


Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

Нижегородский государственный университет им. Н.И. Лобачевского
Институт информационных технологий, математики и механики

Вычисление справедливой цены опциона европейского типа

Мееров И.Б., Сысоев А.В., Волокитин В.Д.
Кафедра МОСТ

Содержание

Изложение в основном ведется по работе:
Meyerov I., Sysoyev A., Astafiev N., Burylov I. Performance Optimization of Black-Scholes Pricing // in: High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches.– Elsevier, 2015. – pp. 319-340.
С дополнениями по результатам экспериментов на новых архитектурах.

1. ВВЕДЕНИЕ

1. Введение

Цель лекции *– изучение некоторых принципов оптимизации вычислений в вычислительных 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 серверных процессоров Intel разных поколений
- Анализ результатов оптимизации и распараллеливания

2. ТЕСТОВАЯ ИНФРАСТРУКТУРА

2. Тестовая инфраструктура. Конфигурация 1

- Конфигурация 1: «**старое** программно-аппаратное окружение»

Процессор	2x восьмиядерных процессора Intel Xeon E5-2690 (2.9 GHz)
Память	64 GB
Операционная система	Linux CentOS 6.2
Компилятор, профилировщик	Intel Parallel Studio XE 2013 SP1

2. Тестовая инфраструктура. Конфигурация 2

- Конфигурация 2: «**новое** программно-аппаратное окружение»

Процессор	2x 24-ядерных процессора Intel Xeon Platinum 8260L (2.4 GHz)
Память	192 GB
Операционная система	Linux RedHat 4.8.5-39
Компилятор, профилировщик	Intel Parallel Studio XE 2020 update 1

3. ОЦЕНИВАНИЕ ОПЦИОНОВ ЕВРОПЕЙСКОГО ТИПА

3. Оценивание опционов Европейского типа
3.1. Модель финансового рынка

Модель Блэка – Шоулса

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

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

- 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.

3. Оценивание опционов Европейского типа
3.2. Модель Блэка – Шоулса

- **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

3. Оценивание опционов Европейского типа
3.3. Аналитическое решение

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

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

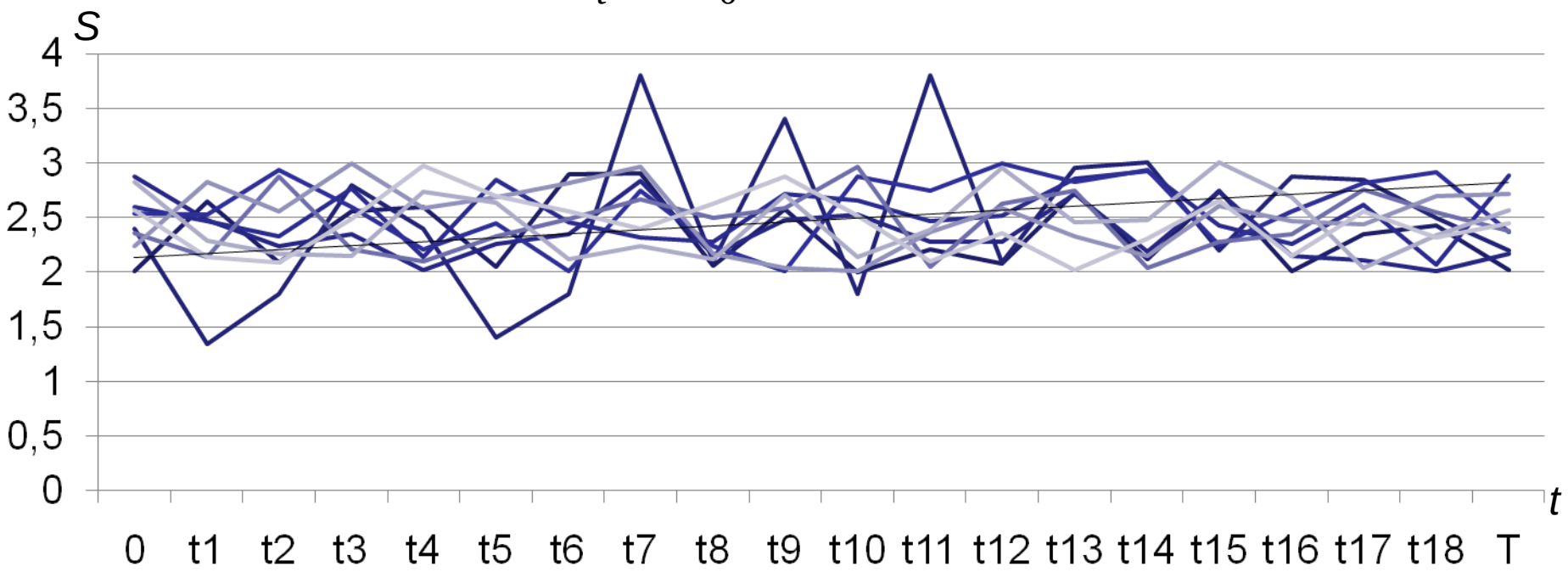
- При постоянных параметрах система имеет аналитическое решение:

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

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

3. Оценивание опционов Европейского типа
3.4. Как работает аналитическое решение?

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



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

3. Оценивание опционов Европейского типа
3.5. Понятие опциона

- *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*.

3. Оценивание опционов европейского типа
3.6. Колл-опцион Европейского типа

- *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 $S_T - K$).

3. Оценивание опционов Европейского типа
3.7. Справедливая цена опциона

- *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)

3. Оценивание опционов Европейского типа
3.8. Формула Блэка – Шоулса

- 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 опционов Европейского типа
3.9. Оценивание набора 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.**

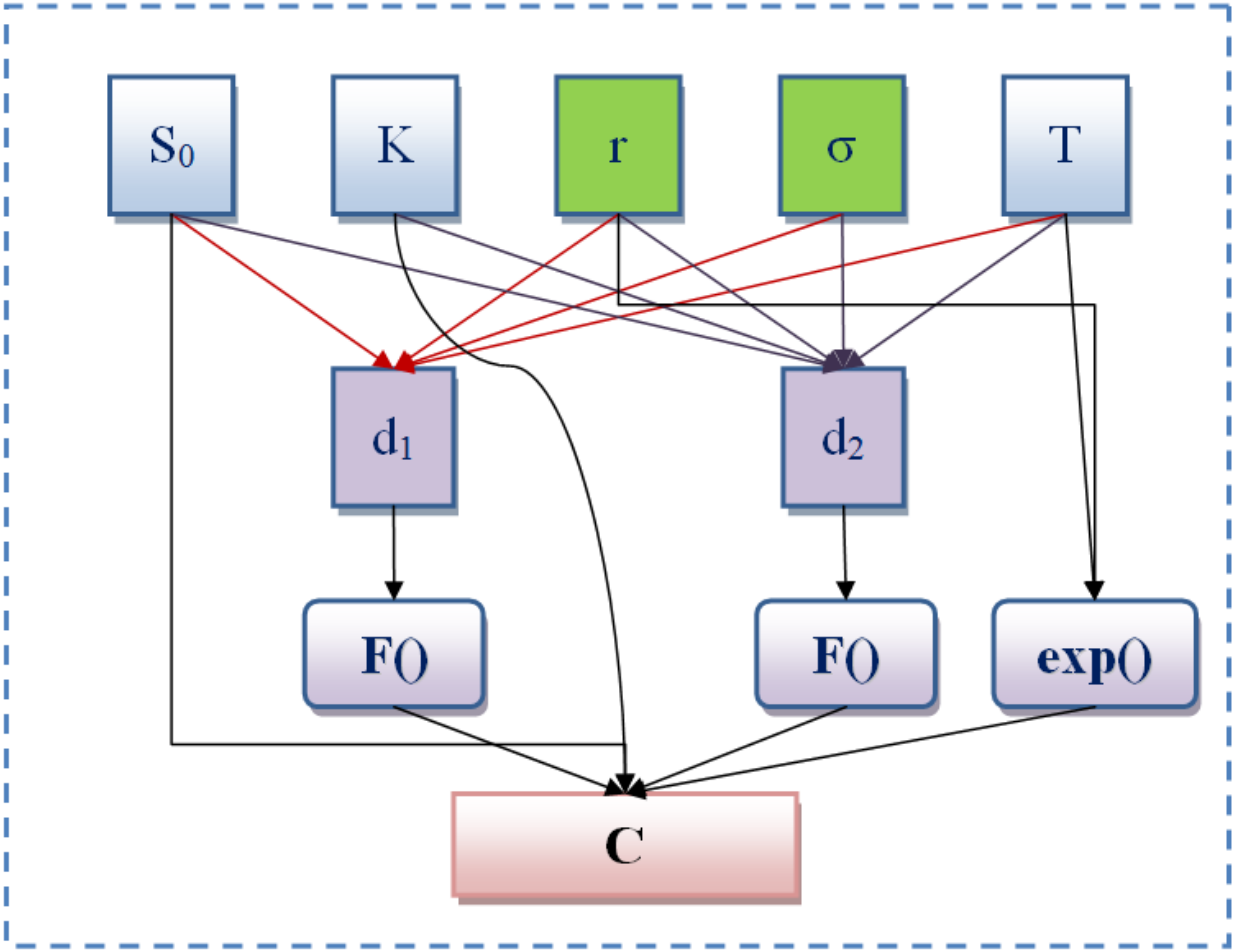
4. ОПТИМИЗАЦИЯ ШАГ ЗА ШАГОМ

4. Оптимизация шаг за шагом
4.1. Схема информационных зависимостей

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

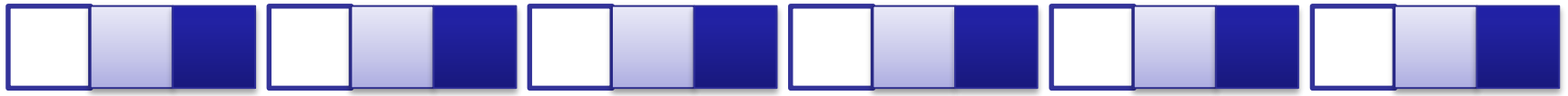
$$d_1 = \ln \frac{S_0}{K} + \frac{\left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$

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



4. Оптимизация шаг за шагом
4.2. Структуры данных

- Массив структур (AoS – Array of Structures)



- Структура массивов (SOA – Structure of Arrays)

Задание: выяснить, что лучше.
Далее в работе используется второй подход (как правило, лучше работает для векторизации вычислений).

4. Оптимизация шаг за шагом
4.3. Проверяем корректность для 1 опциона

```
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;
}
```

4. Оптимизация шаг за шагом
4.3. Проверяем корректность для 1 опциона

```
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;
}
```

4. Оптимизация шаг за шагом
4.4. Набор опционов. Базовая версия…

```
__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;
  }
}
```

4. Оптимизация шаг за шагом
4.4. Набор опционов. Базовая версия…

```
__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;
  }
}
```

4. Оптимизация шаг за шагом
4.4. Набор опционов. Базовая версия

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!

4. Оптимизация шаг за шагом
4.4. Базовая версия.

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!

4. Оптимизация шаг за шагом
4.5. Не смешиваем типы данных

```
__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;
  }
}
```

4. Оптимизация шаг за шагом
4.5. Не смешиваем типы данных (время в сек.)

N	60 000 000	120 000 000	180 000 000	240 000 000
Базовая версия	17,002	34,004	51,008	67,970
Типы данных	**16,776**	**33,549**	**50,337**	**66,989**

Малое ускорение?
Иногда бывает в 3 раза...

4. Оптимизация шаг за шагом
4.6. Эквивалентные преобразования

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

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

Вопрос: почему?

4. Оптимизация шаг за шагом
4.6. Эквивалентные преобразования (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;
  }
}
```

4. Оптимизация шаг за шагом
4.6. Эквивалентные преобразования (время в сек.)

N	60 000 000	120 000 000	180 000 000	240 000 000
Базовая версия	17,002	34,004	51,008	67,970
Типы данных	16,776	33,549	50,337	66,989
Мат. функции	**2,871**	**5,727**	**8,649**	**11,230**

Ускорение вычислений в несколько раз!

4. Оптимизация шаг за шагом
4.7. Векторизация…

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

4. Оптимизация шаг за шагом
4.7. Векторизация…

- 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.

4. Оптимизация шаг за шагом
4.7. Векторизация. Ключевое слово 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;
  }
}
```

4. Оптимизация шаг за шагом
4.7. Векторизация. Директива simd (omp 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;
  }
}
```

4. Оптимизация шаг за шагом
4.7. Векторизация. Директива ivdep

```
__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;
  }
}
```

4. Оптимизация шаг за шагом
4.7. Векторизация (время в секундах)

N	60 000 000	120 000 000	180 000 000	240 000 000
Базовая версия	17,002	34,004	51,008	67,970
Типы данных	16,776	33,549	50,337	66,989
Мат. функции	2,871	5,727	8,649	11,230
Векторизация	**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!

4. Оптимизация шаг за шагом
4.7. Векторизация (обсуждение)

```
__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

4. Оптимизация шаг за шагом
4.8. Что не сработало?

- Далее рассматриваются несложные приемы оптимизации, которые часто бывают очень полезны, но не сработали для данного приложения

4. Оптимизация шаг за шагом
4.8. Что не сработало? Вынос инвариантов

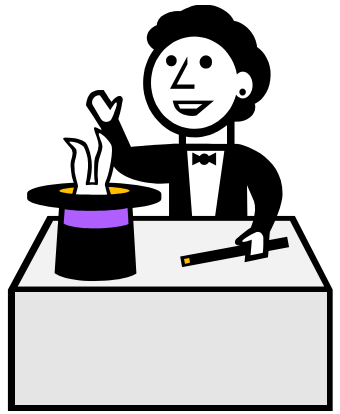
```
const float invsqrt2 = 0.707106781f;
__declspec(noinline) void GetOptionPricesV5(float *pT,
  float *pK, float *pS0, float *pC)
{
  int i;
  float d1, d2, erf1, erf2;
#pragma simd
  for (i = 0; i < N; i++)
  {
    d1 = (logf(pS0[i] / pK[i]) + (r + sig * sig * 0.5f) *
         pT[i]) / (sig * sqrtf(pT[i]));
    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;
  }
}
```

ЕСТЬ И ДРУГИЕ ИНВАРИАНТЫ.
СПРАВИТСЯ ЛИ КОМПИЛЯТОР?
ЗДЕСЬ – ДА, ВООБЩЕ – НЕ ВСЕГДА

4. Оптимизация шаг за шагом
4.8. Что не сработало? Вынос инвариантов (t в сек.)

N	60 000 000	120 000 000	180 000 000	240 000 000
Базовая версия	17,002	34,004	51,008	67,970
Типы данных	16,776	33,549	50,337	66,989
Мат. функции	2,871	5,727	8,649	11,230
Векторизация	0,521	1,036	1,566	2,067
Инварианты	**0,527**	**1,047**	**1,580**	**2,085**

Обычно имеет смысл выносить инварианты



4. Оптимизация шаг за шагом
4.8. Что не сработало? Эквивалентные преобразования

```
__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;
  }
}
```

4. Оптимизация шаг за шагом
4.8. Что не сработало? Эквивалентные преобразования

N	60 000 000	120 000 000	180 000 000	240 000 000
Базовая версия	17,002	34,004	51,008	67,970
Типы данных	16,776	33,549	50,337	66,989
Мат. функции	2,871	5,727	8,649	11,230
Векторизация	0,521	1,036	1,566	2,067
Инварианты	0,527	1,047	1,580	2,085
Корень	**0,538**	**1,071**	**1,614**	**2,133**

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

4. Оптимизация шаг за шагом
4.8. Что не сработало? Выравнивание массивов

```
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, AVX-512: 64
- memalign() -> __mm_malloc()
- Windows: __declspec(align(XX)) float T[N];
  Linux: float T[N] __attribute__((aligned(64)));
- #pragma vector aligned, __assume_aligned, __assume

4. Оптимизация шаг за шагом
4.9. Более сложные оптимизации

- Далее рассматриваются более сложные приемы оптимизации и обсуждается некоторые вопросы корректности измерения времени

4. Оптимизация шаг за шагом
4.10. Понижение точности (время в сек.)

- 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
Базовая версия	17,002	34,004	51,008	67,970
Типы данных	16,776	33,549	50,337	66,989
Мат. функции	2,871	5,727	8,649	11,230
Векторизация	0,521	1,036	1,566	2,067
Инварианты	0,527	1,047	1,580	2,085
Корень	0,538	1,071	1,614	2,133
Точность	**0,438**	**0,871**	**1,314**	**1,724**

4. Оптимизация шаг за шагом
4.11. Параллелизм (время в сек.)

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

N	60 000 000	120 000 000	180 000 000	240 000 000
Базовая версия	17,002	34,004	51,008	67,970
Типы данных	16,776	33,549	50,337	66,989
Мат. функции	2,871	5,727	8,649	11,230
Векторизация	0,521	1,036	1,566	2,067
Инварианты	0,527	1,047	1,580	2,085
Корень	0,538	1,071	1,614	2,133
Точность	0,438	0,871	1,314	1,724
OpenMP (16 ядер)	**0,058**	**0,084**	**0,126**	**0,153**

4. Оптимизация шаг за шагом
4.12. «Прогрев»

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

- **Обсуждение: честно или нет?**

4. Оптимизация шаг за шагом
4.12. «Прогрев» (время в сек.)

N	60 000 000	120 000 000	180 000 000	240 000 000
Базовая версия	17,002	34,004	51,008	67,970
Типы данных	16,776	33,549	50,337	66,989
Мат. функции	2,871	5,727	8,649	11,230
Векторизация	0,521	1,036	1,566	2,067
Инварианты	0,527	1,047	1,580	2,085
Корень	0,538	1,071	1,614	2,133
Точность	0,438	0,871	1,314	1,724
OpenMP	0,058	0,084	0,126	0,153
Точность + прогрев	0,409	0,812	1,226	1,603
OpenMP + прогрев	**0,033**	**0,062**	**0,091**	**0,118**

4. Оптимизация шаг за шагом
4.13. Переход на новое оборудование

- Что изменилось в аппаратном обеспечении:
  - Увеличилось количество **вычислительных ядер** (от 16 к 48)
  - Увеличился размер **векторного регистра** (от 256 бит к 512 бит)
  - Увеличилась **пропускная способность памяти** (до 2 раз на канал)
- Что изменилось в программном обеспечении (**компилятор**):
  - Улучшились алгоритмы оптимизации
  - Улучшилась производительность стандартных библиотек
  - Компилятор стал «умнее» и может существенно менять код
- Что изменилось в языке (директивы):
  - **#pragma simd** – более не поддерживается, так как появилась аналогичная функциональность в стандарте OpenMP – **#pragma omp simd**

4. Оптимизация шаг за шагом
4.14. Новое оборудование (время в сек.)

	E5-2690 (16 ядер)	Platinum 8260L (48 ядер)	
N	240 000 000	240 000 000	960 000 000
Базовая версия	67,970	5,145	20,677
Типы данных	66,989	3,150	12,418
Мат. функции	11,230	1,930	7,600
Векторизация	2,067	1,768	7,095
Инварианты	2,085	1,761	7,049
Корень	2,133	1,783	7,335
Точность	1,724	1,114	4,498
OpenMP	0,153	0,076	0,296
OpenMP + прогрев	**0,118**	**0,058**	**0,278**

ЧТО ПРОИЗОШЛО С КАЖДОЙ ОПТИМИЗАЦИЕЙ?

ПОЧЕМУ ТАКОЕ НИЗКОЕ УСКОРЕНИЕ НА 48 ЯДРАХ?

4. Оптимизация шаг за шагом
4.15. VTune – анализ памяти

Elapsed Time: 6.789s		
CPU Time:	23.941s	
Memory Bound:	65.7%	of Pipeline Slots
L1 Bound:	5.3%	of Clockticks
L2 Bound:	N/A*	of Clockticks
L3 Bound:	2.5%	of Clockticks
DRAM Bound:	N/A*	of Clockticks
DRAM Bandwidth Bound:	3.4%	of Elapsed Time
Store Bound:	22.6%	of Clockticks
NUMA: % of Remote Accesses:	36.8%	
UPI Utilization Bound:	3.7%	of Elapsed Time
Loads:	11,248,337,440	
Stores:	5,280,158,400	
LLC Miss Count:	0	
Average Latency (cycles):	15	
Total Thread Count:	50	
Paused Time:	0s	

*N/A is applied to metrics with undefined value. There is no data to calculate the metric.

ПАМЯТЬ ЯВЛЯЕТСЯ «УЗКИМ ГОРЛЫШКОМ» ПРИЛОЖЕНИЯ

ОЧЕНЬ БОЛЬШОЙ ПРОЦЕНТ НА ДОСТУП НА ЗАПИСЬ В ПАМЯТЬ. 1 ИЗ 4 МАССИВОВ ПАМЯТИ ИСПОЛЬЗУЕТСЯ ТОЛЬКО НА ЗАПИСЬ (25%)

ОГРОМНОЕ КОЛИЧЕСТВО УДАЛЕННОГО ДОСТУПА

- Для двухсокетной системы, если вся память привязана к одному процессору, то доля удаленного доступа не м.б. больше **50%**

4. Оптимизация шаг за шагом
4.16. NUMA-friendly выделение памяти

- Когда происходит выделение физической памяти (RAM)?
- Пропускная способность не растет так быстро, как вычислительная способность процессоров

```
#pragma omp parallel for simd
  for (i = 0; i < N; i++) {
    pT[i] = T;
    pS0[i] = S0;
    pK[i] = K;
    pC[i] = 0;
  } // NUMA-friendly initialization
  ...
```

- Изменяется участок кода, который не участвует в замерах времени, но он сильно сказывается на производительности
- Позволит ли теперь прогрев улучшить время работы программы?

4. Оптимизация шаг за шагом
4.17. Новое оборудование (время в сек.)

	E5-2690 (16 ядер)	Platinum 8260L (48 ядер)	
N	240 000 000	240 000 000	960 000 000
Базовая версия	67,970	5,145	20,677
Типы данных	66,989	3,150	12,418
Мат. функции	11,230	1,930	7,600
Векторизация	2,067	1,768	7,095
Инварианты	2,085	1,761	7,049
Корень	2,133	1,783	7,335
Точность	1,724	1,114	4,498
OpenMP	0,153	0,076	0,296
OpenMP + NUMA	-	0,023	0,091
NUMA + прогрев	**-**	**0,022**	**0,089**

4. Оптимизация шаг за шагом
4.18. Архитектурно-специфичная оптимизация

- 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. Это давало существенный прирост производительности на Xeon Phi 1-го поколения, но никак не повлияло на новые Xeon.
- **#pragma vector nontemporal**

5. ЗАКЛЮЧЕНИЕ

5. Заключение

- Программная оптимизация всегда останется востребованной
- Со временем, возможно, потребуется проводить оптимизацию, отличную от текущий, так как технологии не стоят на месте
- Не всякая оптимизация, которая кажется изначально перспективной, даст значимый прирост в скорости работы
- Оптимизирующие компиляторы становятся «умнее» с каждым годом, но они не смогут сделать все за программиста

Литература

1. Ширяев А.Н. Основы стохастической финансовой математики. – Фазис, 2004. – 1076с.
2. Meyerov I., Sysoyev A., Astafiev N., Burylov I. Performance Optimization of Black-Scholes Pricing // in: High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches.– Elsevier, 2015. – pp. 319-340.
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.

Авторский коллектив курса

- Мееров Иосиф Борисович, к.т.н., доцент, зам. зав. каф. МОСТ ИИТММ
- Волокитин Валентин Дмитриевич, аспирант каф. МОСТ ИИТММ

Авторы курса благодарят А. Сысоева, И. Бурылова и Н. Астафьева за продуктивную совместную работу и помощь в подготовке первой версии материалов.

Контакты

Нижегородский государственный университет
http://www.unn.ru

Институт информационных технологий, математики и механики
http://www.itmm.unn.ru

Мееров И.Б.
meerov@vmk.unn.ru