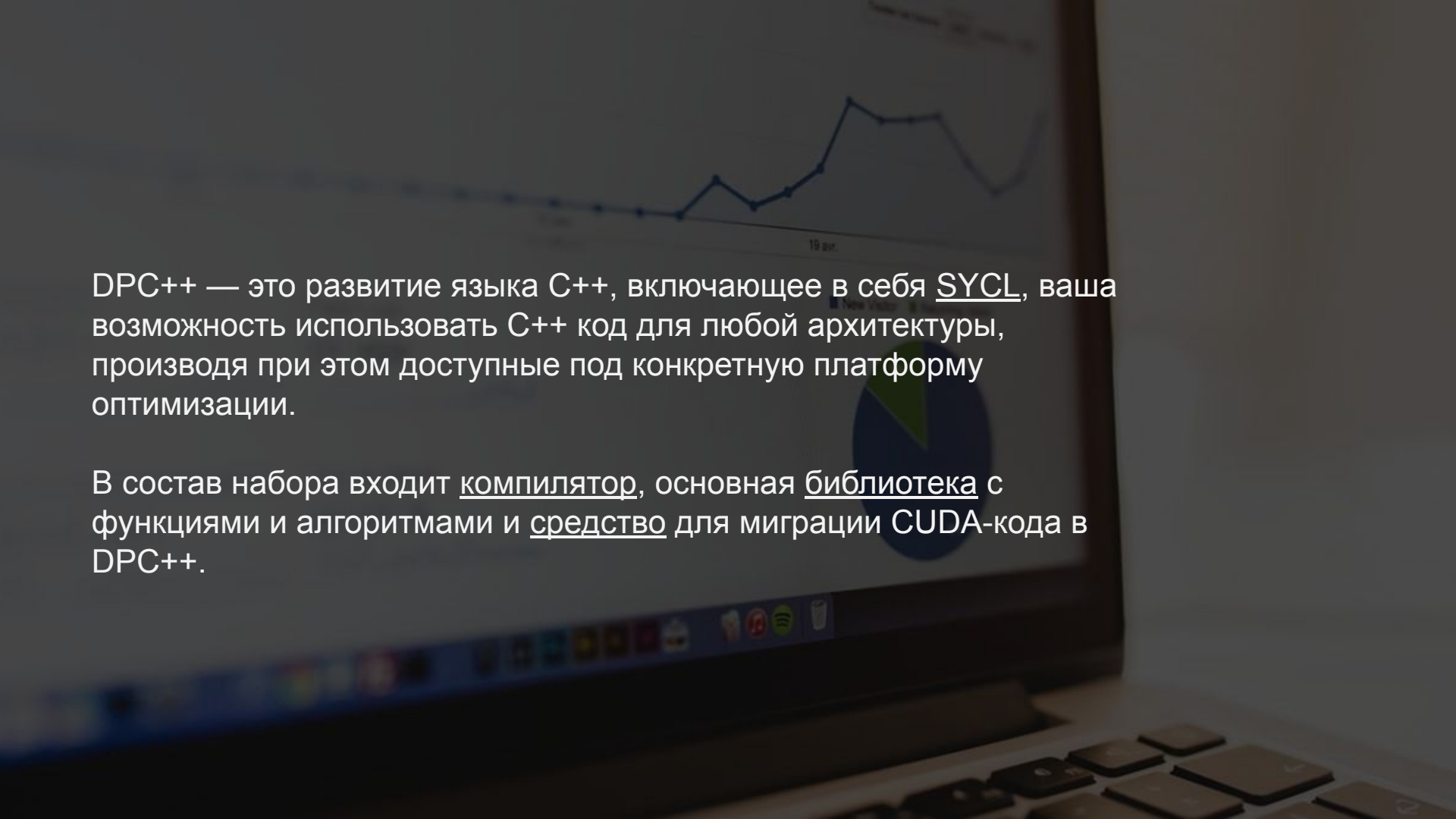
Возможности нового языка Data Parallel C++ (DPC++) от Intel

Студент: Койнов В.В., ММФ НГУ
Руководитель: Киреев С.Е., ИВМиМГ



DPC++ — это развитие языка C++, включающее в себя SYCL, ваша возможность использовать C++ код для любой архитектуры, производя при этом доступные под конкретную платформу оптимизации.

В состав набора входит компилятор, основная библиотека с функциями и алгоритмами и средство для миграции CUDA-кода в DPC++.

Отправка команд

```
// Отправка команды в очередь (здесь инициализируется матрица S)
  deviceQueue.submit([&](cl::sycl::handler &cgh) {

        // Получение аксессора ( доступа к буфферу) на запись на устройстве
        auto A = bufferA.template get_access<sycl_read>(cgh);
        auto B = bufferB.template get_access<sycl_read>(cgh);
        auto X = bufferX.template get_access<sycl_write>(cgh);

        // Выполнение ядра (device code)
        cgh.parallel_for<class simple_vector_add<T>>(numOfItems, [=](cl::sycl::id<1> wiID) {

              // Запись через аксессор
              X[wiID] = A[wiID] + B[wiID];
        });});
```

Последовательность посылок

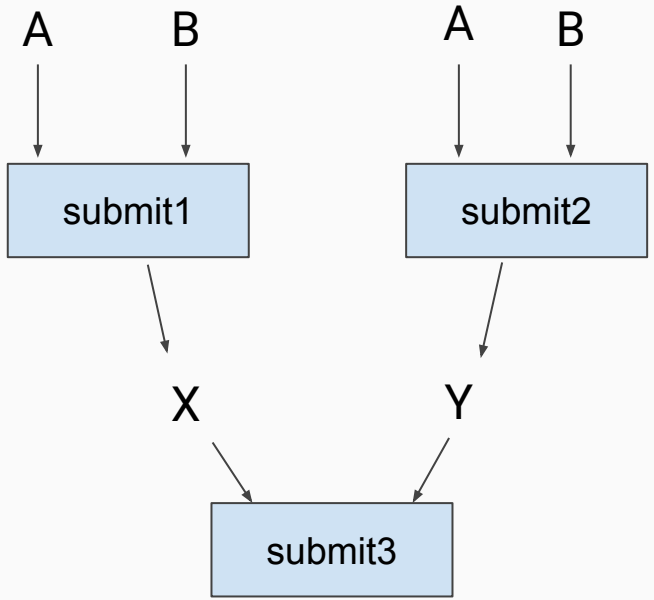
- Каждый аксессор внутри submit принимает информацию о типе доступа к памяти
- Можно ли использовать эту информацию для последовательности посылок?

```
// Читает A, B пишет X
deviceQueue.submit([&](cl::sycl::handler &cgh) { /* …. */ }

// Читает A, B пишет Y
deviceQueue.submit([&](cl::sycl::handler &cgh) { /* …. */ }

// Читает X, Y пишет C
deviceQueue.submit([&](cl::sycl::handler &cgh) { /* …. */ }
```

https://github.com/tilir/ocl/blob/master/sycl/sycl_vectoradd_deps.cc

Граф зависимостей

- Явная работа с аксессорами создает явный граф зависимости по работе с памятью
- SYCL по спецификации обязан строить этот граф
- В OpenCL для этого пришлось бы брать events
- SYCL избавляет от кучи ручного труда



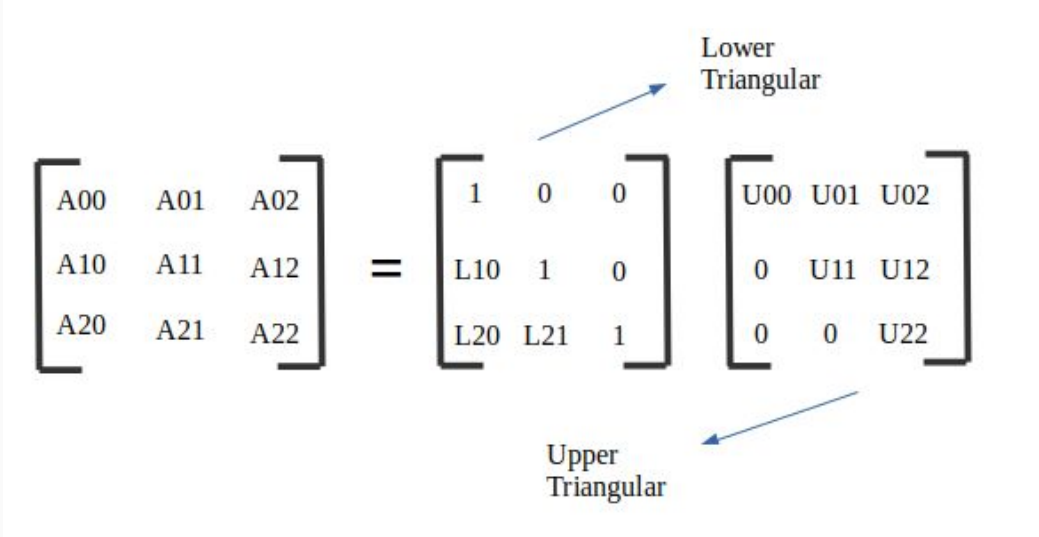
Постановка задачи

Возможности нового языка
Data Parallel C++ (DPC++)
от Intel

- изучить модель программирования
- реализовать ряд тестовых задач
- проанализировать производительность

Тестовая задача: LU-разложение

Многие приложения требуют решения системы Ax = b. LU-разложение - эффективная технология для решения. Матрица A раскладывается на верхнюю (U) и нижнюю (L) матрицу так, что A = LU.



Алгоритм LU-разложения. Неблочная версия

```
for k←0 to n−1 do
        // Division step
        for i←k+ 1 to n−1 do
                a[i][k]←a[i][k] / a[k][k]

        // Elimination step
        for i←k+ 1 to n−1 do
                for j←k+ 1 to n−1 do
                        a[i][j]←a[i][j] − a[i][k] ∗ a[k][j]
```

Последовательный алгоритм LU-разложения

Алгоритм LU-разложения. Неблочная версия

```
for k←0 to n−1 do
        // Division step
        #pragma omp parallel for
        for i←k+ 1 to n−1 do
                a[i][k]←a[i][k] / a[k][k]

        // Elimination step
        #pragma omp parallel for
        for i←k+ 1 to n−1 do
                for j←k+ 1 to n−1 do
                        a[i][j]←a[i][j] − a[i][k] ∗ a[k][j]
```

OpenMP распараллеливание по строкам (RowBlock) алгоритма LU-разложения

https://github.com/0x/matrix_lu/blob/master/src/matrix_lu_omp.cpp

Алгоритм LU-разложения. Неблочная версия

```
for k←0 to n−1 do
        // Division step
        deviceQueue.submit([&](cl::sycl::handler &cgh) { /* …. */ }

        // Elimination step
        deviceQueue.submit([&](cl::sycl::handler &cgh) { /* …. */ }
```

DPC++ распараллеливание по строкам (RowBlock) алгоритма LU-разложения

https://github.com/0x/matrix_lu/blob/master/src/matrix_lu_dpcpp.cpp

Выбор SYCL устройства (device)

Ключи для выбора устройства:

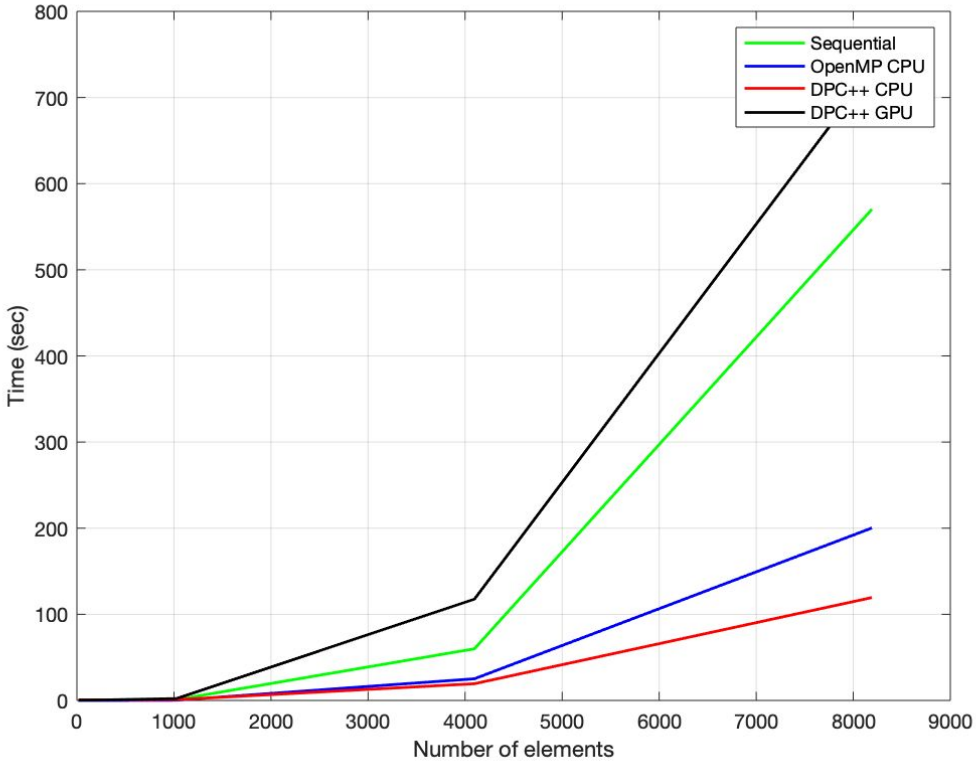
- cl::sycl::intel::fpga_selector
- cl::sycl::intel::fpga_emulator_selector
- cl::sycl::cpu_selector
- cl::sycl::gpu_selector

```
// Инициализация очереди устройства.
// Используется для постановки ядра в очередь.
queue q(d_selector, dpc_common::exception_handler);
```

Неблочное LU-разложение. Тестирование

Device	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz (8 ядер)	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz (8 ядер)	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz (8 ядер)	Intel(R) Gen9 HD Graphics NEO (Intel UHD Graphics 620)
Size \ Method	Sequential program	OpenMP, sec	DPCPP, sec	DPCPP, sec
a(16, 16)	5.231e-06	0.000631678	0.633203	0.437078
a(64, 64)	0.00023276	0.00105018	0.633351	0.439744
a(256,256)	0.0135918	0.00569035	0.632432	0.522781
a(1024,1024)	0.915733	0.259853	0.920819	2.16231
a(4096,4096)	59.9594	25.1494	19.4197	117.609
a(8192,8192)	570.248	200.233	119.451	731.798

Неблочное LU-разложение. Тестирование



алгоритм блочного варианта LU-разложения

```
for i←0 to num_blocks−1 do
        proc_lu(block_size, a_glob[i][i])

        for k←i+1 to num_blocks−1 do
                proc_u(block_size, a_glob[i][i], a_glob[i][k])

         for j←i+1 to num_blocks−1 do
                proc_l(block_size, a_glob[i][i], a_glob[j][i])

                for l←i+1 to num_blocks−1 do
                proc_g(block_size, a_glob[j][i], a_glob[i][l], a_glob[j][l])
        }
 }
```

Сравнение DPC++ и OpenMP

```
// in: al[ny][n],au[n][nx], inout: ag[ny][nx] : ag -= al*au.
void proc_g(const int ny, const int nx, const int n, double *al1d, double
*au1d, double *ag1d) {
  buffer al(al1d, range(ny, n));
  buffer au(au1d, range(n, nx));
  buffer ag(ag1d, range(ny, nx));

  q.submit([&](handler &h) {
          // Read from al and au, write to ag.
          auto AL = al.get_access<access::mode::read>(h);
          auto AU = au.get_access<access::mode::read>(h);
          auto AG = ag.get_access<access::mode::read_write>(h);

          // Execute kernel.
          h.parallel_for(range(ny, nx), [=](id<2> index) {
                  // Get global position in Y and X direction.
                  int row = index[0];
                  int col = index[1];

                  // Compute the result of one element of AG.
                  for (int k = 0; k < n; k++) {
                    AG[row][col] -= AL[row][k] * AU[k][col];
                  } }); });
}
```

```
// in: al[ny][n],au[n][nx], inout: ag[ny][nx] : ag -= al*au.
void proc_g(const int ny,const int nx,const int n,double *al1d,double
*au1d,double *ag1d) {

#pragma omp task depend(in:al1d,au1d) depend(inout:ag1d)
 {
          auto al = (double (*)[ny][n]) al1d;
          auto au = (double (*)[n][nx]) au1d;
          auto ag = (double (*)[ny][nx]) ag1d;

          for (int j = 0; j < ny; j++)
             for (int i = 0; i < n; i++) {
                for (int k = 0; k < nx; k++) {
                   (*ag)[j][k] -= (*al)[j][i] * (*au)[i][k];
                }
}
}
```

https://github.com/0x/matrix_lu/blob/master/src/matrix_lu_block_dpcpp_func.cpp

OpenMP GPU offload

```
// in: al[ny][n],au[n][nx], inout: ag[ny][nx] : ag -= al*au
void proc_g(const int ny, const int nx, const int n, double *AL, double *AU, double *AG)
{

        // Parallelize on target device.
        #pragma omp target teams distribute parallel for map(to : AL[0:ny*n], AU[0:n*nx]) \
                map(tofrom : AG[0:ny*nx]) thread_limit(128)
        {
                for (int j = 0; j < ny; j++)
                    for (int i = 0; i < n; i++)
                        for (int k = 0; k < nx; k++) {

                                AG[j * ny + k] -= AL[j * ny + i] * AU[i * nx + k];
                        }
        }
}
```

LU-разложение, блочный вариант. Выбор оптимальных параметров

Device	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz (8 ядер)	Intel(R) Gen9 HD Graphics NEO (Intel UHD Graphics 620)	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz (8 ядер)	Intel(R) Gen9 HD Graphics NEO (Intel UHD Graphics 620)
Size \ Method	OpenMp	OpenMp	DPCPP	DPCPP
a(256,256)	1 / 256	1 / 256	1 / 256	1 / 256
a(512,512)	1 / 512	1 / 512	1 / 512	1 / 512
a(1024,1024)	4 / 256	1 / 1024	1 / 1024	4 / 256
a(2048,2048)	1 / 2048	4 /512	1 / 2048	16 / 128
a(4096,4096)	1 / 4096	16 / 256	16 / 256	16 / 256

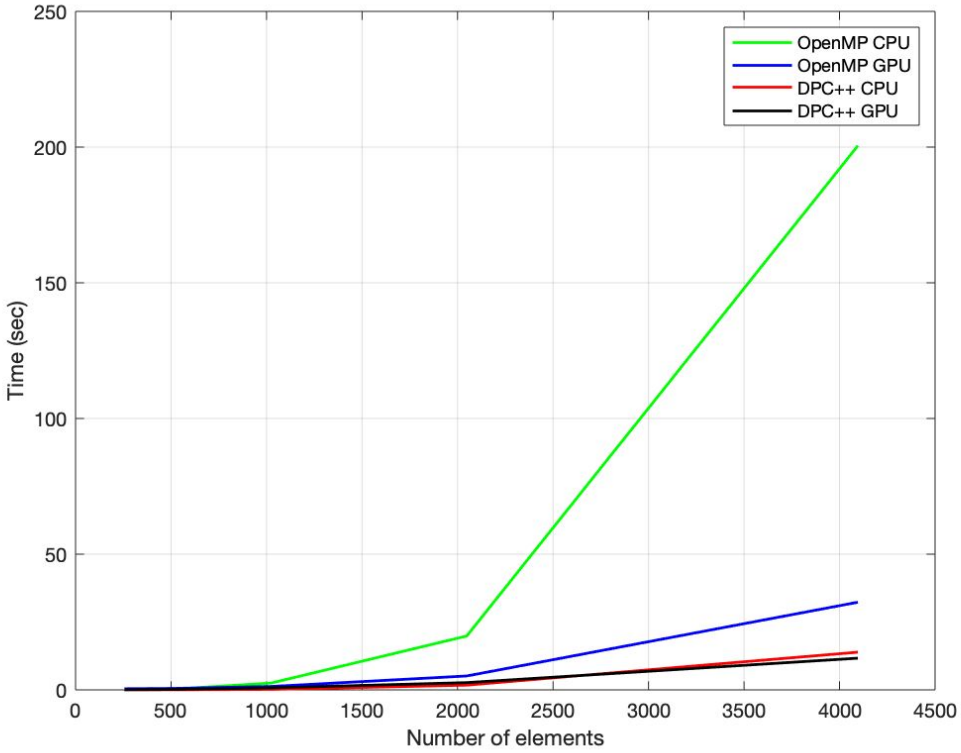
Количество блоков

Кол-во блоков / Размер блока

Тестирование.
LU-разложение, блочный вариант

Device	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz (8 ядер)	Intel(R) Gen9 HD Graphics NEO (Intel UHD Graphics 620)	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz (8 ядер)	Intel(R) Gen9 HD Graphics NEO (Intel UHD Graphics 620)
Size \ Method	OpenMp, sec	OpenMp, sec	DPCPP, sec	DPCPP, sec
a(256,256)	0.032967	0.418325	0.0119641	0.0661128
a(512,512)	0.27489	0.53109	0.0387406	0.183716
a(1024,1024)	2.50786	1.25357	0.18235	0.744324
a(2048,2048)	19.845	5.11883	1.7373	2.67311
a(4096,4096)	200.507	32.2993	13.8899	11.6821

Тестирование.
LU-разложение, блочный вариант



Выводы

Познакомился с возможностями DPC++, а также технологиями task и offloading OpenMP.

Реализовал параллельные варианты программы LU-разложения, используя DPC++, OpenMP на CPU и GPU.

Провел тестирования разработанных программ.

На примере задачи LU-разложения было показано, что версия неблочной программы с DPC++ почти в два раза превосходит версию с OpenMP на CPU с размером области a(4096, 4096) элементов.

Также результаты подтверждают преимущества DPC++ в кроссплатформенности и простоте использования для гетерогенных архитектур.

Спасибо за внимание!

Вопросы или предложения,
email: koynov95@gmail.com