İş parçacığının, öbek içindeki yerinin tutan üç boyutlu bir vektördür.

warpSize

Donanımın tek seferde işlediği *warp*'ın büyüklüğünü belirtmek için tanımlanmıştır. Çok iş parçacıklı uygulamalar için gerek duyulmaktadır.

2.5. Zaman Uyumlama İşlevi

Aynı öbek içindeki iş parçacıkları arasında zaman uyumlama yapmak için kullanılan işlevdir.

```
_syncthreads()
```

Zaman uyumlama işlevinin bulunduğu noktada, o öbekteki bütün iş parçacıkları gelene kadar beklenileceği; paylaşımlı belleğe ve genel belleğe yapılan erişimlerin tamamlanacağı garanti edilir.

2.6. Atomik İşlevler

Koşut programlama yapılırken, ortak bellek alanını kullanmak bellekteki değerlerin kararlılığını sağlama konusunda sıkıntılara neden olabilmektedir. CUDA, atomik işlevlerin bulunduğu kütüphane ile, bir bellek alanı üzerinde, okuma-günleme-yazma işlemlerinin bir zaman anında tek bir iş parçacığı tarafından yapılacağı güvencesini vermektedir. Bu kütüphane ve işlevler hakkında daha geniş bilgi CUDA'nın resmi sitesinden bulunabilir [6][7].

3. MPI YAZILIM KİTAPLIĞININ CUDA İLE TASARIMI VE GERÇEKLEŞTİRİMİ

3.1. MPI

MPI (*Message-Passing Interface*), koşut programlama için tasarlanmış, ileti geçişli kütüphane arayüzü belirtimidir. MPI ile koşut bilgisayarlar için bir programlama modeli oluşturulmuştur. Bu model, ortak uygulama yapan işler arasında, bir işin adres evrenindeki verinin diğer bir işe taşınması ve işlenmesi olarak tanımlanır.

MPI, birçok sistem sağlayıcı, bilim insanı ve uygulama geliştirici tarafından desteklenmektedir. Son olarak Eylül 2009'da açıklanan MPI_2.2 sürümü bulunmaktadır. C, C++ ve Fortran dilleri için yazılım kitaplığı tanımlanmıştır.

MPI'ın esas amacı, taşınabilir, kolay uygulanabilir, hızlı ve esnek ileti geçişli uygulamalar geliştirilebilmesinin sağlanmasıdır. Koşut sistemlerden kaynaklanan iletişim sırasındaki hatalar, gecikmeler gibi sorunlardan yalıtılmış, farklı ortamlarda yazılım üretilmesine olanak sağlanması hedeflenmiştir.

MPI kütüphanesi 250'den fazla yordam içermektedir, Ancak, *minimal set* olarak adlandırılan, 6 yordam ile tamamen işlevsel ileti geçişli uygulama geliştirmek mümkündür. Bu yordamlar, MPI kütüphanesini kullanırken koşut işlenecek bölümü başlatmak ve sonlandırmak, koşut işleme ortamı hakkında bilgi alabilmek ve ileti göndermek-almak için kullanılan yordamlardır [1] (Çizelge 3-1).

Çizelge 3-1. Koşut programlama için gerekli temel MPI yordamları

`MPI_Init`	Uygulamayı başlatmak için
`MPI_Finalize`	Uygulamayı sonlandırmak için
`MPI_Comm_size`	Çalışan görev sayısı bilgisi için
`MPI_Comm_rank`	Görevin etiket bilgisi için
`MPI_Send`	İleti göndermek için
`MPI_Recv`	İleti almak için

Tez çalışması kapsamında, daha önce geliştirilmiş algoritmalar ve uygulamalar içinde en çok kullanılan MPI yordamları seçilmiş, 40 yordamın gerçekleştirimi yapılmış ve ekran kartı üzerinde çalıştırılabilmesi sağlanmıştır.

3.2. MPI Yazılım Kitaplığının Çalışma Ortamı Tasarımı

MPI'ın, uygulama geliştiriciye sağladığı yararlardan biri donanım mimarisiyle geliştirilecek yazılımı birbirinden soyutlayabilmesidir. Uygulama geliştiriciye dağıtılmış bellekli bir mimari ve tek komut/tek veri işleyen bir bilgisayar ortamı sunarak kodlama karmaşıklığını azaltır. MPI yazılım kitaplığını geliştirmek için ise donanımın özelliklerini bilmek ve ona uygun kodlama geliştirmek gerekmektedir.

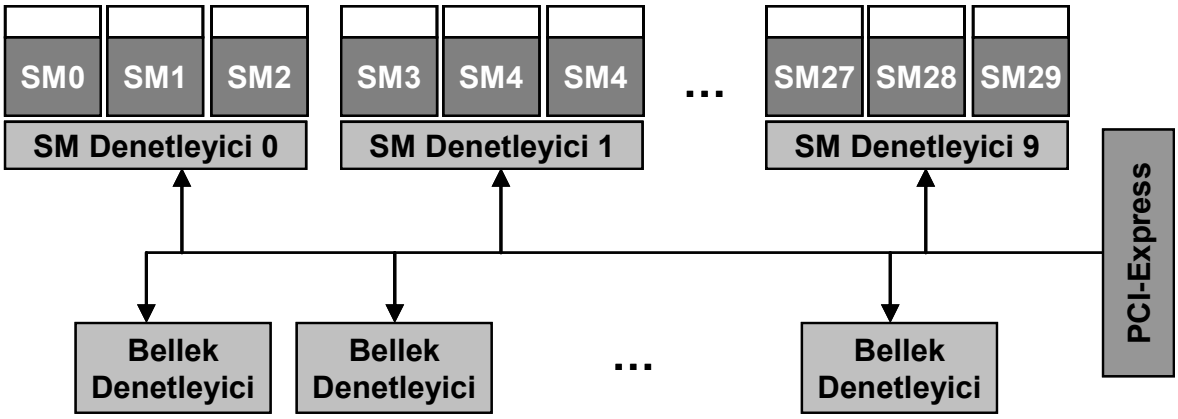
Ekran kartları, koşut programlamaya uygun donanıma sahiptirler. Ancak alışılagelmiş koşut bilgisayar mimarilerinden farklılıkları bulunmaktadır. Hem dağıtılmış hem de paylaşımlı belleğe sahiptirler. Çok komut/çok veri (MIMD, *Multi Instruction / Multi Data*) işleyebilme yetenekleri bulunmaktadır. Sahip olduğu işlemciler de kendi içinde koşut olarak çalışan küçük işlemcilerden oluşur. NVIDIA, bu mimariyi SIMT (*Single Instruction, Multiple Thread*, tek komut, çok işlem dizisi) olarak isimlendirmiştir[4]. NVIDIA tarafından, kendi ürettiği ekran kartları için, genel amaçlı koşut uygulama geliştirmeyi sağlayan, bir programlama modeli geliştirilmiştir. Ekran kartları üzerinde programla yapabilmek için bu model ve aygıt donanım özellikleri incelenmelidir.

3.2.1. Sistem ağı

Tez çalışmasında kullanılan, NVIDIA GeForce GTX 280 ekran kartı üzerinde, *streaming multiprocessor* (SM) olarak isimlendirilen, 30 adet grafik işlemci bulunmaktadır. Bu işlemcilerin kendi özel bellekleri vardır ve diğer işlemcilerden bağımsız olarak çalışabilmektedir. Ayrıca, işlemcilerin ortak olarak kullanabilecekleri, aygıt üzerinde bulunan, ayrı bir bellek daha vardır (Şekil 3-1). Bilgisayardan gelen veri ve komutların tutulduğu bu bellek, genel bellek olarak adlandırılır. Kapasitesi 1 GB'tı bulmaktadır.

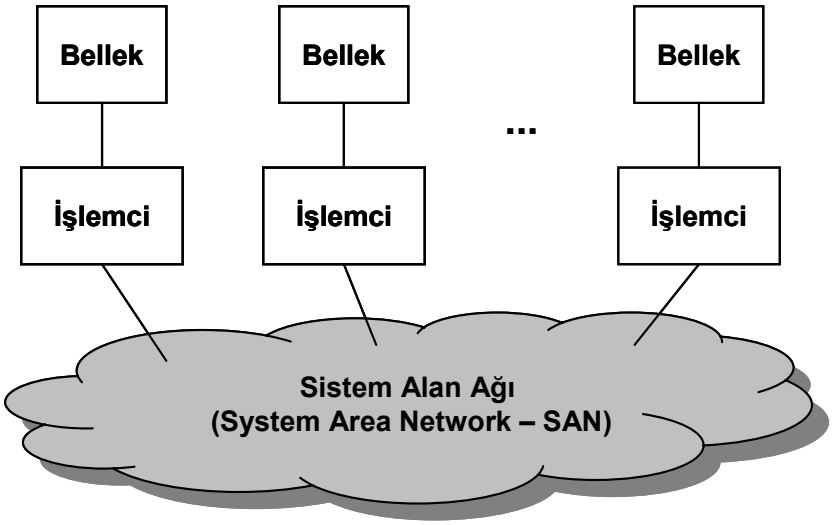
Uygulama geliştiricilerin işlemciler üzerinde kontrolünü CUDA sağlamaktadır. Programlamacı, koşut olarak çalıştırılmasını istediği bir görevi öbeklere (block) ayırır. CUDA derleyicisi gerekli kodu oluşturarak her öbeğin bir işlemci tarafından çalıştırılmasını sağlar. Kullanılabilecek öbek sayısı, ekran kartı üzerindeki işlemci

sayısından fazla olabilir[1]. Öbeklerin işlemcilere nasıl dağıtılacağı ekran kartının yetkisindedir.



Şekil 3-1. GeForce GTX 280 Ekran Kartı Mimarisi

Koşut programlama için kullanılan ekran kartının bu mimarisi, MPI yazılım kitaplığında kullanılacak yordamlar için uygun bir ortam oluşturmaktadır. Tez çalışmasında, dağıtılmış bellekli MIMD bilgisayar mimarisi bu sistem üzerine uygulanmaya çalışılmıştır (Şekil 3-2).



Şekil 3-2. Dağıtılmış Bellekli MIMD Bilgisayar Mimarisi

Ekran kartı üzerindeki genel bellek, sistem ağı olarak kullanılmış; grafik işlemciler de dağıtılmış bellekli MIMD bilgisayar sistemini oluşturan işlemciler olarak düşünülmüştür. Bu sistem üzerinde çalışacak olan, kullanıcının önceden belirteceği öbekler de MPI görevleri olarak tasarlanmıştır.

[1] NVIDIA GeForce GTX 280 ekran kartı üzerindeki GPU için en fazla 240 öbek (*block*) tanımlanabilir.

3.2.2. Görevler arası iletişim

Görevlerin birbirleriyle iletişimi genel bellekte, her öbek için ayrılan özel yastık alanları üzerinden yapılması sağlanmıştır. Bunun için, iletinin tutulduğu bir dizi, iletişimin sağlıklı yapılabilmesi için gerekli kontrol değişkenleri ile sistem hataları veya bilgi mesajları için kullanılan karakter dizilerinden oluşan bir veri yapısı kullanılmıştır.

İleti aktarımı sırasında, bir görevin yastık alanı üzerinde aynı anda birden çok görev tarafından değişiklik yapılmasını engellemek için bir kilitleme yöntemi geliştirilmiştir. Geliştirilen kilitlenme tipleri ve kuralları şunlardır:

Kendi alanını kilitleme: `kilitle(`*`yastık alan`*`)`

Yastık alanı, kendisi ya da başka bir görev tarafından kullanılacak her görev, bu alanı kilitleyerek gerekli kontrol değerlerini günler. Diğer görevler tarafından erişilebilmesi için bellek alanının hazırlanmasında kullanılır.

Hedef alanı bekleme: `bekle(`*`yastık_alan, beklenen_işlev`*`)`

Farklı görevin yastık alanına erişecek her görev, hedef alanın hazır olmasını bekler. Geliştirilen MPI yordamları, hedef görevlerin yastık alanlarını kullanmak için kendi işlev tipine göre bir bildirim bekler. Eğer ileti, hedef görev tarafından bekleniyorsa ileti geçişi sağlanır.

Zaman Uyumlama: `sync()`

Toplu iletişim yordamları için, görevlerin zaman uyumu yapmaları gerekmektedir. Bunun için görevlerin, birbirlerini beklemesini sağlamak amacıyla kullanılan üçüncü kilitlenme tipidir.

MPI yordamlarının geliştirimi yukarda anlatılan tasarım modeline uyularak sağlanmıştır. Tez metninin ilerleyen bölümünde geliştirilen yordamlar hakkında bilgi verilecektir. Diğer yordamların gerçekleştirimine temel oluşturan bazı yordamların çalışmasının ekran kartı üzerinde nasıl modellendiği anlatılacaktır.

3.3. MPI Yazılım Kitaplığının Gerçekleştirimi

Gerçekleştirimi yapılan yordamlar MPI'ın C dili söz dizimi için koyduğu kurallara uygun olarak, CUDA programlama modeli kullanılarak gerçekleştirilmiştir. Bu yordamlar 4 ana başlık altında toplanmıştır.

3.3.1. Görevden göreve iletişim

Görevden göreve iletişim yordamları, MPI iletişim düzeneğinin temelini oluşturur. İletişim, iki görev arasında iletinin geçirilmesidir. Bütün ileti geçirme yordamları da iletilerini birbirlerine aktararak çalışırlar. İleti aktarımını, zaman uyumlu ve zaman uyumsuz olarak yapmak mümkündür.

Tez kapsamında, MPI görevden göreve iletişim yordamlarının gerçekleştirimi hedeflenmiş; zaman uyumlu iletişim yordamlarının gerçekleştirimi tamamlanmıştır. İki görev arasında ileti geçirmek için kullanılan MPI yordamları şunlardır:

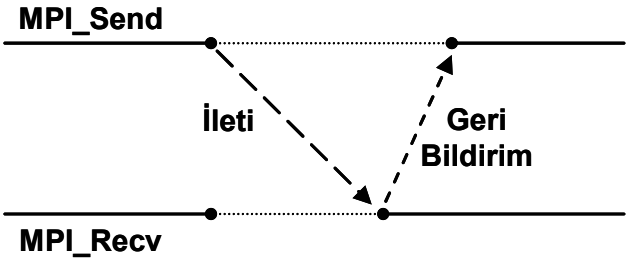
MPI_SEND	MPI_RECV
MPI_SENDRECV	MPI_SENDRECV_REPLACE
MPI_GET_COUNT	

Zaman uyumuz görevden göreve iletişim yordamları donanım ve yazılım modelindeki kısıtlardan dolayı gerçekleştirilememiştir.

İki görev arasında ileti geçirmek için iletiyi alacak olan görevin yastık alanı kullanılır. İletiyi gönderecek olan görev hedef görevin yastık alanına sahip olduğu veriyi kaydeder. Alıcı görev de kendi bellek alanından bu değeri okur. Bu işlemler, kendi alanını kilitleme ve hedef alanı bekleme tipi kilitlenmeler kullanılarak yapılır.

MPI_SEND – MPI_RECV

MPI_SEND yordamı; bir görevin, kendi belleğinde bulunan bir iletiyi başka bir yordama göndermesini sağlar. MPI_RECV yordamı; gönderilen iletiyi almak için kullanılır. Bu iki yordam karşılıklı olarak, birbirini bekler şekilde çalışır (Şekil 3-3). Koşut programlama için temel oluşturan yordamlar arasındadırlar.



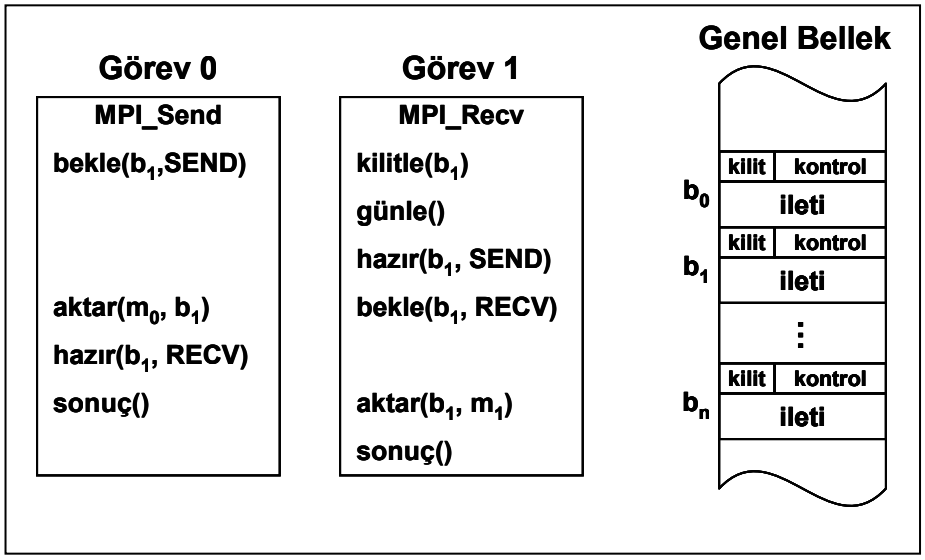
Şekil 3-3. Zaman uyumlu mesajlaşma

Bu iki yordamları çalışması şu şekilde izah edilebilir (Şekil 3-4):

- İletiyi alacak olan yordam MPI_RECV yordamıdır. MPI_RECV yordamı, kendini çağıran görevin yastık alanını kilitler. Beklenen ileti ve görevle ilgili bilgileri günler. İletinin gönderilmesi bekler.

- MPI_SEND yordamını çağıran görev, hedef yastık alanının hazır olmasını bekler. Hedef alan hazır olduğunda gerekli kontrolleri yapar ve iletiyi aktarır. Aktarım işlemi tamamlandığında, MPI_SEND yordamı, bu sonucu hedef göreve bildirir.

- Beklemede olan görev, iletiyi yastık alandan kendi bellek alanına kaydeder. Böylece süreç tamamlanmış olur.



Şekil 3-4. MPI_SEND – MPI_RECV yordamların çalışması

İleti gönderen yordamın, hedef görevi ve iletisinin imi bellidir. Ancak, ileti bekleyen görevler için bu durum her zaman mümkün olmamaktadır. Görevin herhangi bir görevden gelen bir iletiyi de kabul etmesi gerekebilir. Bu nedenle, özel (*wildcard*) değerler kullanılır. Herhangi bir görevden ileti almak için MPI_ANY_SOURCE, herhangi bir ime sahip iletiyi almak için MPI_ANY_TAG tanımlayıcıları bulunmaktadır.

MPI_RECV yordamı çağırılırken, diğer değiştirgelere ek olarak MPI_STATUS veri yapısında bir değişken daha kullanılır. MPI_RECV, bu değişkeni, gelen iletinin kaynak, im ve uzunluk bilgileriyle günler.

Diğer görevden göreve iletişim yordamları da bu iki yordamın birleştirilmesi ile gerçekleştirilmiştir.

MPI Yordamları

- MPI_SEND (buf, count, datatype, dest, tag, comm)

girdi	buf	gönderilecek iletinin adresi
girdi	count	iletinin uzunluğu
girdi	datatype	iletinin veri türü
girdi	dest	iletinin gönderileceği hedef görevin kodu
girdi	tag	iletinin imi
girdi	comm	görevin bağlı olduğu ağ (*communicator*)

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int
             dest, int tag, MPI_Comm comm)
```

- MPI_RECV(buf, count, datatype, source, tag, comm, status)

girdi	buf	alınacak iletinin kaydedileceği adres
girdi	count	iletinin uzunluğu
girdi	datatype	iletinin veri türü
girdi	source	iletinin alınacağı kaynak görevin kodu
girdi	tag	iletinin imi
girdi	comm	görevin bağlı olduğu ağ (*communicator*)
çıktı	status	ileti geçişiyle ilgili durum bilgisi

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int
             source, int tag, MPI_Comm comm, MPI_Status *status)
```

- MPI_SENDRECV (sendbuf, sencount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

girdi	sendbuf	gönderilecek iletinin adresi
girdi	sendcount	gönderilecek iletinin uzunluğu
girdi	sendtype	gönderilecek iletinin veri türü
girdi	dest	iletinin gönderileceği hedef görevin kodu
girdi	sendtag	gönderilecek iletinin imi
çıktı	recvbuf	alınacak iletinin kaydedileceği adres
girdi	recvcount	alınacak iletinin uzunluğu
girdi	recvtype	alınacak iletinin veri türü
girdi	source	iletinin alınacağı kaynak görevin kodu
girdi	recvtag	alınacak iletinin imi
girdi	comm	görevin bağlı olduğu ağ (*communicator*)
çıktı	status	ileti geçişiyle ilgili durum bilgisi

```
int MPI_Sendrecv (void *sendbuf, int sendcount, MPI_Datatype
          sendtype, int dest, int sendtag, void *recvbuf, int
          recvcount, MPI_Datatype recvtype, int source, int
          recvtag, MPI_Comm comm, MPI_Status *status)
```

MPI_SENDRECV yordamı, tek bir işlemde bir iletiyi bir göreve göndermek, bir başka iletiyi de herhangi bir görevden, aynı iletişim ağında olmak koşuluyla, almak için kullanılır. Hem MPI_SEND hem de MPI_RECV yordamlarının özelliklerini içerir.

- MPI_SENDRECV_REPLACE (buf, count, datatype, dest, sendtag, source, recvtag, comm, status)

Girdiçıktı	buf	iletinin bulunacağı adres
girdi	count	iletinin uzunluğu
girdi	datatype	iletinin veri türü
girdi	dest	iletinin gönderileceği hedef görevin kodu
girdi	sendtag	gönderilecek iletinin imi
girdi	source	iletinin alınacağı kaynak görevin kodu
girdi	recvtag	alınacak iletinin imi
girdi	comm	görevin bağlı olduğu ağ (*communicator*)
çıktı	status	ileti geçişiyle ilgili durum bilgisi

```
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype
          datatype, int dest, int sendtag, int source, int
          recvtag, MPI_Comm comm, MPI_Status *status)
```

MPI_SEDRECV_REPLACE yordamının kullanım amacı da, tek bir işlemde bir iletiyi bir göreve göndermek, bir başka iletiyi de herhangi bir görevden almaktır. MPI_SEDRECV yordamından farkı değiştirge sayısını azaltılmasıdır. Yordamı çağıran görevde, gönderilecek iletinin bulunduğu bellek alanı alınacak ileti için de kullanılır.

- MPI_GET_COUNT (status, datatype, count)

girdi	status	ileti geçişiyle ilgili durum bilgisi
girdi	datatype	iletinin veri türü
girdi	count	alinan iletinin uzunluğu

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,
          int *count)
```

Alınan bir iletinin uzunluğunu öğrenmek için kullanılan yordamdır. Sorgulanan veriler, verilen veri yapısı içinde tutulan değerler kullanılarak hesaplanır. Veri yapısı, iletiyi alan yordam tarafından günlenmiştir.

3.3.2. Toplu iletişim

MPI toplu iletişim yordamları, bir grup görevin kendi aralarında iletişimini sağlamak için tanımlanmıştır. Aynı grup içindeki bütün yordamlar tarafından çağırılır ve kullanılır.

MPI toplu işlem yordamları şu şekilde sınıflandırılır[2]:

a. Tek Görevden Bütün Görevlere: İletişim, bir görevdeki veri için yapılır. İletiler bütün görevlere iletilir.
   i. MPI_BCAST
   ii. MPI_SCATTER, MPI_SCATTERV
b. Bütün Görevlerden Tek Göreve: İşlemler her görev için uygulanır. Sonuç tek bir görevde toplanır.
   i. MPI_GATHER, MPI_GATHERV
   ii. MPI_REDUCE
c. Bütün Görevlerden Bütün Görevlere: Toplu işlemler her görev için uygulanır. Sonuç, bütün görevlere iletilir.
   i. MPI_ALLGATHER, MPI_ALLGATHERV
   ii. MPI_ALLTOALL, MPI_ALLTOALLV, MPI_ALLTOALLW
   iii. MPI_ALLREDUCE, MPI_REDUCE_SCATTER
d. Diğer: Yukarıdaki sınıflandırmaya uymayan işlemler
   i. MPI_BARRIER
   ii. MPI_SCAN, MPI_EXSCAN

Toplu iletişim yordamları gerçekleştirilirken, iletişim ortamının tek bir bellek olması, iki görev arasındaki iletişimlerden farklı olarak bir kısıt getirmektedir. Ekran kartı üzerindeki genel bellekte, birden fazla işlemciden gelen, okuma ve yazma isteklerinin bir sıraya konması gerekir. Bu isteklerin artması da başarımı düşürür.

Tahmin edileceği gibi yazma işlemi, okuma işleminden daha fazla zaman alır. Bu nedenle, bir iletiyi birden fazla göreve göndermek için; her birinin yastık alanını kullanmak yerine, ana görevin yastık alanı kullanılmış; diğer görevlerin de bu alandan okuması sağlanmıştır.

Toplu iletişim yordamların çalışması birbirlerine benzemektedir. Bazı yordamların gerçekleştirimi, diğer yordamların kullanılması ile de sağlanabilmektedir. Ancak, bu durum genelde başarımın düşmesine neden olmaktadır. Tez çalışmasında, yordamların gerçekleştirimi için daha önceden gerçekleştirilen yordamların kullanılmamasına özen gösterilmiştir.

3.3.2.1. Tek görevden bütün görevlere

MPI_BCAST

Bir görevden, aynı grup içindeki diğer görevlere iletinin aktarılması için kullanılan yordamdır. Her göreve aktarılan ileti aynıdır.

MPI_BCAST yordamının çalışması şu şekilde açıklanabilir (Şekil 3-5):

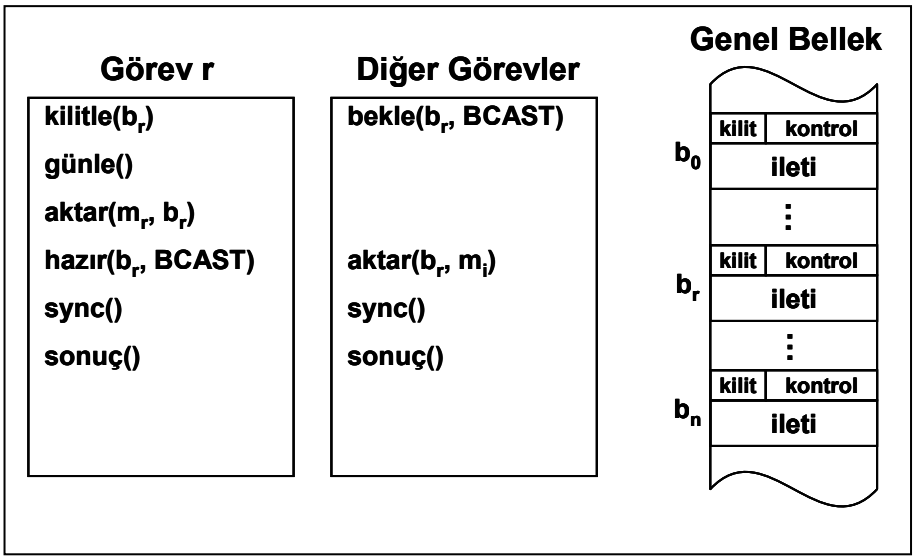
- İletiyi gönderecek olan görev, genel bellekteki kendi yastık alanını kilitler. Gerekli değerleri günler ve iletiyi kaydeder.
- Diğer görevler, gönderici görevin yastık alanı hazır oluncaya kadar bekler. Kaynak bellek alanı hazır olduğunda gerekli kontrolleri yaparak iletiyi alırlar. Aktarma işlemini tamamlayan her görev bunu ana göreve bildirir.
- Tüm görevlerin iletiyi alması ile MPI_BCAST yordamının çalışması sonlanır.

MPI Yordam Tanımları

- MPI_BCAST (buffer, count, datatype, root, comm)

girdiçıktı	buffer	gönderilecek/alınacak iletinin adresi
girdi	count	iletinin uzunluğu
girdi	datatype	iletinin veri türü
girdi	root	iletiyi gönderecek görevin kodu
girdi	comm	görevin bağlı olduğu ağ (*communicator*)

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm)
```



Şekil 3-5. MPI_BCAST yordamının çalışması

- MPI_SCATTER (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

girdi	sendbuf	gönderilecek iletilerin adresi
girdi	sendcount	gönderilecek iletilerin uzunluğu
girdi	sendtype	gönderilecek iletilerin veri türü
çıktı	recvbuf	alınacak iletinin kaydedileceği adres
girdi	recvcount	alınacak iletinin uzunluğu
girdi	recvtype	alınacak iletinin veri türü
girdi	root	iletileri gönderecek olan görevin kodu
girdi	comm	görevin bağlı olduğu ağ (*communicator*)

```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype
                sendtype, void *recvbuf, int recvcount,
                MPI_Datatype recvtype, int root, MPI_Comm comm)
```

MPI_SCATTER yordamı, tek bir görevdeki veriyi parçalayıp, aynı grup içindeki görevlere dağıtmak için, bütün görevlerdeki iletilerin eşit büyüklükte olduğu durumlarda kullanılır. İletiler, görevlerin sırasına göre aktarılır. MPI_BCAST yordamına benzer bir algoritma ile çalışır. Tek farkı görevler, genel bellekteki bütün veriyi değil, sadece kendine ait olan bölümü okur.

- MPI_SCATTERV (sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

girdi	sendbuf	gönderilecek iletilerin adresi
girdi	sendcounts	gönderilecek iletilerin uzunlukları
girdi	displs	gönderilecek iletilerin göreli adresleri
girdi	sendtype	gönderilecek iletilerin veri türü
çıktı	recvbuf	alınacak iletinin kaydedileceği adres
girdi	recvcount	alınacak iletinin uzunluğu
girdi	recvtype	alınacak iletinin veri türü
girdi	root	iletileri gönderecek olan görevin kodu
girdi	comm	görevin bağlı olduğu ağ (*communicator*)

```
int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs,
          MPI_Datatype sendtype, void *recvbuf, int recvcount,
          MPI_Datatype recvtype, int root, MPI_Comm comm)
```

MPI_SCATTER yordamıyla aynı işleve sahiptir. Ana verinin farklı uzunluklarda ve görevlerin sırasından bağımsız olarak dağıtılabilmesi için kullanılır.

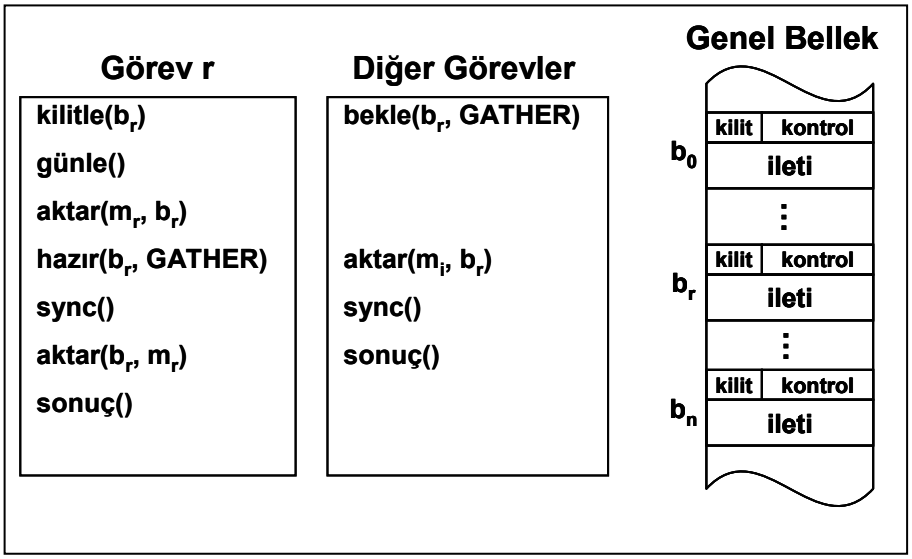
3.3.2.2. Bütün görevlerden tek göreve

MPI_GATHER

MPI_GATHER yordamı, aynı grup içindeki görevlerdeki verilerin tek bir görevde toplanmasını sağlamak için, bütün görevlerdeki iletilerin aynı büyüklükte olması durumunda kullanılır. Ayrı görevlerden gelen veri tek bir veri olarak, görevlerin sırasına göre birleştirilerek toplanır.

Yordamın çalışması Şekil 3-6'da açıklanmıştır:

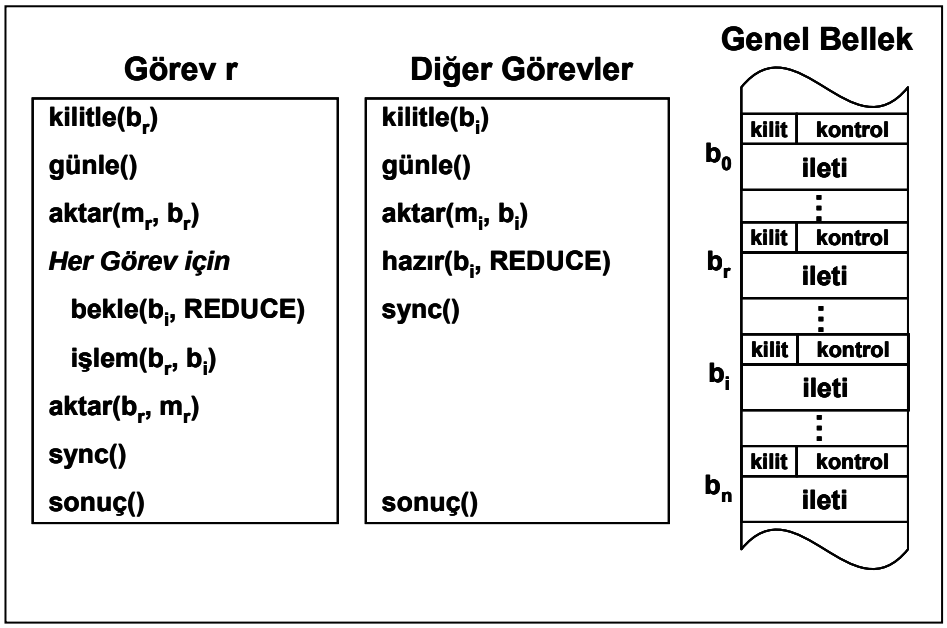
- Ana görev kendi alanını kilitler. Gerekli değerleri günleyerek diğer görevlere yastık alanın hazır olduğunu bildirir.
- Bütün görevler kendi iletilerini ana görevin yastık alanına kaydederler.
- İletinin son hali ana görevin yastık alanındadır. Ana görev, veriyi kendi bellek alanına aktarır.



Şekil 3-6. MPI_GATHER yordamının çalışması

Geliştirilen yöntem, benzer diğer yordamların geliştirimine de temel oluşturur.

MPI_REDUCE



Şekil 3-7. MPI_REDUCE yordamının çalışması

Farklı görevlere dağıtılmış veriler üzerinde matematiksel işlemlerin yapılabilmesi için kullanılan bir yordamdır. Sonuç ana göreve iletilir. Kullanılabilecek işlemler, MPI'ın önceden tanımlanmış bulunan şu işlemlerdir:

MPI_MAX (en büyüğü bulma)
MPI_MIN (en küçüğü bulma)
MPI_SUM (toplama)
MPI_PROD (çarpma)

MPI, kullanıcılara kendi işlemlerini tanımlamak için imkan sağlamaktadır. Özel işlev tanımlamaları ile kullanıcı tanımlı işlemler gerçekleştirilir. Tez çalışmasında bu yordamların geliştirilmesine çalışılmış, donanım ve yazılımdaki kısıtlar nedeniyle gerçekleştirilememiştir.

MPI_REDUCE yordamının çalışması şu şekilde açıklanabilir(Şekil 3-7):

- Her görev iletisini kendi yastık alanına kaydeder.
- Ana görev, sırayla bütün görevlerin yastık alanlarını dolaşarak uygulanması istenen işlemi gerçekleştirir.
- Sonuç, ana görevin yastık alanında bulunur. İlgili görev kendi bellek alanına bu veriyi aktarır.

MPI Yordam Tanımları

- MPI_GATHER (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

girdi	sendbuf	gönderilecek iletinin adresi
girdi	sendcount	gönderilecek iletinin uzunluğu
girdi	sendtype	gönderilecek iletinin veri türü
çıktı	recvbuf	alınacak iletilerin kaydedileceği adres
girdi	recvcount	alınacak iletilerin uzunluğu
girdi	recvtype	alınacak iletilerin veri türü
girdi	root	iletileri alacak olan görevin kodu
girdi	comm	görevin bağlı olduğu ağ (*communicator*)

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype
               sendtype, void *recvbuf, int recvcount,
               MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- MPI_REDUCE (sendbuf, recvbuf, count, datatype, op, root, comm)

girdi	sendbuf	gönderilecek iletinin adresi
çıktı	recvbuf	alınacak iletilerin kaydedileceği adres
girdi	count	iletilerin uzunluğu
girdi	datatype	iletilerin veri türü
girdi	op	iletiler üzerinde gerçekleştirilecek işlem
girdi	root	iletileri alacak olan görevin kodu
girdi	comm	görevin bağlı olduğu ağ (*communicator*)

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
       MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

- MPI_GATHERV (sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)

girdi	sendbuf	gönderilecek iletinin adresi
girdi	sendcount	gönderilecek iletinin uzunluğu
girdi	sendtype	gönderilecek iletinin veri türü
çıktı	recvbuf	alınacak iletilerin kaydedileceği adres
girdi	recvcounts	alınacak iletilerin uzunlukları
girdi	displs	iletilerin kaydedileceği göreli adresler
girdi	recvtype	alınacak iletilerin veri türü
girdi	root	iletileri alacak olan görevin kodu
girdi	comm	görevin bağlı olduğu ağ (*communicator*)

```
int MPI_Gatherv(void *sendbuf, int sendcount, MPI_Datatype
          sendtype, void *recvbuf, int *recvcounts, int *displs,
          MPI_Datatype recvtype, int root, MPI_Comm comm)
```

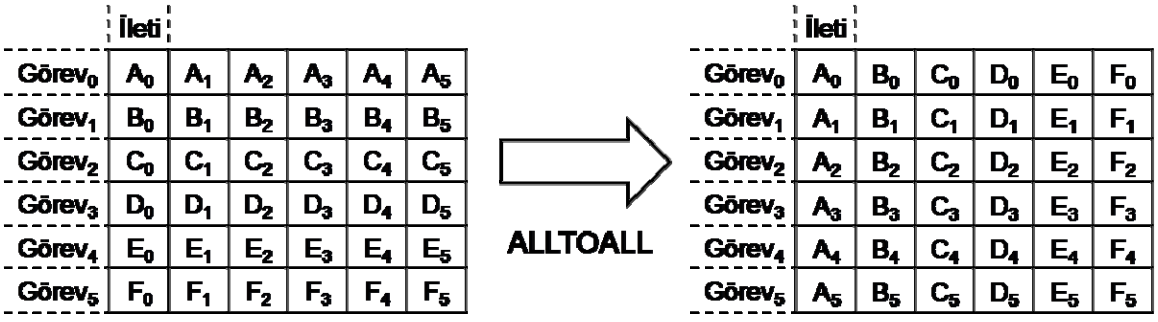
MPI_GATHER yordamıyla aynı işlevi görür. Farklı uzunluklarda olan ve görev sırasından bağımsız bir düzen içinde birleştirilmesi gereken veriler için esneklik sağlayan MPI_GATHERV yordamı kullanılır.

3.3.2.3. Bütün görevlerden bütün görevlere

Bu yordamlar, yukarıda bahsedilen geliştirme yöntemlerinin birleştirilmesiyle gerçekleştirilmiştir.

MPI_ALLTOALL

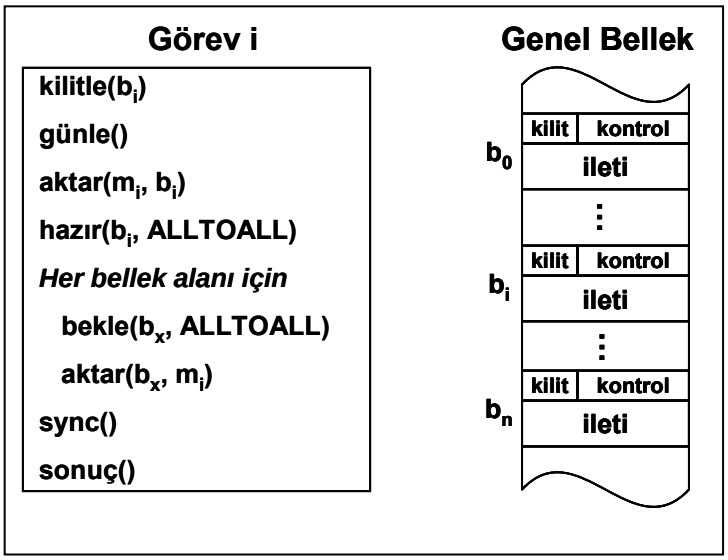
Grup içindeki bütün görevler arasında verilerin toplanması ve dağıtılması (GATHER/SCATTER) için kullanılır. Gerçekleştirimi de bu iki algoritmanın birleştirilmesi ile yapılmıştır.



Şekil 3-8. MPI_ALLTOALL işleminin gösterimi

Çalışması şu şekilde özetlenebilir:

- Her görev kendi yastık alanını kilitler ve iletisini buraya kaydeder.
- Her görev, sırayla bütün görevlerin yastık alanını dolaşarak her iletinin kendisine ait olan bölümünü alır. İleti parçaları toplanarak bütün bir iletiyi oluşturur.



Şekil 3-9. MPI_ALLTOALL yordamının çalışması

MPI Yordam Tanımları

- MPI_ALLTOALL (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

girdi	sendbuf	gönderilecek iletilerin adresi
girdi	sendcount	gönderilecek iletilerin uzunluğu
girdi	sendtype	gönderilecek iletilerin veri türü
çıktı	recvbuf	alınacak iletilerin kaydedileceği adres
girdi	recvcount	alınacak iletilerin uzunluğu
girdi	recvtype	alınacak iletilerin veri türü
girdi	comm	görevin bağlı olduğu ağ (*communicator*)

```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype
      sendtype, void *recvbuf, int recvcount, MPI_Datatype
      recvtype, MPI_Comm comm)
```

- MPI_ALLTOALLV (sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)

girdi	sendbuf	gönderilecek iletilerin adresi
girdi	sendcounts	gönderilecek iletilerin uzunlukları
girdi	sdispls	gönderilecek iletilerin göreli adresleri
girdi	sendtype	gönderilecek iletilerin veri türü
çıktı	recvbuf	alınacak iletilerin kaydedileceği adres
girdi	recvcounts	alınacak iletilerin uzunlukları
girdi	rdispls	iletilerin kaydedileceği göreli adresleri
girdi	recvtype	alınacak iletilerin veri türü
girdi	comm	görevin bağlı olduğu ağ (*communicator*)

```
int MPI_Alltoallv(void *sendbuf, int *sendcounts, int *sdispls,
        MPI_Datatype sendtype, void *recvbuf, int *recvcounts,
        int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
```

MPI_ALLTOALL yordamı ile aynı işlevi görür. Ek olarak, görevlerin gönderdiği ya da aldığı ileti boylarının eşit olmadığı gereksinimler için kullanılır.

- MPI_ALLTOALLW (sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm)

girdi	sendbuf	gönderilecek iletilerin adresi
girdi	sendcounts	gönderilecek iletilerin uzunlukları
girdi	sdispls	gönderilecek iletilerin göreli adresleri
girdi	sendtypes	gönderilecek iletilerin veri türleri
çıktı	recvbuf	alınacak iletilerin kaydedileceği adres
girdi	recvcounts	alınacak iletilerin uzunlukları
girdi	rdispls	iletilerin kaydedileceği göreli adresleri
girdi	recvtypes	alınacak iletilerin veri türleri
girdi	comm	görevin bağlı olduğu ağ (*communicator*)

```
int MPI_Alltoallw(void *sendbuf, int sendcounts[], int sdispls[],
       MPI_Datatype sendtypes[], void *recvbuf, int recvcounts[],
       int rdispls[], MPI_Datatype recvtypes[], MPI_Comm comm)
```

MPI_ALLTOALLV yordamının geliştirilmesine ihtiyaç duyulan gereksinimlere ek olarak, iletilerin farklı veri türünde olması durumunda kullanılabilecek yordamdır.

- MPI_ALLGATHER (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

girdi	sendbuf	gönderilecek iletinin adresi
girdi	sendcount	gönderilecek iletinin uzunluğu
girdi	sendtype	gönderilecek iletinin veri türü
çıktı	recvbuf	alınacak iletilerin kaydedileceği adres
girdi	recvcount	alınacak iletilerin uzunluğu
girdi	recvtype	alınacak iletilerin veri türü
girdi	comm	görevin bağlı olduğu ağ (*communicator*)

```
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype
        sendtype, void *recvbuf, int recvcount, MPI_Datatype
        recvtype, MPI_Comm comm)
```

MPI_ALLGATHER yordamının MPI_GATHER yordamından farkı, verilerin tek bir göreve değil, bütün görevlere iletilmesidir. Gerçekleştirimi MPI_BCAST ve MPI_GATHER yordamlarının kullanılmasıyla mümkündür. Tez çalışmasında, başarımı olumsuz yönde etkilememek için bu yordamların algoritmaları birleştirilerek geliştirilmiştir.

- MPI_ALLGATHERV (sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

girdi	sendbuf	gönderilecek iletinin adresi
girdi	sendcount	gönderilecek iletinin uzunluğu
girdi	sendtype	gönderilecek iletinin veri türü
çıktı	recvbuf	alınacak iletilerin kaydedileceği adres
girdi	recvcounts	alınacak iletilerin uzunlukları
girdi	displs	iletilerin kaydedileceği göreli adresleri
girdi	recvtype	alınacak iletilerin veri türü
girdi	comm	görevin bağlı olduğu ağ (*communicator*)

```
int MPI_Allgatherv(void *sendbuf, int sendcount, MPI_Datatype
          sendtype, void *recvbuf, int *recvcounts, int *displs,
          MPI_Datatype recvtype, MPI_Comm comm)
```

Farklı uzunluklarda olan ve görev sırasından bağımsız bir düzen içinde birleştirilmesi gereken verilerin iletimi için kullanılır. Gerçekleştirimi MPI_ALLGATHER yordamının gereksinimleri karşılayacak şekilde geliştirilmesiyle sağlanmıştır.

- MPI_ALLREDUCE (sendbuf, recvbuf, count, datatype, op, comm)

girdi	sendbuf	gönderilecek iletilerin adresi
çıktı	recvbuf	alınacak iletilerin kaydedileceği adres
girdi	count	iletilerin uzunluğu
girdi	datatype	iletilerin veri türü
girdi	op	iletiler üzerinde gerçekleştirilecek işlem
girdi	comm	görevin bağlı olduğu ağ (*communicator*)

```
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
           MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

İşlevi, MPI_REDUCE yordamı ile aynıdır. Üretilen sonuç bütün görevlere dağıtılır. Gerçekleştirimi, MPI_REDUCE yordamının MPI_BCAST algoritması ile geliştirilmesiyle sağlanmıştır.

- MPI_REDUCE_SCATTER (sendbuf, recvbuf, recvcounts, datatype, op, comm)

girdi	sendbuf	gönderilecek iletilerin adresi
çıktı	recvbuf	alınacak iletilerin kaydedileceği adres
girdi	recvcounts	alınacak iletilerin uzunlukları
girdi	datatype	iletilerin veri türü
girdi	op	iletiler üzerinde gerçekleştirilecek işlem
girdi	comm	görevin bağlı olduğu ağ (*communicator*)

```
 int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int
     *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

MPI_REDUCE yordamının sonucunun bütün görevler arasında bölünerek dağıtılması için kullanılır. İletilerin boylarının farklı olması da mümkündür.

MPI_REDUCE yordamının, MPI_SCATTER algoritması ile geliştirilmesiyle gerçekleştirilmiştir.

3.3.2.4. Diğer MPI yordam tanımları

- MPI_BARRIER (comm)

girdi	comm	görevin bağlı olduğu ağ (*communicator*)

```
int MPI_Barrier(MPI_Comm comm)
```

Aynı grup içindeki görevler arasında zaman uyumlaması yapmak için kullanılan engel noktasıdır. Kilitlenme yöntemleri kullanılarak gerçekleştirilmiştir. Her görev, ilk görevin bellek alanı üzerine erişim yapana kadar bütün görevler bekletilir.

- MPI_SCAN (sendbuf, recvbuf, count, datatype, op, root, comm)

girdi	sendbuf	gönderilecek iletinin adresi
çıktı	recvbuf	alınacak iletilerin kaydedileceği adres
girdi	count	iletilerin uzunluğu
girdi	datatype	iletilerin veri türü
girdi	op	iletiler üzerinde gerçekleştirilecek işlem
girdi	root	iletileri alacak olan görevin kodu
girdi	comm	görevin bağlı olduğu ağ (*communicator*)

```
int MPI_Scan(void* sendbuf, void* recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Farklı görevlerde bulunan veriler üzerinde matematiksel işlemlerin yapılması için kullanılır. Yordam, sırasıyla görevlerdeki veriler üzerinde işlem yapar ve o ana kadarki sonucu ilgili göreve iletir. MPI_REDUCE algoritmasının düzenlenmesi ile gerçekleştirilmiştir.

- MPI_EXSCAN (sendbuf, recvbuf, count, datatype, op, root, comm)

girdi	sendbuf	gönderilecek iletinin adresi
çıktı	recvbuf	alınacak iletilerin kaydedileceği adres
girdi	count	iletilerin uzunluğu
girdi	datatype	iletilerin veri türü
girdi	op	iletiler üzerinde gerçekleştirilecek işlem
girdi	root	iletileri alacak olan görevin kodu
girdi	comm	görevin bağlı olduğu ağ (*communicator*)

```
int MPI_Exscan(void* sendbuf, void* recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

MPI_SCAN yordamıyla aynı şekilde çalışır. Sadece sonuç üretmeye ilk görevden değil, bir sonraki görevden başlar.

3.3.3. İşlem topolojileri

Topolojiler (ilinge), koşut bilgisayar programlama için özellikle donanım ortamının yazılım içinde kullanılabilmesini sağlamak amacıyla oluşturulmuştur. Bilgisayarlar arasındaki iletişim bazı durumlarda sıkıntı olabilirken, yazılım içinde bu durum öngörülerek uygulama geliştirilmesi bir avantaja da dönüşebilmektedir.

Ekran kartlarında donanım uygulama geliştiricilerden soyutlanmıştır. Ancak, önceden geliştirilmiş MPI uygulamalarının kullanabilmesi ve MPI görev topolojileri ile uygulama geliştirilebilmesi için tez çalışmasında topoloji yordamlarının da gerçekleştirimi yapılmıştır.

Topolojilerin oluşturulması için ihtiyaç duyulan veri yapılarında;

- Topoloji tipi
- Kartezyen topolojileri için;
    - Boyut sayısı
    - Her boyuttaki görev sayısı
    - Görevlerin boyutlardaki sıralanma koşulu
    - Her görevin sıra numarası
- Çizge topolojileri için;
    - Her görevin komşu bilgileri

tutulmaktadır [2].

MPI, kitaplığın gerçekleştiriminde standartları belirlemekte, gereksinimlerin karşılanmasında kullanılan yöntemleri kısıtlamamaktadır. Bir topolojinin oluşturulması için gerekli kuralları belirlemiş, somut olarak topolojinin kodlanmasını zorunlu tutmamıştır. Tez çalışmasında da gerekli değerlerin veri yapılarında tutulması yeterli görülmüş, ekran kartlarının işleme performanslarının yüksek olmasının bir avantaj oluşturacağı öngörüsüyle görevlerin topolojideki koordinatlarının bellekte tutulmak yerine her seferinde hesaplanarak belirlenmesi sağlanmıştır.

3.3.3.1. Kartezyen topolojisi

MPI_CART_CREATE	MPI_CART_SUB
MPI_CARTDIM_GET	MPI_CART_GET
MPI_CART_RANK	MPI_CART_COORDS
MPI_CART_MAP	

MPI_CART_CREATE

MPI_CART_CREATE yordamı, ağ içindeki görevlerden bir kartezyen topolojisinin oluşturulmasını sağlar. Belirlenen boyut sayısı ve her boyuttaki görev sayısına göre topoloji şekillendirilir. İsteğe bağlı olarak görevlerin ağdaki sıralaması da değiştirilebilmektedir. Ağ içindeki bütün görevler tarafından çağırılması gereklidir.

MPI standartlarında MPI_CART_CREATE yordamı, isteğe bağlı olarak görevlerin ağdaki sıralaması donanım sistemine göre en uygun şekilde tekrar yapılmasını sağlayacak şekilde tanımlanmıştır. Ekran kartı üzerinde geliştirilen uygulamalarda en uygun düzen görevlerin sıralı düzenidir. İlgili gereksinim CUDA tarafından karşılanmaktadır.

Bir topolojinin yaratılması, görevlerden yeni bir grup/ağ oluşturulmasını gerektirir. Bu işlemin kontrolü ilk göreve verilmiştir. İlgili veri yapılarının kullanılması, ağın boyu, görevlerin sıralaması grup içindeki ilk görev tarafından gerçekleştirilir. Diğer görevler ağın oluşmasını ve bilgilerinin kendilerine gönderilmesini beklerler.

MPI Yordam Tanımları

- MPI_CART_CREATE (comm_old, ndims, dims, periods, reorder, comm_cart)

girdi	comm_old	görevin bağlı olduğu ağ (*communicator*)
girdi	ndims	oluşturulacak topolojinin boyut sayısı
girdi	dims	her boyuttaki görev sayısı
girdi	periods	görevlerin boyutlardaki sıralanma koşulu
girdi	reorder	görevlerin yeniden sıralanma koşulu
çıktı	comm_cart	kartezyen topolojisi oluşturulmuş yeni ağ

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
        int *periods, int reorder, MPI_Comm *comm_cart)
```

- MPI_CART_SUB (comm, remain_dims, newcomm)

girdi	comm	görevin bağlı olduğu ağ (*communicator*)
girdi	remain_dims	ayrılacak boyutları belirtme koşulu
çıktı	newcomm	yeni oluşturulan ağ

```
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims,
        MPI_Comm *new_comm)
```

Kartezyen topolojisindeki bir ağın, alt gruplara bölünmesini sağlamak için kullanılır. Oluşturulan her alt ağ, yeni bir ağdır. MPI_COMM_SPLIT yordamı kullanılarak gerçekleştirilmiştir.

- MPI_CARTDIM_GET (comm, ndims)

girdi	comm	görevin bağlı olduğu ağ (*communicator*)
çıktı	ndims	topolojinin boyut değeri

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

Topolojinin kaç boyutunun olduğu bilgisini sorgulamak için kullanılır.

- MPI_CART_GET (comm, maxdims, dims, periods, coords)

girdi	comm	görevin bağlı olduğu ağ (*communicator*)
girdi	maxdims	ağdaki boyut sayısı
çıktı	dims	her boyuttaki görev sayısı
çıktı	periods	görevlerin boyutlardaki sıralanma koşulu
çıktı	coords	yordamı çağıran görevin topolojideki yeri

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims,
        int *periods, int *coords)
```

Kartezyen topolojisindeki görevlerin koordinat bilgilerini sorgulayabilmeleri için tanımlanmıştır. Topoloji veri yapılarında, görevlerin numaraları tutulmaktadır. Koordinat bilgisi bu numaralar kullanılarak hesaplanır ve göreve bildirilir.

- MPI_CART_RANK (comm, coords, rank)

girdi	comm	görevin bağlı olduğu ağ (communicator)
girdi	coords	yordamı çağıran görevin topolojideki yeri
çıktı	rank	yordamı çağıran görevin kod bilgisi

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

Koordinat bilgisi verilen görevin numarasını sorgulamak için kullanılır. Yordamın gerçekleştirimi, gelen veri üzerinde gerekli hesaplamanın yapılması ile sağlanmıştır.

- MPI_CART_COORDS (comm, rank, maxdims, coords)

girdi	comm	görevin bağlı olduğu ağ (communicator)
girdi	rank	yordamı çağıran görevin kod bilgisi
girdi	maxdims	ağdaki boyut sayısı
çıktı	coords	numarası verilen görevin topolojideki yeri

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims,
        int *coords)
```

Bilgileri verilen görevin topoloji içindeki yerinin sorgulanması için kullanılır. Verilen değerlerin hesaplanması ile sonuç üretilir.

- MPI_CART_MAP (comm, ndims, dims, periods, newrank)

girdi	comm	görevin bağlı olduğu ağ (*communicator*)
girdi	ndims	oluşturulacak topolojinin boyut sayısı
girdi	dims	her boyuttaki görev sayısı
girdi	periods	görevlerin boyutlardaki sıralanma koşulu
çıktı	newrank	görevin ağdaki yeni kod bilgisi

```
int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims,
        int *periods, int *newrank)
```

Varolan topoloji ağının donanım ortamına en uygun şekilde yeniden oluşturulmasını sağlamak için tanımlanmıştır. CUDA modeli için anlamlı değildir. Geri dönüş değeri olarak, görevin numarası tekrar kullanılmaktadır.

3.3.3.2. Çizge topolojisi yordamları

MPI_GRAPH_CREATE	MPI_GRAPHDIMS_GET
MPI_GRAPH_GET	MPI_GRAPH_NEIGHBORS_COUNT
MPI_GRAPH_NEIGHBORS	MPI_GRAPH_MAP

Çizge topolojilerinin gerçekleştirimi, kartezyen topolojileri ile benzerdir. Yordamlar çağırılırken verilen değerler, ilgili topolojinin veri yapısına kaydedilir ve sorgulanan bilgi hesaplanarak bildirilir.

MPI Yordam Tanımları

- MPI_GRAPH_CREATE (comm_old, nnodes, index, edges, reorder, comm_graph)

girdi	comm_old	görevin bağlı olduğu ağ (*communicator*)
girdi	nnodes	oluşturulacak topolojideki görev sayısı
girdi	index	komşu görev dizisi değerleri
girdi	edges	görevlerin sahip olduğu komşu görev bilgileri
girdi	reorder	görevlerin yeniden sıralanma koşulu
çıktı	comm_graph	çizge topolojisi oluşturulmuş yeni ağ

```
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index,
        int *edges, int reorder, MPI_Comm *comm_graph)
```

Bir çizge topolojisi yaratmak için kullanılır. MPI_CART_CREATE yordamı gibi çizge topolojisi veri yapısının değerlerini günleyerek, topolojiyi oluşturur. Diğer topoloji yordamları, bu veri yapısını kullanarak sorgulama yaparlar.

- MPI_GRAPHDIMS_GET (comm, nnodes, nedges)

girdi	comm	görevin bağlı olduğu ağ (*communicator*)
çıktı	nnodes	topolojideki görev sayısı
çıktı	nedges	görevlerin sahip olduğu komşu görev bilgileri

```
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)
```

Çizge topoloji bilgilerini sorgulamak için kullanılır.

- MPI_GRAPH_GET (comm, maxindex, maxedges, index, edges)

girdi	comm	görevin bağlı olduğu ağ (*communicator*)
girdi	maxindex	bilgi alınacak değer için dizi boyu bilgisi
girdi	maxedges	bilgi alınacak değer için dizi boyu bilgisi
çıktı	index	komşu görev dizisi değerleri
çıktı	edges	görevlerin sahip olduğu komşu görev bilgileri

```
int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges,
        int *index, int *edges)
```

Çizge topoloji bilgilerini sorgulamak için kullanılır.

- MPI_GRAPH_NEIGHBORS_COUNT (comm, rank, nneighbors)

girdi	comm	görevin bağlı olduğu ağ (*communicator*)
girdi	rank	yordamı çağıran görevin kodu
çıktı	nneighbors	görevin komşu sayısı

```
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank,
        int *nneigbors)
```

Bir görevin komşu sayısını sorgulamak için kullanılır.

- MPI_GRAPH_NEIGHBORS (comm, rank, maxneighbors, neighbors)

girdi	comm	görevin bağlı olduğu ağ (*communicator*)
girdi	rank	görevin kodu
girdi	maxneighbors	bilgi alınacak değer için dizi boyu bilgisi
çıktı	neighbors	istenen görev için komşu görev bilgileri

```
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int
         maxneighbors, int *neighbors)
```

Bir görevin komşu bilgilerini sorgulamak için kullanılır.

- MPI_GRAPH_MAP (comm, nnodes, index, edges, newrank)

girdi	comm	görevin bağlı olduğu ağ (*communicator*)
girdi	nnodes	oluşturulacak topolojideki görev sayısı
girdi	index	komşu görev dizisi değerleri
girdi	edges	görevlerin sahip olduğu komşu görev bilgileri
çıktı	newrank	görevin ağdaki yeni kod bilgisi

```
int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index, int
        *edges, int *newrank)
```

MPI_CART_MAP gibi, topoloji ağının yeniden oluşturulmasını sağlamak için tanımlanmıştır. Geri dönüş değeri olarak, görevin numarası tekrar kullanılmaktadır.

3.3.3.3. Topoloji tür bilgisi sorgulama

- MPI_TOPO_TEST (comm, status)

girdi	comm	görevin bağlı olduğu ağ (*communicator*)
çıktı	status	ağın topoloji türü

```
int MPI_Topo_test(MPI_Comm comm, int *status)
```

Oluşturulan bir ağın, hangi topoloji türünden olduğu bilgisini sorgulamak için kullanılır.

3.3.4. Ortam yönetimi

Uygulama geliştirme ortamı hakkında bilgi almak ve ortamı yönetmek için kullanılan yordamlardır. İhtiyaç duyulan genel yordamlar tez çalışmasında geliştirilmiştir.

- MPI_COMM_SIZE (comm, size)

girdi	comm	görevin bağlı olduğu ağ (*communicator*)
çıktı	size	ağdaki görev sayısı

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

Ağdaki görev sayısını bildirir.

- MPI_COMM_RANK (comm, rank)

girdi	comm	görevin bağlı olduğu ağ (*communicator*)
çıktı	rank	yordamı çağıran görevin kod bilgisi

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Yordamı çağıran görevin, görev kodunu bildirir.

- MPI_COMM_SPLIT (comm, color, key, newcomm)

girdi	comm	görevin bağlı olduğu ağ (*communicator*)
girdi	color	oluşturulacak alt küme için kontrol değeri
girdi	key	görevin yeni ağda alacağı kod değeri bilgisi
çıktı	newcomm	oluşturulan yeni ağ

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
        MPI_Comm *newcomm)
```

Bir ağı, verilen renk ve anahtar değerlerine göre alt gruplara bölmek için kullanılır. Aynı renk bilgisine ait görevler kendi içinde yeni bir ağ oluşturur.

- MPI_INIT()

```
int MPI_Init()
```

- MPI_FINALIZE()

```
int MPI_Finalize()
```

MPI uygulamalarında koşut olarak çalıştırılan bölümün başlangıç ve bitiş noktalarıdır. İletişim başlaması ve bitmesi için gerekli günlemelerin yapılabilmesi amacıyla tanımlanmıştır. Tez çalışmasında, uygumla geliştiriciye sunulan ortamın önceden hazırlanmış olmasından dolayı ek bir gerçekleştirime ihtiyaç duyulmamıştır.

3.4. Uygulama Geliştirilirken Kullanılabilecek Yardımcı İşlevler

CUDA ile uygulama geliştirmek, programlamacılara bazı kısıtlar getirmektedir. Bu kısıtların en büyüğü, işlevin ekran kartı üzerinde çalıştığı sürece bir çıktı üretmemesidir. Geliştirilen kütüphane kullanılarak kodlanan bir uygulama, baştan sona, ekran kartı üzerinde çalışacaktır. Uygulamanın herhangi bir evresinde geri bildirim alınmak istendiğinde, ya da olası hataların fark edilmesi gerektiğinde bu ihtiyaçların karşılanması mümkün olmayacaktır. Bu tip gereksinimleri karşılamak için, bazı yordamlar tanımlanmıştır.

```
printStr (msg, str, len)        printInt (msg, val)
printFloat (msg, val)           printEndl();
```

Bu yordamlar, uygulama anında, kodun herhangi bir yerinde, özel bellek alanına ihtiyaç duyulan bildirimlerin aktarılmasına imkan verir. Uygulama tamamlandıktan sonra, bu iletilerin ekrana getirilmesi de sağlanmaktadır.

3.5. Uygulama Geliştirilirken Uyulması Gereken Kısıtlar

Tez kapsamında geliştirilen uygulamayı kullanmak için bazı kurallar ve kısıtlar bulunmaktadır. Uygulama geliştiricilerin bu kurallara uyarak yazılım geliştirmeleri gerekir. Bu kuralların bir kısmı geliştirilen MPI yazılım kitaplığı gereksinimlerinden, bir kısmı ise CUDA programlama modeli ile uygulama geliştirmedeki kısıtlardan dolayı oluşturulmuştur.

Kurallara ve kısıtlara uygun örnek bir kodlama EK-2'de verilmiştir.

Uygulama geliştiricilerin uyması gereken 2 temel kural vardır:

1. Her işlevin önüne `__device__` niteleyicisi konulmalıdır.
2. Her değişkenin önüne `__shared__` niteleyicisi eklenmelidir.

Ek olarak bazı kısıtlar mevcuttur.

- Uygulama içinde çok boyutlu diziler kullanılamamaktadır. Çok boyutlu diziler, tek boyuta çevrilmelidir.
- Göstergeler için çalışma anında bellek ayrılamaz. Göstergeler, önceden ayrılmış bellek alanı üzerinde kullanılır.
- Özyineli işlevler desteklenmemektedir.
- Ekran kartı üzerindeki işlemcilerin özel belleklerinin tamamı uygulama geliştiricinin kullanımına bırakılmıştır. Bu belleğin büyüklüğü 16KB'tır. Görev sayısı 30'un üzerine çıktığında bu değer azalmaktadır.

Kullanılabilecek C veri türleri ve karşılık gelen MPI veri türleri çizelgede gösterilmiştir (Çizelge 3-2).

Çizelge 3-2. Uygulama geliştirilirken kullanılabilecek MPI ve C veri türleri

MPI Veri Türleri	**C Veri Türleri**
MPI_CHAR	`char`
MPI_SHORT	`short`
MPI_INT	`int`
MPI_LONG	`long int`
MPI_LONG_LONG_INT	`long long int`
MPI_LONG_LONG	
MPI_SIGNED_CHAR	`unsinged char`
MPI_UNSIGNED_CHAR	
MPI_UNSIGNED_SHORT	`unsinged short`
MPI_UNSIGNED	`unsinged int`
MPI_UNSIGNED_LONG	`unsinged long int`
MPI_UNSIGNED_LONG_LONG	`unsinged long long int`
MPI_FLOAT	`float`
MPI_DOUBLE	`double`
MPI_LONG_DOUBLE	`long double`

4. ÖNCEDEN GELİŞTİRİLMİŞ UYGULAMALAR İLE KİTAPLIĞIN KULLANIMI

Tez çalışmasında geliştirilen MPI yazılım kitaplığı, gereksinimleri karşılaması ve başarımı yönünden test edilmiştir. Bu amaçla önceden geliştirilmiş bazı MPI algoritmaları ve uygulamalar kullanılmış, önceki bölümde belirtilen programlama kısıtlarına uygun olarak düzenlenmiştir. Bu bölümde temel MPI yordamlarıyla geliştirilen bir arama algoritması ve kartezyen topolojisinin kullanıldığı Cannon Algoritması ile matris-matris çarpımı uygulamalarının yazılım kitaplığıyla kullanımı anlatılacaktır.

4.1. En Büyük Değeri Bulma Uygulaması

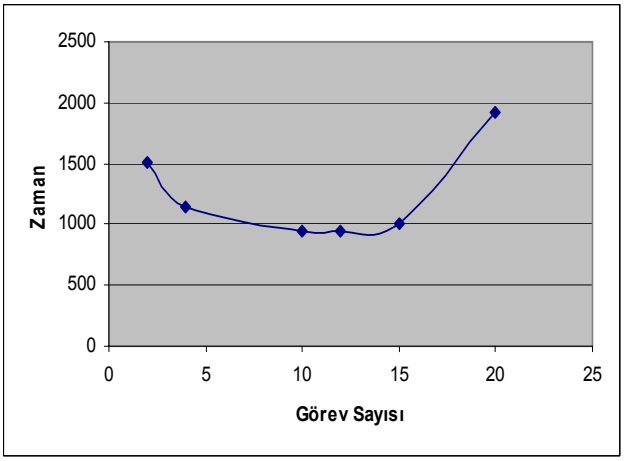
Yazılım kitaplığının başarım testi için kullanılmış olan bir uygulamadır. MPI'ın temel 6 yordamını içerir. Farklı MPI yordamları ile de gerçekleştirilmesi mümkündür. Kullanılan yordamlar şunlardır:

MPI_INIT	MPI_FINALIZE
MPI_COMM_SIZE	MPI_COMM_RANK
MPI_SEND	MPI_RECV

Uygulama bir adet yönetici görev ve değişik sayıda çalışan görevden oluşturulmuştur. Yönetici görev, arama yapılacak veri kümesini çalışan görevlere dağıtır. Bütün görevler sahip olduğu veri öbeği içinde en büyük değeri ararlar. Bulunan değerler ana göreve iletilir ve son olarak görev sayısı kadar en büyük değerler yönetici görevde toplanır. Bu değerler içindeki en büyük değer de bulunup uygulama sonlanır.

Uygulama, işlem yoğunluğunun artırılarak, işlemci sayısının başarım üzerindeki etkisini ölçmek için kullanılmıştır. Uygulama `int` veri türündeki değerler ile 2, 4, 10, 12, 15 ve 20 görev kullanılarak çalıştırılmış ve başarımı Şekil 4-1'de gösterilmiştir.

Görev Sayısı	Süre (µs)
2	1505
4	1144
10	946
12	938
15	1003
20	1923



Şekil 4-1. En Büyük Değeri Bulma Uygulaması Başarımı

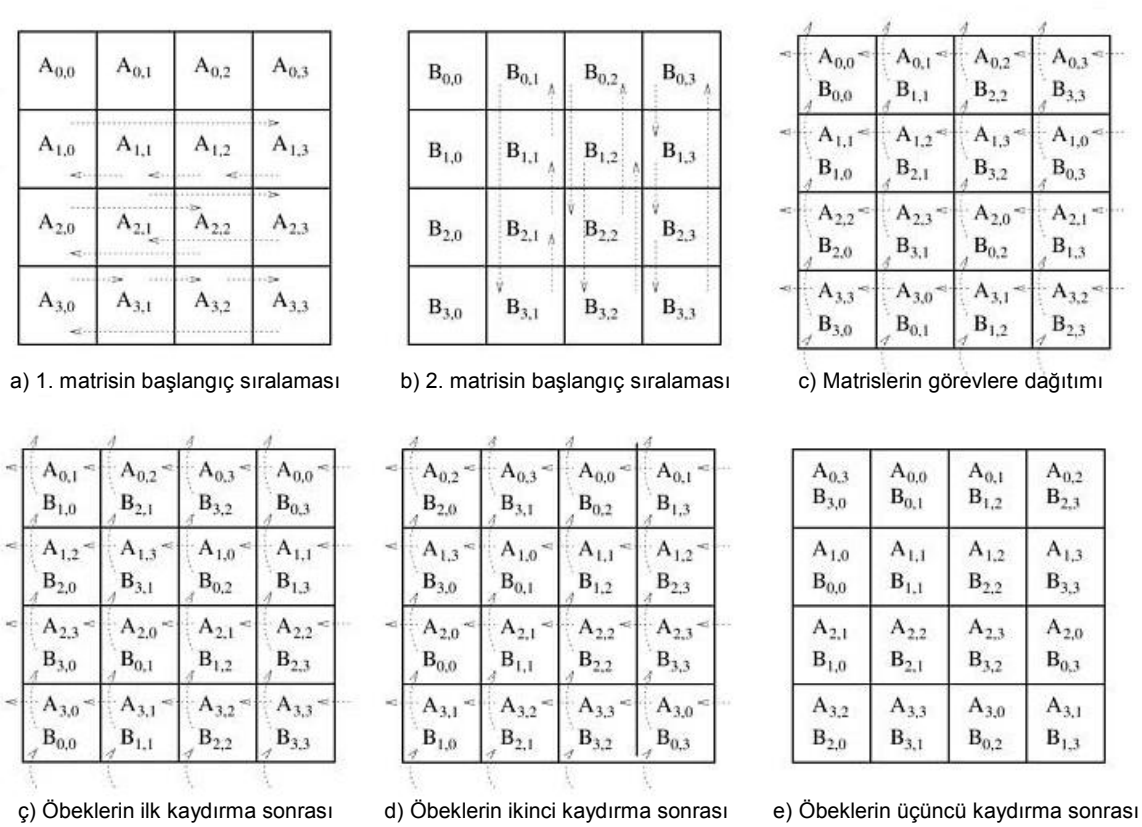
4.2. Koşut Matris Çarpma Uygulaması

Matris çarpma uygulaması kartezyen topolojisi ile geliştirilmiştir. Yazılım kitaplığının başarımını ve topolojilerin kullanımını örneklemek için tez metninde açıklanmıştır. Kullanılan MPI yordamları şunlardır:

MPI_INIT
MPI_COMM_SIZE
MPI_BARRIER
MPI_SCATTER
MPI_CART_CREATE
MPI_CART_SUB
MPI_FINALIZE
MPI_COMM_RANK
MPI_BCAST
MPI_GATHER
MPI_CART_COORDS
MPI_SENDRECV_REPLACE

Koşut uygulamalarda kullanılan matris çarpma algoritmalarından biri Cannon Algoritması'dır. Matris çarpma işlemlerinde belleğin etkin olarak kullanılmasını sağlar. Cannon Algoritması basit olarak şu şekilde açıklanabilir (Şekil 4-2):

- Matrisler görev sayısının kareköküyle orantılı olarak öbeklere ayrılır. Öbekler, ilk matris için yatay sırada, bulunduğu satırın sıra sayısı kadar sola kaydırılır. İkinci matriste ise bulunduğu sütunun sırası kadar yukarı kaydırılır. Oluşan yeni matrisin öbekleri görevlere dağıtılır.
- Her görev, sahip olduğu alt matrisler üzerinde çarpma işlemini gerçekleştirir ve ilk matrisin öbeğini soldaki göreve, ikinci matrisin öbeğini ise bir üst göreve iletir.



Şekil 4-2. 16 görev için Cannon Algoritması'nın kullanımı [1]

- Görevler her yeni alt matris ile çarpma işlemi yaparak bir önceki matris ile toplarlar. Bu işlemler, ilk baştaki yatay ve dikey sıradaki bütün öbeklerin işlenmesi bitene kadar sürer.
- İşlemlerin sonucu bir görevde toplanır ve matris çarpma işlemi tamamlanır.

Uygulama içinde çok sayıda dizi ve değişken kullanılmaktadır. Bu nedenle uygulama, bellek alanın büyüklüğü göz önüne alınarak, Çizelge 4-1'deki değerlerle denenmiş ve çalışma süresi kaydedilmiştir.

Çizelge 4-1. Koşut Matris Çarpma Uygulaması başarımı

Görev Sayısı	**Matrislerin Büyüklüğü (`int`)**	**İşlem Süresi (ms)**
4	8 x 8	1,60
4	16 x 16	2,35
16	8 x 8	3,06
16	16 x 16	3,94

4.3. Uygulamaların Başarımının Yorumlanması

Uygulamaların başarım sonuçları şu şekilde yorumlanmıştır:

- Çalışan görev sayısının artması, uygulamanın başarımını düşürmektedir. Bunun nedeni iletişim gereksinimlerinin artmasıdır. Görevler birbirleriyle genel bellek üzerinden haberleşmektedir. Verilerin işlenme hızı ise belleğe erişim hızından çok yüksektir. Sistem üzerindeki iletişim arttıkça, görevler arasında iletişimden kaynaklanan kısıtlar oluşmaktadır. Ek olarak, görevler genel belleğe erişim yaparken birbirlerini beklemek zorunda da kalmaktadır. Bu da başarımı düşürmektedir. Görevlerin hesaplama karmaşıklığının yüksek olduğu uygulamalarda koşut uygulamanın artıları gözlemlenebilmektedir. Ancak bellek alanlarının büyüklüğü, işlem hızlarının yanında başarımı gösterebilmek için yeterli değildir.

- Veri büyüklüğünün artması, kabul edilebilir derecede çalışma süresini artırmaktadır. İşlemcilerin büyük veriler üzerinde işlem yapması başarımı fazla etkilememekte, ancak, bu verileri diğer görevlere iletirken kullanılan genel belleğin erişim hızının düşük olması çalışma hızını yavaşlatmaktadır.

5. SONUÇLAR VE ÖNERİLER

MPI, bilgisayar mühendisliği öğrencilerinin ve koşut uygulama geliştiricilerinin öğrenmesi gereken koşut işlem alanındaki önemli bir standarttır. Birçok kuruluş tarafından desteklendiği için, farklı dillerde çeşitli gerçekleştirimleri bulunmaktadır. Ancak koşut bilgisayar donanımlarının oluşturulması büyük maddi yük getirmektedir.

Günümüzde ekran kartları, normal bilgisayarlar gibi genel amaçlı programlama için kullanılabilmektedir. NVIDIA'nın ekran kartları üzerinde uygulama geliştirmek için başlattığı proje, kısa sürede birçok programlamacı tarafından benimsenmiş ve kullanılmaya başlanmıştır. Bu modelin gelişimi halen devam etmektedir.

Tez çalışmasında, farklı iki model bir araya getirilmiş, bu alandaki bazı eksikliklerin giderilmesine çalışılmıştır.

- Öncelikle, MPI yazılım kitaplığıyla ilgilenen öğrencilerin, bir bilgisayar donanımı içinde, düşük maliyetli bir aygıt vasıtasıyla, gerçek bir koşut bilgisayar mimarisine sahip olmaları sağlanmıştır.

- MPI ile uygulama geliştiren programlamacıların, ekran kartlarının yüksek başarımını kullanabilmesi ve donanımı kullanmak için ek bir öğrenme külfeti altına girmemeleri amaçlanmıştır.

- CUDA ile koşut programlama yapan uygulama geliştiricilerin, varolan MPI algoritmalarını kullanabilmelerine olanak sağlanmıştır.

Yapılan testlerde, geliştirilen uygulamalar ile ekran kartlarının yüksek başarıma ulaşabildikleri görülmüştür.

Kullanılan CUDA programlama modelinin henüz yeni gelişmekte olması, donanımın aynı zamanda ticari bir ürün olması karşılaşılan sorunlara ilgili yeterince kaynak bulunamamasına neden olmuştur. Hedeflenen bazı MPI yordamları, donanımdan kaynaklan kısıtlar nedeniyle gerçekleştirilememiştir.

Geliştirilen kodun ekran kartı üzerinde çalışması, hata bulma işlemi (*debug*) yapılmasına imkan vermemektedir. Bu nedenle, tasarımdan kaynaklanan hataların yakalanması, kod üzerindeki hatalı bölümün bulunması da çok zaman ve emek kaybına neden olmuştur. Ayrıca, hata ihtimalini azaltmak, bağımsız çalışan işlemcilerin birbirlerinin çalışmasını engellememesi için birçok kısıtlayıcı kullanılmış, bu durum da aygıta ek yük getirmiştir.

Kilitlenmeler için kullanılan atomik işlevlerin genel belleğe yaptığı erişimler uzun zaman almaktadır. MPI yordamlarının düzgün çalışabilmesi için bu işlevlerin döngüler içinde defalarca çağırılması gerekmiştir. Bu durum da başarımın düşmesine; birçok işlevin kullanıldığı uygulamalarda, görev sayısı işlemci sayısının birkaç katına çıktığında uygulamanın kilitlenmesine neden olmaktadır.

Ekran kartları donanımının ve CUDA modelindeki gelişmelerin, tez çalışması kapsamında geliştirilen MPI yazılım kitaplığı yordamlarının ve uygulamaların başarımlarını arttıracağı öngörülmektedir. Bu gelişmelerin diğer MPI yordamlarının da gerçekleştiriminin yapılabilmesine olanak sağlayacağı düşünülmektedir.

CUDA programlama modeli, başarımı artırmak için iş dizilerinin kullanılmasına olanak verir. Mevcut uygulamada iş dizileri karmaşıklığı arttırmamak için kullanılmamıştır. Ancak, geleceğe yönelik önerilebilecek yöntemler için, bazı yordamların çoklu iş parçacığı (*multithread*) ile gerçekleştirimleri yapılmıştır. Bu yordamlarda, çoklu iş parçacıklarının özellikle veri aktarımında yüksek başarım sağladığı görülmüştür. Ek olarak, aritmetik işlemlerde de kullanılabilecekleri tespit edilmiştir.

EKLER DİZİNİ

EK 1. NVIDIA GEFORCE GTX 280 EKRAN KARTI

Son 10 yılda, NVIDIA grafik işlemcileri (GPU), bir anlamda evrim geçirmiştir. Özelleşmiş, üç boyutlu belli işlevleri yapabilen grafik işlemcilerden; görsel işlemleri yüksek başarımla sağlayabilen büyük oranda programlanabilir, çoklu iş dizisi kullanan koşut işlem mimarilerine dönüşmüştür [8].

Tez çalışmasında koşut programlama donanımı olarak kullanılan NVidia GeForce GTX 280 model ekran kartı, NVIDIA'nın ikinci nesil bütünleşik grafik ve hesaplama mimarisi olan GeForce GTX 200 ailesindendir. Kişisel bilgisayarlarda kullanılabilecek üst sınıf ekran kartıdır.

1.1. GeForce GTX 200 GPU Mimarisi

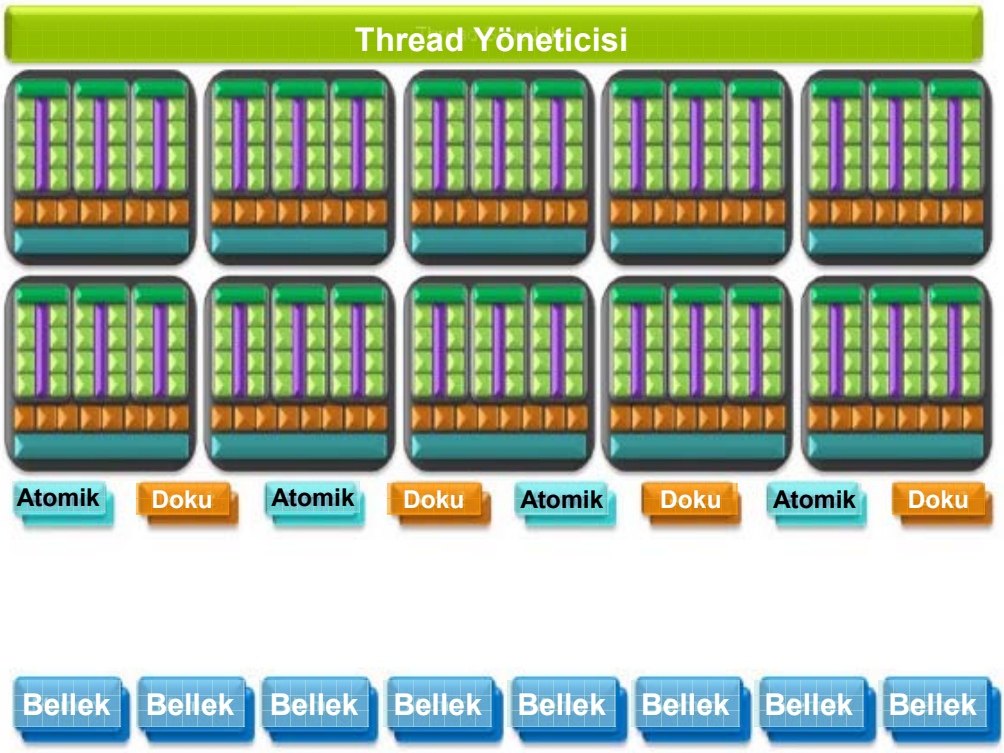
GTX 200, 10 iş parçacığı işleme kümesinden (TPC, Thread Processing Cluster) oluşur. Her TPC, 3 adet *streaming multiprocessor*'den (SM, duraksız işlem çoklu işlemci) ve bu işlemcilerin belleğe erişimini sağlayan ardışık düzen biriminden meydana gelir. Çoklu işlemciler, sekizli tek komut/çok veri işleme çekirdeğine (SP, Streaming Processor), işleme birimine, komut yönetimine sahiptir[9]. Her SM'in, çekirdekleri tarafından kullanılabilecek paylaşımlı belleği bulunur.

Şekil EK1-1'de, ekran kartının donanımı gösterilmiştir. Şekilde görülen donanım tabanlı iş parçacığı (*thread*) yöneticisi, iş dizilerinin TPC birimleri üzerinde çalışmasını programlar. Kart üzerinde işlemcilerin belleğe erişimleri için bellek arayüz birimleri (bellek denetleyicisi), doku bellek için de ikinci seviye ön bellek bulunmaktadır. Atomik işlevlerin gerçekleştirimi, donanım üzerindeki özel bir birimin kontrolünde sağlanmaktadır.

SIMT Modeli: CUDA, iki değişik koşut bilgisayar işleme modelini kullanır. Grafik işlemcilerin yönetimi için etkin mimari MIMD(Multiple Instruction/Multiple Data, Çok Komut/Çok Veri)'dir. Her SM'nin kendi içindeki yönetimi için ise SIMT(Single Instruction/Multiple Thread) mimarisi tasarlanmıştır[8].

NVIDIA, SIMT modelini SIMD(Single Instruction/Multiple Data, Tek Komut/Çok Veri) mimarisinin bir çeşidi olarak tanımlar. SIMD modelinde, iş yapan vektör genişliği bellidir ve veri vektörler arasında dağıtılmıştır. Aynı vektör içinde bulunan birimler, ortak adres evreninin kullandıkları için birbirleriyle sınırsız iletişim

sağlamaktadır. SIMT modelinde ise, çalışma genişliği mikro mimari özelliği olarak donanım tarafından kontrol edilmektedir. Komutları işleten diziler birbirinden bağımsız hareket edebilmektedir. İş parçacıkları paylaşımlı belleğe ortak olarak erişebilmekte yazmaçlarını ise sadece kendi kullanmaktadırlar[9].



Şekil EK1-1. GeForce GTX 280 GPU Koşut İşleme Mimarisi [8]

Yazmaç Kütüğü: GTX 280 ekran kartında yazmaç kütükleri 64KB büyüklüğündedir. Her SM için 16K yazmaç bulunur. SP başına da 2K yazmaç düşmektedir ve bu yazmaçlar 128 *thread* tarafından da paylaşılabilir. Bir yazmacın boyu 32 bittir. Çift duyarlı veriler iki yazmacın birlikte kullanılması ile işlenir.

Paylaşımlı Bellek: Her SM 16KB paylaşımlı belleğe sahiptir. Bu belleklere erişim, yazmaçlara olduğu gibi çok hızlıdır. Paylaşımlı bellek, aynı SM'yi kullanan farklı öbekler (*block*) arasında paylaştırılır. 32 bitlik bellek arayüzü bulunur.

Paylaşımlı bellek üzerinde atomik işlevler de kullanılabilir. Bellek alanı üzerinde tek zamanda okuma ve yazma işlemleri yapmak mümkündür.

Genel Bellek: GeForce GTX 280 ekran kartı üzerinde 1GB bellek bulunmaktadır. Genel belleğe erişim diğer belleklere oranla çok yavaştır. Her iş parçacığı 64 bitlik veri erişimi yapabilmekte bu sayede tek seferde 512 bit veri erişimi sağlanabilmektedir. Çoklu iş parçacığı kullanımında veri aktarım başarımının arttırılmasına çalışılmıştır.

1.2. Teknik Özellikler

Çizelge EK1-1. GeForce GTX 280 GPU Teknik Özellikleri [10]

Çekirdek	240
İşlemci Hızı	1296 MHz
Doku İşleme Hızı	48.2 (milyar/sn)
Bellek Hızı	1107 MHz
Bellek Büyüklüğü	1 GB
Bellek Arayüz Genişliği	512 bit
Bellek Veri Yolu Genişliği	141.7 (GB/sn)
Sayısal Çözünürlük	2560 x 1600
VGA Çözünürlük	2048 x 1536
Veri Çıkış Arayüzleri	HDTV, Çift kanallı DVI
Diğer Görüntüleme Desteği	Çoklu Görüntü Birimi HDMI HDCP
Yükseklik	111 mm
Uzunluk	267 mm
Genişlik	Çift Yuva
En Yüksek GPU Sıcaklığı	105 C^0
Grafik Kart Gücü	236 W
Sistem Güç İhtiyacı	550 W
Güç Kaynağı Bağlayıcı Arayüzü	6-8 pin

EK 2. CUDA İLE GELİŞTİRİLMİŞ ÖRNEK MPI UYGULAMASI

Tez çalışmasında geliştirilen yazılım kitaplığının kural ve kısıtlarına uyularak geliştirilmiş örnek kod aşağıda verilmiştir. Verilen kod parçasında, işlev ve değişken tanımlamaları gösterilmiş; kısıtlara uygun olarak MPI yordamlarının kullanımı örneklenmiştir.

```
__device__ void MPImain()
{
   //görev sayısı 10 iken
   __shared__ float dizi[10], _2BDizi[100];
   __shared__ int gorev_kodu, gorev_sayisi;
   __shared__ int i, j;

   MPI_Init();
   MPI_Comm_size(MPI_COMM_WORLD, &gorev_sayisi);
   MPI_Comm_rank(MPI_COMM_WORLD, &gorev_kodu);

   for(i = 0; i < 10; i++)
      dizi[i] = gorev_kodu;

   //dizileri 0 nolu görevde topla
   MPI_Gather(dizi,10,MPI_FLOAT,_2BDizi,10,MPI_FLOAT,0,MPI_COMM_WOLRD);

   if(gorev_kodu == 0){
      printStr("Önce:\n", "",0);
      for(i = 0; i < 10 ; i++){
         for(j = 0 ; j < 10; j++)
            printFloat(" ", _2BDizi[i * 10 + j]);
         printEndl();
      }
   }

   //çift görevlerin dizilerini tek görevlere kopyala
   if( gorev_kodu == gorev_sayisi%2 )
      MPI_Send(dizi, 10, MPI_FLOAT, gorev_kodu+1, 0, MPI_COMM_WORLD);
   else
      MPI_Recv(dizi, 10, MPI_FLOAT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD);

   //dizileri 0 nolu görevde topla
   MPI_Gather(dizi,10,MPI_FLOAT,_2BDizi,10,MPI_FLOAT,0,MPI_COMM_WOLRD);

   if(gorev_kodu == 0){
      printStr("Sonra:\n", "",0);
      for(i = 0; i < 10 ; i++){
         for(j = 0 ; j < 10; j++)
            printFloat("", _2BDizi[i * 10 + j]);
         printEndl();
      }
   }

   MPI_Finalize();
}
```

EK 3. TEZ METNİNDE AÇIKLANAN UYGULAMALARIN KODLARI

Yazılım kitaplığının testi için kullanılan uygulamaların kodları ve uygulamaları açıklamaları aşağıda verilmiştir.

1. En Büyük Değeri Bulma Uygulaması

En büyük değeri bulma uygulaması; işlem/iletişim oranını arttırmak için kullanılan doğrusal arama algoritmasının kullanıldığı bir uygulamadır. Aşağıda kodu verilen uygulama, büyük bir veri kümesi içindeki en büyük sayısal değeri aramaktadır. İlk görev arama yapılacak diziyi diğer görevlere parçalayarak dağıtır. Her görev, kendisine iletilen dizi parçası içinde en büyük değeri arar. Bulduğu değeri ana göreve iletir. Ana görev gelen değerler arasındaki en büyük değeri bulur ve ekranda gösterir.

```
#define   LEN   3600
__device__ int global_array[LEN];

//global_array dizisinin içi yordam çağırılmadan önce doldurulmuştur.

__device__ void max_num()
{
   __shared__ int rank, size;
   __shared__ int MatrixA[LEN], MatrixB[50], max_num;
   __shared__ MPI_Status status;
   __shared__ int len, i,j;

   MPI_Init();
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   len = LEN / size;
   max_num = 0;
   //dizi 0 no'lu görev tarafından alınır.
   if(rank == 0)
      for(i = 0; i < LEN; i ++)
            MatrixA[i] = global_array[i];

   //arama yapılacak dizi görevlere dağıtılır.
   if(rank == 0)
      for(i = 1; i < size; i++)
         MPI_Send(&MatrixA[i*len], len, MPI_INT, i, 0, MPI_COMM_WORLD);
   else
      MPI_Recv(MatrixA, len, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
  //en büyük değer aranır.
   for(i = 0; i < len; i ++)
      if(max_num < MatrixA[i])
         max_num = MatrixA[i];
   //her görev kendi en büyük değerini 0 no'lu göreve iletir.
   if(rank == 0)
      for(i = 1; i < size; i++)
         MPI_Recv(&MatrixB[i], 1, MPI_INT,i, 1, MPI_COMM_WORLD, &status);
```

```
  else
      MPI_Send(&max_num, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
   //En büyük değerler arasındaki en büyük değer bulunur.
   if(rank == 0){
      for(i = 1; i < size; i ++)
         if(max_num < MatrixB[i])
            max_num = MatrixB[i];
      printInt("MaxNum: ", max_num);
      printEndl();
   }
   MPI_Finalize();
}
```

2. Koşut Matris Çarpma Uygulaması

Tez çalışmasında kullanılan Cannon Algoritması'yla geliştirilmiş koşut matris çarpma uygulamasının gerçekleştirimi aşağıda bulunmaktadır. Örnek MPI uygulamaları [11] arasından seçilen bu uygulama, geliştirilen yazılım kitaplığında çalışabilmesi için düzenlenmiştir.

```
__device__ void cannon_mm(int global_MatrixA[], int global_MtrixB[])
{
   __shared__ int irow,icol,jcol,iproc,jproc,index,Proc_Id,Root;
   __shared__ int A_Bloc_MatrixSize, B_Bloc_MatrixSize;
   __shared__ int NoofRows_A, NoofCols_A, NoofRows_B, NoofCols_B;
   __shared__ int NoofRows_BlocA, NoofCols_BlocA, NoofRows_BlocB;
   __shared__ int NoofCols_BlocB , Matrix_Size[4], ndims;
   __shared__ int Local_Index, Global_Row_Index, Global_Col_Index;
   __shared__ int source, destination, send_tag, recv_tag;
   __shared__ int Periods[2],Dimensions[2],Coordinates[2],Remain_dims[2];
   __shared__ MPI_Status status;
   __shared__ int size, rank, i, proc, row, col;
   __shared__ MPI_Comm cart_comm,row_comm, col_comm;

   __shared__ float Matrix_A[256], Matrix_B[256], Matrix_C[256];
   __shared__ float A_Bloc_Matrix[64], B_Bloc_Matrix[64],
                    C_Bloc_Matrix[64];
   __shared__ float MatA_array[256], MatB_array[256], MatC_array[256];

   MPI_Init ();
   MPI_Comm_size(MPI_COMM_WORLD, &(size));
   MPI_Comm_rank(MPI_COMM_WORLD, &(rank));

   proc = (int)sqrt((double) size);
   ndims=2;
   Root=0
   Dimensions[0] = Dimensions[1] = proc;

   //Wraparound mesh in both dimensions.
   Periods[0] = Periods[1] = 1;

   //Create Cartesian topology  in two dimnesions and  Cartesian
   //decomposition of the processes
   MPI_Cart_create(MPI_COMM_WORLD, ndims, Dimensions, Periods, 1,
                   &(cart_comm));
   MPI_Cart_coords(cart_comm, rank, ndims, Coordinates);
```

```
row = Coordinates[0];
col = Coordinates[1];

//Construction of row communicator and column communicators
//(use cartesian row and columne machanism to get
//Row/Col Communicators)
Remain_dims[0] = 0;
Remain_dims[1] = 1;

//The output communicator represents the column containing the process
MPI_Cart_sub(cart_comm, Remain_dims, &(row_comm));

Remain_dims[0] = 1;
Remain_dims[1] = 0;

//The output communicator represents the row containing the process
MPI_Cart_sub(cart_comm, Remain_dims, &(col_comm));

//Reading Input
if (rank == Root){
   Matrix_Size[0] = 16;
   Matrix_Size[1] = 16;
   Matrix_Size[2] = 16;
   Matrix_Size[3] = 16;

   for(i = 0; i < 256 ; i++){
      Matrix_A[i] = global_MatrixA[i];
      Matrix_B[i] = global_MatrixB[i];
   }
}

//Send Matrix Size to all processors
MPI_Barrier(cart_comm);
MPI_Bcast (Matrix_Size, 4, MPI_INT, 0, cart_comm);

NoofRows_A = Matrix_Size[0];
NoofCols_A = Matrix_Size[1];
NoofRows_B = Matrix_Size[2];
NoofCols_B = Matrix_Size[3];

NoofRows_BlocA = NoofRows_A / proc;
NoofCols_BlocA = NoofCols_A / proc;

NoofRows_BlocB = NoofRows_B / proc;
NoofCols_BlocB = NoofCols_B / proc;

A_Bloc_MatrixSize = NoofRows_BlocA * NoofCols_BlocA;
B_Bloc_MatrixSize = NoofRows_BlocB * NoofCols_BlocB;

//Rearrange the input matrices in one dim arrays by approriate order
if (rank == Root) {

   // Rearranging Matrix A
   for (iproc = 0; iproc < proc; iproc++){
      for (jproc = 0; jproc < proc; jproc++){
         Proc_Id = iproc * proc + jproc;
         for (irow = 0; irow < NoofRows_BlocA; irow++){
            Global_Row_Index = iproc * NoofRows_BlocA + irow;
            for (icol = 0; icol < NoofCols_BlocA; icol++){
               Local_Index  = (Proc_Id * A_Bloc_MatrixSize) +
                  (irow * NoofCols_BlocA) + icol;
```

```
                Global_Col_Index = jproc * NoofCols_BlocA + icol;
                MatA_array[Local_Index] = Matrix_A[Global_Row_Index *
                    NoofRows_A + Global_Col_Index];
             }
          }
       }
    }

    // Rearranging Matrix B
    for (iproc = 0; iproc < proc; iproc++){
       for (jproc = 0; jproc < proc; jproc++){
          Proc_Id = iproc * proc + jproc;
          for (irow = 0; irow < NoofRows_BlocB; irow++){
             Global_Row_Index = iproc * NoofRows_BlocB + irow;
             for (icol = 0; icol < NoofCols_BlocB; icol++){
                Local_Index = (Proc_Id * B_Bloc_MatrixSize) +
                   (irow * NoofCols_BlocB) + icol;
                Global_Col_Index = jproc * NoofCols_BlocB + icol;
                MatB_array[Local_Index] = Matrix_B[Global_Row_Index *
                    NoofRows_B+Global_Col_Index];
             }
          }
       }
    }
} //if loop ends here

MPI_Barrier(cart_comm);

// Scatter the Data  to all processes by MPI_SCATTER
MPI_Scatter (MatA_array, A_Bloc_MatrixSize, MPI_FLOAT, A_Bloc_Matrix ,
   A_Bloc_MatrixSize , MPI_FLOAT, 0, cart_comm);

MPI_Scatter (MatB_array, B_Bloc_MatrixSize, MPI_FLOAT, B_Bloc_Matrix,
   B_Bloc_MatrixSize, MPI_FLOAT, 0, cart_comm);

//Do initial arrangement of Matrices
if(row !=0){
   source   = (col + row) % proc;
   destination = (col + proc - row) % proc;
   recv_tag =0;
   send_tag = 0;
   MPI_Sendrecv_replace(A_Bloc_Matrix, A_Bloc_MatrixSize, MPI_FLOAT,
      destination, send_tag, source, recv_tag, row_comm, &status);
}
if(col !=0){
   source   = (row + col) % proc;
   destination = (row + proc - col) % proc;
   recv_tag =0;
   send_tag = 0;
   MPI_Sendrecv_replace(B_Bloc_Matrix, B_Bloc_MatrixSize, MPI_FLOAT,
      destination,send_tag, source, recv_tag, col_comm, &status);
}

for(index=0; index<NoofRows_BlocA*NoofCols_BlocB; index++)
   C_Bloc_Matrix[index] = 0;
```

```
// The main loop
send_tag = 0;
recv_tag = 0;
for(i=0; i<proc; i++){
   index=0;
   for(irow=0; irow<NoofRows_BlocA; irow++){
      for(icol=0; icol<NoofCols_BlocB; icol++){
         for(jcol=0; jcol<NoofCols_BlocA; jcol++){
            C_Bloc_Matrix[index] += A_Bloc_Matrix[irow *
               NoofCols_BlocA + jcol] * B_Bloc_Matrix[jcol *
               NoofCols_BlocB + icol];
         }
         index++;
      }
   }
   // Move Bloc of Matrix A by one position left with wraparound
   source   = (col + 1) % proc;
   destination = (col + proc - 1) % proc;
   MPI_Sendrecv_replace(A_Bloc_Matrix, A_Bloc_MatrixSize, MPI_FLOAT,
      destination,send_tag, source, recv_tag, row_comm, &status);

   // Move Bloc of Matrix B by one position upwards with wraparound
   source   = (row + 1) % proc;
   destination = (row + proc - 1) % proc;
   MPI_Sendrecv_replace(B_Bloc_Matrix, B_Bloc_MatrixSize, MPI_FLOAT,
      destination, send_tag, source, recv_tag, col_comm, &status);
}

MPI_Barrier(cart_comm);

// Gather output block matrices at processor 0
MPI_Gather (C_Bloc_Matrix, NoofRows_BlocA * NoofCols_BlocB, MPI_FLOAT,
   MatC_array,NoofRows_BlocA*NoofCols_BlocB, MPI_FLOAT, Root,
   cart_comm);

// Rearranging the output matrix in a array by approriate order
if (rank == Root) {
   for (iproc = 0; iproc < proc; iproc++){
      for (jproc = 0; jproc < proc; jproc++){
         Proc_Id = iproc * proc + jproc;
           for (irow = 0; irow < NoofRows_BlocA; irow++){
            Global_Row_Index = iproc * NoofRows_BlocA + irow;
            for (icol = 0; icol < NoofCols_BlocB; icol++){
               Local_Index = (Proc_Id * NoofRows_BlocA *
                   NoofCols_BlocB) + (irow * NoofCols_BlocB) + icol;
               Global_Col_Index = jproc * NoofCols_BlocB + icol;
               Matrix_C[Global_Row_Index*NoofRows_B+Global_Col_Index]=
                   MatC_array[Local_Index];
            }
         }
      }
   }

   printStr ("-------MATRIX MULTIPLICATION RESULTS ----------\n",
      "",0);
   printInt(" Processor ",rank);
   printInt(", Matrix A : Dimension ",NoofRows_A);
   printInt(" * ", NoofCols_A);
   printEndl();
```

```
      for(irow = 0; irow < NoofRows_A; irow++) {
         for(icol = 0; icol < NoofCols_A; icol++)
            printFloat(" ", Matrix_A[irow*NoofRows_A+icol]);
         printEndl();
      }
      printEndl();

      printInt(" Processor ",rank);
      printInt(", Matrix B : Dimension ",NoofRows_B);
      printInt(" * ", NoofCols_B);
      printEndl();
      for(irow = 0; irow < NoofRows_B; irow++){
         for(icol = 0; icol < NoofCols_B; icol++)
            printFloat(" ", Matrix_B[irow*NoofRows_B+icol]);
         printEndl();
      }
      printEndl();

      printInt(" Processor ",rank);
      printInt(", Matrix C : Dimension ",NoofRows_A);
      printInt(" * ", NoofCols_B);
      printEndl();
      for(irow = 0; irow < NoofRows_A; irow++){
         for(icol = 0; icol < NoofCols_B; icol++)
            printFloat(" ", Matrix_C[irow*NoofRows_A+icol]);
         printEndl();
      }
   } //if rank == Root

   MPI_Finalize();
}
```

KAYNAKLAR DİZİNİ

1. A. Grama, G. Karypis, V. Kumar ve A. Gupta “Introduction to Parallel Computing“, Addison Wesley, 2003
2. MPI: A Message-Passing Interface Standart, version 2.2, 2009
3. İnternet: Message Passing Interface Forum, 2009, MPI Resmi Forumu http://www.mpi-forum.org/
4. NVidia CUDA, Programming Guide, version 2.3, 2009
5. NVidia CUDA, Quickstart Guide, 2008
6. NVidia CUDA, Reference Manual, version 2.3, 2009
7. İnternet: NVidia CUDA, 2009, Resmi NVidia CUDA Sitesi, http://www.nvidia.com/object/cuda_home.html
8. NVIDIA GeForce GTX 200 GPU Architectural Overview, Technical Brief, 2008
9. D.Kanter, 2008, “NVIDIA’s GT200: Inside a Paralle Processor”, Real World Technologies Article
10. İnternet: NVIDIA GeForce GTX 280 GPU, 2009, Resmi NVIDIA Ürün Sitesi, http://www.nvidia.com/object/product_geforce_gtx_280_us.html
11. İnternet: MPI Examples, http://www.cse.iitd.ernet.in/~dheerajb/MPI/Document/hos_cont.html

ÖZGEÇMİŞ

Adı Soyadı : İbrahim TANRIVERDİ

Doğum Yeri : Amasya

Doğum Yılı : 1982

Medeni Hali : Bekar

Eğitim ve Akademik Durumu:

Lise	:	1997-2000	Malatya Fen Lisesi
Lisans	:	2000-2004	Hacettepe Üniversitesi Mühendislik Fakültesi Bilgisayar Mühendisliği Bölümü
Yabancı Dil	:	İngilizce	

İş Tecrübesi:

2004-2007	Araştırma Görevlisi, Hacettepe Üniversitesi Bilgisayar Mühendisliği Bölümü
2009-	Yazılım Mühendisi, HAVELSAN A.Ş.