

Апробация парадигмы автоадаптации GPGPU-программ в задачах аэрогидродинамики

М.А. Кривов, М.Н. Пritула, П.С. Иванов

В этой статье на примере двумерной и трехмерной модельных задач показан эффект от применения **технологии автоадаптации программ**, заключающейся в динамической подстройке значений ряда параметров расчетной программы под обрабатываемые данные и используемый графический ускоритель, что позволяет получить дополнительное ускорение до 50%. Эту технологию авторы протестировали на всех последних поколениях GPU (графических процессоров) от NVIDIA. Рассмотрены параметры, которые можно выделить в большинстве задач аэрогидродинамики (CFD), и оптимальные значения которых зависят от множества факторов.

1. Введение

Общей и, даже можно сказать, универсальной проблемой, которая появляется при разработке GPGPU-программ (то есть программ, задействующих в вычислениях графический процессор видеокарты) для проведения научных расчетов из различных предметных областей, являются слишком частые изменения в архитектуре GPU, выпускаемых даже одним производителем. К примеру [1], при переходе от ускорителя NVIDIA Tesla C1060 к NVIDIA Tesla C2075 количество ядер увеличилось почти в два раза – с 240 до 448. Последующий переход к очередному поколению ускорителей – NVIDIA Tesla K20 – также привел к увеличению количества ядер, но уже почти в 5.5 раз – с 448 до 2496. Очевидно, что все подобные изменения требуют внесения соответствующих модификаций в расчетные программы. Однако для учета данных особенностей необходимо проведение оптимизации под каждую целевую архитектуру, что является крайне трудоемким процессом, так как новые поколения GPU появляются примерно каждые два года. В противном случае могут наблюдаться даже такие ситуации, когда перенос существующей программы на новый GPU с вдвое большей производительностью приводит лишь к замедлению расчетов.

Одним из подходов к решению проблемы являются технологии автоадаптации программ, получившие в англоязычной литературе известность как *autotuning*. Их смысл заключается во встраивании в исходную программу механизмов, которые позволяют при ее запуске оценить возможности используемой аппаратной платформы, после чего осуществить некую подстройку программы. В соответствии с этой парадигмой, процесс происходит полностью автоматически, без участия программиста. Благодаря автоподстройке, удается

без особых затрат обеспечить поддержку различных поколений GPU. Более того, подобным образом можно создавать программы, которые сумеют сами себя ускорить при работе на моделях GPU, еще не вышедших в момент написания ПО, что в рамках традиционных подходов к оптимизации является невозможным. При этом надо признать, что в настоящий момент ускорение, достигаемое за счет подобных технологий, оказывается достаточно скромным – порядка всего лишь нескольких десятков процентов. Однако при проведении расчетов в течение продолжительного времени данный выигрыш может оказаться весьма существенным, так как позволяет не только быстрее получить результат, но и понизить расходы как на закупку вычислительных узлов, так и на оплату потребляемой электроэнергии.

Цель статьи – проиллюстрировать влияние на скорость расчетов ряда параметров, которые можно выделить практически во всех GPGPU-программах, проводящих вычисления на структурированных сетках (под которыми в дальнейшем будут пониматься в том числе и адаптивные сетки). При этом оптимальные значения данных параметров напрямую зависят от используемых моделей GPU, в результате чего предугадать их заранее не представляется возможным. Также будет показан эффект от автоматизации подбора параметров с помощью разработанной авторами технологии автотюнинга *TTG Apptimizer*.

2. Схожие работы

Наиболее близкой по тематике работой, в которой реализована рассматриваемая парадигма и результаты которой используются в коммерческом продукте, является реализация системы автоадаптации программы через директивы компилятора HMPP [2]. Компилятор автоматически подбирает такие параметры, как, например, развертка циклов (другими словами, сколько итераций необходимо “склеить” в одну большую), разбиение данных на блоки, а также переупорядочивание вложенных циклов. Так как подобные модификации программы осуществляются исключительно компилятором, то для разработчика они совершенно незаметны, но при этом позволяют получить некоторое ускорение. В упомянутой статье [2] тестирование проводилось для разных типов задач, в том числе на задаче свертки двумерного и трехмерного изображений с некоторым ядром, а также тестовых примерах из бенчмарка *PolyBench*. Если говорить о результатах, то в первую очередь они оказываются заметными

при смене архитектуры ускорителя – на некоторых тестах рассматриваемые подходы позволили ускорить расчеты до двух раз.

Еще одним примером реализации рассматриваемой парадигмы является работа [3], в которой проводилась оценка влияния ряда параметров на скорость работы *GPGPU*-программы, а также осуществлялось тестирование разработанного динамического автотюнера *AtuneRT*. Достижимое ускорение оценивалось при использовании трех *GPU* из двух разных поколений и составило несколько десятков процентов. Стоит отметить, что в качестве одной из модельных задач в этой работе была выбрана операция *scan* из стандартной библиотеки *thrust*, которую удалось ускорить таким образом на 13%. Другой интересной особенностью является малое время обучения – порядка 10÷20 итераций, что удалось достичь за счет учета специфики оптимизируемых параметров.

Надо отметить, что идея автоадаптации программ получила развитие не только в виде инструментов для разработчика, как в двух рассмотренных выше работах, но и как свойство реализуемого алгоритма, что можно проиллюстрировать работой [4]. В ней, в частности, ставилась задача повысить скорость операции перемножения плотных матриц за счет подстройки алгоритма под целевую платформу. За счет схожих адаптаций размеров ряда блоков удалось заметно ускорить эту операцию для матриц с размером порядка 6144×6144.

Работы в данном направлении, естественно, не ограничиваются вышеперечисленными. Однако все они имеют схожие особенности:

1) предпринимается попытка повысить скорость расчетов за счет учета специфики обрабатываемых данных и аппаратной платформы;

2) оптимизация осуществляется автоматически (полностью или частично), не требуя от программиста специальных знаний о специфичных особенностях всех целевых платформ.

3. Оптимизируемые параметры

Особенностью структурированных сеток является возможность их отображения в равномерную сетку (возможно, с выколотыми элементами или подобластями), которая в дальнейшем может быть представлена в памяти как двумерный или трехмерный массив. В частности, это позволяет получить всех соседей каждого элемента просто путем смещения в массивах на строго фиксированные и заранее известные величины. В случае же структурированных адаптивных сеток возможно их отображение в набор равномерных сеток с выколотыми подобластями, которые будут периодически изменяться, но при этом тоже позволят получать соседей произвольного элемента путем сдвига по соответствующему массиву.

При решении различных уравнений математической физики на подобных сетках с использованием графических ускорителей в большинстве случаев можно выделить следующие общие параметры:

1) Количество элементов, обрабатываемых одной нитью

Идеология *GPGPU* предполагает, что количество нитей вычислений должно быть большим (обычно рекомендуется создавать в 8÷16 раз больше нитей, чем количество ядер *GPU*). При работе с большими сетками с некоторого момента выигрыш от подобного увеличения числа нитей становится незначительным, однако благодаря их “утяжелению” иногда удается получить некоторое дополнительное ускорение. В рассматриваемых задачах это означает, что с некоторого момента одной нитью следует обрабатывать сразу несколько элементов сетки, количество которых (а также их расположение) и является параметром.

2) Использование разделяемой памяти

Такой тип памяти является более быстрым, поэтому при работе с двумерными и трехмерными массивами возможно его использование в качестве программного кэша. Однако на современных ускорителях уже имеется аппаратный кэш *L1*, расположенный в памяти точно такого же типа. Поэтому возникает вопрос – в каких случаях следует полагаться на аппаратный кэш, а когда лучше самостоятельно реализовывать кэширование участков обрабатываемых массивов? Соответственно, параметром является тип реализации доступа к глобальной памяти.

3) Выбор типа вычислителя

В случае, когда расчеты могут проводиться с использованием только одного вычислителя, а в системе есть несколько подходящих устройств, то имеет смысл подбирать то, которое обеспечит наилучшую производительность. Обычно это сводится к перебору всех *GPU*, а также попытке проводить расчеты на всех ядрах *CPU* (и на встроенном *iGPU* при его наличии).

4) Разбиение области на блоки

Другой достаточно важный момент заключается в разбиении всей расчетной области на двумерные прямоугольные блоки, каждый из которых будет обработан одним потоковым мультипроцессором *GPU*. Проблемой является тот факт, что при выборе размеров блока требуется обеспечить возможность одновременного выполнения одним мультипроцессором как можно большего количества подобных блоков. С одной стороны, для этого требуется выбирать как можно меньшие, но кратные количеству ядер размеры (что позволит полностью задействовать ресурсы *GPU*), с другой – увеличение размера блоков в общем случае позволяет уменьшить количество выполняемых операций, и тем самым повысить скорость расчетов. Параметрами в данном случае являются ширина и высота блока.

5) Задание размера теневых граней

При проведении расчетов на нескольких *GPU* или сразу на *GPU*-кластере, требуется

минимизировать количество барьерных синхронизаций. Одним из вариантов является разбиение всей области на частично перекрывающиеся подобласти. В результате этого граничные значения удастся получить путем дублирования одних и тех же вычислений на “соседних” GPU, тем самым понизив частоту барьерных синхронизаций. Размер подобных теневого грани и является оптимизируемым параметром.

Все вышеперечисленные параметры на первый взгляд кажутся достаточно простыми и очевидными, однако их оптимальные значения зависят как от размера сеток, так и от типа используемых вычислителей и реализованного алгоритма, в результате чего их невозможно определить в момент компиляции. Более того, количество всевозможных комбинаций оказывается значительным, поэтому полный перебор с целью определения подходящих значений при проведении реальных расчетов практически неосуществим. Таким образом, логичным шагом становится автоматизация их подбора с помощью технологий автотюнинга.

4. Результаты тестирования

4.1. Схема тестирования

Для подбора значений, перечисленных в предыдущем разделе параметров, применялась разработанная авторами технология динамической адаптации программ, реализованная в виде библиотеки *TTC Apptimizer*. Суть её заключается в использовании модификаций классических методов оптимизации для поиска экстремумов функционала, определенного как время выполнения одной итерации алгоритма в зависимости от произвольного набора параметров. Ключевым моментом технологии являются именно произведенные модификации используемых методов, которые позволяют за минимальное количество замеров выйти на локальный экстремум, а также учитывают множество факторов – типа зависимости производительности от номера итерации (при фиксированных параметрах), кэш-эффектов и др.

В качестве тестовых программ были реализованы две описанные в последующих разделах модельные задачи, в рамках которых проводились расчеты на двумерной и трехмерной сетке. Подбор ряда перечисленных выше параметров осуществлялся для сеток разного размера и на всех последних поколениях GPU компании NVIDIA, начиная от ускорителя *GeForce 8800 GTX* и заканчивая *GeForce 680 GTX*. Ускорители серии *Tesla* не привлекались сознательно, так как в тестах использовалась только одинарная точность и, что более важно, далеко не все GPU от NVIDIA нашли воплощение в серии *Tesla*. Поэтому на основе массовых видеокарт можно более качественно оценить эффект от подобной оптимизации программ.

Выигрыш от применения технологий автотюнинга оценивался как ускорение, которое удалось получить за счет предложенной авторами реализации

этой идеи, по сравнению с исходной (“статической”) программой. Сразу надо отметить, что в нижеприведенных результатах не учитывается время обучения, так как при проведении промышленных расчетов его вклад оказывается крайне малым, в то время как в рассматриваемых тестовых программах из-за небольшого количества итераций (порядка $500 \div 1000$) оно может оказаться существенным. Другим моментом, на котором стоит остановиться, является вопрос об адекватности исходных значений параметров, которые использовались в статической версии. Действительно, подобрав “плохие” начальные параметры, можно получить колоссальное ускорение. Чтобы избежать этого, при разработке тестовых программ для каждого параметра вручную перебирались несколько наиболее “популярных” значений (например, для размера блока брались величины 128, 256 и 512), после чего выбиралась та комбинация, которая на текущем GPU и текущих данных обеспечивает наилучшую производительность.

4.2. Тестирование на двумерной сетке

Для двумерной сетки тестирование проводилось при решении уравнения теплопроводности в постановке задачи Дирихле с использованием явной разностной схемы, непрерывная и дискретная запись которой имеет следующий вид:

$$\frac{U}{\partial t} = \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}$$

$$0 < x < 1$$

$$0 < y < 1$$

$$0 < t < T$$

$$U(x, y, t=0) = f(x, y)$$

$$U(x, y, t)|_{x=0,1} = \phi(y, t)$$

$$U(x, y, t)|_{y=0,1} = \theta(x, t)$$

$$V_{i,j}^{k+1} = V_{i,j}^k + dt \cdot \left(\frac{V_{i-1,j}^k - 2 \cdot V_{i,j}^k + V_{i+1,j}^k}{dx^2} + \frac{V_{i,j-1}^k - 2 \cdot V_{i,j}^k + V_{i,j+1}^k}{dy^2} \right)$$

$$i = \overline{1, M-1}, j = \overline{1, N-1}, k = \overline{0, K-1}$$

$$dx = \frac{1}{M}, dy = \frac{1}{N}, dt = \frac{T}{K}$$

$$V_{i,j}^0 = f(i \cdot dx, j \cdot dy)$$

$$V_{i,j}^k|_{i=0,M} = \phi(j \cdot dy, k \cdot dt)$$

$$V_{i,j}^k|_{j=0,N} = \theta(i \cdot dx, k \cdot dt),$$

где M и N задают размеры двумерной сетки, накладываемой на дискретизируемую область, а число K определяется исходя из условия Куранта.

В реализации данного алгоритма было выделено три типа параметров: размеры двумерного блока, которые используются при разбиении области для её последующего распределения по всем потоковым мультипроцессорам GPU, и номер задействованного в системе устройства (в наших тестах – либо все ядра CPU, либо одно устройство GPU). Реализация для центрального процессора была разработана с применением расширений SSE и технологии *OpenMP*, поэтому для небольших сеток она

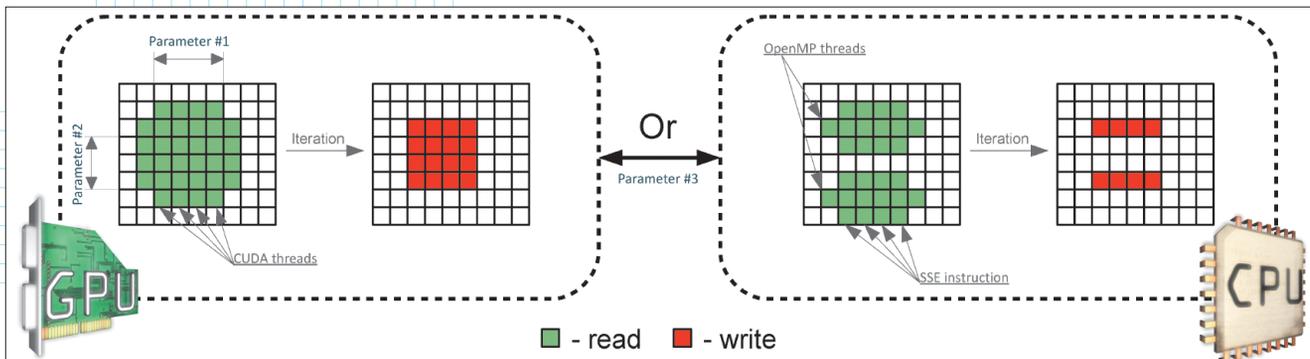


Рис. 1. Введенные параметры в двумерной тестовой задаче

оказывается предпочтительнее, чем реализация для GPU, созданная с использованием технологии NVIDIA CUDA. Озвученные выше параметры проиллюстрированы на рис. 1.

Перед тем как приводить результаты полного тестирования на всех типах ускорителей, имеет смысл рассмотреть эффект от технологий автотюнинга для одной системы на базе двух 6-ядерных процессоров Intel Xeon 5650 и трех ускорителей Tesla M2050, но для разных данных (рис. 2).

Как видно, на небольших сетках предпочтительнее проводить расчеты на CPU, поскольку в этом случае не требуется копировать данные по шине PCI-E. Однако с некоторого момента массивы перестают помещаться в кэш процессора, в результате чего производительность резко падает, и более выгодным становится использование графического ускорителя. Более того, даже для него путем подбора всего двух параметров удается существенно повысить скорость вычислений, что особенно заметно на больших сетках.

Результаты полного тестирования на различных типах ускорителей и сетках разного размера представлены в табл. 1. Модели графических ускорителей в ней указаны в сокращенном виде (например, GF9800 обозначает NVIDIA GeForce 9800 GTX+), а значения производительности определены как два числа – производительность “статической” версии (в гигафлопсах) и ускорение, достигнутое

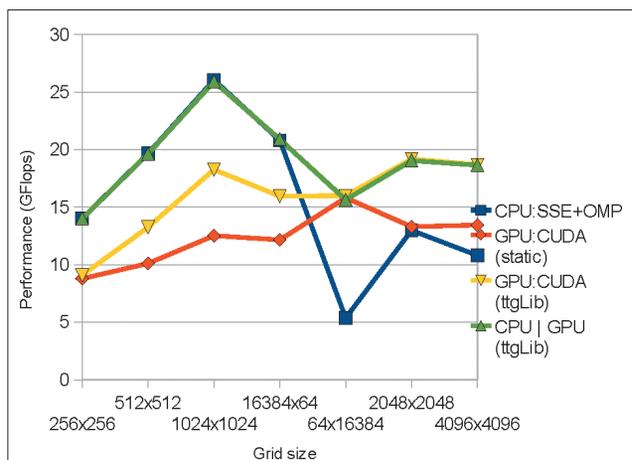


Рис. 2. Производительность разных реализаций алгоритма решения двумерной тестовой задачи

за счет автоадаптации программы (в процентах). Стоит отметить, что вычисления проводились только на графических ускорителях без использования центрального процессора.

Нетрудно заметить, что среднее ускорение (посчитанное как среднее арифметическое от достигнутых ускорений на всех тестах) для большинства GPU составляет несколько десятков процентов. Обусловлено это тем, что с ростом количества

Табл. 1. Результаты тестирования технологии на двумерной задаче на всех GPU

Размер сетки	GF 8800	GF 9800	GF 285	GF 480	GF 580	GF 680
256x256	3.63+3.86%	3.32+0.60%	3.00+2.00%	8.12+7.02%	9.11+0.11%	7.84+21.56%
384x384	3.90+1.54%	3.76+0.27%	3.53+0.85%	9.22+12.26%	11.95+6.86%	10.24+12.79%
512x512	5.07+2.37%	4.83+0.83%	5.17+16.05%	9.49+26.98%	14.05+15.09%	12.49+18.33%
768x768	6.26+2.4%	6.11+0.65%	7.68+23.18%	10.66+40.24%	17.08+28.57%	15.00+39.47%
1024x1024	6.84+1.75%	6.66+0.30%	9.35+28.45%	11.28+44.15%	18.00+27.11%	16.29+36.22%
1536x1536	7.29+2.06%	6.68+3.14%	11.05+37.83%	11.8+42.20%	19.91+27.32%	17.57+33.92%
2048x2048	7.49+1.47%	6.88+2.62%	11.87+42.21%	11.89+42.39%	20.78+25.99%	18.31+31.95%
3072x3072	7.58+1.98%	6.82+2.79%	12.5+45.6%	11.99+43.70%	21.5+21.35%	18.66+32.10%
4096x4096	7.67+1.43%	6.75+3.41%	12.76+46.63%	11.95+46.44%	21.98+18.38%	18.79+30.81%
6144x6144	7.59+2.64%	6.72+3.72%	12.93+46.64%	11.92+46.06%	21.98+17.93%	18.63+32.37%
Среднее ускорение	.+ 2.14%	.+ 1.78%	.+ 28.77%	.+ 35.14%	.+ 18.87%	.+ 28.95%

вычислительных ядер требуется иначе проводить разбиение данных на блоки – с учетом как специфики самого ускорителя, так и размера сетки. Максимальный эффект был зафиксирован на сетках большого размера и GPU поколения Fermi (обозначены как GF480 и GF580), на которых программу удалось ускорить в 1.46 раза. Минимальный – на сетке размера 384×384 и ускорителе с архитектурой G92b (GF9800), когда наблюдалось даже замедление программы на 0.27%. Это объясняется отсутствием в этом тесте возможностей для ускорения программы и влиянием накладных расходов операционной системы и драйвера. Действительно, поскольку применяемая технология автоадаптации основана на поиске экстремума функционала времени выполнения одной итерации, то любые системные задержки могут привести к выбору ложного экстремума. А для случаев, когда программа и так уже оптимальна, это означает некоторую, хоть и крайне незначительную, потерю производительности.

В табл. 2 приведены результаты аналогичного тестирования, но с тем отличием, что технологии автоадаптации было разрешено “перебрасывать” вычисления на центральный процессор Intel Xeon E3-1230.

В соответствии с полученными результатами, устаревшие поколения GPU, основанные на архитектурах G80 и G92b (обозначены как GF8800 и GF9800), на данной задаче и всех тестовых сетках оказались менее производительными, чем современный CPU. Что является более интересным наблюдением – для более новых, но всё же устаревших поколений GT200 и Fermi (GF285, GF480, GF580), выигрыш от переноса вычислений на центральный процессор тоже оказывается заметным, но лишь для сеток небольшого размера. Наконец, в случае использования современной архитектуры GPU Kepler1 (GF680) перенос вычислений на центральный процессор не позволяет достичь какого-либо ускорения ни на одной сетке, поэтому в подобных сценариях использовался лишь графический ускоритель.

Как показывает практика, в рабочих машинах реальных пользователей расчетных программ

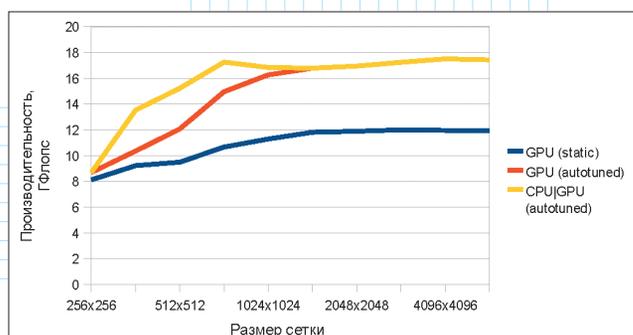


Рис. 3. Результаты тестирования технологии на двумерной задаче и ускорителе GeForce 480 GTX

могут быть установлены какие угодно ускорители, поэтому такой вид оптимизации, как динамический выбор наиболее подходящего устройства, может оказаться крайне полезным. Однако его реализация программистом “вручную” обычно является достаточно трудоемкой, в связи с чем выбор используемого вычислителя (CPU или GPU) в большинстве случаев ложится исключительно на пользователя, который при запуске программы должен указать соответствующий аргумент. Применение технологий автоадаптации программ фактически устраняет эту проблему, позволяя в ряде случаев получить “бесплатное” ускорение вычислений до трех раз.

Имеет смысл еще раз отдельно проиллюстрировать результаты работы рассматриваемой технологии при использовании только одного ускорителя NVIDIA GeForce 480 GTX, что показано на рис. 3.

Таким образом, динамически подстраивая всего лишь три параметра тестовой программы, удается её ускорить в среднем в 1.45 раза; при этом проведение оптимизации подобных видов практически не требует дополнительных затрат со стороны разработчика.

4.3. Тестирование на трехмерной сетке

В качестве теста для трехмерного случая было выбрано уравнение Пуассона в постановке задачи Дирихле, решаемое с помощью итерационного

Табл. 2. Результаты тестирования технологии на двумерной задаче на всех связках CPU+GPU

Размер сетки	GF 8800	GF 9800	GF 285	GF 480	GF 580	GF 680
256×256	3.63+127.83%	3.32+156.10%	3.00+193.23%	8.12+7.02%	9.11+0.11%	7.84+21.56%
384×384	3.90+251.26%	3.76+244.85%	3.53+273.19%	9.22+46.75%	11.95+8.76%	10.24+12.79%
512×512	5.07+193.95%	4.83+212.99%	5.17+193.91%	9.49+60.38%	14.05+15.09%	12.49+18.33%
768×768	6.26+172.93%	6.11+177.40%	7.68+117.97%	10.66+61.73%	17.08+28.57%	15.00+39.47%
1024×1024	6.84+142.39%	6.66+162.30%	9.35+82.96%	11.28+49.20%	18.00+27.11%	16.29+36.22%
1536×1536	7.29+23.47%	6.68+42.714%	11.05+37.83%	11.8+42.20%	19.91+27.32%	17.57+33.92%
2048×2048	7.49+27.82%	6.88+35.16%	11.87+42.21%	11.89+42.39%	20.78+25.99%	18.31+31.95%
3072×3072	7.58+26.12%	6.82+38.70%	12.5+45.60%	11.99+43.70%	21.5+21.35%	18.66+32.10%
4096×4096	7.67+22.36%	6.75+44.06%	12.76+46.63%	11.95+46.44%	21.98+18.38%	18.79+30.81%
6144×6144	7.59+37.71%	6.72+48.68%	12.93+46.64%	11.92+46.06%	21.98+17.93%	18.63+32.37%
Среднее ускорение	+ 102.59%	+116.30%	+ 108.02%	+ 44.59%	+ 19.06%	+ 28.95%

метода Якоби. Таким образом, непрерывная задача имела следующий вид:

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + \frac{\partial^2 U}{\partial z^2} = f(x, y, z)$$

$$0 < x < 1$$

$$0 < y < 1$$

$$0 < z < 1$$

$$U(x, y, z)|_{(x, y, z) \in G} = g(x, y, z)$$

Дискретная задача может быть записана как:

$$\frac{V_{i-1,j,l} - 2 \cdot V_{i,j,l} + V_{i+1,j,l}}{dx^2} + \frac{V_{i,j,l-1} - 2 \cdot V_{i,j,l} + V_{i,j,l+1}}{dy^2} + \frac{V_{i,j,l-1} - 2 \cdot V_{i,j,l} + V_{i,j,l+1}}{dz^2} = f(i \cdot dx, j \cdot dy, l \cdot dz)$$

$$i = \overline{1, M-1}, j = \overline{1, N-1}, l = \overline{0, K-1}$$

$$dx = \frac{1}{M}, dy = \frac{1}{N}, dz = \frac{1}{K}$$

$$V_{i,j,l} |_{i \in \{0, N\}, j \in \{0, M\}, l \in \{0, K\}} = g(i \cdot dx, j \cdot dy, l \cdot dz),$$

где M , N и K задают размер накладываемой на область сетки.

Поскольку объем независимых операций крайне большой, то в реализации были введены три основных параметра: первые два определяют размер двумерного CUDA-блока, а третий задает количество узлов, обрабатываемых одной нитью. Другими словами, одним мультипроцессором выполняются операции сразу для подобласти, высота и ширина (координаты X и Y) которой задаются двумя первыми параметрами, а глубина (координата Z) – третьим. Кроме того, был введен дополнительный четвертый параметр, который задает тип вычислительного ядра – с использованием разделяемой памяти или без; все четыре параметра показаны на рис. 4.

Перед тем, как переходить непосредственно к описанию результатов тестирования, следует показать, что оптимальные значения введенных параметров действительно зависят от модели ускорителя, а также существенно влияют на производительность. На рис. 5 приведен график скорости расчетов в зависимости от количества элементов, обрабатываемых одной нитью, для ускорителя *NVIDIA GeForce 285 GTX*.

В вычислительном ядре под номером один не используется разделяемая память, поэтому в большинстве случаев это ядро оказывается не столь эффективным, так как в данном GPU нет аппаратного кэша. При этом укрупнение нитей дает положительный и достаточно заметный эффект: при

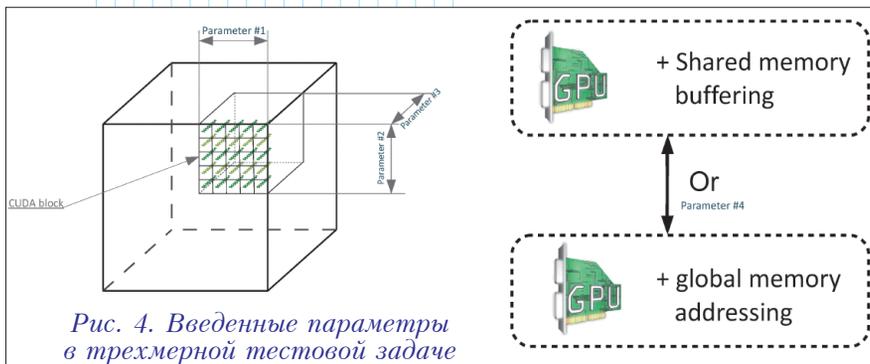


Рис. 4. Введенные параметры в трехмерной тестовой задаче

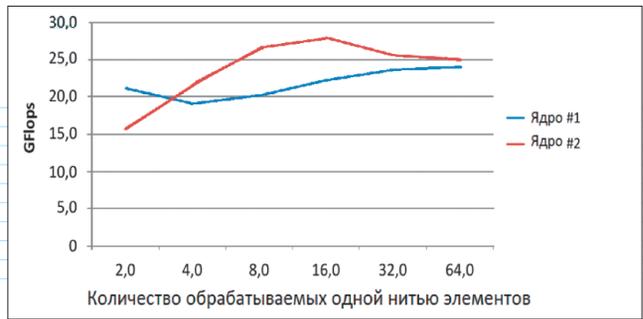


Рис. 5. Зависимость производительности от одного из параметров, ускоритель GeForce 285 GTX

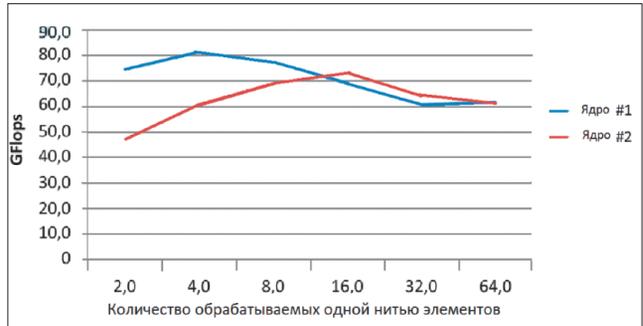


Рис. 6. Зависимость производительности от одного из параметров, ускоритель GeForce 580 GTX

увеличении количества обрабатываемых элементов с 1 до 16 удается получить ускорение до 75%. Таким образом, при ручной оптимизации программист может сделать вывод, что следует всегда использовать реализацию ядра с разделяемой памятью и обрабатывать по 16 элементов одной нитью. Однако при переходе на более новый ускоритель *GeForce 580 GTX* выясняется, что этот вывод является ошибочным, что проиллюстрировано на рис. 6.

Как легко заметить, из-за смены ускорителя оптимальным вариантом оказался выбор ядра без разделяемой памяти и обработка по 4 элемента одной нитью. Если же в программе всегда использовать значения параметров, взятые на основе рис. 5, то в данном случае она будет работать на 11% медленнее. И это всё при том, что оба теста проводились только на одной сетке размером $128 \times 128 \times 128$ элементов, а варьировался всего один параметр. При реальных же расчетах количество подобных параметров может доходить до десятка, а сетки, естественно, будут иметь совершенно разные

размеры. Таким образом, в “статической” версии программы потери от “неправильного” выбора значений оптимизационных параметров могут оказаться весьма существенными.

По аналогии с предыдущим разделом, отдельно следует проиллюстрировать эффект от технологии автоадаптации программ для одного ускорителя *GPU GeForce 680 GTX*, но для сеток разного размера. Соответствующие результаты приведены на рис. 7.

Табл. 3. Результаты тестирования технологии на трехмерной задаче на всех ускорителях

Размер сетки	GF 8800	GF 9800	GF 285	GF 480	GF 580	GF 680
64×64×64	9.7 + 10.1%	11.7 + 0.4%	23.1 + 4.1%	54.2 + 3.0%	61.1 + 17.9%	55.9 + 9.9%
96×96×96	10.1 + 18.5%	12.3 + 9.2%	29.9 + 1.9%	59.0 + 22.9%	69.3 + 25.0%	60.3 + 15.2%
128×128×128	10.2 + 16.1%	9.4 + 30.4%	23.1 + 4.2%	55.2 + 25.4%	63.5 + 28.6%	47.7 + 49.2%
160×160×160	10.3 + 21.2%	11.7 + 9.6%	25.5 + 16.8%	61.5 + 22.7%	71.4 + 33.1%	54.9 + 33.2%
192×192×192	10.2 + 21.2%	10.8 + 12.0%	25.8 + 6.1%	60.5 + 22.7%	70.4 + 33.5%	49.1 + 48.9%
Среднее ускорение	+17.4%	+12.3%	+6.6%	+19.4%	+27.6%	+31.3%

Как мы видим, в зависимости от размеров сетки подбор более подходящих параметров позволяет получить ускорение до 40%. При этом на исходной сетке, для которой проводилась ручная оптимизация статической версии, выигрыш практически незаметен, что объясняется удачным подбором начальных значений всех четырех параметров: разбиением на блоки 16×16×8 и выбором версии без разделяемой памяти.

Результаты тестирования на всех рассматриваемых видах ускорителей и разных сетках приведены в табл. 3. Как и в предыдущем разделе, в таблице указана достигнутая производительность (в гигафлопсах), а также дополнительное ускорение от применения технологии автоадаптации программ (в процентах).

Максимальный эффект был достигнут на устройствах последнего поколения – GPU *Kepler1*, представителем которого является ускоритель *GeForce 680 GTX*: на нём ускорение составило от 10% до 49%. Это объясняется тем, что исходная тестовая программа (без автоадаптации) была разработана задолго до появления архитектуры *Kepler1*, поэтому заложенные в ней значения параметров даже без привязки к размерам сеток оказались далеко не самыми оптимальными.

5. Заключение

В статье, на примере предложенной авторами реализации парадигмы *autotuning* в виде технологии *TG Apptimizer*, показан механизм адаптации двух тестовых программ, решающих модельные задачи из области аэрогидродинамики на двумерной и трехмерной сетках. Благодаря полуавтоматической оптимизации, достигнуто ускорение вычислений от нескольких десятков процентов до двух-трех раз – в зависимости от размера расчетной области и используемого ускорителя. Тестирование, проведенное для всех современных поколений GPU компании *NVIDIA*, подтвердило основное преимущество

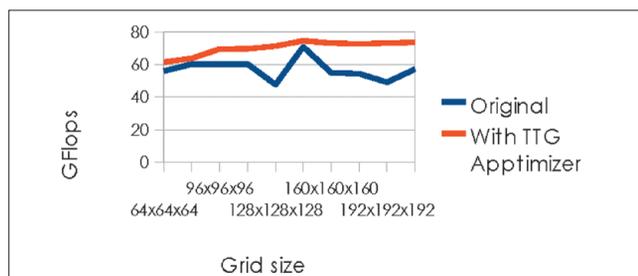


Рис. 7. Результаты тестирования технологии на трехмерной задаче и ускорителе *GeForce 680 GTX*

рассматриваемой парадигмы: с её помощью можно обеспечить оптимизацию программы даже для таких архитектур, к которым разработчик еще не имеет доступа и о специфике которых он не знает (в том числе и из-за того, что соответствующие ускорители еще не разработаны).

В качестве основного отличия от работ предшественников можно назвать попытку более глобально подойти к реализации идеи автоадаптации. В частности, подстройка ПО производится не только для одного-двух основных параметров, которые присутствуют в любой GPGPU-программе, а сразу для множества параметров, причем, свойственных не только GPU: часть параметров удается выделить исключительно благодаря учету специфики предметной области. Более того, в отличие от работ предшественников, тестирование проводилось на всех поколениях GPU, что позволяет более точно оценить ускорение вычислений, достигаемое за счет подобных оптимизаций.

Дальнейшие усилия, в первую очередь, будут направлены на оценку эффекта от использования подобных технологий для решения проблем из самых разных предметных областей; намечено провести аналогичное тестирование для задач, узким местом в которых является исключительно доступ к памяти. С учетом того, что в последующих поколениях ускорителей планируется добавление поддержки общей памяти, вполне может оказаться, что за счет адаптации опять же нескольких параметров скорость расчетов можно будет существенно повысить. ☺

Об авторах:

М.А. Кривов – ООО “ТТГ Лабс”, МГУ им. М.В. Ломоносова

М.Н. Пругула – ООО “ТТГ Лабс”

П.С. Иванов – МГУ им. М.В. Ломоносова

Литература

- Кривов М.А. От G80 и до GK110 // Суперкомпьютеры, 2013 (январь-февраль), с. 48–51.
- Grauer-Gray S., Xu L., Searles R., Ayalamayajula S., Cavazos J. Auto-tuning a high-level language targeted to GPU Codes // Innovative Parallel Computing (InPar 2012), 2012, pp. 1–10.
- Tillmann M., Karcher T., Dachsbacher C., Tichy W. Application-independent autotuning for GPU // International Conference on Parallel Computing (ParCo 2013): Application Autotuning for HPC (Architectures), 2013.
- Li Y., Dongarra J., Tomov S. A Note on auto-tuning GEMM for GPUs // Proceedings of the 9th International Conference on Computational Science (ICCS), 2009, part I, pp. 884– 892.