**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**

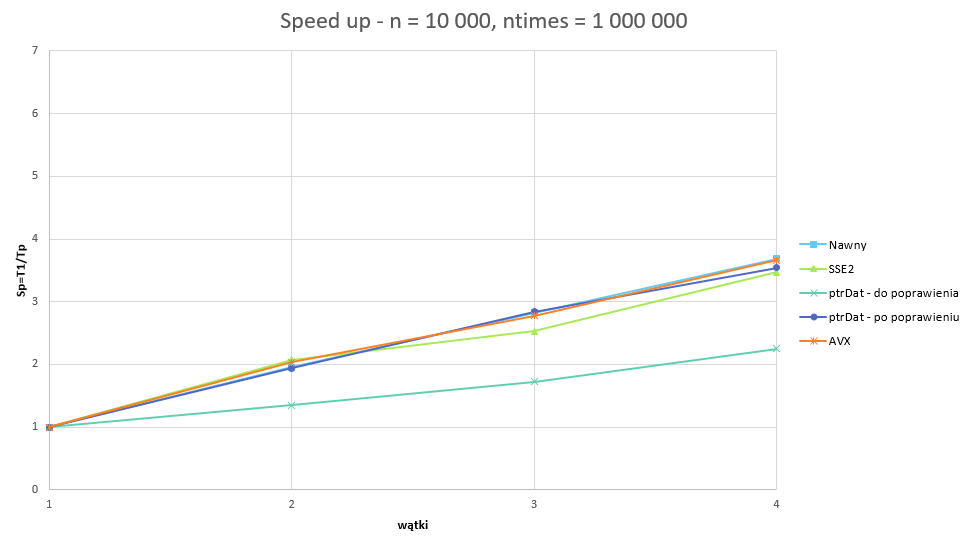
|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Ilość wątków | Naive: | | SSE2 | | ptrDat->res += X\_i[i]\*Y\_i[i] | | ptrDat->res += X\_i[i]\*Y\_i[i] | | AVX | |
| rrr += X\_i[i]\*Y\_i[i] | | do poprawienia | | po poprawieniu | |
| Czas oblicz., 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**

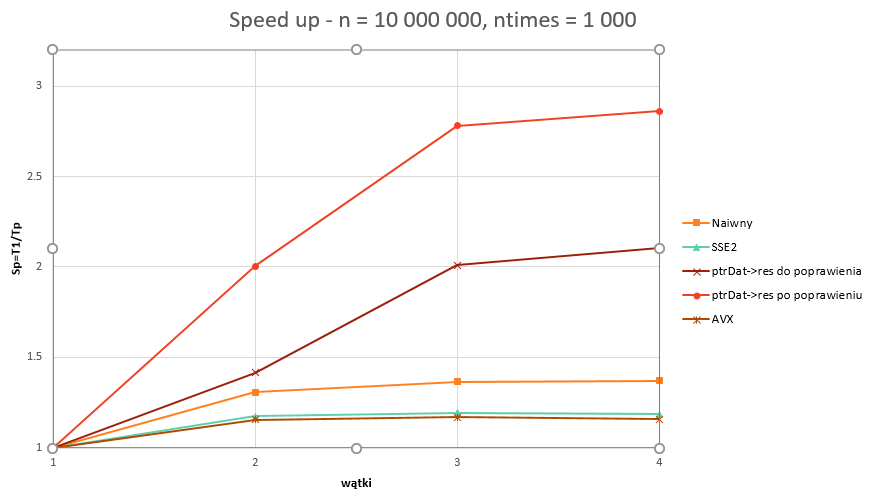
|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Ilość wątków | Naive: | | SSE2 | | ptrDat->res += X\_i[i]\*Y\_i[i] | | ptrDat->res += X\_i[i]\*Y\_i[i] | | AVX | |
| rrr += X\_i[i]\*Y\_i[i] | | do poprawienia | | po poprawieniu | |
| Czas oblicz., 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.