УДК 004.93:004.43

ПАРАЛЛЕЛЬНЫЙ АЛГОРИТМ ПОИСКА МАКСИМУМА ЦЕЛЕВОЙ ФУНКЦИИ МЕТОДОМ ДИНАМИЧЕСКОГО ПРОГРАММИРОВАНИЯ С ПОМОЩЬЮ ТЕХНОЛОГИИ CUDA

Э.Н. СЕРЕДИН

Объединенный институт проблем информатики
Сурганова, 6, Минск, 220012, Беларусь

Поступила в редакцию 28 января 2016

Предлагается параллельный алгоритм поиска максимума целевой функции с помощью технологии программирования видеокарт CUDA на основе модифицированного метода динамического программирования. Описываются основные особенности алгоритма, позволившие сократить на несколько порядков количество требуемых вычислений и объем используемой памяти. Приводятся оценки быстродействия версий алгоритма для выполнения на процессоре и видеокарте.

Ключевые слова: параллельный алгоритм, максимум целевой функции, метод динамического программирования, технология программирования видеокарт CUDA.

Введение

До сих пор актуальной является задача разработки алгоритмов поиска глобального максимума целевой функции. Наиболее известными являются алгоритмы, построенные на основе градиентных методов, алгоритмов случайного поиска, алгоритмов имитации отжига, генетических и эволюционных алгоритмов [1–3]. При исследовании и тестировании ранее разработанного алгоритма интерактивного выделения линейных объектов на аэрофотоснимках и космических изображениях [4–5] необходимо было подобрать алгоритм поиска глобального максимума целевой функции, который смог бы эффективно заменить переборный алгоритм.

Суть алгоритма выделения линейных объектов состоит в следующем: пользователь в интерактивном режиме выделяет два пиксела a и b, которые ограничивают участок линейного объекта и соединяются прямой линией. Далее отрезок $[a,b]$ разбивается на заданное число узлов, в которых затем строятся направляющие, перпендикулярные данному отрезку. С учетом ограничения угла автоматически среди всех возможных выбирается та ломаная, которая в наибольшей мере заполнена градиентом изображения, ортогональным к звеньям этой ломаной. Найденная ломаная принимается в качестве приближения участка линейного объекта. На рис. 1 показан пример нахождения оптимального решения с учетом ограничения максимального допустимого угла между соседними ломаными линиями.

На шагах 6 и 7 алгоритма выделения линейных объектов необходимо найти максимальную сумму модулей синусов углов между градиентом изображения и отрезком ломаной при полном переборе всех возможных путей из точки a в точку b с учетом допустимого максимального угла между отрезками соседних ломаных. Тестирование реализации алгоритма полного перебора всех возможных вариантов на процессоре показало абсолютную ее неприменимость на практике из-за очень большого времени вычислений, которое для восьми ядер процессора i7-4770K составило приблизительно 50 мин для 6 узлов и 512 точек на каждой направляющей без ограничения угла.

Большинство из известных алгоритмов поиска глобального максимума целевой функции оказались непригодны для использования из-за ряда наложенных требований и ограничений

(количество операций, объем используемой памяти, быстродействие, ограничение угла). Поэтому было решено разработать новый алгоритм поиска глобального максимума целевой функции, который учитывал бы все предъявляемые требования и ограничения.

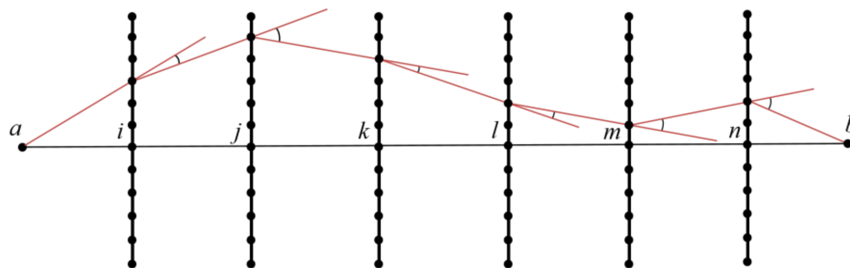

Рис. 1. Пример нахождения оптимального решения алгоритмом выделения линейных объектов

В данной работе предлагается параллельный алгоритм поиска максимума целевой функции методом динамического программирования. Использование алгоритма позволяют уменьшить на несколько порядков количество требуемых вычислений, сократить время его выполнения, а также объем одновременно используемой памяти. Представленный параллельный алгоритм специально разработан для использования технологии программирования видеокарт CUDA [6–8] или любых других технологий с поддержкой принципа «одиночный поток команд – множественное число обрабатываемых потоков».

Общее описание алгоритма поиска максимума целевой функции

Способы решения проблемы ускорения алгоритма традиционно сводятся к двум основным подходам: уменьшению количества необходимых операций и поиску или разработке более эффективной программной реализации алгоритма. Представленный алгоритм разрабатывался с учетом обоих подходов. Уменьшения количества операций удалось достигнуть за счет разработки алгоритма, основанного на модифицированной версии метода динамического программирования, который позволил находить приближения ломаными заданной кривизны за $O(d^3(n-2)+d^2)$ операций. Приведем краткое описание алгоритма на примере шести направляющих (рис. 1). Координаты точек на направляющих перпендикулярных отрезку $[a,b]$ обозначим через i,j,k,l,m,n.

Алгоритм. Вычисляем суммы модулей синусов углов $S_i, S_{i,j}, S_{j,k}, S_{k,l}, S_{l,m}, S_{m,n}, S_n$ между градиентом изображения и ломаной линией для всех возможных отрезков и сохраняем их. Затем вычисляем максимальные суммы $S^*_{i,j} = S_i + S_{i,j}$ и сохраняем координаты точки i для всех значений $S^*_{i,j}$. Аналогично, переходя к следующим направляющим, вычисляем оставшиеся суммы с сохранением координат промежуточных точек. Результатом выполнения всех действий будет нахождение максимальной суммы. Затем по ранее сохраненным координатам для этой максимальной суммы строим ломаную линию, приближающую линейный объект. Подробное описание алгоритма приводится в статье [4].

Разработка нового параллельного алгоритма поиска максимума целевой функции и его программной реализации на видеокарте с применением технологии программирования CUDA позволила положительно ответить на вопрос о возможности ускорения работы алгоритма за счет переработки алгоритмической и программной его частей.

Параллельный алгоритм поиска максимума целевой функции на CUDA

Операции копирования из оперативной памяти в память видеокарты и наоборот на сегодняшний день являются одними из самых ресурсоемких при работе с технологией CUDA. Предложенный метод динамического программирования позволил не только сократить общее число необходимых операций, но и сократить объемы копирования и одновременного использования памяти на каждом из шагов алгоритма. Теперь запуск алгоритма стал возможен и на малобюджетных видеокартах с небольшим количеством памяти.

Для реализации описанного метода динамического программирования потребовалась разработка алгоритма, учитывающего специфику технологии CUDA. Рассмотрим основные подходы построения алгоритма на примере восьми точек разбиения (рис. 2).

Первое, на что можно обратить внимание – это то, что вычисление промежуточных сумм можно производить одновременно двигаясь из точки a в точку b и наоборот. Вычисление сумм $S^*_{i,j,k} = S_i + S_{i,j} + S_{j,k}$ и $S^*_{n,m,l} = S_n + S_{m,n} + S_{l,m}$ производится идентично. Отрезок $[a,b]$ целесообразно разбивать на равные части, то же касается и точек на направляющих линиях. Преимуществом такого разбиения является возможность вычисления углов, ограничивающих кривизну решения, только для одной половины многоугольника (рис. 2), которая находится выше или ниже отрезка $[a,b]$. Оставшаяся половина является зеркальным отображением другой относительно отрезка $[a,b]$.

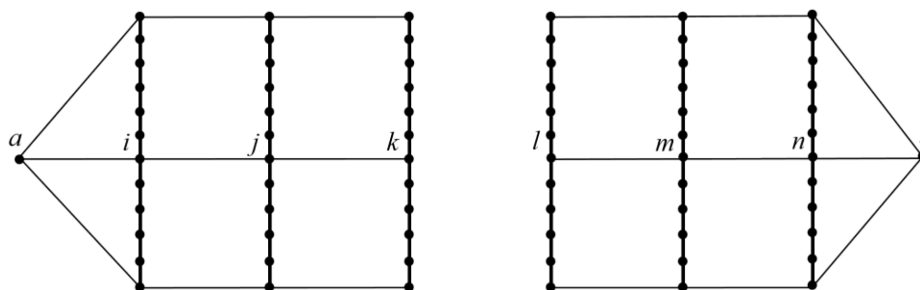

Рис. 2. Пример построения алгоритма с восемью точками разбиения

Результатом полного перебора всех возможных значений для трех направляющих i, j, k будет трехмерный массив данных, что требует значительных объемов используемой памяти. Если же рассмотреть пути вычисления сумм не от точки a к точкам направляющей k, а в обратном направлении, то можно обойтись использованием двумерного массива. Для этого необходимо для всех сумм $S_{j,k}$ найти и сохранить точку на направляющей i, в которой сумма $S_i + S_{i,j} + S_{j,k}$ становится максимальной. В итоге получится два двумерных массива: один с промежуточными суммами $P_{j,k}$, а другой с координатами точек на направляющей i для всех значений $P_{j,k}$. Аналогично для $S^*_{n,m,l}$ получаем $P_{l,m}$ и сохраненные координаты точек для направляющей n.

Следующим этапом идет объединение массивов $S_{k,l}$ с $P_{j,k}$ или $P_{l,m}$. В случае объединения $S_{k,l}$ с $P_{j,k}$ общей направляющей у них является k. Поступая аналогично ранее описанному приему, объединим эти два массива в массив с координатами $P_{k,l}$ и сохранением координат точек для направляющей j. В итоге получается два массива: $P_{k,l}$ и ранее сохраненные координаты направляющих i и j; $P_{l,m}$ и ранее сохраненные координаты направляющей n.

Аналогично, объединяя массивы $P_{k,l}$ и $P_{l,m}$, получаем итоговый двумерный массив сумм Q и координаты всех направляющих для всех значений Q. Остается найти максимальную сумму в массиве Q и взять в качестве результата ранее сохраненные значения координат для всех направляющих.

Приведем пошаговое описание алгоритма поиска максимума целевой функции для восьми точек разбиения на CUDA.

Первая часть.

Шаг 1. Создаются массивы в памяти видеокарты: source1, source2, source3, source5, source6, source7 – для вычисленных ранее сумм модулей синусов углов между градиентом изображения и ломаной линией; result1, result2 – для сохранения промежуточных сумм и координат направляющих.

Шаг 2. Копируются в source1, source2, source3, source5, source6, source7 соответствующие данные из массивов в оперативной памяти.

Шаг 3. Запускается ядро программы на CUDA со следующими параметрами:

grid(N / 2, N / 2)

block(N / 2)

Kernel<<<grid, block, addshmem>>> (…)

где N – количество точек на направляющих, grid – конфигурация блоков на сетке, block – конфигурация нитей в блоке, addshmem – дополнительно выделяемая каждому блоку разделяемая память для промежуточных вычислений сумм и углов.

Шаг 4. На видеокарте задается 2D-индексация блоков и 1D-индексация нитей. Блоками индексируются координаты двумерных массивов $P_{j,k}$ и $P_{l,m}$, а нитями – координаты третьих соответствующих направляющих i и n, для которых производятся вычисления промежуточных максимальных сумм и допустимых углов.

Шаг 5. На видеокарте для всех значений координат массивов $P_{j,k}$ и $P_{l,m}$ вычисляются допустимые углы, не превышающие заранее заданного максимального угла.

Шаг 6. На видеокарте с помощью редукции [6] вычисляются максимальные суммы из ранее вычисленных промежуточных сумм, а также сохраняются значения координат i и n.

Шаг 7. Результирующие данные записываются в массивы result1 и result2.

Шаг 8. Копируются данные из result1 и result2 в массивы, находящиеся в оперативной памяти.

Вторая часть.

Шаг 9. Создаются массивы в памяти видеокарты: source4 – для вычисленных ранее сумм модулей синусов углов между градиентом изображения и ломаной линией; result, result1 – для сохранения промежуточных сумм и координат направляющих.

Шаг 10. Копируются в source4 и result1 соответствующие данные из массивов в оперативной памяти.

Шаг 11. Запускается ядро программы на CUDA со следующими параметрами:

grid(N / 2, N / 2)

block(N / 2)

Kernel<<<grid, block, addshmem>>> (…)

где N – количество точек на направляющих, grid – конфигурация блоков на сетке, block – конфигурация нитей в блоке, addshmem – дополнительно выделяемая каждому блоку разделяемая память для промежуточных вычислений сумм и углов.

Шаг 12. На видеокарте задается 2D-индексация блоков и 1D-индексация нитей. Блоками индексируются координаты двумерных массивов $P_{j,k}$ и $P_{k,l}$, а нитями – координаты общей направляющей k для которой производится вычисление промежуточных максимальных сумм и допустимых углов.

Шаг 13. На видеокарте для всех координат массивов $P_{j,k}$ и $P_{k,l}$ вычисляются допустимые углы, не превышающие заранее заданного максимального угла.

Шаг 14. На видеокарте с помощью редукции вычисляется максимальная сумма из ранее вычисленных промежуточных сумм, а также сохраняются значения координат i и j.

Шаг 15. Результирующие данные записываются в массив result.

Шаг 16. Копируются данные из result в массив, находящийся в оперативной памяти.

Третья часть.

Шаг 17. Создаются массивы в памяти видеокарты: result, result2 – для сохранения промежуточных сумм и координат направляющих.

Шаг 18. Копируются в result и result2 соответствующие данные из массивов в оперативной памяти.

Шаг 19. Запускается ядро программы на CUDA со следующими параметрами:

grid(N / 2, N / 2)

block(N / 2)

Kernel<<<grid, block, addshmem>>> (…)

где N – количество точек на направляющих, grid – конфигурация блоков на сетке, block – конфигурация нитей в блоке, addshmem – дополнительно выделяемая каждому блоку разделяемая память для промежуточных вычислений сумм и углов.

Шаг 20. На видеокарте задается 2D-индексация блоков и 1D-индексация нитей. Блоками индексируются координаты двумерных массивов $P_{k,l}$ и $P_{l,m}$, а нитями – координаты общей направляющей l, для которой производится вычисления промежуточных максимальных сумм и допустимых углов.

Шаг 21. На видеокарте для всех координат массивов $P_{k,l}$ и $P_{l,m}$ вычисляются допустимые углы не превышающие заранее заданного максимального угла.

Шаг 22. На видеокарте с помощью редукции вычисляется максимальная сумма из ранее вычисленных промежуточных сумм, а также сохраняются значения координат i, j, k, n.

Шаг 23. Результирующие данные записываются в массив result.

Шаг 24. Копируются данные из result в массив, находящийся в оперативной памяти.

Шаг 25. Находится максимальная сумма в массиве result и сохраняются все координаты i, j, k, l, m, n.

При увеличении количества точек разбиения отрезка $[a,b]$ в алгоритм будут добавляться части, аналогичные шагам 9–16 (вторая часть) за каждую добавленную точку. В случае уменьшения точек разбиения до 4–7, алгоритм будет упрощаться за счет отбрасывания некоторых его частей.

Некоторые части алгоритма могут выполняться независимо друг от друга. Это дает возможность одновременного выполнения независимых частей алгоритма на разных устройствах. При этом, с увеличением числа точек разбиения, масштабируемость алгоритма ограничивается только количеством доступных вычислительных устройств.

Выполнение алгоритма можно еще ускорить, если отказаться от одного из его преимуществ – универсальности. Обратим внимание, что вычисление углов, ограничивающих кривизну ломаной, делается на основе взаимного геометрического расположения отрезка $[a,b]$ и построенных на нем направляющих. Таким образом, если для решения задачи допустимо разбивать отрезок $[a,b]$ и его направляющие на заранее известные произвольные части (не обязательно равные), то можно предварительно вычислить максимальные углы для всех случаев взаимного расположения отрезков ломаной. При этом можно сразу произвести расчеты для максимально необходимого числа точек на направляющих.

Также были изучены ограничивающие многоугольники, получаемые при различных значениях максимального допустимого угла между соседними отрезками ломаной в процессе поиска оптимального решения. К сожалению, при исследовании геометрического расположения ломаной внутри этих многоугольников не удалось выявить каких-либо закономерностей, позволивших бы ограничить область поиска оптимального решения. Без введения дополнительных условий и ограничений остается слишком много вариантов построения ломаной и ограничивающих многоугольников, которые могут принимать различные формы на определенных шагах работы алгоритма (рис. 3).

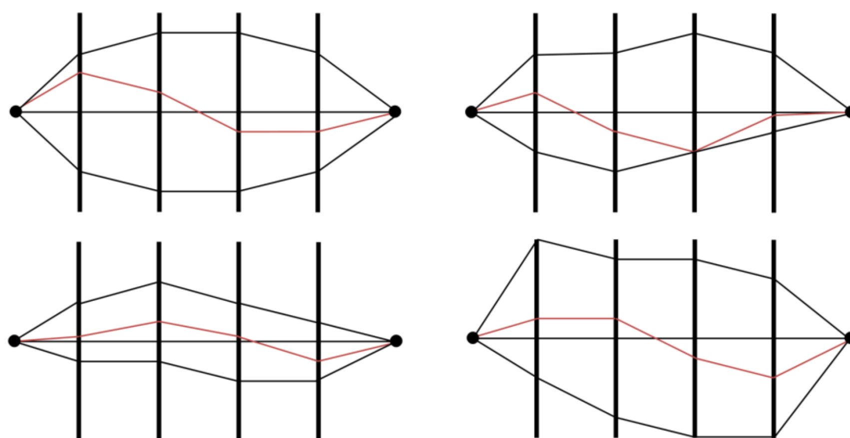

Рис. 3. Примеры построения ломаной и ограничивающих многоугольников

Результаты и их обсуждение

При реализации процессорной версии алгоритма использовалась технология OpenMP [9, 10], а для версии на видеокарте – CUDA. В табл. 1 и 2 приведены результаты времени выполнения алгоритма на процессоре и видеокарте.

Приблизительные значения в табл. 1 были получены путем вычисления времени выполнения одной итерации алгоритма и умножении его на их общее количество. Данные некоторых ячеек таблицы скрыты из-за слишком больших получаемых значений.

Полученные данные тестирования показали применимость данного алгоритма для практической работы при использовании версии для видеокарты. Приведенное время расчетов для видеокарты включает в себя и время копирования данных между процессором и видеокартой, и наоборот. Стоит отметить, что непосредственные вычисления на видеокарте занимают приблизительно лишь третью часть от этого времени. Также к преимуществам алгоритма можно отнести то, что он изначально был спроектирован для хорошей масштабируемости расчетов. Запуск алгоритма на более производительных видеокартах даст возможность использования большего количества точек разбиения и точек на направляющих при сохранении времени вычислений на приемлемом уровне для практической работы.

Таблица 1. **Время выполнения алгоритма без ограничения угла между соседними ломаными линиями (полный перебор) C++ версия для 8 ядер процессора (CPU Intel Core i7-4770K), сек**

Количество точек разбиения	Количество точек на направляющих			
	128	256	512	1024
5	0,3	1,9	13,6	114
6	30	~7,5 мин	~1ч 50 мин	~29 ч
7	~53 мин	~30 ч	~36 дн	–
8	~ 91 ч	~284 дн	–	–

Таблица 2. **Время выполнения алгоритма без ограничения угла между соседними ломаными линиями CUDA версия (GPU GTX 750Ti), мс**

Количество точек разбиения	Количество точек на направляющих			
	128	256	512	1024
5	1,9	7,5	53,1	460,9
6	2,5	11,4	88,5	787,8
7	3,6	14,8	108,1	1004,4
8	5,4	22,0	157,0	1397,8

Заключение

Разработан параллельный алгоритм поиска максимума целевой функции методом динамического программирования с использованием технологии CUDA. Описаны различные подходы построения алгоритма, которые позволяют уменьшить на несколько порядков объем требуемых вычислений, сократить время его выполнения, а также количество одновременно используемой памяти. Приведенные оценки быстродействия показывают применимость данного алгоритма для практической работы. Изначально предложенный алгоритм разрабатывался для ускорения работы алгоритма интерактивного выделения линейных объектов на аэрофотоснимках и космических изображениях, но может быть использован в таком виде или модифицированном и в других задачах, особенно там, где необходим перебор всех вариантов с наложенными ограничениями на получаемое решение. Важной особенностью разработанного алгоритма является возможность одновременной работы его программной реализации на нескольких видеокартах и процессорах. При этом масштабируемость алгоритма ограничивается только количеством доступных вычислительных устройств.

PARALLEL ALGORITHM SEARCHING OF THE OBJECTIVE FUNCTION MAXIMUM BY DYNAMIC PROGRAMMING METHOD USING CUDA TECHNOLOGY

E.N. SEREDIN

Abstract

Parallel algorithm searching the maximum of the objective function using CUDA technology based on the modified method of dynamic programming is presented. Describes the features of parallel software implementations of the algorithm, which allows to reduce by several orders of magnitude the number of required calculations and memory usage. The results of performance software implementations of the algorithm are shown for modern processors and video cards.

Keywords: parallel algorithm, the maximum of the objective function, dynamic programming method, CUDA.

Список литературы

1. *Растригин Л.А.* Статистические методы поиска. М., 1968.
2. *Дорогов В.Г., Теплова Я.О.* Введение в методы и алгоритмы принятия решений. М., 2012.
3. *Гилл Ф. Мюррей У., Райт М.* Практическая оптимизация. М., 1985.
4. *Середин Э.Н., Залесский Б.А.* // Информатика. 2014. № 4. С. 66–73.
5. Zalesky B.A., Seredin E.N. // Proc. of 12th Intern. Conf. PRIP 2014. Minsk, 2014. P. 329–334.
6. *Боресков А.В., Харламов А.А.* Основы работы с технологией CUDA. М., 2010.
7. *Сандерс Дж. Кэндрот Э.* Технология CUDA в примерах: введение в программирование графических процессоров. М., 2011.
8. CUDA Toolkit Documentation. [Электронный ресурс]. − Режим доступа: http://docs.nvidia.com/cuda/index.html. – Дата доступа: 28.01.2016.
9. OpenMP. [Электронный ресурс]. − Режим доступа: http://openmp.org. – Дата доступа: 28.01.2016.
10. Parallelization Using OpenMP. [Электронный ресурс]. − Режим доступа: https://software.intel.com/en-us/articles/parallelization-using-openmp. – Дата доступа: 28.01.2016.