Применение технологий автоадаптации программ для решения CFD задач на структурированных сетках с использованием GPU

М.А. Кривов[1,2], М.Н. Притула[1], П.С. Иванов[2]

ООО «ТТГ Лабс»[1], МГУ им. М.В. Ломоносова[2]

В работе на примере двумерной и трёхмерной модельных задач показан эффект от использования технологии автоадаптации программ, заключающейся в динамической подстройке значений ряда параметров программы под обрабатываемые ей данные и используемый ускоритель, что позволяет получить дополнительное ускорение до 50%. Осуществлено тестирование данной технологии на всех последних поколениях GPU от NVidia, а также приведен список подобных параметров, которые можно выделить в большинстве CFD задач, и оптимальные значения которых зависят от множества факторов.

1. Введение

Общей и, даже можно сказать, «универсальной» проблемой, которая появляется при разработке GPGPU-программ для проведения научных расчётов 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 же переход к очередному поколению ускорителей NVidia Tesla K20 также привёл 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 расчётные программы. Однако для учёта данных особенностей необходимо проведение оптимизации под каждую целевую архитектуру, что является крайне трудоёмким процессом, так как новые поколения GPU появляются примерно каждые два года. В противном же случае могут даже наблюдаться ситуации, когда перенос существующей программы на новый GPU с вдвое большей производительностью приводит лишь к замедлению расчётов.

Одним из подходов к решению озвученной проблемы являются технологии автоадаптации программ, получившие в англоязычной литературе известность как autotuning. Их смысл заключается во встраивании в исходную программу механизмов, которые позволяют при её запуске оценить возможности используемой аппаратной платформы, после чего осуществить под неё некую подстройку программы. В соответствии с этой парадигмой, данный процесс происходит полностью автоматически и без участия программиста, и в то же время благодаря ему удаётся без особых затрат обеспечить поддержку различных поколений GPU. Более того, подобным образом возможно создание программ, которые сумеют сами себя ускорить при работе на ещё не вышедших моделях GPU, что в рамках традиционных подходов к оптимизации является невозможным. При этом стоит признать, что в настоящий момент ускорение, достигаемое за счёт подобных технологий, оказывается достаточно скромным – всего порядка нескольких десятков процентов. Однако при проведении расчётов 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.

Целью данной статьи является попытка оценить влияние на скорость расчётов 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 их подбора путём использования 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]. В ней компилятором автоматически подбираются такие параметры, как, например, развёртка 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 тестирование проводилось для разных типов задач, в том числе на задаче свёртки двумерного и трёхмерного 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 расчёты до двух раз.

Ещё 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. Достигаемое ускорение оценивалось при использовании трёх GPU из двух разных поколений и составило несколько десятков процентов. Стоит отметить, что в качестве одной из модельных задач в данной работе была выбрана операция scan из стандартной библиотеки thrust, которую удалось ускорить таким образом на 13%. Другой интересной особенностью данной работы является малое время обучения — порядка 10-20 итераций, что удалось достичь за счёт учёта специфики оптимизируемых параметров.

При этом стоит 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 счёт подстройки алгоритма под целевую платформу, и за счёт схожих адаптаций размеров ряда блоков удалось заметно ускорить данную операцию для матриц с размером порядка 6144x6144.

Работы в данном направлении, естественно, не ограничиваются вышеперечисленными. Однако все они имеют схожие особенности — (1) за счёт учёта специфики обрабатываемых данных и аппаратной платформы предпринимается попытка повысить скорость расчётов, и (2) данный вид оптимизации должен осуществляться полностью или частично автоматически, не требуя от программиста специальных знаний об 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 трёхмерный массив. В частности, это позволяет для каждого элемента получить всех его соседей просто путём смещения в данных 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 также позволять получать соседей произвольного элемента путём 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:

– Количество элементов, обрабатываемых одной нитью. Очевидно, что идеология GPGPU предполагает, что количество нитей вычислений должно быть крайне большим (обычно рекомендуется создавать в 8-16 раз больше нитей, чем количество ядер GPU). При работе же с большими сетками с некоторого момента выигрыш от подобного увеличения числа нитей становится незначительным, однако благодаря их «утяжелению» иногда удаётся 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.

– Использование разделяемой памяти. Данный тип памяти является более быстрым, поэтому при работе с двумерными и трёхмерными 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 как раз тип реализации доступа к глобальной памяти.

– Выбор типа вычислителя. В случае если расчёты 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 расчёты на всех ядрах CPU, а также встроенном iGPU (при его наличии).

– Разбиение области на блоки. Другой достаточно важный момент заключается в разбиении всей расчётной области на двумерные прямоугольные блоки, каждый из которых будет обработан одним потоковым мультипроцессором GPU. Проблемой является тот факт, что при выборе размеров данного 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 также повысить скорость расчётов. Параметрами в данном случае являются ширина и высота блока.

– Задание размера теневых граней. При проведении расчётов 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 удаётся получать путём дублирования одних и тех же вычислений на «соседних» GPU, тем самым понизив частоту барьерных синхронизаций. При этом размер подобных теневых граней и является оптимизируемым параметром.

Все вышеперечисленные параметры на первый взгляд кажутся достаточно простыми и очевидными, однако оптимальные для них 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 крайне большим, поэтому их полный перебор с целью определения подходящих значений при проведении реальных расчётов практически неосуществим. И логичным шагом становится как раз автоматизация их подбора с помощью технологий автотюнинга.

4. Результаты тестирования

4.1. Схема тестирования

Для подбора значений перечисленных в предыдущем разделе параметров была использована разработанная авторами технология динамической адаптации программ, реализованная в виде библиотеки TTG Apptimizer. Суть её заключается в использовании модификаций классических методов оптимизации для поиска экстремумов функционала, определённого как время выполнения одной итерации алгоритма в зависимости от

произвольного набора параметров [5]. Ключевым моментом данной технологии являются именно произведённые 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 многое другое.

В качестве тестовых программ были реализованы две описанные в последующих разделах модельные задачи, в рамках которых проводились расчёты на двумерной и трёхмерной сетке. Подбор ряда перечисленных выше параметров осуществлялся для сеток разного размера и на всех последних поколениях GPU от NVidia, начиная от ускорителя GeForce 8800 GTX, и заканчивая GeForce 680 GTX. Ускорители серии Tesla не были выбраны сознательно, так как в тестах использовалась только одинарная точность, а, что более важно, далеко не все GPU от NVidia нашли воплощение в серии Tesla. Поэтому использование массовых видеокарт позволяет более глобально оценить эффект от оптимизации программ подобным образом.

Таким образом, выигрыш от применения технологий автотюнинга оценивался как ускорение, которое удалось получить с использованием предложенной авторами реализации данной идеи, по сравнению с исходной («статической») программой. Сразу стоит отметить, что в нижеприведённых результатах не учитывается время обучения, так как при проведении промышленных расчётов его вклад оказывается крайне малым, в то время как в рассматриваемых тестовых программах из-за небольшого количества итераций (порядка 500-1000) 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 этого избежать, 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:

$$\frac{\partial U}{\partial t}=\frac{\partial^2 U}{\partial x^2}+\frac{\partial^2 U}{\partial y^2} \qquad V^{k+1}_{i,j}=V^k_{i,j}+dt\cdot\left(\frac{V^k_{i-1,j}-2\cdot V^k_{i,j}+V^k_{i+1,j}}{dx^2}+\frac{V^k_{i,j-1}-2\cdot V^k_{i,j}+V^k_{i,j+1}}{dy^2}\right)$$

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

$$0<y<1 \qquad dx=\frac{1}{M},\ dy=\frac{1}{N},\ dt=\frac{T}{K}$$

$$0<t<T \qquad V^0_{i,j}=f(i\cdot dx,\ j\cdot dy)$$

$$U(x,y,t=0)=f(x,y) \qquad V^k_{i,j}|_{i=0,M}=\phi(j\cdot dy,\ k\cdot dt)$$

$$U(x,y,t)|_{x=0,1}=\phi(y,t) \qquad V^k_{i,j}|_{j=0,N}=\theta(i\cdot dx,\ k\cdot dt)$$

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

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, а именно – размеры двумерного блока, которые используются при разбиении области для её последующего распределения по всем потоковым мультипроцессорам GPU, и номер используемого в системе устройства (в данный тестах — либо все ядра CPU, либо один GPU). Реализация для центрального процессора была разработана с использованием расширений SSE и технологии OpenMP, поэтому для небольших сеток она оказывается предпочтительнее, чем реализация для GPU, созданная с использованием технологии NVidia CUDA. Озвученные выше параметры проиллюстрированы на следующей схеме:

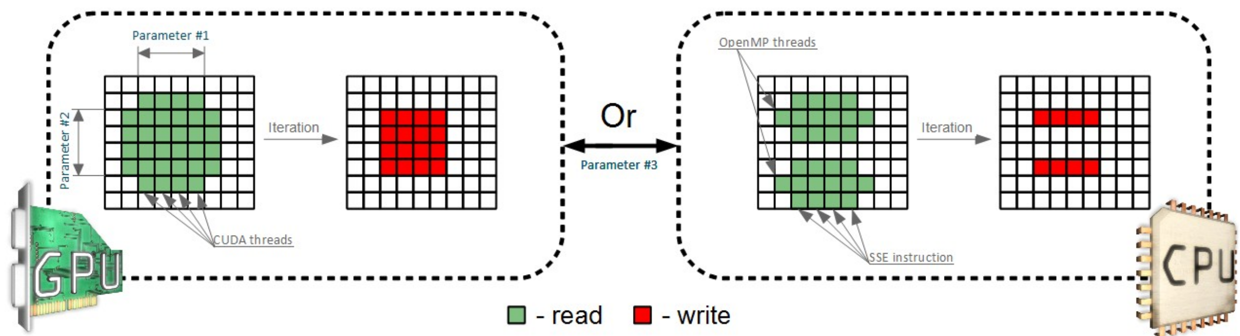

Рис. 1. Введённые параметры в двумерной тестовой задаче

Эффект от применения технологии автотюнинга в рамках рассматриваемой задачи при использовании только одной системы, оснащённой двумя шестиядерными процессорами Intel Xeon 5650 и тремя ускорителями Tesla M2050, представлен на следующем графике:

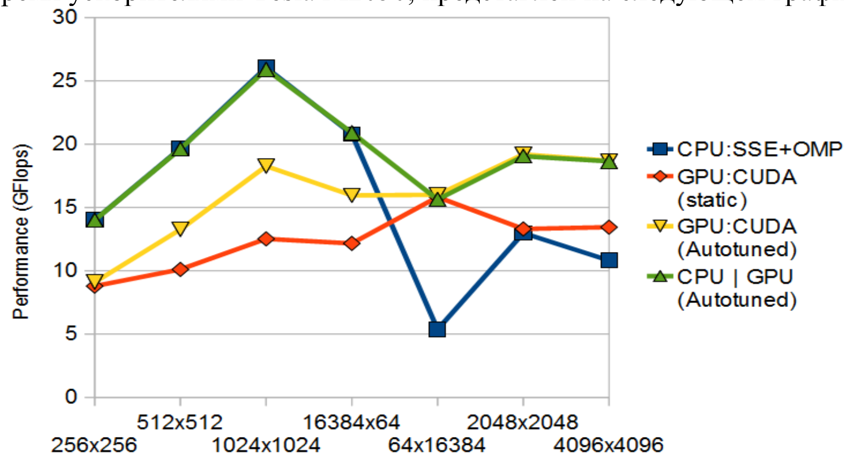

Рис. 2. Производительность разных реализаций алгоритма решения двумерной тестовой задачи

Как видно, на небольших сетках предпочтительнее проводить расчёты на CPU, так как 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 путём подбора всего двух параметров удаётся существенно повысить скорость вычислений, что особенно заметно на больших сетках.

Результаты полного тестирования на различных типах ускорителей и сетках разного размера представлены в следующей таблице:

Таблица 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%

Где сокращения вида GF 9800 обозначают модель графического ускорителя (например, NVidia GeForce 9800 GTX+), а значения производительности определены как два числа —

производительность «статической» версии (в гигафлопсах) и ускорение, достигнутое за счёт автоадаптации программы (в процентах). Стоит отметить, что при тестировании, результаты которого представлены выше, вычисления проводились только на графических ускорителях и без использования центрального процессора.

Как можно заметить, 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 вычислительных ядер требуется иначе проводить разбиение данных на блоки — как с учётом как специфики самого ускорителя, так и размера сетки. Максимальный эффект был зафиксирован на сетках большого размера и GPU поколения Fermi (обозначены как GF 480 и GF 580), на которых программу удалось ускорить в 1.46 раза. Минимальный — на сетке размера 384x384 и ускорителе с архитектурой G92b (GF 9800), когда наблюдалось даже замедление программы на 0.27%. Это объясняется отсутствием в данном тесте возможностей для ускорения программы и влиянием накладных расходов операционной системы и драйвера. Действительно, так как используемая 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 следующей таблице приведены результаты аналогичного тестирования, но в котором технологии автоадаптации разрешалось «перебрасывать» вычисления на центральный процессор Intel Xeon E3-1230.

Таблица 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**
256x256	3,63+127,83%	3,32+156,10%	3,00+193,23%	8,12+7,02%	9,11+0,11%	7,84+21,56%
384x384	3,90+251,26%	3,76+244,85%	3,53+273,19%	9,22+46,75%	11,95+8,76%	10,24+12,79%
512x512	5,07+193,95%	4,83+212,99%	5,17+193,91%	9,49+60,38%	14,05+15,09%	12,49+18,33%
768x768	6,26+172,93%	6,11+177,40%	7,68+117,97%	10,66+61,73%	17,08+28,57%	15,00+39,47%
1024x1024	6,84+142,39%	6,66+162,30%	9,35+82,96%	11,28+49,20%	18,00+27,11%	16,29+36,22%
1536x1536	7,29+23,47%	6,68+42,714%	11,05+37,83%	11,8+42,20%	19,91+27,32%	17,57+33,92%
2048x2048	7,49+27,82%	6,88+35,16%	11,87+42,21%	11,89+42,39%	20,78+25,99%	18,31+31,95%
3072x3072	7,58+26,12%	6,82+38,70%	12,5+45,60%	11,99+43,70%	21,5+21,35%	18,66+32,10%
4096x4096	7,67+22,36%	6,75+44,06%	12,76+46,63%	11,95+46,44%	21,98+18,38%	18,79+30,81%
6144x6144	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%

В соответствии с полученными результатами, устаревшие поколения GPU, основанные на архитектурах G80 и G92b (обозначены как GF 8800 и GF 9800), на данной задаче и всех тестовых сетках оказались менее производительными, чем современный CPU, поэтому и не использовались. Что является более интересным наблюдением — на более новых, но также устаревших поколениях GPU в лице GT200 и Fermi (GF 285, GF 480 и GF 580), выигрыш от переноса вычислений на центральный процессор тоже оказывается заметным, но лишь для сеток небольшого размера. Наконец, в случае использования современной архитектуры GPU Kepler1 (GF 680) 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, у пользователей реальных расчётных программ в рабочих машинах может стоять совершенно произвольный ускоритель, поэтому данный вид оптимизации, а именно динамический выбор наиболее подходящего устройства, может оказаться крайне полезным. Однако его реализация программистом «вручную» обычно является достаточно трудоёмкой, поэтому выбор используемого вычислителя (CPU или GPU) в большинстве случаев ложится исключительно на пользователя, который при запуске программы должен указать соответствующий аргумент. В случае же применения технологий автоадаптации программ эта проблема фактически пропадает, позволяя в ряде случаев получить «бесплатное» ускорение до 3 раз.

Также стоит ещё раз отдельно проиллюстрировать результаты работы рассматриваемой технологии при использовании только одного ускорителя NVidia GeForce 480 GTX, что показано на следующем графике:

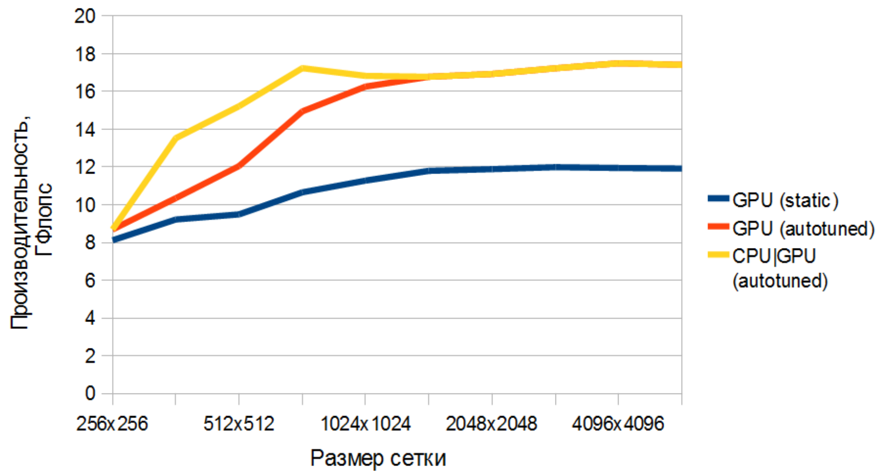

Рис. 3. Результаты тестирования технологии на двумерной задаче и ускорителе GeForce 480 GTX

Таким образом, динамически подстраивая всего три параметра данной тестовой программы, удаётся в среднем её ускорить в 1.45 раза, при этом на проведение подобных видов оптимизации практически не требуются дополнительные затраты со стороны разработчика.

4.3. Тестирование на трёхмерной сетке

В качестве теста для трёхмерного случая было выбрано уравнение Пуассона в постановке задачи Дирихле, решаемое с помощью итерационного метода Якоби. Таким образом, непрерывная задача имела следующий вид:

$$\frac{\partial^2 U}{\partial x^2}+\frac{\partial^2 U}{\partial y^2}+\frac{\partial^2 U}{\partial z^2}=f(x,y,z)$$
$$0<x<1$$
$$0<y<1$$
$$0<z<1$$
$$U(x,y,z)|_{(x,y,z)\in G}=g(x,y,z)$$

А дискретная задача может быть записана как:

$$\frac{V_{i-1,j,l}-2\cdot V_{i,j,l}+V_{i+1,j,l}}{dx^2}+\frac{V_{i,j-1,l}-2\cdot V_{i,j,l}+V_{i,j+1,l}}{dy^2}+\frac{V_{i,j,l-1}-2\cdot V_{i,j,l}+V_{i,j,l+1}}{dz^2}=f(i\cdot dx,j\cdot dy,l\cdot dz)$$
$$i=\overline{1,M-1},\ j=\overline{1,N-1},\ l=\overline{0,K-1}$$
$$dx=\frac{1}{M},\ dy=\frac{1}{N},\ dz=\frac{1}{K}$$
$$V_{i,j,l}|_{i\in\{0,\mathrm{N}\},\ j\in\{0,\mathrm{M}\},\ l\in\{0,\mathrm{K}\}}=g(i\cdot dx,j\cdot dx,l\cdot dz)$$

где M, N и K задают размер накладываемой на область сетки. Так как объём независимых операций крайне большой, то в реализации были введены три основных параметра – первые два определяют размер двумерного CUDA-блока, а третий задаёт 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 параметром. Также был введён дополнительный четвёртый параметр, который задаёт тип вычислительного ядра – с использованием разделяемой памяти или без. Все данные четыре параметра показаны на следующей иллюстрации:

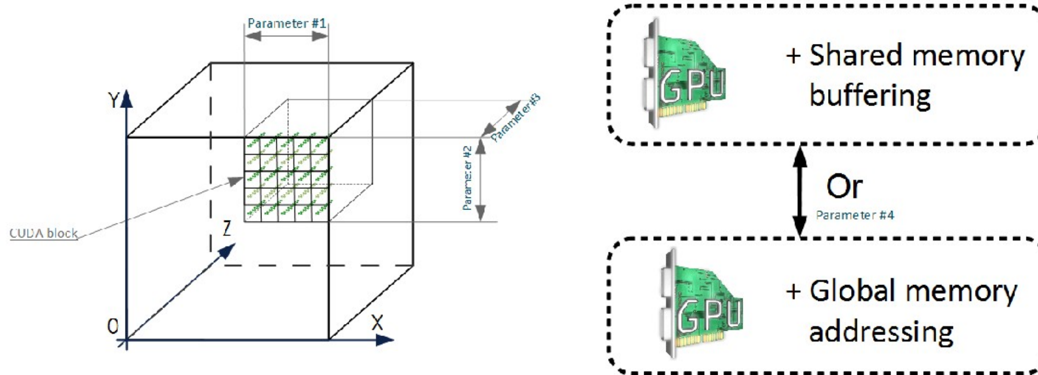

Рис. 4. Введённые параметры в трёхмерной тестовой задаче

Перед тем, как переходить непосредственно к описанию результатов тестирования, следует показать, что оптимальные значения введённых параметров действительно зависят от модели ускорителя, а также существенно влияют на производительность. Для этого ниже приведён график скорости расчётов в зависимости от количества элементов, обрабатываемых одной нитью, при использовании ускорителя NVidia GeForce 285 GTX:

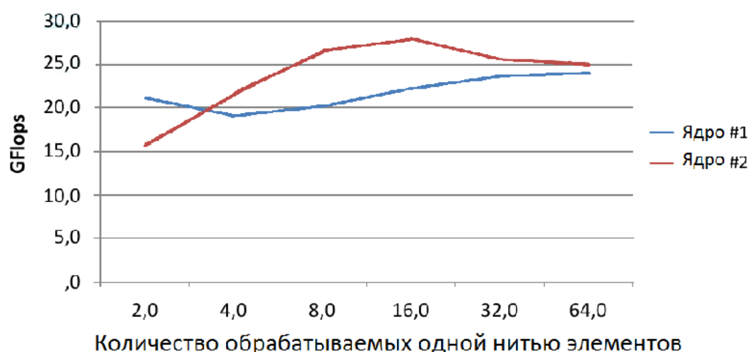

Рис. 5. Зависимость производительности от одного из параметров, ускоритель GeForce 285 GTX

В вычислительном ядре под номером один не используется разделяемая память, поэтому в большинстве случаев оно оказывается не столь эффективным, так как в данном GPU нет аппаратного кэша. При этом эффект от укрупнения нитей даёт положительный, и достаточно заметный эффект — при увеличении количества обрабатываемых элементов с 1 до 16 удаётся получить ускорение до 75%. Таким образом, при ручной оптимизации программистом может быть сделан вывод, что следует всегда использовать реализацию ядра с разделяемой памятью, и одной нитью обрабатывать по 16 элементов. Однако при переходе на более новый ускоритель NVidia GeForce 580 GTX данные выводы оказываются ошибочными, что проиллюстрировано на следующем графике:

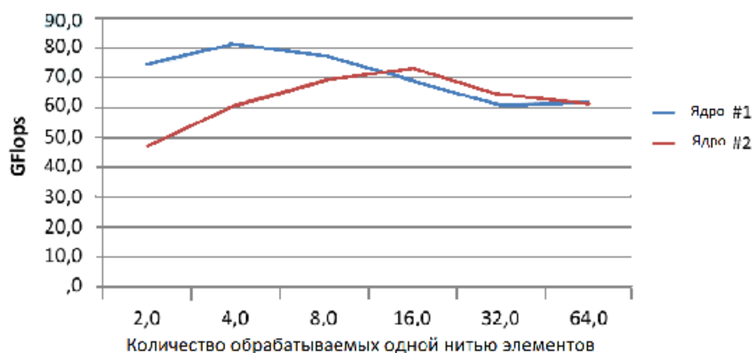

Рис. 6. Зависимость производительности от одного из параметров, ускоритель GeForce 580 GTX

Как легко заметить, из-за смены ускорителя оптимальным вариантом оказался выбор ядра без использования разделяемой памяти, и обработка по 4 элемента одной нитью. Если же в программе всегда использовать значения параметров, полученные исходя из предыдущих наблюдений, то в данном случае она работала бы на 11% медленнее. И это при том, что оба теста проводились только на одной сетке размером 128x128x128 элементов, а варьировался всего один параметр. При реальных же расчётах количество подобных параметров может доходить до десятка, а сетки, естественно, будут иметь совершенно разные размеры, поэтому в «статической» версии программы потери от «неправильного» выбора значений оптимизационных параметров могут оказаться весьма существенными.

Также в аналогии с предыдущим разделом отдельно стоит проиллюстрировать эффект от технологии автоадаптации программ только при использовании одного ускорителя GPU GeForce 680GTX, но для сеток разного размера. Соответствующие результаты приведены на следующем графике:

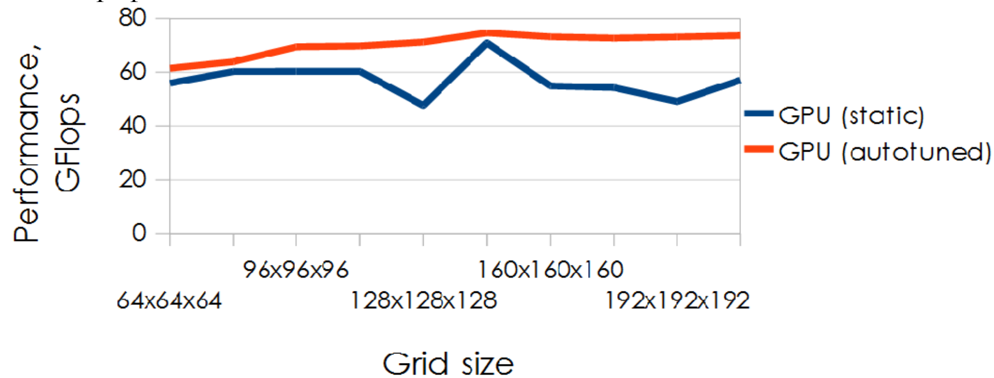

Рис. 7. Результаты тестирования технологии на трёхмерной задаче и ускорителе GeForce 680 GTX

Как видно, в зависимости от размеров сетки подбор более подходящих параметров позволяет получить ускорение до 40%, в то время как на исходной сетке, для которой проводилась ручная оптимизация статической версии, выигрыш практически незаметен, что объясняется удачным подбором начальных значений всех четырёх параметров, а именно разбиением на блоки 16x16x8 и выбором версии без использования разделяемой памяти. Результаты тестирования на всех рассматриваемых видах ускорителей и разных сетках приведены ниже:

Таблица 3. Результаты тестирования технологии на трёхмерной задаче на всех ускорителях

Размер сетки	GF 8800	GF 9800	GF 285	GF 480	GF 580	GF 680
64x64x64	9.7 + 10.1%	11.7 + 0.4%	23.1 + 4.1%	54.2 + 3.0%	61.1 + 17.9%	55.9 + 9.9%
96x96x96	10.1 + 18.5%	12.3 + 9.2%	29.9 + 1.9%	59.0 + 22.9%	69.3 + 25.0%	60.3 + 15.2%
128x128x128	10.2 + 16.1%	9.4 + 30.4%	23.1 + 4.2%	55.2 + 25.4%	63.5 + 28.6%	47.7 + 49.2%
160x160x160	10.3 + 21.2%	11.7 + 9.6%	25.5 + 16.8%	61.5 + 22.7%	71.4 + 33.1%	54.9 + 33.2%
192x192x192	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 и в предыдущем разделе, в таблице 3 приводится достигнутая производительность (в гигафлопсах), а также дополнительное ускорение от применения технологии автоадаптации программ (в процентах). Максимальный эффект был достигнут на последнем поколении GPU Kepler1, представителем которого является ускоритель GeForce 680 GTX — на нём ускорение составило от 10% до 49%. Это объясняется тем, что исходная тестовая программа была разработана задолго до появления данной архитектуры, поэтому заложенные в ней значения параметров даже без привязки к размерам сеток оказались далеко не самыми оптимальными.

5. Заключение

В данной работе на примере предложенной авторами реализации парадигмы автоадаптации GPGPU-программ, известной в англоязычной литературе как autotuning, показана схема оптимизации двух тестовых программ, решающих модельные задачи из области аэрогидродинамики на двумерной и трёхмерной сетках. Благодаря проведённой в соответствии с данной парадигмой полуавтоматической оптимизации удалось достичь ускорения от нескольких десятков процентов до 2-3 раз в зависимости от размера расчётной области и используемого ускорителя. При этом тестирование проводилось для всех современных поколений GPU от NVidia, что позволяет утверждать об основном преимуществе рассматриваемой парадигмы — с её помощью возможно проводить оптимизацию программы даже для таких архитектур, к которым разработчик ещё не имеет доступа. В том числе и из-за того, что соответствующие ускорители ещё не были разработаны.

Если сравнивать полученные результаты с предшествующими работами, то в качестве основного отличия можно назвать попытку более глобально подойти к реализации идеи автоадаптации. В частности, в данной работе подстройка производится не только для одного-двух основных параметров, которые присутствуют в любой GPGPU-программе, а сразу для множества параметров (причём свойственных не только GPU), часть из которых удаётся выделить исключительно благодаря учёту специфики предметной области. Более того, в отличие от работ предшественников, тестирование проводилось на всех поколениях GPU, что позволило более точно оценить достигаемое за счёт подобных оптимизаций ускорение.

В качестве дальнейшей работы, направленной в первую очередь на оценку эффекта от использования подобных технологий для решения проблем из самых разных предметных областей, планируется провести аналогичное тестирование для задач, узким местом в которых является исключительно доступ к памяти. С учётом того, что в последующих поколениях ускорителей планируется добавление поддержки общей памяти, вполне может оказаться, что за счёт аналогичной адаптации нескольких параметров скорость расчётов можно будет существенно повысить.

Литература

1. М.А. Кривов, От G80 и до GK110 // Журнал «Суперкомпьютеры», январь-февраль 2013, с. 48-51.

2. S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, J. Cavazos, Auto-tuning a High-Level Language Targeted to GPU Codes // Innovative Parallel Computing (InPar), 2012, p. 1-10

3. M. Tillmann , T. Karcher , C. Dachsbacher , W.Tichy, Application-independent Autotuning for GPU // ParCo-Symposium: Application Autotuning for HPC (Architectures), 2013.

4. Y. Li, J. Dongarra, S. Tomov, A Note on Auto-tuning GEMM for GPUs // Proceeding ICCS '09 Proceedings of the 9th International Conference on Computational Science: Part I, pp. 884 - 892.

5. М.А. Кривов, М.Н. Притула, С.А. Гризан, П.С. Иванов. Оптимизация приложений для гетерогенных архитектур. Проблемы и варианты решения // Информационные технологии и вычислительные системы, 2012, № 3.