УДК 681.3

Решение СЛАУ большой размерности на базе графического процессора в задаче построения изоповерхностей

Бабков В.С., Титаренко К.К.
Донецкий национальный технический университет
victor@babkov.name

Abstract

Babkov V., Titarenko K. Solve of LSE based on graphical processor in isosurface building task. *In work realization of LSE solve stage in the task of isosurface construction from 3D-scanning data is considered. Realization of direct and iterative method of decision is considered from point of minimization of time expenses on the system solve. Methods are explored on GPU and CPU. Conclusions about applicability of CUDA technology to the decision of the systems are done.*

Введение

Задача построения изоповерхности по множеству дискретных точек в трехмерном пространстве в области компьютерной графики возникает при обработке результатов 2D и 3D-сканирования [1]. Математически задача сводится к интерполяции в трехмерном пространстве и одним из часто используемых в данном случае методов является метод радиальных симметричных функций (RBF). Разновидности данного метода подробно описаны и проанализированы в работах [2-7]. Основной недостаток метода – значительная временная сложность, связанная с решением СЛАУ большой размерности (10^6-10^9). В работе [8] предложен иерархический метод RBF, который может быть использован для построения поверхностей (2D), а в работе [9] предложена модификация иерархического метода RBF, ориентированная на построение изоповерхностей 3D. Оба указанных метода адаптированы к распараллеливанию.

Поскольку задача построения изоповерхности во многих приложениях (медицина, виртуальная реальность, трехмерное лазерное сканирование) относится к классу задач компьютерной графики, то целесообразным является адаптация методов и алгоритмов для решения с использованием графических процессоров (GPU).

Иерархические методы, предложенные в [8, 9], с точки зрения реализации на GPU, имеют тот недостаток, что GPU – это SIMD-система, а наибольший выигрыш во временных характеристиках иерархический метод может обеспечить только на MIMD-системах.

Поэтому в работе ставится задача исследовать эффективную реализацию этапа решения СЛАУ при построении изоповерхности прямым методом RBF с помощью графического процессора.

Математическая постановка задачи

Как определено в [2], любую изоповерхность в трехмерном пространстве можно представить в виде:

$$f(x, y, z) = 0,$$

где f - это функция вида:

$$f(p) = \sum_{i=1}^{N} \lambda_i \varphi(p, c_i), \qquad (1)$$

где

$\lambda_{1..N}$ - весовые коэффициенты;

φ - радиальная симметричная функция;

$c_{1..N}$ - множество точек в трехмерном пространстве.

Задача построения изоповерхности сводится к поиску функции вида (1). Для этого необходимо найти коэффициенты λ как решение системы вида:

$$\begin{bmatrix} \varphi(c_1,c_1) & \varphi(c_1,c_2) & \dots & \varphi(c_1,c_N) \\ \varphi(c_2,c_1) & \varphi(c_2,c_2) & \dots & \varphi(c_2,c_N) \\ \dots & \dots & \dots & \dots \\ \varphi(c_N,c_1) & \varphi(c_N,c_2) & \dots & \varphi(c_N,c_N) \end{bmatrix} * \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \dots \\ \lambda_3 \end{bmatrix} = \begin{bmatrix} h_1 \\ h_2 \\ \dots \\ h_N \end{bmatrix}, \qquad (2)$$

где

h - значение функции для точек, лежащих на изоповерхности (т.е. 0). Т.к. такое построение системы приводит к тривиальному решению, то для обеспечениияя разрешимости системы вводятся дополнительные условия. На основе

имеющихся точек в пространстве рассчитываются дополнительные, так называемые off-surface точки (см. рис. 1) [3, 6].

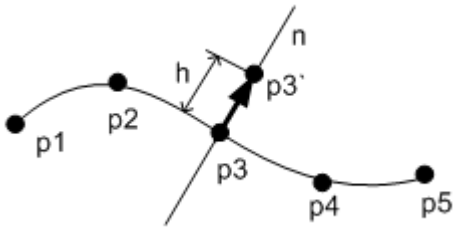

Рисунок 1 – Генерация off-surface точек

Для каждой заданной точки, лежащей на изоповерхности, вдоль нормали откладывается отрезок величины h. Координаты новой точки добавляются к множеству заданных, а соответствующий свободный член системы полагается равным h. Также функция (1) переписывается в виде:

$$f(p) = \sum_{i=1}^{N} \lambda_i \varphi(p, c_i) + P(p),$$

где

$P(p)$ - вспомогательный полином первой степени.

В результате система (2) записывается в виде:

$$\begin{bmatrix} \varphi(c_1,c_1) & \varphi(c_1,c_2) & \dots & \varphi(c_1,c_N) & c_1^x & c_1^y & c_1^z & 1 \\ \varphi(c_2,c_1) & \varphi(c_2,c_2) & \dots & \varphi(c_2,c_N) & c_2^x & c_2^y & c_2^z & 1 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \varphi(c_N,c_1) & \varphi(c_N,c_2) & \dots & \varphi(c_N,c_N) & c_N^x & c_N^y & c_N^z & 1 \\ c_1^x & c_2^x & \dots & c_N^x & 0 & 0 & 0 & 0 \\ c_1^y & c_2^y & \dots & c_N^y & 0 & 0 & 0 & 0 \\ c_1^z & c_2^z & \dots & c_N^z & 0 & 0 & 0 & 0 \\ 1 & 1 & \dots & 1 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \dots \\ \lambda_N \\ p^x \\ p^y \\ p^z \\ 1 \end{bmatrix} = \begin{bmatrix} h_1 \\ h_2 \\ \dots \\ h_N \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Матрица коэффициентов системы обладает свойством симметричности, а при использовании в качестве функции RBF полигармонического сплайна r^3 содержит нули на главной диагонали.

Решение такой системы на графическом процессоре является предметом исследования данной работы.

GPU как инструмент для параллельной реализации интенсивных вычислительных процедур

Ориентация на высококачественную трехмерную графику реального времени превратила современное программируемое графическое процессорное устройство (GPU) высокопараллельное, мультипотоковое и многоядерное устройство с огромными вычислительными мощностями и высокой пропускной способностью памяти (см. рис. 2) [10].

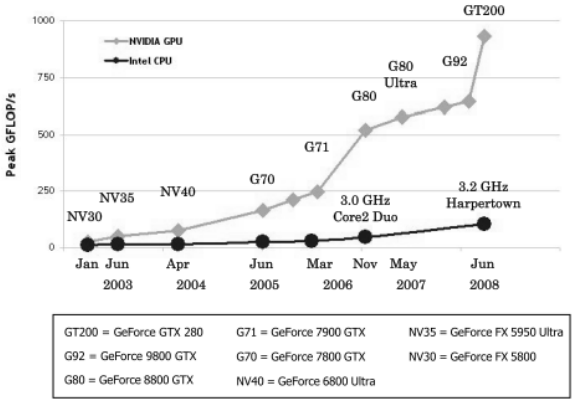

Рисунок 2 – Рост пиковых вычислительных мощностей CPU и GPU, flops

Такая разница вычислительных мощностей CPU и GPU вызвана тем, что GPU предназначен для высокоинтенсивных вычислений (рендеринг графики), и поэтому разработан таким образом, что большее количество транзисторов предназначено для непосредственно обработки данных, чем для кэширования и выполнения инструкций передачи управления (см. рис. 3).

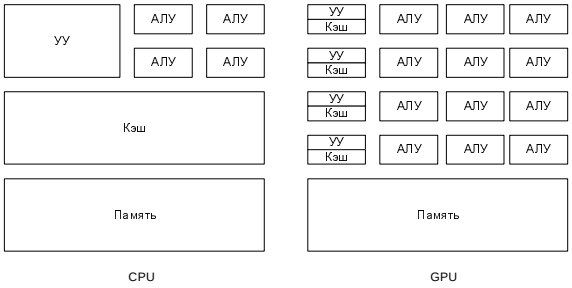

Рисунок 3 – Сравнение CPU и GPU

Наилучшим образом, графическое процессорное устройство предназначено для параллельной обработки данных, предполагающей выполнение одних и тех же операций над каждым элементом набора. Одна и та же вычислительно-интенсивная программа (отношение арифметических операций к операциям с памятью выше единицы) параллельно вызывается для каждого элемента из набора данных, в виду чего нет надобности в сложных механизмах передачи управления, наличии большого объема кэша, а также это позволяет снизить время задержки при доступе к памяти.

Для практической реализации параллельных алгоритмов на GPU наиболее распространенной в данный момент технологией является технология CUDA [10].

CUDA (Compute Unified Device Architecture) – модель параллельного программирования, а также программная среда, предназначенная для предоставления разработчикам возможности использования мощностей параллельных мультипроцессоров видеокарты для вычислений общего назначения. Являясь расширением языка программирования Си, данная модель позволяет снизить затраты времени и усилий на обучение, а также на разработку приложений.

В основе модели лежат три ключевых абстракции – иерархия групп потоков, разделяемая память и барьерная синхронизация. Эти абстракции позволяют программисту разбить задачу на несколько подзадач, которые будут выполняться независимо и параллельно, а затем на еще более мелкие подзадачи, которые смогут выполняться совместно и параллельно.

Несомненным преимуществом, помимо параллельного выполнения, является переносимость. Приложения CUDA отличает высокая кроссплатформенность: переносимость на ряд операционных систем, а также отсутствие необходимости учитывать специфику конкретного устройства, на котором будет выполняться скомпилированное приложение.

Типовая архитектура GPU, представленного моделью CUDA, выглядит так: см. рис. 4. Архитектура базируется на масштабируемом массиве потоковых мультипроцессоров. Блоки потоков сетки распределяются по мультипроцессорам, потоки блока потоков исполняются параллельно на одном мультипроцессоре. Как только блок завершился, на его место назначается новый и т. д.

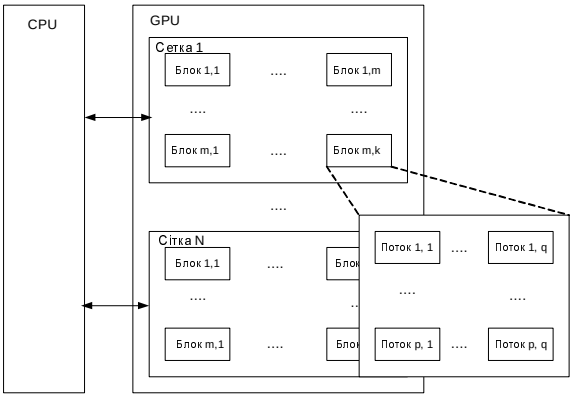

Рисунок 4 – Типовая архитектура GPU CUDA

Мультипроцессор содержит 8 скалярных ядер, два специальных блока для вычисления трансцендентных функций, многопоточный блок инструкций и находящуюся на том же кристалле разделяемую память (on-chip shared memory). Он создает, управляет и запускает параллельные потоки с нулевыми задержками на планирование.

Для управления сотнями потоков, мультипроцессор (МП) задействует новую архитектуру SIMT (single instruction – multiple threads). МП отображает каждый поток на одно скалярное ядро, и каждый поток исполняется независимо от других, со своим указателем инструкции и собственными состояниями регистров. SIMT-блок МП занимается планированием и управлением потоками группами по 32, называемыми ворпами (warps). Различные потоки, составляя SIMT-ворп, начинают исполняться с одного и того же адреса, но могут свободно ветвиться и выполняться независимо.

Получая на вход блоки потоков, МП разбивает их на ворпы и далее планированием занимается SIMT-блок. Каждый раз во время выдачи инструкции, SIMT-блок выбирает ворп, готовый к выполнению и выдает инструкцию всем активным потокам ворпа. Ворп исполняет одну общую инструкцию в момент времени, в виду чего максимальная эффективность достигается, когда все 32 потока следуют одному пути исполнения. Если же возникают ветвления, ворп последовательно выполняется для каждого ветвления, после чего выполняется возврат к одному пути исполнения. Расхождения при ветвлении происходят только внутри ворпа, различные ворпы выполняются независимо.

SIMT архитектура имеет достаточно преимуществ в сравнении с SIMD (single instruction – multiple data), позволяя ветвления, не ограничивая способы организации памяти и т. д.

Реализация решения СЛАУ методом LDLT-разложения

Метод LDLT предполагает симметричность исходной матрицы коэффициентов, что соответствует рассматриваемому типу матриц. Алгоритм реализации метода LDLT можно найти, например, в [11].

Анализ алгоритма показывает, что наиболее ресурсоемким участком являются векторные операции, а именно операция сложения векторов с масштабированием.

Ввиду того, что алгоритм метода LDLT содержит достаточно большое число векторных операций (сложение, перестановки, умножение), допустимо сделать предположение, что, реализовав эти операции с помощью CUDA, можно достичь некоторого ускорения. При этом существенной проблемой, которая может повлиять на производительность, является низкая скорость

доступа к памяти GPU, а также высокая интенсивность вызовов векторных операций.

Так как LDLT метод не является параллельным, а подразумевает лишь интенсивное использование векторных операций, которые в свою очередь могут быть распараллелены, возникают два способа его реализации с помощью CUDA. Первый предполагает хранение данных в памяти GPU, что означает перенос практически всех функциональных блоков (в том числе и не подлежащих параллельному исполнению) на GPU. Второй способ предусматривает загрузку в память GPU данных, непосредственно необходимых для данной векторной операции, и их выгрузку после завершения операции. Первый способ реализован как модификация алгоритма [11], в котором функциональные блоки заменены соответствующими подфункциями CUDA. Второй способ предполагает реализацию на CUDA операции сложения векторов с масштабированием (как наиболее интенсивно используемую – 89,60%), сопряженную с перемещением данных между основной памятью и памятью GPU.

Графики времени выполнения в зависимости от размерности приведены на рисунке 5.

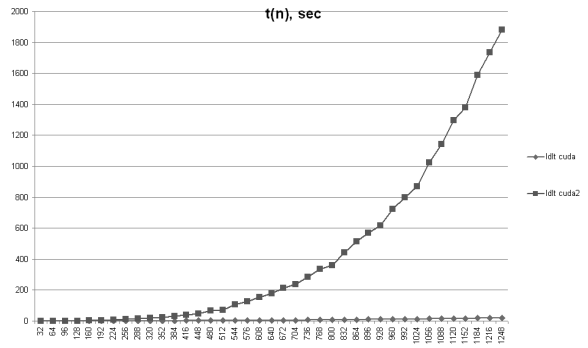

Рисунок 5 – Зависимость времени решения от размерности системы для CUDA-реализаций

График на рис. 5 отображает существенное отставание в производительности второго способа реализации.

Причины падения производительности можно выяснить, проанализировав отчет профайлера, рис. 6.

В результате анализа можно сделать вывод, что серьезной проблемой является низкая скорость передачи данных. И хотя ядро (функция исполняется на GPU), выполняющее сложение векторов с масштабированием, можно оптимизировать, к желаемому повышению производительности это не приведет – более 70% времени (и этот показатель растет с ростом размерности задачи) тратится не на вычисления, а на передачу данных.

Проведем сравнение результатов первой модификации и исходной реализации метода LDLT. График зависимости времени от размерности приведен на рис. 7.

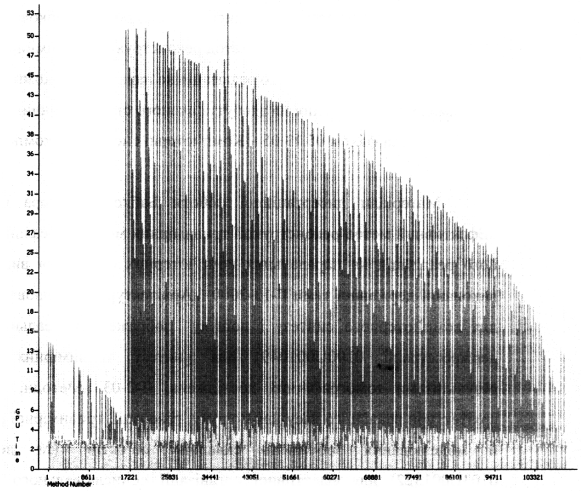

Рисунок 6 – Хронология использования GPU

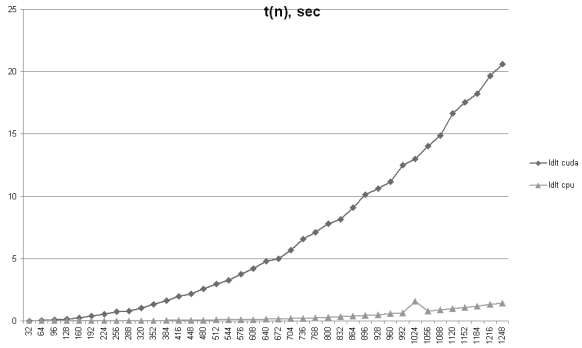

Рисунок 7 – Зависимость времени решения от размерности системы для GPU и CPU

График на рис. 7 иллюстрирует, что первая модификация также не является более производительной, чем исходный метод. Это обусловлено тем, что при условии размещения данных в памяти GPU (в целях исключения траты времени на их передачу) приходится переносить на GPU полностью непараллельные функциональные блоки, некоторые из которых интенсивно используют предикаты, что неизбежно для LDLT, а также невозможностью сведения операций доступа к памяти только к объединенному (coalesced) чтению/записи, что продиктовано спецификой LDLT и существенно влияет на производительность. Также специфика метода предполагает частый вызов ядер с малой загрузкой за один вызов, что негативно сказывается в виде повышения накладных

расходов на запуск ядер. Это подтверждается графиком на рис. 8.

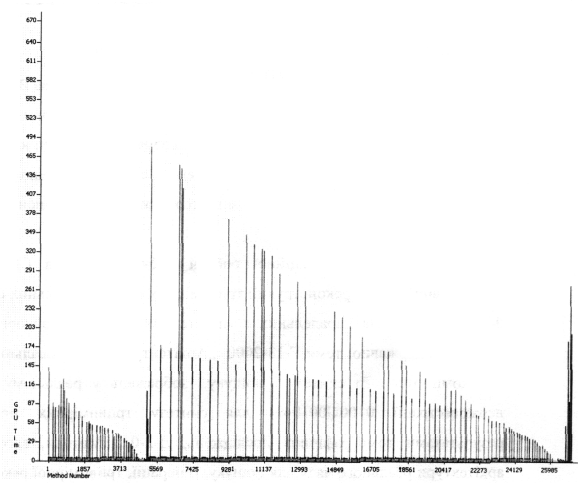

Рисунок 8 – Хронология использования GPU

Преобладание низких столбцов (значения, не превышающие 10) подтверждает предположение о частых вызовах ядер с минимальной загрузкой, обеспечивая накладные расходы на вызов ядер, что вносит существенный вклад в общую производительность, снижая ее. Таким образом, даже дальнейшая оптимизация ядер не даст достаточного прироста производительности, что позволяет сделать вывод о неприемлемости использования обеих модификаций с применением CUDA для реализации метода LDLT ввиду его специфики и рассмотренных проблем.

Реализация решения СЛАУ итерационным методом Якоби

Основным требованием метода Якоби является диагональное преобладание матрицы коэффициентов для обеспечения сходимости [11]. При условии преобразования исходной матрицы к необходимому виду, решение СЛАУ можно осуществить данным итерационным методом.

Основным процессом метода Якоби является осуществление очередной итерации, что представляет собой умножение преобразованной матрицы коэффициентов на вектор с последующим сложением с преобразованным вектором свободных членов, что может производиться параллельно. Также данный процесс позволяет оптимизировать доступ к памяти, обеспечив объединенное (coalesced) чтение и объединенную запись.

Далее на рис. 9-10 приводятся графики производительности метода Якоби (21 итерация), реализованного с помощью CUDA и базовой реализации метода LDLT, применяемых к одной и той же тестовой СЛАУ.

Согласно графику (рис. 11) выигрыш в производительности значительно возрастает с ростом размерности задачи, что предполагалось ввиду высокой параллельности метода, а также возможности оптимизировать доступ к памяти GPU.

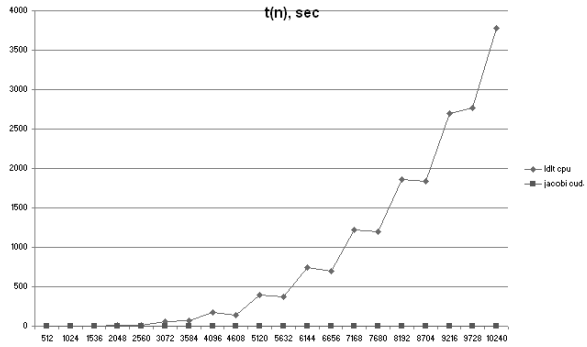

Рисунок 9 – Зависимость времени решения от размерности системы для LDLT и метода Якоби

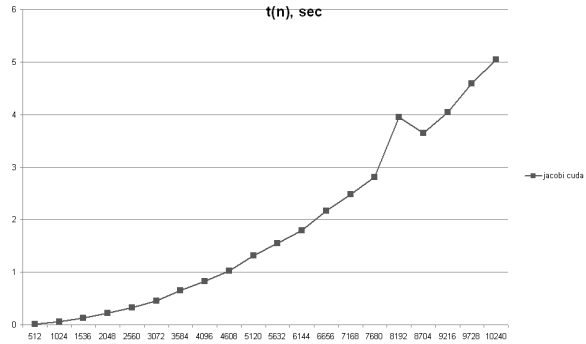

Рисунок 10 – Зависимость времени решения от размерности для метода Якоби

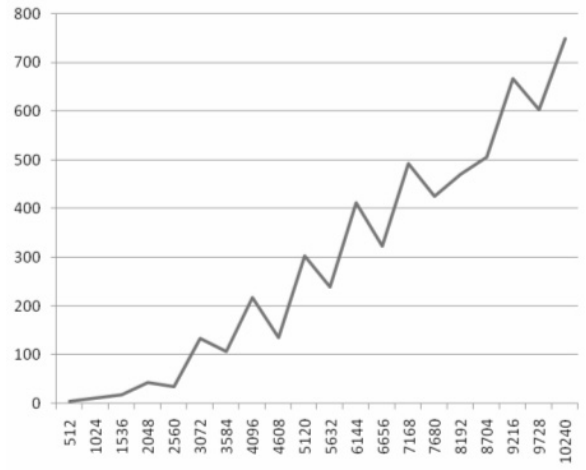

Рисунок 11 – Выигрыш в производительности (в разах)

При использовании итерационного метода необходимо исследовать погрешность метода (см. рис. 12).

Под погрешностью здесь подразумевается максимальный модуль разности между исходными корнями и полученными (тестовая СЛАУ строилась при известных корнях).

Полученный график иллюстрирует, что точность в обоих случаях падает с ростом размерности, однако итерационный метод показывает лучший результат (количество итераций фиксировано и составляет 21 итерацию), чем прямой.

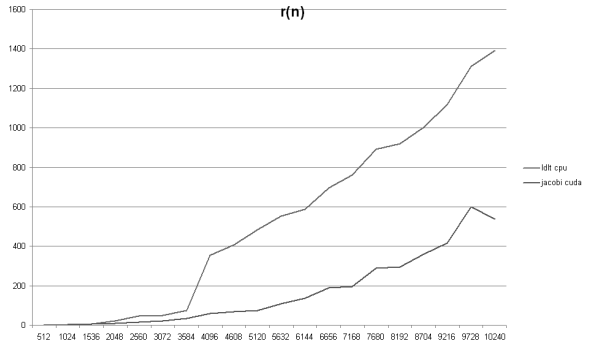

Рисунок 12 – График зависимости погрешности от размерности системы

Реализация решения СЛАУ на базе CPU

В качестве альтернативы последовательному решению на CPU также был рассмотрен вариант многопотокового выполнения на разных ядрах многоядерного CPU.

Результат показан в таблице 1. В таблице использованы следующие обозначения:

Cuda – решение с использованием CUDA;

Cuda2 – реализация на CUDA только векторно-матричных операций;

Cpu – решение на CPU в один поток;

Cpu_mt – решение на CPU в несколько потоков.

Из таблицы можно сделать вывод, что худшие результаты при многопотоковом выполнении обусловлены затратами на запуск потока (как известно, это одна из самых ресурсоемких операций).

Выводы

В ходе выполнения работы были рассмотрены реализации на GPU двух методов решения СЛАУ: прямого – метода LDLT-разложения, и итерационного – метода Якоби. Для метода LDLT были реализованы две модификации, а метод Якоби был полностью реализован с применением CUDA.

В ходе рассмотрения эффективности реализаций различных методов было выяснено, что использовать CUDA целесообразно только для высокопараллельных методов, допускающих возможность оптимизации доступа к памяти.

Таблица 1 – Результаты эксперимента на тестовой СЛАУ

Кол. точек	Время решения, с				
	LDLT				Якоби
	cuda	cuda2	cpu	cpu_mt	cuda
32	0,016	0,094	0	0,062	0
64	0,047	0,359	0	0,281	0
96	0,094	1,032	0	0,672	0
128	0,172	1,875	0	1,188	0,016
160	0,265	3,359	0	1,89	0
192	0,375	5,266	0	3,172	0
224	0,563	8,453	0,016	4,063	0,016
256	0,718	12	0,016	4,922	0,016
288	0,766	14,407	0	6,297	0
320	1,016	18,953	0,016	7,766	0
352	1,313	23,844	0,016	9,407	0,015
384	1,641	31,234	0,032	11,188	0,015
416	1,984	41,375	0,031	13,172	0,016
448	2,188	49,188	0,047	15,312	0,016
480	2,578	65,406	0,047	17,563	0,015
512	2,969	70,234	0,094	20,016	0,031
544	3,266	106,609	0,078	22,64	0,016
576	3,766	126,766	0,094	25,579	0,031
608	4,218	154,344	0,093	28,344	0,015
640	4,782	176,312	0,141	31,5	0,016
672	5	214,734	0,141	34,75	0,031
704	5,688	238,329	0,203	38,39	0,015
736	6,547	285,328	0,219	42,047	0,031
768	7,125	334,484	0,265	45,39	0,031
800	7,813	360,719	0,297	48,14	0,047
832	8,172	442,343	0,344	51,766	0,047
864	9,078	513,484	0,375	55,813	0,046
896	10,125	569,75	0,469	60,796	0,063
928	10,641	616,641	0,5	64,703	0,047
960	11,157	721,672	0,578	70,187	0,047
992	12,516	799,032	0,656	75,11	0,062
1024	12,984	868,485	1,594	80,563	0,047
1056	14,031	1025,42	0,782	84,859	0,047
1088	14,89	1144,13	0,891	89	0,063
1120	16,672	1295,89	0,969	94,954	0,062
1152	17,546	1379,36	1,094	100,11	0,062
1184	18,234	1587,84	1,188	105,782	0,078
1216	19,672	1737,03	1,328	113,563	0,093
1248	20,609	1882,89	1,421	118,953	0,094

Прирост производительности в таком случае обеспечивается возможностью параллельного вычисления сразу нескольких компонент решения, независимо друг от друга, а также возможностью объединенного доступа к памяти (coalesced access).

Так как. в задаче реконструкции изоповерхности в реальном времени ставятся жесткие временные ограничения, а матрица системы обладает свойством симметричности, но не гарантирует положительной определенности, диагонального преобладания и т.п. качеств, то целесообразны два подхода:

- прямое решение LDLT со значительным увеличением временных затрат;
- итерационное решение, но с предварительной подготовкой матрицы к виду, обеспечивающему сходимость метода.

При этом использование GPU как платформы для решения СЛАУ оправдано только в случае итерационного метода решения и значения $t < 0.1$ с для систем с $n \to 10^4$ являются удовлетворительным результатом для реконструкции изоповерхности в реальном времени.

Среди дальнейших направлений деятельности можно отметить рассмотрение других итерационных методов, допускающих параллельность и оптимизацию доступа к памяти, а также поиск способов приведения исходной матрицы к виду, позволяющему применить тот или иной итерационный метод решения СЛАУ с целью выигрыша в производительности для СЛАУ больших размерностей.

Также целесообразным представляется рассмотрение многопотоковой реализации LDLT метода на CPU с оптимизацией управления потоками и применением SSE инструкций процессора.

Литература

1. Бабков В.С. 3D-моделювання об’єктів на основі 2D та 3D-проекційних даних / В.С. Бабков // Матеріали IV науково-практичної конференції „Донбас-2020: наука і техніка – виробництву”, 27-28 травня 2008 р. – Донецьк: ДонНТУ Міністерства освіти і науки, 2008. – С. 383–387
2. Function representation in geometric modeling: concepts, implementation and applications / A. Pasko, V. Adzhiev, A. Sourin, V. Savchenko // The Visual Computer, 1995. – V.11, №8. – P. 429–446
3. Reconstruction and Representation of 3D Objects with Radial Basis Functions / J.C. Carr, R.K. Beatson, J.B. Cherrie [et al.] // ACM SIGGRAPH, August 12-17, 2001. – Los Angeles, USA. – P. 67–76
4. Turk G. Variational implicit surfaces / G. Turk, J.F. O’Brien // Technical Report GIT-GVU-99-15, 1998. – Georgia Institute of Technology, USA
5. Morse B.S. Interpolating implicit surfaces from scattered surface data using compactly supported radial basis functions / B.S. Morse, T.S. Yoo, P. Rheingans [et al.] // SMI 2001 International Conference, May, 2001. – P. 89–98
6. Surface rendering for parallel slice of contours from medical imaging / W. Qiang, Z. Pan, C. Chun, B. Jiajun // Computing in science & engineering, 2007. – V.9, №1. – P. 32–37
7. Башков Е.А. Исследование возможностей применения RBF-алгоритма и его модификаций для построения поверхностных компьютерных моделей в медицинской практике / Е.А. Башков, В.С. Бабков // Сборник трудов международной конференции "Моделирование-2008", 14-16 мая 2008 г. – Киев: Институт проблем моделирования в энергетике им. Г.Е. Пухова, т. 1, 2008. - С. 166–171
8. Pouderox J. Adaptive hierarchical RBF interpolation for creating smooth digital elevation models / J. Pouderox [et al.] // Proc. 12-th ACM Int. Symp. Advances in Geographical information Systems, 2004. – ACP Press. – P. 232–240
9. Бабков В.С. Модифікація ієрархічного методу RBF для отримання 3D-моделей за результатами лазерного сканування / В.С. Бабков // IV Міжнародна науково-практична конференція „Сучасні проблеми і досягнення в галузі радіотехніки, телекомунікацій та інформаційних технологій”: тези доповіді, 24-26 вересня, 2008 р. – Запоріжжя, ЗНТУ. – С. 116–117
10. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. Version 2.0 // NVidia Corp., 2008
11. Golub G.H. Matrix computations / G.H. Golub , C.F. Van Loan: 3-rd ed. – London: John Hopkins University Press, 1996. – 687 p.

Поступила в редколлегию 15.03.2009