MZT

Laboratorium 4

Przemysław Kleszcz

Nr albumu: 124624

Platforma testowa

Procesor	Intel Core i5-7440HQ, 2.8GHz, 6MB Cache
RAM	DDR4 - 16 GB
System operacyjny	Microsoft Windows 10 Pro
Środowisko programistyczne	Visual Studio Professional 2017 v15.5.7
Środowisko uruchomieniowe	.NET Framework 4.6.01055

Tworzenie wątków

Struktura danych zostaje wypełniona i przekazana do metody w celu utworzenia wątku. Po utworzeniu wątków należy zawiesić wątek pierwotny. Zawieszenie wątku odbywa się poprzez wykonanie funkcji **WaitForMultipleObjects**.

```
for (ip = 0; ip < np; ip++)
{
	//wypelnic dane dla potoku ip
	tDat[ip].loc_N = loc_N;
	tDat[ip].N = N;
	tDat[ip].NoThreads = np;
	tDat[ip].ntimes = ntimes;
	tDat[ip].threadNo = ip;
	tDat[ip].X = X;
	tDat[ip].Y = Y;
	tDat[ip].res = 0;

	//stworzyc potok ip
	hThread[ip] = CreateThread(NULL, 0, funTab[imethod], &tDat[ip], 0, NULL);
}

//zawiesic pierwotny potok dokad potoki liczace nie skonca prace
WaitForMultipleObjects(np, hThread, true, INFINITE);
```

Algorytm naiwny

```
DWORD WINAPI ThreadFunc1(LPVOID lpParam)
{
	THREAD_DATA *ptrDat = (THREAD_DATA *)(lpParam);

	register double tmp;
	for (int j = 0; j < ptrDat->ntimes; j++) {
		tmp = 0;
		for (int i = ptrDat->threadNo*ptrDat->loc_N; i < (ptrDat->threadNo +
		1)*ptrDat->loc_N; ++i)
			tmp += ptrDat->X[i] * ptrDat->Y[i];

		ptrDat->res = tmp;
	}

	ptrDat->ret = 0;
	return 0;
}
```

SSE2

```
DWORD WINAPI ThreadFunc2(LPVOID lpParam)
{
	THREAD_DATA *ptrDat = (THREAD_DATA *)(lpParam);
	__m128d sum0, sum1, sum2, sum3;
	__m128d x0, x1, x2, x3;
	__m128d y0, y1, y2, y3;
	_declspec(align(16)) double buf0[2];
	_declspec(align(16)) double buf1[2];
	_declspec(align(16)) double buf2[2];
	_declspec(align(16)) double buf3[2];
	double *ptrX = ptrDat->X;
	double *ptrY = ptrDat->Y;
	int start = ptrDat->threadNo*ptrDat->loc_N;
	int stop = (ptrDat->threadNo + 1)*ptrDat->loc_N;
	int lb = 8;
	int rest = ptrDat->loc_N % lb;
	for (int j = 0; j < ptrDat->ntimes; ++j) {
		sum0 = sum1 = sum2 = sum3 = _mm_setzero_pd();
		int i;
		for (i = start; i < stop - rest; i += lb) {
			//0
			x0 = _mm_loadu_pd(ptrX + i);
			y0 = _mm_loadu_pd(ptrY + i);
			x0 = _mm_mul_pd(x0, y0);
			sum0 = _mm_add_pd(sum0, x0);
			//1
			x1 = _mm_loadu_pd(ptrX + i + 2);
			y1 = _mm_loadu_pd(ptrY + i + 2);
			x1 = _mm_mul_pd(x1, y1);
			sum1 = _mm_add_pd(sum1, x1);
			//2
			x2 = _mm_loadu_pd(ptrX + i + 4);
			y2 = _mm_loadu_pd(ptrY + i + 4);
```

```
                        x2 = _mm_mul_pd(x2, y2);
                        sum2 = _mm_add_pd(sum2, x2);
                        //3
                        x3 = _mm_loadu_pd(ptrX + i + 6);
                        y3 = _mm_loadu_pd(ptrY + i + 6);
                        x3 = _mm_mul_pd(x3, y3);
                        sum3 = _mm_add_pd(sum3, x3);
                }
                _mm_store_pd(buf0, sum0);
                ptrDat->res = buf0[0] + buf0[1];
                _mm_store_pd(buf1, sum1);
                ptrDat->res += buf1[0] + buf1[1];
                _mm_store_pd(buf2, sum2);
                ptrDat->res += buf2[0] + buf2[1];
                _mm_store_pd(buf3, sum3);
                ptrDat->res += buf3[0] + buf3[1];
                for (i = stop - rest; i < stop; ++i) {
                        ptrDat->res += *(ptrX + i) * *(ptrY + i);
                }
        }

        ptrDat->ret = 0;
        return 0;
}
```

AVX

```
DWORD WINAPI ThreadFunc3(LPVOID lpParam)
{
        THREAD_DATA *ptrDat = (THREAD_DATA *)(lpParam);

        __m256d sum0, sum1, sum2, sum3, rx0, rx1, rx2, rx3, ry0, ry1, ry2, ry3;
        double *buf0, *buf1, *buf2, *buf3, *ptr_x, *ptr_y;
        int buf_size = 4;

        int nr = 16;
        int rest = ptrDat->loc_N % nr;

        int it = 0;
        int i = 0;
        int k = 0;

        register double dot = 0;

        try
        {
                buf0 = (double *)_aligned_malloc(buf_size * sizeof(double), 32);
                buf1 = (double *)_aligned_malloc(buf_size * sizeof(double), 32);
                buf2 = (double *)_aligned_malloc(buf_size * sizeof(double), 32);
                buf3 = (double *)_aligned_malloc(buf_size * sizeof(double), 32);
        }
        catch (bad_alloc aa)
        {
                cout << "memory allocation error" << endl;
                system("pause");
                exit(1);
        }
```

```
for (it = 0; it < ptrDat->ntimes; it++) {

	sum0 = _mm256_setzero_pd();
	sum1 = _mm256_setzero_pd();;
	sum2 = _mm256_setzero_pd();;
	sum3 = _mm256_setzero_pd();

	ptr_x = ptrDat->X + (ptrDat->threadNo * ptrDat->loc_N);
	ptr_y = ptrDat->Y + (ptrDat->threadNo * ptrDat->loc_N);
	dot = 0;

	for (i = 0; i < ptrDat->loc_N - rest; i += nr) {

		_mm_prefetch((const char *)(ptr_x + nr), _MM_HINT_T0);
		_mm_prefetch((const char *)(ptr_y + nr), _MM_HINT_T0);

		rx0 = _mm256_load_pd(ptr_x);
		rx1 = _mm256_load_pd(ptr_x + 4);
		rx2 = _mm256_load_pd(ptr_x + 8);
		rx3 = _mm256_load_pd(ptr_x + 12);

		ry0 = _mm256_load_pd(ptr_y);
		ry1 = _mm256_load_pd(ptr_y + 4);
		ry2 = _mm256_load_pd(ptr_y + 8);
		ry3 = _mm256_load_pd(ptr_y + 12);

		rx0 = _mm256_mul_pd(rx0, ry0);
		rx1 = _mm256_mul_pd(rx1, ry1);
		rx2 = _mm256_mul_pd(rx2, ry2);
		rx3 = _mm256_mul_pd(rx3, ry3);

		sum0 = _mm256_add_pd(sum0, rx0);
		sum1 = _mm256_add_pd(sum1, rx1);
		sum2 = _mm256_add_pd(sum2, rx2);
		sum3 = _mm256_add_pd(sum3, rx3);

		ptr_x += nr;
		ptr_y += nr;
	}

	_mm256_store_pd(buf0, sum0);
	_mm256_store_pd(buf1, sum1);
	_mm256_store_pd(buf2, sum2);
	_mm256_store_pd(buf3, sum3);

	dot += buf0[0] + buf0[1] + buf0[2] + buf0[3]
		+ buf1[0] + buf1[1] + buf1[2] + buf1[3]
		+ buf2[0] + buf2[1] + buf2[2] + buf2[3]
		+ buf3[0] + buf3[1] + buf3[2] + buf3[3];

	for (i = 0; i < rest; i++) {
		dot += ptr_x[i] * ptr_y[i];
	}
}

_aligned_free(buf0);
_aligned_free(buf1);
```

```
	_aligned_free(buf2);
	_aligned_free(buf3);

	ptrDat->res = dot;
	ptrDat->ret = 0;
	return 0;

	ptrDat->ret = 0;
	return 0;
}
```

Algorytm do poprawy

Algorytm wykorzystuje strukturę THREAD_DATA do zapisywanie rezultatów w związku z tym jest narażony na konflikty pamięci podręcznej. Poprawić tą sytuację można poprzez zastosowanie cache linii – aby różne procesory miały dostęp do różnych adresów pamięci.

```
DWORD WINAPI ThreadFunc4(LPVOID lpParam)
{
	THREAD_DATA *ptrDat = (THREAD_DATA *)(lpParam);

	for (int j = 0; j < ptrDat->ntimes; ++j)
	{
		ptrDat->res = 0;
		for (int i = ptrDat->threadNo*ptrDat->loc_N; i < (ptrDat->threadNo +
		1)*ptrDat->loc_N; ++i)
			ptrDat->res += ptrDat->X[i] * ptrDat->Y[i];
	}

	ptrDat->ret = 0;
	return 0;
}

struct THREAD_DATA
{
	int threadNo;	//thread number
	int NoThreads;	//number of threads
	int N;		//dimension of vector
	int loc_N;	//dimension of subvector
	int ntimes;	//number of repititions
	int ret;	//return value from thread
	double *X;	//vector X - read only
	double *Y;	//vector Y - read only
	double res;	//dot product for subvectors
	char cacheLine[128];
};
```

Aby odnaleźć czas wykonania zadań przed zamieszczeniem cache linii należało jedynie zakomentować odpowiedni fragment:

```
//char cacheLine[128];
```

Rezultaty

Zadanie 1 – N = 10000, ntimes = 1000000

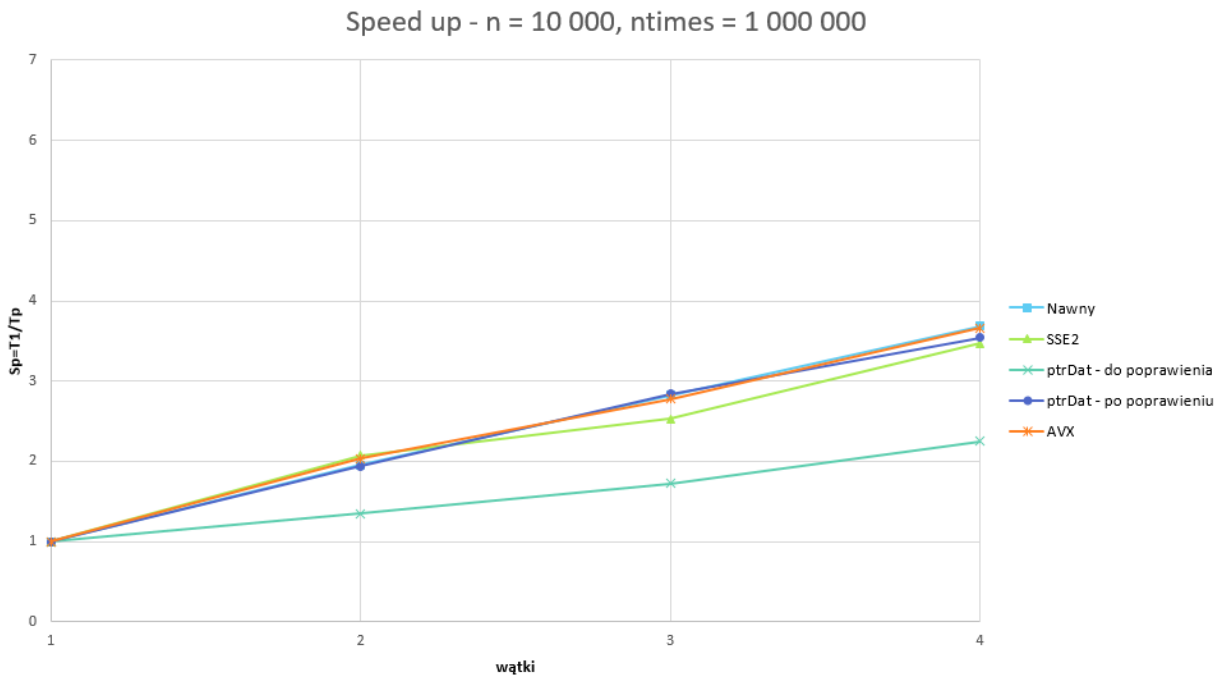
Ilość wątków	Naive: rrr += X_i[i]*Y_i[i]		SSE2		ptrDat->res += X_i[i]*Y_i[i] do poprawienia		ptrDat->res += X_i[i]*Y_i[i] po poprawieniu		AVX	
	Czas oblicz., ms	Sp = T1/Tp	Czas oblicz., ms	Sp = T1/Tp	Czas oblicz., ms	Sp = T1/Tp	Czas oblicz., ms	Sp = T1/Tp	Czas oblicz., ms	Sp = T1/Tp
1	11282	1	3203	1	28078	1	28188	1	2515	1
2	5781	1.95157	1547	2.07046	20781	1.35114	14547	1.93772	1234	2.0381
3	4000	2.8205	1266	2.53002	16282	1.72448	9922	2.84096	906	2.7759
4	3062	3.68452	922	3.47397	12500	2.24624	7954	3.54388	687	3.6608
5	3125	3.61024	547	5.85558	10812	2.59693	7547	3.73499	469	5.3625
6	3078	3.66537	563	5.68917	9579	2.9312	7407	3.80559	421	5.9739

Zadanie 2 – N = 10000000, ntimes = 1000

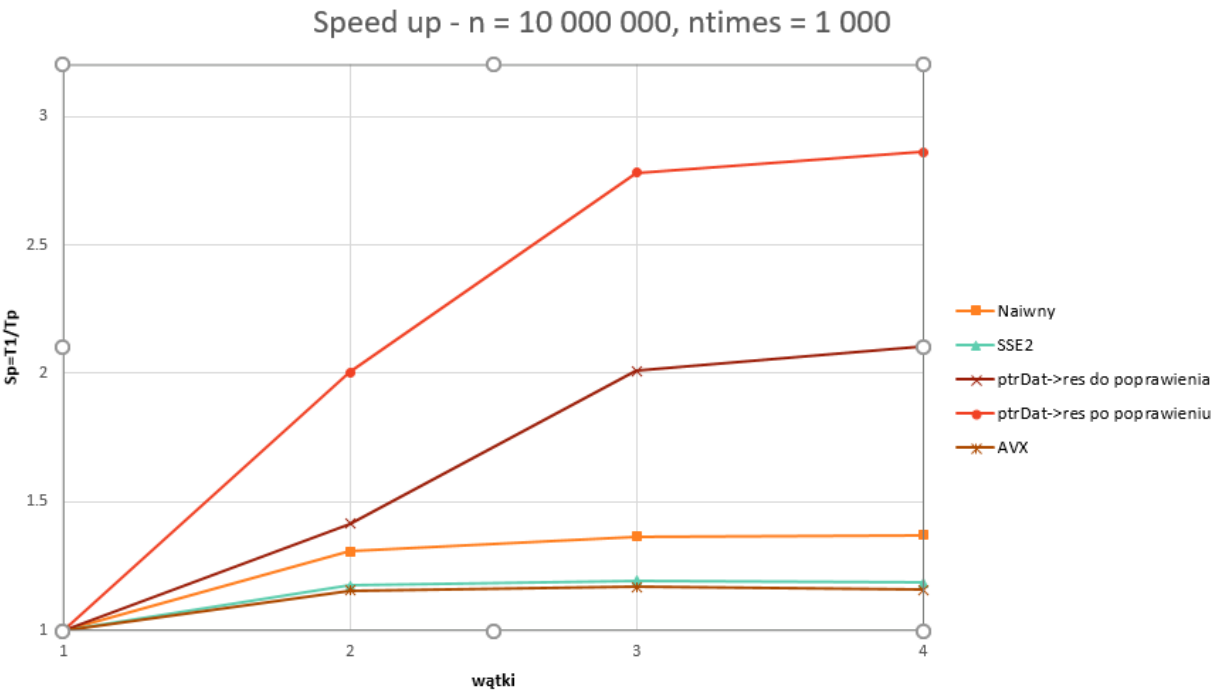
Ilość wątków	Naive: rrr += X_i[i]*Y_i[i]		SSE2		ptrDat->res += X_i[i]*Y_i[i] do poprawienia		ptrDat->res += X_i[i]*Y_i[i] po poprawieniu		AVX	
	Czas oblicz., ms	Sp = T1/Tp	Czas oblicz., ms	Sp = T1/Tp	Czas oblicz., ms	Sp = T1/Tp	Czas oblicz., ms	Sp = T1/Tp	Czas oblicz., ms	Sp = T1/Tp
1	13938	1	11938	1	32750	1	33704	1	11703	1
2	10656	1.308	10172	1.17361	23156	1.41432	16828	2.00285	10141	1.15403
3	10203	1.36607	10000	1.1938	16297	2.00957	12125	2.77971	10000	1.1703
4	10172	1.37023	10078	1.18456	15578	2.10232	11782	2.86063	10109	1.15768
5	10516	1.32541	10672	1.11863	11953	2.7399	10969	3.07266	10312	1.13489
6	11296	1.23389	10703	1.11539	11813	2.77237	11688	2.88364	10985	1.06536

Wykresy

Zadanie 1 – N = 10000, ntimes = 1000000



Zadanie 2 – N = 10000000, ntimes = 1000



Wnioski

Jak widać zadanie drugie nie daje się wydajnie przyśpieszyć. Dzieje się tak ponieważ rozmiar danych na ktorych operujemy jest na tyle duży, że nie mieści się w pamięci podręcznej, przez co dostęp do nich trwa dłużej.

Zmienna lokalna zapewnia możliwość bezpiecznego zapisu wyników operacji bez narażenia na konflikty pamięci podręcznej.

Aby zapewnić stabilne przyspieszenie na dowolnym sprzęcie wykorzystując elementy struktury – jednym z rozwiązań jest stosowanie cache linii.