Antoni Wójcik, 12.04.2024r.

Optymalizacja Kodu Na Różne Architektury, gr. 5

Zadanie domowe nr 1

I Parametry procesora

Parametr	Wartość
Producent	Apple
Model	Apple M1
Architektura	ARM
Mikroarchitektura	"Firestorm" (rdzenie wydajnościowe)
	"Icestorm" (rdzenie energooszczędne)
Rdzenie	8 (4 wydajnościowe + 4 energooszczędne)
Wątki	8
Max. częstotliwość	3.2 GHz (rdzenie wydajnościowe)
	2.06 (rdzenie energooszczędne)
L1 cache	192+128 KB /rdzeń (rdzenie wydajnościowe)
	128+64 KB /rdzeń (rdzenie energooszczędne)
L2 cache	12 MB (rdzenie wydajnościowe)
	4 MB (rdzenie energooszczędne)
L3 cahce (last level cahce)	8 MB
GFlops	2 290 (FP32 Single Precision)
GFlops/rdzeń	286.25

Źródło: Wikipedia (niestety bardzo ciężko o porządną dokumentacje od Apple)
Źródło: Cpu-monkey

II Zmiany wynikające z architektury procesora

Aby tutorial działał poprawnie, dokonałem następujących zmian w `makefile`:
Zmieniłem

```
...
CC          := gcc
LINKER      := $(CC)
CFLAGS      := -O2 -Wall -msse3
```

```
LDFLAGS    := -lm
...
```

Na:

```
...
CC         := clang
LINKER     := $(CC)
CFLAGS     := -O2 -Wall -target arm64-apple-macos
LDFLAGS    := -lm
...
```

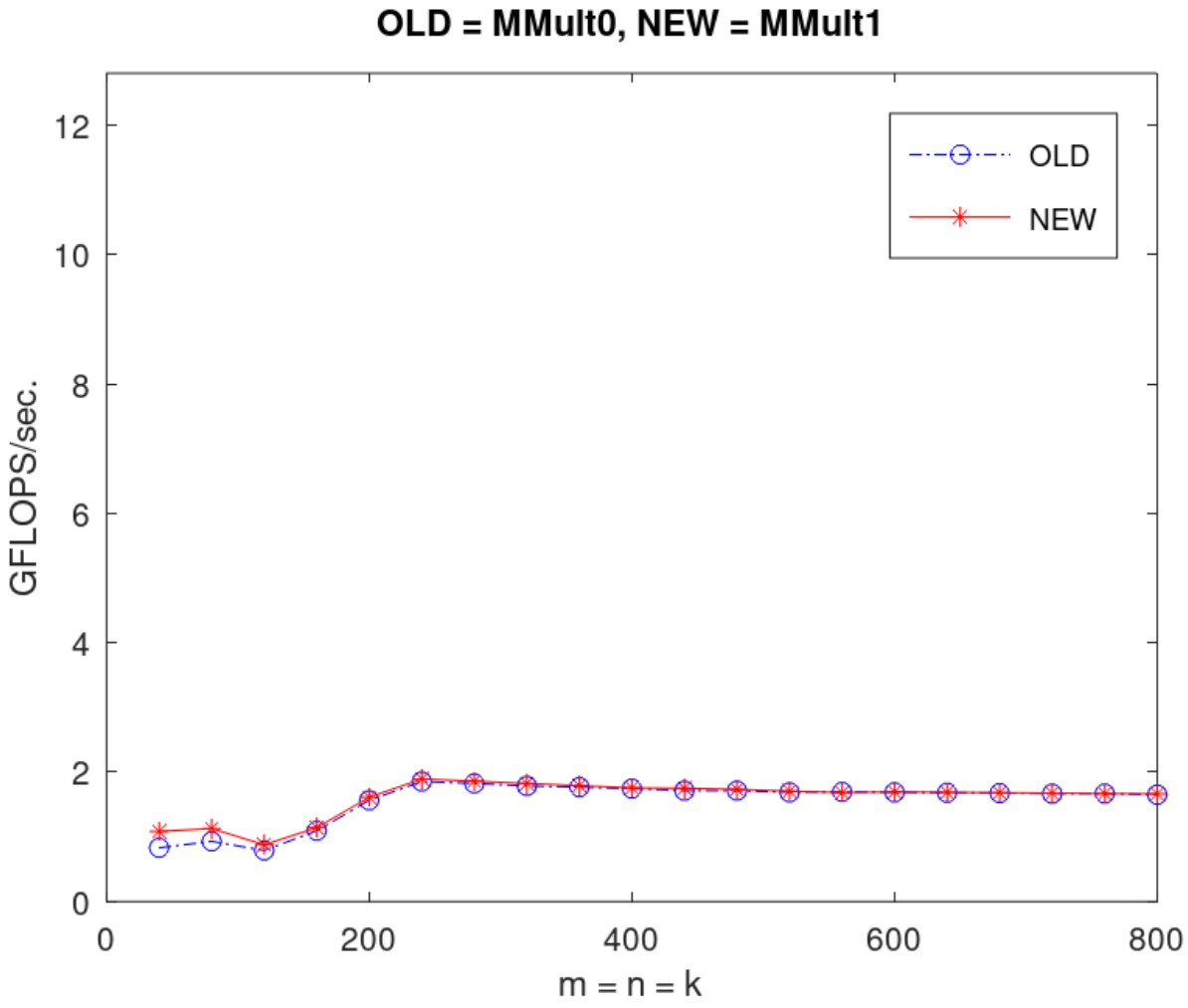
III Optymalizacje

Test konfiguracji:

Wykonując `make run` oraz `PlotAll` otrzymałem wykres prezentujący brak jakichkolwiek optymalizacji.

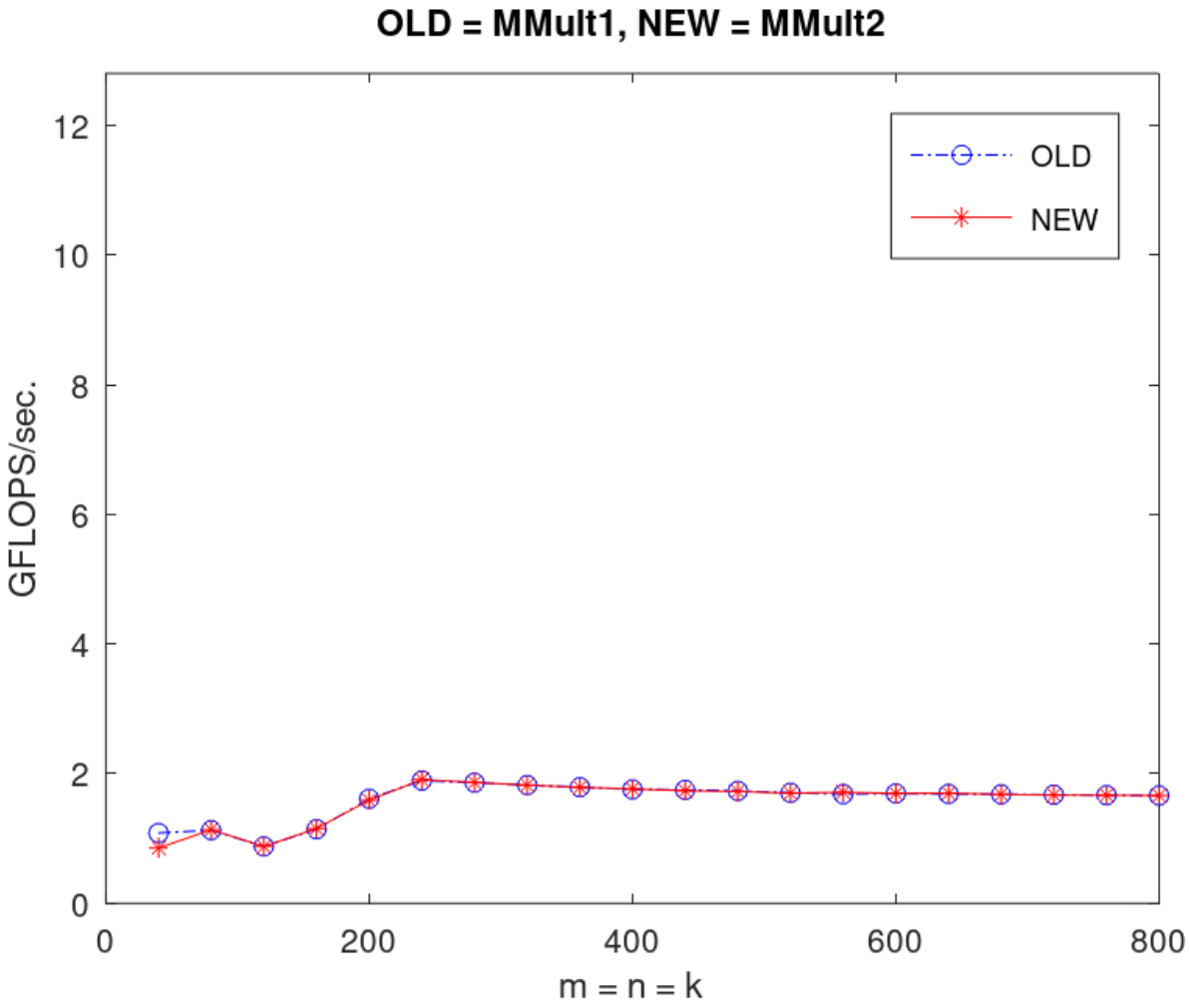
Optymalizacja 1

Dokonano optymalizacji poprzez zastosowanie funkcji `AddDot`, która jest wywoływana dla każdego elementu macierzy wynikowej C. Wynik:



Optymalizacja 2

W kolejnej wersji kodu wprowadzono kolejną optymalizację poprzez unrolling pętli wewnętrznej, czyli "rozwinięcie" jej przez 4 iteracje. Wynik:

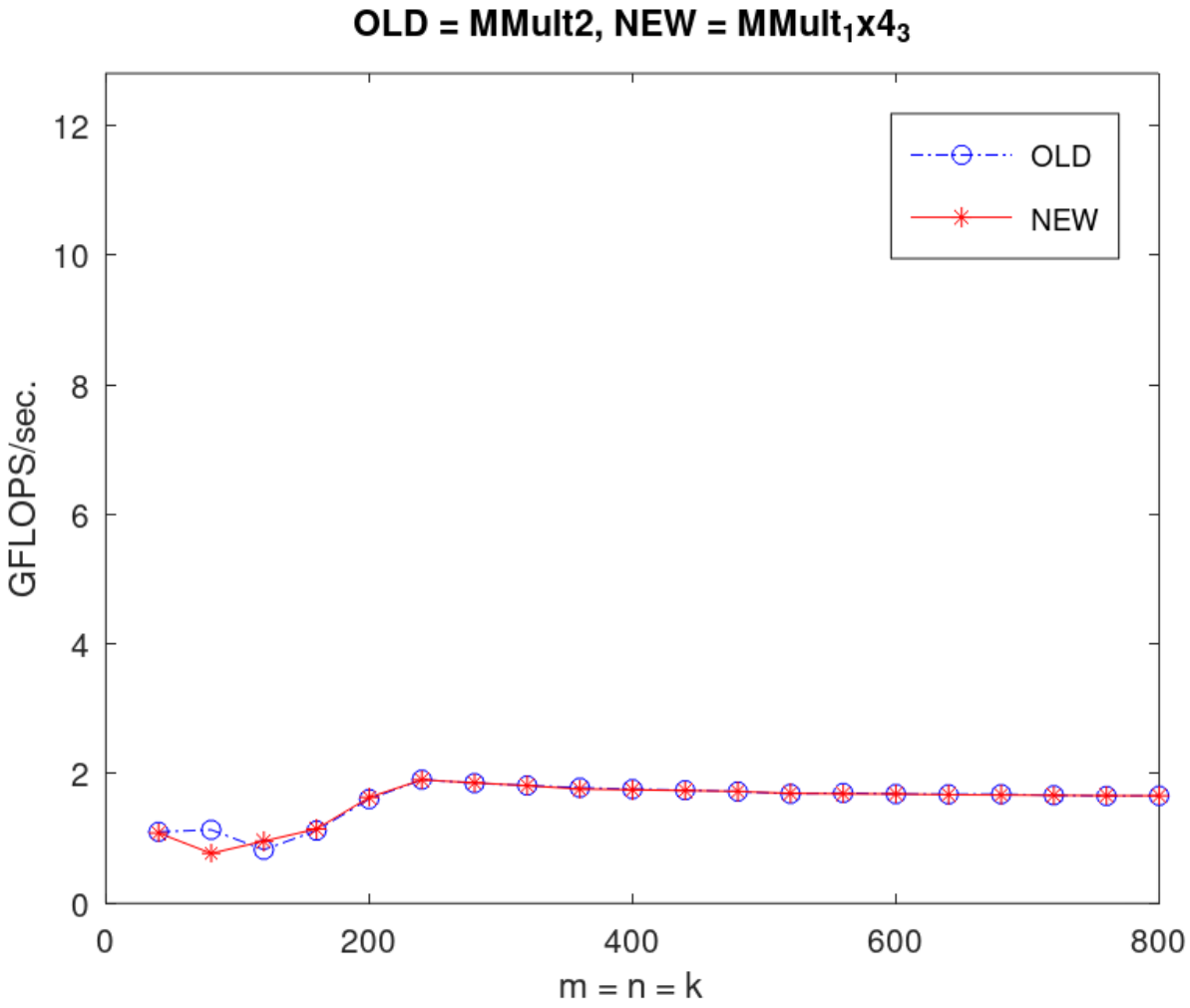


Powysze optymalizacje nie wpłynęły znacząco na wydajność, są jedynie podstawą dalszych optymalizacji.

Optymalizacja (1x4) 3

W kolejnej iteracji optymalizacji kodu wprowadzono kolejną zmianę, dzieląc operacje mnożenia macierzy na mniejsze bloki wewnątrz funkcji `MY_MMult`. Zmiany w kodzie obejmują dodanie nowej funkcji AddDot1x4,

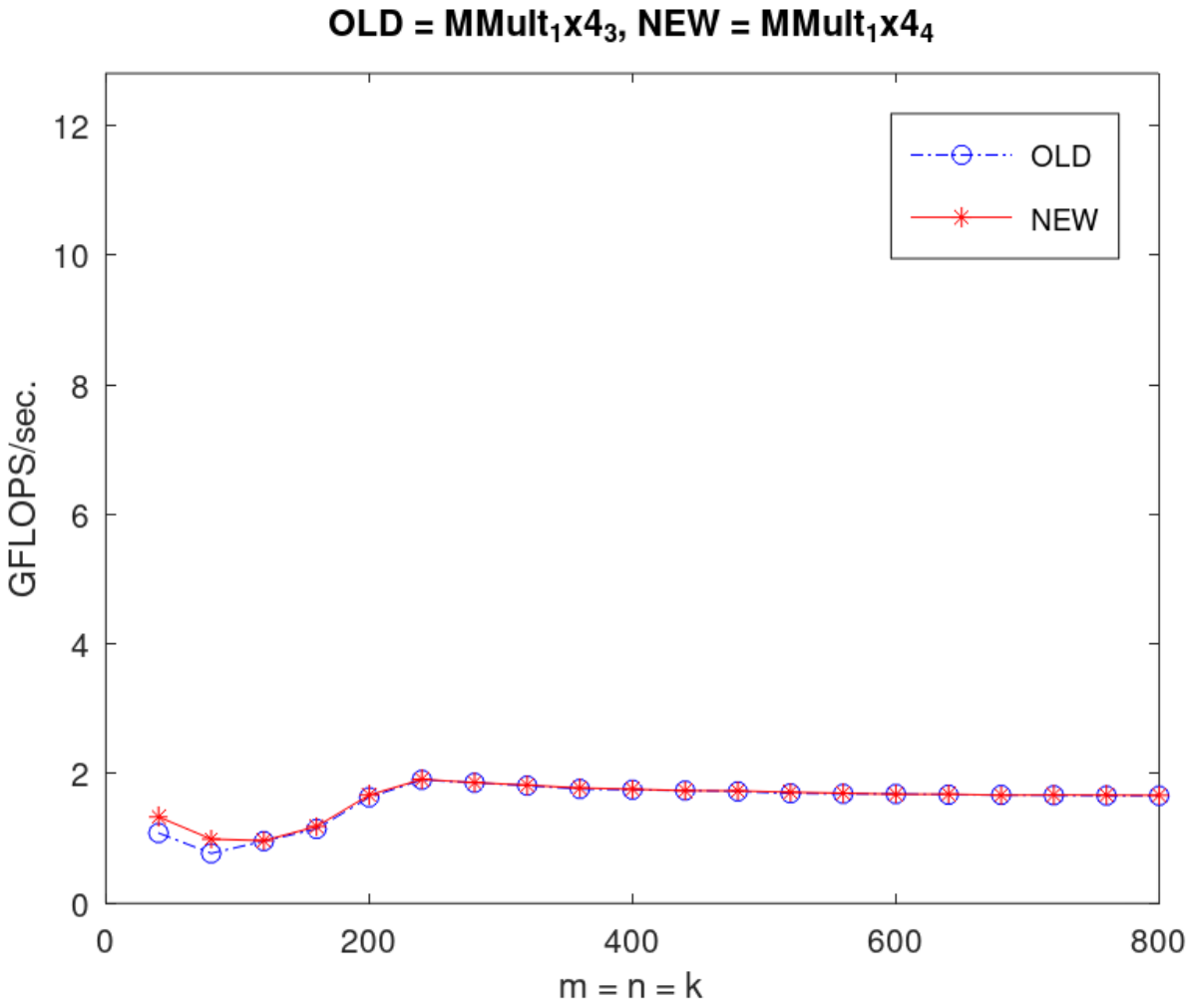
która oblicza cztery kolejne elementy macierzy wynikowej C jednocześnie. Wynik:



Optymalizacja (1x4) 4

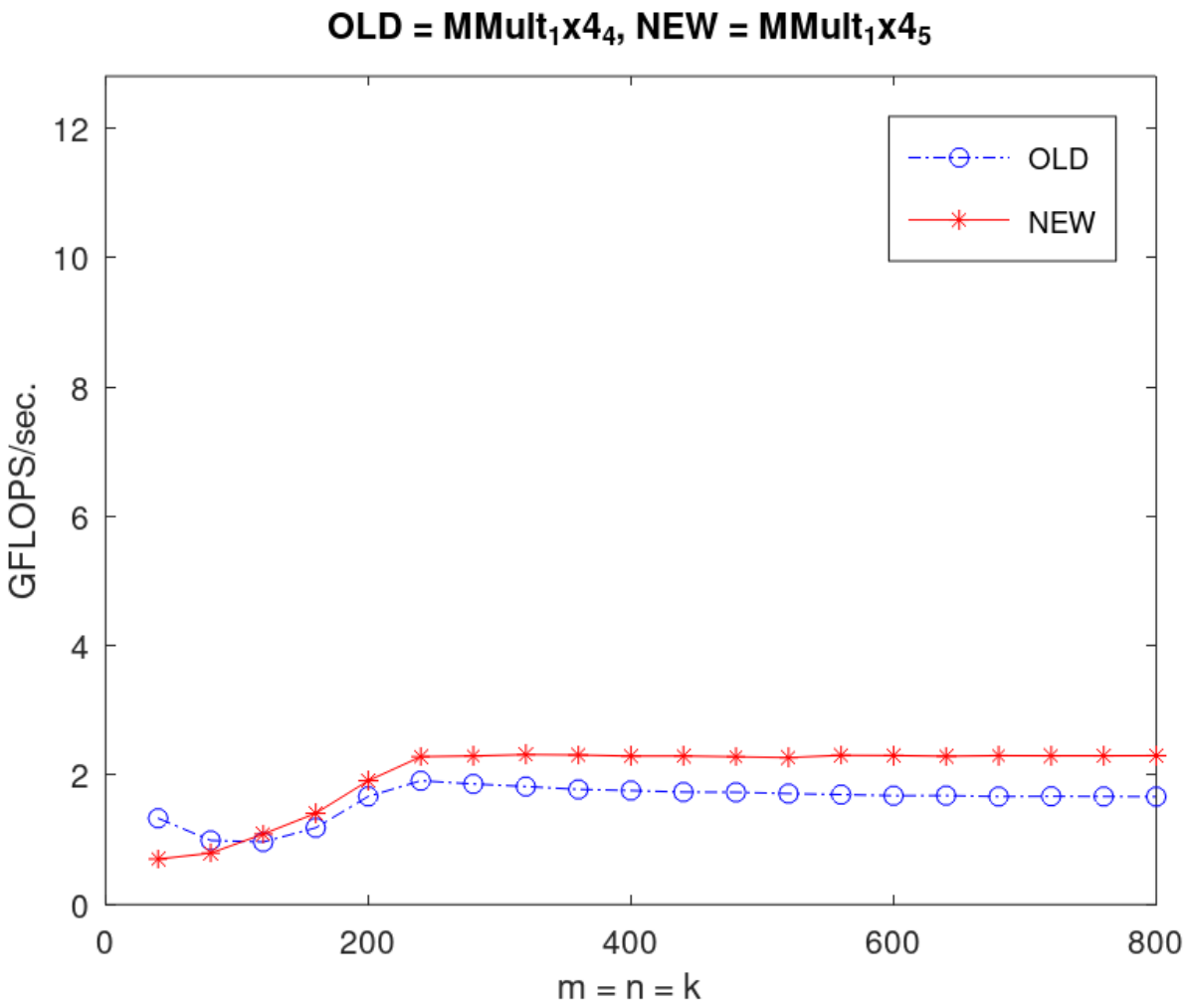
Zamiast wywoływać funkcję `AddDot`, każda z czterech iteracji w funkcji `AddDot1x4` wykonuje bezpośrednio operacje mnożenia macierzy i dodawania, co pozwala uniknąć kosztu wywoływania funkcji.

Wynik:



Optymalizacja (1x4) 5

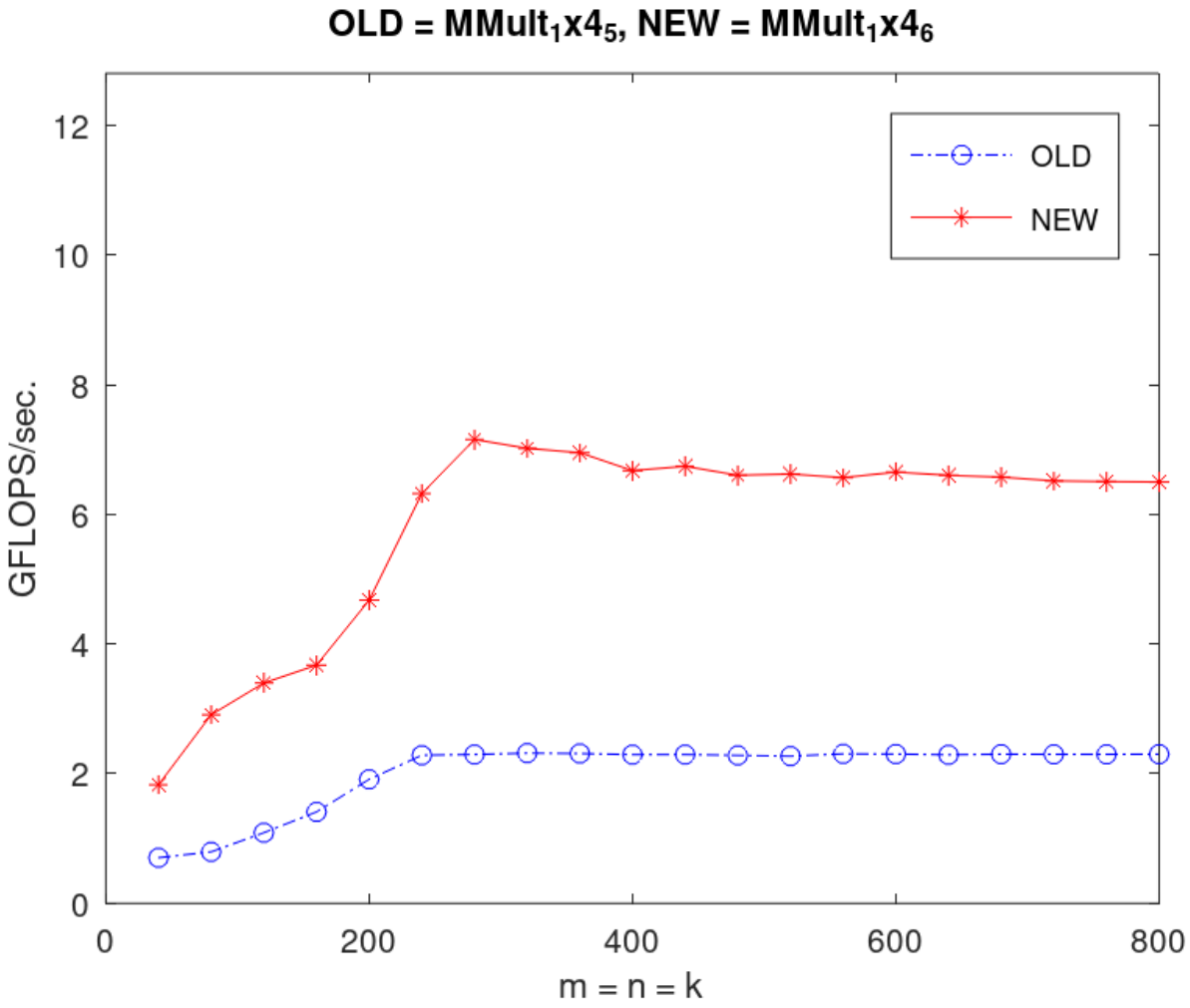
Scalenie czterech pętli `for` w funkcji `AddDot1x4` w jedną pętlę, która oblicza cztery kolejne elementy macierzy wynikowej `C` równocześnie. Wynik:



Optymalizacja (1x4) 6

W tej zmianie wprowadzono wykorzystanie rejestrów procesora do akumulacji wartości w funkcji `AddDot1x4`. Zamiast bezpośredniego zapisywania wyników do pamięci podręcznej, wartości są akumulowane w rejestrach procesora, co może znacznie zwiększyć wydajność poprzez zmniejszenie

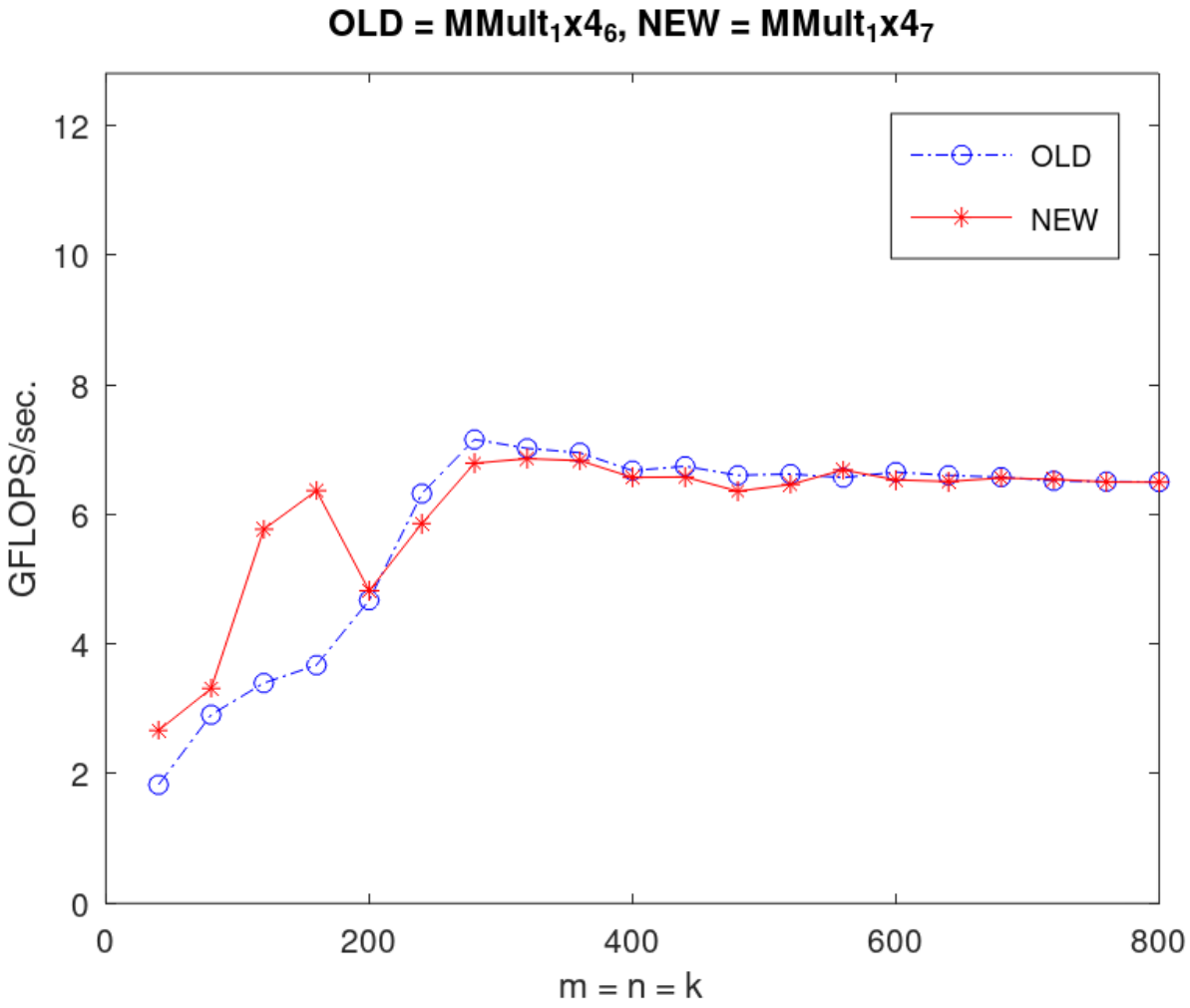
dostępu do pamięci. Zaobserwowano znaczący wzrost wydajności. Wynik:



Optymalizacja (1x4) 7

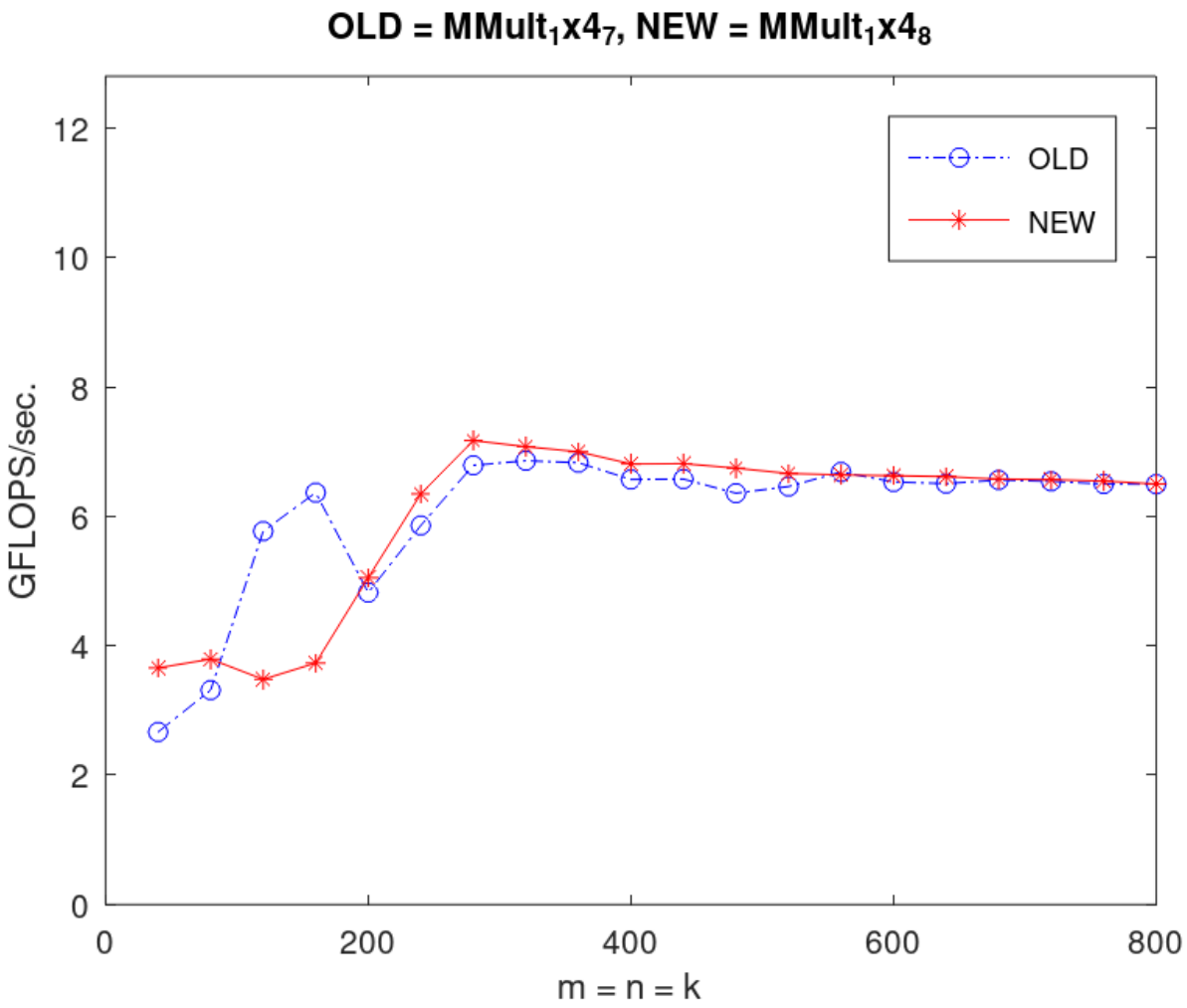
Wykorzystanie wskaźników do śledzenia położenia w czterech kolumnach macierzy `B`. Zamiast adresów kolumn macierzy `B` przekazywanych bezpośrednio do funkcji, teraz wskaźniki `bp0_pntr`, `bp1_pntr`, `bp2_pntr`, `bp3_pntr` śledzą aktualne pozycje w tych kolumnach, co pozwala na bezpośrednie

przesunięcie wskaźników wewnątrz pętli, zamiast wyliczania adresów w każdej iteracji. Wynik:



Optymalizacja (1x4) 8

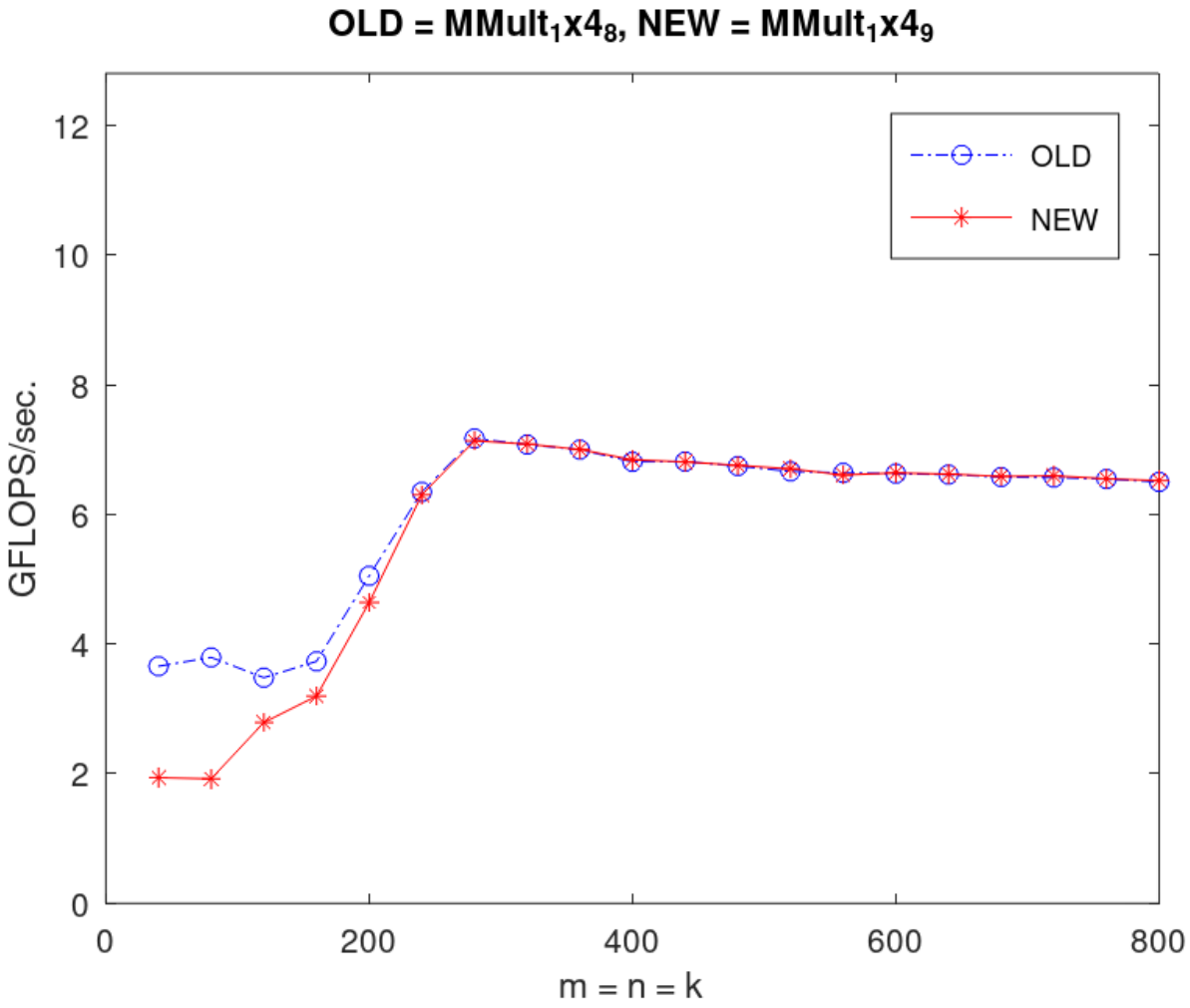
Rozwinięcie pętli o czynnik 4 (`p` zwiększane jest o 4 w każdej iteracji). Dzięki temu można teraz wykonać cztery iteracje pętli w jednej iteracji zwykłej pętli `for`. Wynik:



Optymalizacja (1x4) 9

Dodano zastosowanie adresowania pośredniego (indirect addressing) dla dostępu do elementów macierzy B. Zamiast używać wskaźników do bezpośredniego dostępu do kolejnych elementów macierzy B, teraz wskaźniki te są zwiększane o 4 w każdej iteracji pętli, co pozwala na odczytanie czterech kolejnych

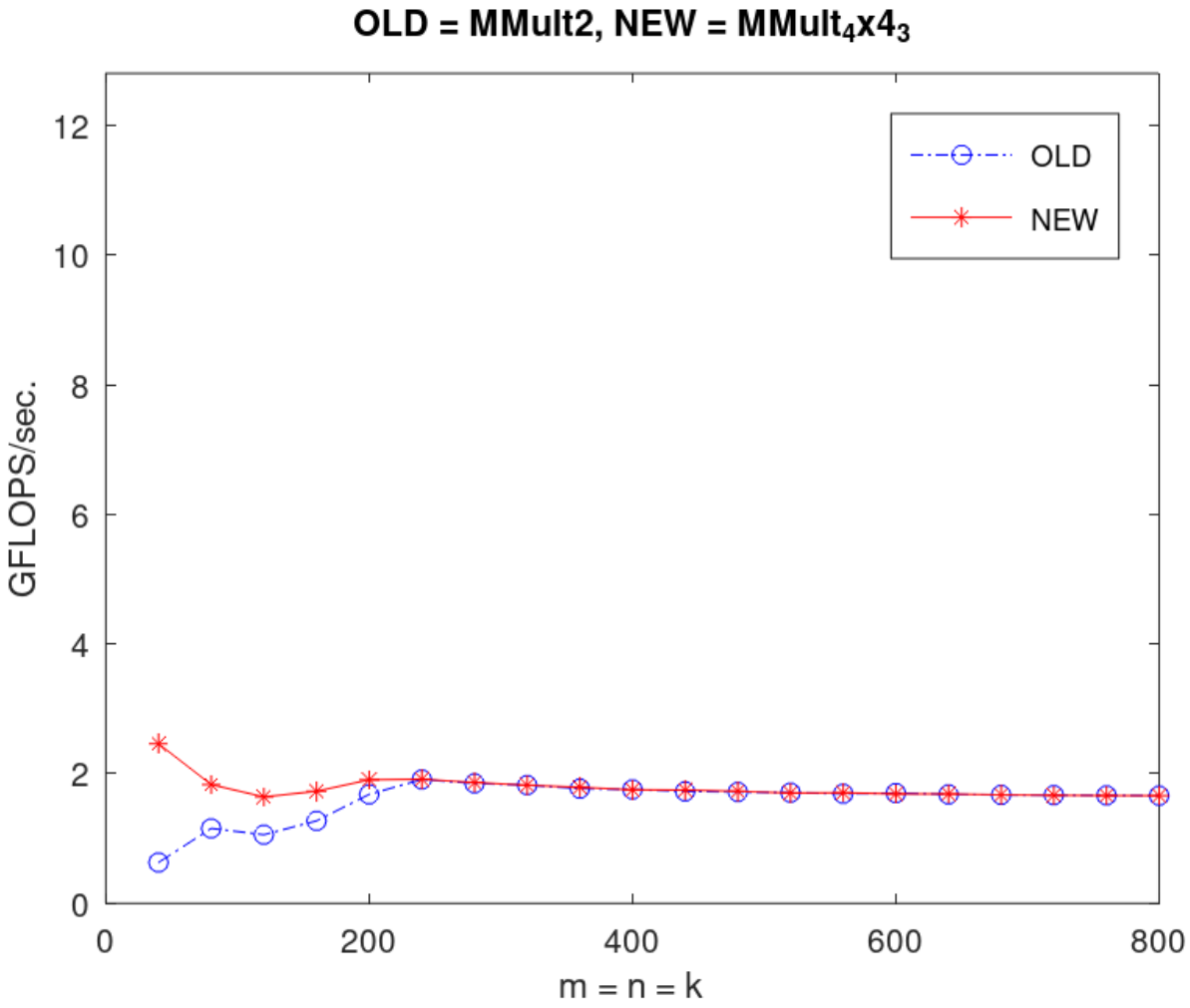
elementów macierzy w jednej iteracji pętli zamiast jednego elementu w jednej iteracji. Wynik:



Optymalizacja (4x4) 3

Dzięki unrollingu pętli i obliczaniu 4x4 bloków macierzy C naraz, zwiększa się lokalność danych. Oznacza to, że dane potrzebne do obliczeń są bardziej prawdopodobne do przechowywania w cache'u procesora, co

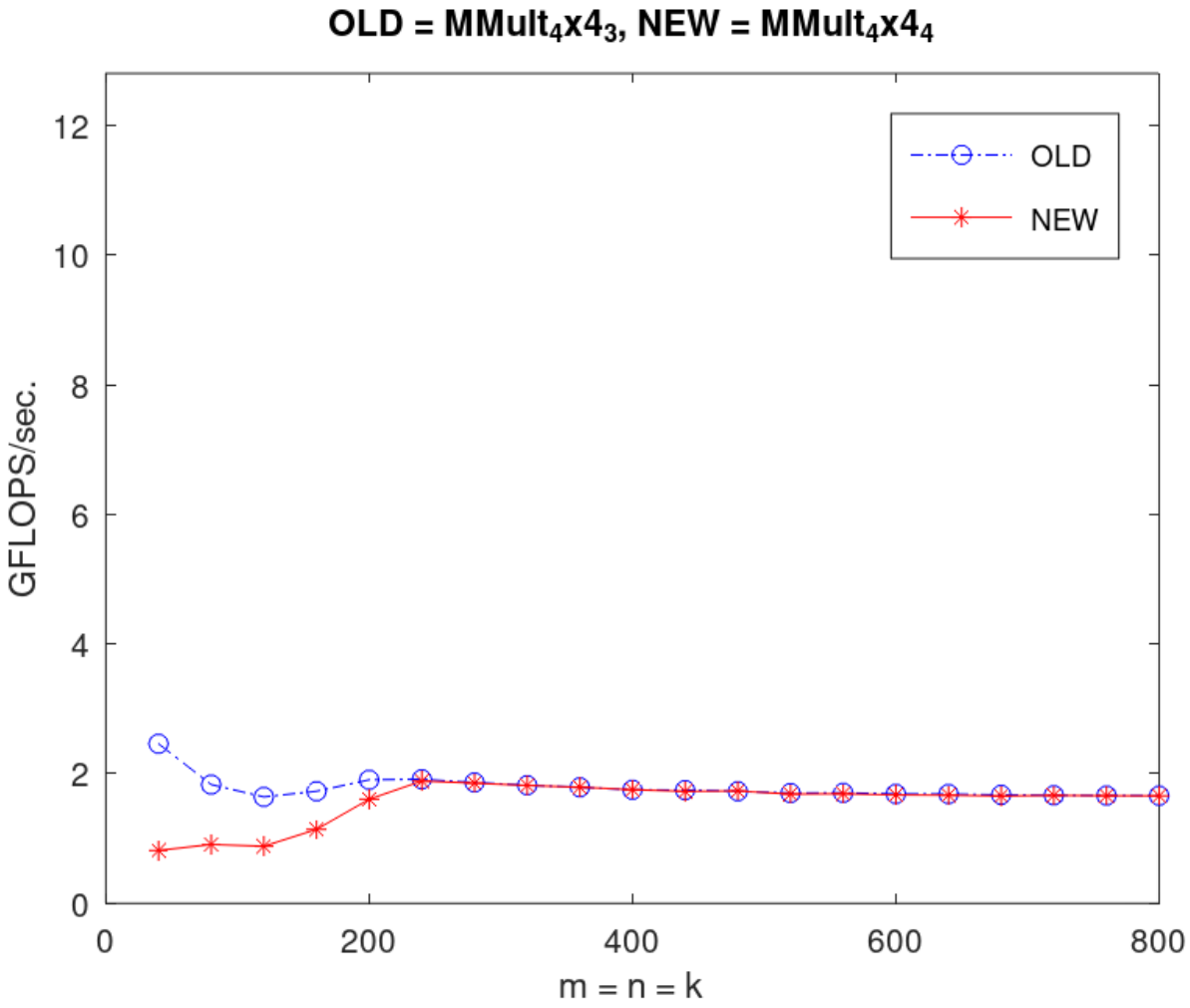
może znacznie przyspieszyć wykonywanie operacji. Wynik:



Optymalizacja (4x4) 4

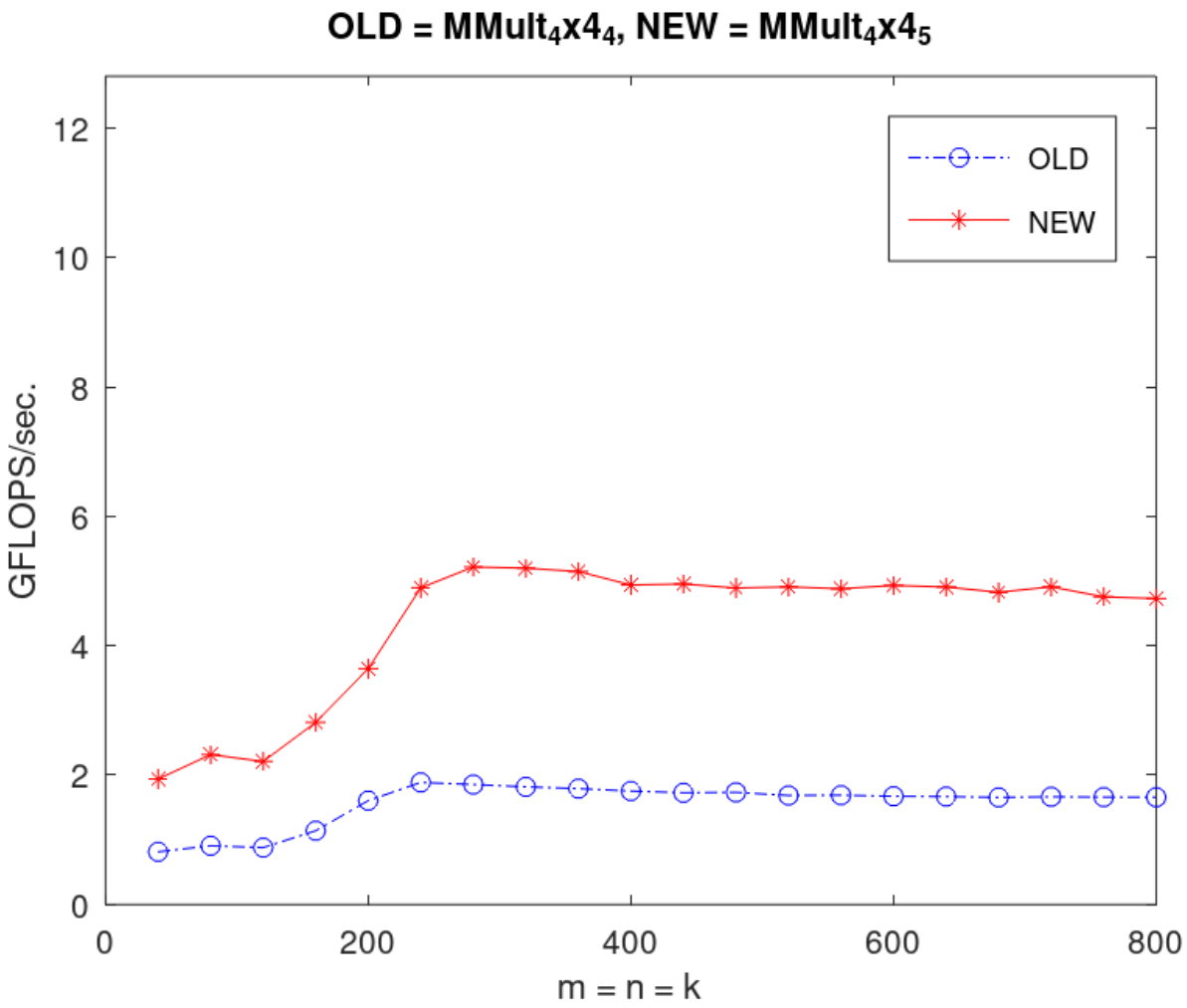
Zmieniono kod poprzez zastąpienie wywołań funkcji AddDot przez bezpośrednie obliczenia w funkcji AddDot4x4 oraz usunięcie pętli wewnętrznej. Dodatkowo, dostęp do elementów macierzy został

zoptymalizowany poprzez bezpośrednie odwołanie się do nich. Wynik:



Optymalizacja (4x4) 5

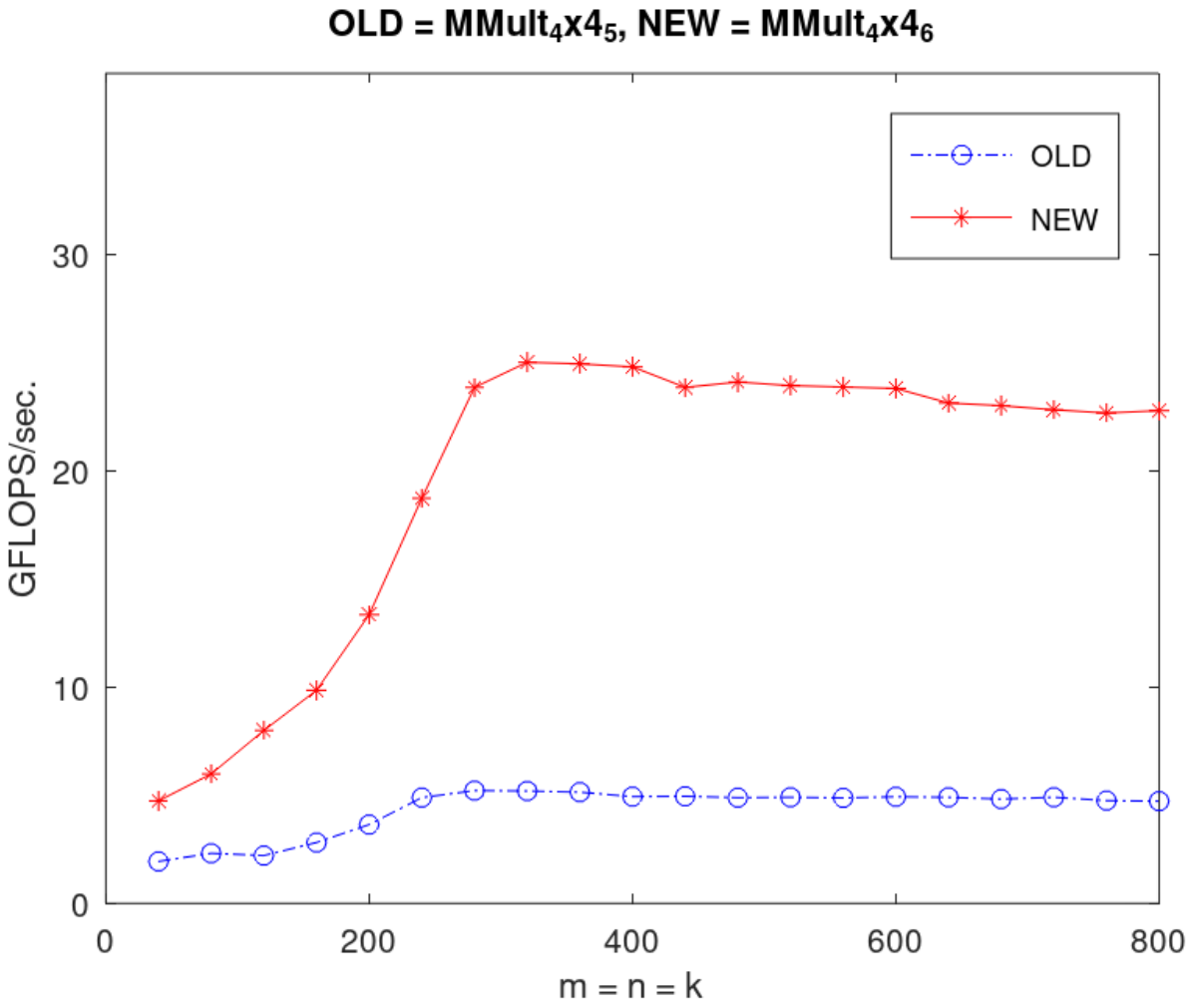
Scalenie czterech pętli w jedną, co umożliwia obliczanie czterech iloczynów skalarnych jednocześnie. Wynik:



Optymalizacja (4x4) 6

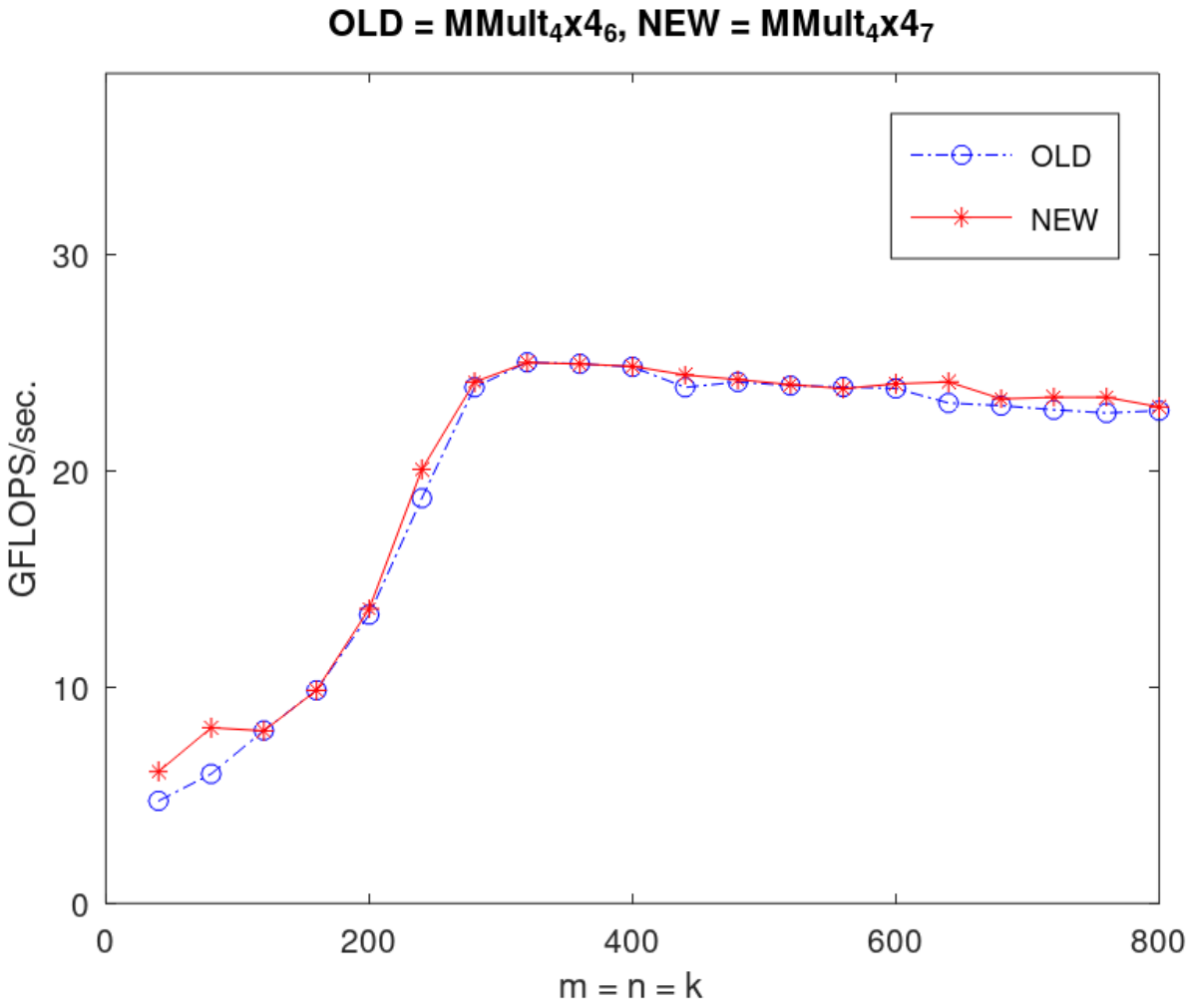
Zastosowano rejestrowanie, aby zmaksymalizować wykorzystanie rejestrów procesora. Odpowiednio, zmienne rejestrowe przechowują częściowe sumy zamiast bezpośrednio aktualizować wartości w macierzy `C` w każdej iteracji pętli. Na końcu pętli wartości zmiennych rejestrowych są dodawane do odpowiednich elementów macierzy `C`. **Ta zmiana spowodowała kolosalny wzrost wydajności, konieczne bylo a**

przeskalowanie wykresu. Wynik:



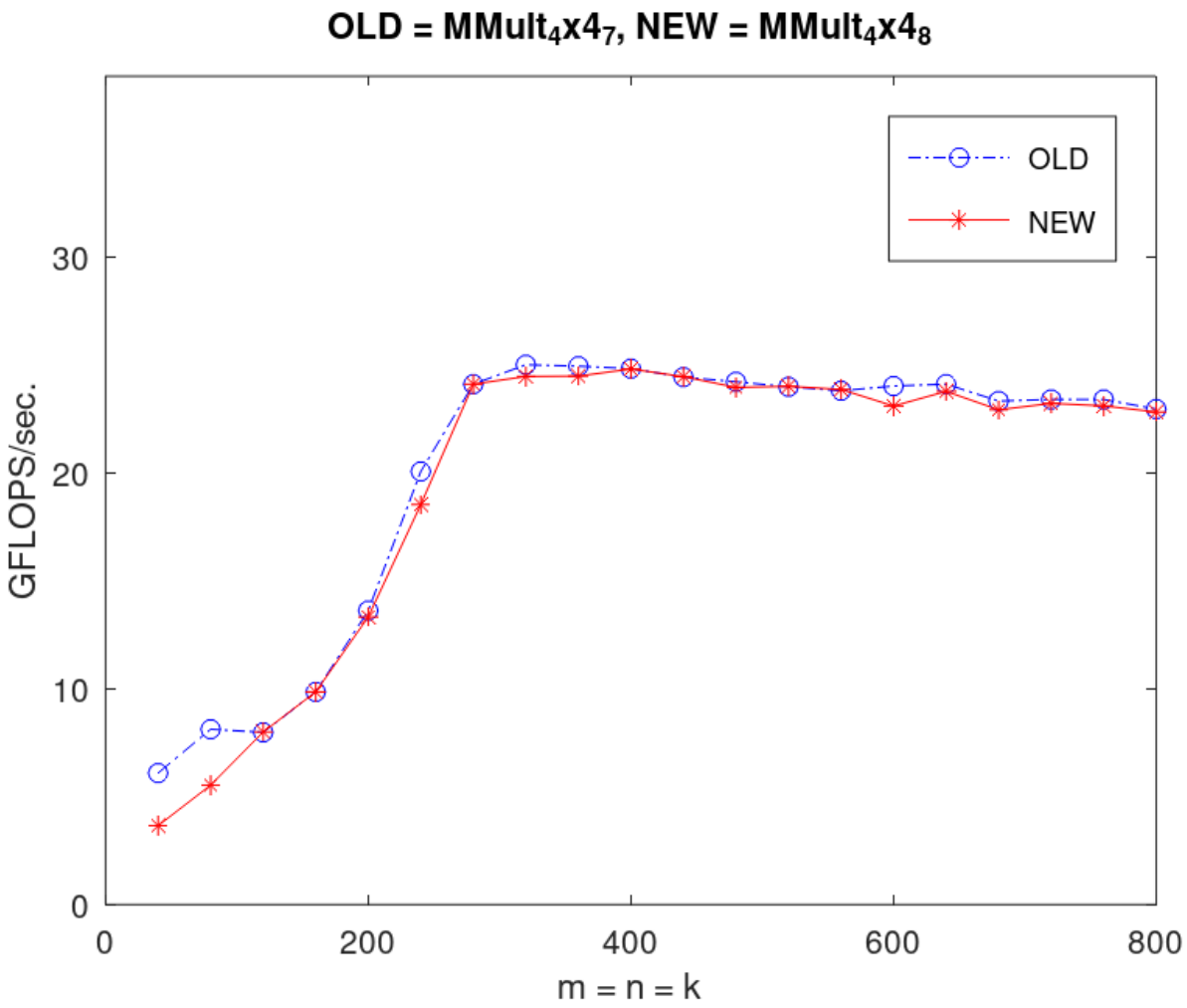
Optymalizacja (4x4) 7

Wykorzystanie wskaźników do śledzenia aktualnej pozycji w czterech kolumnach macierzy B. Wynik:



Optymalizacja (4x4) 8

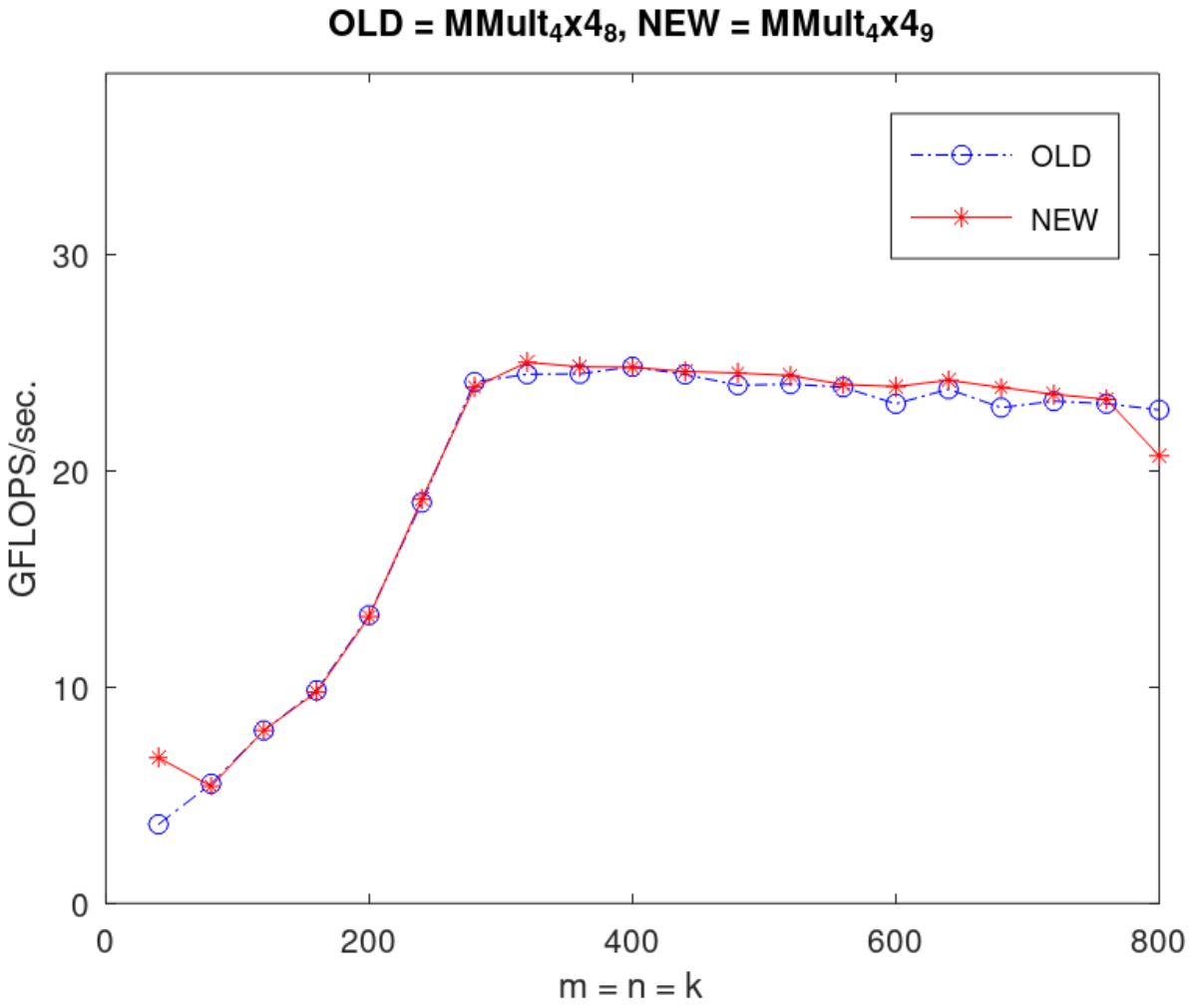
Wykorzystanie rejestrów do przechowywania elementów wiersza macierzy B. Ponadto użyto wskaźników do śledzenia aktualnej pozycji w czterech kolumnach macierzy. Wynik:



Optymalizacja (4x4) 9

W tej wersji kodu zmieniono kolejność obliczeń, aby zwiększyć wykorzystanie rejestrów i potencjalnie przyspieszyć działanie programu. Najpierw obliczane są wartości dla pierwszych dwóch wierszy macierzy

C, a następnie dla kolejnych dwóch. Wynik:



Optymalizacja (4x4) 10

Wykorzystanie instrukcji wektorowych i rejestrów wektorowych (`__m128d`) do przyspieszenia obliczeń. Zamiast przetwarzać pojedyncze liczby zmiennoprzecinkowe, obliczenia są wykonywane na parach wartości (`double`) w jednym rejestrze wektorowym. **Uwaga!**
Z uwagi na architekturę mojego procesora musiałem dokonać następujących zmian:

```
#include <arm_neon.h>  // Include NEON intrinsics header for ARM
architecture

typedef float32x2_t v2df_t;  // Define a vector type for NEON operations

void AddDot4x4(int k, double *a, int lda, double *b, int ldb, double *c,
int ldc) {
    int p;

    v2df_t
        c_00_c_10_vreg, c_01_c_11_vreg, c_02_c_12_vreg, c_03_c_13_vreg,
        c_20_c_30_vreg, c_21_c_31_vreg, c_22_c_32_vreg, c_23_c_33_vreg,
        a_0p_a_1p_vreg, a_2p_a_3p_vreg,
        b_p0_vreg, b_p1_vreg, b_p2_vreg, b_p3_vreg;

    double *b_p0_pntr, *b_p1_pntr, *b_p2_pntr, *b_p3_pntr;
```

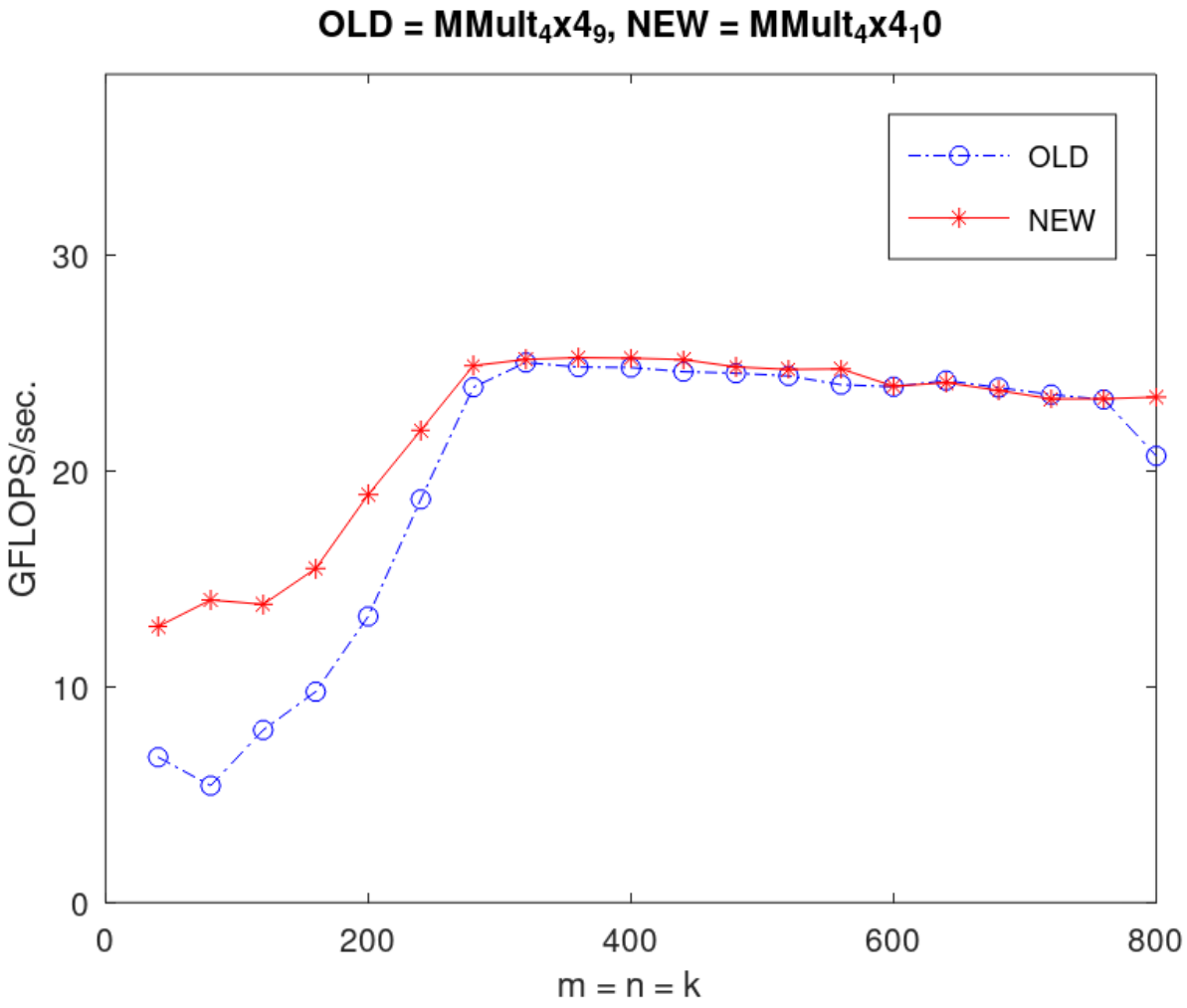
```
    b_p0_pntr = &B(0, 0);
    b_p1_pntr = &B(0, 1);
    b_p2_pntr = &B(0, 2);
    b_p3_pntr = &B(0, 3);

    c_00_c_10_vreg = vdup_n_f32(0.0f);
    c_01_c_11_vreg = vdup_n_f32(0.0f);
    c_02_c_12_vreg = vdup_n_f32(0.0f);
    c_03_c_13_vreg = vdup_n_f32(0.0f);
    c_20_c_30_vreg = vdup_n_f32(0.0f);
    c_21_c_31_vreg = vdup_n_f32(0.0f);
    c_22_c_32_vreg = vdup_n_f32(0.0f);
    c_23_c_33_vreg = vdup_n_f32(0.0f);

    for (p = 0; p < k; p++) {
        a_0p_a_1p_vreg = vld1_f32((float32_t *)&A(0, p));
        a_2p_a_3p_vreg = vld1_f32((float32_t *)&A(2, p));

        b_p0_vreg = vdup_n_f32((float32_t)*b_p0_pntr++);
        b_p1_vreg = vdup_n_f32((float32_t)*b_p1_pntr++);
        b_p2_vreg = vdup_n_f32((float32_t)*b_p2_pntr++);
        b_p3_vreg = vdup_n_f32((float32_t)*b_p3_pntr++);

        c_00_c_10_vreg = vfma_lane_f32(c_00_c_10_vreg, a_0p_a_1p_vreg,
b_p0_vreg, 0);
        c_01_c_11_vreg = vfma_lane_f32(c_01_c_11_vreg, a_0p_a_1p_vreg,
b_p1_vreg, 0);
        c_02_c_12_vreg = vfma_lane_f32(c_02_c_12_vreg, a_0p_a_1p_vreg,
b_p2_vreg, 0);
        c_03_c_13_vreg = vfma_lane_f32(c_03_c_13_vreg, a_0p_a_1p_vreg,
b_p3_vreg, 0);

        c_20_c_30_vreg = vfma_lane_f32(c_20_c_30_vreg, a_2p_a_3p_vreg,
b_p0_vreg, 0);
        c_21_c_31_vreg = vfma_lane_f32(c_21_c_31_vreg, a_2p_a_3p_vreg,
b_p1_vreg, 0);
        c_22_c_32_vreg = vfma_lane_f32(c_22_c_32_vreg, a_2p_a_3p_vreg,
b_p2_vreg, 0);
        c_23_c_33_vreg = vfma_lane_f32(c_23_c_33_vreg, a_2p_a_3p_vreg,
b_p3_vreg, 0);
    }

    C(0, 0) += c_00_c_10_vreg[0];
    C(0, 1) += c_01_c_11_vreg[0];
    C(0, 2) += c_02_c_12_vreg[0];
    C(0, 3) += c_03_c_13_vreg[0];

    C(1, 0) += c_00_c_10_vreg[1];
    C(1, 1) += c_01_c_11_vreg[1];
    C(1, 2) += c_02_c_12_vreg[1];
    C(1, 3) += c_03_c_13_vreg[1];

    C(2, 0) += c_20_c_30_vreg[0];
    C(2, 1) += c_21_c_31_vreg[0];
```

```
        C(2, 2) += c_22_c_32_vreg[0];
        C(2, 3) += c_23_c_33_vreg[0];

        C(3, 0) += c_20_c_30_vreg[1];
        C(3, 1) += c_21_c_31_vreg[1];
        C(3, 2) += c_22_c_32_vreg[1];
        C(3, 3) += c_23_c_33_vreg[1];
    }
```

W swojej wersji:

- zrezygnowałem z bibliotek `mmintrin.h` oraz `xmmintrin.h`, `pmmintrin.h` oraz `emmintrin.h`, które nie są dostępne na moim procesorze.
- zamiast tego użyłem bibliotekę `arm_neon.h`, która zawiera instrukcje wektorowe dla architektury ARM.
- zmieniłem typ wektora na `float32x2_t`, który reprezentuje wektor dwóch wartości zmiennoprzecinkowych pojedynczej precyzji.
- zmieniłem funkcje na odpowiednie dla architektury ARM, np. `vdup_n_f32()`, `vld1_f32()`, `vfma_lane_f32()`.

Wynik:



Optymalizacja (4x4) 11

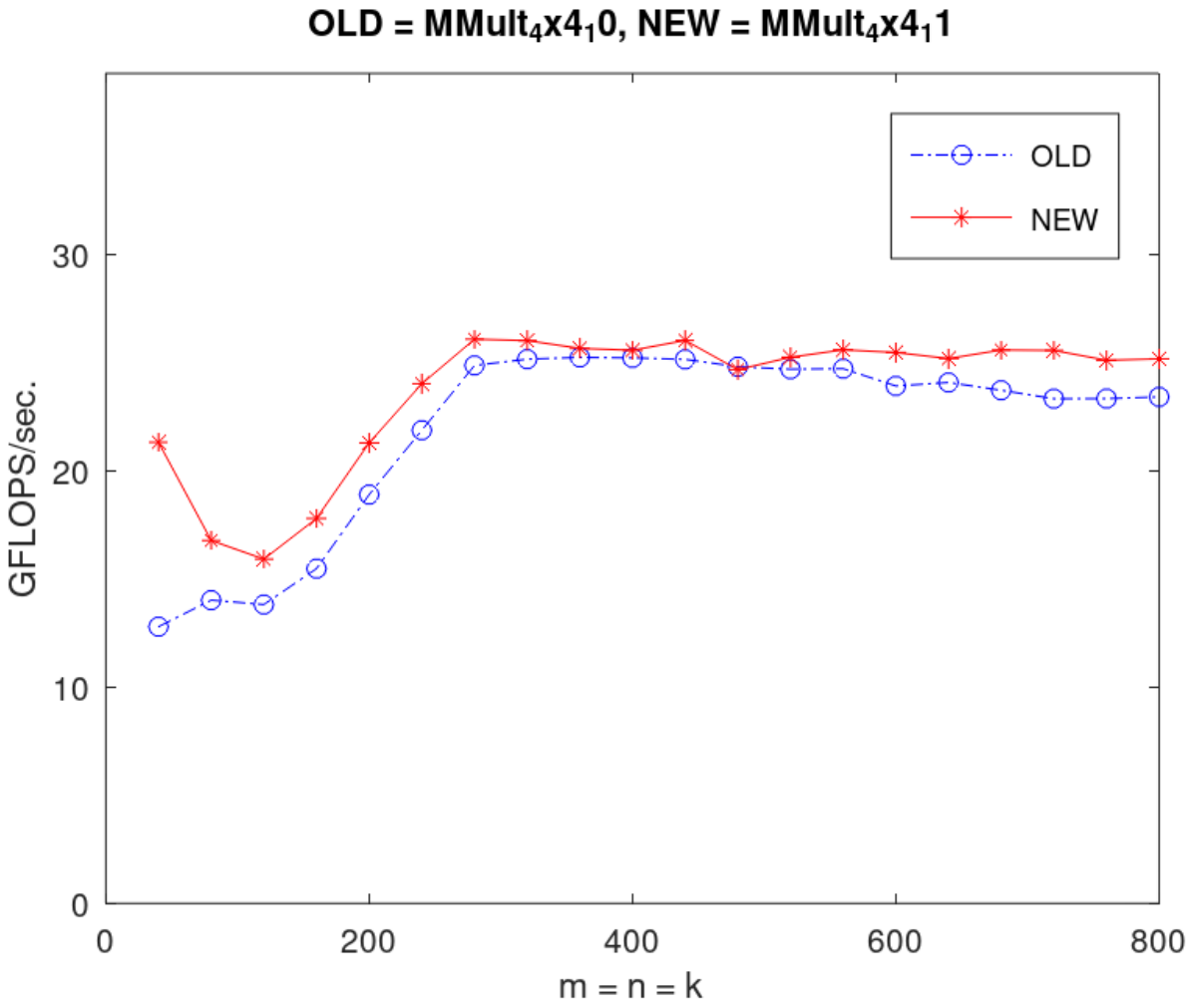
W tej wersji kodu dodano parametry `mc` i `kc`, które określają rozmiar bloków. Algorytm jest teraz zorganizowany w taki sposób, że obliczenia są wykonywane na blokach o rozmiarze `mc x kc` macierzy `C`, co może zwiększyć wydajność poprzez lepsze wykorzystanie pamięci podręcznej procesora i umożliwienie optymalizacji pętli. Dodano również funkcję `InnerKernel`, która jest odpowiedzialna za obliczenia na bloku `mc x kc`, co pozwala na lepszą modularność kodu i czytelność. Kod (musiałem dokonać znów zmiany w kodzie, aby działał na moim procesorze):

```
#include <arm_neon.h>

/* Define macros so that the matrices are stored in column-major order */
#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Block sizes */
#define mc 256
#define kc 128

#define min( i, j ) ( (i)<(j) ? (i): (j) )

/* Declare AddDot4x4 function */
void AddDot4x4( int k, double *a, int lda,  double *b, int ldb, double *c,
int ldc );

/* Routine for computing C = A * B + C */
void InnerKernel( int m, int n, int k, double *a, int lda,
                                       double *b, int ldb,
                                       double *c, int ldc );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, p, pb, ib;

  /* This time, we compute a mc x n block of C by a call to the
InnerKernel */

  for ( p=0; p<k; p+=kc ){
    pb = min( k-p, kc );
    for ( i=0; i<m; i+=mc ){
      ib = min( m-i, mc );
      InnerKernel( ib, n, pb, &A( i,p ), lda, &B(p, 0 ), ldb, &C( i,0 ),
ldc );
    }
  }
}

void InnerKernel( int m, int n, int k, double *a, int lda,
                                       double *b, int ldb,
                                       double *c, int ldc )
{
```

```
  int i;

  for (int j=0; j<n; j+=4 ){        /* Loop over the columns of C,
unrolled by 4 */
    for ( i=0; i<m; i+=4 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */

      AddDot4x4( k, &A( i,0 ), lda, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}

void AddDot4x4( int k, double *a, int lda,  double *b, int ldb, double *c,
int ldc )
{
  /* So, this routine computes a 4x4 block of matrix A

           C( 0, 0 ), C( 0, 1 ), C( 0, 2 ), C( 0, 3 ).
           C( 1, 0 ), C( 1, 1 ), C( 1, 2 ), C( 1, 3 ).
           C( 2, 0 ), C( 2, 1 ), C( 2, 2 ), C( 2, 3 ).
           C( 3, 0 ), C( 3, 1 ), C( 3, 2 ), C( 3, 3 ).

     Notice that this routine is called with c = C( i, j ) in the
     previous routine, so these are actually the elements

           C( i  , j ), C( i  , j+1 ), C( i  , j+2 ), C( i  , j+3 )
           C( i+1, j ), C( i+1, j+1 ), C( i+1, j+2 ), C( i+1, j+3 )
           C( i+2, j ), C( i+2, j+1 ), C( i+2, j+2 ), C( i+2, j+3 )
           C( i+3, j ), C( i+3, j+1 ), C( i+3, j+2 ), C( i+3, j+3 )

     in the original matrix C

     And now we use vector registers and instructions */

  int p;
  float64x2_t
    c_00_c_10_vreg,    c_01_c_11_vreg,    c_02_c_12_vreg,
c_03_c_13_vreg,
    c_20_c_30_vreg,    c_21_c_31_vreg,    c_22_c_32_vreg,
c_23_c_33_vreg,
    a_0p_a_1p_vreg,
    a_2p_a_3p_vreg,
    b_p0_vreg, b_p1_vreg, b_p2_vreg, b_p3_vreg;

  double
    /* Point to the current elements in the four columns of B */
    *b_p0_pntr, *b_p1_pntr, *b_p2_pntr, *b_p3_pntr;

  b_p0_pntr = &B( 0, 0 );
  b_p1_pntr = &B( 0, 1 );
  b_p2_pntr = &B( 0, 2 );
  b_p3_pntr = &B( 0, 3 );
```

```
    c_00_c_10_vreg = vdupq_n_f64(0.0);
    c_01_c_11_vreg = vdupq_n_f64(0.0);
    c_02_c_12_vreg = vdupq_n_f64(0.0);
    c_03_c_13_vreg = vdupq_n_f64(0.0);
    c_20_c_30_vreg = vdupq_n_f64(0.0);
    c_21_c_31_vreg = vdupq_n_f64(0.0);
    c_22_c_32_vreg = vdupq_n_f64(0.0);
    c_23_c_33_vreg = vdupq_n_f64(0.0);

    for ( p=0; p<k; p++ ){
      a_0p_a_1p_vreg = vld1q_f64( &A( 0, p ) );
      a_2p_a_3p_vreg = vld1q_f64( &A( 2, p ) );

      b_p0_vreg = vld1q_dup_f64( b_p0_pntr++ );   /* load and duplicate */
      b_p1_vreg = vld1q_dup_f64( b_p1_pntr++ );   /* load and duplicate */
      b_p2_vreg = vld1q_dup_f64( b_p2_pntr++ );   /* load and duplicate */
      b_p3_vreg = vld1q_dup_f64( b_p3_pntr++ );   /* load and duplicate */

      /* First row and second rows */
      c_00_c_10_vreg = vmlaq_f64(c_00_c_10_vreg, a_0p_a_1p_vreg, b_p0_vreg);
      c_01_c_11_vreg = vmlaq_f64(c_01_c_11_vreg, a_0p_a_1p_vreg, b_p1_vreg);
      c_02_c_12_vreg = vmlaq_f64(c_02_c_12_vreg, a_0p_a_1p_vreg, b_p2_vreg);
      c_03_c_13_vreg = vmlaq_f64(c_03_c_13_vreg, a_0p_a_1p_vreg, b_p3_vreg);

      /* Third and fourth rows */
      c_20_c_30_vreg = vmlaq_f64(c_20_c_30_vreg, a_2p_a_3p_vreg, b_p0_vreg);
      c_21_c_31_vreg = vmlaq_f64(c_21_c_31_vreg, a_2p_a_3p_vreg, b_p1_vreg);
      c_22_c_32_vreg = vmlaq_f64(c_22_c_32_vreg, a_2p_a_3p_vreg, b_p2_vreg);
      c_23_c_33_vreg = vmlaq_f64(c_23_c_33_vreg, a_2p_a_3p_vreg, b_p3_vreg);
    }

    C( 0, 0 ) += c_00_c_10_vreg[0];  C( 0, 1 ) += c_01_c_11_vreg[0];
    C( 0, 2 ) += c_02_c_12_vreg[0];  C( 0, 3 ) += c_03_c_13_vreg[0];

    C( 1, 0 ) += c_00_c_10_vreg[1];  C( 1, 1 ) += c_01_c_11_vreg[1];
    C( 1, 2 ) += c_02_c_12_vreg[1];  C( 1, 3 ) += c_03_c_13_vreg[1];

    C( 2, 0 ) += c_20_c_30_vreg[0];  C( 2, 1 ) += c_21_c_31_vreg[0];
    C( 2, 2 ) += c_22_c_32_vreg[0];  C( 2, 3 ) += c_23_c_33_vreg[0];

    C( 3, 0 ) += c_20_c_30_vreg[1];  C( 3, 1 ) += c_21_c_31_vreg[1];
    C( 3, 2 ) += c_22_c_32_vreg[1];  C( 3, 3 ) += c_23_c_33_vreg[1];
}
```

Wynik:



Optymalizacja (4x4) 12

W tej wersji kodu dodano funkcję `PackMatrixA`, która jest odpowiedzialna za pakowanie kolumn macierzy `A` do bufora `packedA`. Następnie ten bufor jest wykorzystywany w funkcji `InnerKernel`, aby obliczyć wartości macierzy `C`. Dodano również parametry `pb` i `ib` do pętli w funkcji `MY_MMult`, które określają rozmiar bloków obliczeń, co pozwala na lepszą kontrolę nad operacjami na macierzach. Kod (musiałem dokonać znów zmiany w kodzie, aby działał na moim procesorze):

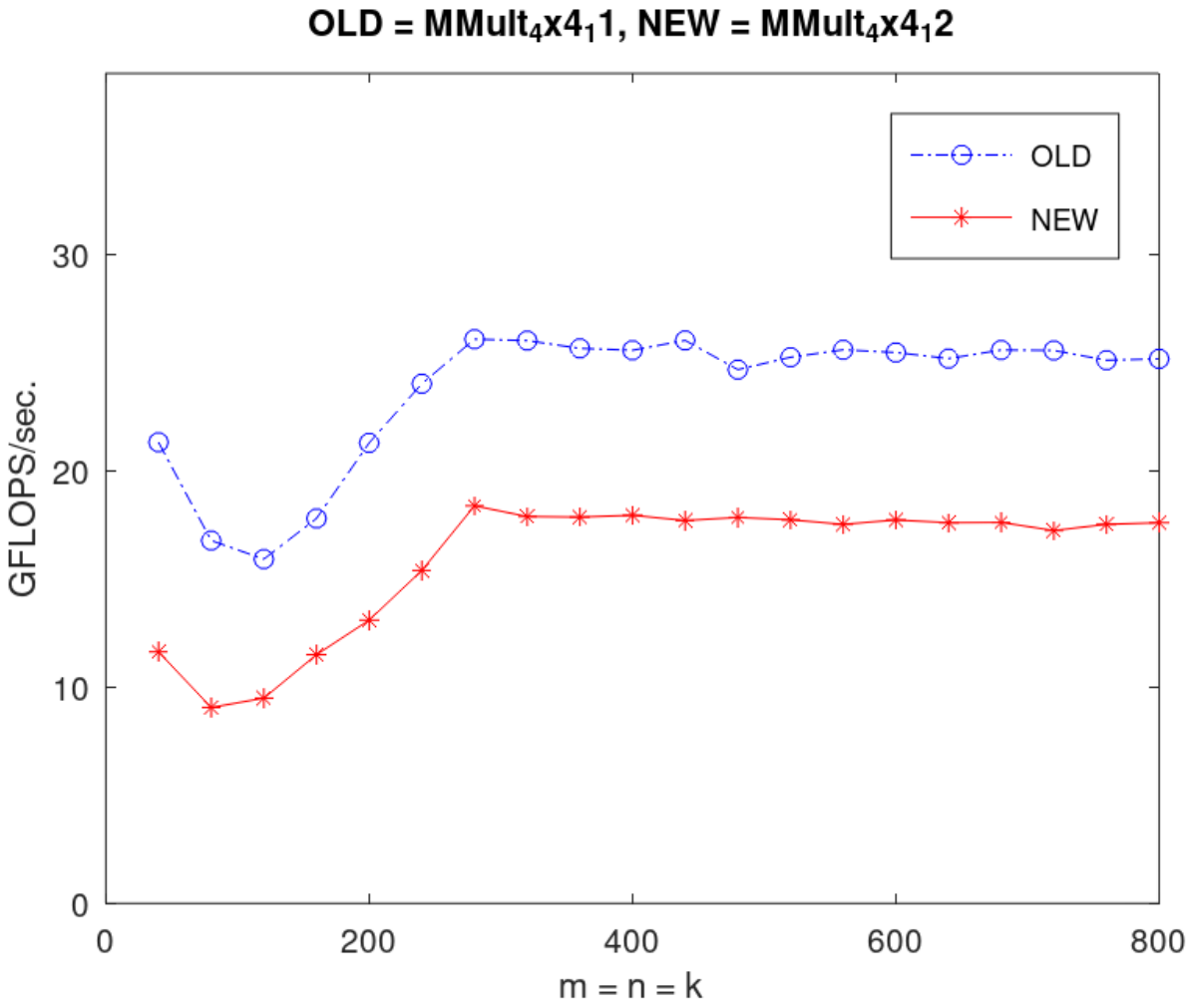
```
#include <arm_neon.h>

/* Create macros so that the matrices are stored in column-major order */
#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Block sizes */
#define mc 256
#define kc 128

#define min( i, j ) ( (i)<(j) ? (i): (j) )

/* Function declarations */
```

```
void AddDot4x4(int k, double *a, int lda, double *b, int ldb, double *c,
int ldc);
void PackMatrixA(int k, double *a, int lda, double *a_to);

void MY_MMult(int m, int n, int k, double *a, int lda, double *b, int ldb,
double *c, int ldc);

void InnerKernel(int m, int n, int k, double *a, int lda, double *b, int
ldb, double *c, int ldc)
{
    int i, j;
    double packedA[m * k];

    for (j = 0; j < n; j += 4)
    { /* Loop over the columns of C, unrolled by 4 */
        for (i = 0; i < m; i += 4)
        { /* Loop over the rows of C */
            /* Update C(i,j), C(i,j+1), C(i,j+2), and C(i,j+3) in one
routine (four inner products) */
            PackMatrixA(k, &A(i, 0), lda, &packedA[i * k]);
            AddDot4x4(k, &packedA[i * k], 4, &B(0, j), ldb, &C(i, j),
ldc);
        }
    }
}

void MY_MMult(int m, int n, int k, double *a, int lda, double *b, int ldb,
double *c, int ldc)
{
    int i, p, pb, ib;

    /* Compute a mc x n block of C by a call to the InnerKernel */
    for (p = 0; p < k; p += kc)
    {
        pb = min(k - p, kc);
        for (i = 0; i < m; i += mc)
        {
            ib = min(m - i, mc);
            InnerKernel(ib, n, pb, &A(i, p), lda, &B(p, 0), ldb, &C(i, 0),
ldc);
        }
    }
}

void PackMatrixA(int k, double *a, int lda, double *a_to)
{
    int j;

    for (j = 0; j < k; j++)
    { /* loop over columns of A */
        double *a_ij_pntr = &A(0, j);

        *a_to++ = *a_ij_pntr;
        *a_to++ = *(a_ij_pntr + 1);
```

```
        *a_to++ = *(a_ij_pntr + 2);
        *a_to++ = *(a_ij_pntr + 3);
    }
}

void AddDot4x4(int k, double *a, int lda, double *b, int ldb, double *c,
int ldc)
{
    int p;
    float64x2_t c_00_c_10_vreg, c_01_c_11_vreg, c_02_c_12_vreg,
c_03_c_13_vreg,
        c_20_c_30_vreg, c_21_c_31_vreg, c_22_c_32_vreg, c_23_c_33_vreg,
        a_0p_a_1p_vreg, a_2p_a_3p_vreg,
        b_p0_vreg, b_p1_vreg, b_p2_vreg, b_p3_vreg;

    double *b_p0_pntr, *b_p1_pntr, *b_p2_pntr, *b_p3_pntr;

    b_p0_pntr = &B(0, 0);
    b_p1_pntr = &B(0, 1);
    b_p2_pntr = &B(0, 2);
    b_p3_pntr = &B(0, 3);

    c_00_c_10_vreg = vdupq_n_f64(0.0);
    c_01_c_11_vreg = vdupq_n_f64(0.0);
    c_02_c_12_vreg = vdupq_n_f64(0.0);
    c_03_c_13_vreg = vdupq_n_f64(0.0);
    c_20_c_30_vreg = vdupq_n_f64(0.0);
    c_21_c_31_vreg = vdupq_n_f64(0.0);
    c_22_c_32_vreg = vdupq_n_f64(0.0);
    c_23_c_33_vreg = vdupq_n_f64(0.0);

    for (p = 0; p < k; p++)
    {
        a_0p_a_1p_vreg = vld1q_f64(&A(0, p));
        a_2p_a_3p_vreg = vld1q_f64(&A(2, p));

        b_p0_vreg = vdupq_n_f64(*b_p0_pntr++);
        b_p1_vreg = vdupq_n_f64(*b_p1_pntr++);
        b_p2_vreg = vdupq_n_f64(*b_p2_pntr++);
        b_p3_vreg = vdupq_n_f64(*b_p3_pntr++);

        c_00_c_10_vreg = vfmaq_f64(c_00_c_10_vreg, a_0p_a_1p_vreg,
b_p0_vreg);
        c_01_c_11_vreg = vfmaq_f64(c_01_c_11_vreg, a_0p_a_1p_vreg,
b_p1_vreg);
        c_02_c_12_vreg = vfmaq_f64(c_02_c_12_vreg, a_0p_a_1p_vreg,
b_p2_vreg);
        c_03_c_13_vreg = vfmaq_f64(c_03_c_13_vreg, a_0p_a_1p_vreg,
b_p3_vreg);

        c_20_c_30_vreg = vfmaq_f64(c_20_c_30_vreg, a_2p_a_3p_vreg,
b_p0_vreg);
        c_21_c_31_vreg = vfmaq_f64(c_21_c_31_vreg, a_2p_a_3p_vreg,
b_p1_vreg);
```

```
        c_22_c_32_vreg = vfmaq_f64(c_22_c_32_vreg, a_2p_a_3p_vreg,
b_p2_vreg);
        c_23_c_33_vreg = vfmaq_f64(c_23_c_33_vreg, a_2p_a_3p_vreg,
b_p3_vreg);
    }

    for (int idx = 0; idx < 2; ++idx)
    {
        if (idx == 0 || idx == 2 || idx == 4 || idx == 6)
        {
            c[idx] += vgetq_lane_f64(c_00_c_10_vreg, 0);
            c[idx + 1] += vgetq_lane_f64(c_01_c_11_vreg, 0);
            c[idx + 2] += vgetq_lane_f64(c_02_c_12_vreg, 0);
            c[idx + 3] += vgetq_lane_f64(c_03_c_13_vreg, 0);
            c[idx + 4] += vgetq_lane_f64(c_20_c_30_vreg, 0);
            c[idx + 5] += vgetq_lane_f64(c_21_c_31_vreg, 0);
            c[idx + 6] += vgetq_lane_f64(c_22_c_32_vreg, 0);
            c[idx + 7] += vgetq_lane_f64(c_23_c_33_vreg, 0);
        }
        else
        {
            c[idx] += vgetq_lane_f64(c_00_c_10_vreg, 1);
            c[idx + 1] += vgetq_lane_f64(c_01_c_11_vreg, 1);
            c[idx + 2] += vgetq_lane_f64(c_02_c_12_vreg, 1);
            c[idx + 3] += vgetq_lane_f64(c_03_c_13_vreg, 1);
            c[idx + 4] += vgetq_lane_f64(c_20_c_30_vreg, 1);
            c[idx + 5] += vgetq_lane_f64(c_21_c_31_vreg, 1);
            c[idx + 6] += vgetq_lane_f64(c_22_c_32_vreg, 1);
            c[idx + 7] += vgetq_lane_f64(c_23_c_33_vreg, 1);
        }
    }
}
```

Wynik:



Optymalizacja (4x4) 13

W tej wersji kodu dodano warunek `if (j == 0)` w pętli w funkcji `InnerKernel`, który sprawdza, czy aktualnie iterujemy po pierwszej kolumnie macierzy `B`. Jeśli tak, to wywoływana jest funkcja `PackMatrixA` do spakowania kolumny macierzy `A` do bufora `packedA`. Następnie ten bufor jest wykorzystywany w funkcji `InnerKernel` do obliczenia wartości macierzy `C`. Ta zmiana pozwala na zminimalizowanie powtarzalności pakowania kolumn macierzy `A` i poprawia wydajność poprzez zoptymalizowanie dostępu do danych. Kod (musiałem dokonać znów zmiany w kodzie, aby działał na moim procesorze):

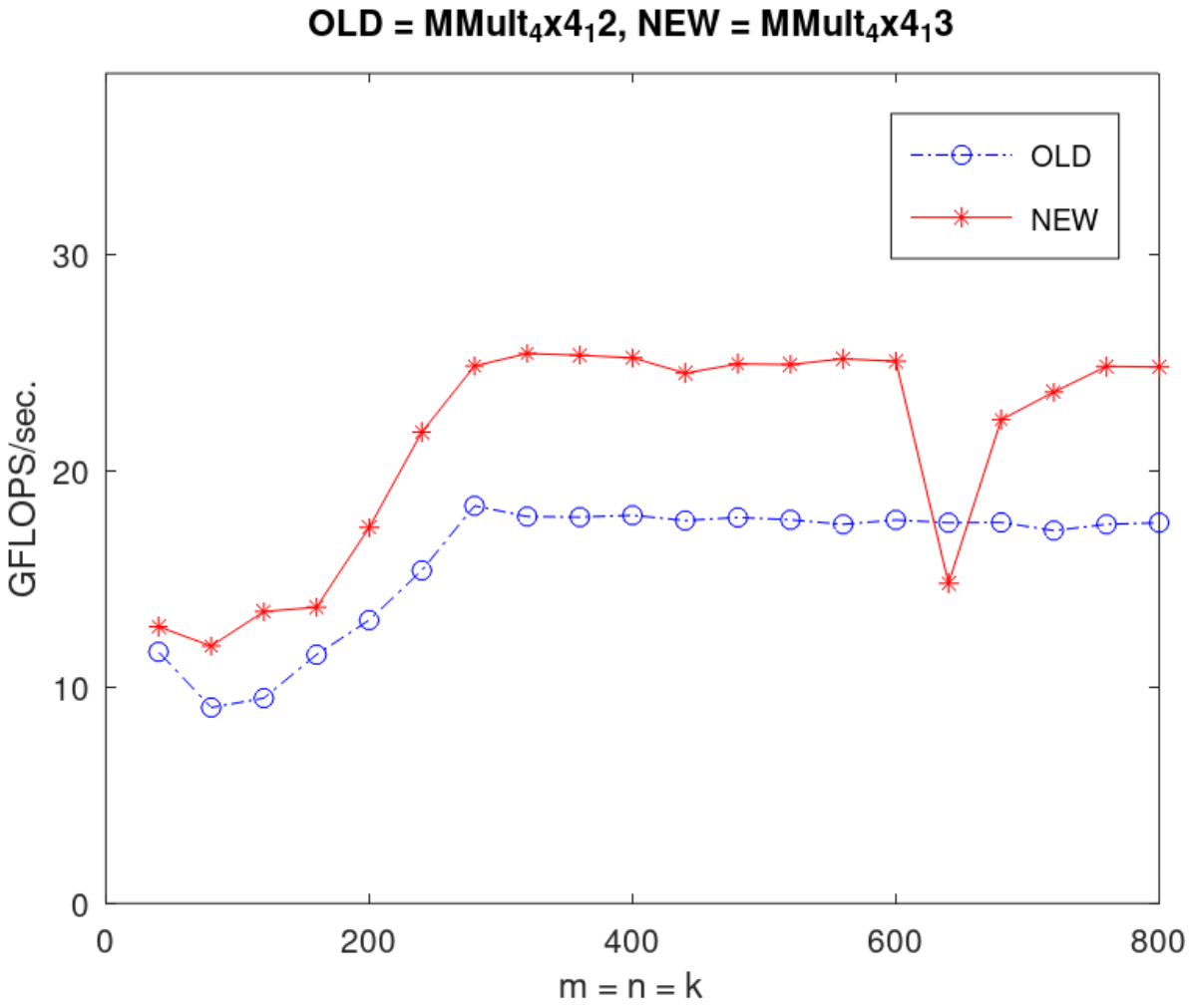
```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Block sizes */
#define mc 256
#define kc 128

#define min( i, j ) ( (i)<(j) ? (i): (j) )
```

```
/* Routine for computing C = A * B + C */

void AddDot4x4( int, double *, int, double *, int, double *, int );
void PackMatrixA( int, double *, int, double * );

void InnerKernel( int m, int n, int k, double *a, int lda,
                                       double *b, int ldb,
                                       double *c, int ldc )
{
  int i, j;
  double
    packedA[ m * k ];

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled
by 4 */
    for ( i=0; i<m; i+=4 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */
      if ( j == 0 ) PackMatrixA( k, &A( i, 0 ), lda, &packedA[ i*k ] );
      AddDot4x4( k, &packedA[ i*k ], 4, &B( 0,j ), ldb, &C( i,j ), ldc );
    }
  }
}

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, p, pb, ib;

  /* This time, we compute a mc x n block of C by a call to the
InnerKernel */

  for ( p=0; p<k; p+=kc ){
    pb = min( k-p, kc );
    for ( i=0; i<m; i+=mc ){
      ib = min( m-i, mc );
      InnerKernel( ib, n, pb, &A( i,p ), lda, &B(p, 0 ), ldb, &C( i,0 ),
ldc );
    }
  }
}



void PackMatrixA( int k, double *a, int lda, double *a_to )
{
  int j;

  for( j=0; j<k; j++){  /* loop over columns of A */
    double
      *a_ij_pntr = &A( 0, j );

    *a_to++ = *a_ij_pntr;
```

```
    *a_to++ = *(a_ij_pntr+1);
    *a_to++ = *(a_ij_pntr+2);
    *a_to++ = *(a_ij_pntr+3);
  }
}

#include <arm_neon.h>

typedef float32x2_t v2df_t;

void AddDot4x4(int k, double *a, int lda, double *b, int ldb, double *c,
int ldc) {
    int p;
    v2df_t
    c_00_c_10_vreg, c_01_c_11_vreg, c_02_c_12_vreg, c_03_c_13_vreg,
    c_20_c_30_vreg, c_21_c_31_vreg, c_22_c_32_vreg, c_23_c_33_vreg,
    a_0p_a_1p_vreg,
    a_2p_a_3p_vreg,
    b_p0_vreg, b_p1_vreg, b_p2_vreg, b_p3_vreg;

    double
    *b_p0_pntr, *b_p1_pntr, *b_p2_pntr, *b_p3_pntr;

    b_p0_pntr = &B(0, 0);
    b_p1_pntr = &B(0, 1);
    b_p2_pntr = &B(0, 2);
    b_p3_pntr = &B(0, 3);

    c_00_c_10_vreg = vdup_n_f32(0);
    c_01_c_11_vreg = vdup_n_f32(0);
    c_02_c_12_vreg = vdup_n_f32(0);
    c_03_c_13_vreg = vdup_n_f32(0);
    c_20_c_30_vreg = vdup_n_f32(0);
    c_21_c_31_vreg = vdup_n_f32(0);
    c_22_c_32_vreg = vdup_n_f32(0);
    c_23_c_33_vreg = vdup_n_f32(0);

    for (p = 0; p < k; p++) {
        a_0p_a_1p_vreg = vld1_f32((float32_t *)a);
        a_2p_a_3p_vreg = vld1_f32((float32_t *)(a + 2));
        a += 4;

        b_p0_vreg = vdup_n_f32(*b_p0_pntr++);
        b_p1_vreg = vdup_n_f32(*b_p1_pntr++);
        b_p2_vreg = vdup_n_f32(*b_p2_pntr++);
        b_p3_vreg = vdup_n_f32(*b_p3_pntr++);

        c_00_c_10_vreg = vmla_lane_f32(c_00_c_10_vreg, a_0p_a_1p_vreg,
b_p0_vreg, 0);
        c_01_c_11_vreg = vmla_lane_f32(c_01_c_11_vreg, a_0p_a_1p_vreg,
b_p1_vreg, 0);
        c_02_c_12_vreg = vmla_lane_f32(c_02_c_12_vreg, a_0p_a_1p_vreg,
b_p2_vreg, 0);
        c_03_c_13_vreg = vmla_lane_f32(c_03_c_13_vreg, a_0p_a_1p_vreg,
```

```
b_p3_vreg, 0);

        c_20_c_30_vreg = vmla_lane_f32(c_20_c_30_vreg, a_2p_a_3p_vreg,
b_p0_vreg, 0);
        c_21_c_31_vreg = vmla_lane_f32(c_21_c_31_vreg, a_2p_a_3p_vreg,
b_p1_vreg, 0);
        c_22_c_32_vreg = vmla_lane_f32(c_22_c_32_vreg, a_2p_a_3p_vreg,
b_p2_vreg, 0);
        c_23_c_33_vreg = vmla_lane_f32(c_23_c_33_vreg, a_2p_a_3p_vreg,
b_p3_vreg, 0);
    }

    C(0, 0) += vget_lane_f32(c_00_c_10_vreg, 0);
    C(0, 1) += vget_lane_f32(c_01_c_11_vreg, 0);
    C(0, 2) += vget_lane_f32(c_02_c_12_vreg, 0);
    C(0, 3) += vget_lane_f32(c_03_c_13_vreg, 0);

    C(1, 0) += vget_lane_f32(c_00_c_10_vreg, 1);
    C(1, 1) += vget_lane_f32(c_01_c_11_vreg, 1);
    C(1, 2) += vget_lane_f32(c_02_c_12_vreg, 1);
    C(1, 3) += vget_lane_f32(c_03_c_13_vreg, 1);

    C(2, 0) += vget_lane_f32(c_20_c_30_vreg, 0);
    C(2, 1) += vget_lane_f32(c_21_c_31_vreg, 0);
    C(2, 2) += vget_lane_f32(c_22_c_32_vreg, 0);
    C(2, 3) += vget_lane_f32(c_23_c_33_vreg, 0);

    C(3, 0) += vget_lane_f32(c_20_c_30_vreg, 1);
    C(3, 1) += vget_lane_f32(c_21_c_31_vreg, 1);
    C(3, 2) += vget_lane_f32(c_22_c_32_vreg, 1);
    C(3, 3) += vget_lane_f32(c_23_c_33_vreg, 1);
}
```

Wynik:



Optymalizacja (4x4) 14

Funkcja `MY_MMult` została zmodyfikowana tak, aby przekazywać dodatkowy argument `first_time` do funkcji `InnerKernel`, który informuje, czy jest to pierwsze wywołanie tej funkcji w danym cyklu obliczeń. Natomiast funkcja `InnerKernel` została zmodyfikowana w taki sposób, że teraz wewnętrzne pętle są odwrócone - najpierw pakowana jest macierz B, a następnie wykonywane są obliczenia. Dodatkowo, dodano funkcję `PackMatrixB`, która jest wywoływana w pętli zewnętrznej funkcji `InnerKernel`, aby zapakować macierz B do postaci zorientowanej wierszami.

Kod (musiałem dokonać znów zmiany w kodzie, aby działał na moim procesorze):

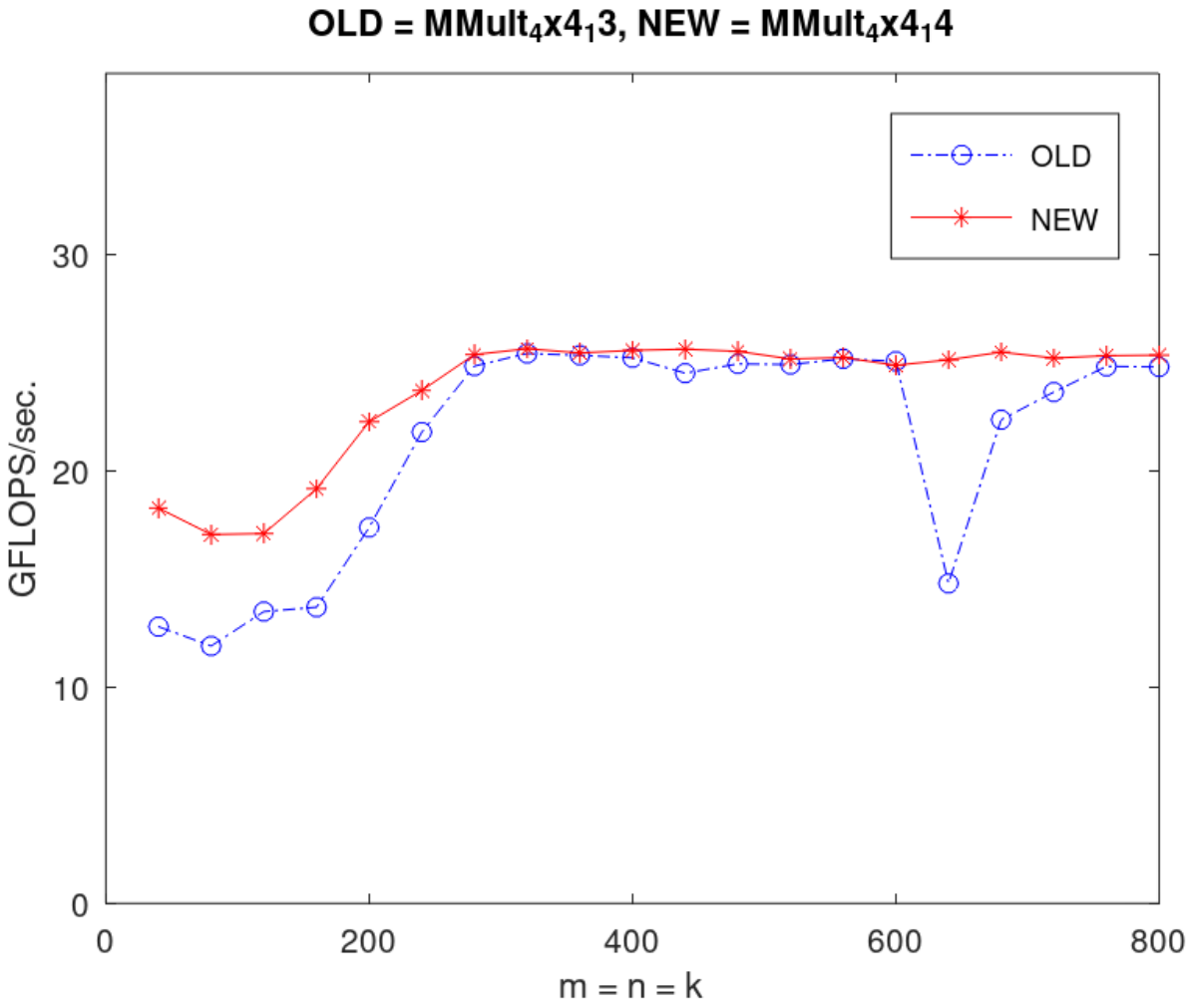
```
/* Create macros so that the matrices are stored in column-major order */

#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Block sizes */
#define mc 256
#define kc 128

#define min( i, j ) ( (i)<(j) ? (i): (j) )
```

```
/* Routine for computing C = A * B + C */

void AddDot4x4( int, double *, int, double *, int, double *, int );
void PackMatrixA( int, double *, int, double * );
void PackMatrixB( int, double *, int, double * );
void InnerKernel( int, int, int, double *, int, double *, int, double *,
int, int );

void MY_MMult( int m, int n, int k, double *a, int lda,
                                    double *b, int ldb,
                                    double *c, int ldc )
{
  int i, p, pb, ib;

  /* This time, we compute a mc x n block of C by a call to the
InnerKernel */

  for ( p=0; p<k; p+=kc ){
    pb = min( k-p, kc );
    for ( i=0; i<m; i+=mc ){
      ib = min( m-i, mc );
      InnerKernel( ib, n, pb, &A( i,p ), lda, &B(p, 0 ), ldb, &C( i,0 ),
ldc, i==0 );
    }
  }
}

void InnerKernel( int m, int n, int k, double *a, int lda,
                                       double *b, int ldb,
                                       double *c, int ldc, int first_time
)
{
  int i, j;
  double
    packedA[ m * k ], packedB[ k*n ];

  for ( j=0; j<n; j+=4 ){        /* Loop over the columns of C, unrolled
by 4 */
    PackMatrixB( k, &B( 0, j ), ldb, &packedB[ j*k ] );
    for ( i=0; i<m; i+=4 ){        /* Loop over the rows of C */
      /* Update C( i,j ), C( i,j+1 ), C( i,j+2 ), and C( i,j+3 ) in
     one routine (four inner products) */
      if ( j == 0 )
    PackMatrixA( k, &A( i, 0 ), lda, &packedA[ i*k ] );
      AddDot4x4( k, &packedA[ i*k ], 4, &packedB[ j*k ], k, &C( i,j ), ldc
);
    }
  }
}

void PackMatrixA( int k, double *a, int lda, double *a_to )
{
  int j;
```

```
  for( j=0; j<k; j++){  /* loop over columns of A */
    double
      *a_ij_pntr = &A( 0, j );

    *a_to     = *a_ij_pntr;
    *(a_to+1) = *(a_ij_pntr+1);
    *(a_to+2) = *(a_ij_pntr+2);
    *(a_to+3) = *(a_ij_pntr+3);

    a_to += 4;
  }
}

void PackMatrixB( int k, double *b, int ldb, double *b_to )
{
  int i;
  double
    *b_i0_pntr = &B( 0, 0 ), *b_i1_pntr = &B( 0, 1 ),
    *b_i2_pntr = &B( 0, 2 ), *b_i3_pntr = &B( 0, 3 );

  for( i=0; i<k; i++){  /* loop over rows of B */
    *b_to++ = *b_i0_pntr++;
    *b_to++ = *b_i1_pntr++;
    *b_to++ = *b_i2_pntr++;
    *b_to++ = *b_i3_pntr++;
  }
}

#include <arm_neon.h>

typedef float32x2_t v2df_t;

void AddDot4x4(int k, double *a, int lda, double *b, int ldb, double *c,
int ldc) {
    /* Define ARM NEON registers for the computation */
    float32x2_t c_00_c_10_vreg = vdup_n_f32(0);
    float32x2_t c_01_c_11_vreg = vdup_n_f32(0);
    float32x2_t c_02_c_12_vreg = vdup_n_f32(0);
    float32x2_t c_03_c_13_vreg = vdup_n_f32(0);
    float32x2_t c_20_c_30_vreg = vdup_n_f32(0);
    float32x2_t c_21_c_31_vreg = vdup_n_f32(0);
    float32x2_t c_22_c_32_vreg = vdup_n_f32(0);
    float32x2_t c_23_c_33_vreg = vdup_n_f32(0);

    /* Loop through each element of the matrices */
    for (int p = 0; p < k; p++) {
        float32x2_t a_0p_a_1p_vreg = vld1_f32((float32_t *)a);
        float32x2_t a_2p_a_3p_vreg = vld1_f32((float32_t *)(a + 2));
        float32x2_t b_p0_vreg = vdup_n_f32(*b);
        float32x2_t b_p1_vreg = vdup_n_f32(*(b + 1));
        float32x2_t b_p2_vreg = vdup_n_f32(*(b + 2));
        float32x2_t b_p3_vreg = vdup_n_f32(*(b + 3));
```

```
        /* Perform matrix multiplication using ARM NEON intrinsics */
        c_00_c_10_vreg = vfma_lane_f32(c_00_c_10_vreg, a_0p_a_1p_vreg,
b_p0_vreg, 0);
        c_01_c_11_vreg = vfma_lane_f32(c_01_c_11_vreg, a_0p_a_1p_vreg,
b_p1_vreg, 0);
        c_02_c_12_vreg = vfma_lane_f32(c_02_c_12_vreg, a_0p_a_1p_vreg,
b_p2_vreg, 0);
        c_03_c_13_vreg = vfma_lane_f32(c_03_c_13_vreg, a_0p_a_1p_vreg,
b_p3_vreg, 0);
        c_20_c_30_vreg = vfma_lane_f32(c_20_c_30_vreg, a_2p_a_3p_vreg,
b_p0_vreg, 0);
        c_21_c_31_vreg = vfma_lane_f32(c_21_c_31_vreg, a_2p_a_3p_vreg,
b_p1_vreg, 0);
        c_22_c_32_vreg = vfma_lane_f32(c_22_c_32_vreg, a_2p_a_3p_vreg,
b_p2_vreg, 0);
        c_23_c_33_vreg = vfma_lane_f32(c_23_c_33_vreg, a_2p_a_3p_vreg,
b_p3_vreg, 0);

        /* Move to the next column of matrix B */
        b += 4;
        /* Move to the next row of matrix A */
        a += 4;
    }

    /* Store the results back to matrix C */
    c[0] += c_00_c_10_vreg[0];
    c[1] += c_01_c_11_vreg[0];
    c[2] += c_02_c_12_vreg[0];
    c[3] += c_03_c_13_vreg[0];
    c[4] += c_00_c_10_vreg[1];
    c[5] += c_01_c_11_vreg[1];
    c[6] += c_02_c_12_vreg[1];
    c[7] += c_03_c_13_vreg[1];
    c[8] += c_20_c_30_vreg[0];
    c[9] += c_21_c_31_vreg[0];
    c[10] += c_22_c_32_vreg[0];
    c[11] += c_23_c_33_vreg[0];
    c[12] += c_20_c_30_vreg[1];
    c[13] += c_21_c_31_vreg[1];
    c[14] += c_22_c_32_vreg[1];
    c[15] += c_23_c_33_vreg[1];
}
```

Wynik:



Optymalizacja (4x4) 15

W tej wersji kodu dokonano kolejnych zmian, dodając dodatkową stałą `nb` oraz statyczną tablicę `packedB`. Stała `nb` określa rozmiar bloku wierszowego macierzy B, który jest używany do pakowania macierzy B w funkcji `InnerKernel`. Natomiast tablica `packedB` jest teraz zadeklarowana jako statyczna, co oznacza, że jest przechowywana w pamięci globalnej i jest współdzielona między różnymi wywołaniami funkcji `InnerKernel`. To zmniejsza liczbę alokacji i dealokacji pamięci, co może poprawić wydajność w przypadku wielokrotnego wywoływania funkcji `InnerKernel` w jednym cyklu obliczeń. Dodatkowo, w funkcji `InnerKernel`, tablica `packedB` jest teraz inicjalizowana tylko raz, jeśli jest to pierwsze wywołanie tej funkcji w danym cyklu obliczeń.

Kod (musiałem dokonać znów zmiany w kodzie, aby działał na moim procesorze):

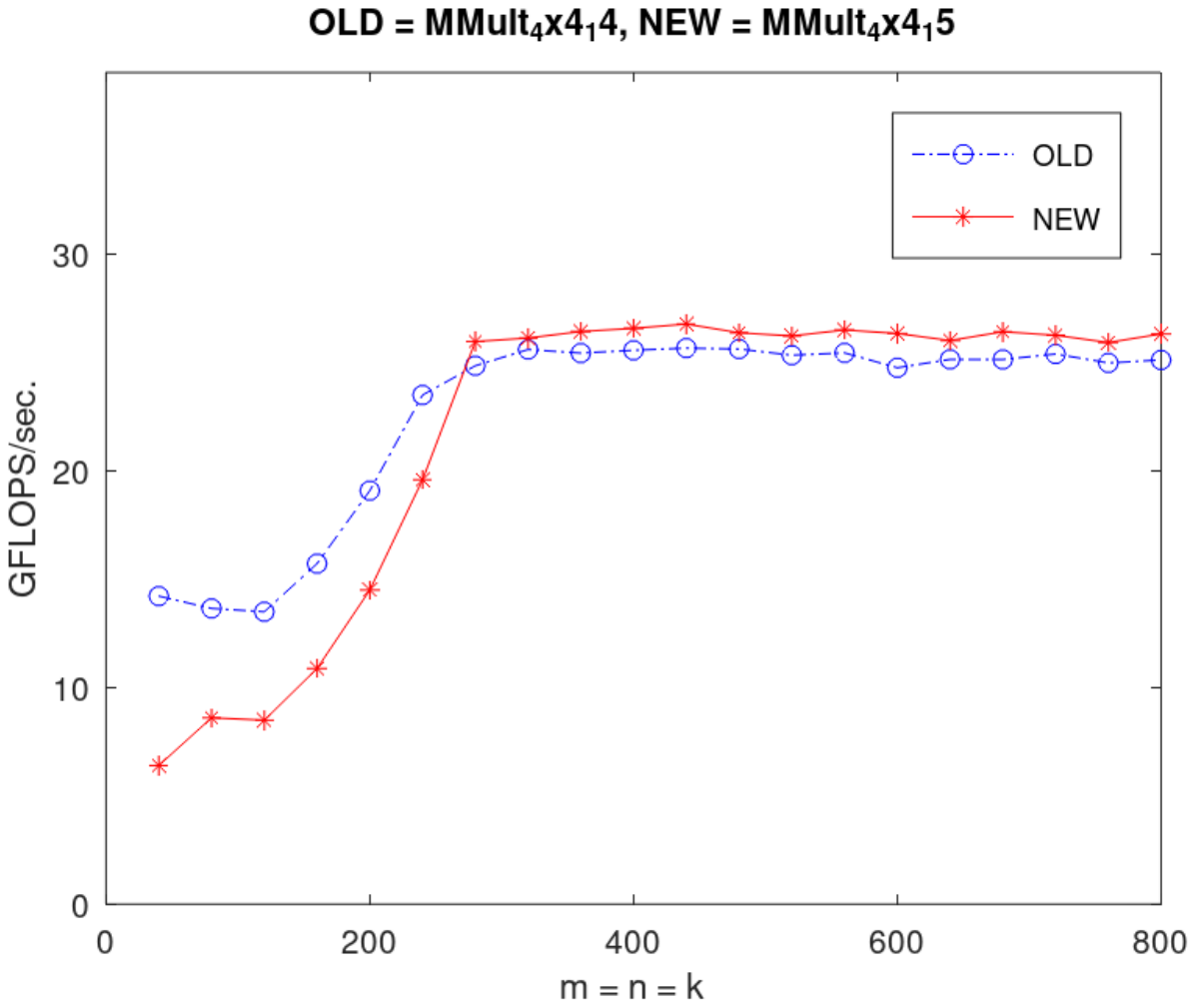
```
#include <arm_neon.h>

/* Define macros for column-major order */
#define A(i,j) a[ (j)*lda + (i) ]
#define B(i,j) b[ (j)*ldb + (i) ]
#define C(i,j) c[ (j)*ldc + (i) ]

/* Block sizes */
#define mc 256
```

```
#define kc 128
#define nb 1000

#define min( i, j ) ( (i)<(j) ? (i): (j) )

/* Function prototypes */
void AddDot4x4(int k, double *a, int lda, double *b, int ldb, double *c,
int ldc);
void PackMatrixA(int k, double *a, int lda, double *a_to);
void PackMatrixB(int k, double *b, int ldb, double *b_to);
void InnerKernel(int m, int n, int k, double *a, int lda, double *b, int
ldb, double *c, int ldc, int first_time);

/* Main matrix multiplication function */
void MY_MMult(int m, int n, int k, double *a, int lda, double *b, int ldb,
double *c, int ldc) {
    int i, p, pb, ib;

    /* Compute a mc x n block of C by a call to InnerKernel */
    for (p = 0; p < k; p += kc) {
        pb = min(k - p, kc);
        for (i = 0; i < m; i += mc) {
            ib = min(m - i, mc);
            InnerKernel(ib, n, pb, &A(i, p), lda, &B(p, 0), ldb, &C(i, 0),
ldc, i == 0);
        }
    }
}

/* Inner kernel function */
void InnerKernel(int m, int n, int k, double *a, int lda, double *b, int
ldb, double *c, int ldc, int first_time) {
    int i, j;
    double packedA[m * k];
    static double packedB[kc * nb];  // Note: using a static buffer is not
thread safe...

    for (j = 0; j < n; j += 4) {  // Loop over the columns of C, unrolled
by 4
        if (first_time)
            PackMatrixB(k, &B(0, j), ldb, &packedB[j * k]);
        for (i = 0; i < m; i += 4) {  // Loop over the rows of C
            if (j == 0)
                PackMatrixA(k, &A(i, 0), lda, &packedA[i * k]);
            AddDot4x4(k, &packedA[i * k], 4, &packedB[j * k], k, &C(i, j),
ldc);
        }
    }
}

/* Function to pack matrix A */
void PackMatrixA(int k, double *a, int lda, double *a_to) {
    for (int j = 0; j < k; j++) {  // Loop over columns of A
        double *a_ij_pntr = &A(0, j);
```

```
        *a_to++ = *a_ij_pntr;
        *(a_to++) = *(a_ij_pntr + 1);
        *(a_to++) = *(a_ij_pntr + 2);
        *(a_to++) = *(a_ij_pntr + 3);
    }
}

/* Function to pack matrix B */
void PackMatrixB(int k, double *b, int ldb, double *b_to) {
    for (int i = 0; i < k; i++) {  // Loop over rows of B
        *b_to++ = *b++;
        *b_to++ = *b++;
        *b_to++ = *b++;
        *b_to++ = *b++;
    }
}

/* Function to perform dot product */
void AddDot4x4(int k, double *a, int lda, double *b, int ldb, double *c,
int ldc) {
    /* ARM NEON registers for computation */
    float64x2_t c_00_c_10_vreg = vdupq_n_f64(0);
    float64x2_t c_01_c_11_vreg = vdupq_n_f64(0);
    float64x2_t c_02_c_12_vreg = vdupq_n_f64(0);
    float64x2_t c_03_c_13_vreg = vdupq_n_f64(0);
    float64x2_t c_20_c_30_vreg = vdupq_n_f64(0);
    float64x2_t c_21_c_31_vreg = vdupq_n_f64(0);
    float64x2_t c_22_c_32_vreg = vdupq_n_f64(0);
    float64x2_t c_23_c_33_vreg = vdupq_n_f64(0);
    float64x2_t a_0p_a_1p_vreg, a_2p_a_3p_vreg;
    float64x2_t b_p0_vreg, b_p1_vreg, b_p2_vreg, b_p3_vreg;

    for (int p = 0; p < k; p++) {
        a_0p_a_1p_vreg = vld1q_f64(a);
        a_2p_a_3p_vreg = vld1q_f64(a + 2);
        a += 4;

        b_p0_vreg = vld1q_dup_f64(b++);
        b_p1_vreg = vld1q_dup_f64(b++);
        b_p2_vreg = vld1q_dup_f64(b++);
        b_p3_vreg = vld1q_dup_f64(b++);

        c_00_c_10_vreg = vfmaq_laneq_f64(c_00_c_10_vreg, a_0p_a_1p_vreg,
b_p0_vreg, 0);
        c_01_c_11_vreg = vfmaq_laneq_f64(c_01_c_11_vreg, a_0p_a_1p_vreg,
b_p1_vreg, 0);
        c_02_c_12_vreg = vfmaq_laneq_f64(c_02_c_12_vreg, a_0p_a_1p_vreg,
b_p2_vreg, 0);
        c_03_c_13_vreg = vfmaq_laneq_f64(c_03_c_13_vreg, a_0p_a_1p_vreg,
b_p3_vreg, 0);
        c_20_c_30_vreg = vfmaq_laneq_f64(c_20_c_30_vreg, a_2p_a_3p_vreg,
b_p0_vreg, 0);
        c_21_c_31_vreg = vfmaq_laneq_f64(c_21_c_31_vreg, a_2p_a_3p_vreg,
b_p1_vreg, 0);
```

```
            c_22_c_32_vreg = vfmaq_laneq_f64(c_22_c_32_vreg, a_2p_a_3p_vreg,
b_p2_vreg, 0);
            c_23_c_33_vreg = vfmaq_laneq_f64(c_23_c_33_vreg, a_2p_a_3p_vreg,
b_p3_vreg, 0);
        }

        /* Store results back to matrix C */
        vst1q_f64(c, c_00_c_10_vreg);
        vst1q_f64(c + ldc, c_01_c_11_vreg);
        vst1q_f64(c + 2 * ldc, c_02_c_12_vreg);
        vst1q_f64(c + 3 * ldc, c_03_c_13_vreg);
        vst1q_f64(c + 4 * ldc, c_20_c_30_vreg);
        vst1q_f64(c + (4 * ldc) + 1, c_21_c_31_vreg);
        vst1q_f64(c + (4 * ldc) + 2, c_22_c_32_vreg);
        vst1q_f64(c + (4 * ldc) + 3, c_23_c_33_vreg);
    }
```

Wynik:



IV Zbiorcze wyniki

V Podsumowanie