Реализация программного обеспечения для цифровой голографии в среде CUDA

В.И. Гужов, С.П. Ильиных, П.С. Рыжов
НГТУ, Новосибирск, Россия

Аннотация: **в статье рассматривается разработка программного обеспечения для цифровой голографии в среде *CUDA* оптимизированного по критерию времени выполнения.**

Ключевые слова: **цифровая голографическая интерферометрия, *GPGPU, CUDA*, параллелизация алгоритма цифровой голографической интерферометрии, дискретное преобразование Френеля, подавление постоянной составляющей, удаление спекл-шумов.**

ВВЕДЕНИЕ

В данной работе рассматривается применение графического профессора (ГП, *англ. graphics processing unit, GPU*) с программно-аппаратной архитектурой параллельных вычислений *CUDA* (*англ. Compute Unified Device Architecture*), для научных вычислений в сфере цифровой голографической интерферометрии (ЦГИ), которая позволяет существенно увеличить вычислительную производительность [1, 4], благодаря использованию графических процессоров фирмы *NVidia*. Для этого были решены задачи оптимизации таких этапов ЦГИ, как дискретное преобразование Френеля, подавление постоянной составляющей и удаление действительного изображения, устранение фазовой неоднозначности, устранение спекл-шумов и другие этапы. Реализация этапов алгоритма осуществлена для графических карт одного из последних поколений *GeForce* 700 Series, а также *Quadro* K4000, которые оптимизированы для выполнения операций с вещественными числами одинарной точности. Результатом параллелизации программного обеспечения (ПО) является значительное (в 1,5-15 раз) уменьшение времени выполнения шагов алгоритма.

Полученные результаты предоставляют возможность применения алгоритма для цифровой голографической интерферометрии реального времени.

ПОСТАНОВКА ЗАДАЧИ

В результате аналитического обзора и анализа существующих методов голографической интерферометрии [17-29] было выяснено, что наиболее эффективно данная задача может быть решена при помощи *CUDA*-параллелизации метода цифровой голографической интерферометрии, использующего преобразование Френеля.

После исследования возможностей *CUDA Toolkit* 7.0, а также аппаратной архитектуры *CUDA* для параллелизации этапов алгоритма было решено использовать такие модули *CUDA Toolkit*, как *cuFFT* для выполнения дискретного преобразования Фурье, *cuBLAS* для выполнения основных операций линейной алгебры. Также используются *CUDA* ядра, т.е. самописные функции, которые выполняются параллельно на графических процессорах *CUDA*. Этапы цифровой голографической интерферометрии и способ их реализации в разрабатываемом программном обеспечении (ПО) представлены на *Рис*. 1.

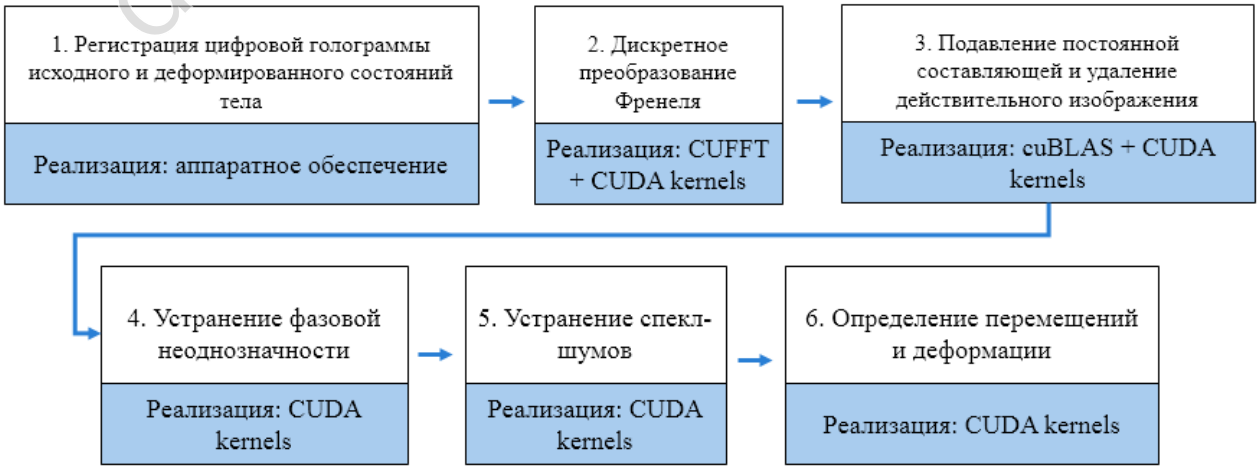


*Рис.*1. Этапы цифровой голографической интерферометрии и их реализации

Алгоритм цифровой голографической интерферометрии детально рассмотрен в следующем разделе.

ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ

Метод цифровой голографической интерферометрии базируется на измерении различий оптических разностей хода света (*рис.*2), вызываемых изменением коэффициента преломления в прозрачных средах или деформацией твёрдых тел [3]. Основой метода является интерференция двух световых полей, являющихся двумя разными состояниями объекта (до и после деформации). Наглядным примером ЦГИ может служить последовательность этапов, представленных на рисунках 2 и 3, где исходный объект – пластинка с нанесённой решетчатой текстурой. Регистрация цифровых голограмм исходного и деформированного состояния тел осуществляется при помощи специальной голографической установки. Голограммы, зарегистрированные до и после деформации пластины, являются исходными данными для дальнейшей цифровой голографической интерферометрии (*рис.* 3).

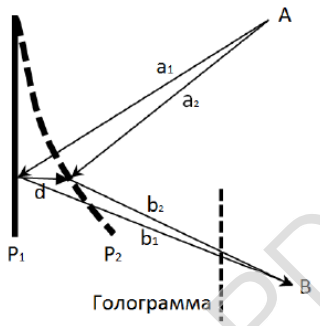


*Рис.*2 – Схема голографической интерферометрии [3].

На рисунке 2 форма объекта исследования до деформации обозначена P_1, после деформации – P_2. При этом в рассматриваемой точке произошло смещение на величину d. Из голограммы восстанавливается объект при помощи света, испускаемого из точки A, а рассматривается из точки B. Оптическая разность хода лучей δ может быть вычислена как

$$\delta(x,y) = \left(a_1(x,y)+b_1(x,y)\right) - \left(a_2(x,y)+b_2(x,y)\right) \quad (1)$$

где a_1, a_2, b_1, b_2 – расстояния, обозначенные на *Рис.* 2.

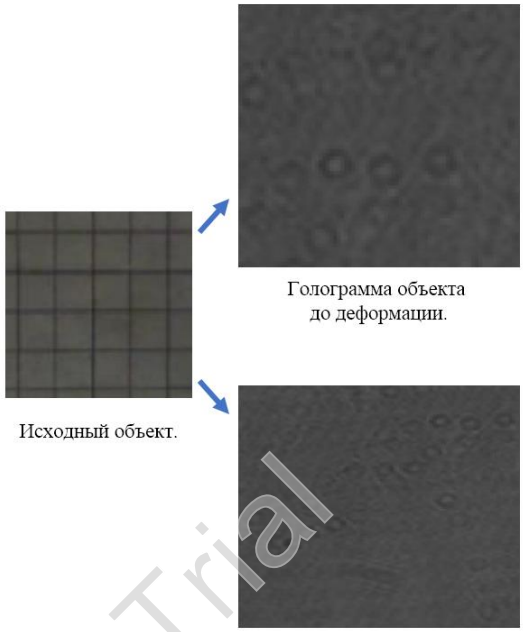


Рис.3 – Регистрация голограмм.

В цифровой голографической интерферометрии строится цифровая голограмма для каждого из состояний объекта, выполняется построение поля разницы фаз, характеризующее смещение исходного объекта (*рис.* 4).

Преобразование Френеля определяется следующим образом:

$$\Phi(v) = \int_{-\infty}^{\infty} E(x) e^{-j(2\pi v + x)^2} dx \quad (2)$$

Введём $\mu = 2v$, тогда

$$\Phi(v) = \int_{-\infty}^{\infty} E(x) e^{(-j(2\pi\mu x)+(-j2\pi v^2)+(-jx^2)} dx = e^{-j\left(\frac{2\pi\mu}{2}\right)^2} \int_{-\infty}^{\infty} \left[E(x) e^{-jx^2}\right] e^{-j2\pi\mu x} dx \quad (3)$$

Таким образом, чтобы выполнить преобразование Френеля функции $E(x)$, её необходимо умножить на e^{-jx^2}, выполнить преобразование Фурье и полученный Фурье-образ $F(\mu)$ умножить на $e^{-j\left(\frac{2\pi\mu}{2}\right)^2}$.

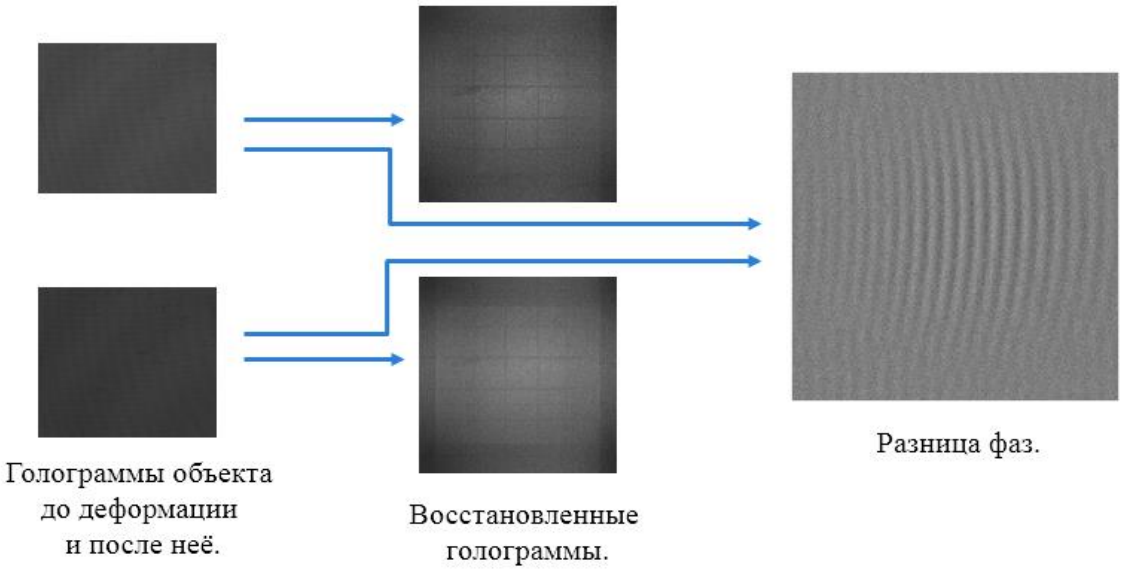


Рис. 4 - Этапы цифровой голографической интерферометрии.

В идеальном случае во внеосевой голографии все составляющие восстановленного изображения оказываются пространственно отделены друг от друга, и достаточно просто выбрать только нужную часть для дальнейшей работы. На практике при малых углах отклонения опорного пучка изображения, находящиеся в разных дифракционных порядках, частично перекрываются. При этом если действительное изображение расфокусировано, то при наложении на мнимое оно будет образовывать зашумление, которое можно подавить вместе с постоянной составляющей. Поэтому при использовании малых углов отклонения опорного пучка важно, чтобы одно из изображений в 1 или -1 порядках дифракции было расфокусировано.

Если рассматривать яркость частей изображения в разных дифракционных порядках, то яркость постоянной составляющей будет на порядок выше остальных. Следовательно, при цифровом восстановлении в процессе нормализации значений большая часть диапазона будет приходиться на постоянную составляющую, что не обеспечит необходимого контраста для выделения остальных составляющих. Визуально постоянная составляющая выглядит как яркий прямоугольник в центре восстановленного изображения, представленный на рисунке 5.

Интенсивность постоянной составляющей в спектральном диапазоне согласно [16] определяется следующим соотношением

$$Z = \sum_{k=0}^{N-1}\sum_{l=0}^{N-1} U_h(k\Delta x, l\Delta y) \qquad (4)$$

где N – размер голограммы в пикселах. Для того чтобы удалить постоянную составляющую авторы предлагают отфильтровать голограмму фильтром частот, эквивалентным средней интенсивности голограммы. Зачастую это означает получение пилообразных функций фазы при голографическом восстановлении

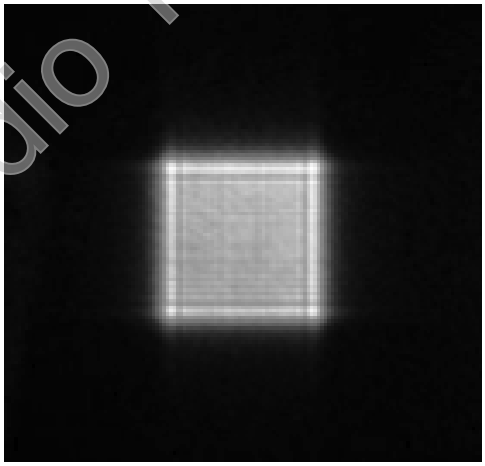


*Рис.*5 – Постоянная составляющая [3].

$$U_h^{'}(k\Delta x, l\Delta y) - \frac{1}{N^2}\sum_{k=0}^{N-1}\sum_{l=0}^{N-1} U_h(k\Delta x, l\Delta y) \qquad (5)$$

Проблемой данного метода является то, что восстановленное изображение также теряет интенсивность. К достоинствам этого метода можно отнести его вычислительную простоту.

Фазы, кратные 2π , при вычислении поля фаз по цифровым голограммам будут неотличимы друг от друга, данное явление называется фазовой неоднозначностью. Алгоритмы устранения фазовой неоднозначности называют алгоритмами развёртывания фазы. На практике это означает получение пилообразных функций фазы при голографическом восстановлении (*рис.* 6).

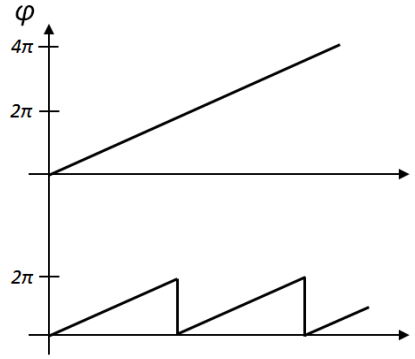


*Рис.*6 - Полная фаза (сверху) и восстановленная фаза (снизу).

Существует большое количество алгоритмов и их модификаций для развёртывания фазы [3]. Многие из них формируют так называемую пороговую функцию, которая показывает, в какой точке функции фазы необходимо сделать скачок на 2π или - -2π (*рис.* 7).

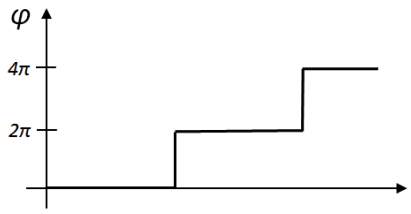


*Рис.*7 - Пороговая функция для развёртывания фазы.

Самым простым способом получения пороговой функции является анализ полученного изображения фазы. Там, где в соседних точках изображения будет скачок значений по модулю больший π, будет находиться один из переходов функции. Знак перехода будет определяться знаком разницы значений в этих точках.

Получить полную фазу по двум восстановленным голограммам можно пользуясь следующей формулой:

$$d(x,y) = \Delta\varphi(x,y)\frac{\lambda}{4\pi} \qquad (6)$$

Таким образом, можно вычислить значение поля смещения в каждой из точек восстановленного изображения и построить трёхмерную модель смещения поверхности. Следует отметить, что полученное поле смещения будет относительное, т.е. представлять собой разницу между двумя состояниями объекта.

Для устранения возникающих при голографическом восстановление спекл-шумов необходим сглаживающий фильтр. Однако пилообразный характер функции разности фаз сильно сужает круг допустимых вариантов. Важно понимать, что при фильтрации места фазовых переходов на $\pm 2\pi$ не должны быть сглажены, иначе алгоритм развёртывания фазы может пропустить фазовый переход. Следовательно, фильтр должен содержать априорную информацию о пилообразном характере функции разности фаз. Фильтр Калмана, широко используемый в областях обработки различного рода сигналов, позволяет задавать априорную информацию о характере системы.

Алгоритм Калмана состоит их двух повторяющихся этапов: экстраполяции и коррекции. На первом этапе делается предсказание состояния системы в текущий момент времени с учётом неточности измерений на основании состояния системы в предыдущий момент времени. На втором, измеренное значение корректирует предсказанное также с учётом неточности и зашумленности.

Рассмотрим математическое представление фильтра Калмана. На этапе экстраполяции вычисляются две величины: предсказанное состояние системы *xk'* и предсказанное значение ошибки ковариации *Pk'*. Напомним, что ковариация – это мера линейной зависимости двух случайных величин. Индекс *k* обозначает текущее состояние, *k-1* – предыдущее. Таким образом, предсказанное состояние системы можно вычислить как

$$x_k^{'} = F_k x_{k-1} + B_k u_{k-1} \qquad (7)$$

где *Fk* – матрица перехода между состояниями, определяющая динамическую модель системы; *xk-1* – состояние системы в предыдущий момент времени; *Bk* – матрица применения управляющего воздействия, которой мы можем пренебречь в нашем случае, поскольку управляющих воздействий нет; *uk-1* – управляющее воздействие в прошлый момент времени.

Рассмотрим одномерный вариант фильтра Калмана применительно к нашему процессу. Как уже было отмечено, измерения *xk* будут представлять отфильтрованные значения разности фаз. Матрицу, определяющую отношение между измерениями и состоянием системы, *Hk* возьмём единичной. Это будет означать, что все измерения (точки функции разности фаз) одинаково влияют на состояние системы (функции разности фаз). Матрица перехода между состояниями *Fk*, будет определять динамику функции разности фаз. Именно в нее и заложим пилообразный характер функции. Тогда *Fk* будет определено как

$$F_k = c_1 \frac{x_{k-1} \bmod 2\pi}{2\pi} + c_2 \qquad (8)$$

где *mod* – операция вещественного остатка от деления, *c1* и *c2* – матрицы настроечных коэффициентов.

Ковариацию шума всего процесса в целом *Qk* и ковариацию шума измерений *Rk* будем подбирать экспериментально, причем для удобства возьмем их одинаковыми на всей протяженности функции. Тогда для фильтрации одномерной функции

разности фаз можно определить следующий фильтр Калмана

$$\begin{cases} x_k^{'}\left(c_1 \cdot \dfrac{x_{k-1} \bmod 2\pi}{2\pi} + c_2\right) x_{k-1} \\ P_k^{'} = \left(c_1 \cdot \dfrac{x_{k-1} \bmod 2\pi}{2\pi} + c_2\right)^2 P_{k-1} + Q \\ x_k = x_k^{'} + \dfrac{P_k^{'}}{P_k^{'} + R}(y_k - x_k^{'}) \\ P_k = \dfrac{P_k^{'} R}{P_k^{'} + R} \end{cases} \quad (9)$$

Предложенный вариант фильтра Калмана является одномерным, вместе с тем возможен и двумерный случай. Для его реализации одномерный фильтр Калмана сначала может применяться для каждой строки изображения, затем для каждого столбца. Таким образом, предложенный вариант фильтра Калмана позволяет эффективно подавлять шумы, возникающие при цифровом голографическом восстановлении. Он не препятствует процессу развёртывания фазы благодаря заданию в фильтре априорной информации о характере функции разности фаз. Данный этап является завершающим.

ПРОФИЛИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В качестве инструмента профилирования используются средства анализа, доступные в интегрированной среде разработки (*англ. Integrated Development Environment, IDE)* Visual Studio [2]. Для того, чтобы обнаружить участки кода, которые выполняются наиболее продолжительное время и/или вызываются с наибольшей частотой, мы запускаем Performance Wizard (мастер производительности) из меню Analyze (анализ) в Visual Studio 2012/2008. В Visual Studio 2005 эта функциональность доступна в Performance Tools (инструменты производительности) из меню Tools (инструменты).) Это запустит трёхэтапный мастер, где первый этап позволяет указать целевой проект или web-сайт. Второй этап предлагает два различных метода профайлинга: sampling (сэмплинг) и instrumentation (инструментирование). После завершения работы мастера будет создана новая сессия оценки производительности. Сессия включает в себя целевое приложение, но на начальном этапе в ней отсутствуют отчёты о производительности. Для того, чтобы начать профайлинг, необходимо нажать кнопку Launch with Profiling (начать профилирование). После того, как будет запущено приложение и выполнена профилируемая задача, если оно не будет закрыто автоматически, то необходимо его незамедлительно закрыть по завершении. Visual Studio автоматически добавляет вновь созданный отчёт о производительности в сессию и начинает анализировать его. По завершении анализа Visual Studio Profiler отобразит Performance Report Summary (отчёт о производительности), который включает в себя наиболее «дорогие» (выполняющиеся во время профайлинга наиболее продолжительно) функции в виде списка (*рис.* 8) и дерева (*рис.* 9). Отчёт показывает эти функции двумя различными способами. Первый измеряет работу, выполненную напрямую или косвенно представленными функциями. Для каждой функции числа представляют накопленное значение для тела функции, а также всех дочерних вызовов. Второй способ не учитывает время, затраченное на дочерние вызовы. Во фрагментах таблиц, показанных на рисунках 8 и 9, столбцы значений “inclusive time” отображают время, или процент от общего времени, затраченного на выполнение тела функции и её дочерних вызовов. Столбцы значений “exclusive time” отображают лишь время, или процент от общего времени, затраченного на выполнение функции без учёта ещё дочерних вызовов. Таким образом можно обнаружить участки кода, занявшие наибольшее время выполнения программы, а затем оптимизировать код, используя стандартные практики [11-13].

Function Name	Number of Calls	Elapsed Inclusive Time %	Elapsed Exclusive Time %	Avg Elapsed Inclusive Time	Avg Elapsed Exclusive Time
holography::ImageLoader::Load	1	33,10	29,22	10 835,59	9 567,14
`vector constructor iterator'	4 096	49,14	22,84	3,93	1,83
alglib::complex::complex	16 777 216	17,45	17,45	0,00	0,00
_RTC_CheckEsp	33 560 576	9,49	9,49	0,00	0,00
holography::Fresnel::Reconstruct	1	56,86	6,33	18 615,42	2 072,31
holography::Experiment::Step	1	61,88	5,02	20 259,07	1 643,65
main	1	100,00	3,75	32 736,94	1 227,77
_FreeImage_GetPixelColor@16	16 777 216	3,13	3,13	0,00	0,00
operator new[]	20 485	1,57	1,57	0,03	0,03
_FreeImage_Load@12	1	0,99	0,99	324,83	324,83
holography::ReferenceBeam::GenerateField	1	0,16	0,16	53,16	53,16

*Рис.*8 - Фрагмент таблицы результатов профайлинга последовательного C++ кода: функции, выполняемые наиболее продолжительное время.

Function Name	Number of Calls	Elapsed Inclusive Time %	Elapsed Exclusive Time %	Avg Elapsed Inclusive Time	Avg Elapsed Exclusive Time
CUDA-holography-project.exe	0	100,00	0,00	0,00	0,00
mainCRTStartup	1	100,00	0,00	32 737,10	0,01
__tmainCRTStartup	1	100,00	0,00	32 737,09	0,06
main	1	100,00	3,75	32 736,94	1 227,77
holography::Experiment::Undertake	1	96,25	0,00	31 509,14	0,24
holography::Experiment::Step	1	61,88	5,02	20 259,07	1 643,65
holography::Fresnel::Reconstruct	1	56,86	6,33	18 615,42	2 072,31
`vector constructor iterator'	4 096	49,14	22,84	3,93	1,83
alglib::complex::complex	16 777 216	17,45	17,45	0,00	0,00
_RTC_CheckEsp	16 781 312	8,85	8,85	0,00	0,00
operator new[]	8 194	1,23	1,23	0,05	0,05
holography::Experiment::LoadImag	1	34,12	0,00	11 168,61	0,00
holography::ImageLoader::Load	1	34,12	0,00	11 168,57	0,00
holography::ImageLoader::Lo	1	33,10	29,22	10 835,59	9 567,14
_FreeImage_GetPixelColor	16 777 216	3,13	3,13	0,00	0,00

*Рис.*9 - Фрагмент таблицы результатов профайлинга последовательного C++ кода: дерево вызовов функций.

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ

Приход многоядерных центральных процессоров (ЦП) и многоядерных графических процессоров (ГП) означает то, что параллельные системы становятся актуальнее как в сфере научных вычислений, так и у обычных пользователей. Кроме того, параллелизм современных систем продолжает масштабироваться согласно закону Мура, что делает актуальной задачей разработку программного обеспечения, которое может работать параллельно, наиболее эффективно используя возрастающее число процессорных ядер. Так, например, графические 3D приложения прозрачно масштабируются на значительно разнящееся число ядер.

Параллельная программная модель CUDA разработана для того, чтобы решить эту задачу, оставляя низкий порог вхождения для программистов, знакомых со стандартным языком программирования, таким, как C. В её основе лежит три основных абстракции – иерархия групп потоков, разделяемая память и барьерная синхронизация. Данные абстракции легко раскрываются программистам, как минимальный набор расширений языка.

Данная декомпозиция сохраняет выразительность языка, позволяя потокам взаимодействовать для решения подпроблем, в то же время делает возможным автоматическую масштабируемость. Действительно, каждый блок потоков может быть использован планировщиком в любом доступном мультипроцессоре ГП, в любом порядке - параллельно или последовательно. Таким образом, скомпилированная CUDA программа может быть выполнена на любом числе мультипроцессоров, как это иллюстрировано на рисунке 10, только исполняющей системе необходимо знать число физических мультипроцессоров.

CUDA C расширяет язык, позволяя программистам определять C функции, называемые ядрами, которые при вызове выполняются N раз параллельно в N различных потоках CUDA, в противоположность обычным C функциям, которые выполняются только один раз при вызове.

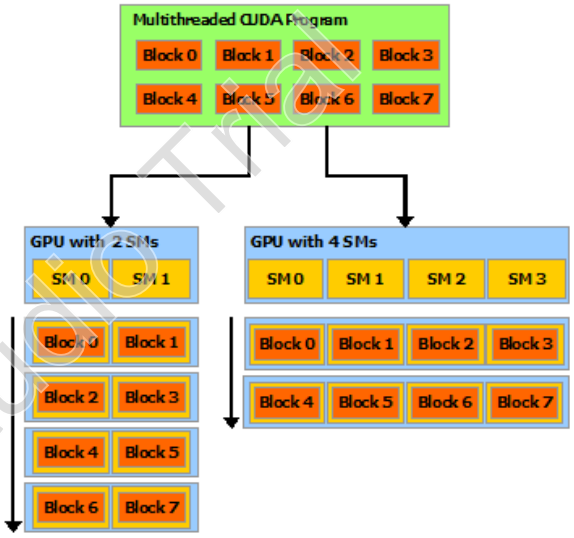


*Рис.*10 – Автоматическая масштабируемость CUDA [4].

Ядра определяются спецификатором объявления __global__, а число CUDA потоков, в которых выполняется ядро, указывается с использованием расширения синтаксиса <<<…>>> (см. расширения языка C[4]). Каждый поток, исполняющий ядро, получает уникальный идентификатор, доступный в пределах ядра через встроенную threadIdx переменную.

В качестве наглядного примера различия в принципе работы параллельного и последовательного кода на листингах 1 и 2 представлены фрагменты кода функции генерации поля опорного пучка, задействованной в разрабатываемом ПО. При использовании стандартного подхода на C++ выполнение функции осуществляется последовательно на процессоре, при использовании CUDA ядра выполнение функции осуществляется параллельно сразу во множестве вычислительных блоков видеокарты CUDA. В данном случае операция выполняется для 65 тыс. элементов, где в последовательном варианте осуществляется проход по двумерному массиву, а в параллельном варианте ядро выполняется параллельно для множества

элементов, индекс которых вычисляется на начальном этапе из имеющихся данных о текущем блоке и потоке, а также о размерности блока.

Листинг 1. Последовательный код C++

```
void GenerateFieldSerial(double ccd_pixel_dist, int size_in_pixel, double angle,
alglib::complex**& output_field) {
   // ...
   for (int i = 0; i < size_in_pixel; i++) {
      double phase_angle = -wave_num * sin(angle) * ccd;
      double line_real = cos(phase_angle) * intensity;
      // ...
      for (int j = 0; j < size_in_pixel; j++) {
         outputField[i][j].x = line_real;
         // ...
      }
      // ...
   }
}

// Выполнение операции для 65 тыс. элементов
GenerateFieldSerial(ccd_pixel_dist, 256, angle, output_field);
```

Листинг 2. Параллельный код CUDA

```
__global__ void GenerateFieldParallel(double ccd_pixel_dist, double angle,
cufftComplex*& outputField) {
   int index = blockDim.x * blockIdx.x + threadIdx.x;
   int i = index / size_in_pixel;
   int j = index % size_in_pixel;
   // ...
   double phase_angle = -wavenum * sin(angle) * (ccd + ccdPixelDist * i);
   double line_real = cos(phase_angle) * intensity;
   // ...
   outputField[i * size_in_pixel + j].x = line_real;
   // ...
}

// Выполнение операции для 65 тыс. элементов
GenerateFieldParallel<<<256, 256>>>(ccd_pixel_dist, angle, output_field);
```

РЕАЛИЗАЦИЯ И ИСПОЛЬЗОВАНИЕ ЯДЕР CUDA

Код ядра CUDA циклического сдвига матрицы, используемый в разрабатываемом ПО, представлен на листинге 3. Сначала вычисляется индекс текущего элемента при помощи имеющихся данных о порядковых номерах блока и потока, на котором выполняется данное ядро. Затем вычисляются индексы элементов матрицы, которые будут смещены. На заключительном этапе выполняется операция обмена для данных элементов.

CUDA является языком, который обеспечивает параллелизм на двух уровнях. Существуют потоки и блоки потоков. Данные параметры задаются при вызове ядра между тремя треугольными скобками $<<< B,T >>>$, где B – число блоков, а T – число потоков в каждом блоке. Архитектура CUDA такова, что пользователю, т.е. программисту, неизвестно, что параллелизм состоит из четырёх уровней. В действительности, блок потоков разделяется на подблоки, называемые варпами (*англ.* warp). Размером варпа является число потоков в варпе, данная единица используется в аппаратное реализации для организации доступа к памяти и отправке инструкций [6].

Блок потоков, планируемый на потоковых мультипроцессорах (ПМ), разделяется на варп. Таким образом, блоки имеют разделяемую память, доступная внутри блока, т.к. находится на одном ПМ. Число блоков каждого ПМ зависит от лимита устройства, и его загруженности. Максимальное число блоков ПМ – 8 для 1.0-2.x версий вычислительной совместимости (ВЧ) и 16 для версий 3.x [7].

Наиболее оптимальным в общем случае является 128 или 256 потоков на каждый блок, тем не менее для каждой конкретной задачи может существовать лучшее большее или меньшее количество потоков на блок. Выбор слишком большого числа потоков может вызвать замедление работы из-за ограниченного числа регистров, выбор слишком малого числа потоков может вызывать сложности с разделяемой памятью, а также количеством блоков из-за ограничения в 8 (или 16)

блоков на каждый ПМ, что приведёт к неэффективному использованию ГП [8].

Ядро циклического сдвига вызывается для половины элементов матрицы, таким образом будет совершено $\frac{N}{2}$ обменов элементов параллельно, где N – количество элементов в матрице.

Листинг 3. Код ядра CUDA циклического сдвига матрицы

```
__global__ void MatrixHalfSizeCyrcularShiftKernel(cufftComplex *matrix, int
size) {
    int index = blockDim.x * blockIdx.x + threadIdx.x;

    int source_i = index / size;
    int source_j = index % size;

    int destination_i = (source_i + (size / 2)) % size;
    int destination_j = (source_j + (size / 2)) % size;

    cufftComplex temp = matrix[destination_i * size + destination_j];
    matrix[destination_i * size + destination_j] = matrix[source_i * size +
source_j];
    matrix[source_i * size + source_j] = temp;
}
```

Листинг 4. Фрагмент C++ кода вызова ядра CUDA

```
#define THREADS_PER_BLOCK 256

void MatrixCUDAUtils::MatrixHalfSizeCyrcularShiftInplace(cufftComplex*& matrix,
int size) {
    int blocks_count = ((size * size) / 2) / THREADS_PER_BLOCK;
    MatrixHalfSizeCyrcularShiftKernel<<<blocks_count,
THREADS_PER_BLOCK>>>(matrix, size);
    cudaError_t error_code = cudaGetLastError();
    if (error_code != cudaSuccess) {
        fprintf(stderr, "Failed to launch MatrixHalfSizeCyrcularShiftKernel
kernel (error code %s)!\n", cudaGetErrorString(error_code));
        exit(EXIT_FAILURE);
    }
}
```

ИСПОЛЬЗОВАНИЕ КОНСТАНТНОЙ ПАМЯТИ CUDA

Современные графические процессоры обладают высоким вычислительным потенциалом. Данное обстоятельство на начальном этапе вызвало значительный интерес к тому, чтобы использовать ГП для вычислений общего назначения вместо ЦП. Зачастую для ГП с большим количеством вычислительных блоков узким местом является не пропускная способность вычислительных блоков, а ширина канала между памятью ЦП и ГП. Из-за большого количество арифметически-логических устройство (АЛУ) на графическом процессоре иногда невозможно поставлять такой объём данных для обработки, чтобы не допускать простоя вычислительных блоков. Таким образом стоит исследовать способы, которыми можно уменьшить объём трафика памяти, необходимого для решения задачи [9].

Язык CUDA предоставляет в распоряжение другой вид памяти, известный как константная память. Из названия становится ясно, что этот вид памяти используется для данных, неизменных в течение выполнения CUDA ядра.

Всего доступно 64 КБ константной памяти на ГП. Константная память кэшируется, таким образом обращение к константной памяти эквивалентно чтению из памяти устройства, только в случае кэш-промаха, в противном случае данная операция будет эквивалентна чтению из константного кэша [4].

Для всех потоков половины варпа, чтение из кэша осуществляется также быстро, как и чтение из регистров, если все потоки обращаются к одному и тому же адресу. Доступ потоков к разным адресам в пределах половины варпа последователен, таким образом трудоёмкость данной операции возрастает линейно с увеличением количества различных адресов, по которым обращаются потоки в пределах половины варпа.

Для того, чтобы объявить переменную в константной памяти, используется ключевое слово __constant__, как это продемонстрировано на листинге 5. Переменные в константной памяти всегда объявляются в глобальной области, таким образом, механизм объявления переменных в константной памяти эквивалентен объявлению переменных в разделяемой памяти.

Функция cudaMemcpyToSymbol() – это специальная версия функции cudaMemcpy(), где осуществляется копирование из памяти устройства в область константной памяти ГП, пример использования данной функции в разрабатываемом ПО представлен на листинге 6.

Операция чтения из константной памяти, пример которой представлен на листинге 7, может быть оттранслирована между потоками одной половины варпа, таким образом эффективно уменьшая общее количество операций чтения. Т.к. константная память кэширована, то последовательное обращение к одному и тому же адресу не увеличит трафик между блоком и памятью устройства.

Константы, известные на момент компилирования программы, должны быть определены при помощи макросов препроцессора (к примеру, #define). Использование переменных в константной памяти устройства обоснованно в случае, если точно известно, что значения не будут изменены в течение времени выполнения ядра. Если же количество констант относительно небольшое, и их значения влияют на ветвления при выполнении ядра, то лучшим вариантом может также оказаться использование шаблонов CUDA ядер, где в качестве параметров шаблонов используются константы [15].

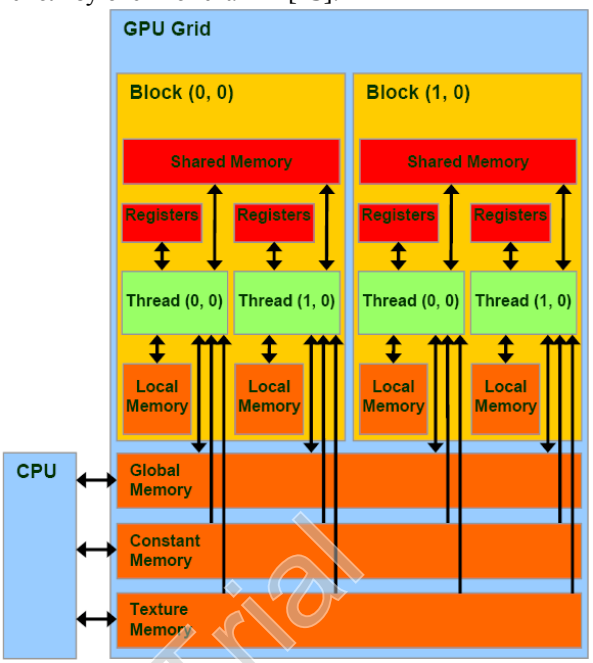


*Рис.*11 – Модель памяти CUDA [9].

Листинг 5. Фрагмент кода глобального пространства

```
#define FACTOR_ARRAY_SIZE 2
__constant__ double device_factor[FACTOR_ARRAY_SIZE];
```

Листинг 6. Фрагмент последовательного C++ кода

```
double host_factor[FACTOR_ARRAY_SIZE];
double host_factor[0] = -HOLO_PI * waveLength * z_distance;
double host_factor[1] = -wavenum * z_distance;
cudaMemcpyToSymbol(device_factor, host_factor, sizeof(double) *
FACTOR_ARRAY_SIZE);
```

Листинг 7. Фрагмент кода CUDA ядра

```
double var_real = cos(device_factor[0] * dist);
double var_img = sin(device_factor[1] * dist);
```

ДВУМЕРНОЕ ДИСКРЕТНОЕ ПРЕОБРАЗОВАНИЕ ФРЕНЕЛЯ СРЕДСТВАМИ CUFFT

Если плоскость голограммы представлена в пространственной области, то при восстановлении преобразование Фурье переведет эту область в частотную. Алгоритм преобразования Френеля обеспечивает простое масштабирование восстановленного изображения, однако это накладывает ряд ограничений на конструкцию измерительной системы, в частности нижняя и верхняя границы допустимого расстояния записи голограммы становятся значимым фактором. Помимо этого, уменьшенное восстановленное изображение объекта занимает меньшую площадь в дискретном цифровом изображении. В ходе вычисления преобразования Френеля преобразование Фурье применяется 1 раз. Из непрерывного преобразования Фурье можно получить дискретное и использовать алгоритм быстрого преобразования Фурье (БПФ).

Библиотека NVidia CUDA Fast Fourier Transform (cuFFT) предоставляет простой интерфейс для вычисления быстрого преобразования Фурье на порядок быстрее, в сравнении с вычислением на ЦПУ. Используя сотни процессорных ядер графических процессоров NVidia, cuFFT обеспечивает высокую производительность вычислений с плавающей точкой на ГП без необходимости разработки собственной реализации БФП на ГП. Широко используемые в различных приложениях от вычислительной физики до обработки изображений и обработки сигналов, cuFFT является эффективным решением для вычисления БФП для комплексных или вещественных наборов данных. Библиотека cuFFT использует алгоритмы, базированные на известных

алгоритмах Кули-Тьюки и Блуштейна, таким образом вы можете быть уверены, что будут получены точные результаты эффективным методом [5].

Т.к. в используемом методе необходимо применить преобразование Фурье для матрицы комплексных чисел, то будет использовано двумерное преобразование Фурье библиотеки cuFFT, реализация которого представлена на листинге 8.

На начальном этапе определяется план выполнения, задаются параметры преобразования. Далее выделяется память на видеокарте для матрицы, затем осуществляется копирование матрицы из оперативной памяти в память видеокарты и выполняется двумерное преобразование Фурье на графической карте. На заключительном этапе данные копируются обратно в оперативную память.

Листинг 8. Дискретное преобразование Фурье с использованием библиотеки функций cuFFT

```
cufftHandle plan2D;
cufftPlan2d(&plan2D, size, size, CUFFT_C2C);
cufftComplex *device_CUDA_data;
checkCudaErrors(cudaMalloc((void**)&device_CUDA_data, sizeof(cufftComplex) *
size * size));
checkCudaErrors(cudaMemcpy(device_CUDA_data, host_CUDA_data,
sizeof(cufftComplex) * size * size, cudaMemcpyHostToDevice));
cufftExecC2C(plan2D, device_CUDA_data, device_CUDA_data, forward ? CUFFT_FORWARD
: CUFFT_INVERSE);
checkCudaErrors(cudaMemcpy(host_CUDA_data, device_CUDA_data,
sizeof(cufftComplex) * size * size, cudaMemcpyDeviceToHost));
cufftDestroy(plan2D);
checkCudaErrors(cudaFree(device_CUDA_data));
```

ОПТИМИЗАЦИЯ СТАТИСТИЧЕСКИХ ВЫЧИСЛЕНИЙ НА CUDA

Как правило, потоки могут безопасно взаимодействовать между собой, только если они существуют на одном блоке. Существует техническая возможность взаимодействий потоков с разных блоков, но это гораздо сложнее, и использование этой возможности делает код склонным к большому количеству ошибок в программе.

В настоящем разделе рассматривается написание ядер, потоки которых безопасно и эффективно взаимодействуют друг с другом для нахождения среднего значения массива вещественных чисел. На CUDA также легко реализовать вычисление таких статистических значений, как стандартное отклонение, среднее, минимум, максимум и т.д., используя методы, подобные приведенному далее решению.

Рассмотренная ниже методика является официально рекомендованным методом выполнения reduction operations (операций по сокращению) [14]. Т.к. для решения задачи потоки должны разделять память, важно правильно сформировать структуру блоков потоков. Переменные должны разделять 8 байт * 256 потоков / блок = 2 КБ / блок. Т.к. ПМ имеет всего 16 КБ shared (разделяемой) памяти, таким образом приемлемо использование 2 КБ разделяемой памяти. Более того, блоки будут состоять из 256 потоков каждый, что даёт 256 x 1 x 1 конфигурацию.

В коде на листинге 9 есть цикл while, в котором с каждой итерацией число активных потоков сокращается вдвое. Подобное сокращение является популярное техникой т.к. может быть параллелизовано. Изначально, блок потоков состоит из 256 потоков, первый раз цикл while выполняется, потоки 0..127 будут сравнивать их средние значения с потоками 128..255. Результаты будут сохранены в разделяемой памятью под индексами 0..127. При следующем проходе цикла только потоки 0..63 будут активны и т.д. Необходимо обратить внимание, что несмотря на тот факт, что большая часть потоков будет простаивать в течение выполнения ядра, поиск среднего значения массива на ГП остаётся значительно быстрее, чем аналогичная операция, выполняемая на ЦП [4].

Также необходимо обратить внимание на то, что в ядре вызывается функция __syncthreads(). Эта функция является барьерной для всех потоков в пределах каждого блока. Ни один поток не может продолжить выполнение, пока все потоки не достигнут участка кода вызова данной функции. Может показаться, что это замедлит выполнение из-за того, что часть потоков будет простаивать из-за того, что достигнет точки синхронизации раньше остальных, но совершенно необходимо синхронизовать потоки. Лишь используя __syncthreads() можно гарантировать, что все потоки каждый раз будут в пределах одной итерации цикла while, таким образом обеспечивая корректное чтение значений из разделяемой памяти. Без вызова __syncthreads() возникает состояние гонки.

Т.к. не существует удовлетворительного способа обмена данными между блоками потоков, результирующие значения будут записаны в global (глобальную) память. Эти значения в дальнейшем могут быть скопированы обратно в оперативную память ПК для окончательной обработки.

Листинг 9. Ядро CUDA усреднение значений в массиве

```
__global__ void getAverageKernel(double *array, double *avg_value) {
    __shared__ double avg[THREADS_PER_BLOCK];

    int array_index = BLOCKS_PER_GRID_ROW * THREADS_PER_BLOCK * blockIdx.y + 256
* blockIdx.x + threadIdx.x;
    avg[threadIdx.x] = array[array_index];
    __syncthreads();
    int nTotalThreads = blockDim.x;

    while(nTotalThreads > 1) {
        int halfPoint = (nTotalThreads >> 1);
        if (threadIdx.x < halfPoint) {
            avg[threadIdx.x] += avg[threadIdx.x + halfPoint];
            avg[threadIdx.x] /= 2;
        }
        __syncthreads();

        nTotalThreads = (nTotalThreads >> 1);
    }

    if (threadIdx.x == 0) {
        avg_value[BLOCKS_PER_GRID_ROW * blockIdx.y + blockIdx.x] = avg[0];
    }
}
```

ЗАКЛЮЧЕНИЕ

В статье описан процесс оптимизации программного обеспечения для цифровой голографии в среде CUDA. Параллелизованная версия алгоритма работает значительно быстрее для средних и больших размеров голограмм (с разрешением от 512 x 512 до 4096 x 4096).

В сравнении с последовательной версией время выполнения операций сократилось от полутора до пятнадцати раз. Время выполнения параллельного алгоритма для голограмм малого разрешения занимает порядка четверти секунды, таким образом делая возможным его применение в системах реального времени. Программа зарегистрирована в фонде алгоритмов и программ Сибирского Отделения РАН [30].

ЛИТЕРАТУРА

[1] http://supercomputingblog.com/cuda/cuda-tutorial-3-thread-communication/

[2] https://msdn.microsoft.com/en-us/magazine/cc337887.aspx

[3] Разработка системы неразрушающего контроля на основе методов цифровой голографической интерферометрии

[4] http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[5] https://developer.nvidia.com/cuFFT

[6] http://stackoverflow.com/questions/3606636/cuda-model-what-is-warp-size

[7] http://stackoverflow.com/questions/16986770/cuda-cores-vs-thread-count

[8] http://stackoverflow.com/questions/10934240/blocks-threads-warpsize

[9] http://cuda-programming.blogspot.ru/2013/01/what-is-constant-memory-in-cuda.html

[10] http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

[11] Рефакторинг. Улучшение существующего кода

[12] Optimizing software in C++ An optimization guide for Windows, Linux and Mac platforms http://www.agner.org/optimize/optimizing_cpp.pdf

[13] http://web.archive.org/web/20080729033434/http://www.abarnett.demon.co.uk/tutorial.html

[14] https://www.sharcnet.ca/help/index.php/CUDA_tips_and_tricks

[15] http://stackoverflow.com/questions/16119923/using-constants-with-cuda

[16] Kreis, T. Suppression of the dc term in digital holography [Text] / T. Kreis, W. Juptner // Optical Engineering. – 1997. – Vol. 36. – P. 2357-2360.

[17] Madrigal R., Acebal P. , Blaya S., Carretero L, Fimia A., Serrano F. GPU-based calculations in digital holography // Holography: Advances and Modern Trends III. – 2013. - Vol. 8776.

[18] S. Lee, H. C. Wey, D. K. Nam and D. S. Park. GPU implementation of wave field translation method for fast hologram generation // Optics and Photonics for Information Processing VIII. – 2014. - Vol. 9216.

[19] Koki Murano1, Tomoyoshi Shimobaba1,, Atsushi Sugiyama1, Naoki Takada1, Takashi Kakue1, Minoru Oikawa1, Tomoyoshi Ito. Fast computation of computer-generated hologram using Xeon Phi coprocessor // Computer Physics Communications. – 2014.

[20] N. Múnera Ortiz, C. A. Trujillo and J. García-Sucerquia. Digital holographic interferometry accelerated with GPU: application in mechanical micro-deformation measurement operating at video rate // 8th Iberoamerican Optics Meeting and 11th Latin American Meeting on Optics, Lasers, and Applications. – 2013. - Vol. 8785

[21] Yasuyuki Ichihashi, Ryutaro Oi, Takanori Senoh, Hisayuki Sasaki, KokiWakunami, and Kenji Yamamoto. A real-time 3D system using electronic holography // SPIE. – 2014.

[22] Mert Dogar, Hazar A. Ilhan, and Meric Ozcan. Real-time reconstruction of digital holograms with GPU // Practical Holography XXVII. – 2013. - Vol. 8644

[23] Koki Muranoa, Tomoyoshi Shimobabaa, Atsushi Sugiyamaa, Naoki Takadab, Takashi Kakuea, Minoru Oikawaa, Tomoyoshi Itoa. Fast computation of computer-generated hologram using Xeon Phi coprocessor // Preprint submitted to Computer Physics Communications. – 2013.

[24] Р.В. Усков. О некоторых особенностях применения технологии CUDA для моделирования переноса излучения // Вестник МГТУ им. Н.Э. Баумана. Сер. “Естественные науки”. 2011. № 3

[25] Гужов В.И. Использование голографической и спекл-интерферометрии при контроле промышленных изделий// Пятая международная научно-техническая конференция "Оптические методы исследования потоков". - Москва: МЭИ.- 1999.- С.107-108.

[26] Цифровая голографическая система реального времени. / Гужов В.И., Ильиных С.П., Хайдуков Д.С., Кабак Е.С. // Сборник научных трудов Новосибирского государственного технического университета, Новосибирск , – 2014.-№4(78) – С. 97 – 112.

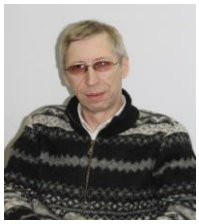
[27] Алгоритм расшифровки интерференционных картин со случайным фазовым сдвигом. / Гужов В.И., Ильиных С.П., Хайдуков Д.С. // Сборник научных трудов Новосибирского государственного технического университета, Новосибирск , – 2014.-№4(78) – С. 79 – 96.

[28] Гужов В.И., Подъяков А.Е., Солодкин Ю.Н., Штейнгольц З.И. Восстановление фазы волнового фронта на основе одномерного преобразования Фурье// Автометрия.- 1992.- №6.- С.21-24.

[29] Цифровая голографическая интерферометрия реального времени для экспериментального исследования напряженно-деформированного сосотояния динамических объектов. / Гужов В.И., Ильиных С.П., Кузнецов Р.А., Кабак Е.С. // Омский научный вестник, Омск , – 2015.-№1(137) – С. 158 – 162.

[30] http://fap.sbras.ru/node/4141.

Владимир Гужов профессор Новосибирского Государственного Технического университета, доктор технических наук. Он является автором 170 научных работ, в том числе является обладателем 4 патентов. Область научных интересов: программные системы, высокоточные измерения.



Сергей Ильиных - доцент Новосибирского Государственного Технического университета, кандидат технических наук. Он является автором более 100 научных трудов, в том числе 1 учебник НГТУ и 4 патента. Область научных интересов: разработка алгоритмов анализа изображений в оптических измерительных системах.

Рыжов Павел Сергеевич – магистрант Новосибирского Государственного Технического университета.

Implementation of Software for Digital Holography in Software CUDA

V.I. Guzhov, S.P. Ilyinikh, P.S. Ryzhov