National Research Lobachevsky State University of Nizhny Novgorod
oneAPI Center of Excellence

Black-Scholes Option Pricing on Intel CPUs and GPUs: Implementation on SYCL and Optimization Techniques

Elena Panova, Valentin Volokitin,
Anton Gorshkov, Iosif Meyerov

RuSCDays'2022
27.09.2022

Введение

- Задача Блэка-Шоулса вычисления справедливой цены Европейского опциона – известный финансовый бенчмарк
- Intel oneAPI – новая технология гетерогенного программирования
- План:
  - Выявить максимальную производительность C++ кода, распараллеленного с использованием OpenMP, на современных центральных процессорах Intel
  - Перенести код на Data Parallel C++ (DPC++)
  - Достигнуть приемлемой производительности DPC++ кода как на Intel CPU, так и на Intel GPU
  - Продемонстрировать некоторые общие приемы оптимизации

The work is supported by the oneAPI Center of Excellence program and by the Ministry of Science and Higher Education of the Russian Federation, project no. 0729-2020-0055

Тестовая инфраструктура (узел CPU)

Узел суперкомпьютера Endeavour	
CPU	2x Intel Xeon Platinum 8260L (Cascade Lake, 24 ядра, 2400 МГц, 6x DDR4-2933)
RAM	192 ГБ
Операционная система	Linux RedHat 4.8.5-39
Компилятор, профилировщик	Intel oneAPI 2022.1 • Intel C++ Compiler Classic • Intel oneAPI DPC++/C++ Compiler • Intel VTune Profiler • Intel Advisor

Тестовая инфраструктура (узел GPU)

Платформа Intel DevCloud	
CPU	Intel Core i9 10920X (3500 МГц)
GPU	Intel Iris Xe MAX (96 EU, 1650 МГц, 4 ГБ RAM, пропускная способность памяти 68 ГБ/сек)
Операционная система	Ubuntu 18.04.3 LTS
Компилятор, профилировщик	Intel oneAPI 2022.1 • Intel oneAPI DPC++/C++ Compiler • Intel VTune Profiler • Intel Advisor

Формула Блэка-Шоулса

Формула Блэка-Шоулса вычисления справедливой цены Европейского колл-опциона

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

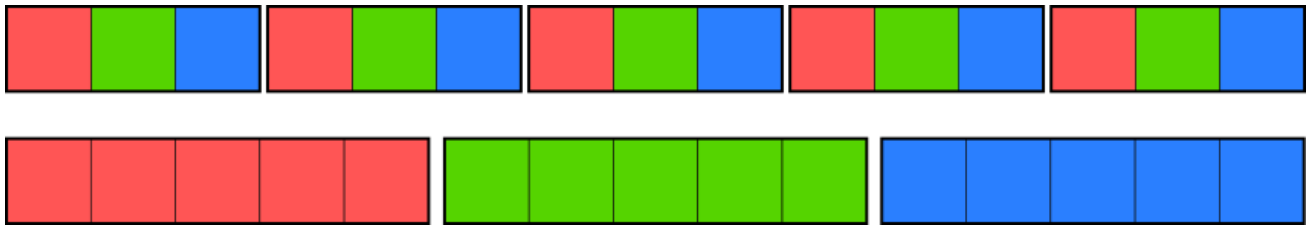
$$d_1 = \ln \frac{S_0}{K} + \frac{\left(r + \frac{\sigma^2}{2}\right) T}{\sigma \sqrt{T}}$$

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

- C – справедливая цена опциона
- S_0 – стартовая цена акции
- K – цена исполнения опциона
- T – время до экспирации опциона
- σ – волатильность
- r – безрисковая процентная ставка
- $F(x)$ - интегральная функция стандартного нормального распределения

Базовая версия C++

- Одинарная точность числа с плавающей запятой (*float*)
- Структура массивов



Array of Structures (AoS)

Structure of Arrays (SoA)

- Математические функции из стандартной библиотеки языка C++

$$F(x) = 0.5 + 0.5\,\mathrm{erf}\frac{x}{\sigma\sqrt{T}}$$

- 5 пачек по 240 000 000 опционов (3.8 ГБ)
- Время вычисления одной пачки ***3.062*** сек

```
const float sig, r;  // volatility, interest rate

void GetOptionPrices (
  float *pT,   // maturity (input)
  float *pK,   // strike price (input)
  float *pS0,  // initial stock price (input)
  float *pC,   // option price (output)
  int N        // number of options
){
  for (int i = 0; i < N; i++)
  {
    float d1 = (std::log(pS0[i] / pK[i]) +
      (r + 0.5f * sig * sig) * pT[i]) / (sig * std::sqrt(pT[i]));
    float d2 = (std::log(pS0[i] / pK[i]) +
      (r - 0.5f * sig * sig) * pT[i]) / (sig * std::sqrt(pT[i]));
    float erf1 = 0.5 + 0.5f * std::erf(d1 / std::sqrt(2.0f));
    float erf2 = 0.5 + 0.5f * std::erf(d2 / std::sqrt(2.0f));
    pC[i] = pS0[i] * erf1 - pK[i] * std::exp((-1.0f) * r * pT[i]) * erf2;
  }
}
```

Векторизация цикла

- Процессор поддерживает инструкции AVX512
- Опции компилятора **-xHost**, **-qopt-zmm-usage=high**
- **#pragma omp simd**

Версия	Время, с
Базовая версия	3.062
Векторизация цикла	*1.155*

- Почему ускорение только 2.65x?
- Базовая версия уже была векторизована
- SSE инструкции по умолчанию
- Почему компилятор не заподозрил зависимости по данным?

Отчет компилятора (базовая версия)

```
<Multiversioned v1>
   remark #25228: Loop multiversioned for Data Dependence
   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15305: vectorization support: vector length 4
   remark #15309: vectorization support: normalized vectorization overhead 0.049
   remark #15300: LOOP WAS VECTORIZED
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 823
   remark #15477: vector cost: 153.250
   remark #15478: estimated potential speedup: 5.070
   remark #15482: vectorized math library calls: 5
   remark #15486: divides: 4
   remark #15488: --- end vector cost summary ---
```

Компилятор подозревал зависимости по данным и создал ***2 версии кода***, скалярную и векторную. Решение о том, какую выбрать, определяется ***во время работы программы***

Цикл уже был векторизован

Длина векторного регистра 4 float элемента (xmm регистры, SSE)

Неточные вычисления

- Финансовые бенчмарки позволяют использовать неточные вычисления
- Быстрые вычисления математических функций
- Опции компилятора:
  - **-fimf-precision=low** (11 bits мантиссы)
  - **-fimf-domain-exclusion=31** (исключение особых значений математических функций, таких как *NaN*, *Inf*, …)

Версия	Время, с
Векторизация цикла	1.155
Неточные вычисления	*0.527*

- Менее точный результат (отличие в пятом десятичном знаке)

Параллелизм

- 2 процессора, 48 физических ядер
- OpenMP, **#pragma omp parallel for**
- Гипертрединг не влияет на производительность

Версия	Время, с
Неточные вычисления	0.527
Параллелизм	*0.053*

- Ускорение 10x
- «Узкое горлышко» приложения – пропускная способность памяти
- Можно ли увеличить производительность параллельного кода?

NUMA-friendly выделение памяти

- 2 процессора, *неравномерный доступ к памяти* (Non-Uniform Memory Access, NUMA)
- Память может быть выделена физически только на одном процессоре, что даст 50% медленного удаленного доступа
- NUMA-friendly инициализация:

```
#pragma omp parallel for
for (int i=0; i<N; i++) {
  // input arrays
  pT[i] = T0;  // T0, S0, K are constants
  pS0[i] = S0;
  pK[i] = K;
  // output array
  pC[i] = 0;
}
```

Результаты на CPU для C++ кода

Версия	Время, с
Базовая версия	3.062
Векторизация цикла	1.155
Неточные вычисления	0.527
Параллелизм	0.053
NUMA-friendly выделение памяти	*0.022*

- Суммарное ускорение параллельной версии 24x
- Общее ускорение после всех оптимизаций **140x**
- Согласно Intel Advisor, производительность ограничена пропускной способностью L3 кэша (roofline)

Перенос кода на DPC++ (CPU)

- SYCL позволяет не заботиться о векторизации и параллелизме
- Вычисление цены одного опциона – work item
- Встроенные в SYCL математические функции
- Buffers & Accessors подход для обмена данными между хостом и устройством
- Инициализация в отдельном ядре (NUMA-friendly подход)
- Переменная окружения **DPCPP_CPU_PLACES=numa_domains** (чтобы окончательно избавиться от удаленного доступа)

Версия	Время, с
Базовая DPC++ версия	0.057
NUMA-friendly инициализация	0.035
DPCPP_CPU_PLACES	0.024

10% замедление относительно оптимизированной C++ версии (0.022 сек)

```
void GetOptionPrices(float *pT, float *pK, float *pS0, float *pC, int N) {
  // buffer declaration
  sycl::buffer<float, 1> pTbuf(pT, sycl::range<1>(N));
  sycl::buffer<float, 1> pKbuf(pK, sycl::range<1>(N));
  sycl::buffer<float, 1> pS0buf(pS0, sycl::range<1>(N));
  sycl::buffer<float, 1> pCbuf(pC, sycl::range<1>(N));

  queue.submit([&](auto& handler) {
    // accessor declaration
    sycl::accessor pT(pTbuf, handler, sycl::read_only);
    sycl::accessor pK(pKbuf, handler, sycl::read_only);
    sycl::accessor pS0(pS0buf, handler, sycl::read_only);
    sycl::accessor pC(pCbuf, handler, sycl::write_only);
    // DPC++ kernel
    handler.parallel_for(sycl::range<1>(N), [=](sycl::id<1> i) { /**/ });
  }).wait();

  // data copying from device to host
  sycl::host_accessor pCacc(pCbuf);
  // here we need to process the result
}
```

Эксперименты на GPU

	Основное ядро, с	Инициали-зация, с	Накладные расходы, с	Общее время, с
Выделение памяти для каждой пачки отдельно	0.062	0.061	2.367	2.551
Общая память для всех пачек			0.748	0.932

- Производительность ограничена пропускной способностью памяти
- Следовательно, минимальное время работы ядра 0.056 sec (3.8 ГБ / 68 ГБ/сек); наблюдаемое время работы 0.061
- Откуда берутся накладные расходы? Синхронизация и обмен данными между хостом и устройством
- Можно ли уменьшить накладные расходы?

Управление памятью DPC++

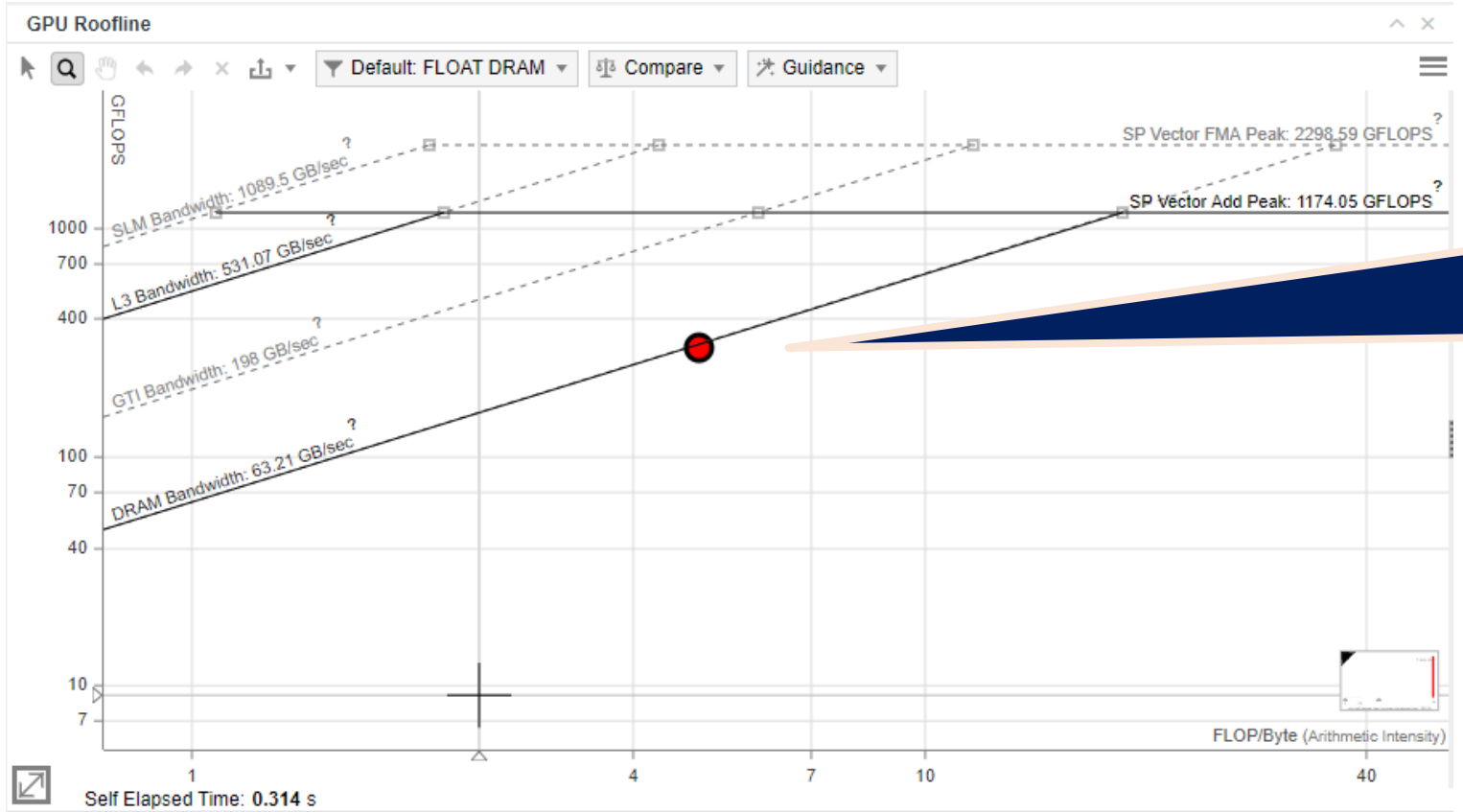
- Ранее мы использовали *Buffers & Accessors* подход
- Альтернатива – *Unified Shared Memory (USM)* подход
- USM: неявный и явный подходы

	Основное ядро, с	Инициали-зация, с	Накладные расходы, с	Общее время, с
Buffers & Accessors	0.062	0.061	0.748	0.932
USM, неявный подход			0.658	0.781
USM, явный подход			*0.369*	*0.492*

Некоторые общие приемы оптимизации на GPU

- Варьирование размера группы и числа потоков
  - Конфигурация по умолчанию оптимальная
- Использование векторных типов данных
  - *sycl::float4*
- Использование другого паттерна доступа к памяти
  - Structure of Arrays -> Array of Structures
  - Хранение входных и выходных данных в разных массивах
- Перечисленные оптимизации не дали результатов для данного бенчмарка.

GPU Roofline



Производительность DPC++ ядра ограничена пропускной способностью памяти (CPU – L3 кэш, GPU – DRAM)

GPU | Recommendations | GPU Source | GPU Assembly

No data locality

Confidence level: low

The GPU kernel is bounded by the **DRAM bandwidth.** Data is read from a slower memory as the GPU kernel processes more data than fits into GRF and GPU caches. This causes performance decrease. The cache line utilization is **100.0%**, which is good enough. It means an ineffective memory access is not the main reason of frequent memory requests. In general, using previously cached data multiple times may improve performance.

Паттерн доступа к памяти эффективен для GPU, но загрузок из памяти слишком много

Выводы

- На CPU оптимизированная DPC++ реализация формулы Блэка-Шоулса на 10% медленнее, чем оптимизированная OpenMP реализация
- На Intel GPU была получена ожидаемая производительность с учетом вычислительных характеристик устройства
- Производительность ограничена пропускной способностью памяти как на CPU, так и на GPU
- Накладные расходы на синхронизацию и обмен данными между хостом и GPU в 6 раз больше времени работы ядра
- Надеемся, что наш опыт будет полезен для других разработчиков, планирующих переносить вычислительный код на DPC++

Литература

1. Fischer Black, and Myron Scholes. "The pricing of options and corporate liabilities." *World Scientific Reference on Contingent Claims Analysis in Corporate Finance: Volume 1: Foundations of CCA and Equity Valuation*. 2019. 3-21.
2. Meyerov, I., et al. "Performance optimization of Black-Scholes pricing." *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*. 2014. 319-340.
3. Reinders, James, et al. *Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL*. Springer Nature, 2021.