УДК 519.863:336.714

ОСОБЕННОСТИ ОПТИМИЗАЦИИ ВЫЧИСЛЕНИЙ В ПРИКЛАДНЫХ ПРОГРАММАХ НА ЯЗЫКЕ С НА ПРИМЕРЕ ОЦЕНИВАНИЯ ОПЦИОНОВ ЕВРОПЕЙСКОГО ТИПА

С.И. Бастраков, Р.В. Донченко, И.Б. Мееров, А.Н. Половинкин

Рассматриваются вопросы оптимизации вычислений в прикладных расчетных программах на языке С. Основной целью оптимизации в данном случае является эффективная утилизация имеющихся вычислительных ресурсов. Изложение проводится на примере решения задачи определения справедливой цены опциона европейского типа. Эффективность предложенного подхода подтверждена вычислительными экспериментами.

Ключевые слова: оптимизация вычислений, многоядерные архитектуры, финансовая математика.

Введение

Оптимизация вычислений для уменьшения времени работы прикладных программ расчетного характера – актуальная задача современной науки и техники. Решением этой задачи активно занимаются ученые и инженеры по всему миру [1–3]. Существенной проблемой является то, что прикладные программисты часто не вполне владеют приемами оптимизации ПО, а системные программисты плохо разбираются в предметных областях, что затрудняет процесс оптимизации. В настоящей работе авторы ставят целью продемонстрировать методику уменьшения времени вычислений, не выходя за рамки многоядерных архитектур и языка программирования высокого уровня, оставаясь в пределах доступного исследователю программно-аппаратного окружения. Изложение ведется на примере языка С, подавляющая часть описанных приемов с тем же успехом может быть применена к программам на языке Fortran (чаще всего именно С или Fortran используются при реализации численных методов). Для иллюстрации приемов оптимизации рассматривается задача из области финансовой математики – определение справедливой цены опциона европейского типа. Процесс оптимизации вычислений иллюстрируется фрагментами программного кода, полученный результат подтверждается вычислительными экспериментами.

Постановка задачи и метод решения

Цена европейского опциона, основанного на облигации, может быть вычислена по следующим формулам [4, 5]:

$$C(t,s) = E(e^{-\int_t^s r(u)du} \max\{P(s,T) - K, 0\}), \quad 0 \le t \le s \le T \qquad (1)$$

где t – текущий момент времени; s – срок исполнения опциона; $P(s, T)$ – цена в момент времени s облигации со сроком погашения T; K – цена исполнения опциона; $r(t)$ – краткосрочная безрисковая ставка в момент времени t. Искомая величина – цена опциона $C = C(0, s)$. (Дополнительная информация об использованных понятиях может быть получена из работы [4]). Математическая постановка задачи, метод решения и вычислительная схема подробно описаны в [5]. Приведем краткое описание схемы проведения вычислений на концептуальном уровне.

Задача решается методом Монте-Карло.

Шаг 1. Инициализация параметров начальными значениями.

Шаг 2. Вычисления для определения цены опциона методом Монте-Карло.

В цикле от 0 до количества повторений в методе Монте-Карло выполнить:

Шаг 2.1. Пусть k – число шагов в разностной схеме решения стохастического дифференциального уравнения (СДУ), описывающего поведение ставки $r(t)$ от момента заключения опционного контракта ($t = 0$) до истечения срока его исполнения ($t = s$). На данном шаге генерируется k нормально распределенных псевдослучайных чисел.

Шаг 2.2. Решение СДУ методом Эйлера. Используются результаты шага 2.1. Вычисляются k значений процентной ставки $r(t)$ на рассматриваемом интервале времени.

Шаг 2.3. Определение цены облигации $P(s,T)$.

Шаг 3. Усреднение полученных данных: вычисление цены опциона $C = C(0, s)$ по формуле (1).

Программная реализация и оптимизация

В рамках данной работы для демонстрации приемов оптимизации по скорости создано несколько версий ПО. Каждая следующая версия улучшает предыдущую, сокращая время работы. Реализация выполнена на языке программирования ANSI C. Кроме того, разработана отдельная версия с использованием NVidia CUDA, оптимизированная под графические процессоры NVidia.

Параметры задачи и окружение. В тестовой задаче выполнялось определение цены 30 опционов с разным значением цены исполнения, использовалось 50000 повторений в методе Монте-Карло и вещественная арифметика двойной точности. Количество опционов было выбрано из тех соображений, чтобы время работы базовой версии составляло порядка 300 секунд. Вычислительный эксперимент производился с использованием следующего программно-аппаратного окружения: 2x INTEL XEON e5520 (4 ядра, 2.27 Ггц), 16 Гб ОЗУ, NVIDIA TESLA c1060 (240 ядер, 1.30 ггц), OPENSUSE LINUX 11.1 (x86_64), kernel 2.6.27.29-0.1-default, glibc 2.9, gcc 4.3.2, INTEL C++ compiler for linux 11, intel math kernel library 10.2.2, nvidia cuda toolkit 2.2.

Базовая версия – элементы реализации. Базовая версия представляет собой неоптимизированную реализацию описанной ранее вычислительной схемы. Для генерации псевдослучайных чисел на шаге 2.1 применяется второй вариант преобразования Бокса–Мюллера [6]; равномерно распределенные на (0,1) числа получаются при помощи генератора MCG59 (реализован по описанию в [7]). Общее время работы исходной версии составляет 327 с.

Базовая версия – профилировка и оптимизирующий компилятор. Первоначальным этапом оптимизации отлаженного работоспособного программного кода является его профилировка. В данной работе использовалась консольная версия Intel VTune Performance Analyzer. Рассмотрим результаты профилирования базовой версии (компилятор gcc, ключ оптимизации – O2).

В таблицах приведены наиболее требовательные к вычислительным ресурсам функции, количество тактов процессора (**NT**) и относительное время работы функций в процентах от общего времени работы программы (**SelfTime**). Время, потраченное на вызов других функций из анализируемых функций, не учитывается.

Описание функции	Имя функции	NT	SelfTime
Моделирование r(t) (Шаг 2.2)	pow	442 342	85,80%
Генерация псевдослучайных чисел (Шаг 2.1)	log	18 541	3,60%
	generateGaussian	11 985	2,32%
	sincos	3 900	0,76%
Вспомогательные вычисления	run	17 459	3,39%

Из результатов профилирования видно, что основную вычислительную нагрузку составляют моделирование процентной ставки (~86%) и генерация псевдослучайных чисел (~7%). Прежде чем приступить к программной оптимизации, попробуем добиться прироста производительности при помощи использования оптимизирующих компиляторов. В рамках данной работы используется один из лучших компиляторов C/C++ – Intel C++ Compiler for Linux 11.

Без каких-либо изменений кода программы общее время работы составляет 99,08 с, что в 3,3 раза меньше, чем для версии, скомпилированной gcc.

Оптимизированная версия – предварительные вычисления. Выполним профилировку базовой версии, откомпилированной при помощи Intel C++ Compiler.

Описание функции	Имя функции	NT (gcc)	NT (Intel)	SelfTime
Моделирование r(t) (Шаг 2.2)	pow.L	442 342	114 993	73,40%
Генерация псевдослучайных чисел (Шаг 2.1)	log.L	18 541	8 496	5,42%
	__libm_sse2_sincos	3 900	4 167	2,66%
Вспомогательные вычисления	run	17 459 + 11 985	26 869	17,15%

Как видно из профиля, время, затраченное на вычисление pow, уменьшилось почти в 4 раза, время вычисления log также уменьшилось почти в 2 раза. Можно заметить, что из списка «исчезла» функция generateGaussian, при этом увеличилось время работы функции run. Это вызвано тем, что функция generateGaussian была встроена компилятором, что привело к уменьшению времени работы (17 459 + 11 985 > 26 869).

Рассмотрим фрагмент кода, выполнение которого занимает 73,4% времени работы программы.

```
r[0] = f0t; phi1 = 0; phi2 = 0; chi1 = 0;
for (i = 0; i < numSteps; i++) {
  dr = (kappa * (f0t - r[i]) + phi1 + phi2 + kappa * chi1) * dt +
     sigma1 * pow(r[i], alpha) * z1[i] + sigma2 * pow(r[i], beta) *z2[i];
  dphi1 = sigma1 * sigma1 * pow(r[i], 2 * alpha) * dt;
  dphi2 = (sigma2 * sigma2 * pow(r[i], 2 * beta) - 2 * kappa * phi2)*dt;
  dchi1 = phi1 * dt + sigma1 * pow(r[i], alpha) * z1[i];
  r[i + 1] = r[i] + dr; phi1 += dphi1; phi2 += dphi2; chi1 += dchi1;
}
```

Несложный анализ приведенного фрагмента показывает, что количество вызовов «тяжеловесной» функции pow избыточно: достаточно вычислить величины `pow(r[i], alpha)`, `pow(r[i], beta)` один раз и затем использовать вычисленные значения, а также их квадраты вместо `pow(r[i], 2*alpha)`, `pow(r[i], 2*beta)`. Кроме того, во всех вхождениях значение `pow(r[i], alpha)` умножается на `sigma1`, а `pow(r[i], beta)` – на `sigma2`, поэтому целесообразно вычислить данные произведения заранее. Ниже приведен усовершенствованный участок кода:

```
r[0] = f0t; phi1 = 0; phi2 = 0; chi1 = 0;
for (i = 0; i < numSteps; i++) {
  double c0, c1;
  c0 = sigma1 * pow(r[i], alpha); c1 = sigma2 * pow(r[i], beta);
  r[i+1] = r[i] + (kappa * (f0t - r[i]) + phi1 + phi2 + kappa * chi1)*dt +
     c0 * z1[i] + c1 * z2[i];
  chi1 += phi1 * dt + c0 * z1[i];
  phi1 += c0 * c0 * dt; phi2 += (c1 * c1 - 2 * kappa * phi2) * dt;
}
```

Общее время работы составляет 64,86 с, что в 1,5 раза быстрее предыдущей версии.

Оптимизированная версия – использование инструкций SIMD. Выполним профилирование разработанной версии программной реализации (столбец **Intel+**) для поиска путей дальнейшей оптимизации.

Описание функции	Имя функции	NT (Intel)	NT (Intel+)	SelfTime
Моделирование r(t) (Шаг 2.2)	pow.L	114 993	67 360	65,58%
Генерация псевдослучайных чисел (Шаг 2.1)	log.L	8 496	8 442	8,22%
	generateGaussian	встроена	8 813	8,58%
	__libm_sse2_sincos	4 167	4 318	4,20%
Вспомогательные вычисления	run	26 869	12 273	11,95%

В результате уменьшения количества вычислений суммарное время работы pow (**NT Intel+**) уменьшилось почти в 2 раза по сравнению с предыдущей версией (**NT Intel**). Время работы остальных функций практически не изменилось, существенное ускорение функции run является мнимым – компилятор решил не встраивать функцию generateGaussian. Исходя из результатов анализа текущей версии кода, на следующем этапе оптимизации было решено использовать векторизацию. Необходимо отметить, что многие циклы векторизуются компилятором Intel автоматически при включении опции -O2. При этом эвристика компилятора предсказывает, ожидается ли выигрыш от векторизации. Для устранения возможных ошибок прогноза существует директива **#pragma vector always**, действующая для ближайшего цикла.

Рассмотрим приведенный выше фрагмент кода. Очевидно, что цикл не может быть векторизован из-за большого количества зависимостей внутри итерации и между итерациями. Тем не менее, возможности для оптимизации все еще остаются.

Попробуем «вручную» векторизовать вызовы pow – самой трудоемкой функции в данной программе. Для этого создадим массивы из двух элементов и цикл из двух итераций. Иногда компилятор откажется от векторизации такого короткого цикла, поэтому воспользуемся директивой **#pragma vector always**. Ответ на вопрос о том, был ли векторизован цикл, можно получить из отчета компилятора о векторизации.

```
r[0] = f0t; phi1 = 0; phi2 = 0; chi1 = 0;
for (i = 0; i < numSteps; i++) {
  double c[2],sigma[2] = {sigma1, sigma2}, power[2] = {alpha, beta};
  int k;
  #pragma vector always
  for (k = 0; k < 2; k++)
     c[k] = sigma[k] * pow(r[i], power[k]);
  …
}
```

В результате изменений время работы данной версии (Intel++) составляет 54,8 с, что на 20% меньше, чем время работы предыдущей версии.

Оптимизированная версия – использование библиотек. Выполним профилировку для поиска возможностей дальнейшей оптимизации.

Описание функции	Имя функции	NT (Intel+)	NT (Intel++)	SelfTime
Моделирование r(t) (Шаг 2.2)	__svml_pow2.A	67 360	53 082	61,20%
Генерация псевдослучайных чисел (Шаг 2.1)	log.L	8 442	8 555	10,06%
	generateGaussian	8 813	8 726	9,86%
	__libm_sse2_sincos	4 318	4 313	4,22%
Вспомогательные вычисления	run	12 273	11 131	12,83%

Как видно из профиля, используется векторная функция вычисления степени *__svml_pow2.A*, ее время работы почти на 20% меньше, чем в предыдущей версии.

Практически все предыдущие оптимизации были направлены на уменьшение времени работы функции возведения в степень pow. В итоге генерация случайных чисел стала занимать существенное время от общего – около 25%, что является поводом для оптимизации и этого этапа вычислений, для чего используется оптимизированная под процессоры Intel библиотека Intel Math Kernel Library (MKL), в состав которой, в частности, входят генераторы псевдослучайных чисел. Так, для генерации нормально распределенных псевдослучайных чисел может быть использована функция vdRngGaussian. Рассмотрим профиль приложения после изменений.

Описание функции	Имя функции	NT (Intel+)	NT (Intel_MKL)	SelfTime
Моделирование r(t) (Шаг 2.2)	__svml_pow2.A	53 082	53 736	70,59%
Генерация псевдослучайных чисел (Шаг 2.1)	_vmldSinCos_26	21 594	10 420	13,69%
	_vdRngGaussianBoxMuller2			
	_vmldLn_26			
	_vmldSqrt_26			
	__vsldBRngMCG59			
Вспомогательные вычисления	run	11 131	10 829	14,23%

Как видно из профиля, время генерации случайных чисел уменьшилось примерно в 2 раза. Общее время работы составляет 48,35 с, что на 13% меньше, чем ранее.

Оптимизированная версия – параллельные вычисления. В заключение попробуем распараллелить наши вычисления для систем с общей памятью. Одним из наиболее простых подходов является применение технологии OpenMP, где для реализации параллельности необходимо лишь использовать дополнительные директивы компилятора. Рассматриваемое приложение выполняет расчет цен нескольких независимых опционов. На практике количество таких опционов может достигать нескольких тысяч,

при этом время расчета цены одного контракта одинаково, не зависит от цены исполнения. Поэтому распараллеливание произведено на уровне задач, как показано ниже.

```
#pragma omp parallel for
for (i = 0; i < numOptions; i++) {
  …
  getAveragePrice_MKL(&data_MKL_openMP[i]);
}
```

Время работы программы на 8 вычислительных ядрах составило 6,98 с, что в 46,87 раз меньше, чем в базовой версии.

Ниже приведены диаграммы, иллюстрирующие ускорение для параллельной версии (рис. 1) и результаты оптимизации (рис. 2).

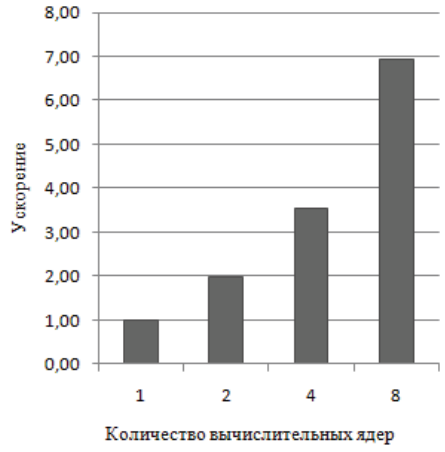

Рис. 1. Ускорение параллельной версии

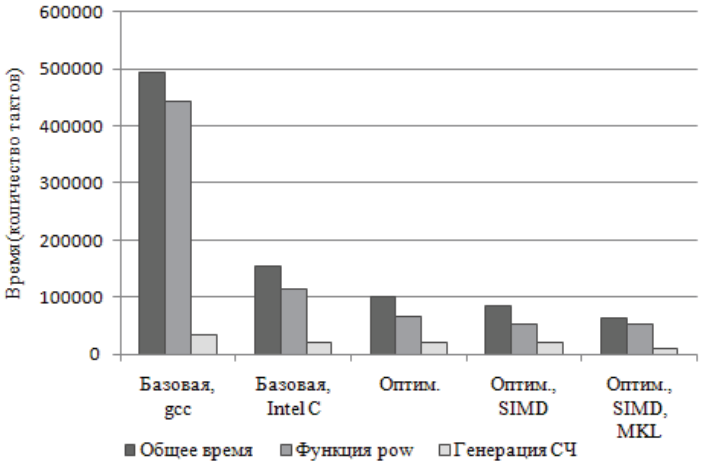

Рис. 2. Результаты оптимизации

Реализация на графическом процессоре. В настоящий момент в научной и научно-популярной литературе большое внимание уделяется программированию на графических процессорах (GPU). Основной акцент в таких публикациях часто делается на пиковую производительность GPU, многократно превосходящую пиковую производительность центрального процессора (CPU). При этом решаемые задачи чаще всего характеризуются большим количеством однотипных вычислений над однотипными данными. При таких условиях (идеальное распараллеливание) GPU, как правило, демонстрируют свою эффективность. В данной работе мы попробовали перенести имеющийся (достаточно хорошо распараллеливающийся) код на GPU с использованием NVidia CUDA, уделяя основное внимание производительности.

В выполненной реализации каждому опциону соответствует блок потоков (каждый из которых назначается на свой мультипроцессор): при этом каждый поток внутри блока выполняет свою часть итераций метода Монте-Карло. Такое распределение работы позволяет разместить необходимые для вычисления опциона общие данные в так называемой разделяемой (общей) памяти, имеющей низкую латентность. Роль центрального процессора в данном случае ограничивается подбором конфигурации запуска вычислительного ядра и копированием входных данных в память GPU, а выходных – обратно. Как и в случае реализации для CPU, использовались вычисления двойной точности. Время работы приложения на GPU составило 6,43 с, что сопоставимо с временем работы на CPU – 6,98 с (в обоих случаях рассмотрены параллельные реализации).

Конечно, одной задачи недостаточно, чтобы делать далеко идущие выводы. Однако идея переноса расчетных программ на GPU как способ быстрого и «бесплатного» увеличения их производительности представляется как минимум дискуссионной из-за следующих проблем. Оптимизация по скорости на GPU требует существенного опыта подобной работы и дополнительной подготовки: написание программ, которые эффективно используют преимущества архитектуры графического процессора без глубокого знания данной архитектуры является на сегодняшний день трудновыполнимым. Так, при наличии навыков работы с NVidia CUDA и работающего на CPU исходного кода перенос сравнительно несложного приложения (около 740 строк, включая исходные и заголовочные файлы с функциями вычислений, тестирования, ввода-вывода, подробными комментариями) потребовал времени, сравнимого со временем написания данного кода с нуля. Дополнительно положение осложняется отсутствием возможностей для полноценной отладки (в настоящий момент компания NVidia выпустила отладчик с ограниченной функциональностью). Архитектура GPU накладывает серьезные ограничения на схему распараллеливания; кроме того, чем больше сложность вычислительной части, тем менее эффективно удается использовать

вычислительную мощь устройства из-за относительной нехватки регистров и общей памяти. Таким образом, представляется, что эффективное использование GPU+NVidia CUDA возможно далеко не во всех задачах и требует наличия в коллективе опытных системных программистов.

Заключение

В работе на примере конкретной задачи исследуются вопросы оптимизации времени работы прикладного ПО для современных вычислительных систем с общей памятью. Суть предлагаемого подхода состоит в эффективном использовании алгоритмической, компиляторной, программной оптимизации, а также специализированных средств – оптимизирующих компиляторов, профилировщиков, математических библиотек. В результате применения рассмотренных методов и средств оптимизации время работы последовательной версии программы сократилось в 6,77 раза, а после распараллеливания – в 46,87 раза при выполнении на 8 вычислительных ядрах.

Работа выполнена в лаборатории «Информационные технологии» (ITLab) ВМК ННГУ.

Литература

1. Gerber R., Bik A., Smith K., Tian X. The Software Optimization Cookbook Second Edition. High Performance Recipes for IA 32 Platforms // Intel Press, 2005.
2. Fog A. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms. – 2009 [Электронный ресурс]. – Режим доступа: http://www.agner.org/optimize/optimizing_cpp.pdf, своб.
3. Касперски К. Техника оптимизации программ. Эффективное использование памяти. – СПб: BHV, 2003. – 464 с.
4. Халл Дж. Опционы, фьючерсы и другие производные финансовые инструменты. – М.: Вильямс, 2007. – 1056 с.
5. Inui K., Kijima M. A Markovian framework in multi-factor HJM models // Journal of financial and quantitative analysis. – 1998. – V. 33. – № 3. – P. 423–440.
6. Box G.E.P. and M.E. Muller. A Note on the Generation of Random Normal Deviates // Ann. Math. Stat. – 1958. – V. 28. – P. 610–611.
7. Intel MKL. Vector Statistical Library Notes, revision -019, section 8.4.4.

Бастраков Сергей Иванович – Нижегородский государственный университет им. Н.И. Лобачевского, студент, sergey.bastrakov@gmail.com

Донченко Роман Владимирович – Нижегородский государственный университет им. Н.И. Лобачевского, студент, ss.donchenko@itlab.unn.ru

Мееров Иосиф Борисович – Нижегородский государственный университет им. Н.И. Лобачевского, кандидат технических наук, доцент, mib@uic.nnov.ru

Половинкин Алексей Николаевич – Нижегородский государственный университет им. Н.И. Лобачевского, аспирант, alexey.polovinkin@gmail.com