Нижегородский государственный университет им. Н.И. Лобачевского
Факультет вычислительной математики и кибернетики

Образовательный комплекс «Введение в принципы функционирования и применения современных мультиядерных архитектур (на примере Intel Xeon Phi)»

Лабораторная работа №4
Оптимизация расчетов на примере задачи вычисления справедливой цены опциона Европейского типа

Мееров И.Б., Сысоев А.В.

При поддержке компании Intel

Нижний Новгород
2013

Содержание

Введение

> Если люди не полагают, что математика проста, то только потому, что они не понимают, как на самом деле сложна жизнь.
>
> *Джон фон Нейман*

В данной лабораторной работе мы иллюстрируем некоторые приемы оптимизации вычислений в научных и инженерных приложениях. Известна обывательская точка зрения, что такие приложения возникают исключительно в «классических» областях, связанных с промышленностью и производством, например, в машиностроении. Однако применение высокопроизводительной вычислительной техники не ограничивается исключительно физическими расчетами. Так, область высокопроизводительных вычислений (HPC, high performance computing) включает приложения из биологии, химии, медицины, экономики, финансов и др. В этой работе мы сосредоточим свое внимание на достаточно простых расчетах, возникающих при анализе финансовых рынков и происходящих на них процессах. В наших учебных курсах мы обращаемся к этой теме не первый раз (см. ЛР «….» из курса ПарЧМ) [3]. Вызвано это тем, что финансовые расчеты – важная, актуальная, востребованная в индустрии часть области HPC. При этом погружение в модели и методы их анализа требует существенной математической подготовки, а реализация алгоритмов для современной аппаратуры – определенной техники программирования. Упомянутая выше лабораторная работа разрабатывалась как часть курса «Параллельные численные методы» и была ориентирована на изучение методов Монте-Карло, аспектов производительности и параллелизма в использовании генераторов случайных чисел, а также методики использования высокопроизводительных генераторов из библиотеки Intel MKL. В данной работе мы не будем углубляться в методы Монте-Карло (эта тема будет затронута в расширенной версии курса), и рассмотрим более простую задачу – вычисление справедливой цены опциона Европейского типа в том случае, когда эта цена может быть вычислена аналитически. Указанная задача является очень простой с точки зрения программирования (подставил все в формулу и вот он ответ), но позволяет показать, что даже в элементарных программах время работы на современной вычислительной техники может отличаться не просто в разы, а на порядки в зависимости от техники программирования – знания основных приемов оптимизации вычислений и навыков их использования.

В работе шаг за шагом иллюстрируется процесс оптимизации программы для обычных процессоров и ускорителей вычислений Intel Xeon Phi.

1. Методические указания

1.1. Цели и задачи работы

Цель данной работы – изучение некоторых принципов оптимизации вычислений в расчетных программах на примере решения задачи вычисления справедливой цены опциона Европейского типа.

Данная цель предполагает решение следующих основных задач:

1. Ознакомление с моделью финансового рынка и базовыми понятиями предметной области.
2. Подготовка базовой версии программы для вычисления цены опциона Европейского типа по формуле Блэка–Шоулса.
3. Пошаговая оптимизация и распараллеливание программы для Intel Xeon.
4. Пошаговая оптимизация и распараллеливание программы для Intel Xeon Phi.
5. Анализ результатов оптимизации и распараллеливания.

1.2. Структура работы

Работа построена следующим образом: вначале дается краткое описание модели финансового рынка, формулируются основные понятия, приводится их интуитивное описание, способствующее пониманию сути и смысла решаемой прикладной задачи. Далее описывается базовая программная реализация, проводится анализ производительности, шаг за шагом в программу вносятся изменения, иллюстрирующие приемы оптимизации: устранение ненужных преобразований типов, вынос инвариантов, эквивалентные преобразования с заменой «тяжелых» математических функций более «легкими», векторизация вычислений, распараллеливание, «прогрев» – снижение накладных расходов на создание потоков, понижение точности вычислений с плавающей запятой, оптимизация работы с памятью (использование streaming stores). Эффекты от оптимизации демонстрируются как на Intel Xeon, так и на Intel Xeon Phi.

1.3. Тестовая инфраструктура

Вычислительные эксперименты проводились с использованием следующей инфраструктуры (табл. 1).

Таблица 1. Тестовая инфраструктура

Процессор	2 восьмиядерных процессора Intel Xeon E5-2690 (2.9 GHz)
Сопроцессор	2 сопроцессора Intel Xeon Phi 7110X (61 ядро)
Память	64 GB
Операционная система	Linux CentOS 6.2
Компилятор, профилировщик	Intel Parallel Studio XE 2013 SP1

1.4. Рекомендации по проведению занятий

Для выполнения лабораторной работы рекомендуется следующая последовательность действий.

1. Кратко ввести основные понятия предметной области.
2. Сформулировать постановку задачи на лабораторную работу, объяснить цели и задачи работы.
3. Продемонстрировать базовую версию кода.
4. Пошагово выполнить предлагаемые оптимизации кода, показать эффект от их внедрения, обсудить особенности применения указанных приемов и возможный эффект.
5. Сравнить производительность на Xeon и Xeon Phi, предложить студентам самостоятельно проделать некоторые оптимизации (в зависимости от наличия свободного доступа к оборудованию, времени и целей, занятие может быть проведено как в режиме мастер-класса, так и в режиме лабораторной работы).

2. Оценивание опционов европейского типа

2.1. Модель финансового рынка

Рассмотрим эволюционирующий в непрерывном времени финансовый рынок, состоящий из двух типов активов – акции (рисковые активы, S) и облигации (безрисковые активы, B). Для моделирования рынка будем использовать широко распространенную модель Блэка – Шоулса [2]. Данная модель после ряда преобразований (см. [2]) представляет собой систему стохастических дифференциальных уравнений следующего вида:

$$dB_t = rB_t dt, \quad B_0 > 0 \tag{1}$$

$$dS_t = S_t\left((r-\delta)dt + \sigma dW_t\right), \quad S_0 > 0 \tag{2}$$

Уравнение (1) – обычное дифференциальное уравнение, описывает поведение цены облигации B_t, на изменение которой влияет *r – процентная ставка* (interest rate). Уравнение (2) – *стохастическое дифференциальное уравнение*, описывающее эволюцию цены акции S_t. Наряду с процентной ставкой r в уравнении присутствуют *ставка дивиденда* δ (dividend rate), *волатильность* σ и *Винеровский случайный процесс* $W = (W_t)_{\{t\geq 0\}}$. Начальные цены акции и облигации (S_0 и B_0 соответственно) считаются заданными.

Поясним кратко экономический смысл рассмотренной модели. Первое уравнение показывает возможности по вкладыванию средств в *безрисковые* активы – облигации. Представим, что мы идем в банк и кладем деньги на депозит под некоторый процент, определяемый банком в зависимости от инфляции и состояния дел в банке и на рынке в целом. Второе уравнение моделирует влияние на изменение цены акции (*рискового актива*) двух групп факторов – детерминированных и случайных. Первое слагаемое – $S_t(r-\delta)dt$ – похоже на правую часть в уравнении (1) с той разницей, что по акциям, в отличие от облигаций, могут платиться дивиденды, что отражается в ставке дивиденда δ. Наибольший интерес представляет слагаемое $S_t\sigma dW_t$, входящее в уравнение аддитивно с рассмотренной частью. Данное слагаемое моделирует влияние случайных, трудно прогнозируемых факторов, на рынок. Волатильность σ характеризует степень случайности процессов на рынке: $\sigma = 0$ – все детерминировано, риск отсутствует. Чем больше σ, тем больше риск (как возможная прибыль, так и возможные убытки). Множитель dW_t – дифференциал от Винеровского случайного процесса – описывает случайные факторы, влияющие на изменение цены акции в момент времени t. Приведем определение Винеровского случайного процесса – математической модели броуновского движения в непрерывном времени:

1. $W_0 = 0$ с вероятностью 1.
2. W_t – процесс с независимыми приращениями.
3. $W_t - W_s \sim N(0, t-s)$, где $s < t$, $N(0, t-s)$ – нормальное распределение с нулевым средним и дисперсией $t-s$.
4. Траектории процесса $W_t(\omega)$ – непрерывные функции времени с вероятностью 1.

Для последующей реализации необходимо сделать несколько важных замечаний.

1. В формулах и расчетах, приведенных далее, для упрощения будем считать, что $\delta = 0$ (модель без дивидендов).
2. Если время измеряется в годах, то все рассмотренные процентные ставки участвуют в уравнениях как числа от 0 до 1 – проценты годовых, выраженные в долях.
3. Будем считать, что все процентные ставки постоянны – не зависят от времени.
4. В рамках предположения о постоянстве процентных ставок система дифференциальных уравнений (1), (2) имеет аналитическое решение. В противном случае, систему приходится решать численно одним из известных методов (метод Эйлера, метод Рунге – Кутты и т.д.). При построении разностной схемы единственное отличие от ОДУ заключается в том, что каждое приращение винеровского случайного процесса моделируется случайным числом, полученным из $N(0, t - s)$.

Итак, при сделанных предположениях система имеет аналитическое решение. В частности, решение уравнения (2) выглядит следующим образом:

$$S_t = S_0 e^{\left(r - \frac{\sigma^2}{2}\right)t + \sigma W_t} \tag{3}$$

2.2. Понятие опциона и справедливой цены

Опцион – производный финансовый инструмент – контракт между сторонами P_1 и P_2, который дает право стороне P_2 в некоторый момент времени t в будущем купить у стороны P_1 или продать стороне P_1 акции по цене K, зафиксированной в контракте. За это право сторона P_2 выплачивает фиксированную сумму (премию) C стороне P_1. При этом K называется ценой исполнения опциона (*страйк,* strike price), а C – *ценой опциона*.

В данной работе рассматривается простейший вариант опциона – *колл-опцион европейского типа на акцию*. Основная идея заключения контракта состоит в игре двух лиц – P_1 и P_2. Вторая сторона выплачивает некоторую сумму C и в некоторый момент времени T (*срок выплаты*, maturity, зафиксирован в контракте) принимает решение: покупать акции по цене K у первой стороны или нет. Решение принимается в зависимости от соотношения цены S_T и K. Если $S_T < K$, покупать акции не выгодно, первая сторона получила прибыль C, а вторая – убыток C. В случае, если $S_T > K$, вторая сторона покупает у первой акции по цене K, в ряде случаев получая прибыль (в зависимости от соотношения между C и $S_T - K$).

Основная проблема заключается в расчете *справедливой цены* такого опционного контракта – цены, при которой наблюдается баланс выигрыша/проигрыша каждой из сторон. Логично определить такую цену как средний выигрыш стороны P_2:

$$C = E\left(e^{-rT}\left(S_T - K\right)^+\right) \tag{4}$$

В формуле (4) $\left(S_T - K\right)^+$ равно либо разности между ценой акции в момент T и ценой исполнения (если цена акции больше), либо нулю, E – математическое ожидание (S_T подвержена действию случайных факторов, смотри формулу (3)). Умножение на экспоненту соответствует т.н. *дисконтированию* – переводит 1 у.е. в момент времени $t = T$ к моменту времени $t = 0$, что соответствует инфляции с процентной ставкой r.

2.3. Метод вычисления справедливой цены опциона

Для вычисления справедливой цены опциона по формуле (4) известно аналитическое решение, которое справедливо при сделанных нами ранее предположениях.

Аналитическое решение описывается формулой Блэка–Шоулса для вычисления цены опциона в момент времени $t = 0$:

$$\begin{aligned} C &= S_0 F(d_1) - K e^{-rT} F(d_2) \\ d_1 &= \ln\frac{S_0}{K} + \frac{\left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}} \\ d_2 &= \ln\frac{S_0}{K} + \frac{\left(r - \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}} \end{aligned} \tag{5}$$

где F – функция стандартного нормального распределения.

Именно эту формулу мы и будем использовать в дальнейших расчетах. Что же здесь считать? – спросите вы. Действительно, на первый взгляд все выглядит элементарным. На деле не все так просто, что мы и увидим далее.

3. Оценивание набора опционов

Конечно, для вычисления цены одного опциона не нужна высокопроизводительная вычислительная техника. На практике организации, работающие на финансовых рынках, вычисляют цены гигантского количества разных

опционов, которые можно выпустить в конкретных рыночных условиях. Учитывая, что время финансовых расчетов существенно влияет на скорость принятия решений, каждая секунда на счету. Поэтому сокращение времени оценивания набора опционов является достаточно важной задачей.

Построим схему информационных зависимостей для одного опциона (рис. 1).

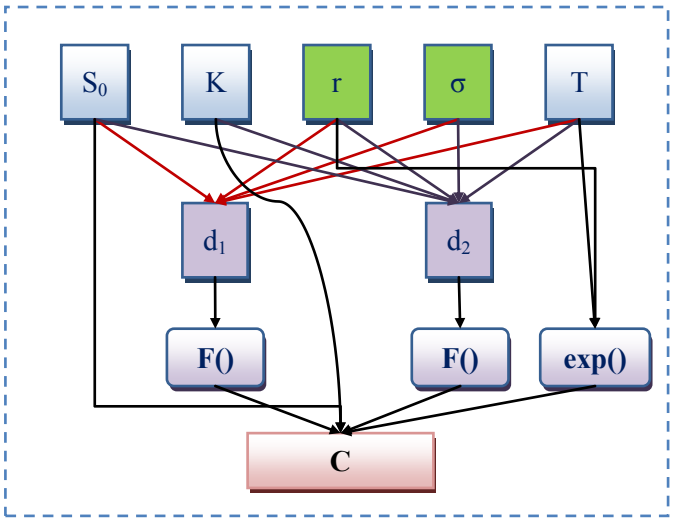

Рис. 1. – Схема информационных зависимостей

При построении набора опционов в общем случае можно разрешить варьировать все 5 параметров (начальная цена акции, страйк, процентная ставка, волатильность, срок исполнения). Тем не менее, мы учтем тот факт, что параметры рынка (процентная ставка и волатильность) в конкретный момент времени одинаковы для всех опционов. Поэтому в дальнейшем в расчетах мы будем считать, что разные опционы отличаются начальной ценой акции, страйком и сроком исполнения.

4. Программная реализация

4.1. Точка отсчета. Расчет по аналитической формуле

Программную реализацию начнем с написания функции, выполняющей расчет по рассмотренной в § 2.3 формуле.

Создадим пустой файл **main.cpp** и напишем в нем заготовку функции **`main()`**.

```
int numThreads = 1;
int N = 60000000;
```

```
int main(int argc, char *argv[])
{
  int version;

  if (argc < 2)
  {
    printf("Usage: <executable> size version
[#of_threads]\n");
    return 1;
  }
  N = atoi(argv[1]);
  version = atoi(argv[2]);
  if (argc > 3)
    numThreads = atoi(argv[3]);

  // Здесь будут вызовы функций для разных способов расчета

  return 0;
}
```

Здесь **`version`** – номер модификации, эксперименты с которой мы будем проводить в конкретный момент времени, **`N`** – число образцов, **`numThreads`** – число потоков в параллельной версии.

Следующим шагом напишем функцию **`GetOptionPrice()`** для расчета цены опциона по формуле (5). Отметим, что здесь и во всех остальных функциях мы используем одинарную точность, то есть тип **`float`**. В силу специфики данной задачи двойная точность в ней явно избыточна (например, в связи с тем, что входные данные изначально приходят с точностью, для которой типа float достаточно). Более того, далее мы увидим, что точность можно понизить дополнительно.

Функция **`GetOptionPrice()`** предваряется объявлением необходимых констант, используемых в расчете. Обращаем внимание на необходимость согласованного объявления данных – поскольку обычно процентная ставка указывается в годовом исчислении, то и время также должно быть в годах.

```
// Волатильность (% годовых 0.2 -> 20%)
const float sig = 0.2f;
// Процентная ставка (% годовых 0.05 -> 5%)
const float r   = 0.05f;
// Срок исполнения опциона (в годах)
const float T   = 3.0f;
// Цена акции в момент времени t=0 (у.е.)
const float S0  = 100.0f;
// Цена исполнения - цена покупки, зафикс. в опционе (у.е.)
const float K   = 100.0f;
```

```
float GetOptionPrice()
{
  float C;
  float d1, d2, p1, p2;

  d1 = (logf(S0 / K) + (r + sig * sig * 0.5f) * T) /
       (sig * sqrtf(T));
  d2 = (logf(S0 / K) + (r - sig * sig * 0.5f) * T) /
       (sig * sqrtf(T));
  p1 = cdfnormf(d1);
  p2 = cdfnormf(d2);
  C  = S0 * p1 - K * expf((-1.0f) * r * T) * p2;

  return C;
}
```

Нетривиальная часть формулы (5) – функция стандартного нормального распределения *F*. Для ее вычисления можно написать собственный довольно несложный код, но учитывая, что это задача не входит напрямую в данную работу, мы использовали готовое решение – функцию **`cdfnormf()`**.

Наконец, последнее, что осталось сделать в этой части лабораторной работы, – вызвать функцию **`GetOptionPrice()`** из функции **`main()`** с выводом результата на печать.

```
int main(int argc, char *argv[])
{
  int version;

  if (argc < 2)
  {
    printf("Usage: <executable> size version
[#of_threads]\n");
    return 1;
  }
  N = atoi(argv[1]);
  version = atoi(argv[2]);
  if (argc > 3)
    numThreads = atoi(argv[3]);

  float res = GetOptionPrice();
  printf("%.8f;\n", res);
  // Здесь будут вызовы функций для разных способов расчета

  return 0;
}
```

Для сборки разработанного кода используем следующую командную строку:

```
icc -O2 main.cpp -o option_prices
```

4.2. Выбор структуры хранения

Обсудим вопрос о хранении данных в нашей программе. Казалось бы, в чем проблема? Нужно завести четыре массива (три для хранения исходных данных и один для хранения результатов), а также несколько отдельных переменных. Дело в том, что порядок расположения данных в памяти существенно влияет на производительность программы. Так, во многих задачах, подобных той, что мы решаем, возникает дилемма: использовать набор массивов (паттерн SoA – structure of arrays) или массив структур (паттерн AoS – array of structures). В первом случае данные будут размещены в памяти так: сначала целиком разместится первый массив, потом целиком второй и т.д. Во втором случае сначала будут размещены все данные, относящиеся к первому объекту предметной области, потом – ко второму и т.д.

Вопрос о том, какой из подходов лучше, не имеет однозначного ответа. В некоторых случаях можно дать рекомендации. Так, например, интуитивно понятно, что в случае, если мы работаем с M массивами, из которых три используются в 95% случаев, а оставшиеся M–3 – в 5% случаев, имеет смысл отдельно хранить три массива и оставшиеся M–3 массива. Такой подход позволит локализовать в памяти те данные, с которыми мы чаще всего работаем.

В данной задаче слушателям предлагается выяснить, какой из подходов более эффективен, экспериментальным путем.

4.3. Нулевая (базовая) версия

Одна из типичных ошибок, которые допускают многие программисты на языке C (не только начинающие), – смешивание типов **`float`** и **`double`** при работе с числами с плавающей запятой. Оставляя за рамками обсуждения вопросы точности получаемого результата, рассмотрим, как это влияет на скорость работы, если данные представлены типом **`float`**, а используемые математические функции вызываются в форме, работающей с типом **`double`**.

В представленном выше коде функции **`GetOptionPrice()`** мы использовали, например, функцию **`logf()`**. Если программист вызовет вместо нее функцию **`log()`**, результат будет следующий. Функция **`log()`** в языке C принимает аргумент типа **`double`** и возвращает число типа **`double`**. Вследствие этого, во-первых, элемент массива **`pT[i]`** будет преобразован из типа **`float`** в тип **`double`** и, во-вторых, далее все вычисления будут

вестись с типом **double** (с преобразованием типов при необходимости) и лишь в самом конце при присваивании результата в элемент массива **p1[0]** будет выполнено обратное преобразование в тип **float**. Учитывая, что работа с числами типа **float** во многих случаях происходит быстрее, чем с типом **double** (в особенности заметна эта разница становится при использовании векторизации), подобные преобразования сказываются на скорости работы кода отрицательно. Давайте посмотрим, насколько велика будет эта разница в нашем случае.

Добавьте в файл **main.cpp** следующую функцию.

```
__declspec(noinline) void GetOptionPricesV0(
  float *pT, float *pK, float *pS0, float *pC)
{
  int i;
  float d1, d2, p1, p2;

  for (i = 0; i < N; i++)
  {
    d1 = (log(pS0[i] / pK[i]) + (r + sig * sig * 0.5) *
          pT[i]) / (sig * sqrt(pT[i]));
    d2 = (log(pS0[i] / pK[i]) + (r - sig * sig * 0.5) *
          pT[i]) / (sig * sqrt(pT[i]));
    p1 = cdfnormf(d1);
    p2 = cdfnormf(d2);
    pC[i] = pS0[i] * p1 - pK[i] *
            exp((-1.0) * r * pT[i]) * p2;
  }
}
```

Теперь модифицируем функцию **main()**. Добавим тип данных – указатель на функцию с прототипом, соответствующим функции **GetOption-PricesV0()**, объявим массив указателей **GetOptionPrices**, и по номеру модификации будем вызывать требуемую функцию.

```
#include "omp.h"

...
// Начало и конец счета
double start, finish;
// Время вычислений
double t;

typedef void (*tGetOptionPrices)(float *pT, float *pK,
  float *pS0, float *pC);
tGetOptionPrices GetOptionPrices[9] =
{ GetOptionPricesV0 };

int main(int argc, char *argv[])
{
```

```
  int i, version;

  if (argc < 2)
  {
    printf("Usage: <executable> size version
[#of_threads]\n");
    return 1;
  }
  N = atoi(argv[1]);
  version = atoi(argv[2]);
  if (argc > 3)
    numThreads = atoi(argv[3]);

  pT  = new float[4 * N];
  pK  = pT + N;
  pS0 = pT + 2 * N;
  pC  = pT + 3 * N;
  for (i = 0; i < N; i++)
  {
    pT[i] = T;
    pS0[i] = S0;
    pK[i] = K;
  }

  float res = GetOptionPrice();
  printf("%.8f;\n", res);

  omp_set_num_threads(numThreads);
  start = omp_get_wtime();
  GetOptionPrices[version](pT, pK, pS0, pC);
  finish = omp_get_wtime();
  t = finish - start;
  printf("v%d: %.8f; %lf\n", version, pC[0], t);

  delete [] pT;
  return 0;
}
```

Отметим, что для удобства измерение времени выполнено с использованием функции OpenMP (учитывая, что в дальнейшем именно с помощью этой технологии мы будем разрабатывать параллельную версию).

Также отметим, что память под массивы, хранящие данные, мы выделяем одной операцией, настраивая далее указатели **pT**, **pK**, **pS0** и **pC**.

Для корректной сборки программы в командную строку нужно добавить ключ **-openmp**:

```
icc -O2 -openmp ...
```

Соберите программу, проведите эксперименты, увеличивая число образцов вдвое от 60 000 000 до 240 000 000, и занесите их в таблицу.

На описанной ранее инфраструктуре авторы получили следующие времена.

Таблица 1. Время работы базовой версии (в секундах)

N	60 000 000	120 000 000	180 000 000	240 000 000
Версия V0	17,002	34,004	51,008	67,970

4.4. Версия 1. Исключение ненужных преобразований типов

Теперь приведем код предыдущей версии в порядок, вызывая правильные функций **`logf()`**, **`sqrtf()`** и **`expf()`** а также правильно указывая константы в коде (все знают, что число **1.0** без суффикса **f** будет рассматриваться как число типа double?).

```
__declspec(noinline) void GetOptionPricesV1(float *pT,
  float *pK, float *pS0, float *pC)
{
  int i;
  float d1, d2, p1, p2;

  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]));
    p1 = cdfnormf(d1);
    p2 = cdfnormf(d2);
    pC[i] = pS0[i] * p1 - pK[i] *
            expf((-1.0f) * r * pT[i]) * p2;
  }
}
```

Добавьте в массив указателей **`GetOptionPrices`**, новую функцию.

```
tGetOptionPrices GetOptionPrices[9] =
{ GetOptionPricesV0, GetOptionPricesV1 };
```

Соберите программу и проведите эксперименты.

На описанной ранее инфраструктуре авторы получили следующие времена.

Таблица 2. Время работы версий 0 и 1 (в секундах)

N	60 000 000	120 000 000	180 000 000	240 000 000
Версия V0	17,002	34,004	51,008	67,970

Версия V1	16,776	33,549	50,337	66,989

Как видно из табл. 2, время работы версии 1 в сравнении с базовой изменилось очень мало. Причина такого результата в функции **`cdfnormf()`**, точнее в ее крайне медленной работе, а также в том, что время работы **`cdfnormf()`** и **`cdfnorm()`** не отличается. На ее фоне выигрыш от исключения ненужных преобразований типов практически не заметен. В других случаях разница может быть очень значительной. Так, авторы данной работы сталкивались с ситуацией, когда разница времени достигала трех раз (!).

4.5. Версия 2. Эквивалентные преобразования: cdfnorm() vs. erf()

Вычисления, которые мы до сих пор выполняли с помощью функции **`cdfnormf()`** можно проделать и другим способом, используя функцию **`erff()`**. Дело в том, что вычисление **`cdfnormf`** является более трудоемким (подумайте, почему это так, рассмотрите интегралы, которые необходимо посчитать).

Используем формулу $cdfnorm(x) = 0.5 + 0.5erf\left(\frac{x}{\sigma\sqrt{T}}\right)$.

Выполнив необходимые математические преобразования, получим следующий код.

```
__declspec(noinline) void GetOptionPricesV2(float *pT,
  float *pK, float *pS0, float *pC)
{
  int i;
  float 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;
  }
}
```

Добавьте в массив указателей **`GetOptionPrices`**, новую функцию.

```
tGetOptionPrices GetOptionPrices[9] =
{ GetOptionPricesV0, GetOptionPricesV1, GetOptionPricesV2
};
```

Соберите программу и проведите эксперименты.

На описанной ранее инфраструктуре авторы получили следующие времена.

Таблица 3. Время работы версий с 0 до 2 (в секундах)

N	60 000 000	120 000 000	180 000 000	240 000 000
Версия V0	17,002	34,004	51,008	67,970
Версия V1	16,776	33,549	50,337	66,989
Версия V2	2,871	5,727	8,649	11,230

Как видно из табл. 3, время работы версии 2 почти в 6 раз меньше, чем версии 1. В дальнейших модификациях будем использовать именно функцию **`erff()`**.

4.6. Версия 3. Векторизация: использование ключевого слова restrict

Возможности алгоритмической оптимизации на этом в основном исчерпаны. Перейдем к векторизации расчетного цикла. Прежде всего, необходимо сменить режим компиляции, указав в командной строке ключ **–mavx**, в противном случае компилятор будет использовать наборы команд SSE, но не AVX.

Проведя эксперименты с собранной версией, можно убедиться, что время работы функции **`GetOptionPricesV2()`** не изменится. Чтобы понять, в чем дело, укажем при сборке ключ **-vec-report3**, «попросив» компилятор выдать отчет о векторизации. Цифра «**3**» соответствует степени подробности отчета, соответствующей выдаче информации о векторизованных/невекторизованных циклах, а также причинах отсутствия векторизации. Заметим, что в последних версиях компилятора появился новый, шестой тип отчета (-vec-report6), дополнительно дающий некоторые рекомендации относительно векторизации.

Отчет о векторизации приведен ниже (рис. 2). К сожалению, в данном случае он не слишком информативен. Заметим, что при компиляции данного кода при помощи Intel C/C++ Compiler for Windows в отчете можно увидеть фразу о том, что возможны зависимости между массивами. В целом, наш опыт говорит о том, что диагностики компиляторов Intel для C/C++ чаще всего позволяют догадаться, что именно мешает векторизации.

Вернемся к вопросу о том, как «уговорить» компилятор векторизовать цикл.

```
sh-4.1$ icc -O2 -openmp -mavx -vec-report3 main.cpp -o option_prices
main.cpp(279): (col. 3) remark: loop was not vectorized: nonstandard loop is not
 a vectorization candidate
main.cpp(99): (col. 3) remark: loop was not vectorized: nonstandard loop is not
a vectorization candidate
main.cpp(115): (col. 3) remark: loop was not vectorized: nonstandard loop is not
 a vectorization candidate
main.cpp(131): (col. 3) remark: loop was not vectorized: nonstandard loop is not
 a vectorization candidate
sh-4.1$
```

Рис. 2. Отчет компилятора о векторизации

Объяснить компилятору, что опасности нет, можно разными способами: используя ключевое слово **restrict** в объявлении формальных параметров функции, с помощью директивы **#pragma ivdep**, с помощью директивы **#pragma simd** перед циклом. Кроме перечисленного, можно указать специальную опцию, которая подскажет компилятору, что массивы в программе никогда не пересекаются, но эта опция действует на всю программу целиком и при неаккуратном использовании может «сломать» код.

Рассмотрим сначала первый вариант. Ключевое слово **restrict** в объявлении указателей говорит о том, что в данную область памяти не будут обращаться при помощи других указателей. Это дает компилятору информацию о том, что массивы **pT**, **pK**, **pS0** и **pC** не пересекаются, а, следовательно, запись **pC[i]** не оказывает никакого влияния на значения в других массивах, то есть можно безопасно менять порядок вычислений, организуя их векторно.

```
__declspec(noinline) void GetOptionPricesV3(
  float * restrict pT, float * restrict pK,
  float * restrict pS0, float * restrict pC)
{
  int i;
  float 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;
  }
}
```

Добавьте в массив указателей **GetOptionPrices**, новую функцию.

```
tGetOptionPrices GetOptionPrices[9] =
{ GetOptionPricesV0, GetOptionPricesV1, GetOptionPricesV2,
  GetOptionPricesV3 };
```

Для корректной сборки программы, в которой используется ключевое слово **restrict**, необходимо указать в командной строке одноименную опцию.

```
icc -O2 -restrict -mavx ...
```

Соберите программу и проведите эксперименты.

На описанной ранее инфраструктуре авторы получили следующие времена.

Таблица 4. Время работы версий с 0 до 3 (в секундах)

N	60 000 000	120 000 000	180 000 000	240 000 000
Версия V0	17,002	34,004	51,008	67,970
Версия V1	16,776	33,549	50,337	66,989
Версия V2	2,871	5,727	8,649	11,230
Версия V3	0,522	1,049	1,583	2,091

Как видно из табл. 4, время работы версии 3 примерно в 5.5 раз меньше, чем версии 2. Процессор, на котором проводились эксперименты, способен с использованием AVX выполнять операции одновременно над 8 элементами типа **float**. Почему код ускорился в меньшее число раз, чем должен, предлагаем выяснить читателям самостоятельно, собрав профиль работы функции, использую Intel VTune Amplifier XE (обратите внимание на вызовы математических функций и примените закон Амдаля).

4.7. Версия 4. Векторизация: использование директивы simd

Второй способ векторизации связан с использованием перед циклом директивы **#pragma ivdep** (ignore vector dependencies). Она подсказывает компилятору, что с нашей точки зрения массивы в цикле не пересекаются. Заметим, что в случае, когда мы ошиблись, компилятор иногда может установить наличие зависимости и проигнорировать нашу подсказку, но чаще всего мы получим некорректно работающий код. Данной директивой надо пользоваться с осторожностью. Заметим, что эта директива иногда используется в паре с **#pragma vector always** – подсказкой компилятору о том, что с нашей точки зрения векторизация, если она возможна, будет эффективна. Как мы упоминали ранее в лекции и практике по векторизации, иногда векторизация не приведет к ускорению, более того, может возникнуть замедление. Как правило, это происходит в тех ситуациях, когда данные лежат в памяти не в том порядке, в котором их нужно запаковывать в векторные регистры. В таких случаях накладные расходы на подготовку данных и запись результатов могут превысить выигрыш от векторного выполнения операций. Однако компилятор не всегда прав. Иногда он считает

векторизацию неэффективной, но мы хотим попробовать. Для этого и нужна **#pragma vector always**.

В настоящий момент основным становится третий способ подсказки компилятору – использование новой директивы **#pragma simd**. Возможности компилятора по векторизации цикла при использовании данной директивы значительно выше, кроме того, существует целый ряд настроек, которых не было ранее. Заметим, что ответственность за корректность результата в этом случае полностью ложится на программиста. Использовать данную директиву нужно с осторожностью.

```
__declspec(noinline) void GetOptionPricesV4(float *pT,
  float *pK, float *pS0, float *pC)
{
  int i;
  float d1, d2, erf1, erf2;

#pragma simd
  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;
  }
}
```

Добавьте в массив указателей **GetOptionPrices**, новую функцию.

```
tGetOptionPrices GetOptionPrices[9] =
{ GetOptionPricesV0, GetOptionPricesV1, GetOptionPricesV2,
  GetOptionPricesV3, GetOptionPricesV4 };
```

Соберите программу и проведите эксперименты.

На описанной ранее инфраструктуре авторы получили следующие времена.

Таблица 5. Время работы версий с 0 до 4 (в секундах)

N	60 000 000	120 000 000	180 000 000	240 000 000
Версия V0	17,002	34,004	51,008	67,970
Версия V1	16,776	33,549	50,337	66,989
Версия V2	2,871	5,727	8,649	11,230
Версия V3	0,522	1,049	1,583	2,091
Версия V4	0,521	1,036	1,566	2,067

Как видно из табл. 5, время работы версии 4 почти не отличается от времени версии 3. Впрочем, этого и следовало ожидать.

4.8. Версия 5. Вынос инвариантов из цикла

Попытаться еще немного ускорить работу функции **`GetOptionPricesV4()`** можно, сэкономив не тех вычислениях, которые можно вынести за цикл. В данном случае это расчет **`1.0f / sqrtf(2.0f)`**.

Вычислим это выражение отдельно и внесем в код константу вида

```
const float invsqrt2 = 0.707106781f;
```

Теперь используем ее в теле функции, заменив попутно деление на умножение.

```
  declspec(noinline) void GetOptionPricesV5(float *pT,
  float *pK, float *pS0, float *pC)
{
  int i;
  float d1, d2, erf1, erf2;

#pragma simd
  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 * invsqrt2);
    erf2 = 0.5f + 0.5f * erff(d2 * invsqrt2);
    pC[i] = pS0[i] * erf1 - pK[i] * expf((-1.0f) * r *
             pT[i]) * erf2;
  }
}
```

Добавьте в массив указателей **`GetOptionPrices`**, новую функцию.

```
tGetOptionPrices GetOptionPrices[9] =
{ GetOptionPricesV0, GetOptionPricesV1, GetOptionPricesV2,
  GetOptionPricesV3, GetOptionPricesV4, GetOptionPricesV5
};
```

Соберите программу и проведите эксперименты.

На описанной ранее инфраструктуре авторы получили следующие времена.

Таблица 6. Время работы версий с 0 до 5 (в секундах)

N	60 000 000	120 000 000	180 000 000	240 000 000

Версия V0	17,002	34,004	51,008	67,970
Версия V1	16,776	33,549	50,337	66,989
Версия V2	2,871	5,727	8,649	11,230
Версия V3	0,522	1,049	1,583	2,091
Версия V4	0,521	1,036	1,566	2,067
Версия V5	0,527	1,047	1,580	2,085

Как видно из табл. 6, время работы версии 5 практически совпадает с временами версий 3 и 4. В данном случае, видимо, компилятор проделал необходимые преобразования кода самостоятельно. Убедиться в этом можно, изучив ассемблерный листинг программы после сборки (укажите в командной строке ключ **-Fa**).

Заметим, что хотя в данном случае мы ничего не выиграли, вынос инвариантов – полезная техника оптимизации, которая, как правило, делает код более понятным и логичным, а также нередко приводит к выигрышу производительности (компилятор далеко не всегда может осуществить подобные преобразования сам).

4.9. Версия 6. Эквивалентные преобразования. Вычисление квадратного корня

Обратим внимание на еще один момент. Замена деления умножением может привести к существенному выигрышу производительности. Еще одна возможная оптимизация – замена выражения вида **`1/sqrtf()`** на вызов функции **`invsqrtf()`**. Проверим, проделал ли компилятор это преобразование самостоятельно.

```
__declspec(noinline) void GetOptionPricesV6(float *pT,
  float *pK, float *pS0, float *pC)
{
  int i;
  float d1, d2, erf1, erf2, invf;
  float sig2 = sig * sig;

#pragma simd
  for (i = 0; i < N; i++)
  {
    invf = invsqrtf(sig2 * pT[i]);
    d1 = (logf(pS0[i] / pK[i]) + (r + sig2 * 0.5f) *
         pT[i]) * invf;
    d2 = (logf(pS0[i] / pK[i]) + (r - sig2 * 0.5f) *
         pT[i]) * invf;
    erf1 = 0.5f + 0.5f * erff(d1 * invsqrt2);
    erf2 = 0.5f + 0.5f * erff(d2 * invsqrt2);
```

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

Добавьте в массив указателей **GetOptionPrices**, новую функцию.

```
tGetOptionPrices GetOptionPrices[9] =
{ GetOptionPricesV0, GetOptionPricesV1, GetOptionPricesV2,
  GetOptionPricesV3, GetOptionPricesV4, GetOptionPricesV5,
  GetOptionPricesV6 };
```

Соберите программу и проведите эксперименты.

На описанной ранее инфраструктуре авторы получили следующие времена.

Таблица 7. Время работы версий с 0 до 6 (в секундах)

N	60 000 000	120 000 000	180 000 000	240 000 000
Версия V0	17,002	34,004	51,008	67,970
Версия V1	16,776	33,549	50,337	66,989
Версия V2	2,871	5,727	8,649	11,230
Версия V3	0,522	1,049	1,583	2,091
Версия V4	0,521	1,036	1,566	2,067
Версия V5	0,527	1,047	1,580	2,085
Версия V6	0,538	1,071	1,614	2,133

Как видно из табл. 7, время работы версии 6 отличается от версий с 3 по 5 в пределах погрешности измерений. Тем не менее, в любом случае проделанные нами оптимизации небесполезны и в случае менее «интеллектуального» компилятора дали бы эффект. Таким образом, именно эту версию мы возьмем за базу в дальнейших экспериментах, включая запуски на сопроцессоре Xeon Phi.

4.10. Версия 6.1. Выравнивание данных

Еще одна потенциально полезная программная оптимизация связана с выравниванием обрабатываемых данных, поскольку невыровненные данные обрабатываются процессором медленнее. С большой вероятностью либо с этим справится компилятор, либо ущерб будет не так велик. Тем не менее, давайте убедимся. Для гарантированного выравнивания заменим операторы выделения/освобождения памяти (**new**/**delete**) на вызовы функций

memalign()[1]/**free()**. Остальной код не изменится, так же как и ключи сборки.

```
int main(int argc, char *argv[])
{
  pT = (float *)memalign(32, 4 * N * sizeof(float));
//  pT  = new float[4 * N];

  ...

  free(pT);
//  delete [] pT;
  return 0;
}
```

Рекомендуемая величина выравнивания (первый параметр в функции **memalign()**) зависит от длины используемых регистров. Для векторного расширения SSE – это 16, для AVX – 32, и для сопроцессора Xeon Phi – 64.

Кроме того, полезно добавить в код перед циклом еще одну директиву:

```
#pragma vector aligned
```

Эта директива подскажет компилятору, что данные, используемые в цикле, выровнены и могут быть использованы соответствующие команды для работы с памятью. Заметим, что если мы «обманем» компилятор, программа будет аварийно завершаться при попытке доступа к невыровненным данным.

Соберите программу и проведите эксперименты. На описанной ранее инфраструктуре авторы получили следующие времена.

Таблица 8. Время работы версий с 0 до 6.1 (в секундах)

N	60 000 000	120 000 000	180 000 000	240 000 000
Версия V0	17,002	34,004	51,008	67,970
Версия V1	16,776	33,549	50,337	66,989
Версия V2	2,871	5,727	8,649	11,230
Версия V3	0,522	1,049	1,583	2,091
Версия V4	0,521	1,036	1,566	2,067
Версия V5	0,527	1,047	1,580	2,085
Версия V6	0,538	1,071	1,614	2,133
Версия V6.1	0,539	1,072	1,617	2,135

[1] Обратите внимание, что функция memalign() не доступна под ОС Windows. Вы можете использовать __mm_malloc() и __mm_free().

Как видно из табл. 8, время работы версии 6.1 фактически совпадает с временем работы версии 6. В данном случае эта техника не дала результата, но это не означает, что так будет всегда.

4.11. Версия 6.2. Пониженная точность

В п. 4.1 мы указали, что использовать в рассматриваемой задаче тип **`double`** нет необходимости. Сейчас мы пойдем еще дальше. В данной задаче точность типа **`float`** также избыточна (вряд ли нам могут понадобится больше 4 знаков после запятой для вычисленной цены опциона) и ее можно еще понизить, ускорив тем самым вычисления. Добиться этого несложно, использую ключи компилятора при сборке программы.

Добавьте в командную строку опции, влияющие на точность вычисления значений математических функций.

```
icc ... -fimf-precision=low -fimf-domain-exclusion=31
```

Соберите программу и проведите эксперименты.

На описанной ранее инфраструктуре авторы получили следующие времена.

Таблица 9. Время работы версий с 0 до 6.2 (в секундах)

N	60 000 000	120 000 000	180 000 000	240 000 000
Версия V0	17,002	34,004	51,008	67,970
Версия V1	16,776	33,549	50,337	66,989
Версия V2	2,871	5,727	8,649	11,230
Версия V3	0,522	1,049	1,583	2,091
Версия V4	0,521	1,036	1,566	2,067
Версия V5	0,527	1,047	1,580	2,085
Версия V6	0,538	1,071	1,614	2,133
Версия V6.1	0,539	1,072	1,617	2,135
Версия V6.2	0,438	0,871	1,314	1,724

Как видно из табл. 9, время работы версии 6.2 примерно на 23% меньше, чем версии 6.1.

4.12. Версия 7. Распараллеливание

Распараллеливание вычислительного цикла в функции **`GetOption-PricesV6()`** не представляет никакого труда в силу отсутствия каких-либо зависимостей между итерациями. Достаточно указать перед ним прагму **omp parallel for** и локализовать все переменные, в которые происходит запись.

```
__declspec(noinline) void GetOptionPricesV7(float *pT,
```

```
  float *pK, float *pS0, float *pC)
{
  int i;
  float d1, d2, erf1, erf2, invf;
  float sig2 = sig * sig;

#pragma simd
#pragma omp parallel for private(invf, d1, d2, erf1, erf2)
  for (i = 0; i < N; i++)
  {
    invf = invsqrtf(sig2 * pT[i]);
    d1 = (logf(pS0[i] / pK[i]) + (r + sig2 * 0.5f) *
         pT[i]) * invf;
    d2 = (logf(pS0[i] / pK[i]) + (r - sig2 * 0.5f) *
         pT[i]) * invf;
    erf1 = 0.5f + 0.5f * erff(d1 * invsqrt2);
    erf2 = 0.5f + 0.5f * erff(d2 * invsqrt2);
    pC[i] = pS0[i] * erf1 - pK[i] * expf((-1.0f) * r *
            pT[i]) * erf2;
  }
}
```

Добавьте в массив указателей **GetOptionPrices**, новую функцию.

```
tGetOptionPrices GetOptionPrices[9] =
{ GetOptionPricesV0, GetOptionPricesV1, GetOptionPricesV2,
  GetOptionPricesV3, GetOptionPricesV4, GetOptionPricesV5,
  GetOptionPricesV6, GetOptionPricesV7 };
```

Учитывая, что ключ **-openmp** уже был нами поставлен в командную строку при сборке ранее, изменений в ключах сборки не понадобится.

Соберите программу и проведите эксперименты.

На описанной ранее инфраструктуре авторы получили следующие времена.

Таблица 10. Время работы версий с 0 до 7 (в секундах), 16 ядер

N	60 000 000	120 000 000	180 000 000	240 000 000
Версия V0	17,002	34,004	51,008	67,970
Версия V1	16,776	33,549	50,337	66,989
Версия V2	2,871	5,727	8,649	11,230
Версия V3	0,522	1,049	1,583	2,091
Версия V4	0,521	1,036	1,566	2,067
Версия V5	0,527	1,047	1,580	2,085
Версия V6	0,538	1,071	1,614	2,133
Версия V6.1	0,539	1,072	1,617	2,135
Версия V6.2	0,438	0,871	1,314	1,724

Версия V7	0,058	0,084	0,126	0,153

Как видно из табл. 10, ускорение растет с ростом объема данных, от 7,59 до 11.27 на 240 млн. образцов.

4.13. Версия 7.1. Прогрев

Времена в последнем эксперименте стали достаточно малы, поэтому накладные расходы, связанные с созданием потоков, могут вносить существенный вклад в общее время работы параллельного кода. Попробуем от них избавиться. Подход основан на том факте, что большинство реализаций стандарта OpenMP не уничтожают потоки, созданные в начале некоторой параллельной секции, по ее окончании, а переводят их в состояние сна, вывод из которого занимает существенно меньше времени. Соответственно, если вызвать функцию **`GetOptionPricesV7()`** дважды подряд в функции **`main()`**, то во втором вызове накладные расходы будут минимальны. В некоторых источниках такой подход называется «прогрев».

Обсудим, является ли честным такой способ оценки производительности. Что касается исключения накладных расходов на создание потоков, сомнений нет. Дело в том, что в реальных программах мы всегда можем организовать однократное создание потоков, в результате чего все последующие параллельные расчеты в рамках конкретной программы будут лишены соответствующих накладных расходов. Однако есть еще один важный момент. Дополнительное уменьшение времени может быть получено за счет так называемого «прогрева» кэша. При повторном вызове функции **`GetOptionPricesV7()`** часть необходимых ей данных может остаться в кэше, что может ускорить ее работу. Хорошо это или плохо? С одной стороны – плохо или, скорее, не совсем честно. Известно, что типичная ошибка – проведение в рамках одного процесса экспериментов с однопоточной, двухпоточной и четырехпоточной версиями программы, работающими с одними и теми же данными. В результате обычно получается сверхлинейное ускорение, которое немедленно исчезает, когда эксперименты ставятся корректно – каждый в своем процессе. С другой стороны, в реальных программах выполняется много расчетов, и данные постепенно подгружаются в кэш.

В целом, наша рекомендация выглядит следующим образом:

1. В случае если речь идет о больших временах, измеряющихся хотя бы в секундах, «прогрев» использовать не нужно.

2. В случае если времена очень малы (доли секунды), а количество потоков велико, имеет смысл использовать «прогрев», в противном случае мы будем измерять не время работы алгоритма, а объем накладных расходов. При этом если мы считаем, что дополнительный «прогрев» кэша в нашей

задаче будет смещать результат, изменяя истинное положение вещей, мы можем реализовать специальный параллельный цикл, который ничего полезного не делает, но приведет к созданию потоков. При этом данные заранее подгружаться в кэш не будут.

Проведем эксперименты с функциями **`GetOptionPricesV6()`** и **`GetOptionPricesV7()`**.

Модифицируем функцию **`main()`** следующим образом.

```
int main(int argc, char *argv[])
{
  ...
  start = omp_get_wtime();
  GetOptionPrices[version](pT, pK, pS0, pC);
  finish = omp_get_wtime();
  t = finish - start;
  printf("v%d: %.8f; %lf\n", version, pC[0], t);

  start = omp_get_wtime();
  GetOptionPrices[version](pT, pK, pS0, pC);
  finish = omp_get_wtime();
  t = finish - start;
  printf("    %.8f; %lf\n", pC[0], t);
  ...
  return 0;
}
```

Соберите программу и проведите эксперименты.

На описанной ранее инфраструктуре авторы получили следующие времена.

Таблица 11. Время работы версий с 0 до 7.1 (в секундах), 16 ядер

N	60 000 000	120 000 000	180 000 000	240 000 000
Версия V0	17,002	34,004	51,008	67,970
Версия V1	16,776	33,549	50,337	66,989
Версия V2	2,871	5,727	8,649	11,230
Версия V3	0,522	1,049	1,583	2,091
Версия V4	0,521	1,036	1,566	2,067
Версия V5	0,527	1,047	1,580	2,085
Версия V6	0,538	1,071	1,614	2,133
Версия V6.1	0,539	1,072	1,617	2,135
Версия V6.2	0,438	0,871	1,314	1,724
Версия V7	0,058	0,084	0,126	0,153
Версия V6.3	0,409	0,812	1,226	1,603
Версия V7.1	0,033	0,062	0,091	0,118

Как видно из табл. 11, ускорение версии 7.1 по отношению к версии 6.3 существенно выше, чем было у версии 7 по отношению к 6.2, и составляет от 12.54 (60 млн. образцов) до 13,61 (240 млн. образцов). Также видно, что версия 6.3 примерно на 7,5% быстрее, чем версия 6.2, а версия 7.1 – на треть быстрее, чем версия 7 (кроме эксперимента с 60 млн. образцов, где разница составляет более 70%, что неудивительно для такого малого объема вычислений).

4.14. Эксперименты на сопроцессоре Xeon Phi

Мощь сопроцессора Xeon Phi проявляется при использовании большого числа потоков (от 60 до 240), при этом каждое отдельное ядро существенно уступает в производительности ядрам процессора Xeon. Таким образом, сравнение имеет смысл проводить, начиная с версии 7. Однако, чтобы картина была полной, рекомендуем читателям провести эксперименты и с начальными версиями. В частности, можно увидеть, какие результаты дает векторизация кода при переходе от версии 2 к версии 3.

Здесь же мы приведем результаты экспериментов, начиная с версии 6, взятой нами за базу для процессора Xeon.

Для сборки программы под сопроцессор Xeon Phi добавьте в командную строку компилятора ключ **`-mmic`**.

Также не забудьте увеличить величину выравнивания в функции **`memalign()`** с 32 до 64.

Соберите программу и проведите эксперименты.

На описанной ранее инфраструктуре авторы получили следующие времена.

Таблица 12. Время работы на сопроцессоре Xeon Phi версий с 6 до 6.3 (в секундах)

N	60 000 000	120 000 000	180 000 000	240 000 000
Версия V6	1,544	3,089	4,633	6,174
Версия V6.1	1,545	3,091	4,634	6,179
Версия V6.2	0,676	1,352	2,027	2,703
Версия V6.3	0,422	0,845	1,269	1,690

Как видно из таблицы 12 на сопроцессоре Xeon Phi переход к вычислениям с пониженной точностью (версия 6.2) дает выигрыш не на 23%, как это было на процессоре, а почти в 2,3 раза. Также видно, что более значительный прирост (порядка 60%) дает прогрев, что неудивительно, принимая во внимание тот факт, что мы создаем на порядок больше потоков. Наконец,

можно отметить, что времена работы версии 6.3. на сопроцессоре практически сравнялись с временами работы на процессоре. Теперь посмотрим на результаты работы параллельных версий.

Таблица 13. Время работы на сопроцессоре Xeon Phi версий с 7 до 7.1 (в секундах), 60 поток

N	60 000 000	120 000 000	180 000 000	240 000 000
Версия V7	0,134	0,149	0,164	0,175
S(V6.2/V7)	5,0336	9,050	12,331	15,437
Версия V7.1	0,008	0,017	0,025	0,033
S(V6.3/V7.1)	50,585	51,178	51,783	51,546

Таблица 14. Время работы на сопроцессоре Xeon Phi версий с 7 до 7.1 (в секундах), 120 потоков

N	60 000 000	120 000 000	180 000 000	240 000 000
Версия V7	0,234	0,255	0,257	0,255
S(V6.2/V7)	2,885	5,303	7,883	10,590
Версия V7.1	0,007	0,014	0,021	0,028
S(V6.3/V7.1)	59,422	59,587	60,389	59,839

Таблица 15. Время работы на сопроцессоре Xeon Phi версий с 7 до 7.1 (в секундах), 240 потоков

N	60 000 000	120 000 000	180 000 000	240 000 000
Версия V7	0,532	0,527	0,533	0,558
S(V6.2/V7)	1,269	2,564	3,800	4,842
Версия V7.1	0,008	0,016	0,024	0,031
S(V6.3/V7.1)	53,286	54,248	53,969	53,964

Как видно из табл. 13-15, версия 7 ускоряется весьма плохо. Как и для центрального процессора, накладные расходы на работу с потоками оказываются сопоставимыми с общим временем работы программы. В то же время за вычетом расходов на работу с потоками (версия 7.1) ускорение получается вполне неплохим (от 50,5 до 60,4). Также видим, что при запуске в 120 потоков ускорение выше, чем при запуске в 60 потоков, что согласуется с техническими особенностями исполнения кода на Xeon Phi.

Однако результаты на Xeon Phi можно улучшить.

4.15. Версия 8. Оптимизация работы с кэш-памятью

В коде функций **`GetOptionPricesVX()`** мы работаем с 4-мя массивами (**`pT`**, **`pK`**, **`PS0`**, **`pC`**). При этом 3 из них используются только для чтения, а один (**`pC`**) для записи. Обратим внимание на тот факт, что значения, кото-

рые мы записываем в длинный массив **pC,** в цикле никак не используются. Таким образом, их кэширование вряд ли имеет смысл (массив **pC** относится к nontemporal data). Для записи напрямую в память, минуя кэш, результатов вычислений, идентифицированных нами как nontemporal data, предназначены так называемые streaming stores, использование которых приводит к уменьшению накладных расходов. В данном конкретном примере мы видим, что цикл имеет огромное число итераций, в нем выполняется сравнительно мало арифметических операций относительно операций с памятью, на Xeon Phi одновременно работают сотни потоков. Совокупность этих факторов приводит к тому, что мы прокачиваем через шину объем данных, сравнимый, а то и превосходящий ее пропускную способность. Использование streaming stores для массива **pC** позволит нам сэкономить по одной операции чтения, нужной для поддержания системы кэш-памяти в согласованном состоянии, на каждой итерации. Проверим этот факт.

```
__declspec(noinline) void GetOptionPricesV8(float *pT,
  float *pK, float *pS0, float *pC)
{
  int i;
  float d1, d2, erf1, erf2, invf;
  float sig2 = sig * sig;

#pragma simd
#pragma vector nontemporal
#pragma omp parallel for private(invf, d1, d2, erf1, erf2)
  for (i = 0; i < N; i++)
  {
    invf = invsqrtf(sig2 * pT[i]);
    d1 = (logf(pS0[i] / pK[i]) + (r + sig2 * 0.5f) *
         pT[i]) * invf;
    d2 = (logf(pS0[i] / pK[i]) + (r - sig2 * 0.5f) *
         pT[i]) * invf;
    erf1 = 0.5f + 0.5f * erff(d1 * invsqrt2);
    erf2 = 0.5f + 0.5f * erff(d2 * invsqrt2);
    pC[i] = pS0[i] * erf1 - pK[i] * expf((-1.0f) * r *
            pT[i]) * erf2;
  }
}
```

Добавьте в массив указателей **GetOptionPrices** новую функцию.

```
tGetOptionPrices GetOptionPrices[9] =
{ GetOptionPricesV0, GetOptionPricesV1, GetOptionPricesV2,
  GetOptionPricesV3, GetOptionPricesV4, GetOptionPricesV5,
  GetOptionPricesV6, GetOptionPricesV7, GetOptionPricesV8
};
```

Соберите программу и проведите эксперименты. На описанной ранее инфраструктуре авторы получили следующие времена.

Таблица 16. Время работы на сопроцессоре Xeon Phi версии 8 (в секундах), 60 ядер

N	60 000 000	120 000 000	180 000 000	240 000 000
Версия V8	0,009	0,018	0,027	0,035
S(V6.3/V8)	46,878	47,505	47,610	47,832

Таблица 17. Время работы на сопроцессоре Xeon Phi версии 8 (в секундах), 120 ядер

N	60 000 000	120 000 000	180 000 000	240 000 000
Версия V8	0,007	0,013	0,019	0,026
S(V6.3/V8)	63,873	65,048	65,426	65,887

Таблица 18. Время работы на сопроцессоре Xeon Phi версии 8 (в секундах), 240 ядер

N	60 000 000	120 000 000	180 000 000	240 000 000
Версия V8	0,007	0,013	0,019	0,026
S(V6.3/V8)	63,222	64,768	65,379	65,420

Как видно из табл. 16-18, времена работы версии 8 по сравнению с версией 7.1 ухудшились на 60 ядрах и улучшились на 120 и 240. Ускорение на 120 и 240 потоках также выросло.

Наконец, для сравнения приведем результаты работы версии 8 на центральном процессоре (16 ядер) и сопроцессоре (120 ядер).

Таблица 19. Сравнение времени работы версии 8 (в секундах)

N	60 000 000	120 000 000	180 000 000	240 000 000
Xeon	0,030	0,061	0,090	0,116
Xeon Phi	0,007	0,013	0,019	0,026

Как видно из табл. 19, время работы параллельной версии с прогревом на сопроцессоре примерно в 4,6 раза лучше, чем на центральном процессоре.

Также отметим, что применения прагмы vector nontemporal почти не ускоряет код на центральном процессоре (сравните времена версий 7.1 и 8), что вполне объяснимо – используется всего 16 потоков, шина не перегружается.

5. Дополнительные задания

1. Ответить на вопросы в тексте.

2. Выяснить, какой способ организации данных в данной задаче более эффективен.

3. Изменить реализацию так, чтобы она производила одновременное вычисление опционов типа Call и Put. Провести эксперименты с разными версиями кода, проверить влияние техник оптимизации на время работы последовательной и параллельной версий для Xeon и Xeon Phi, привести выкладки, подтверждающие факт перегрузки шины на Xeon Phi при использовании значительного числа потоков.

4. Провести эксперименты на Xeon Phi с разным числом потоков. Выяснить, какое число потоков является оптимальным.

6. Литература

6.1. Использованные источники информации

1. Кнут Д. Искусство программирования, том 2. Получисленные методы. – 3-е изд. – М.: Вильямс, 2007. – С. 832.
2. Ширяев А.Н. Основы стохастической финансовой математики. – Фазис, 2004. – 1076с.

6.2. Дополнительная литература

3. Гергель В.П., Баркалов К.А., Мееров И.Б., Сысоев А.В. и др. Параллельные вычисления. Технологии и численные методы. Учебное пособие в 4 томах. – Нижний Новгород: Изд-во Нижегородского госуниверситета, 2013. – 1394 с.

Авторы благодарят Никиту Астафьева и Сергея Майданова за полезные замечания и внимание к работе.