Afyon Kocatepe Üniversitesi Fen ve Mühendislik Bilimleri Dergisi
Afyon Kocatepe University – Journal of Science and Engineering
https://dergipark.org.tr/tr/pub/akufemubid

Araştırma Makalesi / Research Article
e-ISSN: 2149-3367
DOI: https://doi.org/10.35414/akufemubid.1491786
AKÜ FEMÜBİD **25** (2025) 045201 (804-815)
AKU J. Sci. Eng. **25** (2025) 045201 (804-815)

Konvolüsyon İşlemi Üzerinde GPU Eniyileme Yöntemlerinin Etkisi

Makale Bilgisi / Article Info
Alındı/Received: 29.05.2024
Kabul/Accepted: 05.02.2025
Yayımlandı/Published: 04.08.2025

Impact of GPU Optimization Techniques on Convolution Operation

Kadir Emre ÖZER*, Sercan DEMİRCİ[2]

Ondokuz Mayıs Üniversitesi, Mühendislik Fakültesi, Bilgisayar Mühendisliği Bölümü, Samsun, Türkiye

Öz

Konvolüsyon, birçok görüntü işleme algoritmasında kullanılan önemli bir yöntemdir. Convolutional Neural Network (CNN) ve benzeri birçok sinir ağı yapısının giriş katmanlarında konvolüsyon işlemine yer vermesinden dolayı günümüzde daha da önemli bir duruma gelmiştir. Bu sinir ağı modellerinin çok büyük veri kümeleri üzerinde çalışmasından dolayı konvolüsyon işleminde gerçekleştirilebilecek küçük bir iyileştirme, genel başarımı büyük oranda etkileyebilecektir. Bu çalışmada, GPU tabanlı bir konvolüsyon algoritmasının verimliliğini artırmak amacıyla bir takım eniyileme yöntemlerinin etkisi incelenmiştir. Bu kapsamda, bellek erişimlerini azaltmaya yönelik iş parçacığı başına daha fazla veri işlemenin ve var olan bellek erişimlerinin yükünü azaltmak için adanmış belleklerin kullanımı üzerinde durulmuştur. Sonuç olarak, iş parçacığı başına değişken oranlarda veri işlemenin 2,33-2,45 kat arasında değişken hızlanma sağladığı, adanmış belleklerin kullanımının bu oranı 2,50-2,60 aralığına taşıdığı ölçülmüştür. Bunun yanı sıra, çıktı görüntüsünün belleğe yazılması sırasında bellek erişimlerinin daha büyük veri yapılarında birleştirilmesi (vektörleştirilmiş bellek erişimi) bu hız artışını 2,95-3,22 aralığına çıkarmıştır. Önerilen yöntemin en iyi durumda OpenCV ve ArrayFire kütüphane işlevlerine kıyasla sırasıyla 2,72-2,96 ve 4,23-4,68 kat arasında değişen oranlarda daha hızlı olduğu görülmüştür.

Anahtar Kelimeler: *GPGPU; CUDA; Konvolüsyon; İş Parçacığı Kalınlaştırma; Eniyileme; Paralel Programlama.*

Abstract

Convolution is an important method used in many image processing algorithms. It has become even more significant today due to its incorporation in the input layers of Convolutional Neural Networks (CNN) and many similar artificial neural network models. Since these neural network models operate on very large datasets, even a minor improvement in the convolution operation can significantly impact overall performance. In this study, the effects of several optimization methods aimed at enhancing the efficiency of a GPU-based convolution algorithm were examined. Specifically, the focus was on processing more data per thread to reduce memory accesses and utilizing dedicated memory to lower the cost of existing memory accesses. As a result, it was measured that processing varying amounts of data per thread provided a speedup ranging from 2.33x to 2.45x, while the use of dedicated memory increased this range to 2.50x-2.60x. Additionally, packaging memory accesses into larger data structures (vectorized memory access) during the writing of the output image to memory further boosted this speedup to a range of 2.95x-3.22x. The proposed method was found to be 2.72x-2.96x and 4.23x-4.68x faster, respectively, compared to OpenCV and ArrayFire library functions in the best case.

Keywords: *GPGPU; CUDA; Convolution; Thread Coarsening; Optimization; Parallel Programming.*

1. Giriş

Konvolüsyon, birçok görüntü işleme ve bilgisayarlı görü uygulamasında gürültü giderme, kenar bulma ve anlamlandırma gibi amaçlarla görüntüleri filtrelemek için kullanılan bir yöntemdir (İnt. Kyn-1). Convolutional Neural Network (İnt. Kyn-2) ve benzeri sinir ağı yapılarının konvolüsyon aracılığı ile ön işleme gerçekleştirmesi, bu işlemin günümüzdeki önemini artırmıştır. Sinir ağı yapılarının yüksek işlem gücü gereksinimi ve büyük veri kümeleri üzerinde çalışmasından dolayı konvolüsyon işleminde yapılacak ufak bir başarım artışı, yürütme süresinde gözle görülür ölçüde düşüşe neden olabilecektir.

Grafik İşleme Birimleri (GPU) birçok yürütme birimi ve yüksek bellek bant genişliği aracılığıyla veri-paralel işlemler gerçekleştirirken yüksek başarım sunan donanımlardır. Ayrıca, işlemcilere (CPU'lara) göre çok daha yüksek oranda aritmetik mantık birimi içermelerinden dolayı yoğun işlem gereksinimi duyulan durumlarda CPU'ya kıyasla daha yüksek başarım sağlamaktadırlar. Konvolüsyon, doğası gereği veri-paralel (her bir imgecik diğerlerinden bağımsız olarak) işlenebilir

*Sorumlu Yazar/Corresponding Author: Kadir Emre ÖZER e-posta/e-mail: emre.ozer@bil.omu.edu.tr

olmasıyla GPU gibi çok çekirdekli işleme birimlerinde daha verimli gerçekleştirilebilmektedir.

2000'li yıllarda NVIDIA'nın Compute Unified Device Architecture (CUDA) ortamını yayınlamasıyla GPU'lar grafik işlemlerinin yanı sıra genel amaçlı işlemler için de kullanılabilir duruma gelmiştir. Böylelikle, programcılar GPU kaynaklarını kullanarak çok iş parçacıklı çalışan uygulamalar geliştirebilmiştir. CUDA aracılığıyla bir algoritmayı GPU'ya uyarlamak göreceli olarak kolaylıkla gerçekleştirilebilir olsa da birtakım nedenlerden dolayı GPU'yu yüksek verim ile kullanmak her zaman kolay bir süreç değildir. Bunun başlıca nedeni bellek erişimlerinin aritmetik işlemlere göre oldukça yavaş olmasıdır. Bu durum, algoritmaların genellikle bellek darboğazından dolayı başarımının kısıtlanmasına yol açmaktadır. GPU'nun bellek yapısı, temel bir önbellek sıradüzeninin yanı sıra programcı tarafından yönetilebilir özelleşmiş bellek alanları içermesiyle doğru kullanım koşullarında daha yüksek bellek başarımı sağlayabilmektedir.

Konvolüsyon işlemi, yatay veya dikey eksendeki ardışık imgecikler üzerinde gerçekleştirildiğinde, birtakım komşu imgecikler her iki konvolüsyon işleminde ortak (örtüşen) olmaktadır. Bunun yanı sıra, kullanılan filtre matrisi de tüm konvolüsyon işlemleri için ortaktır. Bu ardışık imgeciklerin farklı iş parçacıkları tarafından işlendiği durumda, örtüşen komşu imgecikler ve filtre matrisi için birden çok kez bellek erişimi gerçekleştirilmektedir. Bu çalışma kapsamında, yatay eksende iş parçacığı başına işlenen imgecik sayısını artırarak, filtre matrisinin ve komşu imgeciklerin yeniden kullanımı yoluyla GPU tabanlı bir konvolüsyon algoritmasının bellek erişimlerinin azaltılması amaçlanmıştır. Buna ek olarak, iş yükü artırılmış iş parçacıklarının çıktı değerlerini sıralı bellek adreslerine yazması sırasında daha büyük veri yapılarının (vektörlerin) kullanılmasının, filtre matrisinin sabit bellekte (constant memory) saklanmasının ve imgecik kümelerinin yürütme sırasında paylaşımlı belleğe (shared memory) getirilmesinin bellek erişimleri üzerindeki etkisi de incelenmiştir.

1.1 Literatür

Literatürde yer alan birtakım çalışmalar, konvolüsyon işleminin GPU mimarisindeki başarımını artırmaya yönelik çeşitli yaklaşımlar ortaya koymuştur. Bu bölümde bu çalışmalar incelenmiştir.

Werkhoven vd. (2011) filtre matrisini sabit bellekte saklamayı, bölüt (thread block) tarafından kullanılacak verileri çalışma zamanında paylaşımlı belleğe getirmeyi ve bölüt başına işlenen imgecik sayısını artırmayı (tiling) denemişlerdir. İş parçacığı başına iş yükünün artırarak filtre matrisi ve örtüşen girdi sütunları için yapılan bellek erişimlerini azaltmayı amaçlamışlardır. Bunun yanı sıra, iş yükünü (tiling factor) aygıt sınırlarını gözeterek çalışma zamanında belirlemeyi (adaptive tiling) denemişlerdir. Sonuç olarak, 4,73 kata kadar hızlanma elde ettiklerini, iş yükünü çalışma zamanında belirlemenin başarımı %37 oranında artırdığını bildirmişlerdir.

Iandola vd. (2013) iş parçacığı başına değişken iş yükü (loop unrolling) ve veriyi kullanımından önce yazmaçlara taşımayı (prefetching) denemişlerdir. Bu kapsamda ayrı bellek alanlarından yazmaçlara veri taşınmasını incelemişlerdir. Tasarladıkları algoritmaları OpenCV, ArrayFire, NPP kütüphane işlevleri ile kıyaslamışlardır. Filtre boyutuna ve verinin hangi bellekten yazmaçlara aktarıldığına göre değişmekle birlikte ArrayFire kütüphane işlevine göre 1,2-4,5 kat arasında değişen hızlanma oranlarına ulaştıklarını ve verileri paylaşımlı bellek üzerinden yazmaçlara dağıtmanın bir iyileşme sağlamadığını bildirmişlerdir.

Chen vd. (2017) verilerin yeniden kullanımını (data reuse, register reuse) sağlayabilmek için ilgili görüntü kısmını paylaşımlı belleğe getirmeyi ve dikey eksende iş parçacığı başına değişken sayıda imgecik işlemeyi önermişlerdir. Paylaşımlı bellek kullanımı ile yatay eksende örtüşen sütunlar için evrensel bellek (global memory) erişimini ortadan kaldırmayı, dikey eksende örtüşen satırlar için ise yazmaçların yeniden kullanılmasını sağlamışlardır. Bunun yanı sıra, göreceli olarak küçük boyutundan dolayı filtre matrisini sabit bellekte saklamışlardır. Farklı mimarilerde, paylaşımlı bellek bankalarında çakışmaların (bank conflict) önlenmesi amacıyla, her bir iş parçacığının yatay eksende bir bankanın boyutuna eş değer sayıda (örneğin, banka boyutu 8 byte olan Kepler mimarisinde 2 float) veri işlemesini önermişlerdir. Tasarımlarının, cuDNN kütüphane işlevlerine göre 1x1 filtre matrisi boyutunda 6,16, 3x3 boyutunda 6,43, 5x5 boyutunda 2,90 ve ortalama 5,16 kat daha başarılı olduğunu bildirmişlerdir. 3x3 filtre matrisi boyutunda iş parçacığı başına yatay eksende işlenen verinin banka genişliği ile denk olmadığı durumda başarımın %19 düştüğünü bildirmişlerdir.

Lu vd. (2020), ardışık imgecikler üzerindeki konvolüsyon işlemleri sırasında meydana gelen tekrarlı bellek erişimlerini azaltmak amacıyla iki ayrı yöntem önermiştir. Dikey eksendeki ardışık konvolüsyon işlemlerinde, örtüşen satırları filtre matrisindeki birden çok satır ile çarparak birden çok çıktı verisi işlemeyi; yatay eksendeki ardışık konvolüsyon işlemlerinde örtüşen sütunları çözgü karıştırma (warp shuffle) buyrukları ile iş parçacıklarının yazmaçları arasında dağıtmışlardır. Sonuç olarak, tasarladıkları algoritmanın ArrayFire, NPP ve cuDNN

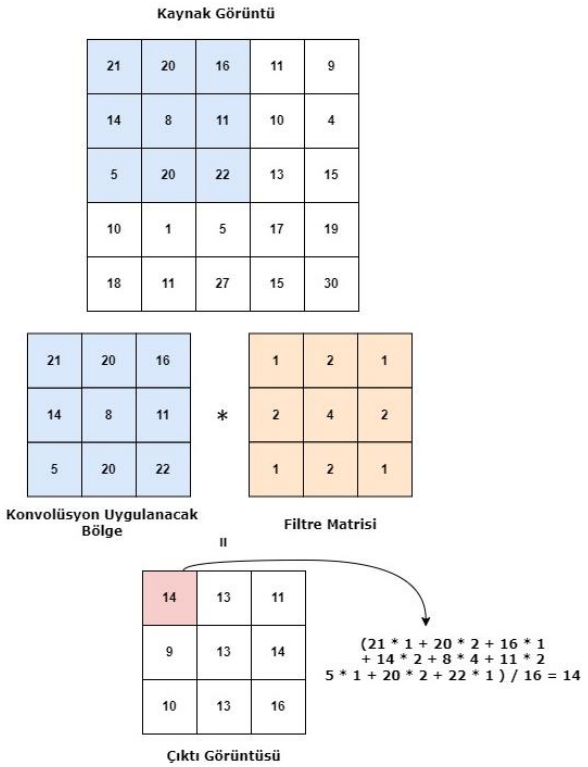
kütüphane işlevlerine kıyasla 2 kattan daha fazla hızlanma sağladığını bildirmişlerdir.

Topçu ve Öz (2022) filtre matrisini sabit bellekte saklamayı ve bölüt tarafından kullanılacak veri kümesini çalışma zamanında paylaşımlı belleğe getirmenin konvolüsyon işlemi üzerindeki etkisi incelemiştir. Bunun yanı sıra, CPU-GPU arasındaki veri aktarımlarını CUDA stream'ler ile gerçekleştirmeyi denemişlerdir. Tasarladıkları algoritmaları Kepler ve Turing mimarisine ait donanımlarda sınamışlardır. Bölüt tarafından kullanılacak veriyi paylaşımlı belleğe getirmenin Turing mimarisinde bir başarım artışı sağlamadığını, filtre matrisini sabit bellekte saklamanın önemli bir iyileştirme sağlamadığını ve CUDA stream'lerin CPU-GPU arasındaki veri aktarım gecikmelerini örttüğünü bildirmişlerdir. En iyi durumda 1,6 kata kadar hızlanma elde etmişlerdir.

2. Materyal ve Metot

2.1 Konvolüsyon

Konvolüsyon, matematiksel olarak iki sinyalin katlanması (evrilmesi) sonucu üçüncü bir sinyalin elde edilmesidir. Görüntü üzerinde düşünüldüğünde girdi görüntüsü ve filtre matrisinin katlanması sonucu filtrelenmiş bir görüntünün elde edilmesidir (İnt. Kyn-3, Smith 1997). Konvolüsyon, bir görüntü üzerinde Şekil 1 ve Denklem 1'deki gibi gerçekleştirilebilir.



Şekil 1. Görüntü Üzerinde Konvolüsyon İşlemi

$$r(i) = (s * k)(i,j) = \sum_n \sum_m s(i-n, j-m) \cdot k(n,m) \qquad (1)$$

r çıktı görüntüsüne, s girdi görüntüsüne, k filtre maskesine, n dikey eksende maske boyutuna, m yatay eksende maske boyutuna ve (i,j) koordinat noktalarına karşılık gelmektedir.

2.2 CUDA

CUDA, ilk sürümü 2006 yılında yayınlanmış, NVIDIA GPU'larının genel amaçlı programlanabilmesi için tasarlanan bir paralel programlama ortamıdır. CUDA ortamı, programcının çekirdek (kernel) olarak adlandırılan işlevler aracılığı ile GPU kaynaklarını kullanarak çok iş parçacıklı çalışan uygulamalar tasarlamasına olanak sağlar. Bir çekirdek çağrılacağı zaman ızgara (grid) ve bölüt (block) derinliği parametreleri kullanılarak değişken sayıda iş parçacığının çekirdek buyruklarını eşzamanlı olarak yürütmesi sağlanır.

CUDA ortamında iki ayrı aygıt ve her bir aygıtın kendisine ait belleği olduğu varsayılır. Böylece, programcı aynı kaynak kod içerisinde GPU kaynaklarını kullanarak paralel, CPU kaynaklarını kullanarak ise seri çalışan yordamlar tasarlayabilmektedir. Bu durumda, hangi aygıtta yürütme yapılacaksa kullanılacak verilerin ilgili aygıtın belleğinde bulunması gerekmektedir. Genellikle, bellek aktarım gecikmeleri çekirdek yürütme sürelerine göre çok daha yüksek olduğundan, uygulamaların başarımını belirlemede önemli bir etken oluşturmaktadır.

2.3 GPU Mimarisi

Günümüz GPU'ları genel olarak Streaming Multiprocessor (SM) olarak adlandırılan çok çekirdekli işleme birimleri, sabit bellek, yapılandırılabilir SRAM dizileri (L1 önbellek ve paylaşımlı bellek), L2 önbellek ve aygıt belleğinden (DRAM) oluşmaktadır (Hijma *et al*. 2023, Aamodt *et al.* 2018).

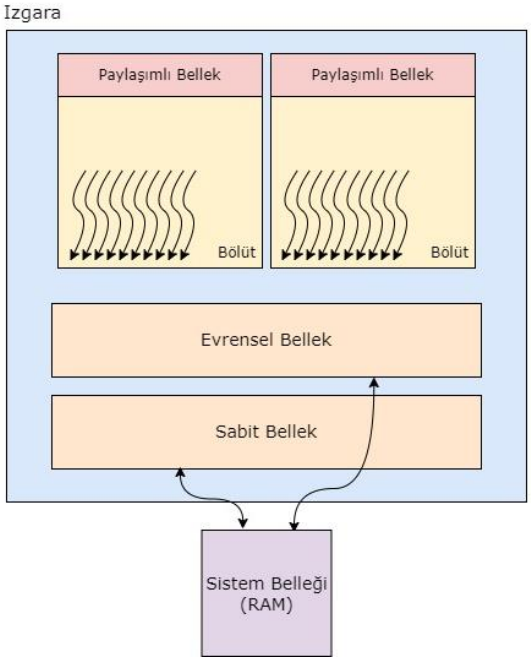
Bir çekirdek çağrımı yapıldığı zaman bölütler SM'ler üzerine dağıtılır. Devamında, bölüte ait çözgüler (warps) SM alt bölümlerine (SM sub-partition) dağıtılır ve yürütülmek için zamanlanır. Çözgü içerisindeki iş parçacıkları aynı anda aynı buyruğu işleyerek (Single Instruction Multiple Threads) yürütme işlemini gerçekleştirir. Bir SM üzerine genellikle birden çok sayıda çözgü dağıtıldığından dolayı bir çözgü yüksek gecikmeye neden olan bir buyruğu (örn. bellek erişimi) gerçekleştirirken bu sırada başka bir çözgü donanım kaynaklarını kullanabilir. Böylelikle gecikme süreleri örtülebilir.

Bir bellek buyruğu gerçekleştireceği zaman bu istek L1 önbellek, L2 önbellek ve DRAM aşamalarından sırayla geçer. Bu durumda karşılaşılabilecek en iyi durum, aranan verinin L1 önbellekte bulunmasıdır. Ancak, verinin L1 bellekte bulunup bulunmayacağı programcı tarafından yönetilebilir bir durum değildir. Bu duruma bir çözüm olarak, L1 önbelleğin bir bölümü (ya da L1 önbellekten ayrı

olarak) programcının yönetebileceği şekilde kullanıma sunulmaktadır. Sıklıkla kullanılacağı bilinen veri kümesi, programcı tarafından buraya konularak DRAM bağımlılığı büyük ölçüde azaltılabilir.

2.3.1 Adanmış Bellekler

CUDA, Şekil 2'de görüldüğü gibi programcının amacı doğrultusunda kullanabileceği türlü bellek alanları sunmaktadır.



Şekil 2. GPU Bellek Çeşitleri (Kirk and Wen-Mei 2016)

• Evrensel Bellek: Genel amaçlı bellek alanıdır. Çekirdeğin çalışması sırasında tüm iş parçacıkları için ortak olan verilerin saklanabileceği bölümdür. GPU'nun dış birimler ile veri alışverişi gerçekleştirmek için kullandığı bellek alanıdır.

• Paylaşımlı Bellek: Bölüt içindeki iş parçacıkları arasında veri paylaşımı ve iletişim işlemleri için kullanılabilen SRAM türünde bir bellek alanıdır. Günümüz mimarilerinde boyutu 16-96KB arasında değişmektedir.

• Sabit Bellek: Yürütme sırasında değişmeyecek verilerin saklanabileceği bir bellek bölümüdür. İçeriğinin değişmeyeceği bilindiğinden dolayı GPU'nun çok çekirdekli yapısında önbellek tutarsızlıklarına neden olmadan daha hızlı veri aktarımı sağlanabilir. Birçok mimaride boyutu 64 KB'dır.

2.4 Konvolüsyon İşleminin İyileştirilmesi

2.4.1 Temel Konvolüsyon Çekirdeği

Temel çekirdek, hiçbir eniyileme yöntemi kullanılmadan algoritmanın GPU üzerinde çalışabilir sürümü olarak nitelendirilebilir. Bu çekirdeğin çalışması sırasında her bir iş parçacığının iş yükü bir birim (bir imgeciği işleyecek şekilde) olacak şekilde tasarlanmıştır. Tüm bellek gereksinimleri evrensel bellek üzerinden karşılanmıştır. Algoritma 1'deki kod parçasında temel konvolüsyon çekirdeği gösterilmiştir.

Algoritma 1: Temel Konvolüsyon Çekirdeği

```
Input: int rows, int cols, unsigned char* input, unsigned char* output, float filter[3][3]
begin
  int ty = blockIdx.x * blockDim.x + threadIdx.x;
  int tx = blockIdx.y * blockDim.y + threadIdx.y;
  int new_val = 0;
  if ((tx > 0 && tx < rows - 1) && (ty > 0 && ty < cols - 1)) {
    new_val += filter[0][0] * input[(tx – 1) * cols + ty - 1];
    new_val += filter[0][1] * input[(tx – 1) * cols + ty];
    new_val += filter[0][2] * input[(tx - 1) * cols + ty + 1];
    new_val += filter[1][0] * input[tx * cols + ty - 1];
    new_val += filter[1][1] * input[tx * cols + ty];
    new_val += filter[1][2] * input[tx * cols + ty + 1];
    new_val += filter[2][0] * input[(tx + 1) * cols + ty - 1];
    new_val += filter[2][1] * input[(tx + 1) * cols + ty];
    new_val += filter[2][2] * input[(tx + 1) * cols + ty + 1];
    output[tx * cols + ty] = new_val;
  }
end
```

rows: görüntünün satır sayısı, cols: görüntünün sütun sayısı, input: girdi görüntüsü, output: çıktı görüntüsü, filter: filtre matrisi.

2.4.2 Sabit Bellek Kullanımı

Sabit bellek, yürütme sırasında sıklıkla kullanılan değişmez nitelikteki verilerin hızlı bir şekilde iş parçacıklarına dağıtılması (broadcasting) üzere özelleşmiştir. Çözgü içerisindeki iş parçacıkları aynı anda sabit belleğin aynı adresine erişme isteğinde bulunursa tek bir bellek erişimi ile bu veri tüm iş parçacıklarına sunulabilir (Brodtkorb *et al.* 2013). Ancak, farklı adreslere erişim istenirse, erişimler serileşir ve başarım kaybına yol açar (Sanders and Kandrot 2010, Kalaiselvi *et al.* 2017). Bu nedenle, aynı anda birçok iş parçacığına sunulması gereken bir verinin sabit bellekte depolanması olumlu bir etki sağlayabilecektir. Ayrıca, sabit bellek içerisinde bulunan bir veri yürütme boyunca değişmez olacağından dolayı önbellek tutarsızlıklarına neden olmayacağından hızlı bir şekilde önbelleklenebilir (Kirk and Wen-Mei 2016).

Temel konvolüsyon çekirdeğinde (Algoritma 1) iş parçacıklarının yürütme sırasında eşzamanlı olarak aynı filtre matrisi verisine erişme olasılığı yüksek olduğundan dolayı filtre matrisini sabit bellekte saklamak evrensel bellek erişimlerini azaltmanın yanı sıra bu verilere erişim hızını da artırabilecektir.

2.4.3 Paylaşımlı Bellek Kullanımı

Paylaşımlı bellek, programcı tarafından yönetilebilir bir önbellek olarak düşünülebilir. Yonga üstü bir bellek olmasından dolayı erişim hızı evrensel belleğe göre

oldukça hızlıdır (Kalaiselvi *et al.* 2017). Bu iki özellik açısından değerlendirildiğinde, sıklıkla kullanılan verilerin paylaşımlı belleğe getirilmesi evrensel bellek erişimlerini azaltarak başarım artışı sağlayabilecektir.

Temel konvolüsyon çekirdeğinde (Algoritma 1) her bir imgecik farklı iş parçacıkları tarafından birçok kez erişilmektedir. Bu durumu ortadan kaldırmak için veriler paylaşımlı belleğe getirilebilir. Böylece, evrensel bellek erişimlerinin azalmasıyla başarım artışı sağlanabilir.

Algoritma 2'deki kod parçasında temel konvolüsyon çekirdeğinin, girdi görüntüsünün paylaşımlı belleğe getirilmesi ile güncellenmiş sürümü gösterilmiştir.

Algoritma 2: Girdi Görüntüsünün Paylaşımlı Belleğe Kopyalanması

Input: int rows, int cols, unsigned char* input, unsigned char* output, float filter[3][3]

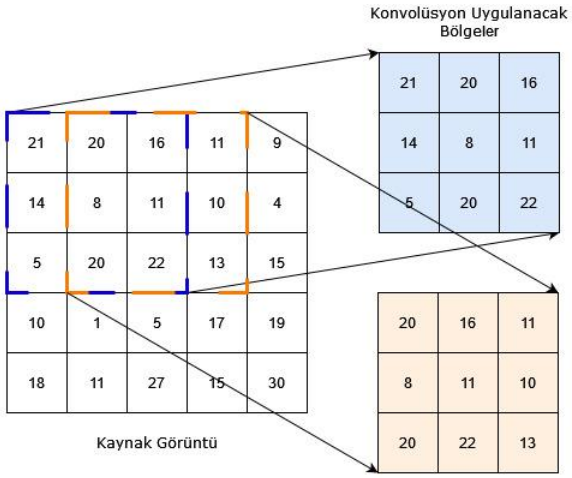
```
begin
  __shared__ unsigned char cache[34][34];
  int ty = blockIdx.x * blockDim.x + threadIdx.x;
  int tx = blockIdx.y * blockDim.y + threadIdx.y;
  unsigned int cy = threadIdx.x + 1;
  unsigned int cx = threadIdx.y + 1;
  int new_val = 0;
  cache[cx][cy] = input[tx * cols + ty];
  if (cx == 1) {
    cache[0][cy] = input[((tx – 1) * cols + ty)];
  }
  if (cx == 32) {
    cache[33][cy] = input[((tx + 1) * cols + ty)];
  }
  if (cy == 1) {
    cache[cx][0] = input[(tx * cols + ty - 1)];
  }
  if (cy == 32) {
    cache[cx][33] = input[(tx * cols + ty + 1)];
  }
  __syncthreads();
  if ((tx > 0 && tx < rows - 1) && (ty > 0 && ty < cols - 1)) {
    new_val += filter[0][0] * cache[cx - 1][cy - 1];
    new_val += filter[0][1] * cache[cx - 1][cy];
    new_val += filter[0][2] * cache[cx - 1][cy + 1];
    new_val += filter[1][0] * cache[cx][cy - 1];
    new_val += filter[1][1] * cache[cx][cy];
    new_val += filter[1][2] * cache[cx][cy + 1];
    new_val += filter[2][0] * cache[cx + 1][cy - 1];
    new_val += filter[2][1] * cache[cx + 1][cy];
    new_val += filter[2][2] * cache[cx + 1][cy + 1];
    output[tx * cols + ty] = new_val;
  }
end
```

rows: görüntünün satır sayısı, cols: görüntünün sütun sayısı, input: girdi görüntüsü, output: çıktı görüntüsü, filter: filtre matrisi, __syncthreads(): bölüt içerisindeki tüm iş parçacıklarının eşzamanlı olmasını sağlayan işlev.

2.4.4 İş Parçacığı Kalınlaştırma

İş parçacığı kalınlaştırma (kalınlaştırma), birkaç iş parçacığının yaptığı işi, tek bir iş parçacığı yapacak şekilde iş parçacığı buyruklarını yeniden düzenlemektir. Başka bir deyişle, aynı işlemi daha az iş parçacığı ile yapmaktır. Böylelikle daha önce her bir iş parçacığı tarafından yapılan birtakım gereksiz işlemler (dallanma, aritmetik vb.) azaltılabilir (Magni *et al.* 2013) ve ortak işlemler için yazmaçlar yeniden kullanılabilir (Yang *et al.* 2012).

Şekil 3'te görüldüğü üzere ardışık imgecikler üzerinde konvolüsyon uygulandığında birtakım imgeciklerin örtüştüğü görülmektedir. Bu durumda, filtre matrisinin ve örtüşen imgeciklerin yeniden kullanımı sağlamak amacıyla kalınlaştırma uygulanabilir. Böylelikle, örtüşen veriler için bellek erişimlerinin azalmasıyla başarım artışı sağlanabilir.



Şekil 3. Ardışık Veriler Üzerinde Konvolüsyon

Algoritma 3'te temel çekirdeğin yatay eksende iki kat kalınlaştırma ile güncellenmiş sürümü gösterilmiştir.

Algoritma 3: İş Parçacığı Yükünün Artırılması

Input: int rows, int cols, unsigned char* input, unsigned char* output, float filter[3][3]

```
begin
  int ty = (blockIdx.x * blockDim.x + threadIdx.x) * 2;
  int tx = blockIdx.y * blockDim.y + threadIdx.y;
  int new_val = 0;
  unsigned char frame[3][3];
  if ((tx > 0 && tx < rows - 1) && (ty > 0 && ty < cols - 1)) {
    frame[0][0] = input[(tx - 1) * cols + ty - 1];
    frame[0][1] = input[(tx - 1) * cols + ty];
    frame[0][2] = input[(tx - 1) * cols + ty + 1];
    frame[1][0] = input[tx * cols + ty - 1];
    frame[1][1] = input[tx * cols + ty];
    frame[1][2] = input[tx * cols + ty + 1];
    frame[2][0] = input[(tx + 1) * cols + ty - 1];
    frame[2][1] = input[(tx + 1) * cols + ty];
    frame[2][2] = input[(tx + 1) * cols + ty + 1];
```

```
    new_val += filter[0][0] * frame[0][0];
    new_val += filter[0][1] * frame[0][1];
    new_val += filter[0][2] * frame[0][2];
    new_val += filter[1][0] * frame[1][0];
    new_val += filter[1][1] * frame[1][1];
    new_val += filter[1][2] * frame[1][2];
    new_val += filter[2][0] * frame[2][0];
    new_val += filter[2][1] * frame[2][1];
    new_val += filter[2][2] * frame[2][2];
    output[(tx * cols + ty)] = new_val >> 4;
    for (int i = 1; i < 2; i++) {
      int _ty = ty + i;
      shift_left(frame);
      if (_ty < cols – 1) {
        frame[0][2] = input[(tx - 1) * cols + _ty + 1];
        frame[1][2] = input[tx * cols + _ty + 1];
        frame[2][2] = input[(tx + 1) * cols + _ty + 1];
        new_val += filter[0][0] * frame[0][0];
        new_val += filter[0][1] * frame[0][1];
        new_val += filter[0][2] * frame[0][2];
        new_val += filter[1][0] * frame[1][0];
        new_val += filter[1][1] * frame[1][1];
        new_val += filter[1][2] * frame[1][2];
        new_val += filter[2][0] * frame[2][0];
        new_val += filter[2][1] * frame[2][1];
        new_val += filter[2][2] * frame[2][2];
        output[(tx * cols + _ty)] = new_val;
      }
    }
  }
end
```

rows: görüntünün satır sayısı, cols: görüntünün sütun sayısı, input: girdi görüntüsü, output: çıktı görüntüsü, filter: filtre matrisi, frame: konvolüsyon için kullanılacak imgecik kümesi, shift_left(): matris sütunlarını bir birim sola kaydıran işlev.

2.4.5 Vektörleştirme

Vektörleştirme, skalar değişkenler (int, float vb.) yerine daha büyük veri yapıları olan vektörlerin (int2, float4 vb.) kullanılmasıdır. Vektör veri yapıları en yaygın olarak erişimlerin sayısını azaltmak (Braak *et al.* 2010, Chakrabarti *et al.* 2012) ve bant genişliğini (Choi *et al.* 2010) artırmak için bellek erişimlerinde kullanılmaktadır. Örnek olarak, bir iş parçacığı sıralı bellek konumlarından 4 kez unsigned char (1 byte) tipinde bir veri okuyorsa bu erişimler bir CUDA vektör değişkeni olan uchar4'e (4 byte) toplanıp tek bir bellek erişimi ile gerçekleştirilebilir. Böylece, aynı veri daha az buyruk ile erişilebilir.

Kalınlaştırılmış bir iş parçacığı tarafından gerçekleştirilen konvolüsyon işlemi düşünüldüğünde bellek erişimlerinin sırasız olmasından dolayı girdi imgeciklerinin okunması sırasında vektörleştirilmiş bellek erişimi gerçekleştirmek olası değildir. Ancak, çıktı imgecikleri sıralı bellek konumlarına yazılacağı için vektör veri yapıları kullanılabilir. Böylelikle, çıktı değerlerinin yazılması, daha az sayıda buyruk ile gerçekleştirilebilir.

3. Bulgular

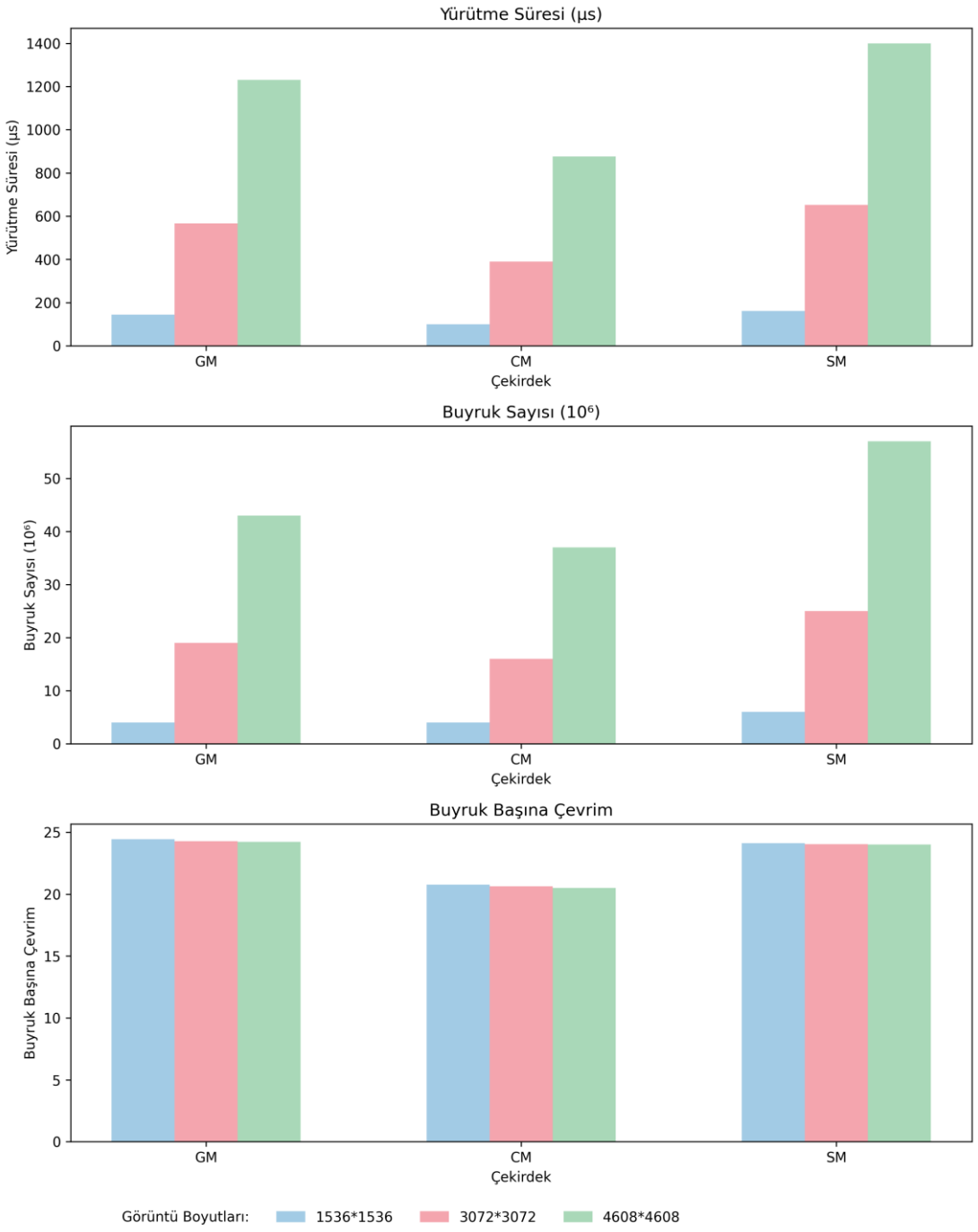
Çalışma kapsamında tasarlanan çekirdekler Çizelge 1'deki gibi isimlendirilmiştir. Bu çekirdekler, Windows 11 işletim sisteminde Çizelge 2'deki donanım üzerinde CUDA 12.6, OpenCV 4.10.0 ve ArrayFire 3.9.0 sürümleri ile çalıştırılmıştır. Çekirdeklere ait bulguların elde edilmesinde Nsight Compute (İnt. Kyn-4) aracı kullanılmıştır.

Çizelge 1. Çekirdeklerin İsimlendirilmesi

Çekirdek	Anlamı
GM	Filtre matrisinin evrensel bellekte saklandığı, bellek erişimlerinin skalar veriler ile gerçekleştirildiği çekirdek
CM	Filtre matrisinin sabit bellekte saklandığı, bellek erişimlerinin skalar veriler ile gerçekleştirildiği çekirdek
SM	İlgili verilerin paylaşımlı belleğe getirildiği, bellek erişimlerinin skalar veriler ile gerçekleştirildiği çekirdek
GM_CF8	GM çekirdeğinin 8 kat kalınlaştırıldığı çekirdek
GM_CF8_Vec	GM çekirdeğinin 8 kat kalınlaştırıldığı, çıktı görüntüsünün vektörler ile yazıldığı çekirdek
SM_CF12_Vec	SM çekirdeğinin 12 kat kalınlaştırıldığı, çıktı görüntüsünün ve paylaşımlı belleğe getirilen verilerin vektörler ile yazıldığı çekirdek
...	...

Çizelge 2. GTX 1660 TI (TU116) Donanımı İçin Teknik Özellikler

Özellik	Değer
Hesaplama Yeteneği	7.5
Bellek Saat Sıklığı	6001 MHz
GPU Saat Sıklığı	1590 MHz
Evrensel Bellek Boyutu	6442 MB
Bellek Veriyolu Genişliği	192-bit
L2 Önbellek Boyutu	1572864 byte
SM Sayısı	24
SM Başına En Fazla İş Parçacığı	1024
SM Başına En Fazla Bölüt	16
SM Başına En Fazla Çözgü	32
SM Başına Yazmaç	65536
SM Başına Paylaşımlı Bellek	65536 byte

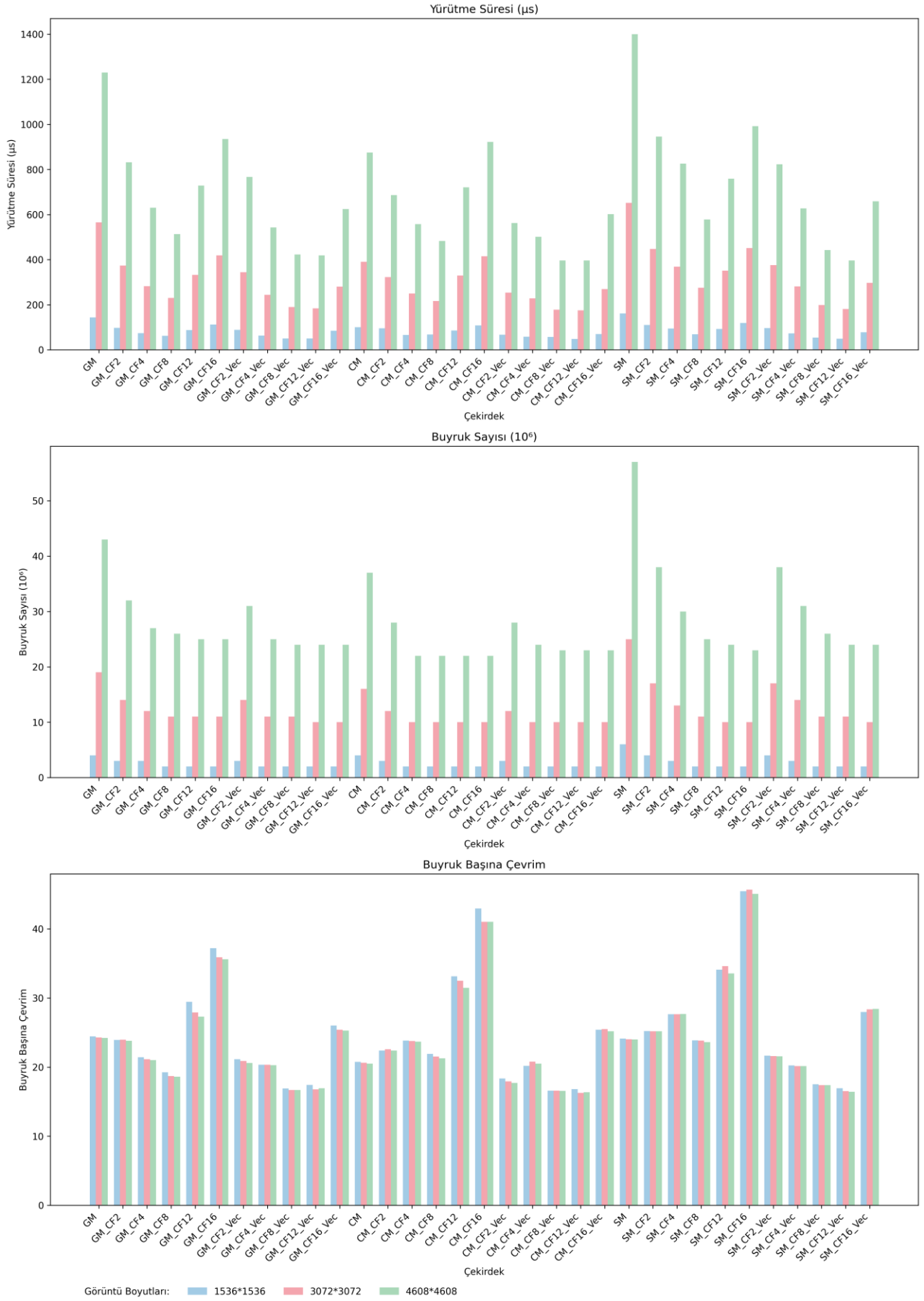


Şekil 4. Temel Çekirdeklerin Özellikleri

Çizelge 3. Çekirdeklerin Karşılaştırılması (3072*3072 Görüntü Boyutu İçin)

Ölçüt	GM	CM	SM
Yürütme Süresi (µs)	565,54	390,11	652,13
Toplam Buyruk Sayısı	19.464.192	16.515.072	25.657.344
Bellek Buyrukları	LDG: 5.308.416 STG: 294.912	LDG: 2.654.208 STG: 294.912	LDG: 3.610.752 STG: 294.912 LDS: 2.604.208 STS: 1.474.560
LDG Başına Bellek Sektörü	1,83	2,66	1,24
Buyruk Başına Ortalama Çevrim	24,28	20,63	24,02

LDG: Load From Global, STG: Store to Global, LDS: Load From Shared, STS: Store to Shared.



Şekil 5. Kalınlaştırma ve Vektörleştirmenin Etkisi

Çizelge 4. Kalınlaştırmanın Etkisi (3072*3072 Görüntü Boyutu İçin)

Çekirdek	Ölçüt	CF = 2	CF = 4	CF = 8	CF = 12	CF = 16
GM	Yürütme Süresi (µs)	373,66	282,30	230,27	331,94	418,82
	Evrensel Bellekten Yükleme Buyruğu (LDG)	3.096.576	1.990.656	1.437.696	1.253.376	1.161.216
	LDG Başına Bellek Sektörü	2,14	3,43	6,68	10,27	14,01
CM	Yürütme Süresi (µs)	322,85	249,86	216,80	329,76	415,17
	Evrensel Bellekten Yükleme Buyruğu (LDG)	1.769.472	1.327.104	1.105.920	1.032.192	995.328
	LDG Başına Bellek Sektörü	2,99	4,65	8,38	12,26	16,18
SM	Yürütme Süresi (µs)	447,10	369,15	275,62	350,98	451,33
	Evrensel Bellekten Yükleme Buyruğu (LDG)	1.990.272	1.179.264	773.760	638.592	571.008
	LDG Başına Bellek Sektörü	1,29	1,94	4,05	6,73	9,71

LDG: Load From Global, CF (Kalınlaştırma Faktörü): iş parçacığı başına iş yükü.

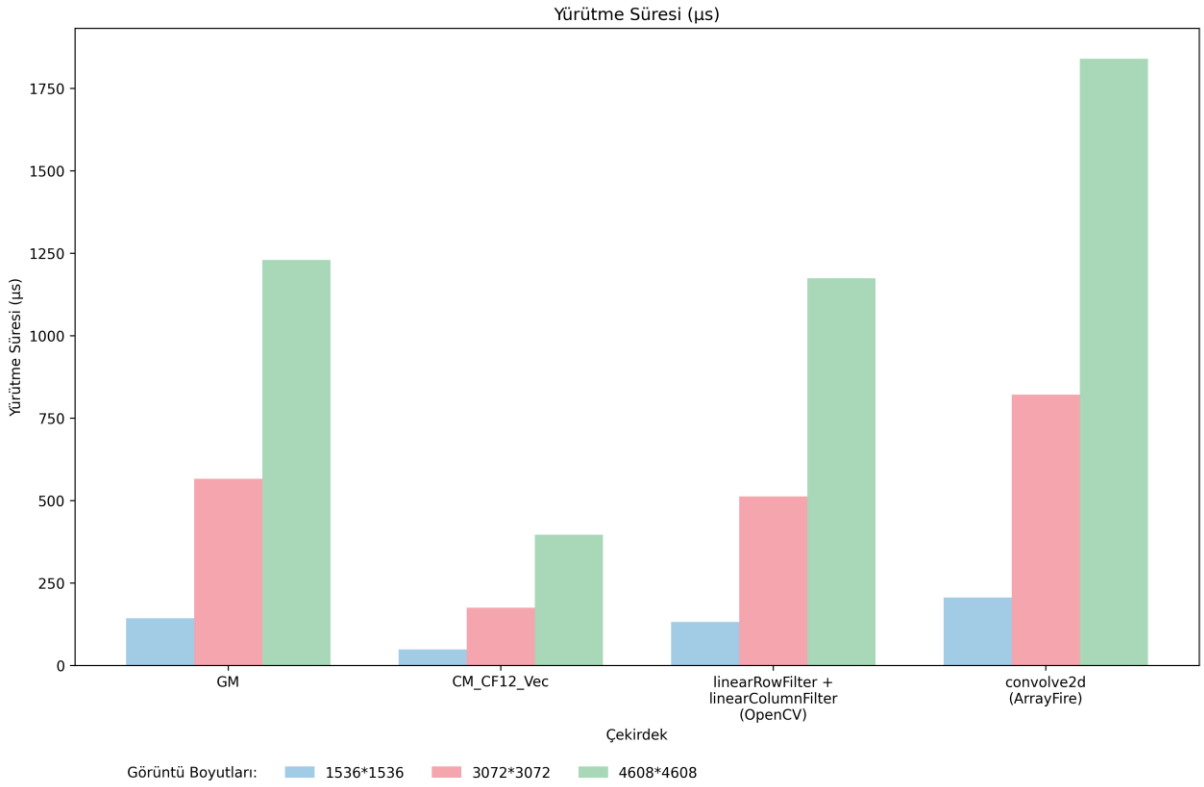
Çizelge 5. Vektörleştirmenin Etkisi (3072*3072 Görüntü Boyutu İçin)

Çekirdek	Ölçüt	CF = 2	CF = 4	CF = 8	CF = 12	CF = 16
GM_Vec	Yürütme Süresi (µs)	344,67	243,68	189,79	184,16	279,97
	Evrensel Belleğe Yazma Buyruğu (STG)	147.456	73.728	73.728	73.728	73.728
	STG Başına Bellek Sektörü	2,00	4,00	7,99	11,99	15,99
CM_Vec	Yürütme Süresi (µs)	254,24	227,87	178,53	175,23	269,89
	Evrensel Belleğe Yazma Buyruğu (STG)	147.456	73.728	73.728	73.728	73.728
	STG Başına Bellek Sektörü	2,00	4,00	7,99	11,99	15,99
SM_Vec	Yürütme Süresi (µs)	375,74	280,80	198,88	180,74	296,64
	Evrensel Belleğe Yazma Buyruğu (STG)	147.456	73.728	73.728	73.728	73.728
	STG Başına Bellek Sektörü	2,00	4,00	7,99	11,99	15,99

LDG: Load From Global, CF (Kalınlaştırma Faktörü): iş parçacığı başına iş yükü.

Çizelge 6. Literatürdeki Çalışmaların Karşılaştırılması

Çalışma	Yöntemler/Öneriler	Elde Edilen Başarım
Werkhoven vd. (2011)	- Filtre matrisini sabit bellekte depolama - Görüntüyü paylaşımlı belleğe kopyalama - İş parçacığı başına uyarlamalı iş yükü (adaptive tiling)	4,73 kata kadar hızlanma; uyarlı iş yükü artırımı ile %37 verim artışı
Iandola vd. (2013)	- İş parçacığı başına daha fazla iş yükü (loop unrolling) - Veriyi önceden getirme (prefetching)	ArrayFire kütüphane işlevine göre 1,2x ile 4,5x arasında değişken hızlanma
Chen vd. (2017)	- Filtre matrisini sabit bellekte depolama - Görüntüyü paylaşımlı belleğe kopyalama - İş parçacığı başına daha fazla iş yükü	1x1 filtre matrisi boyutunda cuDNN'e göre 6,16x; 3x3 boyutunda 6,43x; 5x5 boyutunda 2,90x hızlanma
Lu vd. (2020)	- Yatay eksende çözgü karıştırma (warp shuffle) - Dikey eksende örtüşen satırları birden fazla çıktı için hesaplama	ArrayFire, NPP ve cuDNN kütüphane işlevlerine kıyasla 2 kattan daha fazla hızlanma
Topçu ve Öz (2022)	- Filtre matrisini sabit bellekte depolama - Görüntüyü paylaşımlı belleğe kopyalama - CPU-GPU arası asenkron veri aktarımı (CUDA stream)	1,6 kata kadar hızlanma
Bu çalışma (2024)	- Filtre matrisini sabit bellekte depolama - Görüntüyü paylaşımlı belleğe kopyalama - İş parçacığı başına daha fazla iş yükü - Bellek erişimlerinin vektörleştirilmesi	3,22 kata kadar hızlanma, OpenCV kütüphane işlevine göre 2,72-2,96, ArrayFire kütüphane işlevine göre 4.23- 4.68 kat arasında değişken hızlanma



Şekil 6. Uygulanan Yöntemin ArrayFire ve OpenCV Kütüphane İşlevleri ile Karşılaştırılması

Çizelge 3 ve Şekil 4 incelendiğinde, filtre matrisinin sabit bellekte saklanmasının (CM), verilerin doğrudan kullanılabilirliğini sağlayarak (yazmaca taşınmadan) toplam buyruk sayısını azalttığı görülmüştür (19.464.192'den 16.515.072'ye). Bu yöntemde, filtre matrisi için artık evrensel bellek erişimleri yapılmadığından dolayı evrensel bellek erişimlerinin sayısı %50 oranında azalmıştır (LDG: 5.308.416'dan 2.654.208'e). Öte yandan, görüntü bölümlerinin paylaşımlı belleğe taşınması (SM), paylaşımlı belleğe yapılan okuma ve yazma işlemleri (LDS, STS) nedeniyle toplam buyruk sayısını artırmıştır (19.464.192'den 25.657.344'e). Bu yöntemde, örtüşen veriler paylaşımlı bellek üzerinden okunacağı için evrensel bellek erişimlerinin sayısı azalmıştır (LDG: 5.308.416'dan 3.610.752'ye).

Kalınlaştırma işlemi (Çizelge 4 ve Şekil 5), filtre matrisi verilerinin yeniden kullanımını sağlayarak ve örtüşen verilere yönelik bellek erişimlerini azaltarak evrensel bellek yüklemelerini önemli ölçüde azaltmıştır (LDG: 5.308.416'dan 1.161.216'ya). Ancak, kalınlaştırılmış iş parçacıklarının bellek erişimlerinin daha fazla bellek bölgesine dağılmasından dolayı birim erişim yükü artmıştır (LDG Başına Bellek Sektörü: 2,14'ten 14,01'e). Ayrıca, çıktı verilerinin ardışık bellek konumlarına yazılması sırasında vektör veri yapılarının kullanılması (Çizelge 5), belleğe yazma buyruklarını %75 oranında azaltmıştır (STG: 294.912'den 73.728'e) .

Uygulanan yöntemler, bilindik kütüphane işlevleri ile kıyaslandığında (Şekil 6) temel yöntemin (GM) OpenCV kütüphanesi işlevi ile yakın sonuçlar ürettiği ve uygulanan eniyileme yöntemlerinin (CM_CF12_Vec) yürütme süresini olumlu yönde büyük oranda etkilediği görülmüştür. Bunun yanı sıra, hem önerilen yöntemin hem de OpenCV kütüphane işlevinin ArrayFire kütüphane işlevine göre büyük oranda daha hızlı olduğu görülmüştür.

4. Sonuçlar ve Tartışma

Sonuç olarak, yürütme sırasında değişmez olarak nitelendirilebilecek verilerin sabit bellek aracılığıyla dağıtılmasının yalnızca evrensel bellek erişimlerini azaltmakla kalmayıp, aynı zamanda adres işlemlerinde kolaylık sağlayarak buyruk sayısını azalttığı görülmüştür. Bunun yanı sıra, örtüşen verilerin paylaşımlı belleğe getirilmesinin, tekrarlayan evrensel bellek erişimlerini azalttığı görülmüştür. Ancak, verilerin paylaşımlı belleğe getirilmesindeki gecikmenin başarım artışını kısmen gölgelediği görülmüştür. Kalınlaştırmanın, verilerin yeniden kullanımını sağlayarak toplam bellek erişim sayısını önemli ölçüde azalttığı, ancak bellek erişimlerinin daha çok bellek bölgesine yayılması nedeniyle birim erişim başına düşen yükü artırdığı görülmüştür. Bu durum, kalınlaştırma işlemindeki sınırlayıcı etken olarak değerlendirilmiştir. Ayrıca, kalınlaştırmanın etkisiyle iş parçacıklarının bellekte sıralı konumlara eriştiği durumlarda, bellek erişimlerinin vektör veri yapıları ile

gerçekleştirilmesinin başarımı artırdığı görülmüştür.

Bu çalışma kapsamında eniyileme yaklaşımı olarak belirtilen yöntemler, yürütme süresine etkileri açısından değerlendirilmiştir. Öte yandan, GPU'ların göreceli olarak yüksek güç tüketmeleri ve güç kısıtlı sistemlerde yaygın olarak kullanılması nedeniyle, bu yöntemlerin güç tüketimi üzerindeki etkileri de ayrıca incelenmesi gereken bir konudur. Gelecek çalışmalarda bu çalışmada uygulanan yöntemlerin güç tüketimi üzerindeki etkisinin incelenmesi düşünülmektedir.

Etik Standartlar Bildirgesi
Yazarlar tüm etik standartlara uyduklarını beyan ederler.

Yazarlık Katkı Beyanı
Yazar 1: Kaynaklar, Araştırma, Deney, Yazma, Görselleştirme – orijinal taslak
Yazar 2: Danışmanlık, Biçimsel analiz– orijinal taslak

Çıkar Çatışması Beyanı
Yazarların bu makalenin içeriğiyle ilgili olarak beyan edecekleri hiçbir çıkar çatışması yoktur.

Verilerin Kullanılabilirliği
Bu çalışma sırasında oluşturulan veya analiz edilen tüm veriler, yayınlanan bu makaleye dahil edilmiştir.

5. Kaynaklar

Aamodt, T.M., Fung, W.W.L., Rogers, T.G. and Martonosi, M., 2018. General-purpose Graphics Processor Architectures. Morgan & Claypool Publishers.

Brodtkorb, A.R., Hagen, T.R., Schulz, C. and Hasle, G., 2013. GPU Computing in discrete optimization. Part I: Introduction to the GPU. *EURO Journal on Transportation and Logistics*, 2, **1-2**, 129-157. https://doi.org/10.1007/s13676-013-0025-1

Brodtkorb, A.R., Hagen, T.R. and Sætra, M.L., 2013. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73, **1**, 4-13. https://doi.org/10.1016/j.jpdc.2012.04.003

Chakrabarti, G., Grover, V., Aarts, B., Kong, X., Kudlur, M., Lin, Y., Marathe, J., Murphy, M. and Wang, J. Z., 2012. CUDA: Compiling and optimizing for a GPU platform. *Procedia Computer Science*, 9, 1910-1919. https://doi.org/10.1016/j.procs.2012.04.209

Chen, X., Chen, J., Chen, D.Z. and Hu, X.S., 2017. Optimizing memory efficiency for convolution kernels on Kepler GPUs. *54th Annual Design Automation Conference 2017*. Texas, United States, 1-6. https://doi.org/10.48550/arXiv.1705.10591

Choi, J. W., Singh, A. and Vuduc, R.W., 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *ACM sigplan notices*, *45,* **5**, 115-126. https://doi.org/10.1145/1837853.1693471

Hijma, P., Heldens, S., Sclocco, A., Van Werkhoven, B. and Bal, H.E., 2023. Optimization techniques for GPU programming. *ACM Computing Surveys*, *55,* **11**, 1-81. https://doi.org/10.1145/3570638

Iandola, F.N., Sheffield, D., Anderson, M.J., Phothilimthana, P.M., and Keutzer, K., 2013. Communication-minimizing 2D convolution in GPU registers. *2013 IEEE International Conference on Image Processing*. Melbourne, Australia, 2116-2120. https://doi.org/10.1109/ICIP.2013.6738436

Kalaiselvi, T., Sriramakrishnan, P. and Somasundaram, K., 2017. Survey of using GPU CUDA programming model in medical image analysis. *Informatics in Medicine Unlocked*, *9*, 133-144. https://doi.org/10.1016/j.imu.2017.08.001

Kirk, D.B. and Wen-Mei, W.H., 2016. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann.

Lu, G., Zhang, W. and Wang, Z., 2020. *Optimizing GPU memory transactions for convolution operations*. 2020 IEEE International Conference on Cluster Computing (CLUSTER). Kobe, Japan, 399-403. https://doi.org/10.1109/CLUSTER49012.2020.00050

Magni, A., Dubach, C. and O'Boyle, M.F., 2013. A large-scale cross-architecture evaluation of thread-coarsening. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* Colorado, United States, 1-11. https://doi.org/10.1145/2503210.2503268

Topçu, B. and Öz, I., 2022. *Performance Evaluation of CUDA Optimizations for Convolution Operations.* Yüksek Başarımlı Hesaplama Konferansı (BAŞARIM) 2022. İstanbul, Turkey, 37.

Sanders, J. and Kandrot, E., 2010. CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional.

Smith, S.W., 1997. The Scientist and Engineer's Guide to Digital Signal Processing. California Technical Pub.

van den Braak, G. J., Mesman, B. and Corporaal, H. 2010. Compile-time GPU memory access optimizations. *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. Samos, Greece, 200-207. https://doi.org/10.1109/ICSAMOS.2010.5642066

Van Werkhoven, B., Maassen, J., and Seinstra, F.J. 2011. Optimizing convolution operations in cuda with adaptive tiling. *A4MMC'11: Proc. Workshop on Applications for Multi and Many Core Processors*. California, United States. https://doi.org/10.1016/j.future.2013.09.003

Yang, Y., Xiang, P., Kong, J., Mantor, M. and Zhou, H., 2012. A unified optimizing compiler framework for different GPGPU architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, *9,* **2**, 1-33. https://doi.org/10.1145/2207222.2207225

İnternet kaynakları

1-Ludwig, J., Image Convolution, https://web.pdx.edu/~jduh/courses/Archive/geog481w07/Students/Ludwig_ImageConvolution.pdf, (24.01.2025)

2- O'Shea, K., An Introduction to Convolutional Neural Networks, https://arxiv.org/abs/1511.08458, (24.01.2025)

3- Podlozhnyuk, Victor., Image Convolution with CUDA, https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_64_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf, (24.01.2025)

4- The User Guide for Nsight Compute, https://docs.nvidia.com/nsight-compute/NsightCompute, (24.01.2025)