https://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX

Matt Scarpino

2 Apr 2016 CPOL

რიცხვების ჩახრამუნება AVX -ით და AVX2-ით

საკითხები:

შესავალი

2003 წელს დაიწერა სტატია, რომელიც ახსნილია თუ როგორ უნდა განახორციელოთ SIMD (ერთი ინსტრუქცია, მრავალი მონაცემი) Intel– ის ნაკადისმაგვარი გაფართოებებით (Streaming SIMD Extensions - SSE). SSE არის Intel პროცესორების მიერ მხარდაჭერილი ინსტრუქციების ერთობლიობა, რომლებიც ასრულებენ მაღალსიჩქარიან ოპერაციებს მონაცემთა ვრცელ კრებულებზე.

2008 წელს Intel–მა შემოიტანა მაღალი წარმადობის ინსტრუქციების ახალი სიმრავლე, სახელწოდებით გაღრმავებული ვექტორული გაფართოებები (AVX). ისინი ასრულებენ ბევრ იგივე ოპერაციას, რასაც SSE ინსტრუქციები, მაგრამ მოქმედებენ მონაცემთა უფრო ვრცელ კრებულებზე უფრო დიდი სიჩქარით. ახლახან Intel- მა გამოაქვეყნა დამატებითი ინსტრუქციები AVX2 და AVX512 კომპლექტებში. ეს სტატია ეძღვნება AVX და AVX2 ინსტრუქციებზე წვდომას C -ის განსაკუთრებული ფუნქციების საშუალებით, სახელწოდებით შინაგანი (intrinsic) ფუნქციები.

AVX/AVX2 შინაგანი ფუნქციების მთელი ნაკრების განხილვის ნაცვლად, ამ სტატიაში ყურადღება გამახვილებულია მათემატიკურ გამოთვლებზე. კერძოდ, მიზანია კომპლექსური რიცხვების გამრავლება. ამ ოპერაციის შესასრულებლად AVX/AVX2– ით, საჭიროა სამი სახის შინაგანი ფუნქციები:

- შინაგანი ფუნქციების ინიციალიზება
- არითმეტიკული შინაგანი ფუნქციები
- გადანაცვლება/აღრევის შინაგანი ფუნქციები

ეს სტატია აღწერს შინაგანს ფუნქციებს თოეულ კატეგორიაში და გვიხსნის თუ როგორ გამოიყენებიან ისინი კოდში. ბოლოში ნაჩვენებია თუ როგორ მივუყენოთ ეს ფუნქციები კომპლექსური რიცხვების გადამრავლების ამოცანას.

მნიშვნელოვანია გვესმოდეს განსხვავება პროცესორის ინსტრუქციასა და შინაგან ფუნქციას შორის. AVX ინსტრუქცია არის აწყობის (assembly) ბრძანება, რომელიც ასრულებს განუყოფელ ოპერაციას. მაგალითად, AVX ინსტრუქცია `vaddps` კრებს ორ ოპერანდს და შედეგს ათავსებს მესამეში.

C / C ++ - ში მოქმედების შესასრულებლად, შინაგანი ფუნქცია `_mm256_add_ps()` უშუალოდ `vaddps`-ში აისახება, აერთიანებს აწყობის (assembly) მოქმედებას მაღალი დონის ენის ფუნქციის მოხერხებულობასთან. არაა აუცილებელი რომ შინაგანი ფუნქცია აისახოს ერთ ინსტრუქციაში, მაგრამ AVX / AVX2 შინაგანი ფუნქციები საიმედოდ უზრუნველყოფს მაღალ წარმადობას სხვა C / C ++ სხვა ფუნქციებთან შედარებით.

<<< 1. წინასწარი მოთხოვნები

ამ სტატიის შინაარსის გასაგებად, საჭიროა გაცნობილი იყოთ C და SIMD-ის მუშაობის საფუძვლებთან. კოდის შესასრულებლად, თქვენ გჭირდებათ CPU, რომელიც მხარს უჭერს AVX-ს ან AVX/AVX2-ს. აი ასეთი პროცესორები:

- Intel's Sandy Bridge / Sandy Bridge E / Ivy Bridge / Ivy Bridge E
- Intel's Haswell / Haswell E / Broadwell / Broadwell E
- AMD Bulldozer / Piledriver / Steamroller / Excavator

ყველა CPU, რომელიც მხარს უჭერს AVX2-ს, ასევე მხარს უჭერს AVX-ს. აქ მოცემულია ასეთები:

- Intel's Haswell/Haswell E/Broadwell/Broadwell E
- AMD's Excavator

ამ სტატიაში განხილული ფუნქციების უმეტესი ნაწილი მოცემულია AVX-ის მიერ. მაგრამ რამდენიმე არის AVX2 -ული. მათი განსხვავების მიზნით, AVX2 intrinsics-ებს წინ უწერია (2), ყველა ცხრილში რაც ქვემოთაა.

<<< 2. ვექტორული დაპროგრამების მიმოხილვა

AVX ინსტრუქციები აუმჯობესებს დანართის წარმადობას, ერთდროულად ამუშავებს რა მნიშვნელობების ვრცელ ნაგლეჯებს მათი ცალ-ცალკე დამუშავების ნაცვლად. მნიშვნელობების ამ ნაგლეჯებს *ვექტორებს* უწოდებენ, ხოლო AVX-ის ვექტორებს შეუძლიათ 256 ბიტამდე მონაცემების დატევა. ჩვეულებრივი AVX ვექტორები შეიცავს ოთხ `double` (4 x 64 bits = 256), რვა `float` (8 x 32 bits = 256), ან რვა `int`s (8 x 32 bits = 256).

მაგალითი დაგვანახებს AVX /AVX2-ის ძალას. დავუშვათ, რომ ფუნქციამ ერთი მასივის რვა `float` უნდა გაამრავლოს მეორე მასივის რვა `float`-ზე და შემდეგ შედეგი დაამატოს მესამე მასივს. ვექტორების გარეშე, ფუნქცია შეიძლება ასე გამოიყურებოდეს:

```
multiply_and_add(const float* a, const float* b, const float* c, float* d) {

  for(int i=0; i<8; i++) {
    d[i] = a[i] * b[i];
    d[i] = d[i] + c[i];
  }
}
```

ხოლო AVX2-ით იგი ასე გამოიყურება:

```
__m256 multiply_and_add(__m256 a, __m256 b, __m256 c) {

  return _mm256_fmadd_ps(a, b, c);
}
```

AVX2-ის ეს შინაგანი ფუნქცია `_mm256_fmadd_ps` ამუშავებს 32 `float`-ს, მაგრამ ის არ აისახება ერთ ცალ ინსტუქციაში. ის ასრულებს სამ ინსტრუქციას: `vfmadd132ps`, `vfmadd213ps`, და `vfmadd231ps`. მაინც ეს ბევრად სწრაფია ვიდრე ცალ-ცალკე ელემენტების დატრიალება მარყუჟში.

Intel-ის შინაგანი ფუნქციების სიმძლავრის მიუხედავად, ეს ბევრ პროგრამისტს ანერვიულობს ორი მიზეზის გამო. პირველი, მონაცემთა ტიპებს აქვთ უცნაური სახელები, როგორიცაა `__m256`. მეორე, ფუნქციებს აქვს უცნაური სახელები, როგორიცაა `_mm256_fmadd_ps`. ამიტომ, შინაგანი ფუნქციების დეტალურად განხილვამდე საჭიროა Intel-ის მონაცემთა ტიპებისა და სახელებზე მიღებული შეთანხმებების განხილვა.

<<< AVX პროგრამირების საფუძვლები

ამ სტატიის უმეტესი ნაწილი ფოკუსირებულია მათემატიკასთან დაკავშირებულ შინაგან ფუნქციებზე, რომლებიც გათვალისწინებულია AVX და AVX2 მიერ. სანამ ფუნქციებს მივხედავთ, მნიშვნელოვანია გვესმოდეს სამი პუნქტი:

- მონაცემთა ტიპები
- ფუნქციების სახელებზე მიღებული შეთანხმებები
- AVX პროგრამების კომპილირება

ეს ნაკვეთი ეხება თითოეულ ამ საკითხს და გვთავაზობს მარტივ დანართს, რომელიც გამოაკლებს ერთ ვექტორს მეორისგან.

<<< 3.1. მონაცემთა ტიპები

რამდენიმე შინაგანი ფუნქცია იღებს ტრადიციულ მონაცემთა ტიპებს, როგორიცაა `int` ან `float`, მაგრამ უმეტესობა მუშაობს მონაცემთა ტიპებზე, რომლებიც სპეციფიკურია AVX და AVX2-ისთვის. არსებობს ექვსი ძირითადი ვექტორის ტიპი ქვემოთ ჩმოთვლილია თითოეული მათგანი

Data Type	Description
`__m128`	128-bit ვექტორი, შეიცავს 4 `float`-ს
`__m128d`	128-bit ვექტორი, შეიცავს 2 `double`-ს
`__m128i`	128-bit ვექტორი, შეიცავს მთელებს
`__m256`	256-bit ვექტორი, შეიცავს 8 `float`-ს
`__m256d`	256-bit ვექტორი, შეიცავს 4 `double`-ს
`__m256i`	256-bit ვექტორი, შეიცავს მთელებს

თითოეული ტიპი იწყება ორი ქვედა ხაზით, m- ით და ვექტორის სიგანით ბიტებში. AVX512 მხარს უჭერს 512 ბიტიანი ვექტორის ტიპებს, რომლებიც იწყება _m512– ით, მაგრამ AVX / AVX2 ვექტორები არ სცილდება 256 ბიტს.

თუ ვექტორის ტიპით მთავრდება d-თი, ის შეიცავს `double`-ებს და თუ მას არ აქვს სუფიქსი, ის შეიცავს `float`-ებს. შეიძლება გვეგონოს რომ _m128i და _m256i ვექტორები შეიცავს `int`-ებს, მაგრამ ეს ასე არ არის. მთელი ვექტორის ტიპი შეიძლება შეიცავდეს ნებისმიერი ტიპის მთელებს, `char`, `short` და თვით `unsigned long long`. ანუ, _m256i შეიძლება შეიცავდეს 32 `char`, 16 `short`, 8 `int` -ს, ან 4 `long`-ს. ესენი შესაძლოა იყოს ნიშნიანი ან უნიშნო

<<< 3.2. ფუნქციების სახელებზე მიღებული შეთანხმებები

AVX / AVX2-ის შიანგანი ფუნქციების სახელები შეიძლება თავიდან დამაბნეველი იყოს, მაგრამ დასახელებზე შეთანხმება ნამდვილად სწორხაზოვანია. მას შემდეგ რაც გაერკვევით, შეძლებთ დაახლოებით განსაჯოთ, თუ რას ასრულებს ფუნქცია მისი სახელიდან გამომდინარე. განზოგადებლი AVX / AVX2 შინაგანი ფუნქცია მოიცემა შემდეგნაირად:

`_mm<bitWidth>_<name>_<dataType>`

ამ ფორმატის შემადგენელი ნაწილებია:

1. `bitWidth` - მიუთითებს ფუნქციის მიერ დასაბრუნებელი ვექტორის სიგრძეს. 128 ბიტიანი ვექტორისთვის ეს გამოტოვებულია. 256 ბიტიანისთვის ეს არის 256.
2. `name` - აღწერსშინაგანი ფუნქციის მიერ შესრულებულ მოქმედებას.
3. `dataType` - აღწერს ფუნქციის პირველადი არგუმენტების მონაცემთა ტიპს.

ბოლო ნაწილი, `dataType` შედარებით რთულია. იგი შეიძლება იყოს ერთ-ერთი შემდეგთაგან:

- `ps` - ვექტორი შეიცავს `float`s-ებს (`ps` ნიშნავს packed single-precision)
- `pd` - ვექტორი შეიცავს `double`s (`pd` ნიშნავს packed double-precision)
- `epi8/epi16/epi32/epi64` - ვექტორი შეიცავს 8-bit/16-bit/32-bit/64--ბიტიან ნიშნიან მთელებს
- `epu8/epu16/epu32/epu64` - ვექტორი შეიცავს 8-bit/16-bit/32-bit/64-ბიტიან უნიშნო მთელებს
- `si128`/`si256` - განუსაზღვრელი 128-ბიტიანი ვექტორი ან 256--ბიტიანი ვექტორი
- `m128/m128i/m128d/m256/m256i/m256d` -განსაზღვრავს პარამეტრი ვექტორის ტიპებს როდესაც ისინი განსხვავდება დასაბრუნებელი მნიშვნელობის ტიპისგან.

ნიმუშად განვიხილოთ `_mm256_srlv_epi64`. მაშინაც კი, თუ არ იცით რას ნიშნავს `srlv`, წინდებული `_mm256` გეუბნებათ, რომ ფუნქცია დააბრუნებს 256 ბიტიან ვექტორს და `epi64`. ამბობს, რომ არგუმენტი შეიცავს 64 ბიტიან ნიშნიან მთელს.

მეორე ნიმუშად განვიხილოთ `_mm_testnzc_ps`. `_mm` გულისხმობს, რომ ფუნქცია აბრუნებს 128 ბიტიან ვექტორს. ბოლოში `ps` გულისხმობს, რომ არგუმენტში ვექტორი შეიცავს `float` -ებს.

AVX-ის მონაცემთა ტიპები იწყება ოთხი ზედა რეგისტრის ტირეთი და `m` -ით. ფუნქციები იწყება ერთი ზედა რეგისტრის ტირეთი და ორი `m` -ით.

<<< 3.3. AVX დანართების აგებას

ისეთი დანართის ასაგებად, რომელიც იყენებს AVX-ის შინაგან ფუნქციებს, არ გვჭირდება ბიბლიოთეკების მიბმა. საკმარისია ჩავრთოთ <immintrin.h> სათაურის ფაილი. ეს ფაილი შეიცავს სხვა სათაურის ფაილებს, რანიც ასახავენ AVX/AVX2 -ის ფუნქციებს ინსტრუქციებში.

შემდეგი კოდი გვიჩვენებს თუ როგორ გამოიყურება AVX დანართი:

```
#include <immintrin.h>
#include <iostream>

int main() {

	/* Initialize the two argument vectors */
	__m256 evens = _mm256_set_ps(2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0);
	__m256 odds = _mm256_set_ps(1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0, 15.0);

	/* Compute the difference between the two vectors */
	__m256 result = _mm256_sub_ps(evens, odds);

	/* Display the elements of the result vector */
	float* f = (float*)&result;
	for (size_t i = 0; i < 8; ++i)
		std::cout << f[i] << "  ";
	std::cout << std::endl;
}
```

ამ მაგალითში, ყველა ფუნქცია იწყება `_mm256` -ით და მთავრდება `_ps` -ით. 256 ბიტიანი ვექტორები `float`-ებისგან შედგება, ინციალიზება ხდება ლუწი და კენტი რიცხვებით, მათი სხვაობა ერთიანებია.

შენიშვნა: სტატიაში მოცემულია `gcc` -ში პროგრამის აგებისთვის საჭირო პარამეტრები. აქ მოყვანილი კოდი Visual Studio 2019 -ში არის C++ ენაზე გადაწყობილი. აქ პროექტის ნაგულისხმევი პარამეტრებით პრგრამა სრულდება ორივე რეჟიმში (Debug, Release).

<<< 4. შინაგანი ფუნქციების ინიციალიზება

სანამ ვიმოქმედებთ ვექტორებზე, ჯერ საჭიროა ვექტორის ავსება მონაცემებით. ამიტომ ამ სტატიაში განხილული შინაგანი ფუნქციების პირველი ჯგუფი ვექტორებს აინიციალებს მონაცემებით. ამის გაკეთების ორი გზა არსებობს: ვექტორების ინციალიზება სკალარული მნიშვნელობებით და ვექტორების ინციალიზება მეხსიერებიდან გადმოტვირთული მონაცემებით.

<<< 4.1. ინიციალიზება სკალარული მნიშვნელობებით

AVX გვთავაზობს შინაგან ფუნქციებს, რომლებიც 256 ბიტიან ვექტორში გააერთიანებს ერთ ან მეტ მნიშვნელობას. შემდეგი ცხრილი გვიჩვენებს სახელებს და აღწერს მათ.

მსგავსი შინაგანი ფუნქციები არსებობს 128 ბიტიანი ვექტორების ინციალიზებითვის, მაგრამ ისინი შემოთავაზებულია SSE-ს მიერ. სახელებში ერთადერთი განსხვავება ისაა, რომ `__mm256_` შეცვლილია `_mm_` -ით.

`_mm256_setzero_ps/pd`	აბრუნებს ნულებით შევსებულ მცოცავწერტილიან ვექტორს
`_mm256_setzero_si256`	აბრუნებს მთელ ვექტორს, რომლის ბიტებიც შევსებულია ნულებით
`_mm256_set1_ps/pd`	შეავსებს ვექტორს მცოცავწერტილიანი მნიშვნელობებით
`_mm256_set1_epi8/epi16` `_mm256_set1_epi32/epi64`	შეავსებს ვექტორს მთელით
`_mm256_set_ps/pd`	აინიციალებს ვექტორს 8 float-ით ან 4 double-ით
`_mm256_set_epi8/epi16` `_mm256_set_epi32/epi64`	აინიციალებს ვექტორს მთელებით
`_mm256_set_m128/m128d/` `_mm256_set_m128i`	აინიციალებს 256-ბიტიან ვექტორს ორი 128-ბიტიან ვექტორით
`_mm256_setr_ps/pd`	შებრუნებული რიგით აინიციალებს ვექტორს 8 float-ით ან 4 double-ით
`_mm256_setr_epi8/epi16` `_mm256_setr_epi32/epi64`	შებრუნებული რიგით აინიციალებს მთელებით

ამ ცხრილის პირველი ფუნქციები უფრო ადვილი მისახვედრია. `_m256_setzero_ps` აბრუნებს `__m256` ვექტორს რომელიც შეიცავს 8 `float`-ს, შევსებულს ნულებით. მსგავსად ამისა, `_mm256_setzero_si256` აბრუნებს `__m256i` ვექტორს რომლის ბიტებიც განულებულია. მაგალითად, შემდეგი ხაზის კოდი ქმნის 256 ბიტიან ვექტორს, რომელიც შეიცავს 4 განულებულ `double`-ს:

```
__m256d dbl_vector = _mm256_setzero_pd();
```

მომდევნოები, რომელთა სახელები შეიცავს `set1`-ს, მიითებს ცალ მნიშვნელობას და გაამეორებს მას ვექტორში. მაგალითად, შემდეგი ხაზის კოდი ქმნის ერთ `__m256i`-ს და მის 16 მოკლე მნიშვნელობას გაუტოლებს (თითოეულს) 47-ს:

```
__m256i short_vector = _mm256_set1_epi16(47);
```

დარჩენილი ფუნქციები შეიცავს _set_ ან _setr_. ეს ფუნქციები მიიღებს მნიშვნელობების სერიებს, თითოს ვექტორის თითო ელემენტზე. ეს მნიშვნელობები ჩაჯდება დასაბრუნებელ ვექტორში, და მიმდევრობას აქვს მნიშვნელობა. შემდეგი ფუნქციის გამოძახება დააბრუნებს 8 int-ს, რომელთა მნიშვნელობები იცვლება 1-იდან 8-ის ჩათვლით:

```
__m256i int_vector = _mm256_set_epi32(1, 2, 3, 4, 5, 6, 7, 8);
```

შესაძლოა გაგვიჩნდეს მოლოდინი, რომ მნიშვნელობები იგივე რიგირ შეინახება რა რიგითაც არის მოცემული. მაგრამ Intel-ის არქიტექტურა არის მცირე-ბოლოიანი (ittle-endian), ამიტომ 8-იანი შინახება პირელი, ხოლო 1-იანი ბოლო. ამის შემოწმება შეგვიძლია შემდეგი კოდის საშუალებით:

```
__m256i int_vector = _mm256_set_epi32(1, 2, 3, 4, 5, 6, 7, 8);
int* ptr = (int*)&int_vector;
for (size_t i = 0; i < 8; ++i)
    std::cout << ptr[i] << "   ";
std::cout << std::endl;
```

რის შედეგად დაიბეჭდება **8 7 6 5 4 3 2 1.**

თუ გნებავთ, რომ მნიშვნელობები შინახოს მოცემული რიგით, მაშინ ვექტორი უნდა შევქმნათ _setr_-იანი ფუნქციით:

```
__m256i int_vector = _mm256_setr_epi32(1, 2, 3, 4, 5, 6, 7, 8);
```

შენიშვნა: ჩემთან არ აკეთებს Visual Studio 2019-ში.

<<< 4.2. მონაცემების ჩატვირთვა მეხსიერებიდან

AVX/AVX2-ის ზოგადი გამოყენება გულისხმობს მეხსიერებიდან ვექტორებში მონაცემების ჩამოტვირთვას, ვექტორების დამუშავებას და შედეგების ისევ მეხსიერებაში შენახვას. პირველი ნაწილი სრულდება შემდეგ ცხრილში ჩამოთვლილი შინაგანი ფუნქციებით. ბოლო ორ ფუნქციას უწერია (2), რადგან მხოლოდ AVX2-ის არიან.

_mm256_load_ps/pd	იტვირთება მცოცავწერტილიანი ვექტორი მეხსიერების გაჯერადებული (aligned) მისამართიდან
_mm256_load_si256	იტვირთება მთელი ვექტორი მეხსიერების გაჯერადებული მისამართიდან
_mm256_loadu_ps/pd	იტვირთება მცოცავწერტილიანი ვექტორი მეხსიერების გაუჯერადებელი (unaligned) მისამართიდან
_mm256_loadu_si256	იტვირთება მთელი ვექტორი გაუჯერადებელი მეხსიერების მისამართიდან
_mm_maskload_ps/pd _mm256_maskload_ps/pd	ტვირთავს 128-ბიტ/256-ბიტიანი მცოცავწერტილიანი ვექტორების ულუფებს ნიღბის მიხედვით
(2)_mm_maskload_epi32/64 (2)_mm256_maskload_epi32/64	ტვირთავს 128-ბიტ/256-ბიტიანი მთელი ვექტორების ულუფებს ნიღბის მიხედვით

მონაცემების ვექტორებში ჩატვირთვისას, განსაკუთრებით მნიშვნელოვანია მეხსიერების გაჯერადება. თითოეული _mm256_load_* შინაგანი ფუნქცია იღებს მეხსიერების მისამართს, რომელიც უნდა გაჯერადდეს 32 – ბაიტიან მიჯნაზე. სხვა სიტყვებით, ეს მისამართი უნდა იყოს იყოფოდეს 32-ზე. შემდეგი კოდი გვიჩვენებს, თუ როგორ შეიძლება ამის გამოყენებება პრაქტიკაში:

```
float* aligned_floats = (float*)aligned_alloc(32, 64 * sizeof(float));
... Initialize data ...
__m256 vec = _mm256_load_ps(aligned_floats);
```

ნებისმიერი მცდელობა, რომ `_mm256_load_*` -ით ჩაიტვირთოს გაუჯერადებელი მონაცემები წარმოშობს სეგმენტაციის შეცდომას. თუ მონაცემები არა გაჯერადებული 32-ბიტიანი მიჯნებით, მაშინ უნდა გამოვიყენოთ `_m256_loadu_*` . ეს ნაჩვენებია შემდეგ კოდში:

```
float* unaligned_floats = (float*)malloc(64 * sizeof(float));
... Initialize data ...
__m256 vec = _mm256_loadu_ps(unaligned_floats);
```

დავუშვათ, გვინდა `float` მასივის დამუშავებასი, AVX ვექტორების გამოყენებით, მაგრამ მასივის სიგრძე არის 11. ეს არ იყოფა 8-ზე. ამ შემთხვევაში, მეორე `__m256` ვექტორის ბოლო 5 `float` ელემენტი უნდა გაუტოლდეს ნულს და ისინი არ იქონიებენ გავლენას ოპერაციაზე. ასეთი შერჩევით ჩატვირთვა შეიძლება შესრულდეს ბოლო ცხრილის ბოლოში მოთავსებული `_maskload_` ფუნქციებით.

ყოველი `_maskload_` ფუნქცია იღებს ორ არგუმენტს: მეხსიერების მისამართს და ერთიც მთელი ვექტორი იმდენივე ელემენტით რამდენიცაა დასაბრუნებელი ვექტორში. მთელი მასივის ყოველი ისეთი ელემენტისთვის, რომლის უდიდესი ბიტი ერთია, შესაბამისი ელემენტი დასაბრუნებელი ვეტორში ამოიკითხება მეხსიერებიდან./თუ უდიდესი ბიტი ნულია, მაშინ შესაბამისი ელემენტი დასაბრუნებელ ვეტორში განულდება.

შემდეგი მაგალითი ათვალსაჩინოებს ამ ფუნქციების გამოყენებას. კოდი კითხულლობს 8 `int`-ს ვექტორში, ხოლო ბოლო სამი უნდა განულდეს. გამოიყენება `_mm256_maskload_epi32`, და მისი მეორე არგუმენტი უნდა იყოს `__m256i` ნიღბის ვექტორი. ეს ნიღაბი-ვექტორი შეიცავს 5 `int`-ს, რომელთა უდიდესი ბიტი 1-ია და 3 `int`-ს ნულოვანი უდიდესი ბიტით. კოდი ასეთია:

```
#include <immintrin.h>
#include <iostream>

int main()
{
	int int_array[8] = { 100, 200, 300, 400, 500, 600, 700, 800 };

	/* Initialize the mask vector */
	__m256i mask = _mm256_setr_epi32(-20, -72, -48, -9, -100, 3, 5, 8);

	/* Selectively load data into the vector */
	__m256i result = _mm256_maskload_epi32(int_array, mask);

	/* Display the elements of the result vector */
	int* res = (int*)&result;
	for (size_t i = 0; i < 8; ++i)
		std::cout << res[i] << "  ";
	std::cout << std::endl;
}
```

შედეგი არის **100 200 300 400 500 0 0 0**

აქ არის სამი საკითხი, რაც უნდა აღინიშნოს:

1. კოდი აყენებს ნიღაბი ვექტორის შინაარსს `_setr_` ფუნქციით და არა `_set_`ით, რადგან ის ისე ალაგებს ვექტორის ელემენტებს, როგორც ისინი მოწოდებულია ფუნქციის მიერ.
2. უარყოფითი მთელის უდიდესი ბიტი ყოველთვის ნულია. ამიტომაა რომ ნიღაბი ვექტორი შეიცავს ხუთ უარყოფითს და სამ დადებითს.
3. `_mm256_maskload_epi32` არის AVX2-ის, ამიტომ gcc-ზე კომპილირებისას საჭიროა `mavx2` ალმის ჩართვა `mavx`-ის ნაცვლად.

ამათ გარდა, AVX2 გვთავაზობს gather-ფუნქციებს, რომლებიც ტვირთავენ ინდექსირებულ მონაცემებს მეხსიერებიდან.

<<< 5. არითმეტიკული შინაგანი ფუნქციები

მათემატიკა არის AVX-ების არსებობის პირველადი მიზეზი და ძირითადი მოქმედებები არის შეკრება, გამოკლება, გამრავლება და გაყოფა. ამ ნაკვეთში ჯერ ეს შინაგანი ფუნქციებია წარმოდგენილი, შემდეგ ნაჩვენებია AVX2-ის მიერ შერწყმულ (fused) გაამრავლე-და-დაამატე ფუნქციები.

<<< 5.1. შეკრება და გამოკლება

შემდეგ ცხრილში მოყვანილია AVX/AVX2-ის შინაგანი ფუნქციები, რომლებიც ასრულებენ შეკრებას და გამოკლებას. მათი უმეტესობა მოქმედებს მთელ ვექტორებზე, გაჯერებაზე (saturation, იხ. https://en.wikipedia.org/wiki/Saturation_arithmetic) ზრუნვის გამო.

Data Type	Description
_mm256_add_ps/pd	კრებს ორ მცოცავწერტილიან ვექტორს
_mm256_sub_ps/pd	გამოაკლებს ორ მცოცავწერტილიან ვექტორს
(2)_mm256_add_epi8/16/32/64	კრებს ორ მთელ ვექტორს
(2)_mm236_sub_epi8/16/32/64	გამოაკლებს ორ მთელ ვექტორს
(2)_mm256_adds_epi8/16 (2)_mm256_adds_epu8/16	გაჯერებით კრებს ორ მთელ ვექტორს
(2)_mm256_subs_epi8/16 (2)_mm256_subs_epu8/16	გაჯერებით გამოაკლებს ორ მთელ ვექტორს
_mm256_hadd_ps/pd	ჰორიზონტალურად კრებს ორ მცოცავწერტილიან ვექტორს
_mm256_hsub_ps/pd	ჰორიზონტალურად გამოაკლებს ორ მცოცავწერტილიან ვექტორს
(2)_mm256_hadd_epi16/32	ჰორიზონტალურად კრებს ორ მთელ ვექტორს
(2)_mm256_hsub_epi16/32	ჰორიზონტალურად გამოაკლებს ორ მთელ ვექტორს
(2)_mm256_hadds_epi16	მოკლე მთელების ორ ვექტორს კრებს გაჯერებით
(2)_mm256_hsubs_epi16	მოკლე მთელების ორ ვექტორს გამოაკლებს გაჯერებით
_mm256_addsub_ps/pd	კრებს და აკლებს ორ მცოცავწერტილიან ვექტორს

როდესა ხდება მთელი ვექტორების შეკრება ან გამოკლება, მნიშვნელოვანია გვესმოდეს განსხვავება _add_/_sub_ და _adds_/_subs_ ფუნქციებს შორის. ეს დამატებით s მიუთითებს გაჯერებას (saturation), რაც წარმოიშობა როდესაც შედეგი მოითხოვს მეტ მეხსიერებას ვიდრე ვექტორს შეუძლია შენახვა. ის ფუნქციები რომლებიც ითვალისწინებენ გაჯერებას, ამაგრებენ შედეგს minimum/maximum მნიშვნელობაზე რომლის შენახვაც შეიძლება. თუ ფუნქცია არ ითვალისწინებს გაჯერებას, მაშინ იგი უგულებელყოფს მეხსიერებასთან დაკავშირებულ საკითხებს როდესაც ისინი წარმოიშობა.

მაგალითად, დავუშვათ ვექტორი ინახავს ნიშნიან ბაიტებს, ასე რომ თითოეული ელემენტის მაქსიმალური მნიშვნელობა არის 127(0x7F). თი მოქმედება შეკრებს 98 და 85-ს, მათემატიკური ჯამი არის 183(0xB7).

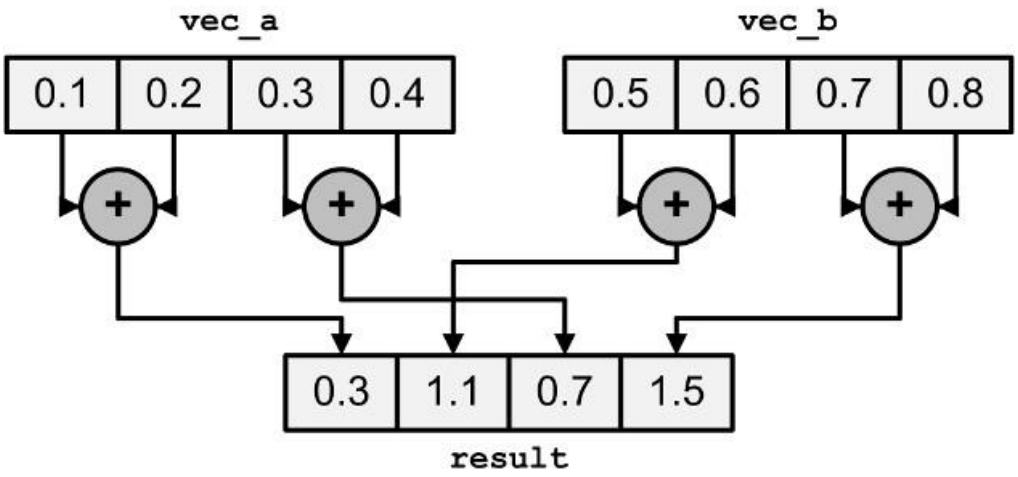
- თუ შეკრება ხდება `_mm256_add_epi8`-ით, გაჯერებას ყურადღება არ მიექცევა და შენახული მნიშვნელობა იქნება -73(0xB7).
- თუ შეკრება ხდება `_mm256_adds_epi8`-ით, შედეგი დამაგრდება მაქსიმალურ მნიშვნელობაზე 127(0x7F).

როგორც სხვა მაგალითი, განვიხილოთ ორი ვექტორი, რომელიც შედგებიან მოკლე ნიშნიანი მთელებისგან. მინიმალური მნიშვნელობა არის -32,768. თუ თქვენ გამოითვლით -18,000 - 19,000, მათემატიკური შედეგი იქნება -37,000 (0xFFFF6F78 როგორც 32 ბიტიანი მთელი).

- თუ გამოკლება ხდება `_mm256_sub_epi16`-ით, გაჯერებას ყურადღება არ მიექცევა და შენახული მნიშვნელობა იქნება 28,536 (0x6F78).
- თუ გამოკლება ხდება `_mm256_subs_epi16`-ით, შედეგი დამაგრდება მინიმალურ მნიშვნელობაზე -32,768 (0x8000).

`_hadd_/_hsub_` ფუნქციები ჰორიზონტალურად ასრულებენ შეკრება-გამოკლებას: იმის ნაცვლად რომ შეკრიბონ ან გამოაკლონ განსხვავებული ვექტორების მნიშვნელობები, ისინი კრებენ ან აკლებენ მეზობელ ელემენტებს თითოეულ ვექტორში. შედეგი შეინახება ურთიერთმონაცვლე წესით. შემდეგ სურათზე `_mm256_hadd_pd` ჰორიზონტალურად კრებს ორ `double` ვექტორს სახელებით A,B:

```
__m256d result = _mm256_hadd_pd(vec_a, vec_b);
```



ცოტა უცნაურად გამოიყურება ასეთი მოქმედებები, მაგრამ ისინი სასარგებლოა კომპლექსური რიცხვების გამრავლების დროს. ეს სტატიის ბოლოშია ახსნილი.

ცხრილის ბოლო, `_mm256_addsub_ps/pd` ფუნქცია მოქმედებს ორ მცოცავწერტილიან ვექტორზე. ლუწი ელემენტები გამოაკლდება, ხოლო კენტები შეიკრიბება. მაგალითად, თუ `vec_a` შეიცას (0.1, 0.2, 0.3, 0.4) , ხოლო `vec_b` შეიცავს (0.5, 0.6, 0.7, 0.8), მაშინ `_mm256_addsub_pd(vec_a, vec_b)` იქნება (-0.4, 0.8, -0.4, 1.2).

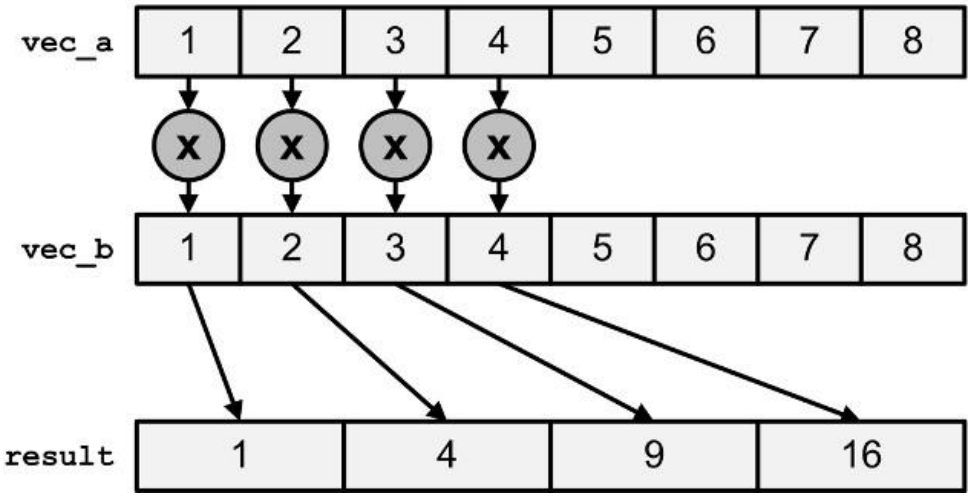
<<< 5.2. გამრავლება და გაყოფა

ცხრილში ჩამოთვლილია AVX / AVX2 შინაგანი ფუნქციები, რომლებიც ასრულებენ გამრავლებას და გაყოფას. ისევე როგორც დამატებისა და გამოკლების შემთხვევაში, მთელ რიცხვებზე მოქმედებისთვის არსებობს განსაკუთრებული შინაგანი ფუნქციები.

Data Type	Description
_mm256_mul_ps/pd	ამრავლებს ორ მცოცავწერტილიან ვექტორს
(2)_mm256_mul_epi32/ (2)_mm256_mul_epu32	32 ბიტიანი მთელების შემცველი ვექტორის უმცროს 4 ელემენტს ამრავლებს
(2)_mm256_mullo_epi16/32	ამრავლებს მთელი რიცხვებს და ინახავს უმცროს ნახევრებს
(2)_mm256_mulhi_epi16/ (2)_mm256_mulhi_epu16	ამრავლებს მთელი რიცხვებს და ინახავს უფროს ნახევრებს
(2)_mm256_mulhrs_epi16	ამრავლებს 16-ბიტიანებს რომ შექმნას 32-ბიტიანი ელემენტები
_mm256_div_ps/pd	გაყოფს ორ მცოცავწერტილიან ვექტორს

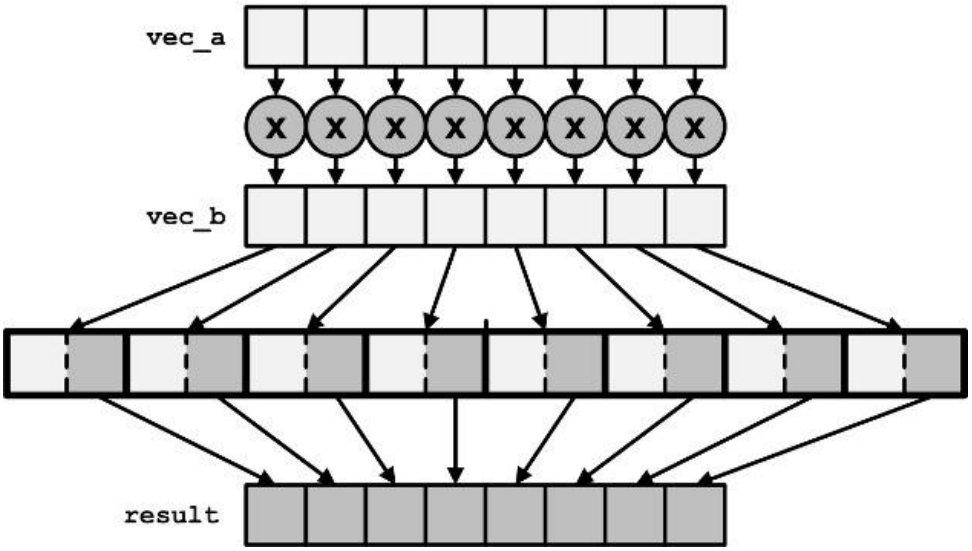
თუ კომპიუტერში მრავლდება ორი N- ბიტიანი რიცხვი, შედეგს შეუძლია დაიკავოს 2N ბიტამდე. ამ მიზეზით, _mm256_mul_epi32 და _mm256_mul_epu32 ფუნქციების მხოლოდ ოთხი უმცროსი ელემენტი მრავლდება ერთად, ხოლო შედეგი არის ვექტორი, რომელიც შეიცავს ოთხ გრძელ მთელი რიცხვს. შემდეგი სურათი გვიჩვენებს, თუ როგორ ხდება ეს:

```
__m256i result = _mm256_mul_epi32(vec_a, vec_b);
```



mullo ფუნქციები მსგავსია მთელი _mul_ ფუნქციების, მაგრამ უმცროსი ელემენტების გამრავლების ნაცვლად, ისინი ორივე ვექტორების ყველა ელემენტს ამრავლებენ და ინახავენ მხოლოდ თითოეული ნამრავლის უმცროს ნახევარს. შემდეგი სურათი გვიჩვენებს, თუ როგორ გამოიყურება ეს:

```
__m256i result = _mm256_mullo_epi32(vec_a, vec_b);
```



_mm256_mulhi_epi16 და _mm256_mulhi_epu16 შინაარსით მსგავსია, მაგრამ ისინი ინახავს მთელი ნამრავლის უფროს ნახევარს.

<<< 5.3. შერწყმული გამრავლება და შეკრება (FMA)

როგორც უკვე აღვნიშნეთ, ორი N- ბიტიანი რიცხვის გამრავლების შედეგმა შეიძლება დაიკავოს 2N ბიტი. ამიტომ, როდესაც ამრავლებთ ორ მცოცავ- წერტილიან a და b მნიშვნელობას, შედეგი არის დამრგვალებული round(a * b), სადაც მრგვალი round((x) აბრუნებს მცოცავ- წერტილიან მნიშვნელობას, რომელიც უახლოესია x-თან. სიზუსტის დაკარგვა იზრდება შემდგომი ოპერაციების შესრულებისას.

AVX2 გთავაზობთ ინსტრუქციებს, რომლებიც აერთიანებს ერთმანეთთან გამრავლებას და შეკრებას. ანუ, round(round(a * b) + c)-ს დაბრუნების ნაცვლად, ისინი round(a * b + c)-ს. შედეგად, ეს ინსტრუქციები იძლევა უფრო მეტ სიჩქარეს და სიზუსტეს, ვიდრე ცალ-ცალკე გამრავლების და შეკრების შემთხვევაში.

შემდეგ ცხრილში ჩამოთვლილია AVMA2 მიერ შემოთავაზებული FMA-ს შინაგან ფუნქციებს და აღწერს თითოეულ მათგანს. ცხრილში მოცემული ყველა ინსტრუქცია პარამეტრებად იღებს სამ ვექტორს, პირობითად a, b და c ..

მონაცემთა ტიპი	აღწერა
(2)_mm_fmadd_ps/pd/ (2)_mm256_fmadd_ps/pd	ორ ვექტორს ამრავლებს და ნამრავლს უმატებს მესამე ვექტორს (res = a * b + c)
(2)_mm_fmsub_ps/pd/ (2)_mm256_fmsub_ps/pd	ორ ვექტორს ამრავლებს და ნამრავლს გამოაკლებს მესამე ვექტორს (res = a * b - c)
(2)_mm_fmadd_ss/sd	ამრავლებს და კრებს ვექტორების უმცროს ელემენტებს (res[0] = a[0] * b[0] + c[0])
(2)_mm_fmsub_ss/sd	ამრავლებს და აკლებს ვექტორების უმცროს ელემენტებს (res[0] = a[0] * b[0] - c[0])
(2)_mm_fnmadd_ps/pd (2)_mm256_fnmadd_ps/pd	ამრავლებს ვექტორებს და ნამრავლის მოპირდაპირეს უმატებს მესამეს (res = -(a * b) + c)
(2)_mm_fnmsub_ps/pd/ (2)_mm256_fnmsub_ps/pd	ამრავლებს ვექტორებს და ნამრავლის მოპირდაპირეს აკლებს მესამეს (res = -(a * b) - c)

(2)_mm_fnmadd_ss/sd	ამრავლებს ორ უმცროს ელემენტს და ნამრავლის მოპირდაპირეს უმატებს მესამის უმცროს ელემენტს (res[0] = -(a[0] * b[0]) + c[0])
(2)_mm_fnmsub_ss/sd	ამრავლებს ორ უმცროს ელემენტს და ნამრავლის მოპირდაპირეს აკლებს მესამის უმცროს ელემენტს (res[0] = -(a[0] * b[0]) - c[0])
(2)_mm_fmaddsub_ps/pd/ (2)_mm256_fmaddsub_ps/pd	ამრავლებს ორ ვექტორს და ნამრავლს მონაცვლეობით ემატება და აკლდება მესამე
(2)_mm_fmsubadd_ps/pd/ (2)_mmf256_fmsubadd_ps/pd	ამრავლებს ორ ვექტორს და ნამრავლს მონაცვლეობით აკლდება და ემატება მესამე

თუ ფუნქციის სახელი მთავრდება _ps ან _pd-თი, პარამეტრი ვექტორების ყველა ელემენტი არი ჩართული მოქმედებებში. თუ ფუნქციის სახელი მთავრდება _ss ან _sd-თი, მხოლოდ უმცროსი ელემენტებია ჩართული. დასაბრუნებელი ვექტორის დანარჩენი ელემენტები უტოლდება პირველი პარამეტრი ვექტორის ელემენტებს. მაგალითად, დავუშვათ vec_a = (1.0, 2.0), vec_b = (5.0, 10.0), და vec_c = (7.0, 14.0). მაშინ, _mm_fmadd_sd(vec_a, vec_b, vec_c) დააბრუნებს (12.0, 2.0)-ს, რადგან (1.0 * 5.0) + 7.0 = 12.0 და 2.0 არის vec_a -ს მეორე ელემენტი.

მნიშვნელოვანია დაინახოთ სხვაობა _fmadd_/_fmsub_ და _fnmadd_/_fnmsub_ ფუნქციებს შორის. ბოლო ფუნქციები ნიშანს უცვლის პირველი ორი პარამეტრი ვექტორის ნამრავლს მანამდე, ვიდრე დაამატებს ან გამოაკლებს მესამე ვექტორს.

fmaddsub და _fmsubadd_ ფუნქციები განსხვავდებიან მესამე ვექტორის ელემენტების შეკრება-გამოკლების მიმდევრობით. _fmaddsub_ ფუნქცია ამატებს კენტ ელემენტებს და აკლებს ლუწ ელემენტებს. _fmsubadd_ ფუნქცია ამატებს ლუწ ელემენტებს და აკლებს კენტ ელემენტებს. შემდეგი კოდი გვიჩვენებს თუ როგორ გამოიყენება პრაქტიკაში _mm256_fmaddsub_pd ფუნქცია:

```
#include <immintrin.h>
#include <iostream>

int main() {
	__m256d veca = _mm256_setr_pd(6.0, 6.0, 6.0, 6.0);

	__m256d vecb = _mm256_setr_pd(2.0, 2.0, 2.0, 2.0);

	__m256d vecc = _mm256_setr_pd(7.0, 7.0, 7.0, 7.0);

	/* Alternately subtract and add the third vector
	   from the product of the first and second vectors */
	__m256d result = _mm256_fmaddsub_pd(veca, vecb, vecc);

	/* Display the elements of the result vector */
	double* res = (double*)&result;
	for (size_t i = 0; i < 4; ++i)
		std::cout << res[i] << "   ";
	std::cout << std::endl;
}
```

შედეგი ასეთია: 5 19 5 19

საინტერესოა, რომ gcc-ზე კოდის კომპილირებისთვის საჭიროა -mfma ალმის ჩართვა.

<<< 6. გადანაცვლება და აღრევა

ბევრ დანართს უწევს ვექტორის ელემენტების გადაადგილება რომ დარწმუნდეს განხორციელებული მოქმედებების სიწორეში. AVX / AVX2 ამ მიზნით გთავაზობთ არაერთ

შინაგან ფუნქციას, ხოლო ორი ძირითადი კატეგორიაა _permute_ ფუნქციები და _shuffle_ ფუნქციები. ეს ნაკვეთი წარმოადგენს ორივე ტიპის ფუნქციებს.

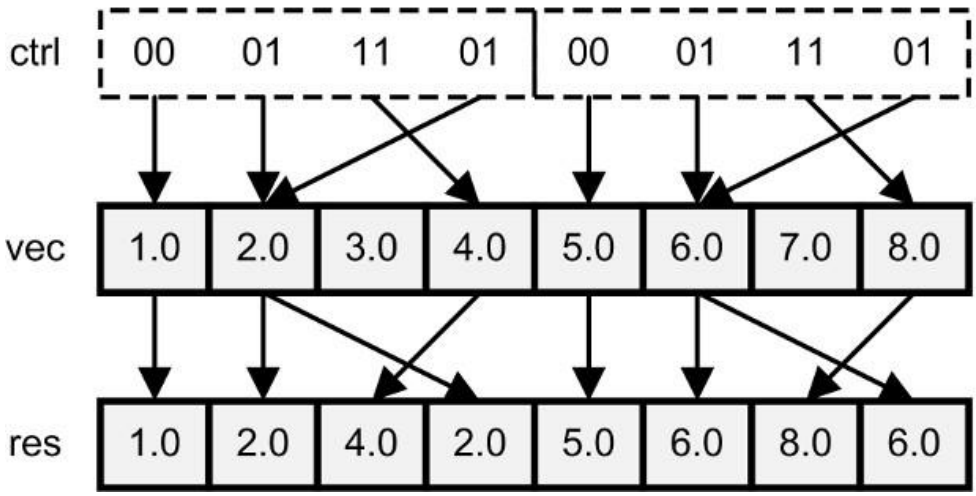
<<< 6.1. გადანაცვლება

AVX გვთავაზობს ფუნქციებს, რომლებიც დააბრუნებენ ვექტორს, რომელიც შეიცავს ვექტორის რიგშეცვლილ ელემენტებს ელემენტებს. შემდეგ ცხრილში ჩამოთვლილია გადამნაცვლებელი ფუნქციები და აღწერილია თითოეული მათგანი.

მონაცემთა ტიპი	აღწერა
_mm_permute_ps/pd/ _mm256_permute_ps/pd	აირჩევა ელემენტები შემავალი ვექტორიდან 8-ბიტიანი მმართველი მნიშვნელობის საფუძველზე
(2)_mm256_permute4x64_pd/ (2)_mm256_permute4x64_epi64	აირჩევა 64 ბიტიანი ელემენტები შემავალი ვექტორიდან 8-ბიტიანი მმართველი მნიშვნელობის საფუძველზე
_mm256_permute2f128_ps/pd	აირჩევა 128 ბიტიანი ნაგლეჯები ორი შემავალი ვექტორიდან 8-ბიტიანი მმართველი მნიშვნელობის საფუძველზე
_mm256_permute2f128_si256	აირჩევა 128 ბიტიანი ნაგლეჯები ორი შემავალი ვექტორიდან 8-ბიტიანი მმართველი მნიშვნელობის საფუძველზე
_mm_permutevar_ps/pd _mm256_permutevar_ps/pd	აირჩევა ელემენტები შემავალი ვექტორიდან მთელი ვექტორის ბიტების საფუძველზე
(2)_mm256_permutevar8x32_ps/ (2)_mm256_permutevar8x32_epi32	აირჩევა 32 ბიტიანი ელემენტები (float-ები ან int-ები) მთელი ვექტორის ინდექსების საფუძველზე

permute შინაგანი ფუნქცია მიიღებს ორ არგუმენტს: შემავალი ვექტორი და 8 – ბიტიანი მმართველი მნიშვნელობა. მმართველი მნიშვნელობის ბიტები განსაზღვრავს თუ შემავალი ვექტორის რომელი ელემენტია ჩასასმელი გამომავალ ვექტორში. _mm256_permute_ps– ისთვის, მმართველი ბიტების თითოეული წყვილი განსაზღვრავს შემავალ ვექტორში რომელიმე ელემენტს, რომელიც აღმოჩნდება გამომავალ ვექტორში იმ ადგილზე, რომელზეც დგას ბიტების ეს წყვილი. ეს საკმაოდ რთულია, და მომდევნო სურათი დაგვეხმარება გათვალსაჩინოებაში:

```
res = _mm256_permute_ps(vec, 0b01110100)
```



შემავალი ვექტორის მნიშვნელობები შესაძლოა მრავალჯერ განმეორდეს გამომავალ ვექტრში, ან სულ არ აღმოჩნდეს იქ.

_mm256_permute_pd–ში მმართველი მნიშვნელობის უმცროსი ოთხი ბიტი არჩევს double-ების მეზობელ წყვილებს შორის. _mm256_permute4x4_pd მსგავსია, მაგრამ იყენებს ყველა მმართველ ბიტს, რომ აირჩიოს თუ რომელი 64–ბიტიანი ელემენტი მოთავსდეს გამომავალში. _permute2f128_ ფუნქციაში, მმართველი მნიშვნელობა აარჩევს 128 – ბიტიან ნაგლეჯს ორი შემავალი ვექტორიდან, ნაცვლად იმისა, რომ შეარჩიოს ელემენტები ერთი შემავალი ვექტორიდან.

permutevar ასრულებს იგივე ოპერაციას, რასაც _permute_ ფუნქციები. მაგრამ ელემენტების შესარჩევად 8–ბიტიანი მმართველი მნიშვნელობების გამოყენების ნაცვლად, ისინი ეყრდნობიან მთელ ვექტორებს რომმელთა ზომა იგივეა რაც შემავალი ვექტორისაა. მაგალითად, შემავალი ვექტორი _mm256_permute_ps-ში არის _mm256, ასე რომ მთელი ვექტორი არის _mm256i. მთელი ვექტორის უფროსი ბიტები ასრულებენ შერჩევას ისე, როგორც _permute_-ის 8 – ბიტიანი მმართველი მნიშვნელობების ბიტები.

<<< 6.2. აღრევა

permute ფუნქციების მსგავსად, _shuffle_ შეარჩევს ელემენტებს ერთი ან ორი შემავალი ვექტორიდან და მოათავსებს მათ გამომავალ ვექტორში.

შენიშვნა: ჩვენთვის არაა მნიშვნელოვანი ამ ფუნქციების დეტალები, ამიტომ მხოლოდ ჩამოვთლით: _mm256_shuffle_ps/pd, _mm256_shuffle_epi8/ და _mm256_shuffle_epi32, (2)_mm256_shufflelo_epi16/ და (2)_mm256_shufflehi_epi16.

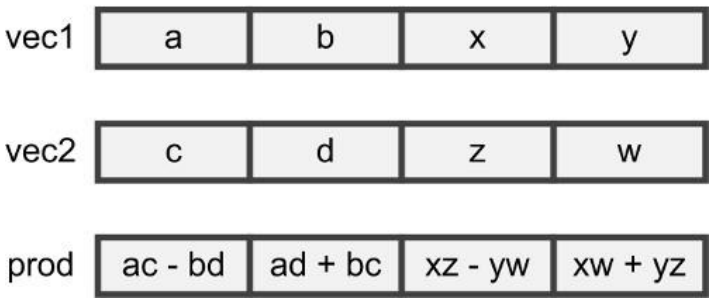
<<< 7. კომპლექსური რიცხვების გამრავლება

კომპლექსური რიცხვების გამრავლება არის შრომატევადი ოპერაცია, რომელიც მრავალჯერ უნდა განხორციელდეს სიგნალის დამუშავების პროგრამებში. კომპლექსური რიცხვი გამოისახება a + bi სახით, სადაც a და b არის ნამდვილი რიცხვები და i არის კვადრატული ფესვი -1-იდან. A-ს ეწოდება ნამდვილი ნაწილი, b-ს ეწოდება წარმოსახვითი ნაწილი. თუ გამრავლებულია (a + bi) და (c + di), რიცხვებს გავამრავლებთ, მიიღება (ac - bd) + (ad + bc)i.

კომპლექსური რიცხვების შენახვა შეიძლება ურთიერთმონაცვლე წესით - ნამდვილი ნაწილი, მისი წარმოსახვითი ნაწილი, მომდევნო კომპლექსური რიცხვის ნამდვილი და წარმოსახვითი ნაწილები და ა.შ.. დავუშვათ, პირველი ვექტორი vec1 არის __m256d , რომელიც ინახავს ორ კომპლექსურ რიცხვს: (a + bi) და (x + yi). ვთქვათ, vec2 აგრეთვე არის __m256d და იგი ინახავს (c + di)-ს და (z + wi) -ს. შემდეგი სურათი გვიჩვენებს ამას. აგრეთვე იმას, რომ შედეგში ინახება ორი ნამრავლი: (ac - bd) + (ad + bc)i , (xz - yw) + (xw + yz)i.

$$(a + bi)(c + di) = ac - bd + (ad + bc)i$$

$$(x + zi)(y + wi) = xz - yw + (xw + yz)i$$



AVX/AVX2-ის შეგვიძლია შემდეგი ალგორითმი გამოვიყენოთ კომპლექსური ვექტორების გასამარავლებლად:

1. გაამრავლეთ vec1 და vec2 და შეინახეთ შედეგი vec3- ში.
2. შეცვალეთ vec2– ის რეალური / წარმოსახვითი მნიშვნელობები.
3. Vec2- ის წარმოსახვითი მნიშვნელობებს შეუცვალეთ ნიშანი.
4. გაამრავლოთ vec1 და vec2 და შეინახეთ შედეგი vec4- ში.
5. გამოიყენეთ ჰორიზონტალური გამოკლება vec3– ზე და vec4– ზე, რათა მიიღოთ პასუხი vec1– ში.

კოდს აქვს სახე:

```
#include <immintrin.h>
#include <iostream>

int main() {

	__m256d vec1 = _mm256_setr_pd(4.0, 5.0, 13.0, 6.0);
	__m256d vec2 = _mm256_setr_pd(9.0, 3.0, 6.0, 7.0);
	__m256d neg = _mm256_setr_pd(1.0, -1.0, 1.0, -1.0);

	/* Step 1: Multiply vec1 and vec2 */
	__m256d vec3 = _mm256_mul_pd(vec1, vec2);

	/* Step 2: Switch the real and imaginary elements of vec2 */
	vec2 = _mm256_permute_pd(vec2, 0x5);

	/* Step 3: Negate the imaginary elements of vec2 */
	vec2 = _mm256_mul_pd(vec2, neg);

	/* Step 4: Multiply vec1 and the modified vec2 */
	__m256d vec4 = _mm256_mul_pd(vec1, vec2);

	/* Horizontally subtract the elements in vec3 and vec4 */
	vec1 = _mm256_hsub_pd(vec3, vec4);

	/* Display the elements of the result vector */
	double* res = (double*)&vec1;
	for (size_t i = 0; i < 4; ++i)
		std::cout << res[i] << "   ";
	std::cout << std::endl;
}
```

შედეგით 21 57 36 127