РЕШЕНИЕ ЗАДАЧ С РЕКУРСИВНЫМ ПАРАЛЛЕЛИЗМОМ НА ГРАФИЧЕСКИХ УСКОРИТЕЛЯХ

А.В. Адинец, П.Ю. Мезенцев

ВВЕДЕНИЕ

В последние годы тенденции развития полупроводниковой индустрии стали меняться. Если 10 лет назад основным способом улучшения производительности компьютеров было увеличение тактовой частоты процессора, то в настоящий момент акцент сместился на параллельность. В центральных процессорах количество вычислительных ядер составляет от двух до четырёх, в будущем их число будет возрастать. В графических ускорителях процесс начался раньше. Для отрисовки графики изначально было удобно выполнять большое количество операций одновременно, т.е. параллельно. Поэтому на данный момент количество вычислительных ядер в графических платах может быть больше 200.

Однако ядрам ГПУ (графическое процессорное устройство) присущи некоторые особенности, что делает процесс разработки программ для графических ускорителей более сложным и трудоёмким. До недавнего времени производить вычисления общего назначения на графических устройствах было возможно только с использованием библиотек для программирования трёхмерной графики. Ситуация изменилась с появлением технологий CUDA от NVIDIA [1] и CTM (close to metal) от ATI [2], которые позволяют работать с ГПУ напрямую.

При написании программ для графического процессора на передний план выходит множество особенностей, которые либо не столь важны, либо вообще не возникают при написании программ для центрального процессора. В качестве примера можно назвать одновременное исполнение всеми ядрами ГПУ одинаковых инструкций над разными данными и критичность времени доступа в память. Из-за этого существуют как задачи, для которых можно получить большой прирост производительности на ГПУ, так и такие, для которых это почти невозможно. В данной работе рассмотрен класс задач с рекурсивным параллелизмом, к описанию которых мы переходим.

ОПИСАНИЕ КЛАССА ЗАДАЧ С РЕКУРСИВНЫМ ПАРАЛЛЕЛИЗМОМ

К классу *задач с рекурсивным параллелизмом* относятся задачи, для решения которых используется функция, рекурсивно вызывающая сама себя В результате каждого вызова задача разбивается на несколько более простых подзадач, так продолжается до тех пор, пока решаемые задачи не оказываются элементарными. Можно изобразить процесс решения задачи следующим образом (рис. 1):

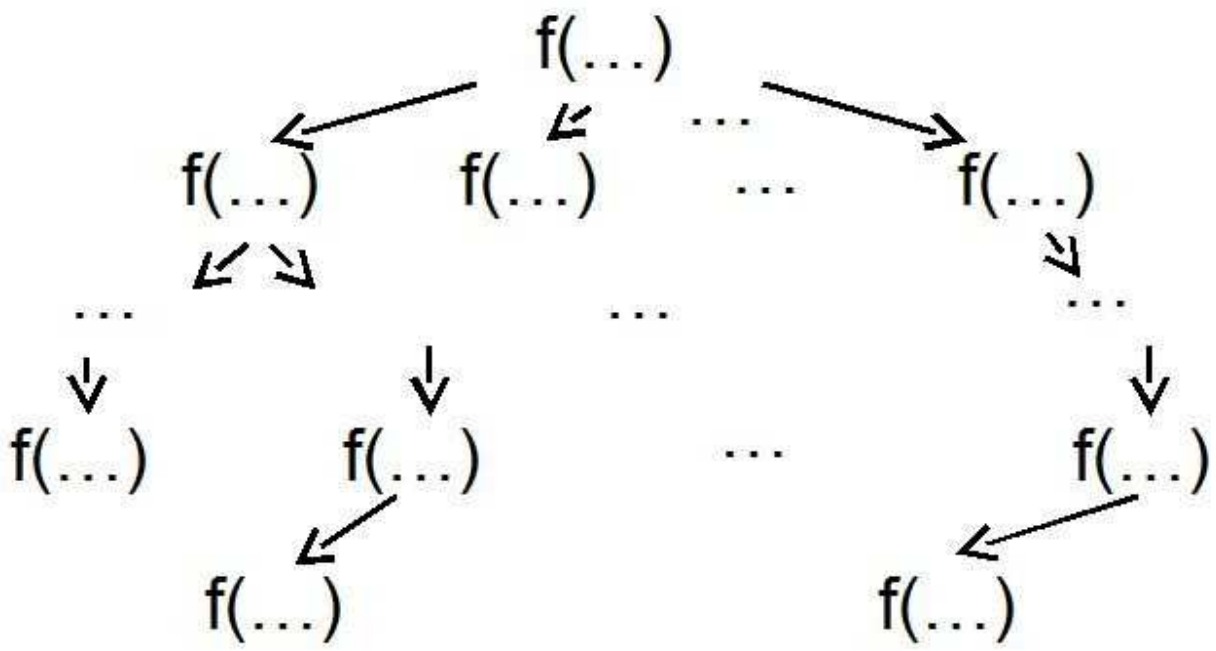

Рис 1

Характерной особенностью рекурсивных вызовов в данном классе задач является их независимость друг от друга, что означает, что эти вызовы можно обрабатывать параллельно. В конечном итоге это создает тот ресурс параллелизма, который позволяет эффективно задействовать ГПУ для решения той или иной задачи. Однако можно заметить, что ресурс параллелизма появляется не сразу, а только по мере того, как глубина рекурсии увеличивается. Начиная с какого-то уровня, зависящего от рассматриваемой задачи, количество вызовов на одном уровне становится достаточным, чтобы полностью загрузить графический процессор.

В качестве примера задач с рекурсивным параллелизмом можно назвать следующие:

- Сортировка:
  - Быстрая сортировка
  - Сортировка слиянием
  - Пространственная сортировка (kd-дерево)
- Игровые задачи:
  - Шахматы, шашки и другие логические игры
  - Решение головоломки “Судоку”
  - Задача N ферзей
- Задачи поиска:
  - Поиск в неупорядоченном дереве
  - Поиск файла с паттерном имени в дереве файловой системы
- Прочие:
  - Трассировка лучей
  - Проверка разрешимости КНФ

К схеме с рекурсивным параллелизмом приводят различные подходы к решению задач. Одним из них является подход «Разделяй и властвуй» [3]. Исходная задача разбивается на несколько подзадач, каждая из которых вычислительно менее сложная чем исходная. Процесс продолжается до тех пор, пока каждая задача не станет элементарной. При этом отдельные подзадачи можно решать параллельно.

Другим распространённым подходом для решения задач является алгоритм поиска с возвратом [4]. В этом случае также возможно использование схемы с рекурсивным параллелизмом. При параллельном решении задач данный алгоритм используется для того, чтобы одновременно проверять несколько шагов, возможных из каждого положения.

Конечной целью работы является создание инструмента для решения задач с рекурсивным параллелизмом. Такой инструмент позволит конечному пользователю использовать ресурсы графической карты для увеличения производительности его приложения без необходимости самостоятельно реализовывать задачу на ГПУ.

АРХИТЕКТУРА CUDA

В данной работе для решения задач с рекурсивным параллелизмом на ГПУ используется технология CUDA. CUDA представляет собой средство для программирования ГПУ от компании NVIDIA. Для изучения CUDA не требуется много усилий, т.к. технология представляет собой расширение языка Си. Однако прежде чем рассматривать это расширение взглянем на архитектуру CUDA:

Устройство, или ГПУ, состоит из нескольких *мультипроцессоров*, каждый из которых имеет доступ к глобальной памяти видеокарты. Внутри каждого мультипроцессора есть несколько *потоковых процессоров* (ПП). Эти процессоры имеют доступ к небольшому количеству памяти, называемой *разделяемой памятью*. Разделяемая память по скорости доступа в несколько раз опережает глобальную память. Каждый мультипроцессор имеет доступ только к своей разделяемой памяти не имеют доступа к разделяемой памяти других мультипроцессоров.

Для программиста модель программирования выглядит следующим образом: сперва программа выполняется на центральном процессоре, выполняются подготовительные операции, необходимые данные записываются в глобальную память видеокарты, после этого вызывается функция, называемая *ядром*, которая

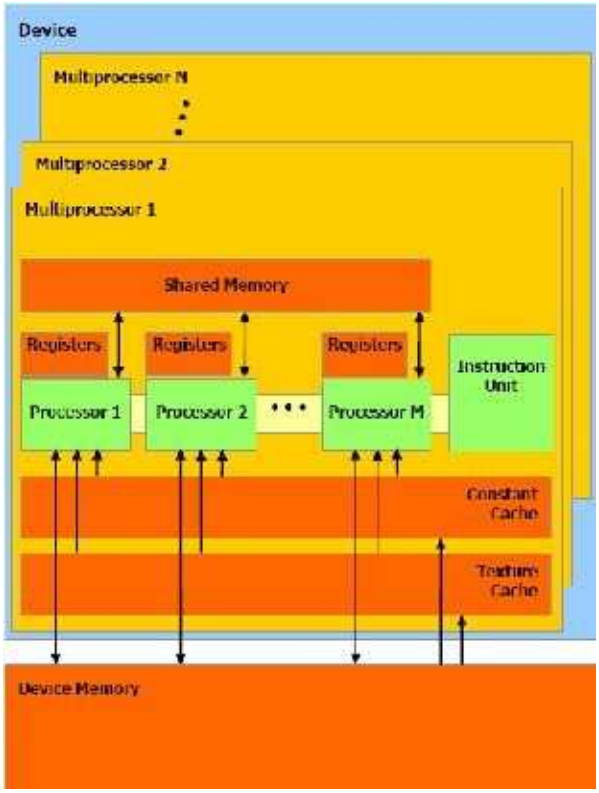

Рис. 2

выполняется на ГПУ. При этом создаётся множество *потоков ГПУ*, каждый из которых исполняет код ядра. Потоки объединяются в *блоки*, каждый из которых может иметь одно, два или три измерения. А блоки, в свою очередь, объединяются в *сетку блоков*. В каждом потоке можно получить его координаты в сетке и блоке, что позволяет каждому потоку проводить операции над разными данными. При этом блок потоков дополнительно разбивается на небольшие группы — *варпы*. Потоки внутри одного варпа физически исполняются синхронно, в режиме ОКМД (одна команда — много данных).

Таким образом для написания CUDA программы необходимо написать код для ЦПУ, выполняющий инициализацию, копирование данных, запуск ядра на ГПУ и освобождение ресурсов, и ядро, которое будет производить вычисления на ГПУ.

РЕАЛИЗАЦИЯ НА ГПУ

Ввиду многих особенностей графических ускорителей полностью перенести решение задачи на ГПУ может быть сложно и, скорее всего, неэффективно. В качестве одной из причин можно назвать отсутствие ресурса параллелизма на первых этапах решения задачи. В связи с этим вычисления разбиваются на два этапа, первый выполняется на центральном процессоре, а второй - на ГПУ (рис. 3).

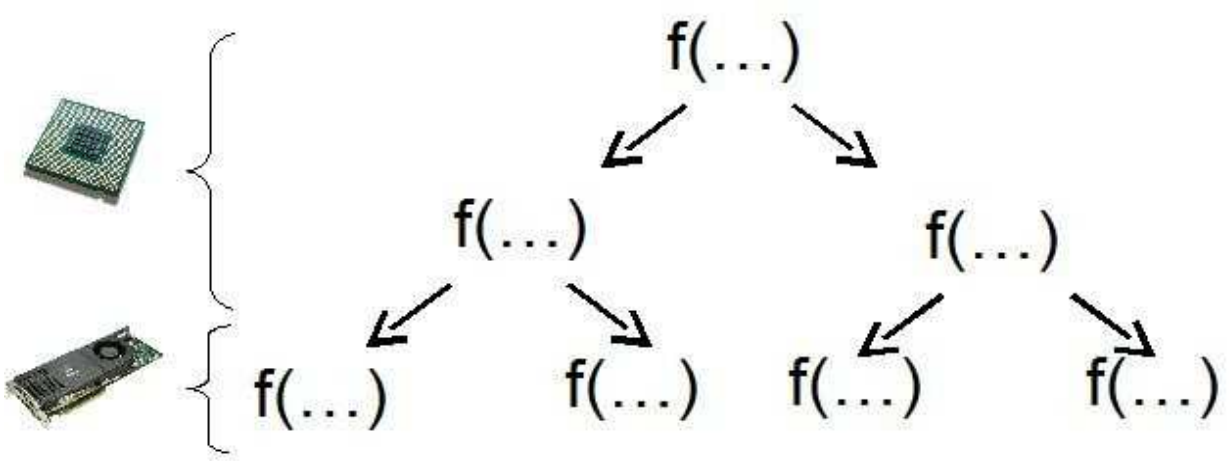

Рис. 3

Во время первого этапа, исполнения на ЦПУ, решаются две задачи:

- Генерация контекстов для дальнейшего исполнения на графическом процессоре.
- Копирование данных в память графического процессора.

Под *контекстом* в данном случае понимается совокупность данных, необходимых ГПУ для продолжения вычислений с того места, где закончил ЦПУ. После создания достаточного для загрузки ГПУ количества контекстов их вместе с остальными данными необходимо скопировать в память графического процессора. Также после выполнения вычисления на графическом ускорителе требуется скопировать результаты обратно в память центрального процессора.

Во время второго этапа на ГПУ порождается количество потоков, равное количеству сгенерированных на первом этапе контекстов. Каждый поток решает свою подзадачу.

В графическом процессоре отсутствует аппаратная поддержка стека, что не позволяет осуществлять рекурсивные вызовы в традиционном понимании. Для решения этой проблемы используется программный стек [5]. В текущей реализации каждый поток имеет свой, независимый от остальных, стек.

Для осуществления рекурсивных вызовов необходимо в начале передавать функции входные параметры, а при более глубоких вызовах сохранять текущее состояние, т.е. значения локальных переменных и точку возврата. После окончания решения своей подзадачи функция возвращает своему «родителю» значение, являющееся результатом ее работы. Во время вызова новой функции, данные предыдущей помещаются в стек, а во время возврата берутся из стека. Данные можно хранить в такой структуре (листинг 1).

```
typedef struct
{
  params parameters;
  loc_vars locals;
  ret_val_type ret_val;
  int state;
} stack_type;
```

листинг 1

Поле *state* используется для запоминания точки возврата и сигнализации завершения выполнения. После вызова или возврата функция определяет, с какого момента выполнять вычисления по значению *state*.

Проиллюстрируем использование этой структуры на примере модельной задачи вычисления чисел Фибоначчи. Обычная рекурсивная функция вычисления числа Фибоначчи выглядит следующим образом (листинг 2).

```
int Fibon_CPU(int n)
{
 if (n>2)
 { int n1 = Fibon_CPU(n-1);
  int n2 = Fibon_CPU(n-2);
  int res = n1 + n2;
  return res;
 }
 else return 1;
}
```

листинг 2

Единственным входным параметром данной функции является число N, номер числа Фибоначчи, которое требуется вычислить. В качестве локальных переменных выступают *n1*, *n2* и *res*. Возвращаемым значением является либо значение *res*, либо 1. Состояний у функции может быть 4:

- 0 если функция только что вызвана
- 1 после вычисления *n1*
- 2 после вычисления *n2*
- 1 после завершения работы

Для создания более универсальной схемы решения задач код, работающий со стеком, отделён от кода, выполняющего содержательную часть решения задачи. Часть, отвечающая за стек, осуществляет запись состояния функций в стек и обратно. Пользователю об этих операциях знать не нужно, от него требуется предоставить структуру, описанную в листинге 1, и вычислительное ядро, которое будет принимать на вход такую структуру и модифицировать её. Также ядро должно определять в зависимости от значения поля *state*, что требуется вычислить в тот или иной момент. Код для работы со стеком приведён в листинге 3.

```
void ker_main(params* init_params, ...) {
 ...
 stack_type stack[N];
 int SP=0; //stack pointer
 int SS=1; //size of stack element
 while (SP>=0)    { //main loop
          if (stack[SP].state >= 0) {
                   stack[SP] = cur_state;
                   SP+=SS;
                  stack[SP] = new_state;
func (&new_state);
          }
          else {
                   SP-=SS;
                   if (SP<0) continue;
                   old_state = stack[SP];
                    func(&old_state);
          }
 }
 res[glob_id]=stack[0].ret_val;
 ...
}
```

листинг 3

Вызов «рекурсивной» функции происходит в цикле, который продолжается до тех пор, пока стек не станет пустым. Перед каждым вызовом формируется переменная типа *stack_type*, которая затем передается в вызываемую функцию. Общая структура кода, выполняющего содержательную часть решения задачи, приведена в листинге 4.

```
void func(params* parametres, loc_vars* locals, ret_val_type* ret_val, int* state)
{
 ... //local variables which don't need to be kept in stack
 switch(state)
 {
 case 0: //first call
          ...
           break;
 case 1:
          ...
          break;
...
 case N:
          ...
          break;
           }
}
```

листинг 4

В каждой ветви блока **switch** содержится часть вычислений, выполняемых в определенном состоянии.

Преимуществом данной схемы является так же тот факт, что можно использовать другую версию стека или иного механизма распределения заданий. И это не отразится на вызываемой функции.

РЕЗУЛЬТАТЫ

На текущий момент было рассмотрено 3 задачи: обход дерева, решение головоломки Судоку и задача N ферзей. В процессе работы над каждой из них проявились те или иные свойства и особенности задач с рекурсивным параллелизмом и их реализации на ГПУ.

В задаче обхода дерева генерируется дерево заданной ширины и глубины, которое затем копируется на ГПУ, и выполняется его обход. В ходе обхода могут решаться разные задачи, например суммирование всех элементов дерева, поиск заданного элемента и т.д. Решение различных задач позволило выявить зависимость эффективности от количества вычислений в каждом узле. При решении простых задач эффективность невысокая в связи с большими затратами на ожидание данных из памяти. Однако с ростом объёма вычислений в каждом узле дерева, производительность ГПУ вскоре начинает превосходить производительность ЦПУ.

При решении головоломки Судоку [6] дается поле, в классическом варианте размер составляет 9 на 9 клеток. Часть клеток заполняется заранее числами от 1 до 9, необходимо заполнить пустые клетки так, чтобы по горизонтали, вертикали и блоке 3 на 3 каждое число встречалось ровно один раз. Можно обобщить задачу на случай размера поля 16 на 16, 25 на 25 и т.д., т.е. в общем случае имеем поле размера n^2 на n^2. По определению, существует ровно одно решение задачи. Были проведены тесты для размеров 9 на 9 и 16 на 16, именно эти размеры встречаются в решаемых на практике головоломках. Решение ищется методом поиска с возвратом. Ускорения для данной задачи получить не удалось. Для размера 9 на 9 ЦПУ требуется в худшем случае порядка 0,3 мс для нахождения ответа, для ГПУ существенными становятся накладные расходы. Для размера 16 на 16 на результат сильно влияет неравномерность загрузки разных потоков, т.е. разная вычислительная нагрузка в разных ветвях решения задачи.

В задаче N ферзей [7] выполнялся подсчёт количества расстановок N ферзей на доске K*K. Сразу можно заметить, что при количестве ферзей N несколько меньшим размера доски K количество вариантов максимально при фиксированном K. Для решения задачи используется алгоритм поиска с возвратом. На центральном процессоре происходит расстановка нескольких ферзей, после чего оставшиеся ферзи расставляются на графическом процессоре.

В результате тестов с различными стартовыми конфигурациями максимальное ускорение составило 20 раз. Замеры производились с использованием Tesla C1060 и Intel Core 2 Quad Q6600. Результаты приведены в таблице 1 и на рисунке 4.

Таблица 1. (Рис. 4)

	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU
доска\доска-ферзи	0	0	1	1	2	2	3	3	5	5	6
9	0,015	0,018	0,115	0,037	0,328	0,071	0,406	0,090			

10	0,084	0,058	0,775	0,109	2,674	0,222	4,056	0,276				
11	0,507	0,046	5,404	0,254	22,123	1,044	41,092	1,833				
12									261,833	11,830	79,107	

По оси X находится размер доски. При каждом размере доски производились расчёты как для количества ферзей, полностью заполняющих доску, так и для несколько меньшего их числа. Наилучший результат получается как раз при неполном заполнении доски, т.к. количество вариантов расстановки, а, следовательно, и ресурс параллелизма, тут особенно велики.

ЗАКЛЮЧЕНИЕ

В ходе исследования было выявлено несколько проблем, которые появляются в связи с решением задач с рекурсивным параллелизмом.

Одной из важных проблем является работа со ссылочными структурами данных на ГПУ. При работе с такими структурами произвольный доступ к данным отсутствует, имеется лишь возможность переходить от одного узла структуры к соседним по ссылкам. Копирование подобных структур необходимо осуществлять за приемлемое время, чтобы не потерять выгоду от использования ГПУ. Так же проблемой является синхронизация данных, хранящихся в памяти ЦПУ и ГПУ.

Ещё одной задачей является определение оптимальной глубины отгрузки на ГПУ. Если породить слишком мало контекстов, то выигрыш от распараллеливания будет незначительным. Если же породить их слишком много, то это потребует излишнего времени обработки на центральном процессоре.

Ещё одной проблемой является неравномерное распределение нагрузки между разными потоками. Нередко возникает ситуация, когда разные ветви задачи для решения требуют разного количества вычислений. Например, такая ситуация возникает при решении головоломки Судоку. В лучшем случае в соседних потоках окажутся ветви, требующие одинакового числа вычислений, в этом случае это не скажется на производительности. Однако, как правило, предугадать нагрузку в каждой ветви затруднительно, из-за чего ветви, имеющие разную вычислительную нагрузку, исполняются одновременно. По этой причине часть времени некоторое количество потоков в варпе работают вхолостую, что отрицательно влияет на производительность.

В настоящее время за каждым потоком фиксируется одна исходная задача, а все возникающие подзадачи обрабатываются тем же потоком при помощи стека. Однако, возможны другие механизмы разделения задач.

Одним из вариантов является реализация стека на блок или на варп [5] (группа потоков внутри одного блока) потоков. В результате потоки смогут обмениваться задачами и более равномерно распределять нагрузку. Другим вариантом является общая очередь, в которую будут добавляться задания от каждого потока и куда потом будут записываться результаты.

На текущий момент рассмотрено сравнительно небольшое количество задач с рекурсивным параллелизмом и способов их решений. Однако уже сейчас видно, что при использовании графических процессоров можно получить прирост производительности при решении подобных задач. В дальнейшем планируется расширить как набор решаемых задач из данной класса, так и круг методов при помощи которых они решаются.

ЛИТЕРАТУРА:

1. http://developer.download.nvidia.com/compute/cuda/2_21/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.1.pdf
2. http://developer.amd.com/gpu_assets/Stream_Computing_User_Guide.pdf
3. А. Ахо, Д. Хопкрофт, Д. Ульман, “Структуры данных и алгоритмы”, издательский дом “Вильямс”, 2003, с.276-280
4. А. Ахо, Д. Хопкрофт, Д. Ульман, “Структуры данных и алгоритмы”, издательский дом “Вильямс”, 2003, с.291-302
5. K. Yang, B. He, Q. Luo, P. V. Sander, J. Shi, Stack-Based Parallel Recursion on Graphics Processors, SIGPLAN symposium on Principles and practice of parallel programming.
6. http://en.wikipedia.org/wiki/Sudoku
7. http://en.wikipedia.org/wiki/Eight_queens_puzzle