SEYREK MATRİS-VEKTÖR ÇARPIMI İÇİN KOŞUT ZAMANDA ÖZELLEŞMİŞ KOD ÜRETİMİ VE DENEYSEL OPTİMİZASYON
RUNTIME PROGRAM GENERATION AND EMPIRICAL OPTIMIZATION FOR SPARSE MATRIX-VECTOR MULTIPLICATION

Barış Aktemur, Asım Yıldız

Bilgisayar Mühendisliği
Özyeğin Üniversitesi
baris.aktemur@ozyegin.edu.tr, asim.yildiz@ozu.edu.tr

Sam Kamin

Bilgisayar Bilimi
University of Illinois at Urbana-Champaign
kamin@cs.illinois.edu

ÖZETÇE

Bu çalışmada seyrek matris-vektör çarpımı için matris içeriğine göre özelleşmiş, yüksek hızlı program üretimi yapan bir kütüphane tasarımı anlatılmaktadır. Kütüphane sinyal işleme uygulamaları, bilimsel hesaplamalar, sonlu eleman analizi gibi mühendislik problemlerinde kullanılan büyük matrisler için kod üretimine olanak verir. Üretilen kod, pek çok seçenek arasından, deneysel optimizasyon yöntemiyle seçilir. Bu sayede koşumun gerçekleştiği makineye en uygun seçimin yapılması hedeflenir.

ABSTRACT

In this work we present the design of a library which generates high-performance sparse matrix-vector multiplication code that is specialized according to the contents of a given matrix. The library targets code generation in particular for matrices of large size as used in signal processing applications, scientific computation and finite element analysis. The generated code is selected out of many choices by empirical optimization to obtain the best code tuned for a particular machine.

1. GİRİŞ

Seyrek matris-vektör çarpımı sinyal işleme uygulamalarının temel işlemlerinden biridir. Temel işlemlerin performansı, uygulamaların hızını önemli ölçüde etkiler. Bu çalışmamızda seyrek matris-vektör çarpımı için yüksek hızlı kod üretimi yapan bir kütüphane yapısını anlatıyoruz. Kütüphanenin ayırdedici yanı (1) koşut zamanda program üretimi, (2) deneysel optimizasyon özelliklerinden ikisine birden sahip olmasıdır.

Koşut-zamanda program üretimi, ancak koşut-zamanda ortaya çıkan bilgiyi kullanarak programcı kontrolünde yapılan program üretimidir. Temel motivasyonu, üretilen programların veriye göre özelleştirilerek hızlandırılmasıdır. JavaScript, Lisp, Perl, PHP, Python ve daha pek çok programlama dilinde `eval` gibi bir işleç vasıtasıyla, dilin bir parçası olarak sunulmaktadır.

Bu çalışma Tübitak 1001 programı ve ABD National Science Foundation tarafından desteklenmektedir.

978-1-4673-0056-8/12/$26.00 ©2012 IEEE

Deneysel optimizasyon, diğer adıyla oto-ayarlama (autotuning), bir programın çeşitli sürümlerini koşumun hedeflendiği mimari üzerinde çalıştırarak test etmek ve en iyi sonucu veren sürümü seçmek yoluyla yapılan optimizasyon yöntemidir.

Koşut-zamanda program üretimi, hedeflenen makinenin özelliklerini görmezden geldiği için performans taşınabilirliği sorunları yaşar. Deneysel optimizasyon ise koşut-zaman parametrelerini dikkate almaz, seçimi sadece statik sürümler arasından yapar. Tasarladığımız kütüphane, koşut-zamanda farklı sürümler arasında deneysel optimizasyon ile seçim yapar ve seçilen programı üretir. Bu sayede her iki yöntemin üstün tarafları birleştirilmiş, zayıf yönleri kapatılmış olur.

Bildiride Bölüm 2 ve 3'te sırasıyla koşut-zamanda kod üretimi ile deneysel optimizasyonu anlatıyoruz. Bölüm 4'te kütüphanenin tasarımını ve gerçekleştirdiğimiz testleri sunuyor, Bölüm 5'te kapanışı yapıyoruz.

2. KOŞUT ZAMANDA KOD ÜRETİMİ

Koşut zamanda program üretimi, program parçalarının birleştirilerek bir araya getirilmesi ile gerçekleştirilir. Amacı, koşut-zaman verisine göre özelleşmiş, daha verimli çalışabilen kod üretmektir. Örneğin, aşağıda sol tarafta matematiksel gösterimi, sağ tarafta ise CSR (Condensed Sparse Row) gösterimi bulunan `M` seyrek matrisini ele alalım. CSR gösteriminde `vals` dizilimi matrisin sıfır olmayan elemanlarını tutar; `rows` dizilimi her bir satırın ilk elemanının `vals` dizilimindeki indeksini, `cols` dizilimi ise her bir elemanın sütun indeksini tutar. İndeksleme sıfırdan başlar.

$$\texttt{M:}\quad \begin{bmatrix} 0 & 1 & 2 & 0 & 0 \\ 0 & 0 & 3 & 4 & 0 \\ 5 & 0 & 0 & 6 & 7 \\ 8 & 0 & 9 & 0 & 0 \\ 0 & 10 & 0 & 11 & 0 \end{bmatrix}$$

`vals`: $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$
`rows`: $\{0, 2, 4, 7, 9, 11\}$
`cols`: $\{1, 2, 2, 3, 0, 3, 4, 0, 2, 1, 3\}$

CSR gösterimine göre genel-geçer matris-vektör çarpımı aşağıdaki şekilde yapılır. Burada `n`, matristeki satır sayısını belirtmektedir. Çarpan vektör `v`, sonuç vektörü ise `w`'dur.

```
void mult(int n, int *rows, int *cols, double *vals,
          double *v, double *w) {
  for(int i = 0; i < n; i++) {
    double y = w[i];
    for(int k = rows[i]; k < rows[i+1]; k++)
      y += vals[k] * v[cols[k]];
    w[i] = y;
  }
}
```

Çarpımı yapılacak matrisin içeriğini biliyorsak, bu kodu matris içeriğine göre özelleştirip hız kazanabiliriz. Örneğin, matris `M` için aşağıdaki gibi özelleşmiş bir fonksiyon yazılabilir. Görüldüğü gibi *for*-döngüleri tam olarak açılmış, tüm matris içeriği koda gömülmüştür. Bu nedenle sadece `v` ve `w` parametrelerine ihtiyaç vardır.

```
void mult(double *v, double *w) {
  w[0] = w[0] + 1 * v[1] + 2 * v[2];
  w[1] = w[1] + 3 * v[2] + 4 * v[3];
  w[2] = w[2] + 5 * v[0] + 6 * v[3] + 7 * v[4];
  w[3] = w[3] + 8 * v[0] + 9 * v[2];
  w[4] = w[4] + 10 * v[1] + 11 * v[3];
}
```

Belli bir matrisle başka pek çok (örn. yüzlerce) vektörü çarpacaksak, genel-geçer CSR çarpım fonksiyonu yerine özelleşmiş fonksiyon tercih edilir, çünkü bu fonksiyon döngü değişkenlerinden ve `rows`, `vals`, `cols` dizilimlerinden yapılan okumalardan arındırılmıştır. Özelleşmiş kodu üretmek için aşağıdakine benzer bir fonksiyon yazabiliriz. (Bu fonksiyonda + işleci aynı zamanda dizgi birleştirimi için kullanılmıştır.)

```
string gen(int n, int *rows, int *cols, double *vals){
  string result = "";
  for(int i = 0; i < n; i++) {
    result += "w[" + i + "] = w[" + i + "]";
    for(int k = rows[i]; k < rows[i+1]; k++)
      result += "+ " + vals[k] + "* v[" + cols[k] + "]";
    result += ";\n";
  }
  return result;
}
```

Koşut-zamanda `M` matrisi belli olduğunda yukarıdaki `gen` fonksiyonu çağrılır ve sonucunda elde edilecek kod derlenerek `M` ile çarpma işlemi yapmak için kullanılır. (Eğer `M`'nin içeriği koşut-zaman öncesinde belliyse kod üretimi erken de yapılabilir; koşut-zamana dek beklemeye gerek yoktur.) Ayrıca dikkat edilmelidir ki `M`'nin pek çok kez çarpma işleminde kullanılacağı varsayılmaktadır; `M` sadece birkaç kez kullanılacaksa kod üretimi yapmak anlamsızdır, çünkü koşut-zamanda derleme yapmak için de zaman harcanacaktır. Bir diğer önemli nokta da sadece seyrek matrisler için kod üretimi yaptığımızdır; gösterdiğimiz yöntem yoğun matrisler için uygun değildir.

2.1. Koşut-zaman kod üretimindeki sorun

Yukarıda bahsedildiği gibi döngü içeren genel-geçer kodu matrise özelleştirerek tam açılımlı hale getirmek mümkündür. Bu şekilde üretilen kodun, küçük matrisler için daha hızlı çalışacağı aşikardır. Fakat matris boyutu çok büyükse, üretilen kod da çok uzun olacaktır. Bu durumda üç nedenden ötürü performans sorunu yaşanmaktadır: (1) Derleyiciler çok uzun kaynak kodlarını optimize etmemektedir. (2) Uzun kod, yazmaç atama (*register allocation*) algoritmalarının başarısını zayıflatmaktadır. (3) Uzun kod, komut önbelleğine sığmamakta, önbellek okumalarında ıskalama (*instruction cache miss*) yaşanmaktadır.

Performansı etkileyen önemli bir unsur da veri erişiminde önbellek kullanımıdır [1, 3]. Çarpım sırasında matris elemanları ile sonuç vektörü `w`'nun elemanları ardışık sırayla erişilirken, çarpan vektör `v`'nin elemanları matrisin her bir satırı için yeniden okunmaktadır. Matris seyrek olduğu için, erişilen vektör elemanları arasında potansiyel olarak düzensizlik mevcuttur; bir işlem vektörün başındaki bir elemana erişirken, hemen sonraki işlem vektörün en sonlarındaki bir elemana erişebilir. Bu da önbelleğin verimsiz kullanımına neden olur.

Sinyal işleme, bilimsel hesaplama, sonlu eleman analizi gibi gerçek-dünya işlemlerinde kullanılan matrisler çoğu kez binlerce elemana sahip, büyük matrislerdir. Bu nedenle, naif bir şekilde tam açılım yaparak üretilen kod, yukarıda belirttiğimiz nedenlerden ötürü iyi performans verememekte, hatta kimi zaman genel-geçer CSR kodundan daha yavaş çalışmaktadır. Bu sorunun üstesinden gelmek için naif tam açılımlı kod üretiminden başka üretim yolları da denemeliyiz. Sıradaki bölümde, kullandığımız yöntemleri açıklıyoruz. Belli bir işlemci uzerinde hangi yöntemin hangi matris için en iyi sonucu vereceğini de deneysel optimizasyon ile bulmayı amaçlıyoruz.

2.2. Matris-vektör çarpımı kod üreticileri

Kullandığımız kod üretim yöntemleri ve aldıkları parametreler bu bölümde açıklanmaktadır.

2.2.1. Şablon

Matrislerde her bir satır, sıfır olmayan elemanlar açısından belli bir şablona sahiptir. Bir şablon, `A` matrisindeki satır i için $\{j - i \mid \mathtt{A}_{i,j} \neq 0\}$ ile tanımlanan kümedir. Örneğin, satır 0'daki sıfır olmayan eleman içeren sütunlar 0, 2, 3 ve 7 ise, bu satırın şablonu {0,2,3,7} olacaktır. Eğer satır 1 aynı sütunlarda sıfır olmayan elemanlara sahipse, şablonu {-1,1,2,6} olacaktır. Satır 7'nin sıfır olmayan elemanları 7,9,10 ve 14. sütunlarda olursa, şablonu satır 0 ile aynıdır. Bölüm 2'deki matris `M`'nin 0. ve 1. satırları ile 3. ve 4. satırları aynı şablonları paylaşmaktadır.

Her bir şablon $s = \{j_0, \ldots, j_k\}$ için açılım aşağıdaki gibi bir döngü olarak yapılır.

```
for(şablonu s olan her bir satır i){
  siradakiEleman = satır i elemanlarının başladığı yer
  w[i] += m[sıradakiEleman++] * v[i+j0]
        + ... + m[sıradakiEleman++] * v[i+jk];
}
```

2.2.2. Bantlı şablon[N]

Seyrek matrislerin çoğu bantlı yapıya sahiptir: elemanları köşegen etrafında toplanmıştır. Ancak az da olsa köşegenden uzakta elemanlar bulunabilir. Bu elemanlar şablon sayısının yüksek olmasına neden olur. Bantlı şablon üreticisi, köşegenin sadece N sütun uzağındaki elemanları içerecek şekilde şablon hesaplaması yaparak kod üretimini uygular. Böylece şablon sayısı ve üretilen kodun boyutu azaltılmış olur. Sınırların dışında kalan elemanlar için tam açılım gerçekleştirilir.

2.2.3. Açılım[M,N]

Önbelleğe erişimi eniyilemek için matris elemanlarına erişim satır-öncelikli düzen yerine bloklar halinde yapılabilir [1]; matris MxN'lik bloklara ayrılır ve çarpım bu bloklar gezilerek yapılır. Bu sayede matris elemanlarına erişimin soldan-sağa ve

yukarıdan aşağıya (satır-öncelikli sırada) olmak üzere düz bir sıra yerine küçük bloklar halinde olması, böylece vektör elemanlarına erişimin dar alanlarda yoğunlaşması sağlanır. Eğer bir bloğun tamamı dolu değilse, bellek erişim hatası almamak için matrise fazladan sıfırlar eklenir. Bu nedenle hız kazanımı ancak düzgün yapılı matrislerde olmaktadır [1]. Blok boyutu 2x3 olduğu zaman üretilen kod aşağıdaki gibidir. Bu kodda `bn`, bloklara ayrılmış matristeki blok satır sayısıdır; `b_rows` blokların satır indeksleri, `b_cols` sütun indeksleridir; `b_vals` matrisin değerlerine gereken yerlerde sıfır eklenmiş değer dizilimidir. Çarpma işlemi öncesinde verinin bu şekilde hazırlanması gerekir. `v` ve `w` yine çarpan ve sonuç vektörleridir.

```
void mult(int bn, int *b_rows, int *b_cols,
          double *b_vals, double *v, double *w) {
  double *y = w;
  int *b_row_start = b_rows;
  int *b_col_idx = b_cols;
  double *b_value = b_vals;
  for (int i = 0; i < bn; i++,y += 2) {
    double d0 = y[0];
    double d1 = y[1];
    for (int jj=b_row_start[i]; jj < b_row_start[i+1];
         jj++, b_col_idx++, b_value+=2*3) {
      d0 += b_value[0] * v[b_col_idx[0] + 0];
      d1 += b_value[3] * v[b_col_idx[0] + 0];
      d0 += b_value[1] * v[b_col_idx[0] + 1];
      d1 += b_value[4] * v[b_col_idx[0] + 1];
      d0 += b_value[2] * v[b_col_idx[0] + 2];
      d1 += b_value[5] * v[b_col_idx[0] + 2];
    }
    y[0] = d0;  y[1] = d1;
  }
}
```

2.2.4. Grup

Bu üretici eşit sayıda eleman içeren satırları gruplayarak her bir grup için çarpma işlemini gerçekleştiren kod üretir. Örneğin, 4 satırı 2 eleman, 1 satırı da 3 eleman içeren `M` matrisi için üretilen kod aşağıdaki gibidir. Çarpma işlemi yapmadan önce satır bilgilerini tutan `rows` diziliminin hazırlanması gerekmektedir.

```
void mult(int *rows, int *cols, double *vals,
          double *v, double *w) {
  int i, row;
  int a = 0, b = 0;
  for (i = 0; i < 4; i++) {
    row = rows[a]; a++;
    w[row] += vals[b]   * v[cols[b]]
            + vals[b+1] * v[cols[b+1]];
    b += 2;
  }
  row = rows[a]; a++;
  w[row] += vals[b]   * v[cols[b]]
          + vals[b+1] * v[cols[b+1]]
          + vals[b+2] * v[cols[b+2]];
  b += 3;
}
```

2.2.5. Blok[N]

Bu üretici, matrisi NxN boyutunda bloklara ayırıp her bir blok için çarpım işlemi yapan kod üretir. Sadece sıfır olmayan eleman içeren bloklar dikkate alınır. Bir blok içinde de sadece sıfır olmayan elemanlar işleme dahil edilir. Örneğin N parametresi 3 olarak verilirse, Bölüm 2'de verilen matris `M` için üretilecek kod aşağıdaki şekilde olacaktır. Fonksiyonların ismi `block_X_Y` şeklindedir; burada `X` ve `Y` bloğun sol üst köşesinin matris içindeki koordinatıdır. N değeri ∞ olduğu zaman tam açılımlı kod üretilmektedir. "Açılım[M,N]" üreticisine kıyasla, bu üretici matrise fazladan sıfır eklemediği ve matrisin elemanlarının değerlerini ve indekslerini koda entegre ettiği için daha avantajlıdır, ancak üretilen kod çok daha uzun olacaktır.

```
void mult(double *v, double *w) {
  block_0_0(v,w);
  block_0_3(v,w);
  block_3_0(v,w);
  block_3_3(v,w);
}
void block_0_0(double *v, double *w) {
  w[0] = w[0] + 1 * v[1] + 2 * v[2];
  w[1] = w[1] + 3 * v[2];
  w[2] = w[2] + 5 * v[0];
}
void block_0_3(double v[], double w[]) {
  w[1] = w[1] + 4 * v[3];
  w[2] = w[2] + 6 * v[3] + 7 * v[4];
}
void block_3_0(double v[], double w[]) {
  w[3] = w[3] + 8 * v[0] + 9 * v[2];
  w[4] = w[4] + 10 * v[1];
}
void block_3_3(double v[], double w[]) {
  w[4] = w[4] + 11 * v[3];
}
```

2.3. Hangi üretim yöntemini kullanmalı?

Görüldüğü gibi pek çok farklı üretim yöntemi bulunmaktadır; bu listeye yenileri de eklenebilir. Yöntemlerin değişik parametrelerle kullanıldığı da düşünüldüğünde, ortaya çok sayıda kod seçeneği çıkacaktır (örn. deneylerimizde bu sayı 53'tür.) Peki bu seçenekler arasından hangisi tercih edilmeli? Bu sorunun cevabını deneysel optimizasyon ile buluyoruz.

3. DENEYSEL OPTİMİZASYON

Modern bilgisayar mimarilerinin karmaşık olması, o mimari üzerinde bir programın hangi sürümünün daha hızlı çalışacağını bulmak için bir model ortaya çıkarmayı zorlaştırır. Mimariler arasındaki küçük farklar programların performansında tahminlerin ötesinde etki yapar; bu da performans taşınabilirliğini, yani her bilgisayar mimarisi üzerinde hızlı çalışabilen kod elde etmeyi güçleştirir. Deneysel optimizasyon bu soruna deney yoluyla çözüm bulmayı amaçlar. Programcı tarafından birbirine işlevsel olarak denk olan kodlar içeren bir aday kod kümesi belirlenir. Aday kodlar aynı işi yapan tamamen farklı algoritmalar olabileceği gibi, birbirinden sadece kullanılan parametreler arasındaki farklarla da ayrılabilirler. Bu nedenle aday kod sayısı oldukça yüksek, hatta potansiyel olarak sonsuz sayıda olabilir. Aday kodlar, önceden belirlenen ya da otomatik üretilen veriler kullanılarak koşumun hedeflendiği bilgisayar üzerinde kurulum-zamanı aşamasında test edilirler. Böylece, eldeki mimari üzerinde, hangi karakteristiğe sahip veri için hangi algoritmanın daha iyi performans gösterdiği bulunur. Koşut-zamana gelindiğinde gerçek verinin özelliklerine bakılır ve kurulum zamanındaki deneylere göre bu özellikler için en hızlı çalıştığı bilinen algoritma devreye sokulur.

FFTW [2] ve Sparsity [1] iyi bilinen deneysel optimizasyon kütüphanelerindendir. FFTW, hızlı Fourier dönüşümü için, Sparsity ise Bölüm 2.2'de verilen MxN açılımını kullanarak matris-vektör çarpımı için deneysel optimizasyon yapar. Son yıllardaki çalışmalar çok çekirdekli işlemciler ve GPU'lar gibi paralel mimariler üzerine odaklanmıştır [3, 4, 5].

4. KÜTÜPHANE TASARIMI

Tasarladığımız kütüphane, program üretimi ve deneysel optimizasyonu aşağıdaki şekilde birleştirir:

Kurulum zamanı (deneysel optimizasyon):

1. Önceden belirlenmiş bir matris kümesindeki her bir matris için tüm aday kodlar (53 adet) üretilir, çalıştırılır ve performansları kaydedilir.
2. Üretilmiş kodların önbellek simülasyonu yapılır. Simülasyon sonuçları ile performans değerleri eşleştirilerek hangi kodun nasıl performans gösterdiği öğrenilir. Bu bilgi koşut-zamanda üretilecek kod seçiminde kullanılır.

Koşut zaman (kod üretimi):

3. Özelleşmiş kod üretimi yapılmak istenen matris kütüphaneye varır. Matrisin içeriğine bakılarak üretilebilecek kodların önbellek davranışının ne olacağı kestirilir.
4. Önbellek kestiriminin ve 2. adımda öğrenilen bilginin ışığında, üretilebilecek kodların performansı tahmin edilir.
5. En hızlı olacağı tahmin edilen 3 aday kod üretilir, ardından çalıştırılır. Gerçek performansı en iyi olan seçilir. Matris için çarpım işleminde bundan sonra bu kod kullanılır.

Sunduğumuz tasarımın başarısını değerlendirmek için altı farklı işlemcili bilgisayarda test yaptık. Bunun için öncelikle mühendislik uygulamalarında kullanılan matrislerin tutulduğu *Matrix Market* sitesinden [6] seyrek matrisler seçtik. Bu seçimi yaparken matris boyutu ve elemanlarının dağılım düzenini mümkün olduğunca çeşitlendirmeye çalıştık. Elimizdeki tüm matrisler için özelleşmiş kod üretimi yaptık. Üretilen her bir kodu çalıştırdık, başarımlarını kaydettik. Böylece her bir makinede hangi matris için hangi kod en iyi sonucu veriyor bulmuş olduk. Elimizdeki matrislerden dördünü yukarıdaki tasarımın 1. adımındaki öğrenme safhasında kullanmak üzere seçtik. Diğer matrisleri 3. adımda gelen matrislermiş gibi düşünerek kütüphanenin tahmin başarısını değerlendirmede kullandık. Öğrenme safhası en uygun matris kümesini bulmak için elimizdeki tüm matrislerin dörtlü kombinasyonlarını denedik.

Önbellek simülasyonu için *valgrind* [7] programının *cachegrind* aracını kullandık. Cachegrind bize hem komut hem de veri önbelleğinin L1 ve L2 seviyelerindeki erişim ve ıska sayılarını verir. Cachegrind değerlerinin performansla doğrusal ilişki içinde olduğu yaklaşımıyla, bu değerleri performans ölçümlerine en küçük kareler yöntemi ile oturttuk. Bulunan katsayıları 4. adımdaki performans tahmini için kullandık. 3. adımdaki önbellek kestiriminde cachegrind'e gerek duymadan, sadece matrisin içeriğine bakarak önbellek simülasyonu yapan bir modülü kütüphaneye eklemekteyiz.

Testlerin sonuçları Tablo 1'de bulunmaktadır. Yer darlığından dolayı sadece iki bilgisayarda 9 matris için sonuç veriyoruz. Her bir bilgisayar için sol sütun en hızlı kodu üreten yöntemi, sağ sütun kütüphanemizin seçtiği yöntemi göstermektedir. Yöntem isimlerinin altındaki yüzdelik değer, karşılık gelen matris için o yöntemin ürettiği kodun koşum zamanının genel-geçer CSR kodunun zamanına oranıdır. Düşük oran daha hızlı kodu ifade eder. Seçtiğimiz yöntemin performans oranı, sol tarafındaki yöntemin oranına ne kadar yakınsa, kütüphane tasarımımız o kadar başarılı demektir.

	Bilgisayar 1		Bilgisayar 2	
Matris	En iyi kod	Seçilen kod	En iyi kod	Seçilen kod
add32	Grup %61.3	Blok[∞] %65.9	Şablon %42.2	Şablon %42.2
cavity02	Şablon %55.5	Bantlı[20] %84.3	Açılım[3,1] %78.7	Açılım[3,1] %78.7
cavity23	Bantlı[500] %70.6	Bantlı[500] %70.6	Şablon %75.2	Şablon %75.2
fidap001	Şablon %55.5	Bantlı[10] %58.9	Grup %85.5	Grup %85.5
fidap005	Blok[∞] %41.3	Blok[∞] %41.3	Blok[∞] %21.3	Blok[∞] %21.3
fidap031	Bantlı[200] %67.5	Bantlı[200] %67.5	Şablon %72.4	Şablon %72.4
fidap037	Grup %80.1	Grup %80.1	Bantlı[10] %89.4	Bantlı[10] %89.4
mhd3200a	Şablon %55.9	Şablon %55.9	Şablon %62.4	Şablon %62.4
memplus	Grup %77.8	Blok[∞] %80.8	Grup %61.9	Blok[∞] %61.9
mhd4800a	Şablon %59.9	Şablon %59.9	Şablon %62.5	Şablon %62.5
nnc666	Blok[∞] %40.6	Blok[∞] %40.6	Blok[∞] %51.3	Blok[∞] %51.3
orsreg1	Şablon %53.1	Şablon %53.1	Şablon %49.2	Şablon %49.2

Tablo 1: Her bir matris için en iyi performansı gösteren kod üretim yöntemleri ile kütüphanenin seçtiği yöntemlerin genel-geçer CSR'a oranla performansları. Düşük oran daha hızlı kodu ifade eder.

5. SONUÇ

Çalışmamızda seyrek matris-vektör çarpımı için koşut-zamanda özelleşmiş kod üretimi yapan, hangi kodun üretileceğine deneysel optimizasyon yöntemiyle karar veren bir kütüphane tasarımını anlattık. Testlerimiz bu tasarımın başarılı olduğunu göstermektedir. Çalışmamız kütüphanenin gerçeklenmesi, çeşitli uygulamalarda denenmesi ve üretilen kodların performanslarının detaylı incelenmesi ile devam etmektedir.

6. KAYNAKÇA

[1] E. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *Int. J. High Perform. Comput. Appl.*, vol. 18, pp. 135–158, 2004.

[2] M. Frigo and S. Johnson, "The design and implementation of FFTW3," *Proc. of the IEEE*, vol. 93(2), 216–231, 2005.

[3] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Supercomputing*, 2007, pp. 38:1–38:12.

[4] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Supercomputing*, 2009, pp. 18:1–18:11.

[5] P. Guo, H. Huang, Q. Chen, L. Wang, E. Lee, and P. Chen, "A model-driven partitioning and auto-tuning integrated framework for sparse matrix-vector multiplication on gpus," in *TeraGrid*, 2011, pp. 2:1–2:8.

[6] "Matrix market," http://math.nist.gov/MatrixMarket/.

[7] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI*, 2007, pp. 89–100.