ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

КАФЕДРА ВТ

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №3

«Оценка характеристик персонального компьютера» по дисциплине «Архитектура вычислительных систем»

Выполнил: студент гр. АММ2-24 Ириков Евгений Алексеевич

Проверил: к.т.н., доцент Кафедры ВТ Перышкова Евгения Николаевна

Новосибирск 2024

Содержание

[Выполнение работы 6](#_Toc55302170)

[Запуск программы 6](#_Toc55302171)

[Результат работы 7](#_Toc55302172)

[Приложение 11](#_Toc55302177)

**Постановка задачи**

*Разработать программу (benchmark) для оценки производительности подсистемы памяти.*

*1. Написать программум(функцию) на языке С/С++/C# для оценки производительности подсистемы памяти.*

*На вход программы подать следующие аргументы.*

*1) Подсистема памяти. Предусмотреть возможность указать подсистему для проверки*

*производительности: RAM (оперативная память), HDD/SSD и flash.*

*2) Число испытаний, т.е. число раз повторений измерений.*

*Пример вызова программы: ./memory\_test –m RAM –b 1024|1Kb –l 10 или*

*./memory\_bandwidth ––memory-type RAM|HDD|SSD|flash*

*––block-size 1024|1Kb*

*––launch-count 10*

*В качестве блока данных использовать одномерный массив, в котором произведение числа элементов*

*на их размерность равна требуемому размеру блока данных. Массив инициализировать случайными*

*значениями. Для тестирования HDD/SSD и flash создать в программе файлы в соответствующих*

*директориях.*

*Измерение времени реализовать с помощью функции clock\_gettime() или аналогичной с точность до*

*наносекунд. Измерять время исключительно на запись элемента в память или считывание из неё, без*

*операций генерации или преобразования данных.*

*На выходе программы в одну строку CSV файла со следующей структурой:*

*[MemoryType;BlockSize;ElementType;BufferSize;LaunchNum;Timer;WriteTime;AverageWriteTime;WriteBandwidth;*

*AbsError(write);RelError(write);ReadTime;AverageReadTime;ReadBandwidthAbsError(read);RelError(read);], где*

*MemoryType – тип памяти (RAM|HDD|SSD|flash) или модель устройства, на котором проводятся испытания;*

*BlockSize – размер блока данных для записи и чтения на каждом испытании;*

*ElementType – тип элементов используемых для заполнения массива данных;*

*BufferSize – размер буфера, т.е. порции данных для выполнения одно операции записи или чтения;*

*LaunchNum – порядковый номер испытания;*

*Timer – название функции обращения к таймеру (для измерения времени);*

*WriteTime – время выполнения отдельного испытания с номером LaunchNum [секунды];*

*AverageWriteTime – среднее время записи из LaunchNum испытаний [секунды];*

*WriteBandwidth – пропускная способность памяти (BLOCK\_SIZE/AverageWriteTime) \* 10 6 [Mb/s]*

*AbsError(write) – абсолютная погрешность измерения времени записи или СКО [секунды];*

*RelError(write) – относительная погрешность измерения времени [%];*

*ReadTime – время выполнения отдельного испытания LaunchNum [секунды];*

*AverageReadTime – среднее время записи из LaunchNum испытаний [секунды];*

*ReadBandwidth – пропускная способность памяти (BLOCK\_SIZE/AverageReadTime) \* 10 6 [Mб/сек.]*

*AbsError(read) – абсолютная погрешность измерения времени чтения или СКО [секунды];*

*RelError(read) – относительная погрешность измерения времени [%].*

*2. Написать программу(функцию) на языке С/С++/C# или скрипт (benchmark) реализующий серию*

*испытаний программы(функции) из п.1. Оценить пропускную способность оперативной памяти при*

*работе с блоками данных равными объёму кэш-линии, кэш-памяти L1, L2 и L3 уровня и превышающего*

*его. Для HDD|SSD и flash провести серию из 20 испытаний с блоками данных начиная с 4 Мб с шагом*

*4Мб. Результаты всех испытаний сохранить в один CSV файл со структурой, описанной в п.1.*

*\* Для HDD|SSD и flash оценить влияние размера буфера (BufferSize) на пропускную способность памяти.*

*3. На основе CSV файла построить сводные таблицы и диаграммы отражающие:*

*1) Зависимость пропускной способности записи и чтения от размера блока данных ( BlockSize) для*

*разного типа памяти;*

*2) Зависимость погрешности измерения пропускной способности от размера блока данных для*

*разного типа памяти;*

*3) Зависимость погрешности измерений от числа испытаний LaunchNum;*

*4) \* Зависимость пропускной способности памяти от размера буфера для HDD|SSD и flash памяти;*

# Выполнение работы

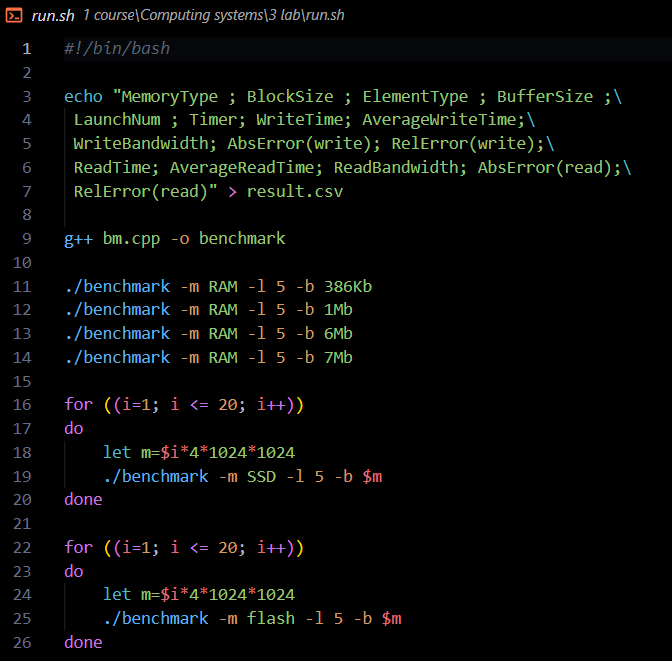
Перед оценкой производительности подсистем памяти нужно определить размера кеш-линиии. Для нашей системы это:

L1 - 386 Kb

L2 - 1 Mb

L3 6 Mb

Запуск программы



Bash-скрипт

Где параметр после названия исполняемого файла — это вид тестируемой памяти, допустимые значения RAM, HDD, SSD, flash.

Для обозначения размера блока памяти доступны наименования K – килобайты, М – мегабайты, ничего - байты

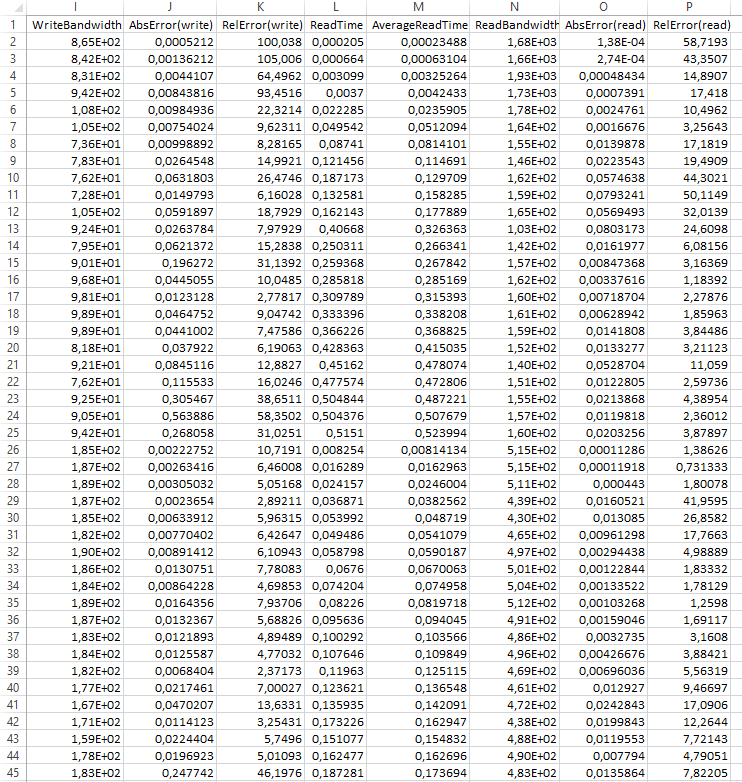
После блока памяти можно указать количество испытаний.

# Результат работы

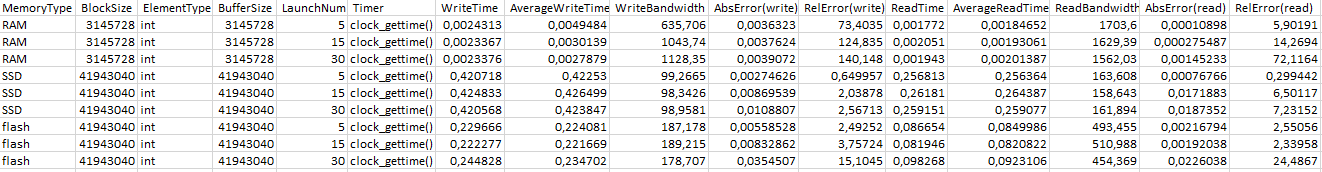
**Запуск программы**

****

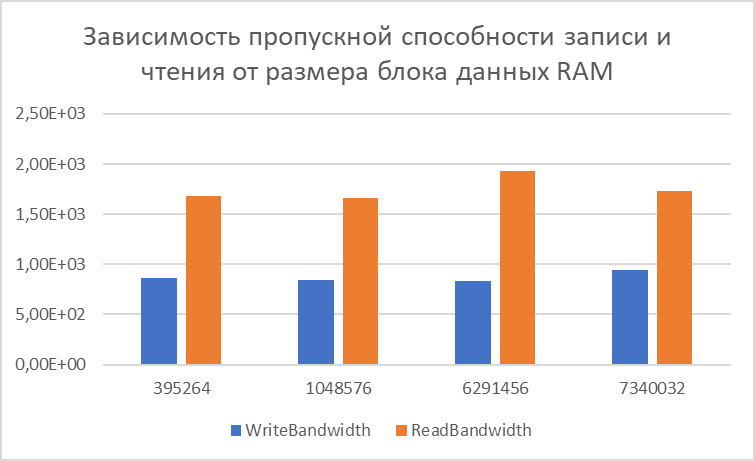
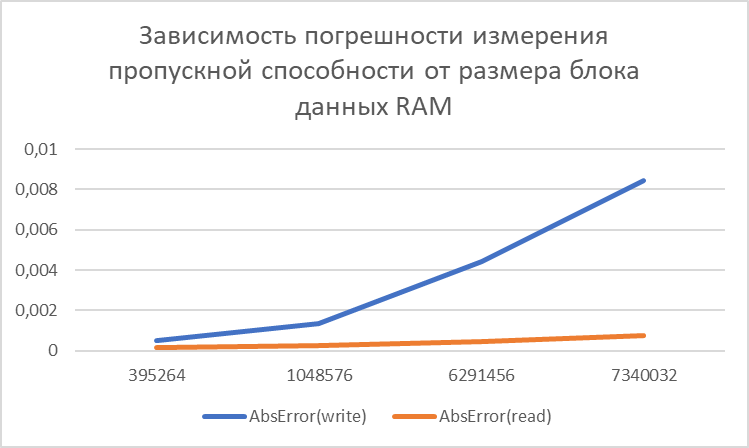
**Для первого и второго задания**

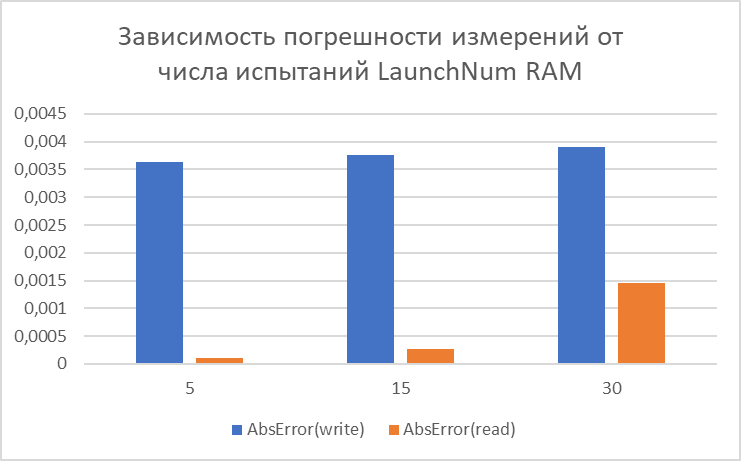
****

**Для третьего задания**

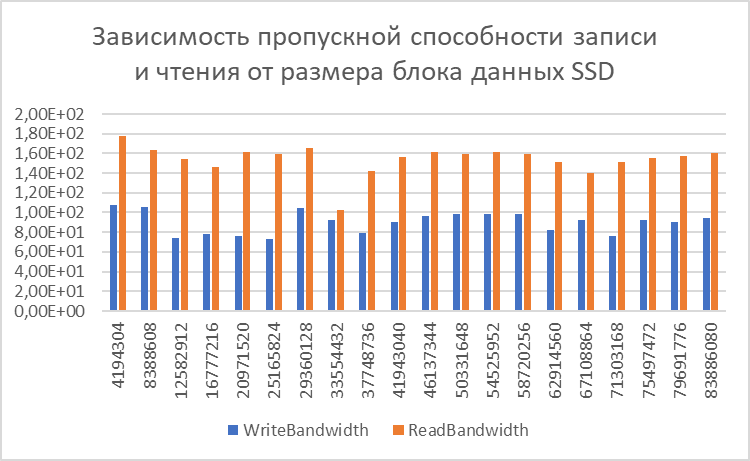
****

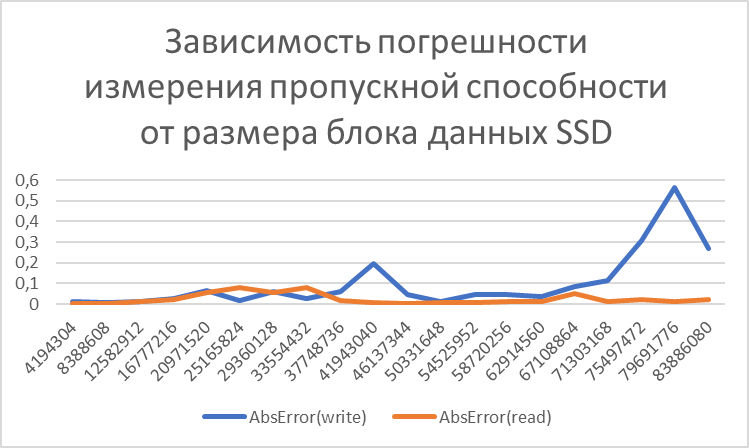
**RAM**

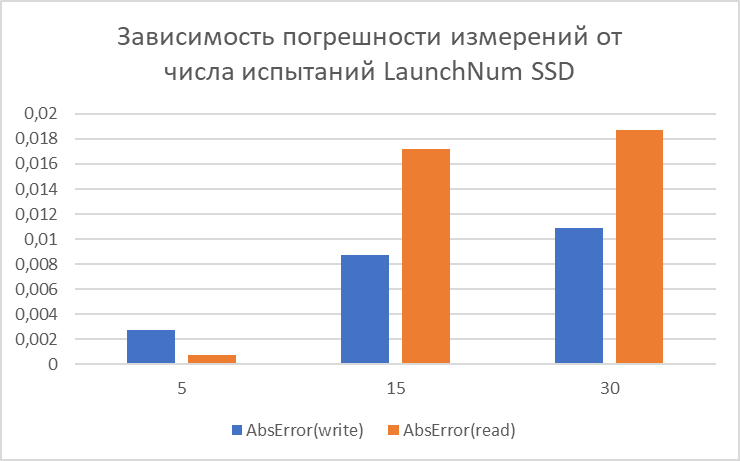
**** ****



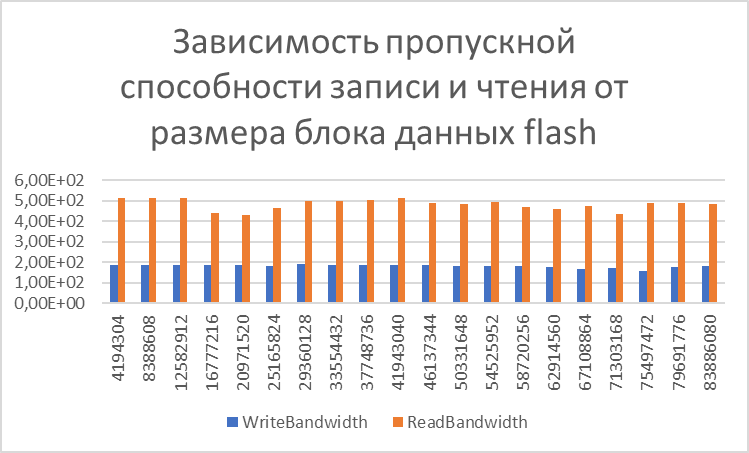
**SSD**

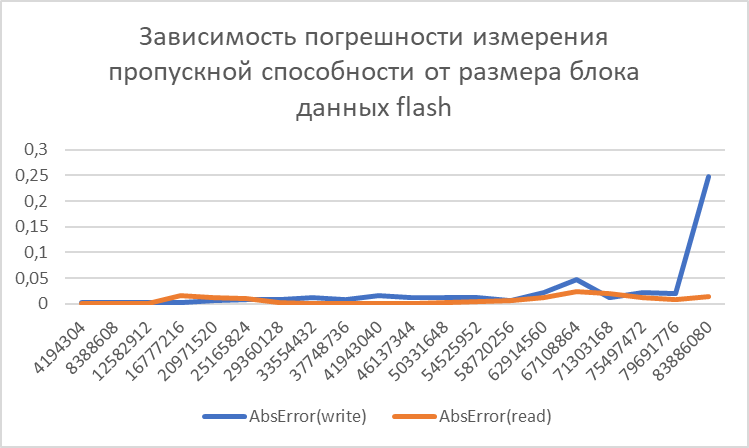
****

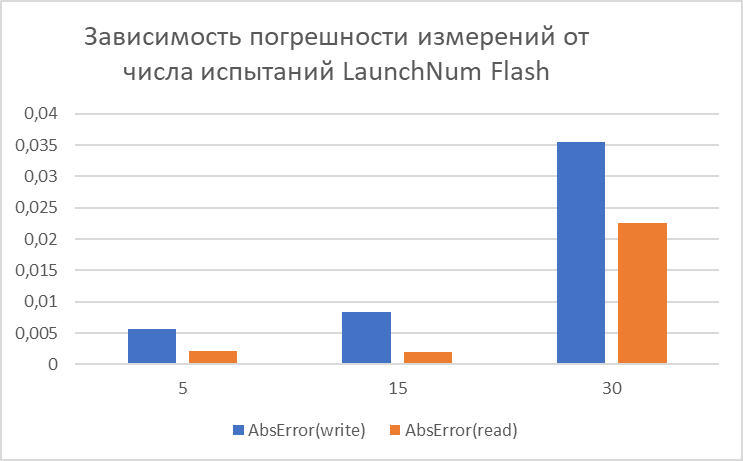
****

****

**FLASH**

****

****

****

# Приложение

#include <iostream>

#include <fstream>

#include <cstring>

#include <string>

#include <ctime>

#include <cmath>

#include <Windows.h>

// L1 386 Kb

// L2 1 Mb

// L3 6 Mb

//clock\_gettime in windows//

LARGE\_INTEGER getFILETIMEoffset()

{

SYSTEMTIME s;

FILETIME f;

LARGE\_INTEGER t;

s.wYear = 1970;

s.wMonth = 1;

s.wDay = 1;

s.wHour = 0;

s.wMinute = 0;

s.wSecond = 0;

s.wMilliseconds = 0;

SystemTimeToFileTime(&s, &f);

t.QuadPart = f.dwHighDateTime;

t.QuadPart <<= 32;

t.QuadPart |= f.dwLowDateTime;

return (t);

}

int clock\_gettime(int X, struct timeval \*tv)

{

LARGE\_INTEGER t;

FILETIME f;

double microseconds;

static LARGE\_INTEGER offset;

static double frequencyToMicroseconds;

static int initialized = 0;

static BOOL usePerformanceCounter = 0;

if (!initialized) {

LARGE\_INTEGER performanceFrequency;

initialized = 1;

usePerformanceCounter = QueryPerformanceFrequency(&performanceFrequency);

if (usePerformanceCounter) {

QueryPerformanceCounter(&offset);

frequencyToMicroseconds = (double)performanceFrequency.QuadPart / 1000000.;

} else {

offset = getFILETIMEoffset();

frequencyToMicroseconds = 10.;

}

}

if (usePerformanceCounter) QueryPerformanceCounter(&t);

else {

GetSystemTimeAsFileTime(&f);

t.QuadPart = f.dwHighDateTime;

t.QuadPart <<= 32;

t.QuadPart |= f.dwLowDateTime;

}

t.QuadPart -= offset.QuadPart;

microseconds = (double)t.QuadPart / frequencyToMicroseconds;

t.QuadPart = microseconds;

tv->tv\_sec = t.QuadPart / 1000000;

tv->tv\_usec = t.QuadPart % 1000000;

return (0);

}

//clock\_gettime in windows//

typedef unsigned int uint;

#define NANOS\_IN\_SEC 1000000000

double\* test\_ram\_w(uint block\_size, uint launch\_count)

{

srand(time(0));

double\* result = new double[launch\_count];

timeval begin, end;

for (int launch = 0; launch < launch\_count; launch++)

{

uint arr\_size = block\_size / sizeof(uint);

uint\* initial\_arr = new uint [arr\_size];

uint\* test\_arr = new uint[arr\_size];

for (uint i = 0; i < arr\_size; i++) initial\_arr[i] = rand() % 100;

clock\_gettime (0, &begin);

for (uint i = 0; i < arr\_size; i++) test\_arr[i] = initial\_arr[i];

clock\_gettime (0, &end);

//std::cout << "sec: " << end.tv\_sec - begin.tv\_sec << '\n';

//std::cout << "nanosec: " << end.tv\_usec - begin.tv\_usec << '\n';

result[launch] = (double)(end.tv\_usec - begin.tv\_usec) / NANOS\_IN\_SEC;

std::cout << result[launch] << '\n';

delete test\_arr;

}

return result;

}

double\* test\_ssd\_w(uint block\_size, uint launch\_count)

{

srand(time(0));

uint arr\_size = block\_size / sizeof(uint);

double\* result = new double[launch\_count];

timeval begin, end;

for (int launch = 0; launch < launch\_count; launch++)

{

double duration = 0;

std::ofstream out("mem\_test", std::ios::binary | std::ios::out);

clock\_gettime (0, &begin);

for (uint i = 0; i < arr\_size; i++)

{

uint randval = rand() % 100;

out.write((char\*) &randval, sizeof(uint));

}

clock\_gettime (0, &end);

out.close();

if (end.tv\_sec - begin.tv\_sec > 0)

duration = (double)((end.tv\_sec - begin.tv\_sec) \* NANOS\_IN\_SEC + end.tv\_usec - begin.tv\_usec) / NANOS\_IN\_SEC;

else

duration = (double)(end.tv\_usec - begin.tv\_usec) / NANOS\_IN\_SEC;

std::cout << "duration: " << duration << '\n';

// std::cout << "sec: " << end.tv\_sec - begin.tv\_sec << '\n';

// std::cout << "nanosec: " << end.tv\_usec - begin.tv\_usec << '\n';

result[launch] = duration;

}

return result;

}

double\* test\_flash\_w(uint block\_size, uint launch\_count)

{

srand(time(0));

uint arr\_size = block\_size / sizeof(uint);

double\* result = new double[launch\_count];

timeval begin, end;

for (int launch = 0; launch < launch\_count; launch++)

{

double duration = 0;

std::ofstream out("/media/zer0chance/Transcend/mem\_test", std::ios::binary | std::ios::out);

clock\_gettime (0, &begin);

for (uint i = 0; i < arr\_size; i++)

{

uint randval = rand() % 100;

out.write((char\*) &randval, sizeof(uint));

}

clock\_gettime (0, &end);

out.close();

if (end.tv\_sec - begin.tv\_sec > 0)

duration = (double)((end.tv\_sec - begin.tv\_sec) \* NANOS\_IN\_SEC + end.tv\_usec - begin.tv\_usec) / NANOS\_IN\_SEC;

else

duration = (double)(end.tv\_usec - begin.tv\_usec) / NANOS\_IN\_SEC;

std::cout << "duration: " << duration << '\n';

// std::cout << "sec: " << end.tv\_sec - begin.tv\_sec << '\n';

// std::cout << "nanosec: " << end.tv\_usec - begin.tv\_usec << '\n';

result[launch] = duration;

}

return result;

}

double\* test\_ram\_r(uint block\_size, uint launch\_count)

{

srand(time(0));

double\* result = new double[launch\_count];

timeval begin, end;

for (int launch = 0; launch < launch\_count; launch++)

{

uint arr\_size = block\_size / sizeof(uint);

uint\* test\_arr = new uint [arr\_size];

uint\* initial\_arr = new uint[arr\_size];

for (uint i = 0; i < arr\_size; i++) initial\_arr[i] = rand() % 100;

clock\_gettime (0, &begin);

for (uint i = 0; i < arr\_size; i++) test\_arr[i] = initial\_arr[i];

clock\_gettime (0, &end);

//std::cout << "sec: " << end.tv\_sec - begin.tv\_sec << '\n';

//std::cout << "nanosec: " << end.tv\_usec - begin.tv\_usec << '\n';

result[launch] = (double)(end.tv\_usec - begin.tv\_usec) / NANOS\_IN\_SEC;

std::cout << result[launch] << '\n';

delete initial\_arr;

}

return result;

}

double\* test\_ssd\_r(uint block\_size, uint launch\_count)

{

srand(time(0));

uint arr\_size = block\_size / sizeof(uint);

double\* result = new double[launch\_count];

timeval begin, end;

for (int launch = 0; launch < launch\_count; launch++)

{

double duration = 0;

std::ifstream ifs("mem\_test", std::ios::binary | std::ios::in);

clock\_gettime (0, &begin);

for (uint i = 0; i < arr\_size; i++)

{

uint val;

ifs.read((char\*) &val, sizeof(uint));

}

clock\_gettime (0, &end);

ifs.close();

if (end.tv\_sec - begin.tv\_sec > 0)

duration = (double)((end.tv\_sec - begin.tv\_sec) \* NANOS\_IN\_SEC + end.tv\_usec - begin.tv\_usec) / NANOS\_IN\_SEC;

else

duration = (double)(end.tv\_usec - begin.tv\_usec) / NANOS\_IN\_SEC;

std::cout << "duration: " << duration << '\n';

// std::cout << "sec: " << end.tv\_sec - begin.tv\_sec << '\n';

// std::cout << "nanosec: " << end.tv\_usec - begin.tv\_usec << '\n';

result[launch] = duration;

}

return result;

}

double\* test\_flash\_r(uint block\_size, uint launch\_count)

{

srand(time(0));

uint arr\_size = block\_size / sizeof(uint);

double\* result = new double[launch\_count];

timeval begin, end;

for (int launch = 0; launch < launch\_count; launch++)

{

double duration = 0;

std::ifstream ifs("/media/zer0chance/Transcend/mem\_test", std::ios::binary | std::ios::in);

clock\_gettime (0, &begin);

for (uint i = 0; i < arr\_size; i++)

{

uint val;

ifs.read((char\*) &val, sizeof(uint));

}

clock\_gettime (0, &end);

ifs.close();

if (end.tv\_sec - begin.tv\_sec > 0)

duration = (double)((end.tv\_sec - begin.tv\_sec) \* NANOS\_IN\_SEC + end.tv\_usec - begin.tv\_usec) / NANOS\_IN\_SEC;

else

duration = (double)(end.tv\_usec - begin.tv\_usec) / NANOS\_IN\_SEC;

std::cout << "duration: " << duration << '\n';

// std::cout << "sec: " << end.tv\_sec - begin.tv\_sec << '\n';

// std::cout << "nanosec: " << end.tv\_usec - begin.tv\_usec << '\n';

result[launch] = duration;

}

return result;

}

inline double count\_avgTime(double\* result, uint launch\_count)

{

double sum = 0;

for (int i = 0; i < launch\_count; i++)

sum += result[i];

return sum / launch\_count;

}

inline double count\_absErr(double\* result, uint launch\_count, double avgWrtTime)

{

double max = fabs(result[0] - avgWrtTime);

for (int i = 1; i < launch\_count; i++)

if (fabs(result[i] - avgWrtTime) > max)

max = fabs(result[i] - avgWrtTime);

return max;

}

int main(int argc, char\*\* argv)

{

std::string mem\_type("");

uint block\_size(0);

uint launch\_count(0);

for (int i = 1; i < argc; i++)

{

if (!strcmp(argv[i], "-m") || !strcmp(argv[i], "--memory-type")) {

if(i + 1 < argc) {

mem\_type = argv[i + 1];

i++;

}

else {

std::cout << "Memory type specified incorrectly\n";

return 1;

}

} else if (!strcmp(argv[i], "-b") || !strcmp(argv[i], "--block-size")) {

if(i + 1 < argc) {

if (argv[i + 1][strlen(argv[i + 1]) - 1] != 'b') { // not Mb or Kb -> Byte

block\_size = strtol(argv[i + 1], NULL, 10);

}

else if (argv[i + 1][strlen(argv[i + 1]) - 2] == 'K') { // Kb

block\_size = strtol(argv[i + 1], NULL, 10) \* 1024;

}

else if (argv[i + 1][strlen(argv[i + 1]) - 2] == 'M') { // Mb

block\_size = strtol(argv[i + 1], NULL, 10) \* 1024 \* 1024;

} else {

std::cout << "Block size specified incorrectly\n";

return 1;

}

i++;

}

else {

std::cout << "Block size specified incorrectly\n";

return 1;

}

} else if (!strcmp(argv[i], "-l") || !strcmp(argv[i], "--launch-count")) {

if(i + 1 < argc) {

launch\_count = strtol(argv[i + 1], NULL, 10);

i++;

} else {

std::cout << "Launch count specified incorrectly\n";

return 1;

}

} else {

std::cout << "Unknown option: " << argv[i] << '\n';

return 1;

}

}

if (launch\_count == 0) {

std::cout << "Launch count is not specified\n";

return 1;

}

if (block\_size == 0) {

std::cout << "Block size is not specified\n";

return 1;

}

if (mem\_type == "") {

std::cout << "Memory type is not specified\n";

return 1;

}

double\* result\_w = nullptr;

double\* result\_r = nullptr;

if (mem\_type == "RAM") {

result\_w = test\_ram\_w(block\_size, launch\_count);

result\_r = test\_ram\_r(block\_size, launch\_count);

} else if (mem\_type == "HDD" || mem\_type == "SSD") {

result\_w = test\_ssd\_w(block\_size, launch\_count);

result\_r = test\_ssd\_r(block\_size, launch\_count);

} else if (mem\_type == "flash") {

result\_w = test\_flash\_w(block\_size, launch\_count);

result\_r = test\_flash\_r(block\_size, launch\_count);

} else {

std::cout << "Unknown memory type\n";

return 1;

}

std::ofstream out("result.csv", std::ios::app);

double avgWrtTime = count\_avgTime(result\_w, launch\_count);

double absErr\_w = count\_absErr(result\_w, launch\_count, avgWrtTime);

double realErr\_w = (absErr\_w / avgWrtTime) \* 100;

out << mem\_type << ';' << block\_size << ";int;" << block\_size << ';'

<< launch\_count << ";clock\_gettime();" << result\_w[launch\_count - 1]

<< ';' << avgWrtTime << ';' << (block\_size / avgWrtTime) / 1000000

<< ';' << absErr\_w << ';' << realErr\_w << ';';

double avgReadTime = count\_avgTime(result\_r, launch\_count);

double absErr\_r = count\_absErr(result\_r, launch\_count, avgReadTime);

double realErr\_r = (absErr\_r / avgReadTime) \* 100;

out << result\_r[launch\_count - 1] << ';' << avgReadTime << ';'

<< (block\_size / avgReadTime) / 1000000 << ';' << absErr\_r << ';'

<< realErr\_r << '\n';

delete result\_w;

delete result\_r;

out.close();

}