МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

**НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №4

по курсу «Архитектура современных микропроцессоров и мультипроцессоров»

**ОПРЕДЕЛЕНИЕ МАКСИМАЛЬНО ДОСТИЖИМЫХ СКОРОСТЕЙ ЧТЕНИЯ, ЗАПИСИ И КОПИРОВАНИЯ ДАННЫХ В RAM**

**Выполнил:** студент 3-го курса гр. 17208

Гафиятуллин А.Р

Новосибирск, 2020

1. **ЦЕЛИ РАБОТЫ:**
   1. научиться определять максимально достижимые скорости чтения, записи и копирования данных в оперативной памяти.
2. **ЗАДАЧИ РАБОТЫ:**
   1. Написать подпрограммы, наиболее быстро выполняющие операции чтения, записи и копирования данных одним потоком.
   2. Написать программу, определяющую максимально возможную скорость чтения, записи и копирования данных при одновременном выполнении операции заданным числом потоков.
   3. Определить максимальную скорость чтения, записи, копирования данных (в GB/s) для двух различных многоядерных вычислительных систем для числа потоков от 1 до числа ядер. Сравнить полученные результаты с техническими характеристиками вычислительных систем. По результатам исследования сделать вывод.
3. **ТЕСТИРУЕМЫЕ СИТСЕМЫ:**
   1. Персональный компьютер:
      * **Процессор**: Intel® Core™ i7-9700F:
        + **Кол-во ядер:** 8;
        + **Поддерживаемый тип памяти:** DDR4-2666;
        + **Максимальная пропускная способность памяти:** 42.6 GB/s;
        + **Количество каналов памяти:** 2.
      * **Память**:
        + **Объем:** 32 Гб;
        + **Эффективная частота:** 2666 МГц;
        + **Режим работы:** двухканальный.
      * **Компилятор**: Intel(R) C++ Intel(R) 64 Compiler for applications running on Intel(R) 64, Version 19.1.1.216 Build 20200306;
      * **Ключи компиляции (наиболее важные**): /O2 /arch:AVX2 /tune:coffeelake /Qopenmp;
   2. Блейд-сервер HP BL2x220c G7:
      * **Процессор:** 2 x Intel Xeon X5670:
        + **Кол-во ядер:** 12;
        + **Поддерживаемый тип памяти**: DDR3 800/1066/1333;
        + **Максимальная пропускная способность памяти:** 64 GB/s;
        + **Количество каналов памяти:** 6.
      * **Память:**
        + **Объем:** 24 Гб;
        + **Эффективная частота:** 1333 МГц;
        + **Режим работы:** шестиканальный.
      * **Компилятор**: gcc version 4.8.5
      * **Ключи компиляции:** -std=c++11 -msse4.1 -fopenmp -O2
   3. Размеры массивов для тестирования: 512 Мбайт.
4. **ХОД РАБОТЫ:**
   1. Написание подпрограмм:
      * **Тип данных:** \_\_m128i;
      * **Использование операций чтения/записи без доступа в кэш:** да;
      * **Раскрутка циклов:** нет (не дало заметного эффекта);
      * **Использование библиотечных функций memchr, memset, memcpy**: нет в местах замера времени.
      * Подпрограммы:
        + Запись:

double test\_write(\_\_m128i \*buffer, \_\_m128i init\_element) {

    double start = omp\_get\_wtime();

    for (size\_t i = 0; i < BUFFER\_SIZE / sizeof(\_\_m128i); i++) {

        \_mm\_stream\_si128(buffer + i, init\_element);     // non-temporal write

    }

    double end = omp\_get\_wtime();

    return end - start;

}

* + - * Чтение:

double test\_read(\_\_m128i\* buffer, int \*res) {

    \_\_m128i read\_element;

    double start = omp\_get\_wtime();

    for (size\_t i = 0; i < BUFFER\_SIZE / sizeof(\_\_m128i); i++) {

        read\_element = \_mm\_stream\_load\_si128(buffer + i); // non-temporal read

    }

    double end = omp\_get\_wtime();

    // assign to prevent optimization

    \*res += ((int\*)&read\_element)[2];

    return (end - start);

}

* + - * Копирование:

double test\_copy(\_\_m128i\* buffer1, \_\_m128i\* buffer2) {

    double start = omp\_get\_wtime();

    for (size\_t i = 0; i < BUFFER\_SIZE / sizeof(\_\_m128i); i++) {

\_mm\_stream\_si128(buffer2 + i, \_mm\_stream\_load\_si128(buffer1 + i));     // non-temporal read-write

    }

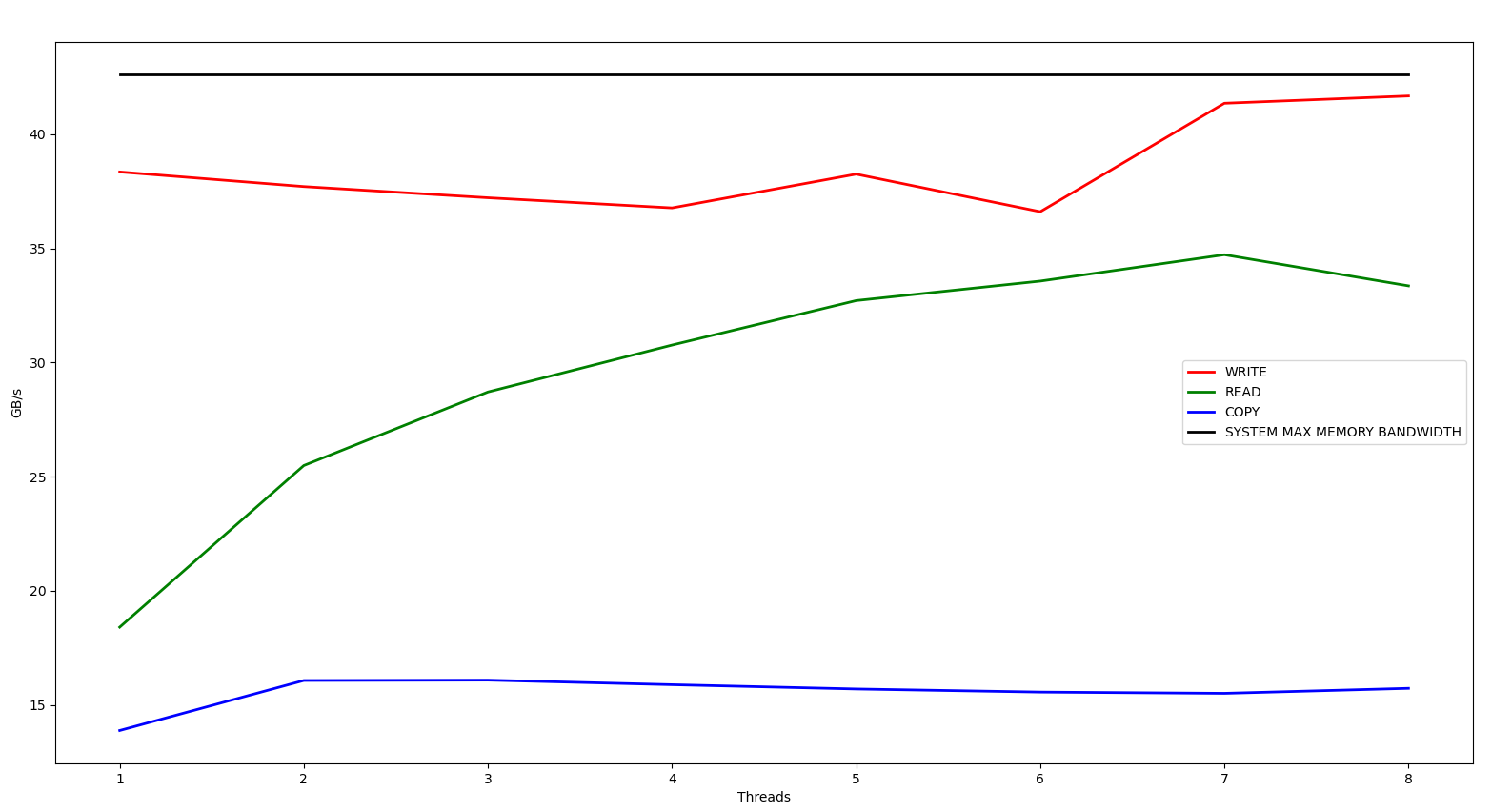
    double end = omp\_get\_wtime();

    return (end - start);

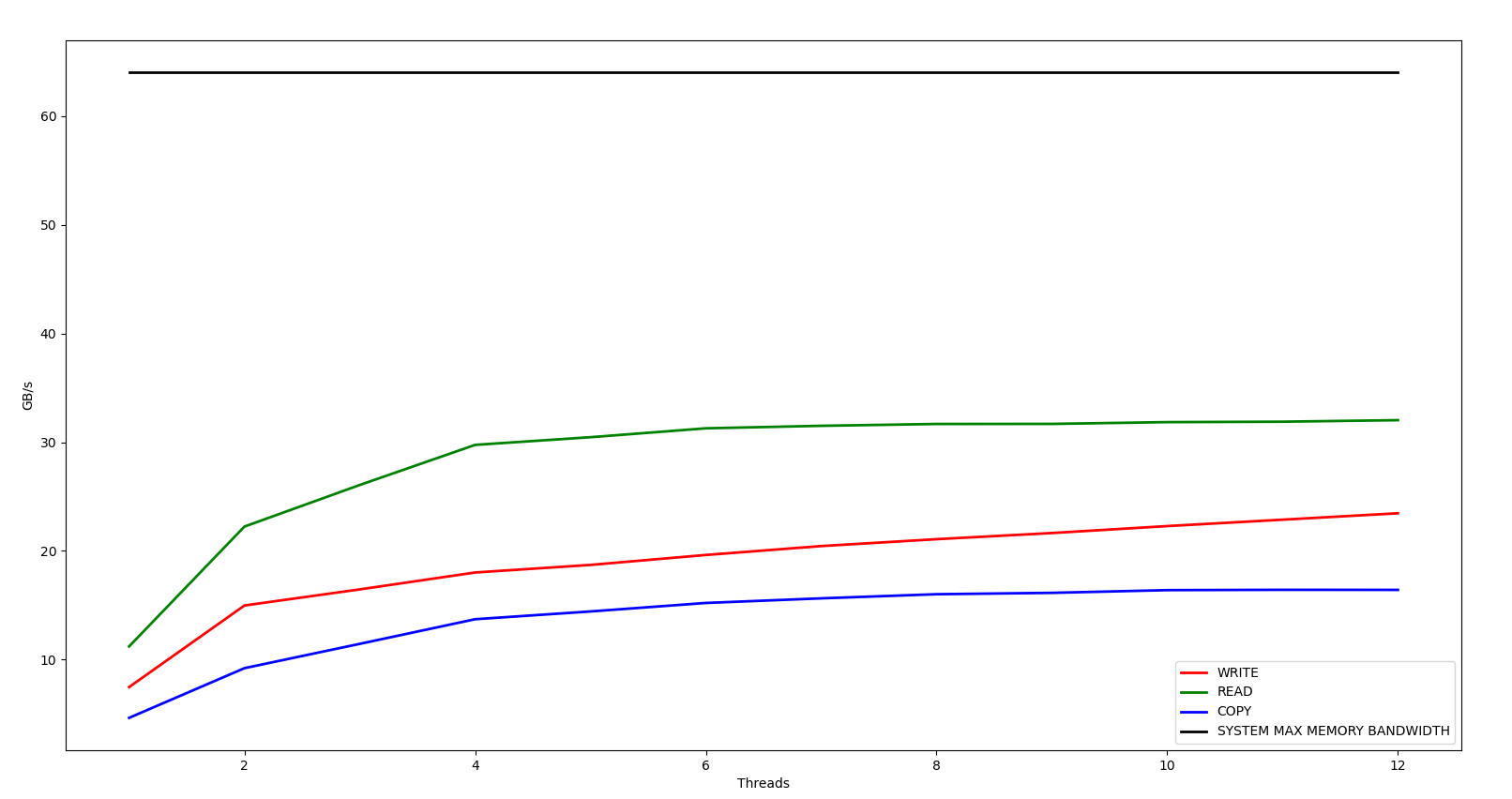
}

**ВАЖНО:** перед началом замера времени необходимо пройтись по массиву, например с помощью **memset.** Это связано с тем, что в современных ОС, реализующих механизм виртуальной памяти, память выделяется **постранично во время очень долгого аппаратного прерывания (PAGE FAULT).** Как оказалось, это очень сильно влияет на результаты.

* 1. Листинг многопоточной программы (см. Приложение 6.1-6.3).
  2. Графики с результатами тестирования:
     + Персональный компьютер:



* + - HP BL2x220c G7:



**Черная линия:** максимальная пропускная способность памяти;

**Красная линия:** запись;

**Зеленая линия:** чтение;

**Синяя линия:** копирование.

Так как сервер обычно довольно сильно загружен, то имеет смысл поближе рассмотреть результаты, полученные на относительно мощном персональном компьютере сразу после загрузки системы, на которой доступны удобные средства профилирования.

**Что можно заметить на этом графике:**

1. Самая быстрая операция – запись, причем скорость работы с памятью вплотную приближается к теоретической максимальной пропускной способности. К тому же она практически максимальная уже на 1 потоке;
2. Скорость работы с памятью при чтении растет постепенно с ростом кол-ва ядер;
3. Копирование самая медленная операция.

Чтобы объяснить эти результаты, было проведено профилирование кода на 1 и 8 потоках с помощью **Intel VTune Profiler 2020**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Операция | Кол-во запросов к LLC на 1 потоке и % промахов | | Кол-во запросов к LLC на 8 потоках и % промахов | |
| READ | 520344028 | 46% | 2491887058 | 90% |
| WRITE | 29029812 | 23% | 217047184 | 18% |
| COPY | 658109478 | 44% | 5020689024 | 89% |

**Какие выводы можно сделать из профилирования:**

1. Кол-во запросов к LLC при операции чтения по крайней мере в **10**(на одном потоке вообще в **17**) раз больше, чем при операции записи и это при том, что используется специальный интринсик **\_mm\_stream\_load\_si128** при чтениии, казалось бы, эти показатели не должны отличаться;
2. При операции записи не только практически не происходит обращений к LLC (как и должно быть из-за интринсика **\_mm\_stream\_si128**), так они еще и достаточно точные (если происходят);
3. Показатели обращений к LLC при копировании и чтении сравнимы, хотя у копирования они ожидаемо больше.

**Возникает закономерный вопрос:** почему **\_mm\_stream\_load\_si128** не работает в нашем случае и обращения к кэшу продолжаются?

Данная операция является оберткой над инструкцией **MOVNTDQA.** Intel специфицирует использование этой инструкции для регионов памяти, помеченных как **USWC** (uncachable speculative write combining). Обычная память такого типа не имеет. Для всех остальных регионов памяти вместо этой инструкции используется обычная **MOVDQA** даже при вызове **\_mm\_stream\_load\_si128,** причем, в таком случае, задействуются обычные (или почти обычные) политики кэширования данных. Учитывая большие размеры тестовых данных и отсутствие повторного их использования, очень много тактов тратится на проверки кэшей, что сильно замедляет работу с памятью.

**Отсюда вытекают следующие выводы:**

1. Для записи почти максимальная пропускная способность может быть достигнута уже на одном потоке, потому что нет трат времени на ожидания ответа от кэшей;
2. При чтении нет возможности достигнуть максимальную пропускную способность сразу из-за долгого ожидания ответа от кэшей, который еще и сообщит о промахе, и все равно придется сходить в память. Большая часть пропускной способности шины остается незанятой, поэтому увеличение кол-ва потоков позволяет заполнить это незанятое пространство, так как на реальных ядрах они могут выполняться действительно параллельно;
3. Скорости копирования такие небольшие, потому что там есть медленное чтение. Так же вносит свои коррективы работа с двумя массивами и запись.

Для сравнения результаты встроенного в **AIDA64 Extreme** бенчмарка памяти. На сайте разработчика указано, что этот бенчмарк использует хорошо оптимизированные архитектурно-специфичные инструкции:

**Чтение**:



(**33.663 Гб/с** – примерно такой же результат получен и в написанной программе).

**Запись:**

**(36.859 Гб/с -** примерно такой же результат получен и в написанной программе, причем скорость записи так же выше скорости чтения).

**Копирование:**

****

**(31.789 Гб/с –** так же самая низкая, но в два раза выше той, которая была получена в написанной программе. Видимо, действительно очень оптимизированные бенчмарки).

**Результаты, полученные на сервере:**

1. В отличие от персонального компьютера, здесь самая быстрая операция — это чтение (интересно, почему? Возможно, из-за нагрузки на сервер и т. д.);
2. Чтение в два раза медленнее максимальной пропускной способности;
3. Копирование осталось самой медленной операцией.

На кластере также был запущен бенчмарк **STREAM.** Из интересующих операций он замерил скорость копирования, которая составила **15.617 Гб/с**, что совпало с результатами, полученными написанной программой.

1. **ВЫВОДЫ:**
   1. Научились определять максимально достижимые скорости чтения, записи и копирования данных в оперативной памяти;
   2. Нужно быть осторожным при работе с виртуальной памятью, она может испортить производительность;
   3. Некоторые интринсики имеют довольно узкие границы применимости.
2. **ПРИЛОЖЕНИЕ:**
   1. **Чтение:**

#include <iostream>

#include <chrono>

#include <emmintrin.h>

#include <smmintrin.h>

#include <omp.h>

#include <vector>

#include <algorithm>

#include <cstring>

#include <cstdlib>

#ifdef \_WIN32

#include <windows.h>    // for SetThreadAffinityMask

#elif \_\_linux\_\_

#include <sched.h>  // for sched\_setaffinity

#endif

constexpr int ALIGN = 32;

constexpr size\_t MBYTES = 512; // test array volume

constexpr size\_t BUFFER\_SIZE = 1024 \* 1024 \* MBYTES;

constexpr int TEST\_TRIES = 50;

double test\_read(\_\_m128i\* buffer, int \*res) {

    \_\_m128i read\_element;

    double start = omp\_get\_wtime();

    for (size\_t i = 0; i < BUFFER\_SIZE / sizeof(\_\_m128i); i++) {

        read\_element = \_mm\_stream\_load\_si128(buffer + i); // non-temporal read

    }

    double end = omp\_get\_wtime();

    // assign to prevent optimization

    \*res += ((int\*)&read\_element)[2];

    return (end - start);

}

void set\_thread\_affinity(int max\_threads) {

#pragma omp parallel num\_threads(max\_threads)

    {

#ifdef \_WIN32

    DWORD\_PTR mask = (1 << omp\_get\_thread\_num());

    SetThreadAffinityMask(GetCurrentThread(), mask);

#elif \_\_linux\_\_

    int cpu = omp\_get\_thread\_num();

    cpu\_set\_t cpuset;

    CPU\_ZERO(&cpuset);

    CPU\_SET(cpu , &cpuset);

    sched\_setaffinity(0, sizeof(cpuset), &cpuset);

#endif

    }

}

int main(int argc, char \*argv[]) {

    int cores\_num = 1;

    if(argc >= 2) {

        cores\_num = atoi(argv[1]);

    }

    set\_thread\_affinity(cores\_num);

    double curr\_speed = 0;

    double result = 0;

    int res = 0;

#pragma omp parallel num\_threads(cores\_num)

        {

            for (int j = 0; j < TEST\_TRIES; j++) {

#pragma omp single

                curr\_speed = 0;

                \_\_m128i\* buffer = (\_\_m128i\*)\_mm\_malloc(BUFFER\_SIZE, ALIGN);

                memset(buffer, 0, BUFFER\_SIZE);

#pragma omp barrier

                test\_read(buffer, &res);

                double elapsed = test\_read(buffer, &res);

                test\_read(buffer, &res);

#pragma omp critical

                {

                    curr\_speed += MBYTES / 1024.0 / elapsed;

                }

#pragma omp barrier

#pragma omp single

                {

                    result = std::max<double>(curr\_speed, result);

                }

                \_mm\_free(buffer);

            }

        }

    std::cout << result;

    return 0;

}

* 1. **Запись:**

#include <iostream>

#include <chrono>

#include <emmintrin.h>

#include <smmintrin.h>

#include <omp.h>

#include <vector>

#include <algorithm>

#include <cstring>

#include <cstdlib>

#ifdef \_WIN32

#include <windows.h>    // for SetThreadAffinityMask

#elif \_\_linux\_\_

#include <sched.h>  // for sched\_setaffinity

#endif

constexpr int ALIGN = 32;

constexpr size\_t MBYTES = 512; // test array volume

constexpr size\_t BUFFER\_SIZE = 1024 \* 1024 \* MBYTES;

constexpr int TEST\_TRIES = 50;

double test\_write(\_\_m128i \*buffer, \_\_m128i init\_element) {

    double start = omp\_get\_wtime();

    for (size\_t i = 0; i < BUFFER\_SIZE / sizeof(\_\_m128i); i++) {

        \_mm\_stream\_si128(buffer + i, init\_element);     // non-temporal write

    }

    double end = omp\_get\_wtime();

    return end - start;

}

void set\_thread\_affinity(int max\_threads) {

#pragma omp parallel num\_threads(max\_threads)

    {

#ifdef \_WIN32

    DWORD\_PTR mask = (1 << omp\_get\_thread\_num());

    SetThreadAffinityMask(GetCurrentThread(), mask);

#elif \_\_linux\_\_

    int cpu = omp\_get\_thread\_num();

    cpu\_set\_t cpuset;

    CPU\_ZERO(&cpuset);

    CPU\_SET(cpu , &cpuset);

    sched\_setaffinity(0, sizeof(cpuset), &cpuset);

#endif

    }

}

int main(int argc, char \*argv[]) {

    int cores\_num = 1;

    if(argc >= 2) {

        cores\_num = atoi(argv[1]);

    }

    set\_thread\_affinity(cores\_num);

    double curr\_speed = 0;

    double result = 0;

#pragma omp parallel num\_threads(cores\_num)

        {

            for (int j = 0; j < TEST\_TRIES; j++) {

#pragma omp single

                curr\_speed = 0;

                \_\_m128i\* buffer = (\_\_m128i\*)\_mm\_malloc(BUFFER\_SIZE, ALIGN);

                \_\_m128i init\_element = \_mm\_set\_epi32(0, 1, 2, 3);

                memset(buffer, 0, BUFFER\_SIZE);    // touch array to allocate

#pragma omp barrier

                test\_write(buffer, init\_element);

                double elapsed = test\_write(buffer, init\_element);

                test\_write(buffer, init\_element);

#pragma omp critical

                {

                    curr\_speed += MBYTES / 1024.0 / elapsed;

                }

#pragma omp barrier

#pragma omp single

                {

                    result = std::max<double>(curr\_speed, result);

                }

                \_mm\_free(buffer);

            }

        }

    std::cout << result;

    return 0;

}

* 1. **Копирование:**

#include <iostream>

#include <chrono>

#include <emmintrin.h>

#include <smmintrin.h>

#include <omp.h>

#include <vector>

#include <algorithm>

#include <cstring>

#include <cstdlib>

#ifdef \_WIN32

#include <windows.h>    // for SetThreadAffinityMask

#elif \_\_linux\_\_

#include <sched.h>  // for sched\_setaffinity

#endif

constexpr int ALIGN = 32;

constexpr size\_t MBYTES = 512; // test array volume

constexpr size\_t BUFFER\_SIZE = 1024 \* 1024 \* MBYTES;

constexpr int TEST\_TRIES = 50;

double test\_copy(\_\_m128i\* buffer1, \_\_m128i\* buffer2) {

    double start = omp\_get\_wtime();

    for (size\_t i = 0; i < BUFFER\_SIZE / sizeof(\_\_m128i); i++) {

        \_mm\_stream\_si128(buffer2 + i, \_mm\_stream\_load\_si128(buffer1 + i));     // non-temporal read-write

    }

    double end = omp\_get\_wtime();

    return (end - start);

}

void set\_thread\_affinity(int max\_threads) {

#pragma omp parallel num\_threads(max\_threads)

    {

#ifdef \_WIN32

    DWORD\_PTR mask = (1 << omp\_get\_thread\_num());

    SetThreadAffinityMask(GetCurrentThread(), mask);

#elif \_\_linux\_\_

    int cpu = omp\_get\_thread\_num();

    cpu\_set\_t cpuset;

    CPU\_ZERO(&cpuset);

    CPU\_SET(cpu , &cpuset);

    sched\_setaffinity(0, sizeof(cpuset), &cpuset);

#endif

    }

}

int main(int argc, char \*argv[]) {

    int cores\_num = 1;

    if(argc >= 2) {

        cores\_num = atoi(argv[1]);

    }

    set\_thread\_affinity(cores\_num);

    double curr\_speed = 0;

    double result = 0;

#pragma omp parallel num\_threads(cores\_num)

        {

            for (int j = 0; j < TEST\_TRIES; j++) {

#pragma omp single

                curr\_speed = 0;

                \_\_m128i\* buffer1 = (\_\_m128i\*)\_mm\_malloc(BUFFER\_SIZE, ALIGN);

                \_\_m128i\* buffer2 = (\_\_m128i\*)\_mm\_malloc(BUFFER\_SIZE, ALIGN);

                memset(buffer1, 0, BUFFER\_SIZE);

                memset(buffer2, 0, BUFFER\_SIZE);

#pragma omp barrier

                test\_copy(buffer1, buffer2);

                double elapsed = test\_copy(buffer2, buffer1);

                test\_copy(buffer1, buffer2);

#pragma omp critical

                {

                    curr\_speed += MBYTES / 1024.0 / elapsed;

                }

#pragma omp barrier

#pragma omp single

                {

                    result = std::max<double>(curr\_speed, result);

                }

                \_mm\_free(buffer1);

                \_mm\_free(buffer2);

            }

        }

    std::cout << result;

    return 0;

}