Kısa Satırlı Matrislere Uygun, Düşük Ön İşleme Maliyetli Seyrek Matris-Vektör Çarpımı

Barış Aktemur
Bilgisayar Mühendisliği Bölümü
Özyeğin Üniversitesi
İstanbul, Türkiye
E-posta: baris.aktemur@ozyegin.edu.tr

***Özet*—Seyrek matris-vektör çarpımı (SpMV) pek çok mühendislik probleminin çözümünde kullanılan, yinelemeli çözücüler tarafından onlarca/yüzlerce defa çalıştırılan, yüksek başarım için kilit roldeki bir işlemdir. Genel olarak seyrek matrislerin tümü için yüksek başarım veren tek bir SpMV yöntemi yoktur. Kimi matris tipleri için oldukça iyi başarım alınmasını sağlayan SpMV yöntemleri olsa da, bu yöntemlerin kullanılabilmesi için yüksek ön işleme veya format çevrim maliyetleri ödemek gerekebilmektedir. Biz bu çalışmada CSRLenGoto adını verdiğimiz yeni bir SpMV yöntemini tanıtıyoruz. CSRLenGoto, en çok kullanılan seyrek matris saklama formatı olan CSR üzerinde çok düşük maliyetli bir ön işleme sonrasında kullanılabilmektedir. CSRLenGoto'nun faydasını en çok satır başına az sayıda (örneğin 1-6 arası) elemanı olan matrislerde görüyoruz. Yaptığımız deneylerde 1,75 kata kadar hızlanma tespit ettik. CSRLenGoto'nun hazırlık maliyeti çok düşük olduğundan, pek çok matris için, birkaç kullanım sonrasında maliyetin telafi edilip toplam zamanda kazanç aşamasına geçmek mümkün olmaktadır.**

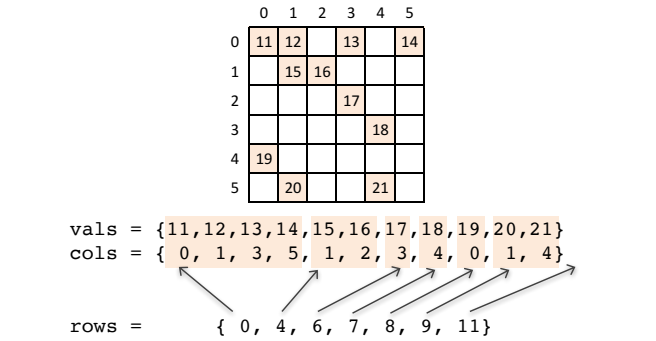
1. Giriş

Seyrek matrisler içlerinde yüksek oranda sıfır içeren matrislerdir. Seyrek matris-vektör çarpımı (*SpMV*) pek çok mühendislik probleminde ve bilimsel hesaplamada kullanılan yapıtaşı mahiyetinde bir işlemdir. Krylov problemleri, yinelemeli çözümleyiciler gibi alanlarda istenen sonuca yakınsamak için bir matris onlarca hatta yüzlerce defa SpMV işlemine girebilir. Bu nedenle başarım açısından kilit bir noktadır. Ancak SpMV'de elde edilen başarımın günümüz bilgisayarlarının işlem kapasitelerinin çok altında kaldığı bilinmektedir [1]. Bu sebeplerden ötürü SpMV işleminin hızlandırılması üzerine pek çok bilimsel çalışma yapılmıştır (detaylı taramalar için bkz. [2], [3]).

Seyrek matrisler, alandan tasarruf sağlayan formatlarda saklanırlar. Bu formatların belki de en popüler ve defakto standart konumunda olanı CSR (*Compressed Sparse Row* [4]) gösterimidir (bir sonraki bölümde detaylı inceleyeceğiz). CSR formatı önbellek uzunluğu, blok boyutu gibi parametreleri olmayan, işlemci mimarilerinin detaylarından bağımsız, tüm matris tiplerinin tutulabileceği bir gösterimdir. Ancak her matris için yüksek başarımlı SpMV yapılabilmesini sağlamaz. Bu sebeple, SpMV'nin hızlandırılması için, bazen sadece belli yapı sınıfındaki matrisleri hedef alan, yeni matris saklama formatları geliştirilebilmektedir [2]. Yeni bir format öne sürerken araştırmacıların zaman zaman gözardı ettiği, ancak mutlaka dikkate alınması gereken bir maliyet vardır: matrisin CSR gibi temel bir formattan yeni formata çevrilmesi. Bu çevrim, CSR gösteriminde SpMV yapmaya oranla çok uzun zaman alacaksa yeni formatın pratikte kullanım alanı sınırlı olacaktır. Çevrim maliyetinin süre yanında yer gereksinimi açısından da dikkate alınması lazımdır, çünkü matris SpMV sonrasında CSR formatı gerektiren başka işlemlerde kullanılacaksa yeni formatın yanında CSR gösteriminin de tutulması gerekecektir. Yani hafızada matrisin iki kopyasının bulunması gerekebilecektir. Bu nedenlerden ötürüdür ki son zamanlarda CSR gösterimini hiç değiştirmeden ya da düşük maliyetli ön işlemeler sonrası kullanan teknikler ortaya çıkmıştır [5], [6], [7], [8].

Çalışmamızda CSR'a çok benzeyen ve bu sayede koşut zamanda format çevrim maliyeti oldukça düşük olan bir gösterim kullanarak SpMV'nin hızlandırılmasını inceliyoruz. Maliyetin düşük olması sayesinde yinelemeli SpMV işlemleri gibi bağlamlarda maliyetin kısa sürede (örn. birkaç yineleme sonrasında) telafi edilip kâra geçilmesi mümkün olmaktadır. Yaklaşımımız döngülerin tamamen açılması üzerinde dayanmaktadır. Deneysel değerlendirmemiz gösteriyor ki bu yeni yöntem sayesinde ortalama satır sayısı düşük matrisler için ciddi miktarda hızlanma elde edilebilmektedir. Özellikle belirtmek gerekirse, çalışmamızın sunduğu katkılar şunlardır:

- *CSRLen* adını verdiğimiz seyrek matris saklama formatını tanıtıyoruz. CSR formatından CSRLen'e çok düşük maliyetle (ortalama 0,10–0,20 SpMV zamanına denk) çevrim yapılabilmektedir.
- CSR tabanlı referans koda göre 1,75 kata kadar hızlanma sağlanabilmektedir.
- Düşük maliyetli hazırlık safhası sayesinde birkaç yineleme sonrasında maliyetler telafi edilip toplam zamanda kâra geçmek mümkün olmaktadır.
- Önerdiğimiz yaklaşım özellikle satır başına ortalama eleman sayısı az olan (örneğin 1–6) seyrek matrisler için iyi sonuç vermektedir.




```
for (int i = 0; i < N; i++) {
  double sum = 0.0;
  for (int j = rows[i]; j < rows[i+1]; j++)
    sum += vals[j] * v[cols[j]];
  w[i] += sum;
}
```

Şekil 1. Örnek bir matris ile CSR gösterimi ve CSR için SpMV kodu.

- Kodumuzu, deneylerimizin başkaları tarafından da tekrarlanabilmesi amacıyla, kamuya açık olarak şu adreste sunuyoruz:
  `https://github.com/ozusrl/thundercat`

Çalışmamız SpMV'nin CPU üzerinde koşturulması üzerine odaklanmıştır; GPU bu aşamada kapsamımız dışındadır.

2. Temel Bilgiler

CSR (*Compressed Sparse Row*) formatında bir matris `vals`, `cols`, `rows` adlarını vereceğimiz üç adet dizi ile temsil edilir. `vals` dizisinde matrisin sıfır olmayan elemanları satır öncelikli sıralamaya göre saklanır. Bir diğer dizi olan `cols`'ta sıfır olmayan elemanların sütun dizini tutulur. Son olarak, `rows` dizisinde matrisin her bir satırının ilk elemanının `vals` ve `cols` dizisindeki dizini tutulur. Örnek bir matris, onun CSR gösterimi ve CSR gösterimi için SpMV kodu Şekil 1'de sunulmuştur. Kodda matrisin satır sayısı `N`'dir; girdi vektörü `v`, çıktı vektörü ise `w`'dur. `spmvCSR` fonksiyonu ile hesaplanan ifade şudur: $w \leftarrow w + M \cdot v$.

`spmvCSR` kodu zayıf başarım göstermesiyle bilinmektedir. Bunun pek çok nedeni vardır. Öncelikle, SpMV bellek-bağımlı bir hesaplamadır [9]. Girdi vektörü `v`'ye yapılan erişimler dolaylı ve düzensizdir; bu da önbelleğin kullanımında ve komut-seviyesindeki eşzamanlılıkta verimsizliğe yol açmaktadır [1], [10]. Seyrek matrisler genelde kısa satırlar içerdiği için `spmvCSR`'deki iç döngünün yineleme sayısı çok düşük olmaktadır. Bu nedenle döngü maliyetleri göreceli olarak yüksek kalmakta, ayrıca işlemcinin dallanma tahmini mekanizması faydalı olamamaktadır [11].

Peki yukarıda belirttiğimiz sorunları alt etmek için ne yapabiliriz? Deneyebileceğimiz bir teşebbüs, derleyicilerin de standart olarak uyguladığı eniyileme dönüşümlerinden biri olan **döngü açılımı**dır. Örneğin, `spmvCSR`'ın iç döngüsünü 4 kez açtığımızda Şekil 2'deki kodu elde ederiz. Burada, satır sayısının dördün tam katı olmadığı durumlarda arta kalan elemanları işlemek için ikinci bir iç

```
for (int i = 0; i < N; i++) {
  double sum = 0.0;
  int j = rows[i];
  for (; j < rows[i+1] - 3; j += 4) {
    sum += vals[j] * v[cols[j]];
    sum += vals[j+1] * v[cols[j+1]];
    sum += vals[j+2] * v[cols[j+2]];
    sum += vals[j+3] * v[cols[j+3]];
  }
  for (; j < rows[i+1]; j++)
    sum += vals[j] * v[cols[j]];
  w[i] += sum;
}
```

Şekil 2. Döngü açılımıyla elde edilen CSR_4 kodu.

```
      // Beklenen matris formatı: CSR
      int j = 0, i = 0, length;
      double sum;
      goto init;
L_5:  sum += vals[j] * v[cols[j]]; j++;
L_4:  sum += vals[j] * v[cols[j]]; j++;
L_3:  sum += vals[j] * v[cols[j]]; j++;
L_2:  sum += vals[j] * v[cols[j]]; j++;
L_1:  sum += vals[j] * v[cols[j]]; j++;
L_0:  w[i] += sum;
      i++;
init: if (i >= N) goto end;
      length = rows[i + 1] - rows[i];
      sum = 0.0;
      goto L_length; // kavramsal
end:  ;
```

Şekil 3. Azami satır uzunluğu 5 olduğu durumda *CSRGoto* kodu.

döngüye ihtiyacımız bulunmaktadır. `spmvCSR` kodundaki iç döngünün k kere açılmasıyla elde edilen kodu CSR_k olarak adlandıracağız. Buna göre `spmvCSR`'ın ilk haline atıfta bulunurken CSR_1 diyeceğiz.

CSR_k'deki ikinci iç döngünün varlığı, seyrek matrislerde bolca bulunan kısa satırlar için fazla ek yük getirmektedir ve malesef CSR_k ümit edilen hızlanmayı vermemektedir.

3. Yeni Yöntem Önerimiz

Bu bölümde çalışmamızın önerdiği SpMV yöntemi olan *CSRLenGoto*'yu anlatacağız. Bunun için öncelikle *CSRGoto* adını verdiğimiz bir ara yöntemi inceleyelim.

***CSRGoto*.** CSR_k yöntemindeki ikinci iç döngüden kurtulmak için CSR_1'deki iç döngüyü k kere değil, tamamen açma yolunu irdeleyelim. Döngü en fazla azami satır uzunluğu kadar yineleneceğinden, döngü içeriğini azami satır uzunluğu kadar açmak yeterli olacaktır. Sonrasında ise, her bir satır için kodun uygun noktasına zıplayabilmemiz gerekiyor. Bunu, kodda gerekli yerlere etiket koyup, bu etiketlere `goto` komutuyla atlayarak yapabiliriz. Bu şekilde işleyen SpMV yöntemine *CSRGoto* diyelim. Bu yöntemi *kavramsal* olarak Şekil 3'te gösteriyoruz. Belirtelim ki bu yöntemin doğru çalışması için matrisin satırlarının sabit uzunlukta olması gerekmez; satırlar farklı uzunluklara sahip olabilir, yeter ki kod azami satır uzunluğuna uygun yazılmış olsun.

```
     xor %eax, %eax                ; j ← 0
     xor %edx, %edx                ; i ← 0
     movsxd (%r11), %rcx           ; rcx ← rows[0]
     jmp init
L_5: ; sum += vals[j] * v[cols[j]]; j++;
     movslq (%r9,%rax,4), %rbx
     movsd (%r8,%rax,8), %xmm1
     incq %rax
     mulsd (%rdi,%rbx,8), %xmm1
     addsd %xmm1, %xmm0
L_4: ... ; L_5 ile aynı
L_3: ... ; L_5 ile aynı
L_2: ... ; L_5 ile aynı
L_1: ... ; L_5 ile aynı
L_0: addsd (%rsi,%rdx,8), %xmm0; sum ← sum + w[i]
     movsd %xmm0, (%rsi,%rdx,8); w[i] ← sum
     incq %rdx                  ; i ← i + 1
init:cmp %rdx, N                ; i ≥ N ise çık
     jge end
     movslq 4(%r11,%rdx,4),%rbx ; rbx ← rows[i+1]
     subq %rbx, %rcx            ; rcx ← -length
     imul 22, %rcx              ; rcx ← -length * delta
     leaq -45(%rip), %r10       ; r10 ← &&L_0
     addq %rcx, %r10            ; r10 ← &&L_0 + rcx
     leaq (%rbx), %rcx          ; rcx ← rbx
     xorps %xmm0, %xmm0         ; sum ← 0
     jmp *%r10                  ; goto L_length
end:
```

Şekil 4. Azami satır uzunluğu 5 olduğu durumda X86_64 mimarisine yönelik *CSRGoto* kodu.

Şekil 3'teki koda kavramsal dememizin nedeni koddaki son satırın geçerli bir C ifadesi olmamasıdır. Burada programlama açısından bir zorluk bulunmaktadır: Zıplanacak adres kodda nasıl ifade edilebilir? C dili standardında bulunmayan ancak GNU, vb. derleyicilerin desteklediği etiket adresleme operatörü (`&&`) ve *computed goto* komutu[1] şu şekilde kullanılabilir:

```
long delta = (&&L_0 - &&L_5) / 5;
goto *(void*)(&&L_0 - length * delta);
```

Fakat bu yaklaşımın çalışması için etiketlerin adreslerinin yeknesak bir şekilde derlenmiş olması gerekir. Deneylerimizde gördük ki derleyicilerin uyguladığı eniyilemeler nedeniyle böyle bir varsayım geçerli değildir. Bu sebeple kodun kaynak seviyesinde değil de çevirme (*assembly*) dili seviyesinde yazılmasını daha uygun bulduk. Çevirme kodunda hangi makina komutlarını kullanacağımıza karar vermek için öncelikle Şekil 3'te verdiğimiz koda benzer kaynak kodları Clang, GNU, icc derleyicileriyle derleyerek X86_64 çevirme koduna dönüştürdük (`-O3` eniyileme seviyesi ile). Derleyicilerin çıktısını incelediğimizde gördük ki *CSRGoto* yönteminin kilit noktasını oluşturan

```
sum += vals[j] * v[cols[j]]; j++;
```

kod satırı aşağıdakine benzer şekilde yerli koda çevriliyor:

```
movslq (%r9,%rax,4), %rbx ; rbx ← cols[j]
movsd (%r8,%rax,8), %xmm1 ; xmm1 ← vals[j]
incq %rax                 ; j++
mulsd (%rdi,%rbx,8), %xmm1; xmm1 ← xmm1 * v[rbx]
addsd %xmm1, %xmm0        ; sum ← sum + xmm1
```

Bu gözlemden hareketle *CSRGoto* yöntemi için Şekil 4'teki kodu yazabiliriz.

1. https://gcc.gnu.org/onlinedocs/gcc-7.1.0/gcc/Labels-as-Values.html

```
int *newRows = new int[N + 1];
for (int i = 0; i < N; i++) {
  int length = rows[i + 1] - rows[i];
  newRows[i] = -length * delta;
}
newRows[N] = β; // hedef 'end' etiketi
```

Şekil 5. CSR'den CSRLen formatına çevrim.

CSRGoto yöntemiyle ilgili önemli bir konu koddaki etiket sayısının en az matristeki azami satır uzunluğu kadar olması gerekliliğidir. Buna çözüm olarak önceden belli satır uzunluklarına göre kod yazılıp (örn. 100, 1000, vs.), matrisin azami satır uzunluğu bulunduktan sonra bu hazır kodlardan uygun olan seçilebilir. Bunu çevirme dili seviyesinde yapabilmek için derleyicilerin satırıcı çevirici (*inline assembler*) mekanizması ve makrolar kullanılabilir, fakat yazmaç isimlerini tutturmak ve derleyiciler arasında taşınabilirliği sağlamak zordur. Bu nedenle doğrudan çevirme dili seviyesinde programlama tercih edilebilir.

Önceden hazır kodlar bulundurmaya alternatif olarak, matrisin azami satır uzunluğu belirlendikten sonra uygun kod dinamik olarak da üretilebilir. Böylece gereğinden uzun kod kullanılmamış olur. Biz bu yaklaşımı seçtik. Kod üretimi oldukça hızlı yapıldığı için (Bölüm 4'te detaylandırıyoruz) bu yaklaşımın ek koşut-zaman maliyeti gözardı edilebilir seviyededir. Bunu çevirme kodu seviyesinde yapabilmek için koşut zamanda X86_64 yerli kodu üretilmesine yönelik arayüz sunan *asmjit*[2] kütüphanesini kullandık.

CSRLenGoto. *CSRGoto* yönteminde her bir satır için nereye zıplamak gerektiğini hesaplamak adına pek çok işlem yapılmaktadır (Şekil 4'te `init` ve `end` etiketleri arasında). Bu hesaplama özellikle kısa satırlarda belirgin şekilde başarım kaybına neden olabilir. Bu işlemlerden program sayacına (`%rip`) bağlı olan dışındakileri (örneğin `length` ve `delta`'ya bağlı olanları) önceden hesaplayabilir ve matris verisi olarak tutabiliriz. Bunun için CSR formatındaki `rows` dizisi üzerinden bir ön işleme geçişi yapar ve her bir satır için ne kadar zıplamak gerektiğini hesaplarız. Bu şekilde bulunan değerlerle oluşan CSR formatı çeşidine *CSRLen*, bu formatı kullanarak SpMV yapan yönteme de *CSRLenGoto* adını verdik. CSR formatından CSRLen'e dönüşüm sadece `rows` dizisinin işlenmesini içerir ve Şekil 5'teki gibidir. *CSRLenGoto* yönteminde ayrıca her satırda çıkış yapma kontrolünü de aradan kaldırabiliriz (`cmp` ve `jge` komutları). Bunun için `end` etiketine zıplamayı sağlayacak adres ekini `rows` dizisine $N+1$'inci eleman olarak koymak yeterlidir. Şekil 5'teki β ifadesi ile kastedilen budur. Bizim gerçekleştirmemizde β değeri 33 olmaktadır.

CSRLen gösteriminde `vals` ve `cols` dizilerinde değişiklik yoktur. Yeni `rows` dizisinin uzunluğu orijinali ile aynı olduğu için SpMV sırasında bellekten okunan matris verisinin boyutu aynı olacaktır. Şekil 5'te `newRows` adlı yeni bir dizi oluşturuyoruz. Eğer CSR gösterimine başka nedenlerle ihtiyaç duyulmayacaksa yeni bir dizi için

2. https://github.com/asmjit/asmjit

```
     xor %eax, %eax                ; j ← 0
     xor %edx, %edx                ; i ← 0
     jmp init
L_5: ; sum += vals[j] * v[cols[j]]; j++;
     movslq (%r9,%rax,4), %rbx
     movsd (%r8,%rax,8), %xmm1
     incq %rax
     mulsd (%rdi,%rbx,8), %xmm1
     addsd %xmm1, %xmm0
L_4: ... ; L_5 ile aynı
L_3: ... ; L_5 ile aynı
L_2: ... ; L_5 ile aynı
L_1: ... ; L_5 ile aynı
L_0: addsd (%rsi,%rdx,8), %xmm0; sum ← sum + w[i]
     movsd %xmm0, (%rsi,%rdx,8); w[i] ← sum
     incq %rdx                  ; i ← i + 1
init:xorps %xmm0, %xmm0         ; sum ← 0
     movslq (%r11,%rdx,4), %rbx; rbx ← rows[i]
     leaq -27(%rip), %r10       ; r10 ← &&L_0
     addq %rbx, %r10            ; r10 ← r10 + rbx
     jmp *%r10                  ; goto L_length
end:
```

Şekil 6. Azami satır uzunluğu 5 olduğu durumda X86_64 mimarisine yönelik *CSRLenGoto* kodu.

```
// Girdi: Azami satır uzunluğu (maxRowLength)
jit->xor_(eax, eax);
jit->xor_(edx, edx);
Label init = jit->newLabel();
jit->jmp(init);

for (int i = 0; i < maxRowLength; ++i) {
  jit->movsxd(rbx, ptr(r9, rax, 2));
  jit->movsd(xmm1, ptr(r8, rax, 3));
  jit->inc(rax);
  jit->mulsd(xmm1, ptr(rdi, rbx, 3));
  jit->addsd(xmm0, xmm1);
}
// L_0:
jit->addsd(xmm0, ptr(rsi, rdx, 3)); // 5 bayt
jit->movsd(ptr(rsi, rdx, 3), xmm0); // 5 bayt
jit->inc(rdx);                      // 3 bayt
// init:
jit->bind(init);
jit->xorps(xmm0, xmm0);             // 3 bayt
jit->movsxd(rbx, ptr(r11, rdx, 2)); // 4 bayt
jit->lea(r10, ptr(rip, -27));       // 7 bayt
// L_0 program sayacından 27 bayt geride
jit->add(r10, rbx);
jit->jmp(r10);
```

Şekil 7. *CSRLenGoto* kodu üreteci.

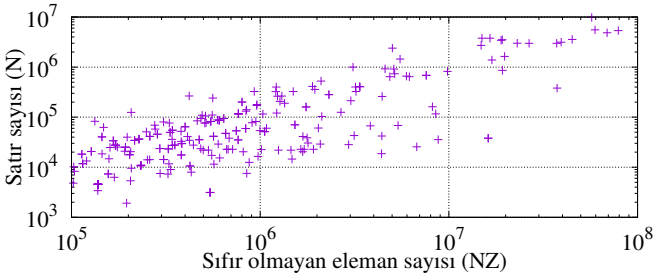
alan açmadan, `rows` dizisinin kendisi de kullanılabilir. Dönüşümün koşut zaman karmaşıklığı $O(\mathtt{N})$'dir.

CSRLenGoto kodu Şekil 4'teki *CSRGoto* koduna çok benzemektedir. Tek ana fark zıplanacak adresin hesaplanması yerine `rows` dizisinden okunmasıdır. Kod Şekil 6'da görülebilir. Belli bir azami satır uzunluğuna göre (`maxRowLength`) bu kodu üretmek için kullandığımız üretici program Şekil 7'de yer almaktadır. Buradaki `jit` isimli nesne *asmjit* kütüphanesinden gelen çeviricidir (*assembler*). Üretilen kodun uzunluğu ve dolayısıyla üretim maliyeti, azami satır uzunluğu ile doğru orantılıdır.

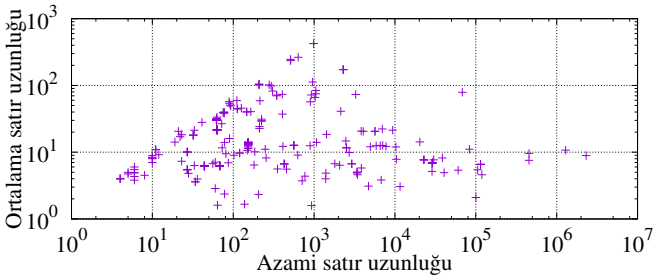
4. Değerlendirme

Bu bölümde CSR_k ve *CSRLenGoto* yöntemlerinin başarımlarını ölçüyor ve karşılaştırıyoruz.

Veri kümesi. Değerlendirmemizde gerçek problemlerde kullanılan matrislerden oluşan bir küme hazırladık. Bunun için University of Florida koleksiyonu [12] olarak bilinen matris deposundan, sıfır olmayan eleman sayısı 100 bin ile 80 milyon arasında, elemanları karmaşık sayı olmayan, kare boyutlu, simetrik olmayan matrisleri filtreledik. Bu bize 249 matris verdi. Matrislerin satır ve eleman sayılarının dağılımı Şekil 8'de, satır uzunlukları Şekil 9'da verilmiştir. Grafiklerde log-ölçeği kullanıldığına dikkat ediniz.



Şekil 8. Veri kümemizdeki 249 matrisin boyutları ve eleman sayıları.



Şekil 9. Veri kümemizdeki 249 matrisin satır uzunlukları.

Deney kurgusu. Testlerimizi X86_64 mimarili, biri Intel diğeri ise AMD işlemcili iki bilgisayar üzerinde gerçekleştirdik. Bilgisayarların özellikleri aşağıdaki gibidir:

İşlemci	Önbellek boyutu (Bayt)			Bellek (GB)
	L1 (I/D)	L2	L3	
Intel Xeon E5-2620 2.00 Ghz, 6-core	32K	256K	15M	16
AMD FX 8320 3.50 Ghz, 8-core	64K/16K	2M	8M	8

Bilgisayarlarımızın deneyler sırasında mümkün olduğunca yüksüz olmasına dikkat ettik. Kodları tek iş parçalı (*single threaded*) olarak çalıştırdık. Ölçüm alırken SpMV işlemini bir döngü içinde belli defalarca çalıştırıp toplam süreyi ölçtük ki ölçümlerdeki gürültü saf dışı kalsın. Bu tekrarlama sayısını matrisin boyutuna göre ve ölçüm yapılacak süre en az $\sim$1 saniye olacak şekilde belirledik. Ölçülen süreyi SpMV tekrarlama sayısına bölerek tek

bir SpMV için geçen süreyi hesapladık. Bu yolla her bir matris ve yöntem için 3 kere ölçüm yaptık, en küçük süreyi (en hızlı koşumu) kaydettik. CSR_k yöntemi için $k = \{1, 4, 8, 16, 32\}$ değerlerini kullandık. *CSRLenGoto* yöntemi için SpMV'de harcanan süre yanında format çevrimi ve kod üretimi için geçen süreleri de ölçtük.

Başarım ifadesi. Her bir bilgisayar için CSR_k yöntemleri arasında en iyi başarımı göstereni referans yöntem olarak belirledik. Bunun için her bir CSR_k sonucunu CSR_k sonuçları arasındaki en iyi değere göre normalize ettik. Intel üzerinde CSR_1 yöntemi istikrarlı bir şekilde diğer CSR_k'lerden daha iyi sonuçlar verdi. Bu nedenle Intel için referans yöntem olarak CSR_1'i seçtik. AMD'li bilgisayarda, tüm matrisler üzerinde ortalama olarak, CSR_1 yöntemi en iyi CSR_k'den %12, CSR_4 %4, CSR_8 %5, CSR_{16} %11, CSR_{32} ise %16 daha kötü sonuç verdi. Bu nedenle AMD için referans yöntem olarak CSR_4'ü belirledik.

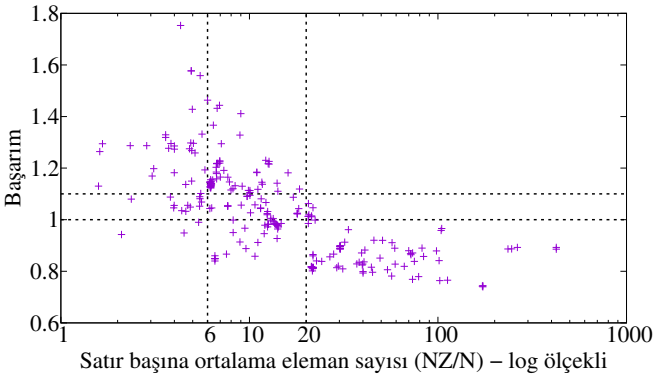
CSRLenGoto yönteminin başarımını referans yöntemin başarımına göreceli olarak ifade edeceğiz. Bunun için referans yöntemin aldığı süreyi, *CSRLenGoto*'nun aldığı süreye bölüyoruz. Bu oranın 1'den küçük olması *CSRLenGoto*'nun referans yönteme göre yavaşlamaya neden olduğu, 1'den büyük olması ise hızlanma sağladığı anlamına gelmektedir.

Görülen başarım. *CSRLenGoto*'nın başarım sonuçlarını incelerken ortalama satır uzunluğu ile alakalı olduğunu gözlemledik. Bu nedenle başarım sonuçlarını ortalama satır uzunluğu ile ilişkilendirerek sunuyoruz.

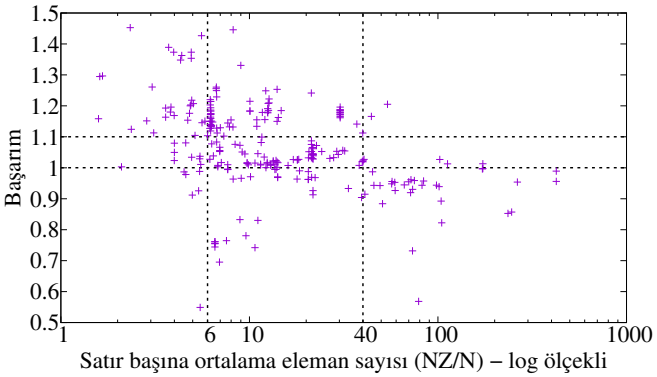
Şekil 10'da Intel işlemcili bilgisayarda elde edilen sonuçlar verilmiştir. Kısa satırlı matrisler için hızlanma görülmektedir. Örneğin, satır uzunluğu 6 veya daha küçük olan 45 matriste ortalama 1,22 kat hızlanma vardır. En yüksek başarım 1,75'tir. Sadece 3 matris için hızlanma kaydedilmemiştir. Matrislerin 30'unda (%66,7'si) en az 1,10 kat hızlanma alınmıştır. Uzun satırlı matrisler içinse durum tam tersidir. Örneğin, satır uzunluğu 20'den fazla olan 86 matris arasında sadece 7 tanesinde hızlanma görülmekte, bunların da hiçbirinde başarım 1,10'un üzerine çıkmamaktadır. Orta uzunlukta satırlara sahip matrislerde hem hızlanma hem de yavaşlama örnekleri vardır. Satır uzunluğu 6 ila 20 olan 118 matriste elde edilen ortalama başarım 1,09'dur. Matrislerin 31'inde (%26,3) yavaşlama varken, 62'sinde (%52,5) başarım 1,10 veya daha fazladır.

AMD işlemcili bilgisayarda da (Şekil 11) benzer durum mevcuttur. Kısa satırlı (≤ 6 eleman) matrislerin 39 tanesinde (%86,7) hızlanma görülürken, bunların 30 tanesinde başarım en az 1,10'dur. En yüksek başarım 1,45'tir. Ortalama satır uzunluğu 6 ile 40 arasında 166 matris vardır. Bunların 135 tanesinde (%81,3) hızlanma alınmıştır (68 tanesinde 1,10 veya üstünde başarım). Satır uzunluğu 40 üzerinde olan 38 matrisin ancak 8'inde (%16.7) hızlanma kaydedilmiştir.

Burada kısa/orta/uzun satır tanımını sonuçları çıplak gözle inceleyerek yaptık. İstatistiki incelemelere dayalı bir tanım da yapılabilir. Örn. *"matrislerinin en az %90'ı 1.00 başarım çizgisi üzerinde olan satır sayısı kısa satır eşiğidir"* vb. tanımlamalara gidilebilir. Bu tip bir sınıflandırmayı ve değerlendirmesini gelecek çalışma olarak bırakıyoruz.



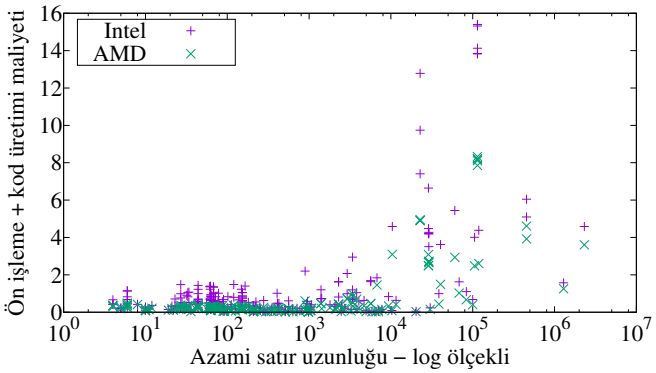
Şekil 10. Intel işlemcili bilgisayarda *CSRLenGoto*'nın başarımı ve ortalama satır uzunluğu ilişkisi.



Şekil 11. AMD işlemcili bilgisayarda *CSRLenGoto*'nın başarımı ve ortalama satır uzunluğu ilişkisi.

Çıkarım. Matristeki ortalama satır uzunluğu ile *CSRLenGoto*'nun başarımı arasında gördüğümüz ilişki şu şekilde kullanılabilir: Bir SpMV kütüphanesi, girdi olarak gelen matrisin ortalama satır uzunluğuna bakar. Bunun için sadece bir bölme işlemi yapmak yeterlidir (eleman sayısı / satır sayısı); matris verisindeki dizileri işlemeye gerek yoktur. Ortalama satır uzunluğu yüksek olarak tanımlanan eşik değerin üzerindeyse (örneğin; 20, 40, vs.) *CSRLenGoto*'nun fayda getirmeyeceğine kanaat getirilip doğrudan referans yöntem ile devam edilebilir. Matrisin ortalama satır uzunluğu düşük veya orta seviye ise, o zaman *CSRLenGoto* için ön işleme ve kod üretimi safhalarına geçilir. Girdi olarak gelen matris için bundan böyle üretilen *CSRLenGoto* kodu kullanılır. Kütüphane, *CSRLenGoto* kullanıldıkça başarımı gözlemeye devam edebilir. Eğer yavaşlama tespit edilirse, o matris için sonraki SpMV işlemlerinde referans yönteme dönüş yapılır.

Ön işleme ve kod üretim maliyeti. *CSRLenGoto* yöntemiyle SpMV yapmadan önce matrisin bir ön işlemeden geçirilerek azami satır uzunluğunun bulunması ve matris verisinin CSR formatından CSRLen formatına çevrilmesi gerektiğini hatırlayalım. Ön işleme maliyetini referans yöntemin aldığı zamana oranlayarak ölçtük. Böylece *"Ön işleme için harcanan zamanda referans yöntemi kullanarak kaç SpMV işlemi yapabilirdik?"* sorusuna yanıt vermiş olu-



Şekil 12. Ön işleme ve kod üretiminin referans yöntemle yapılan bir SpMV işlemi cinsinden maliyeti.

yoruz. Bu maliyet Intel işlemcili bilgisayarda 249 matris için ortalama 0,19, AMD işlemcili bilgisayarda 0,10'dur. Ölçtüğümüz en yüksek ön işleme maliyeti Intel'de 0,59, AMD'de 0,34 olmuştur.

Ön işleme sonrasında koşut zamanda kod üretimi yaparak bulduğumuz azami satır uzunluğuna göre *CSRLenGoto* kodu oluşturduğumuzu belirtmiştik. Şekil 12'de ön işleme ve kod üretiminin toplam masrafı, yine referans SpMV süresi cinsinden, azami satır uzunluğu ile ilişkilendirilerek verilmiştir. Beklendiği gibi azami satır uzunluğu çok yüksek olan matrislerde kod üretimi daha fazla vakit almaktadır. Bu tip durumlar için önceden hazırlanmış kodlar bir kod havuzundan çekilerek üretim yapmaktan kaçınmak mümkündür. Ortalama değerlere baktığımızda ise, ön işleme ve kod üretiminin toplam masrafı Intel'de 1,20, AMD'de 0,61 SpMV olmuştur. Görüldüğü gibi, genel olarak, *CSRLenGoto* yönteminin maliyeti oldukça düşük olmaktadır.

Kâra geçiş noktası. Yinelemeli çözümleyiciler gibi bağlamlarda bir matris için ilk hazırlık maliyeti bir kere ödendikten sonra aynı matris için *CSRLenGoto* yöntemi tekrar tekrar kullanılabilir. Eğer *CSRLenGoto* hızlanma getirdiyse, belli bir yinelemenin ardından toplam sürede kazanç elde etmeye başlarız. Bu, kâra geçiş eşik noktasıdır (*break-even point*). Yani bir başka deyişle, cevabını aradığımız soru şudur: *"SpMV işlemini en az kaç kere yineliyor olmalıyız ki CSRLenGoto yöntemi maliyetini telafi etsin ve başarım kazancı getirmeye başlasın?"*

Kâra geçiş noktalarını Tablo 1'de veriyoruz. Bu veriye sadece kısa ve orta uzunlukta satırlara sahip matrislerden hızlanma verenleri dahil ettik (Intel için 129, AMD için 174 matris). Matrislerin büyük çoğunluğu için daha ilk 5 veya 10 yineleme içinde kazanç elde etmeye başlıyoruz. Hızlanma elde edilemeyen matrisler için elbette bir kâra geçiş noktası bulunmamaktadır. Bu tip durumlarda, SpMV kütüphanesi ölçüm yapıp *CSRLenGoto*'nun yavaş kaldığını tespit ederek ilgili matris için sonraki SpMV işlemlerinde referans SpMV yöntemini kullanabilir. Bu durumda yapılan zarar, *CSRLenGoto* için harcanan ön işleme ve kod üretimi maliyetidir. Bu maliyetlerin de düşük olduğunu görmüştük.

Intel ve AMD işlemcilerde başarım ve kâra geçiş noktalarındaki farkın kaynağı hakkında kesin bir yargımız olmasa da bunun nedeni önbellek boyutları arasındaki fark ve mikro-mimari seviyesindeki farklar (örn. içermeli/dışlamalı önbellek – *inclusive/exclusive cache*) olabilir.

Tablo 1. KÂRA GEÇIŞ NOKTALARI.

	Matris sayısı	
Kara geçiş noktası aralığı	Intel	AMD
[0-5]	76	119
(5-10]	15	25
(10-15]	12	6
(15-20]	3	3
(20-25]	7	4
(25-50]	8	9
(50-100]	3	2
(100-200]	4	2
(200-300]	1	1
300+	-	3
Toplam	129	174

5. İlgili Çalışmalar

SpMV konusunda geçmişte ve günümüzde yoğun araştırmalar vardır. Yakın zamanda yayınlanan detaylı iki tarama konunun güzel bir özetini sunmaktadır [2], [3]. Bu çalışmadaki amacımız CSR formatına mümkün olduğunca yakın bir format kullanmak, böylece ön işleme maliyetlerini düşük tutmak idi. Benzer amaçlı güncel çalışmalara örnek olarak [5], [6], [7], [8] verilebilir. Bu çalışmalarda başarımı artırmak için odaklanılan konu SpMV'nin eşzamanlı hale getirilirken yük dengesinin daha iyi yapılmasıdır. Biz ise doğrudan SpMV'nin sıralı ardışık çalışma hızının artırılmasına odaklandık. Bu bağlamda, matrise CSR veya benzeri formatlara yönelik bölütlemeler (örneğin [5], [13], [14]) uygulandıktan sonra çekirdek işlem olarak *CSRLenGoto* kullanılarak yöntemimiz eşzamanlı hale getirilebilir.

SpMV'nin hızlandırılmasında temel yaklaşımımız döngü açılımıdır. Daha önce döngü açılımlarının hızlanma verebileceğini göstermiştik [15], ancak bunu matris verisini yeniden düzenleyen formatlar için yapmıştık. SpMV için döngü açılımını irdeleyen bir başka çalışma da LCSR adı verilen bir format kullanmaktadır [11]. Bu formatta matris elemanları yeniden sıralanmaktadır. Bu nedenle CSR formatından LCSR'a dönüşüm hem `vals` hem de `cols` dizilerinin yeniden oluşturulmasını gerektirir. *CSRLenGoto* yöntemimiz yeniden sıralanmış matris verisi ile de çalışır hale getirilebilir, ancak bu durum en baştan belirlediğimiz "düşük maliyet" kısıtına uymadığı için matris verisini yeniden düzenlemeye dayalı bir yaklaşımı tercih etmedik. Benzer şekilde, *CSRLenGoto*'nun `vals` dizisini sıkıştırarak veri trafiğini azaltan CSR-VI [16] gibi optimizasyonlarla birleştirilmesi de mümkündür; fakat bu sıkıştırma, maliyeti $O(\texttt{NZ})$ olan ve anahtarlama tablosuna (*hashtable*) erişim gerektiren bir ön işleme gerektirir. Bizim ön işlememizin maliyeti $O(\texttt{N})$ idi (Şekil 5).

6. Sonuç

CSR formatını çok düşük maliyetli bir işlemeden geçirdikten sonra kullanılabilen ve maliyetini kısa sürede telafi eden *CSRLenGoto* adlı bir SpMV yöntemi sunduk. Bu yöntem ortalama satır uzunluğu az olan matrisler için kazanç sağlamakta, uzun satırlı matrisler için genelde fayda getirmemektedir. Bu sayede, basit bir kontrolle yeni bir matris için kullanılıp kullanılmayacağına karar verilebilir. Orta uzunlukta satırları olan matrisler içinse yavaşlama veya hızlanma görülmesi mümkündür. Bu tip matrisler için performans tahmini yapmaya yarayacak bir modelleme gelecekte çalışmamızın devamı olarak incelenebilir.

Kaynaklar

[1] G. I. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Understanding the performance of sparse matrix-vector multiplication," in *16th Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing (PDP'08)*, 2008, pp. 283–292.

[2] D. Langr and P. Tvrdik, "Evaluation criteria for sparse matrix storage formats," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 2, pp. 428–440, Feb. 2016.

[3] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on GPGPUs," *ACM Trans. Math. Softw.*, vol. 43, no. 4, pp. 30:1–30:49, Jan. 2017.

[4] Y. Saad, *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[5] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC '16)*, 2016, pp. 58:1–58:12.

[6] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus for graph applications," in *Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC '14)*, 2014, pp. 781–792.

[7] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC '14)*, 2014, pp. 769–780.

[8] S. Ohshima, T. Katagiri, and M. Matsumoto, "Performance optimization of SpMV using CRS format by considering OpenMP scheduling on CPUs and MIC," in *IEEE Int. Symp. on Embedded Multicore/Manycore SoCs (MCSOC '14)*, 2014, pp. 253–260.

[9] W. Gropp, D. Kaushik, D. Keyes, and B. Smith, "Toward realistic performance bounds for implicit CFD codes," in *Parallel CFD'99*, A. Ecer *et al.*, Eds. Elsevier, 1999.

[10] J. Pichel, D. Heras, J. Cabaleiro, and F. Rivera, "Improving the locality of the sparse matrix-vector product on shared memory multiprocessors," in *12th Euromicro Conf. on Parallel, Distributed and Network-Based Processing*, Feb 2004, pp. 66–71.

[11] J. Mellor-Crummey and J. Garvin, "Optimizing sparse matrix-vector product computations using unroll and jam," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 2, pp. 225–236, 2004.

[12] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[13] A. N. Yzelman and D. Roose, "High-level strategies for parallel shared-memory sparse matrix-vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 116–125, Jan 2014.

[14] W. Yang, K. Li, Z. Mo, and K. Li, "Performance optimization using partitioned SpMV on GPUs and multicore CPUs," *IEEE Transactions on Computers*, vol. 64, no. 9, pp. 2623–2636, Sep. 2015.

[15] S. Kamin, M. J. Garzarán, B. Aktemur, D. Xu, B. Yılmaz, and Z. Chen, "Optimization by runtime specialization for sparse matrix-vector multiplication," in *Generative Programming: Concepts and Experiences (GPCE '14)*, 2014, pp. 93–102.

[16] K. Kourtis, G. Goumas, and N. Koziris, "Exploiting compression opportunities to improve spmxv performance on shared memory systems," *ACM Trans. Archit. Code Optim.*, vol. 7, no. 3, pp. 16:1–16:31, Dec. 2010.