Оптимизация метода решения линейных систем уравнений в OpenFOAM для платформы MPI + CUDA[1]

А.В. Монаков
<amonakov@ispras.ru>
В.А. Платонов
<soh@ispras.ru>
ИСП РАН, 109004, Россия, г. Москва, ул. А. Солженицына, 25.

Аннотация. Описываются работы по оптимизации решателя систем линейных уравнений пакета OpenFOAM для гетерогенной вычислительной системы с GPU-ускорителями в модели MPI + CUDA. В ходе работ предложена параллельная GPU-версия метода предобуславливания AINV и проведен ряд специфичных GPU-оптимизаций, в основном направленных на минимизацию обменов данными с GPU. Представлены экспериментальные результаты тестирования решателя на некоторых задачах OpenFOAM.

Ключевые слова: метод сопряженных градиентов, AINV предобуславливание, OpenFOAM, GPU, MPI

1. Введение

Одним из перспективных методов повышения производительности высокопроизводительных систем является использование гетерогенных систем, в которых многоядерные процессоры дополняются акселераторами, которые за счет специализированной архитектуры позволяют получить высокую производительность для определенных классов задач. В настоящее время с такой целью широко используются графические акселераторы (GPU), архитектура которых оптимизирована для задач с высоким параллелизмом.

В работе рассматривается задача ускорения расчетов в пакете OpenFOAM на кластерах с узлами, оборудованными GPU. OpenFOAM [1] является крупным открытым пакетом, ориентированным в основном на задачи вычислительной гидродинамики. Распараллеливание расчетов в OpenFOAM выполнено на

1 Работа поддержана грантом РФФИ 13-07-12102

основе разделения расчетной области между MPI-процессами. Схема распараллеливания является одноуровневой: каждый MPI-процесс является однопоточным, хотя более предпочтительной может быть двухуровневая схема, в которой MPI-процессы используют механизмы распараллеливания на общей памяти, такие как OpenMP.

Большинство расчетов в OpenFOAM сводится к многократному решению линейных систем. Матрицы этих систем являются сильно разреженными, и расположение ненулевых элементов полностью определяется графом смежности ячеек расчетной сетки (за исключением блочных систем размерность матрицы равна количеству ячеек сетки, а количество ненулевых элементов выше главной диагонали равно количеству внутренних граней). Для решения систем применяются итерационные методы. Обычно основное время счета приходится на решение симметричных линейных систем, причем для решения используется метод сопряженных градиентов или многосеточный метод. Таким образом, перенос итерационных методов решения на акселератор может позволить существенно сократить общее время счета. В этой статье рассматривается перенос метода сопряженных градиентов с простым предобуславливателем на GPU-акселераторы.

2. Метод сопряженных градиентов на GPU

В OpenFOAM используется классический вариант метода сопряженных градиентов с предобуславливанием [2] для системы $\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{b}$:

1:	$\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$
2:	$\boldsymbol{p} = \boldsymbol{0}$
3:	do
4:	$\quad \boldsymbol{s} = \boldsymbol{M}^{-1}\,\boldsymbol{r}$
5:	$\quad \delta_i = (\boldsymbol{s}, \boldsymbol{r})$
6:	$\quad \beta = \delta_i / \delta_{i-1}$
7:	$\quad \boldsymbol{p} = \boldsymbol{s} + \beta\boldsymbol{p}$
8:	$\quad \boldsymbol{s} = \boldsymbol{A}\,\boldsymbol{p}$
9:	$\quad \gamma = (\boldsymbol{s}, \boldsymbol{p})$
10:	$\quad \alpha = \delta_i / \gamma$
11:	$\quad \boldsymbol{x} = \boldsymbol{x} + \alpha\boldsymbol{p}$
12:	$\quad \boldsymbol{r} = \boldsymbol{r} - \alpha\boldsymbol{s}$
13:	while (!stopCondition(|$\boldsymbol{r}$|, ++i))

(где $\boldsymbol{M}^{-1}$ — предобуславливатель, $\boldsymbol{r}$ — вектор невязки).

В случае многопроцессного счета на каждой итерации алгоритма выполняются следующие MPI-коммуникации:

- три глобальные редукции: в операциях взятия скалярного произведения и нормы вектора;
- обмен приграничными коэффициентами вектора $\boldsymbol{p}$ в операции домножения на матрицу $\boldsymbol{A}$ во всех парах MPI-процессов, имеющих общую границу.

При переносе описанного алгоритма на GPU необходимы реализации домножения разреженной матрицы на вектор и предобуславливателя. Для больших систем производительность этих операций является определяющей.

В нашей реализации используется ранее разработанная библиотека, использующая автоматическую настройку для повышения производительности на GPU [3]. Кроме этого, необходимы реализации взятия скалярного произведения, нормы, линейной комбинации векторов на GPU.

Отметим, что в случае однопоточного счета без нетривиальных граничных условий алгоритм не требует синхронизации CPU и GPU за исключением копирования и проверки нормы невязки $|\boldsymbol{r}|$. Копирование можно выполнять асинхронно, обеспечив перекрытие с дальнейшими операциями на GPU. Все скалярные коэффициенты (α, β, γ, δ_i) достаточно хранить в памяти акселератора. Для этого требуются следущие простые модификации:

- процедуры, выполняющие операции линейной алгебры на GPU должны принимать коэффициенты как указатели в память акселератора;
- обновления коэффициентов ($\beta = \delta_i / \delta_{i-1}$, $\alpha = \delta_i / \gamma$) должны выполняться как запуск тривиального ядра на GPU из одной нити.

При многопоточном счете коэффициенты δ_i и γ являются результатами MPI-редукций, и поэтому не могут храниться только в памяти акселератора. Для достижения хорошей эффективности предлагается хранить коэффициенты, участвующие в MPI-редукциях в pinned-памяти: в специально зарегистрированных страницах оперативной памяти, которые доступны для модификации с акселератора (однако стоимость доступа высока, так он выполняется через шину PCI-Express). Коэффициенты, которые не участвуют в MPI-редукциях (α, β) необходимо хранить только в памяти акселератора, так как далее они многократно читаются в ядрах, выполняющих операции над векторами.

3. Предобуславливание

Выбор эффективного метода предобуславливания для GPU не очевиден. В OpenFOAM обычно используется вариант ILU-предобуславливания, но алгоритмы этого семейства плохо подходят для GPU, так как применение такого предобуславливателя по сути представляет собой решение разреженных треугольных систем уравнений, и не обладает высоким

параллелизмом. Хотя параллелизм ILU-предобуславливателей можно повышать за счет изменения нумерации сетки, это приводит к снижению эффективности предобуславливателя.

Существует ряд предобуславливателей, основанных на построении приближенной обратной матрицы системы в явном виде [4]. Применение такого предобуславливателя сводится к домножению на одну или две разреженные матрицы — операции, для которой в любом случае нужна эффективная реализация на GPU. С другой стороны, вычисление такого предобуславливателя является более дорогостоящей операцией по сравнению с ILU.

В нашей реализации используется метод AINV [5], в котором приближенная обратная матрица строится как произведение двух разреженных треугольных матриц:

$$W^T A Z \approx D$$

(W, Z — верхнетреугольные матрицы с единицами на диагонали, D — диагональная матрица). Структура разреженности множителей W и Z определяется скалярным параметром drop tolerance: при построении множителей все элементы, по модулю меньшие drop tolerance, отбрасываются.

Для повышения производительности используются следующие модификации:

- Предобуславливатель хранится с использованием вещественных чисел одинарной точности (float). Поскольку операция домножения на разреженную матрицу ограничена пропускной способностью памяти, а высокая точность для предобуславливателя не требуется, эта оптимизация допустима и ускоряет применение предобуславливателя.
- Хотя формально применение предобуславливателя состоит из домножения на две разреженные матрицы и одну диагональную, при подготовке предобуславливателя все диагональные коэффициенты можно внести в разреженные множители. Это позволяет сократить количество необходимых доступов к памяти.
- Применение предобуславливателя требует домножения на одну верхнетреугольную и одну нижнетреугольную матрицу. При домножении на нижнетреугольную матрицу первыми обрабатываются сравнительно незаполненные строки матрицы, а более плотные строки у «основания» треугольника начинают обрабатываться в последнюю очередь. Для более эффективно сбалансированной нагрузки желательно, наоборот, запускать CUDA-блоки, обрабатывающие наиболее заполненные группы строк в первую очередь. Для этого предлагается переупорядочить разреженные множители AINV в лево-верхнетреугольную форму: пусть предобуславливатель имеет вид $\boldsymbol{UL}$ ($\boldsymbol{L} = \boldsymbol{U}^T$), $\boldsymbol{J}$ — обменная (антидиагональная единичная) матрица; тогда $\boldsymbol{UL} = \boldsymbol{U}(\boldsymbol{JJ})\boldsymbol{L} = (\boldsymbol{UJ})(\boldsymbol{JL})$, где оба множителя имеют лево-вернетреугольную форму и получаются из оригинальных обращением порядка строк или

столбцов. Построение предобуславливателя в таком виде никак не влияет на остальную логику и повышает эффективность работы.

- Наконец, необходимо сократить затраты на вычисление предобуславливателя: однопоточная реализация подсчета AINV на процессоре требует времени, сопоставимого с дальнейшим решением линейной системы на GPU с этим предобуславливателем. Амортизировать эти расходы предлагается следующим образом. Во-первых, разрешить переиспользование предобуславливателя для одной системы на соседних шагах по времени. Во-вторых, переиспользование предобуславливателя открывает возможность вычислять его асинхронно в отдельной нити, параллельно с решением системы на GPU. Асинхронное вычисление предобуславливателя привносит недетерминизм в расчеты, так как моменты обновления предобуславливателя на GPU могут отличаться от запуска к запуску. Кроме того, при выходе из цикла метода сопряженных градиентов желательно останавливать побочную нить, чтобы она не конкурировала с основными расчетами за ресурсы CPU.

4. Поддержка параллельных расчетов

В OpenFOAM параллельный запуск через MPI поддерживается за счет разбиения сетки. Граничные грани получаемых подсеток принадлежат либо множеству оригинальных граничных граней, либо множеству граней процессорных границ, возникших в результате разбиения. Процессорные границы в OpenFOAM обрабатываются как частный случай общих граничных условий (однако в отличие от большинства граничных условий, обусловленных решаемой задачей и задаваемых пользователем, процессорные граничные условия возникают автоматически при разбиении расчетной области между процессами). Все граничные условия обрабатываются при выполнении операции $\boldsymbol{s} = \boldsymbol{A}\,\boldsymbol{p}$ в дополнение к домножению на явно хранимые коэффициенты матрицы. Соответственно, при параллельном счете корректная обработка приграничных коэффициентов обеспечивается неявно за счет виртуальных процессорных граничных условий (в OpenFOAM граничные условия образуют иерархию C++ классов, и обновление коэффициентов реализовано через виртуальные функции). С другой стороны, даже при однопоточном счете эффект от домножения на матрицу системы определяется не только явно хранимыми коэффициентами, но еще и набором граничных условий для текущей сетки.

Таким образом, в GPU-реализации итерационных методов необходимо поддерживать обработку граничных условий в умножении на матрицу системы, и этого будет достаточно для поддержки параллельных запусков. Обработка граничных коэффициентов в OpenFOAM выполняется запуском методов initMatrixInterfaces и updateMatrixInterfaces класса lduMatrix, вызывающих в свою очередь виртуальные методы initInterfaceMatrixUpdate и

updateInterfaceMatrix класса lduInterfaceField. В случае процессорных границ первый метод запускает пересылку локальных граничных коэффициентов, а второй принимает граничные коэффициенты «соседнего» процессора.

Для GPU эти шаги предлагается выполнять следующим образом. Перед выполнением initMatrixInterfaces на акселераторе запускается ядро, которое копирует все граничные коэффициенты в один плотный вектор; для этого по сетке заранее строится индексный массив граничных коэффициентов. Полученный вектор копируется во вспомогательный массив, лежащий в pinned-памяти. Это копирование можно перекрыть с домножением на матрицу на GPU. Далее граничные коэффициенты из вспомогательного массива копируются в массив в соответствии со своей оригинальной нумерацией, и этот массив используется для запуска initMatrixInterfaces. Далее, после выполнения updateMatrixInterfaces нужно проделать аналогичные шаги в обратном порядке: собрать обновленные коэффициенты в плотный массив на процессоре, скопировать на GPU и просуммировать с результатами домножения на матрицу системы. Для повышения эффективности updateMatrixInterfaces и последующее копирование в память акселератора можно перекрывать с другими ядрами на GPU.

5. Перекрытие MPI-коммуникаций

В методе сопряженных градиентов MPI-коммуникации возникают в четырех местах: в умножении на матрицу системы, взятии нормы вектора невязки и вычислении скалярных произведений. Хотя MPI-коммуникации крайне желательно перекрывать с вычислениями, это не выполняется для глобальных редукций после скалярых произведений, которые сразу же используются для последующих вычислений. Обмен граничными коэффициентами в OpenFOAM производится параллельно с умножением на матрицу локально, а взятие нормы вектора невязки теоретически можно выполнять параллельно с другими вычислениями со следующей итерации (однако необходимые для этого вызовы для асинхронных редукций появились только в последней ревизии стандарта MPI: 3.0, и не используются в OpenFOAM).

Таким образом, на каждой итерации цикла сопряженных градиентов выполняются две или три синхронные редукции, во время которых не выполняется никаких вычислений. Для гетерогенных систем с акселераторами проблема усугубляется тем, что перед запуском MPI-редукции необходимо дождаться окончания вычисления нужного коэффициента на акселераторе; при этом желательно избежать полной CPU-GPU синхронизации, заранее запустив другие независимые вычисления на акселераторе.

Существует несколько реализаций метода сопряженных градиентов, направленных на увеличение перекрытия вычислений и коммуникаций. В них возникает компромисс между степенью перекрытия и количеством дополнительно проводимых вычислений и хранимых векторов. Для использования на GPU нами был использован следующий вариант [6]:

```
1:      r = b – Ax
2:      u = M^-1 r
3:      w = Au
4:      γ_-1 = ∞
5:      γ_0 = (r, u)
6:      δ = (w, u)
7:      do
8:          m = M^-1 w
9:          n = Am
10:         β = γ_i / γ_i-1
11:         α = γ_i / (δ – β γ_i / α)
12:         z = n + βz
13:         q = m + βq
14:         s = w + βs
15:         p = u + βp
16:         x = x + αp
17:         r = r – αs
18:         u = u – αq
19:         w = w – αz
20:         γ_i = (r, u)
21:         δ = (w, u)
22:     while (!stopCondition(|r|, ++i))
```

Приведенный алгоритм обладает следующими свойствами:

- глобальные редукции в операциях взятия скалярных произведений и нормы невязки можно полностью перекрыть с умножением на предобуславливатель и матрицу на следующей итерации;
- все три редукции можно выполнять одновременно как один вызов all-reduce на трехкомпонентном векторе $\{\gamma_i, \delta, |r|\}$;
- алгоритм требует меньше ядер на итерацию: вычисления в строках 12-22 требуют запуска лишь двух ядер (первое вычисляет все линейные комбинации и частичные скалярные произведения/нормы в пределах блока нитей; второе ядро состоит из одного блока и собирает частичные суммы в одну);
- обновление граничных коэффициентов при умножении на матрицу можно перекрыть с обновлением векторов ***q***, ***s***, ***p***, ***x***, ***r***, ***u***, если вынести обновление векторов ***z*** и ***w*** в отдельное ядро.

Недостатком этого алгоритма является увеличение количества хранимых на акселераторе векторов и доступов к памяти на каждой итерации. Поскольку пропускная способность памяти является определяющим фактором в производительности, при небольшой стоимости коммуникаций этот метод будет показывать более низкую производительность по сравнению с обычной реализацией.

6. Результаты тестирования

Описанные оптимизации были реализованы и протестированы в модуле решения линейных систем для проекта FOAM-Extend. Для тестирования производительности использовались гетерогенные узлы с процессорами Xeon E5 2690v2 (10 ядер, 3.0 GHz) и акселераторами Tesla K20X. В качестве эталонной процессорной реализации использовался метод PCG с предобуславливателем DIC. Многосеточный метод (GAMG) во всех случаях показывает более высокую производительность.

На тесте motorBike (сетка с 3.6 млн. ячеек) с приложением pisoFoam были получены следующие результаты:

- при запуске 10 процессов и использованием 1 GPU, 100 шагов:
  - с использованием только процессоров:
    - 1136 секунд в методе сопряженных градиентов
    - 2046 секунд общее время счета;
  - с использованием гибридной реализации:
    - 649 секунд в методе сопряженных градиентов;
    - 1536 секунд общее время;
- при запуске 80 процессов и использованием 8 GPU, 500 шагов:
  - с использованием только процессоров:
    - 502 секунды в методе сопряженных градиентов
    - 926 секунд общее время счета;
  - с использованием гибридной реализации:
    - 651 секунда в методе сопряженных градиентов;
    - 1093 секунды общее время.

Конфигурация с одним GPU демонстрирует ускорение в 1.75 раз в методе решения линейных систем. При использовании 8 GPU наблюдается замедление от использования акселераторов. Одной из проблем является низкая эффективность работы акселераторов из-за относительно малого количества элементов (по нашему опыту, для достижения высокой пропускной способности памяти в нашей реализации вычислительные ядра должны обрабатывать векторы с размерностью больше миллиона). С другой стороны, при переходе к 80 ядрам процессорная реализация показывает

сверхлинейное ускорение, так как больше обращений попадает в процессорные кеши.

Табл. 1. Масштабирование GPU-реализации.

Количество акселераторов								
Размер сетки	1	2	3	4	6	8	12	18
Время в методе сопряженных градиентов, с								
2 млн.	220	116	95	71	58			
8 млн.				320	253	172	142	107
Относительная эффективность параллельной работы								
2 млн.	1	0.95	0.77	0.77	0.63			
8 млн.				1	0.8	0.93	0.75	0.66

На синтетическом тесте cavity3d (сетки с 2 и 8 млн. ячеек) с приложением icoFoam проводились запуски для оценки масштабирования GPU-солвера. Использовались акселераторы Tesla K40, с количеством MPI-процессов равным количеству использованных акселераторов. В табл. 1 показаны экспериментальные результаты. Видно, что при менее чем 1 млн. ячеек на акселератор происходит снижение эффективности.

7. Заключение.

В данной работе представлено несколько оптимизаций для гибридной реализации метода сопряженных градиентов. С предложенными оптимизациями на GPU-акселераторах достигается более высокая производительность по сравнению с многоядерными процессорами, но только для задач достаточно большого размера. В будущем планируется исследовать возможности дальнейшего повышения производительности, реализовать метод решения для блочных систем в OpenFOAM (когда уравнения для давления и компонент скоростей решаются совместно) и разработать гибридный вариант для многосеточного метода.

Список литературы

[1]. Страница OpenFOAM — http://openfoam.org/
[2]. Y. Saad. Iterative Methods for Sparse Linear Systems. SIAM. 2003.
[3]. А. Монаков, Е. Велесевич, В. Платонов, А. Аветисян. Инструменты анализа и разработки эффективного кода для параллельных архитектур. Труды Института

системного программирования РАН, том 26, 2014 г. Выпуск 1. Стр. 357-374. DOI: 10.15514/ISPRAS-2014-26(1)-14
[4]. M. Benzi. Preconditioning Techniques for Large Linear Systems: A Survey. Journal of Computational Physics, 182 (2002), pp. 418-477.
[5]. R. Bridson, W.-P. Tang. Refining an Approximate Inverse. Journal on Computational and Applied Math, 123 (2000), Numerical Analysis 2000 vol. III: Linear Algebra, pp. 293-306.
[6]. P. Ghysels, W. Vanroose. Hiding Global Synchronization Latency in the Preconditioned Conjugate Gradient Algorithm. Submitted to Parallel Computing, 2012.

Optimizations for linear solvers in OpenFOAM for MPI + CUDA platform [2]

A. Monakov
<amonakov@ispras.ru>
V. Platonov
<soh@ispras.ru>
ISP RAS, 25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation

Abstract. We describe an implementation of conjugate gradient method on heterogeneous platforms (multiple nodes with GPU accelerators) to be used in OpenFOAM. Several optimizations are described. For conjugate gradient itself, we suggest using device memory for scalars used only on the GPU and pinned memory for scalars used in MPI reductions. For preconditioning, we choose AINV as a suitable preconditioner for GPUs and describe ways to make it more efficient, such as storing in it single precision, laying out factors in upper-left triangular form and computing it on the CPU asynchronously. We describe how multi-GPU computing can be supported together with arbitrary boundary conditions by copying only boundary coefficients from the accelerator to host memory and then using existing OpenFOAM methods on the CPU. To improve overlap of computations and communications, we suggest using a pipelined variant of the conjugate gradient method and describe GPU-specific adjustments. In experimental evaluation, we obtain a 1.75x speedup in the linear solver by using a Tesla K20X accelerator in addition to a 10-core Xeon CPU, but only for sufficiently large problem sizes: below 1 million cells per accelerator the efficiency of GPU computations dimishes.

Keywords: conjugate gradient method, AINV preconditioning, OpenFOAM, GPU, MPI.

References

[1]. OpenFOAM — http://openfoam.org/
[2]. Y. Saad. Iterative Methods for Sparse Linear Systems. SIAM. 2003.
[3]. A. Monakov, E. Velesevich, V. Platonov, A. Avetisyan. Instrumenty analiza i razrabotki effektivnogo koda dlya parallel'nykh arkhitektur [Analysis and development tools for efficient programs on parallel architectures]. Trudy ISP RAN [The Proceedings of ISP RAS], 2014, vol. 26, no 1. pp. 357-374 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-14
[4]. M. Benzi. Preconditioning Techniques for Large Linear Systems: A Survey. Journal of Computational Physics, 182 (2002), pp. 418-477.
[5]. R. Bridson, W.-P. Tang. Refining an Approximate Inverse. Journal on Computational and Applied Math, 123 (2000), Numerical Analysis 2000 vol. III: Linear Algebra, pp. 293-306.

2 The paper is supported by RFBR grant 13-07-12102

[6]. P. Ghysels, W. Vanroose. Hiding Global Synchronization Latency in the Preconditioned Conjugate Gradient Algorithm. Submitted to Parallel Computing, 2012.