Использование GPU для ускорения поиска в ширину на графах*

М.А. Черноскутов

Институт математики и механики УрО РАН

В работе описана реализация алгоритма поиска в ширину на графе с использованием технологии CUDA на GPU, проведено сравнение ее быстродействия с реализацией того же алгоритма на CPU на основе технологии OpenMP. Сравнение производилось с использованием графов, имеющих различное число вершин и ребер. В результате экспериментов выявлено, что GPU опережает CPU по производительности на всем диапазоне тестов.

1. Введение

В последнее время растет популярность использования GPU в качестве вычислителей общего назначения, с помощью которых потенциально можно запрограммировать решения самых разнообразных задач. Сейчас GPU используются в основном для приложений, которые принято называть вычислительно-интенсивными («computing intensive tasks»): механика сплошных сред, молекулярная динамика, трассировка лучей и т.д. Однако растет потребность и в задачах совершенно другой природы – «data intensive tasks», которые характеризуются работой с большими массивами данных и активным использованием нерегулярного доступа в память [1]. На данный момент уже появился специализированный рейтинг Graph500 (аналог рейтинга Top500) [2], целью которого является оценка производительности вычислительных систем при обработке больших массивов данных.

Одним из представителей задач класса «data intensive» является поиск в ширину на графе [3], который в данный момент используется в качестве вычислительного ядра теста Graph500 (пока реализация доступна только для CPU) [4]. Данный выбор обусловлен следующими обстоятельствами [2]:

- алгоритм обхода графа в ширину используется в различных научных и технических приложениях (анализ социальных сетей, информационная безопасность, биоинформатика, нефтедобыча);
- структура графов отображает реальные наборы данных, которые встречаются в задачах;
- результаты масштабирования задачи обхода графов должны хорошо соответствовать масштабированию реальных приложений.

В данный момент известно множество работ по обработке больших графов на CPU и GPU и сравнению их производительности: [5-9].

2. Постановка задачи

Целью работы является анализ эффективности использования GPU в задаче поиска в ширину на графе. Основным критерием эффективности является отношение времени выполнения тестовой задачи (поиск в ширину на заранее выбранном графе) на GPU по сравнению с временем выполнения этой же задачи на CPU. Распараллеливание алгоритма обхода графа осуществляется по всем доступным ядрам GPU и CPU. В качестве входных данных для алгоритма используются графы одинакового размера (имеется в виду не только объем занимаемой памяти, но и количество ребер и вершин). Результатом работы программы является время и скорость обхода заданного графа (измеряется в количестве пройденных ребер в секунду [2]). Ранжирование результатов осуществляется по скорости обхода графа.

* Работа поддержана грантами УрО РАН РЦП-12-П13, РЦП-13-П18. При проведении работ был использован суперкомпьютер “Уран” ИММ УрО РАН.

3. Реализация

В каждом эксперименте использовался ориентированный граф, с заранее заданным числом исходящих из каждой вершины ребер. При этом противоположные концы ребер выбирались случайным образом (возможность образования петель и несвязных компонент при этом не исключается). Графы с подобной структурой часто встречаются в реальных задачах, таких как анализ социальных сетей или биоинформатика. Размер графов выбирался таким образом, чтобы заполнить всю доступную DRAM память на GPU. Используемые в экспериментах графы приведены в табл.1.

Таблица 1. Графы, используемые в эксперименте

Количество исходящих из вершины ребер, шт.	Количество вершин, шт.	Размер графа, ГБ
5	66 060 288	2.21
10	48 234 496	2.52
15	35 651 584	2.52
20	28 311 552	2.53
25	23 068 672	2.49
30	19 922 944	2.52
35	16 777 216	2.44
40	14 680 064	2.41
45	13 631 488	2.49
50	12 582 912	2.53
55	11 534 336	2.54
60	10 485 760	2.50
65	9 437 184	2.43
70	8 388 608	2.31

Для поиска в ширину с использованием GPU реализовано два CUDA-ядра. Первое производит перебор вершин в текущей итерации (в порядке обхода в ширину) и заносит данные об их соседях в специальный массив, а второе, анализируя этот массив, формирует список вершин для следующей итерации алгоритма. Каждому GPU-потоку ставится в соответствие одна вершина, которую он просматривает на каждой итерации (реализация очередей, присущих стандартному алгоритму поиска в ширину, для GPU не эффективна в виду больших накладных расходов на обработку очереди). В текущей реализации в каждом блоке задействовано 1024 потока. Количество блоков варьируется в зависимости от количества исходящих ребер для каждой вершины. Исходный код обоих ядер, а также описание используемой для графа структуры данных представлено на рис.1.

Поиск в ширину на графе с помощью CPU производится аналогичным образом – используется две функции, распараллеленные на все доступные ядра CPU: первая обходит вершины на текущей итерации, а вторая формирует список вершин для следующей итерации. Для описания вершин и ребер графа использовалась та же самая структура данных, что и в версии для GPU. Исходный код обеих функций приведен на рис.2. Как видно, объем кода для обеих версий алгоритма оказался примерно одинаковым.

```
struct node {
  int x;
  int edge[DEGREE];
  int next;
  int visited;
};

__global__ void traversal(struct node *d_arr,int
*d_next_lev,int *d_count)
{
  d_count[0]++;
  int k,i = blockDim.x * blockIdx.x + threadIdx.x;
  if(d_arr[i].next==1 && d_arr[i].visited==0)
  {
    d_count[1]++;
    d_arr[i].next = 0;
    d_arr[i].visited = 1;
    for(k=0;k<DEGREE;++k)
      d_next_lev[d_arr[i].edge[k]] = 1;
  }
}

__global__ void refresh(struct node *d_arr,int *d_next_lev)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if(d_next_lev[i]==1 && d_arr[i].visited==0)
    d_arr[i].next = 1;
  d_next_lev[i] = 0;
}
```

Рис. 1. Обход графа в ширину с использованием GPU

```
int traversal()
{
  int i,k,s = 0;
  int start =
omp_get_thread_num()*(SIZE/omp_get_num_threads());
  int end = start + (SIZE/omp_get_num_threads());
  for(i=start;i<end;++i)
  {
    if(arr[i].next==1 && arr[i].visited==0)
    {
      s++;
      arr[i].next = 0;
      arr[i].visited = 1;
      for(k=0;k<DEGREE;++k)
         next_lev[arr[i].edge[k]] = 1;
    }
  }
  return s;
}

void sync()
{
  int i;
  int start =
omp_get_thread_num()*(SIZE/omp_get_num_threads());
  int end = start + (SIZE/omp_get_num_threads());
  for(i=start;i<end;++i)
  {
    if(next_lev[i]==1 && arr[i].visited==0)
      arr[i].next = 1;
    next_lev[i] = 0;
  }
}
```

Рис. 2. Обход графа в ширину с использованием CPU

Для тестирования использовался GPU Nvidia Tesla M2050, имеющий 448 вычислительных ядер и 3 ГБ DRAM памяти (в действительности, доступным для использования оказалось около 2.5 ГБ из-за включенного ECC и других накладных расходов) и CPU Intel Xeon X5675, имеющий 6 вычислительных ядер и 48 ГБ оперативной памяти. Оба вычислителя располагаются в сервере HP ProLiant SL390s G7 4U.

Программа для GPU скомпилирована с использованием nvcc, входящего в комплект CUDA Toolkit 4.0. Для CPU использовался компилятор icc версии 11.1 и технология OpenMP. Для распараллеливания обхода графа в ширину на все процессоры в CPU и потоки в GPU использовались технологии OpenMP и CUDA. Для генерирования ребер в графах использовался генератор случайных числе rand(), входящий в стандартную библиотеку языка C.

4. Эксперимент

Программам на GPU и CPU подавались графы с одинаковым числом вершин и одинаковым количеством ребер, исходящих из каждой вершины. Эксперименты проводились с графами, имеющими от 5 до 70 исходящих из каждой вершины ребер. Количество вершин в графе варьировалось от 8 388 608 до 66 060 288, в зависимости от объема доступной памяти и числа исходящих из каждой вершины ребер.

Как видно из рис.3, GPU заметно опережает CPU во всех экспериментах. Наибольшее преимущество GPU имеет при обходе графов с малым числом исходящих ребер. При увеличении числа исходящих ребер, преимущество GPU постепенно уменьшается, вследствие роста количества запросов в память, производимых множеством параллельных нитей. В CPU, наоборот, параллельных потоков намного меньше и вычислительные ядра имеют большую скорость работы по сравнению с нитями GPU.

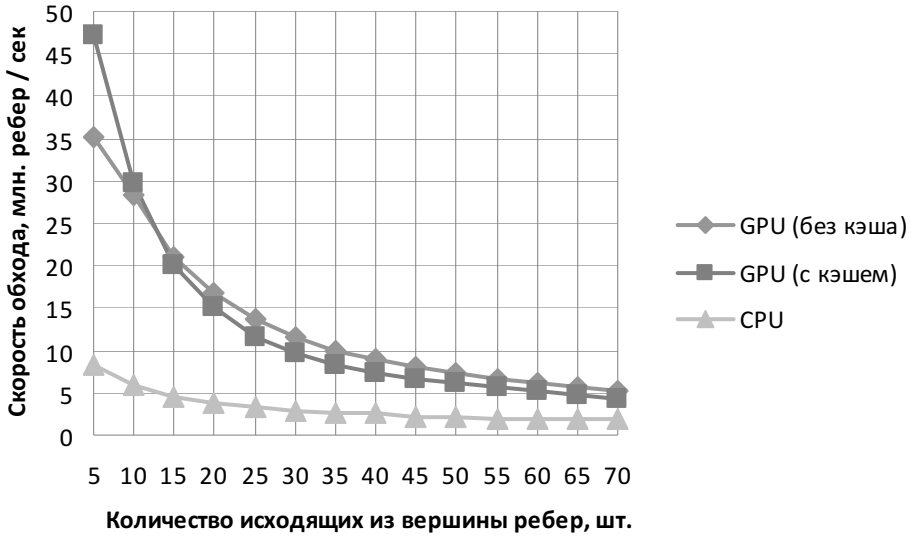

Рис. 3. Скорость обхода графов на GPU и CPU

Достигнутое на GPU ускорение показано на рис.4. Интересной особенностью является влияние кэша L1 в GPU на скорость обхода графа. При малом числе исходящих ребер кэш GPU оказывает положительное влияние на производительность, т.к. может захватить больше данных о вершинах графа, а с ростом их числа, наоборот, показывает ухудшающиеся результаты. К тому же с ростом количества исходящих ребер снижается пространственная и временная локальность [10] и степень промахов кэша из-за этого только увеличивается.

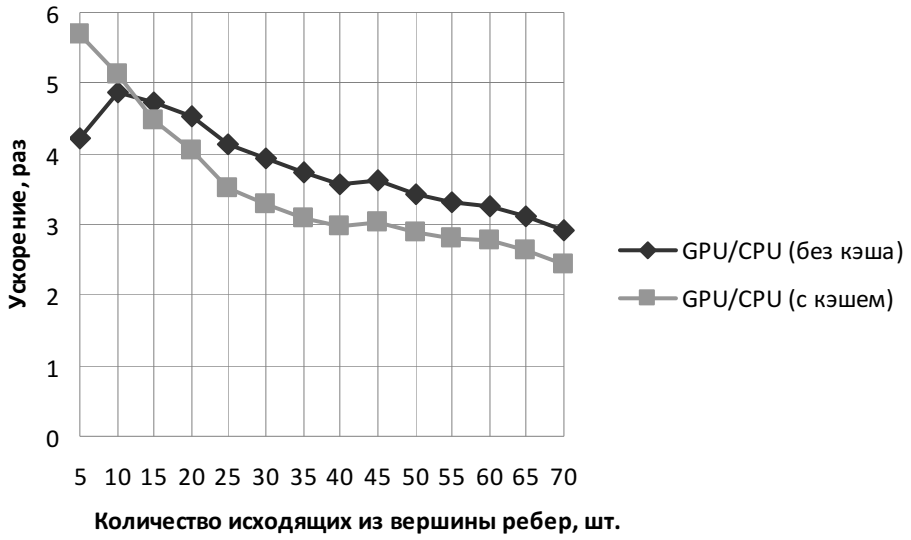

Рис. 4. Ускорение, достигнутое на GPU

5. Заключение

В результате эксперимента выявлено, что поиск на графах в ширину при использовании GPU производится с большей скоростью (от 2.4 до 5.7 раз быстрее), по сравнению с аналогичной задачей на CPU, при условии, что размер графа не превышает объем памяти в GPU. Установлено, что кэш L1 в GPU в большинстве случаев не дает выигрыша в производительности.

В качестве дальнейших направлений исследований интерес представляет:

- обход графов, созданных с помощью генераторов (таких как Kronecker Graph Generator [11]), а также графов полученных в результате решения реальных задач (к примеру графы из Sparse Matrix Collection [12] или DIMACS Implementation Challenge [13]);
- анализ эффективности обхода больших графов в multi-GPU системах, а также в кластерных системах, использующих GPU;
- решение реальных прикладных и фундаментальных задач класса «data intensive tasks» с использованием GPU.

Литература

1. Furht B., Escalante A. «Handbook of Data Intensive Computing», Springer, 2011.
2. Murphy R., Wheeler K., Barrett B., Ang J. «Introducing the Graph 500» // Cray User's Group (CUG), May 5, 2010.
3. Кормен Т., Лейзерсон Ч., Риверс Р., Штайн К. «Алгоритмы. Построение и анализ» – М.: «Вильямс», 2011.
4. Graph 500 Benchmark 1 (“Search”) // URL:http://www.graph500.org/specifications (дата посещения: 01.06.2011).
5. Harish, P. and Narayanan, P.J. «Accelerating large graph algorithms on the GPU using CUDA» // Proceedings of the 14th international conference on High performance computing (Berlin, Heidelberg, 2007), 197–208.
6. Hong, S. et al. «Accelerating CUDA graph algorithms at maximum warp» // Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (New York, NY, USA, 2011), 267–276.

7. Merrill D., Garland M., Grimshaw A. «Scalable GPU graph traversal» // Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP'12), 117–128.
8. Hong S., Oguntebi T., Olukotun K. «Efficient Parallel Graph Exploration on Multi-Core CPU and GPU» // Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (Galveston, TX, USA, October 10-14, 2011), 78–88.
9. Корж А.А. «Масштабирование Data Intensive приложений с помощью библиотеки Dislib на суперкомпьютерах BlueGene/P и Ломоносов» // Материалы Всероссийской научной конференции «Научный сервис в сети ИНТЕРНЕТ», Новороссийск, 19-24 сентября 2011 г., Изд-во Московского Университета, с.126–131.
10. Murphy R., Kogge P. «On the Memory Access Patterns of Supercomputer Applications: Benchmark Selection and Its Implications» // IEEE Transactions on Computers 56(7), July 2007, 937–945.
11. Seshadhri C., Pinar A., Kolda T. «An In-Depth Study of Stochastic Kronecker Graphs» // Proceedings of the 2011 IEEE 11th International Conference on Data Mining (ICDM), 587–596.
12. Davis T., Hu Y. «The University of Florida Sparse Matrix Collection» // ACM Transactions on Mathematical Software, Vol. V, No. N, M 20YY, 1–28.
13. 10th DIMACS Implementation Challenge // URL:http://www.cc.gatech.edu/dimacs10/index.shtml (дата посещения: 01.06.2011).