Нижегородский государственный университет им. Н.И.Лобачевского

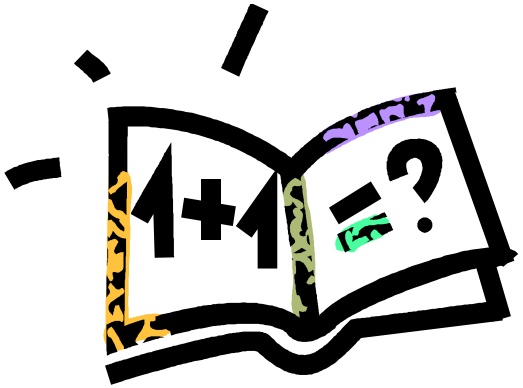
Факультет Вычислительной математики и кибернетики

Программирование для Intel Xeon Phi

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

При поддержке компании Intel

Мееров И.Б., Сысоев А.В.
Кафедра математического обеспечения ЭВМ



Вы думаете, все так просто?

Да, все просто.

*Но совсем не так…**

Очень важно не перестать задавать вопросы.

*Любопытство не случайно дано человеку.**

*** Альберт Эйнштейн**

Содержание

- Введение
- Цель работы
- Тестовая инфраструктура
- Оценивание опционов Европейского типа
- Оптимизация для Xeon и Xeon Phi: шаг за шагом
- Задания для самостоятельной работы
- Литература

Авторы выражают благодарность Никите Астафьеву
и Сергею Майданову за полезные комментарии и обсуждения

1. Приступаем к работе

Цель работы

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

Задачи:

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

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

2. ОЦЕНИВАНИЕ ОПЦИОНОВ ЕВРОПЕЙСКОГО ТИПА

Модель финансового рынка

Модель Блэка-Шоулса

$$dB_t = rB_t dt, \quad B_0 > 0 \qquad (1)$$

$$dS_t = S_t\big((r-\delta)dt + \sigma dW_t\big), \quad S_0 > 0 \qquad (2)$$

- S_t - цена акции в момент времени t
- B_t - цена облигации в момент времени t
- r - процентная ставка (считаем константой)
- σ - волатильность (считаем константой)
- δ - ставка дивиденда (считаем равной нулю)
- $W = (W_t)_{\{t\geq 0\}}$ - Винеровский случайный процесс (E=0 с P=1, независимые приращения, $W_t - W_s \sim N(0, t\text{-}s)$, где $s < t$), траектории процесса $W_t(\omega)$ – непрерывные функции времени с P=1.
- S_0, B_0 - заданы.

Модель финансового рынка

- **Модель Блэка-Шоулса**

$$dB_t = rB_t dt, \quad B_0 > 0$$
$$dS_t = S_t\left(rdt + \sigma dW_t\right), \quad S_0 > 0$$

- Система стохастических дифференциальных уравнений.
- Вообще говоря, *S* – вектор (цен акций). Для упрощения считаем, что *S* – скаляр (рассматривается одна акция).
- В одномерном случае несколько упрощаются выкладки и код, схема вычислений остается той же.

Модель финансового рынка

- **Модель Блэка-Шоулса, уравнение цен акций**

$$dS_t = S_t\left(rdt + \sigma dW_t\right), \quad S_0 > 0$$

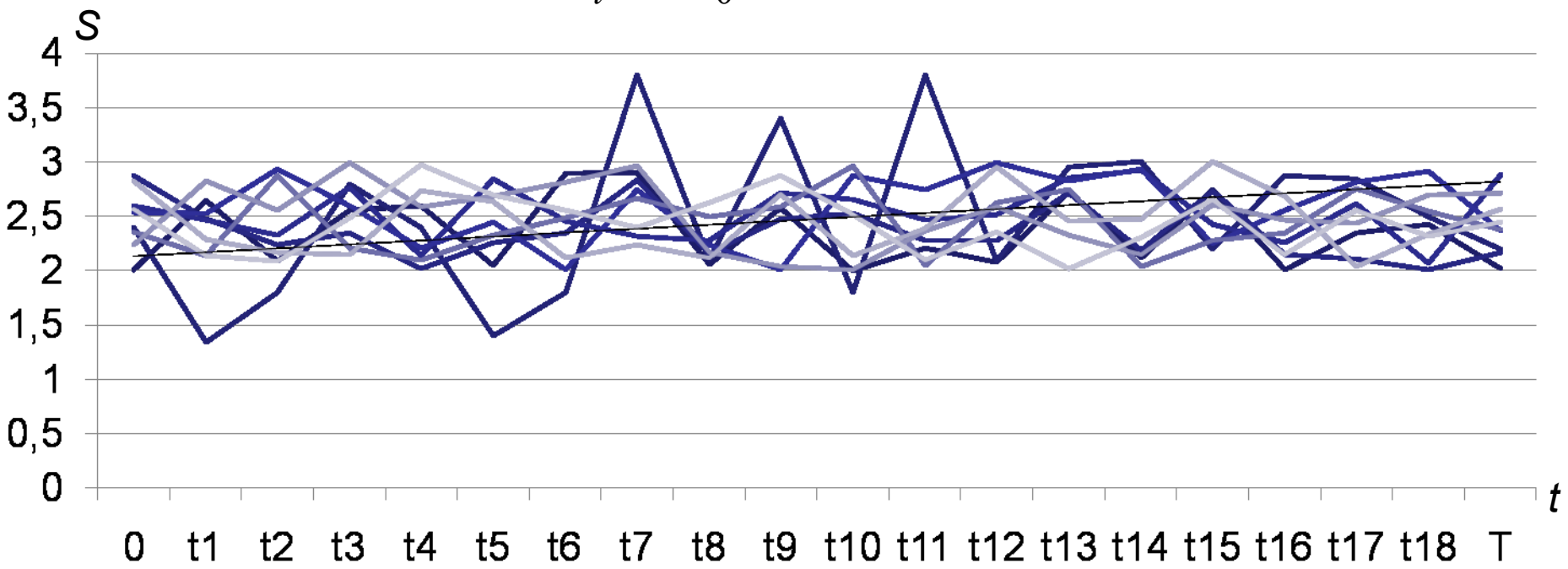
- При постоянных параметрах система имеет аналитическое решение:

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

- В противном случае уравнение (система в многомерном случае) решается одним из стандартных методов (например, Рунге-Кутты), W_t получается из $N(0, t)$.

Как работает формула?

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



- Выполняется моделирование поведения цены акции на рынке, значения W_t - выход генератора случайных чисел.
- Шаг 1 повторяется много раз, далее результаты усредняются.

Опцион Европейского типа на акцию…

- *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.
- K называется ценой исполнения опциона (*страйк,* strike price), а C – *ценой опциона*.

Опцион Европейского типа на акцию…

- *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.
  - 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 $S_T – K$).

Опцион Европейского типа на акцию

- *Справедливая цена* такого опционного контракта – цена, при которой наблюдается баланс выигрыша/проигрыша каждой из сторон.
- Логично определить такую цену как средний выигрыш стороны P_2:
- $$C = E\left(e^{-rT}\left(S_T - K\right)^+\right) \qquad (4)$$

Математическое ожидание

Разность между ценой акции в момент исполнения T и страйком, если она положительна

Дисконтирование (1р. в момент t = T приводится к 1р. при t = 0)

Вычисление справедливой цены опциона

- Для Европейского колл-опциона при сделанных ранее предположениях известно аналитическое решение.
- Аналитическое решение описывается **формулой Блэка – Шоулса** для вычисления цены опциона в момент времени $t = 0$ (*F – функция стандартного нормального распределения*):

$$C = S_0 F(d_1) - K e^{-rT} F(d_2)$$

$$d_1 = \ln \frac{S_0}{K} + \frac{\left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}} \qquad (5)$$

$$d_2 = \ln \frac{S_0}{K} + \frac{\left(r - \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$

3. Оценивание набора опционов

Набор опционов

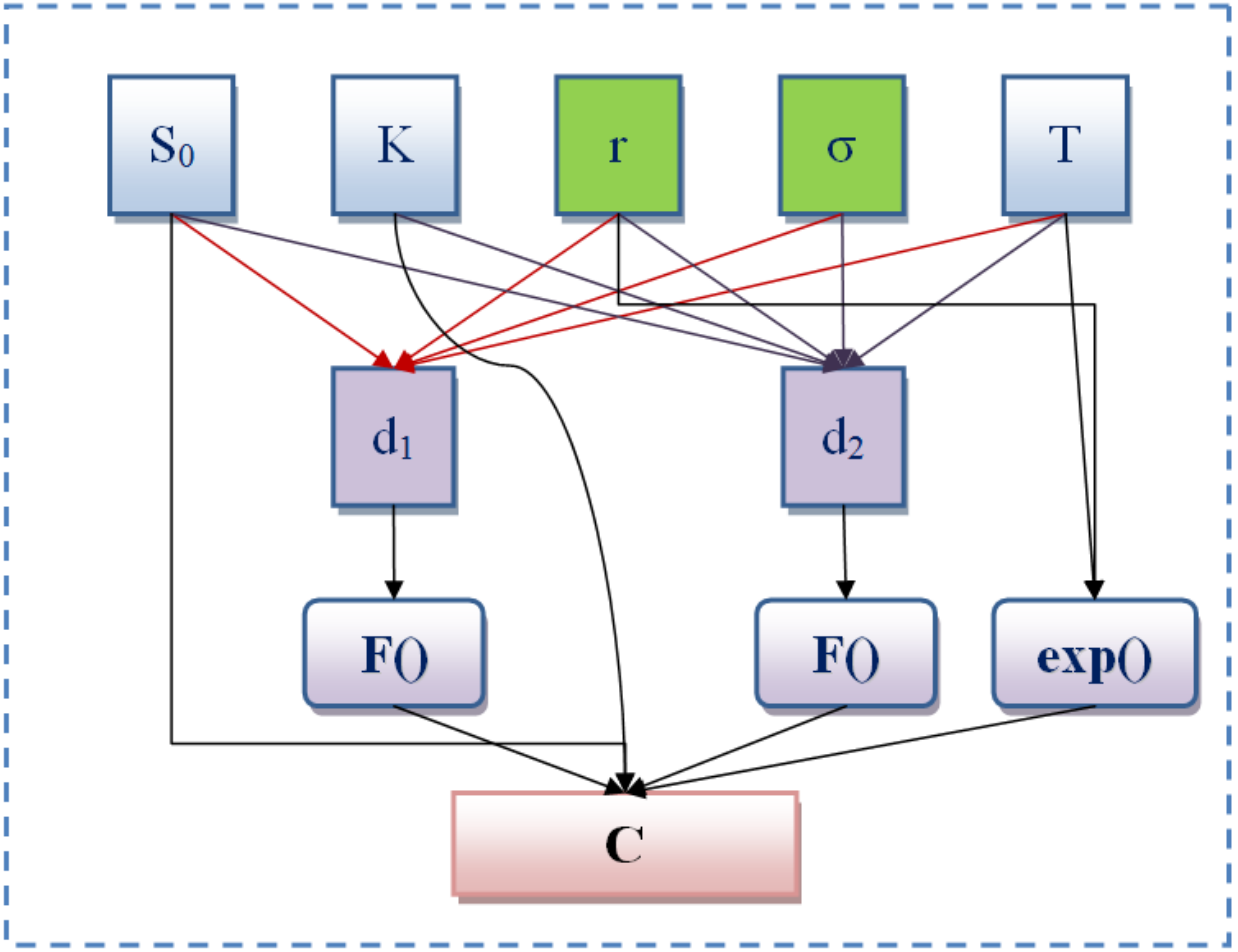
- Для вычисления цены одного такого опциона не нужна высокопроизводительная вычислительная техника.
- На практике организации, работающие на финансовых рынках, вычисляют цены гигантского количества разных опционов, которые можно выпустить в конкретных рыночных условиях.
- Учитывая, что время финансовых расчетов существенно влияет на скорость принятия решений, каждая секунда на счету.
- Сокращение времени оценивания набора опционов является достаточно важной задачей.

Схема информационных зависимостей

$$C = S_0 F(d_1) - Ke^{-rT} F(d_2)$$

$$d_1 = \ln \frac{S_0}{K} + \frac{\left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$

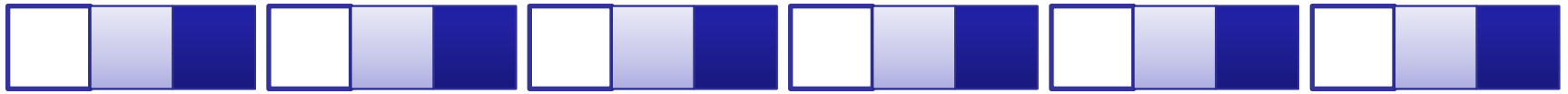
$$d_2 = \ln \frac{S_0}{K} + \frac{\left(r - \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$



4. Программная реализация

Структуры данных

- Массив структур (AoS – Array of Structures)



- Структура массивов (SOA – Structure of Arrays)

Задание: выяснить, что лучше.
Далее в работе используется второй подход.

Точка отсчета. Расчет по аналитической формуле…

```
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

  float res = GetOptionPrice();
  printf("%.8f;\n", res);
  return 0;
}
```

Точка отсчета. Расчет по аналитической формуле

```
const float sig = 0.2f;
const float r   = 0.05f;
const float T   = 3.0f;
const float S0  = 100.0f;
const float K   = 100.0f;

float GetOptionPrice() {
  float C, 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;
}
```

Нулевая базовая версия. Набор опционов…

```
__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;
  }
}
```

Нулевая базовая версия. Набор опционов…

```
__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;
  }
}
```

Нулевая базовая версия. Набор опционов

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

Примем за точку отсчета!

Версия 1. Исключение ненужных преобразований типов…

```
__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 = (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;
  }
}
```

Версия 1. Исключение ненужных преобразований типов…

N	60 000 000	120 000 000	180 000 000	240 000 000
Версия V0 (секунды)	17,002	34,004	51,008	67,970
Версия V1 (секунды)	**16,776**	**33,549**	**50,337**	**66,989**

Малое ускорение?
Иногда бывает в 3 раза…

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

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

Используем функцию **erff()** вместо функции **cdfnormf()**, так как она проще для вычислений.

Вопрос: почему?

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

```
__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;
  }
}
```

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

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**

Ускорение вычислений в несколько раз!

Версия 3. Векторизация: использование ключевого слова restrict…

- Что такое векторизация?
- Что такое restrict?
- Зачем использовать restrict?
- Я использовал restrict, но ничего не произошло. Что я делаю неправильно?

Версия 3. Векторизация: использование ключевого слова restrict…

- Как обстоит дело с векторизацией в нашей программ?
- Добавим ключ –vec-report3 или –vec-report6 (Linux) или –Qvec-report3 или –Qvec-report6 (Windows)
- Добавим ключ –mavx (иначе будет использоваться SSE, но не AVX).

```
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$
```

- «Что это, Бэрримор?» Зависимости по данным, хотя в этот раз отчет предпочел это скрыть.

Версия 3. Векторизация: использование ключевого слова restrict

```
__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;
  }
}
```

Версия 4. Векторизация: использование директивы simd

```
__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;
  }
}
```

Версия 4*. Векторизация: использование директив ivdep и vector always

```
__declspec(noinline) void GetOptionPricesV4(float *pT,
  float *pK, float *pS0, float *pC)
{
  int i;
  float d1, d2, erf1, erf2;
#pragma ivdep
#pragma vector always
  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;
  }
}
```

Версии 3-4. Векторизация

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**

1. Отчет: loop was vectorized (SIMD loop was vectorized)
2. Все 3 способа работают, результаты похожи.
3. Ускорение не в 8 раз, а в 5,43.
Обсуждение п.п. 2 и 3!

Для обсуждения

```
__declspec(noinline) void GetOptionPricesV4(float *pT,
  float *pK, float *pS0, float *pC)
{
  int i;
  float d1, d2, erf1, erf2;
#pragma simd // Intel рекомендует. Но иногда лучше ivdep
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;
  }
}
```

5.43 – РАБОТАЕТ ЗАКОН АМДАЛЯ

Версия 5. Вынос инвариантов из цикла…

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

Версия 5. Вынос инвариантов из цикла

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**

Обычно имеет смысл выносить инварианты

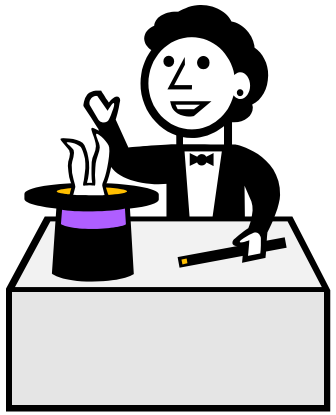
Версия 6. Эквивалентные преобразования. Вычисление квадратного корня…

```
__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;
  }
}
```

Версия 6. Эквивалентные преобразования. Вычисление квадратного корня

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**

Обычно имеет смысл заменять деления умножениями. В данном случае не дало эффект.



Версия 6.1. Выравнивание данных

```
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;
}
```

- SSE: 16, AVX: 32, Xeon Phi: 64
- memalign() -> __mm_malloc()
- Windows: __declspec(align(XX)) float T[N];
  Linux: float T[N] __attribute__((aligned(64)));
- #pragma vector aligned, __assume_aligned, __assume

Версия 6.2. Пониженная точность

- В данной задаче можно понизить точность.
- icc … **-fimf-precision=low -fimf-domain-exclusion=31**

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 7. Распараллеливание

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

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 (16 ядер)	**0,058**	**0,084**	**0,126**	**0,153**

Версия 7.1. Прогрев…

- Исключить накладные расходы на создание потоков.
- Подготовить кэш к работе.

- **Обсуждение: честно или нет?**

Версия 7.1. Прогрев…

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**

Версия 7.1. Прогрев

- 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).
- 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).

Эксперименты на сопроцессоре Xeon Phi…

- Сравнение имеет смысл проводить, начиная с версии 7. Однако, чтобы картина была полной, рекомендуем провести эксперименты и с начальными версиями.
- 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.

Эксперименты на сопроцессоре Xeon Phi. Векторизованный последовательный код

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

- На 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.
- Более значительный прирост (порядка 60%) дает прогрев, что неудивительно, принимая во внимание тот факт, что мы создаем на порядок больше потоков.
- Времена работы версии 6.3. на сопроцессоре практически сравнялись с временами работы на процессоре.

Эксперименты на сопроцессоре Xeon Phi. Векторизованный параллельный код…

60

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

120

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

240

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

Эксперименты на сопроцессоре Xeon Phi. Векторизованный параллельный код

- 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).
- При запуске в 120 потоков ускорение выше, чем при запуске в 60 потоков, что согласуется с техническими особенностями исполнения кода на Xeon Phi.

- **Можно ли что-то улучшить?**

Версия 8. Оптимизация работы с кэш-памятью…

- Используются 4 массивами (**pT**, **pK**, **PS0**, **pC**). При этом 3 из них используются только для чтения, а один (**pC**) для записи.
- Значения, которые 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.
- **#pragma vector nontemporal**

Версия 8. Оптимизация работы с кэш-памятью

60

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

120

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

240

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

Ускорение растет с 54 до 65.

Xeon vs. Xeon Phi

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

Задания для самостоятельной работы

- Выяснить, какой способ организации данных в данной задаче более эффективен.
- 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.
- Провести эксперименты на Xeon Phi с разным числом потоков. Выяснить, какое число потоков является оптимальным.

Литература

- **Использованные источники информации**

  Кнут Д. Искусство программирования, том 2. Получисленные методы. – 3-е изд. – М.: Вильямс, 2007. – С. 832.

  Ширяев А.Н. Основы стохастической финансовой математики. – Фазис, 2004. – 1076с.

- **Дополнительная литература**

  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.

Авторский коллектив

- Мееров Иосиф Борисович,
  к.т.н., доцент, зам. зав. кафедры
  Математического обеспечения ЭВМ факультета ВМК ННГУ
  meerov@vmk.unn.ru
- Сысоев Александр Владимирович,
  к.т.н., ассистент кафедры
  Математического обеспечения ЭВМ факультета ВМК ННГУ
  sysoyev@vmk.unn.ru