Министерство образования и науки Российской Федерации

Федеральное государственное автономное   
образовательное учреждение высшего образования   
Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

Кафедра математического обеспечения и суперкомпьютерных технологий

Отчёт по производственной практике

«Вычисление справедливой цены опциона европейского типа»

**Выполнил:** студент ИИТММ гр. 381503-3,

Романов А. А.

**Проверил:** к.т.н., доцент кафедры МОиСТ ИИТММ,

Мееров И. Б.

Нижний Новгород

2018 г.

Оглавление

[Введение 3](#_Toc523774490)

[1. Основные понятия и обозначения 4](#_Toc523774491)

[1.1. Финансовый рынок 4](#_Toc523774492)

[1.2. Финансовая модель 4](#_Toc523774493)

[1.3. Понятие европейского опциона 6](#_Toc523774494)

[1. Постановка задачи 8](#_Toc523774495)

[2. Методы и алгоритмы 9](#_Toc523774496)

[3. Программная реализация 12](#_Toc523774497)

[4. Результаты экспериментов 12](#_Toc523774498)

[5. Программно-аппаратное окружение 12](#_Toc523774499)

[6. Анализ результатов 13](#_Toc523774500)

[7. Результаты запусков на кластере «Лобачевский» 14](#_Toc523774501)

[8. Результаты запусков на архитектуре KNL-AVX-512 19](#_Toc523774502)

[9. Skylake 22](#_Toc523774503)

[10. ZMM для KNL-AVX-512 и SKL-AVX-512 28](#_Toc523774504)

[Заключение 31](#_Toc523774505)

[Литература 32](#_Toc523774506)

[Приложение А. Фрагменты программного кода основного проекта 33](#_Toc523774507)

[Приложение Б. Фрагменты программного кода проекта, автоматизирующего сбор информации 34](#_Toc523774508)

# Введение

Одним из широко распространённых способов получения прибыли в современном финансовом мире является торговля на биржах, в том числе такими финансовыми инструментами, как акции и облигации. Исходя из возникновения случайных, трудно прогнозируемых событий, влияющих на финансовый рынок, и непрерывной его эволюции, имеет место задача расчёта стоимости купли-продажи активов в кратчайшие сроки (чем быстрее будет получен результат, тем меньше за это время изменится ситуация на рынке).

Так как финансовые расчёты – важная, трудоёмкая и востребованная часть индустрии, область высокопроизводительных вычислений находит своё применение в экономике и занимает там свою нишу. В данной работе будет рассматриваться применение высокопроизводительного программного обеспечения для аналитического вычисления справедливой цены опциона европейского типа. Также будет показано, что техника программирования и оптимизация алгоритмов прямым образом влияет на время расчётов, столь важное для финансового рынка.

После разработки базового алгоритма вычисления цены будут представлены несколько его оптимизированных версий; ускорение – отношение времени работы базового алгоритма к времени работы ускоренного– будет являться доказательством важности техники программирования и рефакторинга готового кода.

# Основные понятия и обозначения

# Финансовый рынок

Финансовый рынок представляет собой совокупность денежных и валютных рынков, рынков ценных (благородных) металлов, рынков финансовых инструментов, включая ценные бумаги. На рынке финансовых инструментов принято различать:

* основные (первичные) инструменты,
* производные (вторичные) инструменты;

последние являются сложными финансовыми инструментами, построенными на базе основных (более элементарных) инструментов. К числу основных финансовых инструментов относят ценные бумаги:

* банковский счет,
* облигации,
* акции.

К производным финансовым инструментам относят:

* опционы,
* фьючерсные контракты,
* варранты,
* свопы,
* комбинации и т. д.

В данной работе рассматриваются операции с самыми популярными финансовыми инструментами – опционами.

# Финансовая модель

(этот раздел написан с использованием материалов из источников [‎2],[‎5].)

Исходя из особенностей организации финансовых рынков различных стран мира выделяют три основные модели: североамериканскую (рыночную, характерные пример – США), европейскую (банковскую, пример – Германия) и смешанную.

К основным характеристикам европейской модели относятся:

* Низкая доля акционерного капитала, высокая доля финансирования за счет выпуска облигационных займов (соотношение облигаций и акций – 10:1);
* Доминирующая роль коммерческих банков на финансовом рынке;
* Традиция прямого кредитования на покрытие дефицита бюджета наряду с выпуском государством ценных бумаг;
* Высокая доля прямого банковского кредита (50-60%) в финансировании экономики страны.

Для моделирования рынка будем использовать модель Блэка-Шоулза ‎[1], представляющую собой систему дифференциальных уравнений:

Уравнение (1) – ОДУ, отражающее изменение цены облигации *.* На влияет *r* – фиксированный процент.

Уравнение (2) – стохастическое ДУ, описывающее эволюцию цены акции . Оно содержит следующие параметры:

* *δ* – ставка дивиденда
* волатильность, изменчивость — статистический финансовый показатель, характеризующий изменчивость цены.
* – Винеровский случайный процесс //Add reference
* – начальная цена акции
* – начальная цена облигации

Система имеет аналитическое решение при следующих ограничениях:

**Ограничения на активы:**

* *безрисковая ставка*. Норма доходности безрискового актива постоянна
* *случайное блуждание*. Цены подчиняются модели геометрического броуновского движения, и мы предположим, что его дрейф и волатильность постоянны
* *Акции не выплачивают дивиденды*.

**Ограничения рынка:**

* Нет возможности арбитража (т. е. нет возможности получить безрисковую прибыль).
* Можно брать и предоставлять в долг любую сумму, даже дробную, наличных денег по безрисковой ставке.
* Можно купить и продать любую сумму, даже дробную, акций (включая «короткие продажи»).
* Вышеуказанные транзакции не несут никаких сборов или затрат.

В этом случае цену акции можно найти с помощью формулы (3):

(3)

# Понятие европейского опциона

**Опцион** — договор, по которому покупатель опциона получает право, но не обязательство, купить или продать актив (акцию – рисковый актив, облигацию – безрисковый актив) по заранее оговоренной цене в течение определенного промежутка времени. В свою очередь, продавец опциона обязан продать актив или купить его у покупателя опциона в соответствии с оговоренными заранее условиями.

Согласно общепринятой терминологии, опционы делятся на два класса:

* опцион покупателя (call option, даёт право покупки)
* опцион продавца (put option, даёт право продажи)

С точки зрения финансовой инженерии важно то, что эти финансовые инструменты "работают в разных направлениях": когда доход от одного растет, доход от другого уменьшается. Именно это обстоятельство объясняет широко распространенную практику диверсификации при оперировании с опционами разных классов, быть может, и в комбинации с другими ценными бумагами.

По времени исполнения опционы классифицируются на два типа: европейские и американские.

Если опцион предъявляется к исполнению только в заранее определенный момент времени , то говорят, что – момент исполнения, а опцион является опционом европейского типа.

Если же опцион может быть предъявлен к исполнению в любой (случайный) момент времени , то говорят, что он является опционом американского типа.

После заключении договора между сторонами *P*1 и *P*2 происходит следующее: Вторая сторона выплачивает денежные средства *C* и в некоторый фиксированный момент времени *T* решает, стоит ли покупать акции по цене *K* у первой стороны. Решение принимается исходя из соотношения цены акции в момент иначальной цены . Если , покупать акции бессмысленно, сторона *P*1получает прибыль *C*, сторона *P*2 теряет *C*. В противном случае *P2* покупает у *P1* акции по цене *K*, в ряде случаев получая прибыль (зависит от соотношения C и ).

**Проблема заключается в нахождении справедливой цены европейского опциона** - минимальной цены контракта, при которой соблюдается баланс между выигрышем и проигрышем каждой из сторон. Часто справедливую цену называют рациональной стоимостью опциона.

Определим эту цену как средний выигрыш второй стороны:

, где (4)

*E* – математическое ожидание (зависит в том числе от стохастических факторов). Произведение экспоненты и скобкиотражает дисконтирование – инфляцию при фиксированной процентной ставке *r*.

Идея образования опционов была поднята Фишером Блэком, Майроном Шоулзом и Робертом Мертоном в 1973 году[‎4]. Выведенная ими формула теперь известна как формула Блэка-Шоулза, за которую в 1997 году Шоулз и Мертон получили Нобелевскую премию.

С помощью этой формулы найдём величину :

(5)

– функция стандартного нормального распределения,

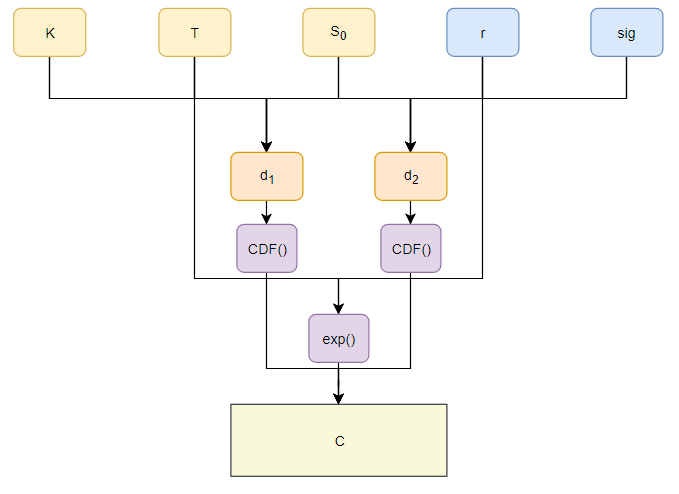
Именно эта формула ляжет в основу алгоритма, вычисляющего справедливую цену.

# Постановка задачи

Пусть известны волатильность (изменчивость) и процентная ставка(будем считать их постоянными). время исполнения опциона, начальная его цена и цена исполнения опциона . Необходимо разработать базовый алгоритм, вычисляющий справедливую цену для набора опционов с помощью формулы (5). После этого оптимизировать его с целью уменьшения времени расчёта. После получения результатов работы алгоритмов на высокопроизводительных системах сравнить время работы базовой и модернизированной версий и сделать вывод о влиянии приёмов программирования на время работы программы БЛАБЛАБЛА АРХИТЕКТУРЫ.

# Методы и алгоритмы

Ниже представлен граф информационных зависимостей для вычисления цены опциона:



Базовая версия алгоритма раз по известным параметрам находит справедливую цену европейского опциона. Схема вычислений состоит последовательном нахождении всех неизвестных формулы (5).

Предварительно объявим массивы pT, pK, pS0, pC, в которых будем хранить время исполнения опциона, фиксированную, начальную и справедливую цену соответственно

В цикле по числу запусков (i от 0 до N):

{

* + - Вычислить d1;
    - Вычислить d2;
    - Вычислить значение функции стандартного нормального распределения для d1;
    - Вычислить значение функции стандартного нормального распределения для d1;
    - Получить справедливую цену опциона по формуле (5).

}

приёмы, выкладки

# **Программная реализация**

**Структура проекта**

Программный комплекс собирается с помощью Cmake – системы автоматизации сборки программного обеспечения из исходного кода – и состоит из нескольких модулей, среди которых:

* Проекты CallOption и CallPutOption//не отражён в отчёте//. Содержат декларации необходимых переменных и функций. Также включают реализации всех алгоритмов для подсчёта справедливой цены европейского опциона на продажу и на продажу и покупку соответственно.
* gtest. Статическая библиотека с google-тестами.
* tests. Содержит реализацию тестового покрытия алгоритмов.

**Основные структуры данных**

*Объявленные переменные:*

int num\_Threads;

int N; //amount of options

int version;

double \_time;

double start, finish;

const float invsqrt2 = 0.707106781f;

const float sig = 0.2f; // volatility; percent per year 0.2 -> 20%

const float r = 0.05f; // the interest rate; percent per year 0.05 -> 5%

const float T = 3.0f; // option execute time (years)

const float S0 = 100.0f; // option price at t == 0;

const float K = 100.0f; // strike price -- price fixed in option

*Новый тип данных:*

Typedef void(\*GetPrices)(float \*pT, float \*pK, float \*pS0, float \*pC);

Ниже приведена таблица, содержащая обозначения и отражающая суть алгоритмов, вычисляющих стоимость опциона:

Таблица 0:

|  |  |  |
| --- | --- | --- |
| Название | | я |
| V0 | Базовая реализация | |
| V1 | Не смешивать типы данных | |
| V2 | Эквивалентые преобразования (Erf) | |
| V3 | Векторизация цикла (restrict) | |
| V4 | Векторизация цикла (директивы) | |
| V5 | Вынос инварианта из цикла | |
| V6 | Использование интринсика invsqrt | |

*Массив указателей на функции, содержащие модифицированные алгоритмы:*

GetPrices option\_array[9] =

{

\_V0, //base

\_V1, //float

\_V2, //erf

\_V3, //restrict

\_V4, //#pragma simd #pragma vector always

\_V5, //#pragma simd invsqrt2\_1

\_V6, //#pragma simd invsqrt2\_2

\_V7, //#pragma simd #pragma omp parallel for private

\_V8, // \_V7 + #pragma vector nontemporal

};

*Пример тестового покрытия:*

Сравним получившуюся цену pC[0] с верным результатом,

TEST(EU\_OP, test\_V0)

{

float \*pT, \*pK, \*pS0;

floatpC;

\*pT = T;

\*pS0 = S0;

\*pK = K;

option\_array[0](pT, pK, pS0, &pC);

ASSERT\_EQ(pC, float(20.9244));

}

после чего говорим о корректности работы алгоритма.

**Версия V1. Исключение ненужных преобразований типов**

Для вычисления справедливой цены достаточно использовать single precision, поэтому заменим функции log, sqrt, exp на соответствующие logf, sqrtf, expf и добавим к числам суффикс **f**.

**Версия V2. Эквивалентные преобразования**

Функцию vsCdfNorm() можно заменить на erff(), так как vsCdfNorm() == 0.5f + erff().

Изменим код:

erf1 = 0.5f + 0.5f \* erff(d1 / sqrtf(2.0f));

erf2 = 0.5f + 0.5f \* erff(d2 / sqrtf(2.0f));

pC[i] = pS0[i] \* erf1 - pK[i] \* expf((-1.0f) \* r \*pT[i]) \* erf2;

**Версия V3. Ключевое слово restrict**

Сначала необходимо сменить режим компиляции с SSE на AVX, используя ключ –xAVX. После этого добавим ключевое слово restrictв декларацию функции:

void \_V3(float\* restrict pT, float\* restrict pK, float\* restrict pS0, float\* restrict pC);

Restrict означает, что на данную область памяти не будут ссылаться другие указатели, то есть массивы не пересекаются, и их можно векторизовать

**Версия V4. Использование директив**

Ещё один способ векторизации заключается в использовании перед циклом директив #pragma simd и #pragma vector always, дающих знать компилятору о том, что с точки зрения программиста массивы не пересекаются и о том, что если векторизация возможна, то она эффективна.

#pragma simd

#pragma vector always

for (i = 0; i < N; i++)

{…}

**Версия V5. Вынос инварианта.**

Чтобы сэкономить время на многократном подсчёте значения, используем изначально объявленную константу.

const float invsqrt2 = 0.707106781f;

Заменим соответствующие строки.

erf1 = 0.5f + 0.5f \* erff(d1 \* invsqrt2);

erf2 = 0.5f + 0.5f \* erff(d2 \* invsqrt2);

**Версия V6. Эквивалентные преобразования. Вычисление квадратного корня.**

Заменим деление умножением, это ещё больше сократит время работы программы. Заменим 1f / sqrtf(pT[i]) на функцию vsInvSqrt()из библиотеки MKL и проверим, сделал ли компилятор замену деления умножением самостоятельно.

**Версия V7. Параллельная версия.**

Используем библиотеку <omp.h> для распараллеливания цикла. Перед началом цикла добавим

#pragma omp parallel for private(invf, d1, d2, erf1, erf2)

**Версия V8. Оптимизация кэша.**

Из четырёх массивов, с которыми мы работаем, только три используются для чтения. В четвёртый (pС) происходит запись результатов, которые в цикле никак не используются, поэтому имеет смысл записывать массив pC напрямую в память, минуя кэш. При таком подходе будут уменьшены накладные расходы на перенос данных, будет сэкономлено по одной операции чтения из кэша за итерацию цикла.

Для того, чтобы дать понять компилятору об отсутствии необходимости кэшировать четвёртый массив, используем #pragma vector nontemporal.

Non-Temporal Store — использование режима прямого доступа в память при осуществлении операций записи. Данный режим доступа осуществляет запись данных в память без предварительного считывания старых данных в систему кэш-уровней процессора, что минимизирует «засорение» кэша процессора ненужными данными, в частности, при операции копирования данных.

# Результаты экспериментов

# Программно-аппаратное окружение

Программное обеспечение:

Операционная система Linux CentOS 6.2

IDLE: Microsoft Visual Studio 2015, PyCharm

Компилятор Intel C++ 17.0

Библиотеки Intel OpenMP, Intel MKL

Аппаратное обеспечение:

Запуски проводились на двух различных кластерах:

**кластер Intel Endeavor.**

Процессор: 2x 20-ядерных процессора Intel Xeon Gold 6148 (2.4GHz) (Skylake)  
Память: 192 GB  
Операционная система: Red Hat Enterprise Linux Server release 7.5 (Maipo)  
Компилятор: Intel C++ Composer 2018 (всоставе Intel Parallel Studio XE 2018 update 2)

# Анализ результатов

Был написан программный комплекс, автоматизирующий сбор результатов: создаётся каталог, содержащий отчёты об оптимизации, лог с минимальным временем работы каждого алгоритма для разного числа потоков и таблицу с конечными результатами.

Ниже будут представлены следующие ключевые моменты исследования:

* сравнение времени работы последовательных версий кода
* сравнение времени работы параллельных версий кода
* эффективность масштабируемости параллельной реализации для параллельных версий
* Roofline-графики для параллельных версий кода
* сравнение времени работы версий кода с точной и неточной арифметикой.
* Использование ZMM-регистров.

# Результаты запусков на кластере «Лобачевский»

// add concrete name of arch and info about arch

Результаты экспериментов приведены в таблицах 1 – 3.

Количество опционов. Каждый запуск выполнен 15 раз, приведено минимальное время работы в миллисекундах.

Таблица 1. Сравнение времени работы последовательных версий кода

//todo: add links to the tables

|  |  |  |
| --- | --- | --- |
| Номер | Версия | Минимальное время, мс |
| V0 | Базовая реализация | 18530 |
| V1 | Не смешивать типы данных | 16440 |
| V2 | Эквивалентые преобразования (Erf) | 5832 |
| V3 | Векторизация цикла (restrict) | 1069 |
| V4 | Векторизация цикла (директивы) | 1070 |
| V5 | Вынос инварианта из цикла | 1068 |
| V6 | Использование интринсика invsqrt | 5678 |

Существенное замедление работы алгоритма V6 требует объяснения. Компилятор вызывает точный корень. При отдельном подсчёте компилятор вынужден предоставить точный результат, так как не знает, каким образом и в каких функциях результат будет использоваться. Если в коде делить на корень, последний заменяется неточным, что в разы быстрее (см V5). Для данной задачи высокая точность вычислений не важна, и экономия времени на вычисление лишних трёх знаков после запятой может дать ускорение в 5 раз.

В дальнейших экспериментах алгоритм V6 будет исключён из рассмотрений.

Таблица 2. Сравнение времени работы параллельных версий кода (16 потоков).

|  |  |  |
| --- | --- | --- |
| Номер | Версия | Минимальное время, мс |
| V7 | #pragma simd #pragma omp parallel for private | 99 |
| V8 | \_V7 + #pragma vector nontemporal | 106 |

Время работы алгоритмов V7 и V8 совпадает с точностью до , это означает, что мощности кластера достаточно для того, чтобы хранить в кэше все четыре массива без потери производительности.

Таблица 3. Эффективность масштабируемости параллельной реализации для версии V8

|  |  |  |
| --- | --- | --- |
| Число потоков | Минимальное время, мс | Ускорение |
| 1 | 1123 | 1 |
| 2 | 577 | 1,964 |
| 4 | 305 | 3,682 |
| 8 | 146 | 7,692 |
| 16 | 106 | 10,594 |

Рис. 1. Пиковое и действительное ускорение

Можно заметить, что параллельная версия имеет ускорение, примерно совпадающее с числом потоков (В случае 16 потоков имеем ускорение 10.5 вместо 16 из-за значительных накладных расходов на создание и уничтожение потоков). Для данной задачи практически полное соответствие фактического ускорения на двух-восьми потоках пиковому – отличный результат.

Рис. 2. Зависимость времени работы алгоритма V8 от числа потоков

После всех оптимизаций алгоритма лучший результат показал V8, сократив время работы по сравнению с базовой версией в раза.

В таблицах 4 и 5 приведены результаты запуска программного кода на кластере «Лобачевский». При компиляции был использован ключ c командой понижения точности float до четырёх знаков после запятой: -fimf-precision=low -fimf-domain-exclusion=31

Таблица 4. Время работы алгоритмов с неточной арифметикой

|  |  |  |  |
| --- | --- | --- | --- |
| Номер | Версия | , мс | , мс |
| V0 | preference 1 | 17781 | 47173 |
| V1 | preference 2 | 15349 | 41044 |
| V2 | erf | 5919 | 15503 |
| V3 | restrict | 953 | 2539 |
| V4 | #pragma simd #pragma vector always | 955 | 2538 |
| V5 | #pragma simd invsqrt2\_1 | 952 | 2537 |

Рис. 3. Время работы последовательных версий алгоритма, опционов, мс

После сравнения значений таблиц 1 и 4 можно утверждать, что сокращение точности вычислений там, где высокая точность не нужна, оправдана и значительно ускоряет работу программы.

Таблица 5. Время работы параллельных версий

|  |  |  |  |
| --- | --- | --- | --- |
| Номер версии | Число потоков | , мс | , мс |
| V7 | 1 | 427 | 1152 |
| 2 | 446 | 1186 |
| 4 | 232 | 619 |
| 8 | 143 | 338 |
| 16 | 86 | 185 |
| V8 | 1 | 420 | 1143 |
| 2 | 445 | 1170 |
| 4 | 231 | 609 |
| 8 | 143 | 334 |
| 16 | 77 | 182 |

Рис. 4. Время работы параллельных версий алгоритма, опционов, мс

Незначительно замедление работы алгоритмов на двух потоках связано с накладными расходами на создание дополнительного потока. Далее наблюдается ускорение, близкое к линейному.

# Результаты запусков на архитектуре KNL-AVX-512

// add concrete name of arch and info about arch

Рассмотрим последовательные версию кода. В архитектуре реализован регистр длиной 512 бит, который одновременно может хранить 16 чисел с плавающей запятой одинарной точности. Алгоритм, содержащий вызов erf(), дал неожиданный результат: имеется ускорение от его векторизации примерно в 27 раз вместо 16. Это связано с тем, что компилятор генерирует два неэквивалентных кода (доказательства) и в векторном коде используется предподсчёт – один раз вычисленный результат используется дважды.

Таблица 6. Время работы последовательных версий алгоритма

|  |  |  |  |
| --- | --- | --- | --- |
| Номер | Версия | , мс | , мс |
| V0 | preference 1 | 89517 | 264187 |
| V1 | preference 2 | 86811 | 254826 |
| V2 | erf | 31589 | 84697 |
| V3 | restrict | 1168 | 3140 |
| V4 | #pragma simd #pragma vector always | 1159 | 3122 |
| V5 | #pragma simd invsqrt2\_1 | 1175 | 3147 |

Перейдём к параллельным версиям. Сначала запускается по одному потоку на ядро, задействуя 1, 2, …, 68 ядер. Если запустить по 2 или 4 потока на ядро, прироста производительности не наблюдается. Это говорит о том, что для данной задачи одно ядро полностью прогрузило все устройства, которые ему необходимы. На данной архитектуре наблюдается ускорение версии V8 относительно V7. //почему? 10 миллисекунд – погрешность, ей следует пренебречь.

Таблица 7. Время работы параллельных версий алгоритма

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Номер версии | Число потоков | , мс | , мс | Ускорение для |
| V7 | 1 | 1112 | 2959 | 1,00 |
| 2 | 554 | 1481 | 1,99 |
| 4 | 275 | 739 | 4,00 |
| 8 | 137 | 367 | 8,06 |
| 16 | 68 | 184 | 16,08 |
| 17 | 64 | 173 | 17,10 |
| 34 | 33 | 87 | 34,01 |
| 68 | 25 | 69 | 42,88 |
| 136 | 24 | 66 | 44,83 |
| 272 | 27 | 75 | 39,45 |
| V8 | 1 | 1101 | 2952 | 1,00 |
| 2 | 552 | 1483 | 1,99 |
| 4 | 273 | 742 | 3,97 |
| 8 | 139 | 369 | 8,00 |
| 16 | 69 | 185 | 15,95 |
| 17 | 65 | 174 | 16,96 |
| 34 | 33 | 87 | 33,93 |
| 68 | 21 | 57 | 51,78 |
| 136 | 19 | 53 | 55,69 |
| 272 | 22 | 60 | 49,20 |

Рис. 5. Время работы параллельных версий алгоритма, опционов, мс.

Пиковое ускорение в данном случае следует рассматривать как отношение времени работы алгоритма на максимально возможном числе ядер ко времени работы на одном ядре. Фактическое ускорение получилось равным 43 при пике в 68. Объяснить это можно с помощью roofline-графика.

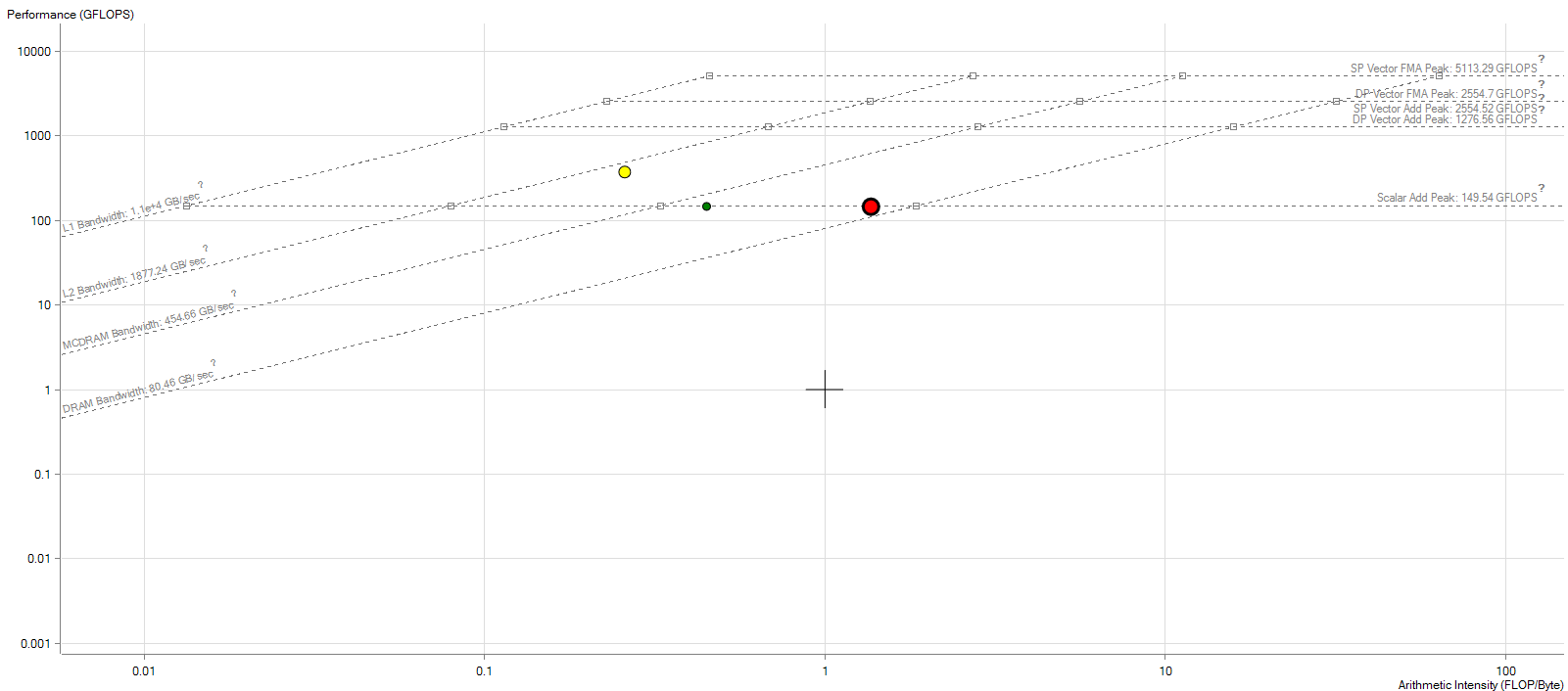


Рис. 6. Roofline-график для версии V8

На горизонтальной оси графика откладывается арифметическая интенсивность – число операций с плавающей запятой на количество прокаченных через шину данных . На вертикальной оси – пропускная способность (GFLOPS). Горизонтальные прямые соответствуют пиковой производительности в конкретных условиях. Диагональные линии соответствуют пропускным способностям подсистемы памяти (DRAM, MCDRAM, кэши L1 и L2). Вертикальная прямая, содержащая точку пересечения диагональной и горизонтальной линий разделяет график на две части. Если алгоритм находится левее вертикали, то может замедляться из-за недостатка памяти. Если правее – из-за пропускной способности.

Точками на графике отмечены алгоритмы, содержащиеся в коде. Жёлтым цветом обозначена функция erf(), зелёным – логарифм, красным – собственный код. Не хватает оверолла.

Erf() почти не останавливает нехватка пропускной способности и арифметической интенсивности, но логарифм и собственный код, имея хорошую арифметическую интенсивность, останавливаются скалярным пиком сложения, поэтому производительность не может стать выше, чем 149,5 GFLOPS.

# Skylake

// add concrete name of arch and info about arch

Исследовалось два режима: набор команд АVХ-2(регистр длиной 256 бит) и АVХ-512. Будут сравниваться времена работы версий для регистров разной длины. Говоря в общем, тенденция, которая прослеживалась в предыдущих результатах, сохраняется (см. рис. 7).

Рис. 7. Сравнение времени работы последовательных версий алгоритма на разных кластерах, опционов, мс

Таблица 8. Время работы последовательных алгоритмов на AVX-256 и AVX-512

|  |  |  |  |
| --- | --- | --- | --- |
| Номер | Версия | AVX-256 | AVX-512 |
| V0 | preference 1 | 47415 | 47676 |
| V1 | preference 2 | 40576 | 40595 |
| V2 | erf | 14540 | 14569 |
| V3 | restrict | 2039 | 1644 |
| V4 | #pragma simd #pragma vector always | 2039 | 1638 |
| V5 | #pragma simd invsqrt2\_1 | 2030 | 1637 |

Ускорение от векторизации при пике в 16 равно восьми. Довольно мало, но объяснимо: на SKL-AVX-512 векторизация неоднозначна, и экспериментально проверено, что она не может ускорить работу кода в 16 раз. Возможно, векторизация реализована с понижением частот.

Рис. 8. Сравнение времени работы последовательных версий алгоритма для регистров разной длины, опционов, мс.

На рис. 8 видно, что при прочих равных условиях имеется незначительный выигрыш во времени за счёт более современной архитектуры узла.

Рассмотрим теперь результаты, полученные при запуске параллельных версий:

Таблица 9. Время работы параллельных алгоритмов на регистрах разной длины, опционов, мс

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номер версии | Число потоков | AVX-256 | Ускорение для AVX-256 | AVX-512 | Ускорение AVX-512 |
| V7 | 1 | 1857 | 1,00 | 1483 | 1,00 |
| 2 | 1131 | 1,64 | 892 | 1,66 |
| 4 | 528 | 3,51 | 392 | 3,78 |
| 8 | 243 | 7,64 | 183 | 8,10 |
| 16 | 126 | 14,73 | 104 | 14,2 |
| 20 | 103 | 18,02 | 96 | 15,4 |
| 40 | 52 | 35,71 | 50 | 29,6 |
| 80 | 52 | 35,71 | 52 | 28,5 |
| V8 | 1 | 1866 | 1,00 | 1490 | 1,00 |
| 2 | 1140 | 1,63 | 722 | 2,06 |
| 4 | 542 | 3,44 | 391 | 3,81 |
| 8 | 249 | 7,49 | 184 | 8,09 |
| 16 | 127 | 14,69 | 102 | 14,6 |
| 20 | 104 | 17,94 | 96 | 15,5 |
| 40 | 52 | 35,88 | 49 | 30,4 |
| 80 | 49 | 38,08 | 49 | 30,4 |

Переход с 40 потоков на 80 – программный гипер-трейдинг – он не дал результатов (так же, как и аппаратный). На SKL-AVX-512 фактическое ускорение хуже, чем на SKL-AVX-256: при равном времени ускорение 30 из 40 на SKL-AVX-512 против 35 из 40 на SKL-AVX-256.

Аппаратный гипер-трейдинг с четырьмя потоками на ядро не даёт прироста скорости, но с двумя потоками имеется ускорение в два раза относительно запуска без гипер-трейдинга. В остальном, результаты имеют тенденцию, аналогичную полученным на кластере «Лобачевский» результатам.

Рис. 9. Ускорение версии V8 для различных наборов инструкций

Ниже представлены roofline-графики для алгоритма V8:

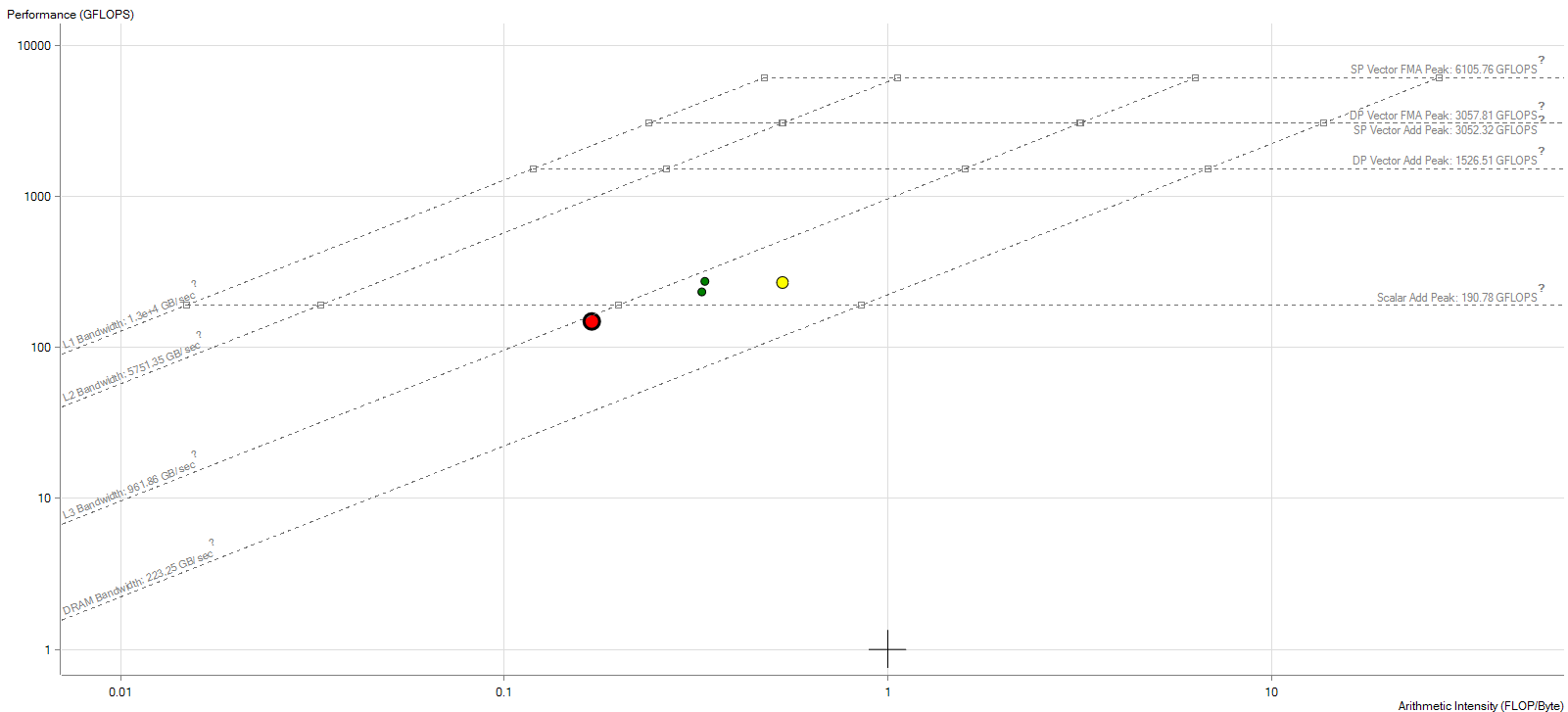


Рис. 10. Roofline-график для SKL-AVX-256

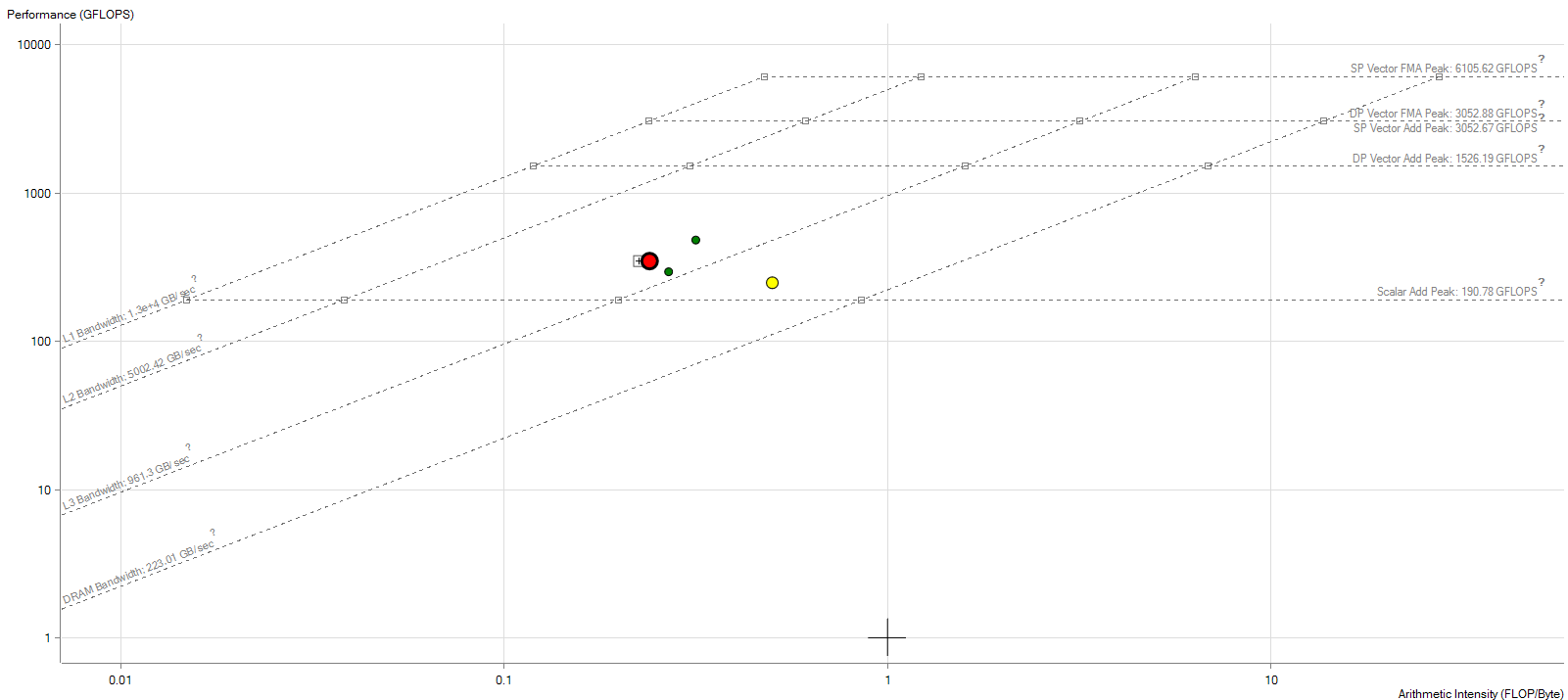


Рис. 11. Roofline-график для SKL-AVX-512

Красная точка на графиках – функция erf(), при длине регистра в 256 бит она ограничена кэшем L3, при длине в 512 бит ограничений в пропускной способности нет.

Жёлтая точка – собственный код. Зелёные точки – логарифмы. Все математические функции на рис. 11 расположены выше, чем на рис. 10. Это говорит о более успешной их реализации в AVX-512.

Рассмотрим также программный код, собранный на узле SKL-512 с ключом компиляции для неточной арифметики. Ниже приведена таблица, содержащая результаты запусков параллельных версий кода с точной и неточной арифметикой.

Таблица 10. Точные и неточные вычисления на SKL-AVX-512

|  |  |  |  |
| --- | --- | --- | --- |
| Номер версии | Число потоков | Точные вычисления | Неточные вычисления |
| V8 | 1 | 1483 | 1128 |
| 2 | 892 | 597 |
| 4 | 392 | 240 |
| 8 | 183 | 122 |
| 16 | 104 | 95 |
| 20 | 96 | 95 |
| 40 | 50 | 52 |
| 80 | 52 | 52 |

Рис. 12. Точные и неточные вычисления на SKL-AVX-512

На рис. 12 видно, что до тех пор, пока неточные вычисления не были ограничены пропускной способностью или математической интенсивностью, они достаточно сильно ускоряли работу алгоритмов. Но при большом числе потоков влияние на скорость неточной математики аннигилируется из-за невозможности кластера считать с нужной пропускной способностью или интенсивностью.

# ZMM для KNL-AVX-512 и SKL-AVX-512

Результаты запусков последовательных и параллельных версий алгоритма с использованием ZMM в целом повторяют результаты запусков без использования ZMM. Следует обратить внимание на следующие два момента:

Незначительное замедление работы параллельных версий на KNL с использованием ZMM

Таблица 11. Сравнение параллельных версий, опционов, мс.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Номер версии | Число потоков | KNL | KNL+ZMM |  |
| V8 | 1 | 2952 | 3037 |  |
| 2 | 1483 | 1516 |  |
| 4 | 742 | 762 |  |
| 8 | 369 | 382 |  |
| 16 | 185 | 193 |  |
| 17 | 174 | 180 |  |
| 34 | 87 | 91 |  |
| 68 | 57 | 58 |  |
| 136 | 53 | 55 |  |
| 272 | 60 | 58 |  |

Видимое ускорение работы параллельных версий на SKL с использованием ZMM на небольшом числе потоков. При 20+ потоках алгоритм ограничен сверху пропускной способностью или арифметической интенсивностью

Таблица 12. Сравнение параллельных версий, опционов, мс.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Номер версии | Число потоков | SKL | SKL+ZMM |  |
| V8 | 1 | 1490 | 1202 |  |
| 2 | 722 | 502 |  |
| 4 | 391 | 287 |  |
| 8 | 184 | 140 |  |
| 16 | 102 | 97 |  |
| 20 | 96 | 95 |  |
| 40 | 49 | 49 |  |
| 80 | 49 | 49 |  |

Ниже представлен roofline-график версии V8 для архитектуры KNL-AVX-512+ZMM.

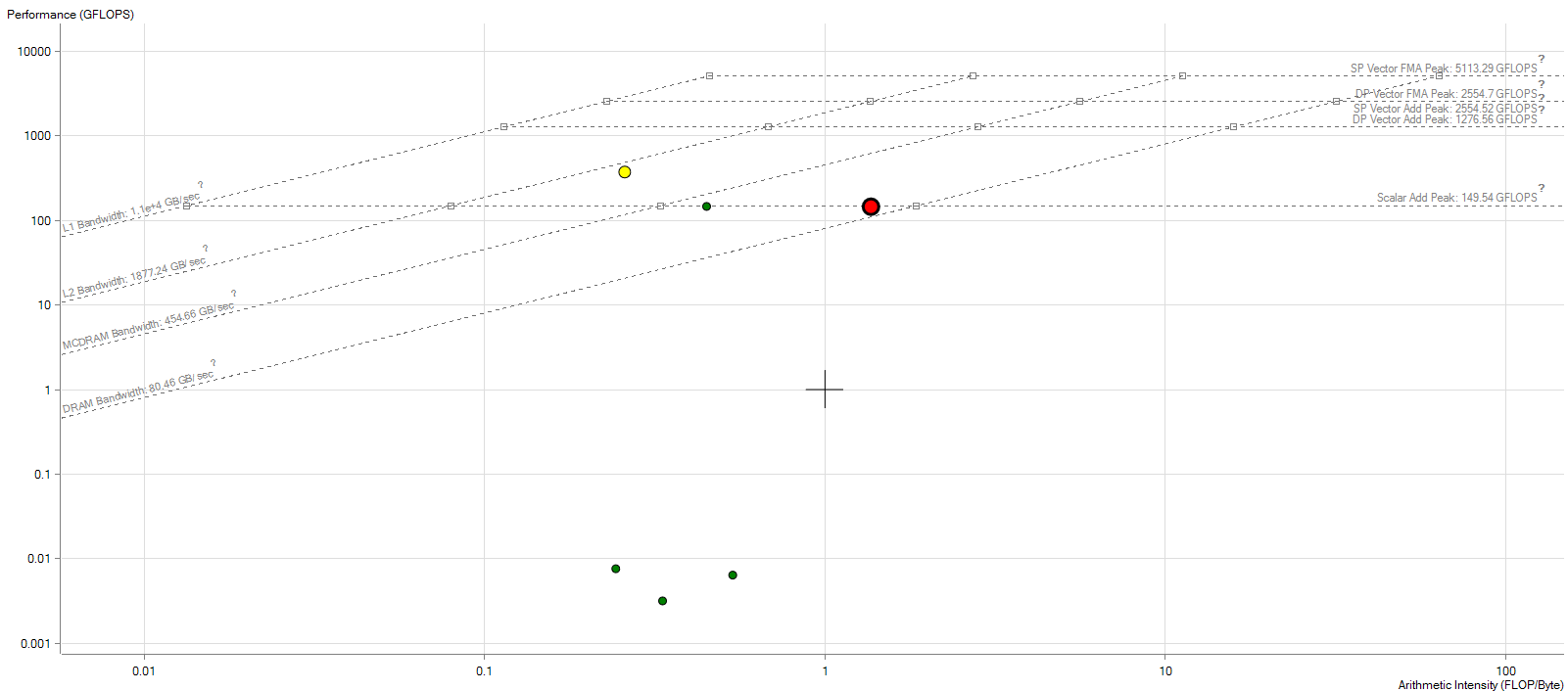


Рис. 13. roofline-model для SKL-AVX-512+ZMM

# Заключение

Была разработана начальная версия алгоритма, вычисляющего справедливую цену опциона по формуле 5. Было разработано 8 различных модификаций базовой версии, каждая из которых давала значительное ускорение. Наибольшее ускорение было получено в последней параллельной версии алгоритма – время работы программы уменьшилось в 175 раз.

Алгоритмы были запущены на различных кластерах, были собраны результаты работы версий и проведён их анализ с использованиям информации из файлов с отчётами об оптимизации и файлов, содержащих дизассемблер. Алгоритмы также анализировались с помощью Intel VTune Performance Analyzer.

Из-за большого объёма однообразных действий возникла потребность в автоматизации работы программного комплекса. Для решения проблемы были написаны скрипты, выполняющие рутинные действия.

Полученное ускорение в работе алгоритмов показывает важность техники программирования и необходимости добиться максимальной оптимизации рабочего кода в тех задачах, где важна скорость получения точного результата.

# Литература

1. Black F., Scholes M. The pricing of options and corporate liabilities // Journal of political economy – 1973 – Т. 81 – №. 3 – С. 637–654. https://www.cs.princeton.edu/courses/archive/fall09/cos323/papers/black\_scholes73.pdf
2. Оптимизация расчетов на примере задачи вычисления справедливой цены опциона Европейского типа. <https://www.intuit.ru/studies/courses/14004/1095/lecture/22917>
3. Справедливая стоимость опциона. <https://utmagazine.ru/posts/17075-spravedlivaya-stoimost-opciona>
4. Модель Блэка-Шоулза <https://en.wikipedia.org/wiki/Black–Scholes_model>
5. Модели финансового рынка и их особенности <https://studopedia.su/10_149780_modeli-finansovogo-rinka-i-ih-osobennosti.html>
6. Документация MKL<http://old.parallel.ru/ftp/libs/mkl/mklman52.pdf>
7. Оценка и оптимизация производительности вычислений на многоядерных системах <https://habr.com/company/intel/blog/277407/>
8. Knights Landing (KNL) <https://www.hotchips.org/wp-content/uploads/hc_archives/hc27/HC27.25-Tuesday-Epub/HC27.25.70-Processors-Epub/HC27.25.710-Knights-Landing-Sodani-Intel.pdf>

# Приложение А. Фрагменты программного кода основного проекта

void \_V8(float \*pT, float \*pK, float \*pS0, float \*pC)

{#pragma simd

#pragma vector nontemporal

#pragma omp parallel for private(d1, d2, erf1, erf2)

for (i = 0; i < N; i++)

{

d1 = (logf(pS0[i] / pK[i]) + (r + sig \* sig \* 0.5f) \*

pT[i]) / (sig \* sqrtf(pT[i]));

d2 = (logf(pS0[i] / pK[i]) + (r - sig \* sig \* 0.5f) \*

pT[i]) / (sig \* sqrtf(pT[i]));

erf1 = 0.5f + 0.5f \* erff(d1 / sqrtf(2.0f));

erf2 = 0.5f + 0.5f \* erff(d2 / sqrtf(2.0f));

pC[i] = pS0[i] \* erf1 - pK[i] \* expf((-1.0f) \* r \* pT[i]) \* erf2;

}

}

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

{

version = atoi(argv[1]);

N= atoi(argv[2]);

num\_Threads = atoi(argv[3]);

float\* pT = new float[4 \* N];

float\* pK = pT + N;

float\* pS0= pT + 2 \* N;

float\* pC = pT + 3 \* N;

for (int i = 0; i < N; i++)

{

pT[i] = T;

pS0[i] = S0;

pK[i] = K;

}

start = omp\_get\_wtime();

option\_array[version](pT, pK, pS0, pC);

finish = omp\_get\_wtime();

\_time = finish - start;

std::cout << \_time << std::endl;

std::cout << pC[1];

delete[] pT;

return 0;

}

# Приложение Б. Фрагменты программного кода проекта, автоматизирующего сбор информации

def one\_res(name):

proc = Popen(name, shell=True, stdout=PIPE, stderr=PIPE)

proc.wait()

res = proc.communicate()

if proc.returncode:

return res[1]

return res[0]

def min\_res(name):

min\_time = 10000

for i in range(count):

time = float(one\_res(name))

if time < min\_time:

min\_time = time

return min\_time

def get\_times(opts\_amo):

for version in range(num\_seq\_ver):

name = os.path.join(os.getcwd(), str(file)) + " " + \

str(version) + " " + str(opts\_amo) + " " + str(1)

print(datetime.datetime.now())

seq\_times.append((min\_res(name)))

print(seq\_times)

for version in par\_ver:

for thr in threads:

#name = affinity[thr] + \

name = os.path.join(os.getcwd(), str(file)) + " " + \

str(par\_ver[version]) +" " + str(opts\_amo) + " " + str(threads[thr])

par\_times.append((min\_res(name)))

print(datetime.datetime.now())

print(par\_times)

def write\_log(opts\_amo):

\_iter = 0

with open('summary\_' + str(opts\_amo) + '.log', 'a') as f:

for i in seq\_times:

info = "%s %s %s %s %s %s" % ("version ", str(name\_version[\_iter]),";amo of opts ",

str(opts\_amo), "; counts ", str(count))

f.write(str(datetime.datetime.now()) + " " + info + "; min time " + str(i) + " seconds" + "\n")

\_iter += 1