Математика и Математическое моделирование
Сетевое издание
МГТУ им. Н.Э. Баумана
http://mathmjournal.ru

Ссылка на статью:
// Математика и Математическое моделирование. МГТУ им. Н.Э. Баумана. Электрон. журн. 2016. № 01. С. 1–27.

DOI: **10.7463/mathm.0116.0841079**

Представлена в редакцию: 04.01.2016
Исправлена: 18.01.2016

УДК 519.688

О применении технологий параллельного программирования для задач матричной алгебры в приложении к спектральному методу анализа, синтеза и идентификации систем управления

Клешнин В. Ю.[1], **Рыбаков К. А.**[1,*]

[*]rkoffice@mail.ru

[1]Московский авиационный институт (национальный исследовательский университет), Москва, Россия

В статье описываются библиотеки матричной алгебры, разработанные с применением современных технологий параллельного программирования для расчетной системы Spectrum. Расчетная система Spectrum может применяться в разных задачах анализа, синтеза и идентификации детерминированных и стохастических динамических систем спектральным методом (в спектральной форме математического описания). Проведен сравнительный анализ разработанных библиотек на основе вычислительных экспериментов с применением различного аппаратного обеспечения (центральных процессоров и графических процессоров видеоадаптеров).

Ключевые слова: матричная алгебра, параллельное программирование, параллельные вычисления, спектральный метод

Введение

Спектральный метод применяется для решения различных задач теории управления: задач анализа, синтеза и идентификации детерминированных и стохастических динамических систем [1–4]. Основу спектрального метода составляет представление функций спектральными характеристиками – векторами, образованными коэффициентами разложения этих функций по ортонормированным функциям. Операторы (звенья систем управления) представляются матрицами. Для автоматизации решения задач спектральным методом разработана расчетная система Spectrum [5].

Базовые операции, выполняемые при использовании спектрального метода, – операции с матрицами больших размеров и целью работы является дополнение расчетной системы Spectrum библиотеками функций работы с матрицами, использующими современ-

ные технологии параллельного программирования для центральных процессоров и графических процессоров видеоадаптеров. Для решения задач спектральным методом необходимо реализовать следующие функции: сложение, вычитание, умножение на число, умножение, возведение в степень с натуральным показателем, нахождение коммутатора, тензорное умножение, транспонирование, нахождение обратной матрицы, вычисление определителя, нахождение решения системы линейных уравнений.

Расчетная система Spectrum разработана в среде программирования Borland Delphi (для текущей версии используется Borland Delphi 7). При разработке дополнительных библиотек использовались различные программные средства: компиляторы C++ из пакетов Microsoft Visual Studio Express 2012 с дополнением nVidia CUDA Toolkit 6.5, Intel Parallel Studio XE 2013; среда программирования Embarcadero Delphi 2010 с библиотекой OmniThreadLibrary 3.02; среда программирования Borland Delphi 7 с библиотекой OpenCL for Delphi 1.2 [6–15].

Для расчетной системы Spectrum были разработаны и протестированы следующие библиотеки с перечисленными выше функциями:

1. Библиотека на Object Pascal / Delphi, среда программирования Embarcadero Delphi 2010 с библиотекой OmniThreadLibrary 3.02.
2. Библиотека на C/C++ с директивами OpenMP, компилятор из пакета Microsoft Visual Studio Express 2012.
3. Библиотека на C/C++ с директивами OpenMP, оптимизирующий компилятор из пакета Intel Parallel Studio XE 2013.
4. Библиотека на C++, использующая технологию Intel Threading Building Blocks, оптимизирующий компилятор из пакета Intel Parallel Studio XE 2013.
5. Библиотека на C++, использующая технологию Intel Cilk Plus, оптимизирующий компилятор из пакета Intel Parallel Studio XE 2013.
6. Библиотека на специализированном расширении C, использующая технологию nVidia CUDA, среда программирования Microsoft Visual Studio Express 2012 с дополнением nVidia CUDA Toolkit 6.5.
7. Библиотека на Object Pascal / Delphi и специализированном расширении C, среда программирования Borland Delphi 7 с библиотекой OpenCL for Delphi 1.2.
8. Библиотека на C++, использующая технологию Microsoft Accelerated Massive Parallelism, компилятор из пакета Microsoft Visual Studio Express 2012.

Перечисленные библиотеки поддерживают матрицы, элементы которых – действительные числа с плавающей запятой одинарной и двойной точности. Размеры матриц ограничены только моделью памяти для 32-битных или 64-битных приложений и конфигурацией компьютера (объемом оперативной памяти, настройками виртуальной памяти).

1. Методика тестирования

Для апробации разработанных библиотек матричной алгебры проводились тесты отдельных матричных операций с квадратными матрицами размеров 1024x1024, 2048x2048, 4096x4096, т.е. матриц размеров $n \times n$ для $n = 1024$, $n = 2048$ и $n = 4096$. Элементы матриц были получены с помощью генератора псевдослучайных чисел, имеющих равномерное распределение на отрезке [0, 1]. Время замерялось при выполнении операций сложения, умножения на число, умножения, транспонирования, нахождения обратной матрицы. Кроме того, проводились смешанные тесты и расчеты для прикладных задач, в частности были решены задачи анализа электрических цепей (RC- и RCL-цепей) с учетом непрерывных и импульсных случайных воздействий [16], а также задачи идентификации стохастических систем управления [17].

Для оценки эффективности тестирование проводилось на различных компьютерах с многоядерными процессорами Intel Core 2 Duo, Core 2 Quad; Intel Core i3, i5, i7; AMD E-Series, Athlon 64, Phenom II. При тестировании библиотек для графических процессоров использовались видеоадаптеры на базе nVidia, AMD и Intel (для Intel – интегрированные в центральный процессор).

Дальнейшие примеры программ и результаты тестов будут приводиться для операции умножения матриц, требующей больше всего вычислительных ресурсов по отношению к другим операциям, перечисленным выше за исключением нахождения коммутатора. Напомним, что произведением матрицы $A = [a_{ij}]$ размеров $w \times q$ и матрицы $B = [b_{ij}]$ размеров $q \times h$ называется такая матрица $C = [c_{ij}]$ размеров $w \times h$, что

$$c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}, \quad i = 1,...,w, \quad j = 1,...,h.$$

Предполагается, что матрицы хранятся в памяти по строкам, начиная с первой, т.е., например, для результирующей матрицы C имеем:

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1h} \\ c_{21} & c_{22} & \cdots & c_{2h} \\ \vdots & \vdots & \ddots & \vdots \\ c_{w1} & c_{w2} & \cdots & c_{wh} \end{bmatrix} \rightarrow [c_{11}\ c_{12} \ldots c_{1h}\ c_{21}\ c_{22} \ldots c_{2h} \ldots c_{w1}\ c_{w2} \ldots c_{wh}].$$

При обращении к памяти для удобства нумерация индексов начинается с нуля, т.е. элементу c_{ij} соответствует смещение $(ih + j)\,s$ относительно начального адреса блока памяти, выделяемого для хранения матрицы, где s – объем памяти для хранения одного элемента матрицы – числа с плавающей запятой ($s = 4$ байта при выборе одинарной точности, $s = 8$ байтов при выборе двойной точности).

Ниже (см. листинг 1) приведен фрагмент модуля поддержки операций с матрицами на языке Object Pascal / Delphi, а именно процедура умножения.

Эквивалентная функция на языке C (если не оговорено особо, то для программ на C/C++ использовался компилятор из пакета Microsoft Visual Studio Express 2012) приведена в листинге 2. Отметим, что все приведенные в статье листинги программ используют тип чисел с плавающей запятой двойной точности, при переходе к одинарной точности требуется заменить тип Double на Single для программ на Object Pascal / Delphi или double на float для программ на C/C++.

Листинг 1. Умножение матриц на языке Object Pascal / Delphi

```
type
  TDataArray = array [Word] of Double;
  PDataArray = ^TDataArray;

procedure Multiply(A, B, C: PDataArray; W, Q, H: Integer);
var
  I, J, K: Integer;
  X: Double;
begin
  for I:=0 to W - 1 do
    for J:=0 to H - 1 do
    begin
      X:=0;
      for K:=0 to Q - 1 do
        X:=X + A^[I * Q + K] * B^[K * H + J];
      C^[I * H + J]:=X;
    end;
end;
```

Листинг 2. Функция умножения матриц на языке C/C++

```
void Multiply(double* a, double* b, double* c, int w, int q, int h)
{
  for (int i = 0; i < w; i++)
    for (int j = 0; j < h; j++)
    {
      double x = 0;
      for (int k = 0; k < q; k++)
        x += a[i * q + k] * b[k * h + j];
      c[i * h + j] = x;
    }
}
```

Время счета приведено в табл. 1 (для листингов 1 и 2 результаты аналогичны), причем в этой таблице и во всех последующих для каждого процессора отдельно указано время (часы : минуты : секунды), соответствующее одинарной точности (первая строка) и двойной точности (вторая строка). Кроме того, указана достигнутая максимальная производительность – количество операций с числами с плавающей запятой в секунду.

Таблица 1. Результаты тестирования библиотеки на языке Object Pascal / Delphi (среда программирования Borland Delphi 7)

	GFLOPS	$n = 1024$	$n = 2048$	$n = 4096$
AMD E-450	0.020	0:01:48.272	0:17:35.451	2:32:17.029
	0.017	0:02:03.665	0:18:50.181	2:40:36.279
AMD Athlon 64 X2 4000+	0.065	0:00:36.947	0:04:24.949	1:06:46.514
	0.064	0:00:33.336	0:07:06.531	2:07:55.121
AMD Phenom II N830	0.074	0:00:29.191	0:04:00.159	0:43:59.796
	0.072	0:00:29.636	0:05:30.559	1:29:25.206
AMD Phenom II X4 925	0.136	0:00:15.797	0:03:26.824	0:41:16.596
	0.117	0:00:18.291	0:04:43.519	1:19:22.738
Intel Core 2 Duo T9300	0.417	0:00:05.156	0:03:35.170	0:29:38.137
	0.270	0:00:07.944	0:03:35.934	0:32:51.572
Intel Core 2 Quad Q9300	0.057	0:00:41.458	0:05:00.087	0:44:23.449
	0.071	0:00:30.096	0:05:38.861	0:49:01.142
Intel Core i3 2120	0.169	0:00:12.731	0:01:47.398	0:15:54.604
	0.175	0:00:12.245	0:01:58.982	0:18:44.677
Intel Core i5 3470	0.281	0:00:07.647	0:01:32.930	0:14:09.309
	0.212	0:00:10.108	0:01:44.188	0:17:10.250
Intel Core i7 2600	0.311	0:00:06.897	0:01:26.959	0:13:53.080
	0.233	0:00:09.201	0:01:39.830	0:15:48.885
Intel Core i7 4930K	0.342	0:00:06.271	0:01:15.128	0:10:58.834
	0.342	0:00:06.272	0:01:03.094	0:13:53.570

Приведенные данные характеризуют модель однопоточных вычислений, они нужны для сравнения с основными результатами работы.

2. Применение библиотеки OmniThreadLibrary

Вероятно, самый простой путь перейти к многопоточным вычислениям для программ на Object Pascal / Delphi – это использование библиотеки OmniThreadLibrary. Первая версия этой библиотеки появилась в 2010 г., она является высокоуровневой надстройкой над средствами операционной системы для создания и управлениями потоками, предоставляя разработчику много возможностей, из которых для библиотеки матричной алгебры достаточно использовать только конструкцию Parallel.ForEach вместо оператора цикла **for** (см. листинг 3). Значительных изменений программа с однопоточной моделью вычислений не требует. Библиотека OmniThreadLibrary может быть использована только в средах разработки, начиная с Delphi 2007, однако для расчетной системы Spectrum используется более ранняя версия Delphi и OmniThreadLibrary была задействована посредством DLL (DLL – динамически загружаемая библиотека для операционной системы Microsoft Windows).

Листинг 3. Процедура умножения матриц на языке Object Pascal / Delphi с использованием библиотеки OmniThreadLibrary (тип PDataArray определяется в листинге 1)

```
1: uses
2:   OtlCommon, OtlCollections, OtlParallel;
3:
4: procedure Multiply(A, B, R: PDataArray; W, Q, H: Integer);
5: begin
6:   Parallel.ForEach(0, W - 1).Execute(
7:     procedure (const I: Integer)
8:     var
9:       J, K, Z: Integer;
10:      S: Double;
11:    begin
12:      Z:=I * Q;
13:      for J:=0 to H - 1 do
14:      begin
15:        S:=0;
16:        for K:=0 to Q - 1 do S:=S + A^[Z + K] * B^[K * H + J];
17:        R^[I * H + J]:=S;
18:      end;
19:    end
20:  )
21: end;
```

Таблица 2. Результаты тестирования библиотеки на языке Object Pascal / Delphi (среда программирования Embarcadero Delphi 2010 с библиотекой OmniThreadLibrary 3.02)

	GFLOPS	***n* = 1024**	***n* = 2048**	***n* = 4096**
AMD E-450	0.037	0:00:57.668	0:09:05.793	1:16:05.153
	0.032	0:01:06.656	0:10:35.383	1:24:28.898
AMD Athlon 64 X2 4000+	0.088	0:00:26.526	0:03:15.898	0:33:19.241
	0.090	0:00:23.796	0:04:02.260	0:58:11.001
AMD Phenom II N830	0.157	0:00:13.718	0:01:58.034	0:17:43.898
	0.177	0:00:12.101	0:02:03.685	0:29:11.185
AMD Phenom II X4 925	0.513	0:00:04.183	0:01:09.532	0:12:46.592
	0.408	0:00:05.264	0:01:29.726	0:21:25.512
Intel Core 2 Duo T9300	0.268	0:00:07.999	0:02:13.316	0:18:44.263
	0.242	0:00:08.866	0:02:13.695	0:20:27.526
Intel Core 2 Quad Q9300	0.058	0:00:37.035	0:05:18.439	0:42:46.811
	0.079	0:00:27.138	0:05:15.308	0:42:58.996
Intel Core i3 2120	0.302	0:00:07.111	0:01:04.768	0:09:18.160
	0.338	0:00:06.354	0:01:08.634	0:09:51.096
Intel Core i5 3470	1.110	0:00:01.934	0:00:25.535	0:04:45.575
	0.985	0:00:02.180	0:00:27.273	0:05:15.233
Intel Core i7 2600	1.215	0:00:01.767	0:00:19.089	0:04:41.404
	1.186	0:00:01.811	0:00:20.844	0:04:56.470
Intel Core i7 4930K	1.965	0:00:01.093	0:00:09.649	0:02:54.148
	1.949	0:00:01.102	0:00:09.971	0:03:00.188

Рост производительности прямо пропорционален числу потоков, которое одновременно может обрабатывать центральный процессор. Но переход с однопоточной модели вычислений к *N*-поточной не означает ускорение в *N* раз. Это ускорение зависит от используемого центрального процессора. Например, использование четырехядерного процессора Intel Core 2 Quad Q9300 не дает ощутимого преимущества, в то время как использование четырехядерных процессоров AMD Phenom II X4 925 и Intel Core i5 3470 обеспечивает рост производительности почти в 4 раза.

3. Применение технологии OpenMP

OpenMP (Open Multi-Processing) – открытый стандарт написания многопоточных программ, созданный в 1997 г. Он состоит из набора библиотечных функций и директив компилятора, позволяя путем вставки директив компилятора в ключевые места исходной программы, например, перед оператором цикла для уже существующих однопоточных приложений, достаточно легко создавать многопоточные приложения на C/C++ и Fortran. Данный стандарт является одним из самых распространенных и поддерживается большинством современных компиляторов. Программы, использующие директивы OpenMP, совместимы и с компиляторами, которые не поддерживают данный стандарт (директивы многопоточности игнорируются), что делает эту технологию наиболее удобной и простой в использовании из всех рассмотренных.

Пример функции умножения матриц на C/C++ приведен в листинге 4. Он отличается от листинга 2 только одной строкой 3: директивой #pragma omp parallel for. Из этого примера хорошо видно, насколько просто адаптировать библиотеку матричной алгебры к многопоточным вычислениям на основе технологии OpenMP.

Библиотека матричной алгебры для расчетной системы Spectrum разработана на языке C/C++ с применением директив OpenMP, подключение к Spectrum осуществляется посредством механизмов DLL, как и для библиотек, описанных ниже.

Листинг 4. Умножение матриц на языке C/C++ с возможностью автоматизированного распараллеливания внешнего цикла (OpenMP)

```
1: void Multiply(double* a, double* b, double* c, int w, int q, int h)
2: {
3:   #pragma omp parallel for
4:   for (int i = 0; i < w; i++)
5:     for (int j = 0; j < h; j++)
6:     {
7:       double x = 0;
8:       for (int k = 0; k < q; k++)
9:         x += a[i * q + k] * b[k * h + j];
10:      c[i * h + j] = x;
11:    }
12: }
```

Таблица 3. Результаты тестирования библиотеки на языке C/C++ (OpenMP)

	GFLOPS	***n* = 1024**	***n* = 2048**	***n* = 4096**
AMD E-450	0.041	0:00:51.984	0:09:09.193	1:16:42.243
	0.034	0:01:03.269	0:09:56.439	1:16:23.849
AMD Athlon 64 X2 4000+	0.091	0:00:25.322	0:03:09.150	0:31:59.924
	0.090	0:00:23.797	0:04:00.578	0:58:26.110
AMD Phenom II N830	0.281	0:00:07.632	0:01:16.637	0:16:12.685
	0.225	0:00:09.526	0:02:01.892	0:29:42.571
AMD Phenom II X4 925	0.646	0:00:03.323	0:01:02.180	0:12:23.589
	0.486	0:00:04.415	0:01:25.240	0:21:31.178
Intel Core 2 Duo T9300	0.903	0:00:02.379	0:01:55.635	0:17:43.703
	0.619	0:00:03.471	0:02:01.489	0:18:38.319
Intel Core 2 Quad Q9300	0.060	0:00:36.038	0:05:06.918	0:41:56.788
	0.082	0:00:26.045	0:05:13.983	0:41:50.549
Intel Core i3 2120	0.403	0:00:05.332	0:00:56.302	0:07:52.335
	0.429	0:00:05.004	0:00:58.587	0:09:19.800
Intel Core i5 3470	1.071	0:00:02.005	0:00:21.218	0:06:15.535
	1.047	0:00:02.052	0:00:23.866	0:06:09.308
Intel Core i7 2600	1.186	0:00:01.811	0:00:17.019	0:04:32.796
	1.152	0:00:01.864	0:00:18.833	0:04:55.684
Intel Core i7 4930K	1.884	0:00:01.140	0:00:09.787	0:02:59.023
	1.949	0:00:01.102	0:00:09.971	0:03:00.188

Как видно из приведенных в табл. 3 результатов, по эффективности библиотека матричной алгебры на языке C/C++ с применением технологии OpenMP аналогична библиотеке на Object Pascal / Delphi с применением OmniThreadLibrary. Такой вывод остается справедливым только при использовании стандартных средств программирования, не применяющих оптимизацию, например, компилятора C/C++ из пакета Microsoft Visual Studio Express 2012.

Использование оптимизирующего компилятора из пакета Intel Parallel Studio 2013 дает существенный прирост производительности без каких-либо изменений в программе. Рост производительности может зависеть от уровня оптимизации, выбор которого осуществляется указанием ключей компилятора. В табл. 4 приведены соответствующие результаты тестов, уровень оптимизации: O3. Время вычислений сокращается в десятки раз. Например, максимальная производительность возросла в 72.3 раза (данные для процессора Intel Core 2 Quad Q9300, одинарная точность).

Таблица 4. Результаты тестирования библиотеки на языке C/C++ (OpenMP, компилятор из пакета Intel Parallel Studio XE 2013)

	GFLOPS	$n = 1024$	$n = 2048$	$n = 4096$
AMD E-450	1.911	0:00:01.124	0:00:10.241	0:01:22.001
	0.923	0:00:02.326	0:00:20.352	0:02:48.200
AMD Athlon 64 X2 4000+	1.705	0:00:01.304	0:00:10.173	0:01:20.629
	0.906	0:00:02.419	0:00:19.086	0:02:31.641
AMD Phenom II N830	4.472	0:00:00.561	0:00:03.881	0:00:30.734
	2.546	0:00:00.893	0:00:06.748	0:00:54.328
AMD Phenom II X4 925	12.783	0:00:00.168	0:00:03.382	0:00:26.996
	2.540	0:00:00.864	0:00:06.793	0:00:54.101
Intel Core 2 Duo T9300	5.607	0:00:00.383	0:00:03.849	0:00:37.776
	2.180	0:00:00.985	0:00:08.808	0:01:35.361
Intel Core 2 Quad Q9300	4.338	0:00:00.495	0:00:04.114	0:00:46.789
	2.073	0:00:01.036	0:00:09.493	0:01:34.714
Intel Core i3 2120	8.324	0:00:00.270	0:00:02.083	0:00:16.512
	3.768	0:00:00.570	0:00:04.811	0:00:40.290
Intel Core i5 3470	26.189	0:00:00.082	0:00:00.824	0:00:06.956
	11.546	0:00:00.186	0:00:01.791	0:00:16.618
Intel Core i7 2600	32.770	0:00:00.070	0:00:00.612	0:00:04.194
	16.519	0:00:00.130	0:00:01.154	0:00:08.788
Intel Core i7 4930K	60.069	0:00:00.046	0:00:00.286	0:00:02.571
	31.172	0:00:00.081	0:00:00.659	0:00:04.409

Отличительная особенность библиотеки на языке C/C++ с директивами OpenMP при использовании оптимизирующего компилятора из пакета Intel Parallel Studio XE 2013 состоит в том, что матрицы, элементы которых – числа с плавающей запятой одинарной точности, умножаются примерно в два раза быстрее, чем матрицы, элементы которых – числа с плавающей запятой двойной точности. То же самое справедливо и для других операций с матрицами. Это связано с эффективным использованием кэша центрального процессора. Для других библиотек такой разницы между типами чисел с плавающей запятой не наблюдается.

Следует отметить, что данные, приведенные в табл. 1–4, могут меняться от теста к тесту, поскольку невозможно совсем исключить влияние фоновых процессов в операционной системе. Например, чтобы иметь представление о влиянии фоновых процессов на время выполнения, тест умножения матриц размеров 2048x2048 был запущен 1000 раз. В результате минимальное время составило 0.633, среднее время: 0.695, среднеквадратическое отклонение составило 0.037 (данные для процессора Intel Core i7 4930K, двойная точность, компилятор C/C++ из пакета Intel Parallel Studio 2013).

4. Применение технологий Intel Threading Building Blocks и Intel Cilk Plus

Intel Threading Building Blocks – библиотека шаблонов C++ для написания многопоточных программ, созданная в 2006 г. Она обладает большим количеством готовых алгоритмов, конструкций и структур данных, ориентированных на использование в параллельных вычислениях. К сожалению, при необходимости распараллеливания уже существующей программы с использованием этой библиотеки потребуется достаточно большое количество изменений, что хорошо видно на примере листинга 5. В связи с этим, отсутствует какая-либо совместимость с компиляторами, которые не поддерживают данную библиотеку.

Листинг 5. Умножение матриц на языке C++ с применением Intel Threading Building Blocks

```
1: #include <tbb/parallel_for.h>
2:
3: class CMultiply
4: {
5:   double* A;
6:   double* B;
7:   double* C;
8:   int W, Q, H;
9: public:
10:   void operator()(tbb::blocked_range<int> r) const {
11:     for (int i = r.begin(); i < r.end(); ++i)
12:     {
13:       int iQ = i * Q;
14:       int iH = i * H;
15:       for (int j = 0; j < H; ++j)
16:       {
17:         double x = 0;
18:         for (int k = 0; k < Q; ++k)
19:           x += A[iQ + k] * B[k * H + j];
20:         C[iH + j] = x;
21:       }
22:     }
23:   }
24:   CMultiply(double* a, double* b, double* c, int w, int q, int h): A(a), B(b), C(c), W(w), Q(q), H(h) {};
25: };
26:
27: void Multiply(double* a, double* b, double* c, int w, int q, int h)
28: {
29:   tbb::parallel_for(tbb::blocked_range<int>(0, w), CMultiply(a, b, c, w, q, h));
30: }
```

Таблица 5. Результаты тестирования библиотеки на языке C++ (Intel Threading Building Blocks, компилятор из пакета Intel Parallel Studio XE 2013)

	GFLOPS	$n = 1024$	$n = 2048$	$n = 4096$
AMD E-450	0.042	0:00:50.792	0:08:22.410	1:09:35.351
	0.035	0:01:01.117	0:08:50.580	1:15:10.039
AMD Athlon 64 X2 4000+	0.094	0:00:25.565	0:03:03.390	0:31:25.688
	0.092	0:00:23.422	0:04:03.968	0:59:44.333
AMD Phenom II N830	0.279	0:00:07.700	0:01:21.652	0:16:17.811
	0.226	0:00:09.500	0:02:02.139	0:29:40.276
AMD Phenom II X4 925	0.636	0:00:03.379	0:01:01.299	0:12:18.308
	0.469	0:00:04.576	0:01:26.326	0:21:30.110
Intel Core 2 Duo T9300	0.929	0:00:02.312	0:01:55.499	0:17:48.179
	0.547	0:00:03.925	0:02:01.375	0:18:24.189
Intel Core 2 Quad Q9300	0.065	0:00:34.743	0:04:47.018	0:35:19.491
	0.089	0:00:24.190	0:04:52.955	0:41:18.190
Intel Core i3 2120	0.370	0:00:05.805	0:00:55.955	0:07:56.210
	0.418	0:00:05.140	0:00:58.529	0:09:24.127
Intel Core i5 3470	1.059	0:00:02.028	0:00:25.624	0:06:16.714
	0.987	0:00:02.175	0:00:32.415	0:06:42.693
Intel Core i7 2600	1.186	0:00:01.810	0:00:15.897	0:04:36.477
	1.172	0:00:01.832	0:00:23.010	0:04:32.734
Intel Core i7 4930K	1.811	0:00:01.186	0:00:09.892	0:02:47.437
	1.800	0:00:01.193	0:00:10.744	0:03:15.391

Для рассматриваемой библиотеки матричной алгебры намного удобнее использовать технологию Intel Cilk Plus вместо Intel Threading Building Blocks. Intel Cilk Plus – расширение языка C++, выпущенное в 2010 г. и позволяющее упростить написание многопоточных программ (изначально язык Cilk был разработан в 1994 г. как расширение C). Intel Cilk Plus обеспечивает возможность организации параллелизма по данным (SIMD).

При применении Intel Cilk Plus достаточно заменить оператор цикла **for** на cilk_for (см. листинг 6). При этом внутренняя реализация распараллеливания идентична варианту с Intel Threading Building Blocks, поэтому данные тестирования библиотеки, использующей технологию Intel Cilk Plus, здесь не приводятся: они аналогичны приведенным в табл. 5.

Листинг 6. Умножение матриц на языке C++ с применением Intel Cilk Plus

```
#include <cilk/cilk.h>

void Multiply(double* a, double* b, double* c, int w, int q, int h)
{
  cilk_for (int i = 0; i < w; i++)
  {
    int iq = i * q;
    int ih = i * h;
    for (int j = 0; j < h; j++)
    {
      double x = 0;
      for (int k = 0; k < q; k++)
        x += a[iq + k] * b[k * h + j];
```

```
14:         c[ih + j] = x;
15:       }
16:     }
17: }
```

Сравнивая результаты тестирования из табл. 3 и 5, можно сделать вывод, что библиотеки, использующие технологии Intel Threading Building Blocks и Intel Cilk Plus, не имеют преимуществ по сравнению с библиотекой на основе OpenMP. Однако в ходе тестирования были получены заслуживающие внимания результаты для умножения матриц с использованием формулы

$$c_{ij} = \sum_{k=1}^{q} a_{ik} b_{jk}, \quad i = 1,...,w, \quad j = 1,...,h,$$

при предварительном транспонировании матрицы $B = [b_{ij}]$. Эти результаты (см. табл. 6) оказываются лучше, чем для библиотеки с директивами OpenMP с применением оптимизирующего компилятора из пакета Intel Parallel Studio XE 2013 даже с учетом времени, которое затрачивается на траспонирование. Результаты тестирования для библиотек на основе Intel Threading Building Blocks и Intel Cilk Plus одинаковы. Ни для какой другой библиотеки из рассмотренных в данной работе такого увеличения производительности при предварительном транспонировании матрицы $B = [b_{ij}]$ не происходит.

Таблица 6. Результаты тестирования библиотеки на языке C/C++ (предварительное транспонирование матрицы $B = [b_{ij}]$, Intel Threading Building Blocks, компилятор из пакета Intel Parallel Studio XE 2013)

	GFLOPS	***n* = 1024**	***n* = 2048**	***n* = 4096**
AMD E-450 APU	1.904 0.951	0:00:01.147 0:00:02.258	0:00:09.021 0:00:18.395	0:01:13.521 0:02:27.313
AMD Athlon 64 X2 4000+	1.794 0.866	0:00:01.255 0:00:02.526	0:00:09.574 0:00:19.846	0:01:21.224 0:02:40.316
AMD Phenom II N830	5.507 2.808	0:00:00.439 0:00:00.848	0:00:03.519 0:00:06.118	0:00:24.956 0:00:50.294
AMD Phenom II X4 925	17.896 2.555	0:00:00.120 0:00:00.846	0:00:03.318 0:00:06.723	0:00:26.537 0:00:54.155
Intel Core 2 Duo T9300	8.590 2.309	0:00:00.250 0:00:00.930	0:00:03.988 0:00:08.351	0:00:36.025 0:01:23.288
Intel Core 2 Quad Q9300	4.540 2.077	0:00:00.473 0:00:01.034	0:00:04.789 0:00:08.770	0:00:50.404 0:01:55.649
Intel Core i3 2120	9.896 4.090	0:00:00.217 0:00:00.525	0:00:01.861 0:00:04.464	0:00:14.951 0:00:44.292
Intel Core i5 3470	40.519 9.539	0:00:00.053 0:00:00.236	0:00:00.821 0:00:01.801	0:00:06.925 0:00:17.003
Intel Core i7 2600	56.513 19.884	0:00:00.038 0:00:00.108	0:00:00.414 0:00:00.955	0:00:03.724 0:00:08.092
Intel Core i7 4930K	71.583 35.205	0:00:00.030 0:00:00.061	0:00:00.261 0:00:00.595	0:00:02.301 0:00:04.737

Отметим, что здесь, как и для библиотеки на языке C/C++ с директивами OpenMP при использовании оптимизирующего компилятора из пакета Intel Parallel Studio XE 2013, матрицы, элементы которых – числа с плавающей запятой одинарной точности, умножаются в два раза быстрее, чем матрицы, элементы которых – числа с плавающей запятой двойной точности. Таким образом, использование чисел с плавающей запятой одинарной точности не только в два раза экономит память, необходимую для хранения матриц, но и позволяет в два раза увеличить скорость вычислений.

5. Применение технологии CUDA

CUDA – платформа для написания многопоточных программ, выпущенная в 2007 г. Она ориентирована на использование графических процессоров nVidia и включает в себя специализированный пакет, позволяющий разрабатывать параллельные программы на специальном расширении языка C и включать в текст программы специальные функции, использующие мощности графического процессора видеоадаптера. Существенным требованием к оборудованию является наличие современного видеоадаптера nVidia.

Функция умножения матриц состоит из двух частей. Первая часть обеспечивает выделение и освобождение памяти видеоадаптера, копирование исходных матриц в эту память и копирование результата из памяти видеоадаптера, передачу настроек многопоточности для графического процессора, команды синхронизации (строки 17–35 в листинге 7). Вторая часть представляет собой программу, выполняемую непосредственно на видеоадаптере (строки 4–15 в листинге 7). В листинге 7 помимо функций, определенных в библиотеке CUDA, используется функция int iceil(int a, int b), которая возвращает целочисленный результат деления a на b с округлением в большую сторону.

Листинг 7. Умножение матриц на языке C/C++ с применением CUDA

```
1: #include <cuda.h>
2: #define BlockSize 8
3:
4: __global__ void kerMultiply(double* a, double* b, double* c, int w, int q, int h)
5: {
6:   int i = BlockSize * blockIdx.x + threadIdx.x;
7:   int j = BlockSize * blockIdx.y + threadIdx.y;
8:   if ((i < w) && (j < h))
9:   {
10:     double sum = 0;
11:     for ( int k = 0; k < q; k++ )
12:       sum += a[i * q + k] * b[k * h + j];
13:     c[i * h + j] = sum;
14:   }
15: }
16:
17: void Multiply(double* A, double* B, double* C, int w, int q, int h)
18: {
19:   double* d_A;
20:   double* d_B;
21:   double* d_C;
22:   cudaMalloc((void**) &d_A, w * q * sizeof(double));
23:   cudaMalloc((void**) &d_B, q * h * sizeof(double));
```

```
24:   cudaMalloc((void**) &d_C, w * h * sizeof(double));
25:   cudaMemcpy(d_A, A, w * q * sizeof(double), cudaMemcpyHostToDevice);
26:   cudaMemcpy(d_B, B, q * h * sizeof(double), cudaMemcpyHostToDevice);
27:   dim3 threads(BlockSize, BlockSize);
28:   dim3 grid(iceil(w, BlockSize), iceil(h, BlockSize));
29:   kerMultiply<<<grid, threads>>>(d_A, d_B, d_C, w, q, h);
30:   cudaThreadSynchronize();
31:   cudaMemcpy(C, d_C, w * h * sizeof(double), cudaMemcpyDeviceToHost);
32:   cudaFree(d_A);
33:   cudaFree(d_B);
34:   cudaFree(d_C);
35: }
```

Результаты, приведенные в табл. 7, а также в последующих таблицах учитывают не только время вычислений на графическом процессоре, но и время, затрачиваемое на копирование матриц-множителей из основной памяти в память видеоадаптера и копирование результирующей матрицы из памяти видеоадаптера в основную память.

При длительных по времени вычислениях на графических процессорах видеоадаптеров возможно досрочное прекращение работы программы из-за особенностей реализации взаимодействия операционной системы и драйвера видеоадаптера. В таком случае либо требуется настроить параметры операционной системы нужным образом (увеличить пороговые значения времени отклика драйвера), либо модифицировать программу расчета, в данном случае программу умножения матриц.

Таблица 7. Результаты тестирования библиотеки на языке C/C++ (CUDA)

	GFLOPS	$n = 1024$	$n = 2048$	$n = 4096$
nVidia Quadro 2000	12.811	0:00:00.221	0:00:01.341	0:00:20.698
	6.298	0:00:00.341	0:00:02.843	0:00:35.745
nVidia GeForce GTX 560 Ti	29.826	0:00:00.091	0:00:00.576	0:00:10.353
	15.675	0:00:00.137	0:00:01.190	0:00:13.951
nVidia GeForce GTX 650M	6.068	0:00:00.354	0:00:02.831	0:00:33.525
	5.522	0:00:00.392	0:00:03.111	0:00:34.765
nVidia GeForce GTX 650 Ti	12.056	0:00:00.182	0:00:01.425	0:00:18.623
	10.964	0:00:00.200	0:00:01.567	0:00:26.763
nVidia GeForce GTX 720M	1.766	0:00:01.531	0:00:09.729	0:02:52.398
	0.898	0:00:02.391	0:00:20.146	–
nVidia GeForce GTX 740M	6.527	0:00:00.329	0:00:02.665	0:00:30.266
	5.753	0:00:00.379	0:00:02.986	0:00:31.492
nVidia GeForce GTX 870M	21.475	0:00:00.100	0:00:00.804	0:00:09.481
	19.545	0:00:00.114	0:00:00.879	0:00:30.093
nVidia GeForce GTX 970	92.864	0:00:00.026	0:00:00.185	0:00:05.773
	72.796	0:00:00.032	0:00:00.236	0:00:11.493
nVidia GeForce GTX 980	107.374	0:00:00.023	0:00:00.160	0:00:05.171
	74.051	0:00:00.029	0:00:00.241	0:00:10.889

При возникновении подобной ситуации, а она более вероятна при использовании бюджетных видеоадаптеров, предлагается использовать следующую формулу для элементов результирующей матрицы:

$$c_{ij} = \sum_{l=0}^{L_{q,m}} \sum_{k=lm+1}^{\min\{(l+1)m,q\}} a_{ik} b_{kj}, \quad i = 1,\ldots,w, \quad j = 1,\ldots,h, \quad L_{q,m} = \begin{cases} [\frac{q}{m}], & \{\frac{q}{m}\} \neq 0, \\ [\frac{q}{m}] - 1, & \{\frac{q}{m}\} = 0. \end{cases}$$

Тогда на графическом процессоре видеоадаптера вычисляются произведения блоков исходных матриц (горизонтальных блоков для левого множителя $A = [a_{ij}]$ и вертикальных – для правого множителя $B = [b_{ij}]$), далее результаты суммируются (между умножениями блоков управление передается основной программе). Натуральный параметр m подбирается экспериментально таким образом, чтобы обеспечить стабильность расчетов. Для разных видеоадаптеров пороговые значения m, вообще говоря, различные. Однако чтобы результаты тестирования были сравнимы, использовалось одно и то же значение, а именно $m = 256$ для матриц размеров 4096x4096 (данное значение подбиралось для nVidia GeForce GTX 650 Ti с небольшим запасом надежности). Для nVidia GeForce GTX 720M при умножении матриц, элементы которых – числа с плавающей запятой двойной точности, такое значение m оказалось слишком большим, поэтому в соответствующей графе табл. 7 стоит прочерк (далее прочерки в таблицах результатов означают невозможность проведения расчетов при указанных параметрах). Для мощных видеоадаптеров, таких как nVidia GeForce GTX 970 и nVidia GeForce GTX 980, это значение наоборот слишком мало, более того, для них можно использовать простой вариант функции умножения из листинга 7, получая меньшее время счета. Для матриц размеров 1024x1024 и 2048x2048 такая методика умножения нецелесообразна (достаточно функции из листинга 7), но с ростом размеров матриц, например, до 8192x8192 или выше параметр m нужно уменьшать и использовать усложненный вариант умножения даже для nVidia GeForce GTX 980.

Вычисления могут быть гораздо эффективнее по затратам времени, если использовать разделяемую (быструю) память. Это замечание относится только к тем операциям с матрицами, при которых происходит многократное обращение к одним и тем же элементам, например, при умножении матриц.

Далее, в листинге 8 содержится только часть, которая выполняется непосредственно на видеоадаптере и использует разделяемую память. К этой части необходимо добавить преамбулу и вспомогательную часть для управления памятью и запуска графического процессора видеоадаптера (строки 1, 2, 17–35 в листинге 7).

Листинг 8. Умножение матриц на языке C/C++ с применением CUDA (использование разделяемой памяти)

```
1: __global__ void kerMultiply (double* a, double* b, double* c, int w, int q, int h)
2: {
3:   int bx = blockIdx.x;
4:   int by = blockIdx.y;
5:   int tx = threadIdx.x;
6:   int ty = threadIdx.y;
7:   int aBegin = q * BlockSize * bx;
8:   int aEnd = aBegin + q - 1;
9:   int aStep = BlockSize;
10:   int bBegin = BlockSize * by;
11:   int bStep = BlockSize * h;
12:   __shared__ double a_cache[BlockSize][BlockSize];
13:   __shared__ double b_cache[BlockSize][BlockSize];
```

```
14:   double sum = 0;
15:   for (int i = aBegin, j = bBegin; i <= aEnd; i += aStep, j += bStep)
16:   {
17:     a_cache[tx][ty] = a[i + q * tx + ty];
18:     b_cache[tx][ty] = b[j + h * tx + ty];
19:     __syncthreads();
20:     for (int k = 0; k < BlockSize; ++k)
21:       sum += a_cache[tx][k] * b_cache[k][ty];
22:     __syncthreads();
23:   }
24:   c[h * BlockSize * bx + BlockSize * by + h * tx + ty] = sum;
25: }
```

Ключевым отличием приведенного в листинге 8 фрагмента программы является то, что размеры матриц должны быть кратны размеру блока BlockSize. Разделяемая память используется для кэширования блоков матриц-множителей размеров BlockSize x BlockSize. Для предыдущего листинга размеры матриц значения не имеют.

Время счета, приведенное в табл. 7 и 8, также зависит от размера блока BlockSize. Для ряда видеоадаптеров размер, равный 8, является оптимальным, например, для nVidia GeForce GTX 650 Ti и nVidia GeForce GTX 980. Для других, возможно, потребуется переопределить величину BlockSize в соответствии с имеющимися рекомендациями или подобрать оптимальную величину экспериментально. Различия в производительности библиотек может быть существенным. Так, установка размера блока BlockSize, равного 16, увеличивает время счета примерно на 50% без использования разделяемой памяти и на 35% с ее использованием, а переход к значению 32 увеличивает время счета почти на 100% и 30% соответственно. Последние данные приведены для nVidia GeForce GTX 980. Для nVidia GeForce GTX 650 Ti различия еще больше.

Таблица 8. Результаты тестирования библиотеки на языке C/C++ (CUDA, разделяемая память)

	GFLOPS	***n* = 1024**	***n* = 2048**	***n* = 4096**
nVidia Quadro 2000	21.913 7.781	0:00:00.100 0:00:00.276	0:00:00.784 0:00:02.261	0:00:06.335 0:00:20.291
nVidia GeForce GTX 560 Ti	57.147 17.766	0:00:00.042 0:00:00.124	0:00:00.313 0:00:00.967	0:00:02.405 0:00:07.819
nVidia GeForce GTX 650M	16.893 12.939	0:00:00.143 0:00:00.198	0:00:01.017 0:00:01.342	0:00:08.369 0:00:10.622
nVidia GeForce GTX 650 Ti	33.054 25.288	0:00:00.066 0:00:00.097	0:00:00.537 0:00:00.729	0:00:04.158 0:00:05.435
nVidia GeForce GTX 720M	3.030 0.945	0:00:00.723 0:00:02.272	0:00:05.669 –	– –
nVidia GeForce GTX 740M	17.495 13.190	0:00:00.128 0:00:00.172	0:00:00.982 0:00:01.313	0:00:08.077 0:00:10.420
nVidia GeForce GTX 870M	57.651 44.278	0:00:00.038 0:00:00.053	0:00:00.298 0:00:00.388	0:00:02.396 0:00:03.109
nVidia GeForce GTX 970	251.260 51.883	0:00:00.013 0:00:00.049	0:00:00.077 0:00:00.347	0:00:00.547 0:00:02.649
nVidia GeForce GTX 980	278.217 59.523	0:00:00.011 0:00:00.043	0:00:00.069 0:00:00.317	0:00:00.494 0:00:02.309

Производительность при использовании разделяемой памяти, как правило, возрастает. Например, лучший результат – увеличение производительности в 2.78 раза (данные для nVidia GeForce GTX 650M, одинарная точность). В то же время на некоторых графических процессорах использование разделяемой памяти привело к сокращению производительности (данные для nVidia GeForce GTX 970 и nVidia GeForce GTX 980, двойная точность). На конечный результат оказывает влияние много факторов (не только размер блока BlockSize), включая архитектуру графического процессора и тип памяти.

При вычислениях на графических процессорах видеоадаптеров приведенные в табл. 7 и 8 (а также в последующих таблицах) данные могут меняться, как и при вычислениях на центральном процессоре, но при этом разброс времени счета незначительный. Например, тест умножения матриц размеров 2048x2048 был запущен 1000 раз. В результате минимальное время составило 1.557, среднее время: 1.563, среднеквадратическое отклонение составило 0.005 (данные для nVidia GeForce GTX 650 Ti, двойная точность, разделяемая память не используется).

6. Применение технологии OpenCL

OpenCL (Open Computing Language) – платформа для написания параллельных программ, исполняемых на различных процессорах (не обязательно графических). В нее входят язык программирования, основанный на языке C, и специализированный интерфейс программирования приложений, т.е. набор констант, структур, функций, классов, предназначенный для более простой разработки программ, использующих возможности современных видеоадаптеров различных производителей. Это полностью открытое, кроссплатформенное программное обеспечение.

Листинг 9 содержит фрагмент программы умножения матриц с применением OpenCL. Приведена часть, которая выполняется непосредственно на видеоадаптере, и процедура, обеспечивающая выделение и освобождение памяти видеоадаптера, копирование исходных матриц в эту память, копирование результата из памяти видеоадаптера и т.п. Дополнительно требуется инициализация OpenCL, выбор платформы и устройства, передача текста программы, исполняемой графическим процессором видеадаптера, и ее компиляция. Эта часть программы не приводится из-за ее громоздкости, с ее аналогом можно ознакомиться в [15].

Как и при использовании технологии CUDA, если вычисления на графических процессорах видеоадаптеров занимают продолжительное время без связи с основной программой, выполняемой на центральном процессоре, возможно досрочное принудительное прекращение вычислений со стороны операционной системы. Поэтому рекомендуется использовать подход к организации вычислений, описанный в предыдущем разделе.

Листинг 9. Умножение матриц на языке Object Pascal / Delphi с применением OpenCL

```
uses
  CTypes, dglOpenGL, CL, CL_GL;

const
  codeOpenCL: PChar =
    '#pragma OPENCL EXTENSION cl_khr_fp64: enable'#10 +
    '__kernel void Multiply(__global double* a, __global double* b, __global double* c, int q, int h) {'#10 +
      'int i = get_global_id(0);'#10 +
      'int j = get_global_id(1);'#10 +
       'int iq = i * q;#10 +
       'double x = 0;'#10 +
       'for (int k = 0; k < q; k++) x += a[iq + k] * b[k * h + j];'#10 +
       'c[i * h + j] = x;'#10 +
     '}';

procedure Multiply(A, B, R: Pointer; W, Q, H: Integer);
var
  MemSizeA, MemSizeB, MemSizeR: Cardinal;
  GPUData1, GPUData2, GPUOutput: cl_mem;
  globalThreads: array[0..1] of size_t;
begin
  MemSizeA:=W * Q * SizeOf(Double);
  MemSizeB:=Q * H * SizeOf(Double);
  MemSizeR:=W * H * SizeOf(Double);
  GPUData1:=clCreateBuffer(context, CL_MEM_READ_ONLY or CL_MEM_COPY_HOST_PTR,
    MemSizeA, A, nil);
  GPUData2:=clCreateBuffer(context, CL_MEM_READ_ONLY or CL_MEM_COPY_HOST_PTR,
    MemSizeB, B, nil);
  GPUOutput:=clCreateBuffer(context, CL_MEM_WRITE_ONLY, MemSizeR, nil, nil);
  clSetKernelArg(OpenCLKernelMMMultiply, 0, SizeOf(cl_mem), @GPUData1);
  clSetKernelArg(OpenCLKernelMMMultiply, 1, SizeOf(cl_mem), @GPUData2);
  clSetKernelArg(OpenCLKernelMMMultiply, 2, SizeOf(cl_mem), @GPUOutput);
  clSetKernelArg(OpenCLKernelMMMultiply, 3, SizeOf(Integer), @Q);
  clSetKernelArg(OpenCLKernelMMMultiply, 4, SizeOf(Integer), @H);
  globalThreads[0]:=W;
  globalThreads[1]:=H;
  clEnqueueNDRangeKernel(CommandQueue, OpenCLKernelMMMultiply, 2, nil,
    @globalThreads, nil, 0, nil, nil);
  clEnqueueReadBuffer(CommandQueue, GPUOutput, CL_TRUE, 0, MemSizeR, R, 0, nil, nil);
  clReleaseMemObject(GPUData1);
  clReleaseMemObject(GPUData2);
  clReleaseMemObject(GPUOutput);
end;
```

Таблица 9. Результаты тестирования библиотеки на языке Object Pascal / Delphi (среда программирования Borland Delphi 7 с библиотекой OpenCL for Delphi)

	GFLOPS	$n = 1024$	$n = 2048$	$n = 4096$
Intel HD Graphics 5100	1.733	0:00:01.239	0:00:35.493	0:16:23.043
	1.467	0:00:01.464	0:00:36.842	0:20:15.536
Intel HD Graphics 5200	4.699	0:00:00.457	0:00:11.458	0:08:24.696
	4.203	0:00:00.511	0:00:12.689	0:10:33.188
AMD Radeon HD 5470	0.442	0:00:04.864	0:00:41.536	0:06:01.760
	0.414	0:00:05.452	0:00:41.479	0:06:33.802
AMD Radeon HD 5670	0.896	0:00:02.397	0:00:19.527	0:02:47.705
	0.843	0:00:02.547	0:00:21.532	0:03:15.479
AMD Radeon HD 5700	2.418	0:00:00.888	0:00:08.807	0:01:55.816
	2.028	0:00:01.059	0:00:10.431	0:02:08.309
AMD Radeon HD 6320	0.161	0:00:13.301	0:01:55.886	0:14:55.680
	0.135	0:00:15.861	0:02:08.418	0:17:46.526
AMD Radeon HD 6370	0.089	0:00:26.707	0:03:27.839	0:25:38.740
	0.098	0:00:22.012	0:03:09.057	0:27:45.475
AMD Radeon HD 7450	2.221	0:00:00.967	0:00:12.309	0:01:53.291
	2.093	0:00:01.026	0:00:12.224	0:02:06.914
nVidia Quadro 2000	0.921	0:00:02.332	0:00:27.404	0:04:14.523
	0.889	0:00:02.415	0:00:24.157	0:05:34.828
nVidia GeForce GTX 560 Ti	1.885	0:00:01.139	0:00:13.779	0:03:03.452
	1.981	0:00:01.084	0:00:14.996	0:03:06.097
nVidia GeForce GTX 650M	1.600	0:00:01.342	0:00:37.797	0:08:46.337
	1.476	0:00:01.455	0:00:37.997	0:09:56.191
nVidia GeForce GTX 650 Ti	2.837	0:00:00.757	0:00:16.873	0:04:28.454
	2.471	0:00:00.869	0:00:17.474	0:04:37.761
nVidia GeForce GTX 720M	0.776	0:00:02.768	0:01:06.299	0:17:05.524
	0.812	0:00:02.646	0:01:09.791	0:20:05.117
nVidia GeForce GTX 740M	4.090	0:00:00.525	0:00:20.266	0:06:33.525
	3.244	0:00:00.662	0:00:22.905	0:07:26.874
nVidia GeForce GTX 870M	5.316	0:00:00.404	0:00:15.550	0:04:16.966
	5.017	0:00:00.428	0:00:16.366	0:05:11.623
nVidia GeForce GTX 970	4.648	0:00:00.462	0:00:16.158	0:04:57.286
	4.075	0:00:00.527	0:00:18.263	0:05:17.345
nVidia GeForce GTX 980	24.129	0:00:00.089	0:00:01.015	0:00:11.428
	21.262	0:00:00.101	0:00:01.205	0:00:24.906

Функции работы с матрицами, реализованные с помощью технологии OpenCL, работают значительно медленнее по сравнению с теми же функциями, реализованными с помощью технологии CUDA (речь, конечно, идет об видеоадаптерах с графическими процессорами nVidia), поэтому пороговые значения m нужно выбирать меньше. В частности

приведенным в табл. 9 результатам соответствует значение $m=16$ для матриц размеров 1024x1024, $m=4$ для матриц размеров 2048x2048 и $m=2$ для матриц размеров 4096x4096. Такой выбор соответствует единому значению, которое обеспечивает стабильность вычислений для всех видеоадаптеров, используемых при тестировании разработанной библиотеки. Безусловно, это не является оптимальным с точки зрения достижения максимальной производительности. С ростом класса видеоадаптера значение m можно увеличивать, это значение можно брать больше при использовании графических процессоров AMD.

Технология OpenCL также позволяет задействовать разделяемую (быструю) память для ускорения вычислений.

7. Применение технологии Microsoft Accelerated Massive Parallelism

C++ Accelerated Massive Parallelism – библиотека и спецификация фирмы Microsoft, использующая технологию DirectX 11, созданные в 2010 г. для написания параллельных программ на языке C++ для гибридных систем. Позволяет выполнять параллельные вычисления как на центральных процессорах, так и на графических процессорах видеоадаптеров без внесения большого количества изменений в исходную программу. Программы, использующие Microsoft Accelerated Massive Parallelism, для корректного исполнения требуют наличия на компьютере операционной системы Microsoft Windows 7, в связи с этим поддерживаются только современными компиляторами и средами, например, Microsoft Visual Studio 2012 и выше.

Листинг 10. Умножение матриц на языке C++ с применением Microsoft Accelerated Massive Parallelism

```
1: #include <amp.h>
2:
3: void Multiply(double* pA, double* pB, double* pC, int w, int q, int h)
4: {
5:   concurrency::array_view<double,2> a(w,q,pA), b(q,h,pB), c(w,h,pC);
6:   concurrency::parallel_for_each(c.extent,[=](concurrency::index<2> i)
7:   restrict(amp)
8:   {
9:     double x = 0;
10:     for (int k = 0; k < q; k++)
11:       x += a(i[0], k) * b(k, i[1]);
12:     c[i] = x;
13:   });
14: }
```

Таблица 10. Результаты тестирования библиотеки на языке C++ (Microsoft Accelerated Massive Parallelism)

	GFLOPS	$n = 1024$	$n = 2048$	$n = 4096$
Intel HD Graphics 2500	3.722	0:00:00.577	0:00:05.010	0:00:41.029
	1.445	0:00:01.501	0:00:11.888	0:01:50.065
Intel HD Graphics 5100	18.879	0:00:00.143	0:00:00.910	0:00:18.085
	5.459	0:00:00.452	0:00:03.147	–
Intel HD Graphics 5200	21.448	0:00:00.141	0:00:00.801	–
	5.784	0:00:00.474	0:00:02.970	0:00:34.400
AMD Radeon HD 6320	2.619	0:00:00.853	0:00:08.289	0:00:52.468
	–	–	–	–
nVidia Quadro 2000	24.684	0:00:00.087	0:00:00.733	0:00:11.420
	12.073	0:00:00.200	0:00:01.423	0:00:12.948
nVidia GeForce GTX 560 Ti	42.108	0:00:00.051	0:00:00.432	0:00:04.508
	34.429	0:00:00.144	0:00:00.499	0:00:06.724
nVidia GeForce GTX 580	89.015	0:00:00.031	0:00:00.193	0:00:04.962
	50.529	0:00:00.054	0:00:00.340	0:00:07.733
nVidia GeForce GTX 650M	13.744	0:00:00.161	0:00:01.250	0:00:12.041
	16.086	0:00:00.145	0:00:01.068	0:00:23.915
nVidia GeForce GTX 650 Ti	34.223	0:00:00.066	0:00:00.502	0:00:06.027
	26.030	0:00:00.088	0:00:00.660	0:00:10.469
nVidia GeForce GTX 720M	3.302	0:00:00.820	0:00:07.534	0:00:41.620
	1.943	0:00:01.105	0:00:09.014	0:01:21.543
nVidia GeForce GTX 740M	15.718	0:00:00.142	0:00:01.093	0:00:22.919
	9.276	0:00:00.237	0:00:01.852	0:00:45.996
nVidia GeForce GTX 870M	59.241	0:00:00.038	0:00:00.290	0:00:03.975
	30.569	0:00:00.074	0:00:00.562	0:00:08.774
nVidia GeForce GTX 970	127.258	0:00:00.019	0:00:00.135	0:00:03.228
	56.327	0:00:00.043	0:00:00.305	0:00:05.272
nVidia GeForce GTX 980	148.102	0:00:00.016	0:00:00.116	0:00:02.753
	58.040	0:00:00.041	0:00:00.296	0:00:04.873

Приведенные в табл. 10 данные соответствуют умножению блоками (горизонтальными для левого множителя $A = [a_{ij}]$ и вертикальными – для правого множителя $B = [b_{ij}]$, $m = 256$) при размерах матриц 4096x4096. Если умножение блоками не применять, то, как и при использовании технологий CUDA и OpenCL, велика вероятность досрочного прекращения работы программы.

Листинг 11. Умножение матриц на языке C++ с применением Microsoft Accelerated Massive Parallelism (использование разделяемой памяти)

```
1: #include <amp.h>
2: #define TileSize 16
3:
4: void Multiply(double* pA, double* pB, double* pC, int w, int q, int h)
5: {
6:   concurrency::array_view<double,2> a(w, q, pA), b(q, h, pB), c(w, h, pC);
7:   concurrency::parallel_for_each(c.extent.tile<TileSize,TileSize>(),
8:   [=](concurrency::tiled_index<TileSize,TileSize> t_idx) restrict(amp)
9:   {
10:     int row = t_idx.local[0];
11:     int col = t_idx.local[1];
```

```
12:     tile_static double a_cache[TileSize][TileSize];
13:     tile_static double b_cache[TileSize][TileSize];
14:     double x = 0;
15:     for (int i = 0; i < a.extent[1]; i += TileSize) {
16:       a_cache[row][col] = a(t_idx.global[0], col + i);
17:       b_cache[row][col] = b(row + i, t_idx.global[1]);
18:       t_idx.barrier.wait();
19:       for (int k = 0; k < TileSize; k++)
20:         x += a_cache[row][k] * b_cache[k][col];
21:       t_idx.barrier.wait();
22:     }
23:     c[t_idx.global] = x;
24:   });
25:   c.synchronize();
26: }
```

При использовании разделяемой памяти для приведенной в листинге 11 функции важно, что размеры матриц должны быть кратны размеру блока TileSize. Результаты тестирования приведены для размера блока, равного 16. При другом размере время вычислений может существенно отличаться от приведенного в табл. 11.

Производительность при использовании разделяемой памяти возрастает. Так, например, для проведенных тестов лучший результат – увеличение производительности в 3.45 раза (данные для nVidia GeForce GTX 650M, одинарная точность).

Таблица 11. Результаты тестирования библиотеки на языке C++ (Microsoft Accelerated Massive Parallelism, разделяемая память)

	GFLOPS	***n* = 1024**	***n* = 2048**	***n* = 4096**
Intel HD Graphics 2500	3.741 1.082	0:00:00.625 0:00:02.015	0:00:04.715 0:00:15.875	0:00:36.734 –
AMD Radeon HD 6320	1.878 –	0:00:01.159 –	0:00:09.147 –	– –
nVidia Quadro 2000	49.941 22.272	0:00:00.046 0:00:00.105	0:00:00.344 0:00:00.799	0:00:02.922 0:00:06.171
nVidia GeForce GTX 560 Ti	128.328 56.052	0:00:00.021 0:00:00.046	0:00:00.147 0:00:00.329	0:00:01.071 0:00:02.452
nVidia GeForce GTX 580	183.008 67.504	0:00:00.021 0:00:00.050	0:00:00.118 0:00:00.308	0:00:00.751 0:00:02.036
nVidia GeForce GTX 650M	47.360 21.556	0:00:00.055 0:00:00.120	0:00:00.404 0:00:00.797	0:00:02.902 0:00:06.961
nVidia GeForce GTX 650 Ti	105.237 41.902	0:00:00.026 0:00:00.056	0:00:00.177 0:00:00.410	0:00:01.306 0:00:03.707
nVidia GeForce GTX 720M	6.758 2.928	0:00:00.329 0:00:00.752	0:00:02.542 0:00:05.867	– –
nVidia GeForce GTX 740M	36.092 21.478	0:00:00.065 0:00:00.111	0:00:00.488 0:00:00.827	0:00:03.808 0:00:06.399
nVidia GeForce GTX 870M	107.880 63.394	0:00:00.023 0:00:00.042	0:00:00.172 0:00:00.290	0:00:01.274 0:00:02.168
nVidia GeForce GTX 970	234.138 66.108	0:00:00.013 0:00:00.039	0:00:00.088 0:00:00.283	0:00:00.587 0:00:02.079
nVidia GeForce GTX 980	264.306 77.083	0:00:00.011 0:00:00.034	0:00:00.074 0:00:00.245	0:00:00.520 0:00:01.783

Отметим также, что несмотря на позиционирование технологии Microsoft Accelerated Massive Parallelism как универсальной, подходящей для видеоадаптеров с графическими процессорами nVidia, AMD и Intel, для подавляющего большинства видеоадаптеров на базе AMD, используемых для тестирования (см. табл. 9 и прил. 2), получить результаты не удалось.

Приложение 1. Конфигурации компьютеров

Ниже приведены основные элементы конфигураций компьютеров, которые использовались для тестирования разработанных библиотек матричной алгебры для центральных процессоров. Принят следующий формат: центральный процессор (тактовая частота, число ядер/число потоков), материнская плата, объем оперативной памяти и ее тип.

1. AMD E-450 APU (1.65 ГГц, 2/2), Samsung 305U1A, 4 Гб DDR3.
2. AMD Athlon 64 X2 4000+ (2.1 ГГц, 2/2), WinFast 6100M2MA, 1 Гб DDR2.
3. AMD Phenom II N830 (2.1 ГГц, 3/3), Hewlett-Packard 147B, 2 Гб DDR3.
4. AMD Phenom II X4 925 (2.8 ГГц, 4/4), Asus M4A79XTD Evo, 2 Гб DDR3.
5. Intel Core 2 Duo T9300 (2.5 ГГц, 2/2), Fujitsu Siemens F44, 2 Гб DDR2.
6. Intel Core 2 Quad Q9300 (2.5 ГГц, 4/4), Asus P5N73-AM, 4 Гб DDR2.
7. Intel Core i3 2120 (3.3 ГГц, 2/4), Hewlett-Packard 339A, 4 Гб DDR3.
8. Intel Core i5 3470 (3.2 ГГц, 4/4), Hewlett-Packard 3397, 4 Гб DDR3.
9. Intel Core i7 2600 (3.8 ГГц, 4/8), Asus P8H77-V LE, 8 Гб DDR3.
10. Intel Core i7 4930K (3.9 ГГц, 6/12), Asus P9X79 LE, 16 Гб DDR3.

Приложение 2. Характеристики видеоадаптеров

Далее перечислены использованные при тестировании видеоадаптеры с указанием основных характеристик: графический процессор (тактовая частота, число ядер), объем оперативной памяти и ее тип.

1. Intel HD Graphics 2500 (1150 МГц, 6), 1 Гб DDR3.
2. Intel HD Graphics 5100 (1300 МГц, 40), 1 Гб DDR3.
3. Intel HD Graphics 5200 (1300 МГц, 80), 1 Гб DDR3.
4. AMD Radeon HD 5470 (750 МГц, 80), 1 Гб DDR3.
5. AMD Radeon HD 5670 (775 МГц, 400), 1 Гб GDDR5.
6. AMD Radeon HD 5700 (850 МГц, 800), 1 Гб GDDR5.
7. AMD Radeon HD 6320 (508 МГц, 80), 1 Гб DDR3.
8. AMD Radeon HD 6370 (750 МГц, 80), 1 Гб DDR3.
9. AMD Radeon HD 7450 (625 МГц, 160), 1 Гб GDDR3.
10. nVidiaQuadro 2000 (625 МГц, 192), 1 Гб GDDR5.
11. nVidia GeForce GTX 560 Ti (830 МГц, 384), 1 Гб GDDR5.
12. nVidia GeForce GTX 650M (950 МГц, 384), 2 Гб DDR3.

13. nVidia GeForce GTX 650 Ti (928 МГц, 768), 2 Гб GDDR5.
14. nVidia GeForce GTX 720M (938 МГц, 96), 2 Гб DDR3.
15. nVidia GeForce GTX 740M (980 МГц, 384), 2 Гб DDR3.
16. nVidia GeForce GTX 870M (967 МГц, 1344), 3 Гб GDDR5.
17. nVidia GeForce GTX 970 (1253 МГц, 1664), 4 Гб GDDR5.
18. nVidia GeForce GTX 980 (1216 МГц, 2048), 4 Гб GDDR5.

Разработанные библиотеки матричной алгебры зарегистрированы в реестре программ для ЭВМ: Рыбаков К.А. Библиотека функций матричной алгебры для многоядерных процессоров / Свидетельство о государственной регистрации программы для ЭВМ № 2015610524 от 13 января 2015 г.; Рыбаков К.А. Библиотека функций матричной алгебры для графических процессоров видеоадаптеров / Свидетельство о государственной регистрации программы для ЭВМ № 2015616429 от 9 июня 2015 г.

Работа выполнена при финансовой поддержке РФФИ (проекты № 12-08-00892-а, 16-07-00419-а).

Список литературы

1. Пантелеев А.В., Рыбаков К.А. Прикладной вероятностный анализ нелинейных систем управления спектральным методом. М.: Изд-во МАИ-ПРИНТ, 2010. 160 с.
2. Пантелеев А.В., Рыбаков К.А. Методы и алгоритмы синтеза оптимальных стохастических систем управления при неполной информации. М.: Изд-во МАИ, 2012. 160 с.
3. Рыбин В.В. Моделирование нестационарных непрерывно-дискретных систем управления спектральным методом в системах компьютерной математики. М.: Изд-во МАИ, 2011. 220 с.
4. Рыбин В.В. Моделирование нестационарных систем управления целого и дробного порядка проекционно-сеточным спектральным методом. М.: Изд-во МАИ, 2013. 160 с.
5. Рыбаков К.А. Программное обеспечение спектрального метода Spectrum // Труды МАИ. 2003, № 14.
6. Embarcadero / Delphi. Режим доступа: http://www.embarcadero.com (дата обращения 24.02.2016).
7. OmniThreadLibrary. Режим доступа: http://www.omnithreadlibrary.com (дата обращения 24.02.2016).
8. Gabrijelčič P. Параллельное программирование с OmniThreadLibrary. Leanpub, 2013. 66 с.
9. OpenMP. Режим доступа: http://openmp.org (дата обращения 24.02.2016).
10. Microsoft Development Network. Режим доступа: http://msdn.microsoft.com (дата обращения 24.02.2016).
11. Intel Parallel Studio. Режим доступа: http://software.intel.com (дата обращения 24.02.2016).

12. nVidia CUDA. Режим доступа: http://developer.nvidia.com (дата обращения 24.02.2016).
13. Варыгина М.П. Основы программирования в CUDA. Красноярск: КГПУ, 2012. 138 с.
14. OpenCL. Режим доступа: https://www.khronos.org/opencl (дата обращения 24.02.2016).
15. OpenCL for Delphi. Режим доступа :http://lab4.fme.vutbr.cz/heatlab/OpenCLforDelphi.html (дата обращения 24.02.2016).
16. Аверина Т.А., Рыбаков К.А. Новые методы анализа воздействия эрланговских дельта-импульсов в задачах радиотехники // Журнал радиоэлектроники. 2014, № 11.
17. Рыбаков К.А. Идентификация стохастических систем в спектральной форме математического описания // Идентификация систем и задачи управления (SICPRO'15). X Международная конференция, Москва, 26–29 января 2015 г.: Тр. конф. М.: Институт проблем управления РАН, 2015. С. 1306–1334.

Mathematics & Mathematical Modelling
Electronic journal of the Bauman MSTU
http://mathmjournal.ru

Mathematics and Mathematical Madelling of the Bauman MSTU, 2016, no. 01, pp. 1–27.

DOI: **10.7463/mathm.0116.0841079**

Received: 04.01.2016
Revised: 18.01.2016

Parallel Programming Application to Matrix Algebra in the Spectral Method for Control Systems Analysis, Synthesis and Identification

V.Yu. Kleshnin[1], K.A. Rybakov[1,*]

*rkoffice@mail.ru

[1]Moscow Aviation Institute (National Research University), Moscow, Russia

Keywords: matrix algebra, parallel programming, parallel computing, spectral method

The article describes the matrix algebra libraries based on the modern technologies of parallel programming for the Spectrum software, which can use a spectral method (in the spectral form of mathematical description) to analyse, synthesise and identify deterministic and stochastic dynamical systems. The developed matrix algebra libraries use the following technologies for the GPUs: OmniThreadLibrary, OpenMP, Intel Threading Building Blocks, Intel Cilk Plus for CPUs nVidia CUDA, OpenCL, and Microsoft Accelerated Massive Parallelism.

The developed libraries support matrices with real elements (single and double precision). The matrix dimensions are limited by 32-bit or 64-bit memory model and computer configuration. These libraries are general-purpose and can be used not only for the Spectrum software. They can also find application in the other projects where there is a need to perform operations with large matrices.

The article provides a comparative analysis of the libraries developed for various matrix operations (addition, subtraction, scalar multiplication, multiplication, powers of matrices, tensor multiplication, transpose, inverse matrix, finding a solution of the system of linear equations) through the numerical experiments using different CPU and GPU. The article contains sample programs and performance test results for matrix multiplication, which requires most of all computational resources in regard to the other operations.

References

1. Panteleev A.V., Rybakov K.A. Prikladnoy veroyatnostnyy analiz nelineynykh sistem upravleniya spektral'nym metodom [Applied Probabilistic Analysis of Nonlinear Control Systems by Spectral Method]. Moscow, MAI Press, 2010. 160 p. (in Russian).
2. Panteleev A.V., Rybakov K.A. Metody i algoritmy sinteza optimal'nykh stokhasticheskikh sistem upravleniya pri nepolnoy informatsii [Methods and Algorithms for Synthesis of Optimal Stochastic Control Systems with Incomplete Information]. Moscow, MAI Press, 2012. 160 p. (in Russian).

3. Rybin V.V. Modelirovanie nestatsionarnykh nepreryvno-diskretnykh sistem upravleniya spektral'nym metodom v sistemakh komp'yuternoy matematiki [Modeling of Nonstationary Continuous-Discrete Control Systems by the Spectral Method on Computers]. Moscow, MAI Press, 2011. 220 p. (in Russian).
4. Rybin V.V. Modelirovanie nestatsionarnykh sistem upravleniya tselogo i drobnogo poryadka proektsionno-setochnym spektral'nym metodom [Modeling of Integer-Order and Fractional-Order Nonstationary Control Systems by Projection-Grid Spectral Method]. Moscow, MAI Press, 2013. 160 p. (in Russian).
5. Rybakov K.A. Spectral method software. Trudy MAI = Moscow Aviation Institute Proceedings, 2003, no. 14. (in Russian).
6. Embarcadero / Delphi. URL: http://www.embarcadero.com.
7. OmniThreadLibrary. URL: http://www.omnithreadlibrary.com.
8. Gabrijelčič P. Parallel Programming with OmniThreadLibrary, Leanpub, 2015. 93 p.
9. OpenMP. URL: http://openmp.org.
10. Microsoft Development Network. URL: http://msdn.microsoft.com.
11. Intel Parallel Studio. URL: http://software.intel.com.
12. nVidia CUDA. URL: http://developer.nvidia.com.
13. Varygina M.P. Osnovy programmirovaniya v CUDA [Programming for CUDA]. Krasnoyarsk, KGPU, 2012. 138 p. (in Russian).
14. OpenCL. URL: https://www.khronos.org/opencl.
15. OpenCL for Delphi. URL: http://lab4.fme.vutbr.cz/heatlab/OpenCLforDelphi.html.
16. Averina T.A., Rybakov K.A. New methods for analyzing of Erlang impulses in radio electronics problems, Zhurnal radioelektroniki = Journal of Radio Electronics, 2014, no. 11. (in Russian).
17. Rybakov K.A. Identification of stochastic systems in the spectral form of mathematical description. Identifikaciya sistem i zadachi upravleniya = System Identification and Control Problems (SICPRO'15). Int. conf., Proceedings, Moscow, Institute of Control Sciences (Russian Academy of Sciences), 2015, pp. 1306–1334. (in Russian).