ЧАСТЬ 2

CUDA: Пустим

Константин Калгин и **Евгений Балдин** не могут спокойно видеть матрицу – им непременно надо ее транспонировать.

Наш эксперт

Константин Калгин
Программист, который превращает персоналку в маленький супер-компьютер.

Наш эксперт

Евгений Балдин
Физик, который действительно знает, что такое нехватка вычислительных ресурсов.

Хочется волшебства... вот бы сказать компьютеру: «Посчитай-ка, голубчик, мою задачу, да побыстрее! А я кофе попью...» Сказать-то можно, да современный компьютер сейчас на это не реагирует. Наверное, это неплохо: проснувшись, машинный разум начнет всячески отлынивать от работы. И человеку все равно придется разбираться с аппаратурой.

Версии CUDA

Следует различать версии программных пакетов CUDA Toolkit (компилятор, профилировщик, библиотеки), CUDA SDK (примеры с исходными кодами, вспомогательные библиотеки), CUDA Driver (системный драйвер) и версии CUDA, которую поддерживает графический ускоритель. В документации Nvidia C Programming Guide версия, которую поддерживает графический ускоритель, называется Compute capability. Для простоты назовем это «версией CUDA».

Версия CUDA состоит из двух чисел – старшего и младшего, например 1.3. Графические ускорители с одинаковым старшим числом имеют одну архитектуру ядра. Младшее число указывает на улучшения в архитектуре ядра. Какой версии соответствует ваше устройство, легко узнать из той же Википедии, в статье CUDA.

Архитектура процессора

Процессор графического ускорителя состоит из планировщика блоков потоков, набора мультипроцессоров и кэша L2. Наличие и объем кэша L2 зависит от версии CUDA, а количество мультипроцессоров – еще и от модели графического ускорителя. Меняя количество мультипроцессоров, производители пропорционально меняют потребляемую мощность и производительность графического ускорителя, и непропорционально его цену.

» **Планировщик блоков потоков** При запуске ядра можно отправить на исполнение до 65535 блоков потоков. Планировщик следит за загруженностью мультипроцессоров, где исполняются блоки потоков и по завершении работы одних блоков отправляет еще не отработанные блоки на освободившиеся мультипроцессоры.

» **Мультипроцессор** состоит из набора потоковых процессоров, планировщика потоков, разделяемой памяти, банка регистров, а также текстурного, константного и L1 кэшей.

» **Кэш L2** автоматически кэширует данные при обращении к глобальной памяти, чем ускоряет как последующий повторный доступ, так и доступ к соседним данным. Кэш L2 появился в архитектуре CUDA, начиная с версии 2.x (Fermi).

Планировщик блоков потоков

» **CUDA 1.x** Пока не исполнились все блоки потоков одного ядра, блоки потоков другого будут планироваться. По сути, это не планирование, а простая раздача работы (блоков потоков) в порядке очередности (координат) и монопольное использование ресурсов графического ускорителя одним ядром.

» **CUDA 2.x (Fermi)** Планировщик может планировать блоки потоков от разных ядер одного процесса, повышая эффективность ускорителя на небольших сетках в несколько блоков потоков.

» **CUDA 3.5 (Kepler)** Реализован динамический параллелизм, позволяющий запускать ядра с самого графического ускорителя и синхронизироваться по результату. Вызовы ядер могут быть вложенными, что открывает большие возможности эффективной реализации задач с нерегулярным и динамическим параллелизмом, а также по переводу части кода, управляющего запуском ядер, на графический ускоритель.

Мультипроцессор

Количество и объем тех или иных элементов мультипроцессора зависит от версии CUDA. На одном мультипроцессоре могут планироваться несколько блоков потоков. Минимальной единицей исполнения и планирования на мультипроцессоре является варп [*англ.* warp – *зд.* скрутка] – группа из 32 потоков одного блока. На каждом такте планировщик выбирает группу потоков, и над каждым потоком из группы исполняется одна и та же команда. Корректная обработка условных переходов потоками одного варпа происходит за счет того, что некоторые потоки/потоковые процессоры могут простаивать, то есть не исполнять текущую команду. Итоговое время обработки ветвей условного перехода в случае, когда произошло разделение потоков варпа, складывается из времен исполнения обеих ветвей.

Имеющиеся у мультипроцессора 4-байтовые регистры делятся между планируемыми потоками, а объем разделяемой памяти – между планируемыми блоками потоков.

» **CUDA 1.x** Содержит 8 потоковых процессоров, выполняющих инструкции с целыми числами и числами с плавающей точкой одинарной точности [float]. Вычисления с плавающей запятой с двойной точностью [double] стали доступны в CUDA 1.3 – на одном мультипроцессоре находилось только одно исполнительное устройство, т. е. скорость работы с двойной точностью было в 8 раз меньше скорости работы с одинарной точностью.

» **CUDA 2.x** Содержит 32 (CUDA 2.0) или 48 (CUDA 2.1) потоковых процессоров, выполняющих инструкции с целыми числами и числами с плавающей точкой одинарной точности. Увеличилось количество исполнительных устройств для работы с числами двойной точности – 16 (CUDA 2.0) и 24 (CUDA 2.1). Количество планировщиков варпов – 2. В CUDA 2.1 на каждом такте каждый планировщик выдает по две информационно независимых инструкции одного из варпов, если только ни одна из инструкций не работает с числами двойной точности.

В CUDA 2.x появился «полноценный» кэш L1 и L2 – ранее все кэши были доступны только на чтение, т. е. кэшировали константные данные. У каждого мультипроцессора кэш первого уровня L1 свой, а кэш второго уровня L2 общий для всех мультипроцессоров графического ускорителя. Объемы кэша первого уровня и разделяемой памяти в сумме дают 64 КБ, и могут быть сконфигурированы с помощью cudaFuncSetCacheConfig() в 48 КБ/16 КБ или 16 КБ/48 КБ, соответственно. Объемы кэша L1 и разделяемой памяти по умолчанию – 16 КБ/48 КБ. Объем кэша L2 – 768 КБ.

» **CUDA 3.x** Содержит 192 потоковых процессора, выполняющих инструкции с целыми числами и числами с плавающей точкой одинарной точности. Количество исполнительных устройств для работы с числами двойной точности – 8 (CUDA 3.0) и 64 (CUDA 3.5). Планировщики усовершенствованы таким образом, что

в работу

теперь могут выдавать по две инструкции за такт вне зависимости от типа инструкций. Объемы кэша L1 и разделяемой памяти теперь могут быть сконфигурированы как 32 КБ/32 КБ. Объем и пропускная способность кэша L2 выросли в два раза.

На архитектурах CUDA 1.x и 2.x, чтобы обменяться значениями регистров между потоками одного блока, необходимо было использовать разделяемую память. В архитектуре CUDA 3.x появились инструкции, позволяющие обмениваться значениями регистров между потоками одного варпа без использования разделяемой памяти. За счет этого экономится время обращения и объем используемой разделяемой памяти.

» **Организация памяти** В графическом ускорителе иерархию памяти составляют следующие элементы: файл регистров, разделяемая память, кэши и глобальная память. В официальной документации количественные данные по латентности приводятся только для глобальной памяти. Остальные данные – качественного характера: латентность регистров равна латентности разделяемой памяти, латентность кэшей меньше латентности глобальной памяти, латентность согласованного доступа в разделяемую/глобальную память существенно меньше латентности несогласованного доступа.

» **Локальная память** В архитектуре CUDA имеется аппаратное ограничение на количество используемых регистров одним потоков – 63 (CUDA < 3.0) и 255 (CUDA 3.5). Если во время компиляции компилятору не хватит доступных регистров, он отобразит их на локальную память. Локальная память – это область в глобальной памяти, выделенная компилятором для хранения локальных значений потоков. Она используется для хранения локальных данных потоков при нехватке регистров или объявления локальных массивов внутри ядра без ключевого слова __shared__:

```
__global__ void kernel(int *a, int *s) {
   int l[N], res;
   int ind = (blockIdx.x*blockDim.x+threadIdx.x)*N;
   for( int j=0; j<N; j++) l[ j ] = a[ ind + j ];
      for( int j=0; j<N; j++) res += l[ j ] * j;
         s[ ind ] = res;
}
```

В этом примере при больших N массив l будет располагаться в локальной памяти. У каждого потока будет свой массив. При малых N компилятор может развернуть циклы, после чего отобразить элементы массива на регистры, поскольку отпадет нужда в обращении к элементам по меняющемуся индексу.

» **Доступ в разделяемую память. Конфликты** Вся разделяемая память делится на 16 (CUDA 1.x) или 32 (CUDA 2.x/3.x) банков. Последовательно расположенные 32-битные слова помещаются в последовательных банках [interleaved]. Пропускная способность каждого банка – 32 бита за два такта. Конфликтом называется одновременное обращение потоков к разным 32-битным словам одного банка. Конфликтные обращения в банк исполняются последовательно. Обращение к разделяемой памяти называется согласованным если отсутствуют конфликты.

› **CUDA 1.x** Инструкция обращения к разделяемой памяти исполняется за два шага – по половинам варпа. Конфликты могут возникнуть только внутри каждой из половин варпа.

› **CUDA 2.x/3.x** Конфликты могут возникнуть в рамках варпа в целом.

» Доступ в глобальную память

› **CUDA 1.0/1.1** Доступ в глобальную память является согласованным, если для каждой половины варпа выполняются следующие условия:

1. Размер слов, к которым обращается каждый поток, равен 4, 8 или 16 байтам.
2. Если размер равен N, то все 16 слов лежат в 16×N-байтном сегменте.
3. Потоки обращаются к словам последовательно: k-й поток в половине варпа обращается к k-му слову в сегменте.

Второе условие для прикладного программиста переформулируется следующим образом: при обращении к элементам массива первый поток каждой половины варпа должен обращаться к элементу, номер которого кратен 16. При согласованном обращении к 4/8/16-байтным словам для каждой половины варпа выполняется одна 64-байтная/одна 128-байтная/две 128-байтных транзакции. Невыполнение инструкции обращения к глобальной памяти некоторыми потоками за счет ранее исполненного условного ветвления не влияет на согласованность.

В случае невыполнения условий согласованного доступа обращение разбивается на 16 отдельных 32-байтных транзакций.

› **CUDA 1.2/1.3** Для этой и последующих архитектур не используется термин согласованного обращения, но описывается алгоритм определения количества и размер транзакций с глобальной памятью. Для CUDA 1.2/1.3 количество и размер транзакций определяется следующим образом.

1. Для каждой половины варпа берется минимальное количество сегментов, которые покрывают все запрашиваемые элементы этой половиной. Размер каждого сегмента равен 32 байтам для 1 байтных данных, 64 байтам для 2 байтных, 128 байтам для 4/8/16 байтных данных.
2. Каждый загружаемый сегмент уменьшается по правилам:

» Если запрашиваемые данные лежат только в левой или правой половине 64/128-байтного сегмента, то сегмент уменьшается до соответствующей 32/64-байтной половины;

» Если запрашиваемые данные лежат только в одной из четвертей 128-байтного сегмента, то сегмент уменьшается до соответствующей 32-байтной части;

› **CUDA 2.x/3.x** Все обращения в глобальную память кэшируются; где именно, определяет программист во время компиляции через флаги: в кэшах L1 и L2 (**-Xptxas -dlcm=ca**, по умолчанию) или только в кэше L2 (**-Xptxas -dlcm=cg**). Вариант кэширования определяет размер транзакций с памятью – 128-байтные для первого случая и 32-байтные для второго. Таким образом, кэширование только в L2 может сократить время обращения в глобальную память в случае, когда потоки одного варпа обращаются к разбросанным [scattered] данным.

» **Единое адресное пространство** Начиная с CUDA 2.x, реализовано единое адресное пространство. Т.е. множество адресов поделено на участки, соответствующие локальной, разделяемой и глобальной памятям. Это существенно упрощает программирование алгоритмов с адресацией, зависящей от данных.

Транспонирование матриц

Для демонстрации некоторых из особенностей архитектуры решим простую модельную задачу: транспонируем матрицу. На вход »

программе подается матрица A размером N×N. На выходе необходимо получить матрицу B, такую, что Bi,j = Aj,i.

Тестирование производительности предложенных алгоритмов будет проводиться на графических ускорителях Nvidia GeForce GTS 8800 (CUDA 1.1, 128 ядер, 512 МБ) и Nvidia Quadro FX 480 (CUDA 1.3, 192 ядра, 1536 МБ).

Для сравнения приведем листинг последовательной реализации транспонирования:

```
void transpose_host(float *a, float *b, int N) {
  for( int i=0; i<N; i++) {
    for( int j=0; j<N; j++) {
      b[ j*N + i ] = a[ i*N + j ];
    }
  }
}
```

Вариант 1

Число порождаемых вычислительных потоков равно числу элементов матрицы. Поток с координатами (i,j):

```
i = threadIdx.x+blockIdx.x * blockDim.x
j = threadIdx.y+blockIdx.y * blockDim.y
```

копирует значение элемента (i,j) из матрицы A в элемент (j,i) матрицы B. При таком подходе потоки одного варпа читают значения из max(32/blockDim.x,1) строк матрицы, что приводит к такому же числу транзакций с памятью на чтение. При этом потоки одного варпа записывают эти значения в min(blockDim.x,32) строк, что приводит к такому же числу транзакций с памятью на запись. Таким образом, при увеличении blockDim.x уменьшается число транзакций на чтение, но увеличивается число транзакций на запись. При уменьшении – наоборот. Полный листинг программы:

```
int N; // matrix size NxN
int BSX, BSY;
__global__ void transpose_1(float* A, float* B, int N) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int j = threadIdx.y + blockIdx.y * blockDim.y;
  B[ j * N + i ] = A[ i * N + j ];
}
float *host_a, *host_b;
float *dev_a,  *dev_b;
int main(){
  N = 1024;
  BSX = BSY = 16;
  host_a = (float*)malloc(sizeof(float)*N*N);
  host_b = (float*)malloc(sizeof(float)*N*N);
  cudaMalloc(&dev_a, sizeof(float)*N*N);
  cudaMalloc(&dev_b, sizeof(float)*N*N);
  cudaMemcpy(dev_a, host_a, sizeof(float)*N*N,
cudaMemcpyHostToDevice);
  dim3 gdim = dim3(N/BSX,N/BSY,1);
  dim3 bdim = dim3(BSX,BSY,1);
  transpose_1 <<< gdim, bdim>>> (dev_a, dev_b, N);
  cudaDeviceSynchronize();
  cudaMemcpy(host_b, dev_b, sizeof(float)*N*N,
cudaMemcpyDeviceToHost);
}
```

Здесь в строках описано ядро transpose_1, транспонирующее матрицу A, результат записывается в матрицу B.

Далее определяются указатели на матрицы host_a и host_b, которые располагаются в оперативной памяти компьютера, и dev_a, dev_b – в глобальной памяти графического ускорителя. Сами указатели будут храниться в оперативной памяти, поскольку они инициализируются в основной программе и в их определении отсутствуют __device__ и __constant__. После этого выделяется область в оперативной (malloc) и глобальной (cudaMalloc) памяти.

Для вызова ядра transpose_1 необходимо определить две переменные структурного типа dim3, содержащие в себе размеры блока потоков и массива блоков потоков. Количество порождаемых потоков равно N2 = (N/BSX)*(N/BSY)*BSX*BSY. Ядру в качестве аргументов передаются указатели на матрицы и линейный размер самих матриц. После вызова ядра основная программа дожидается завершения его исполнения (cudaDeviceSynchronize), чтобы убедиться, что значения всех потоков записаны в память, и копирует данные из глобальной памяти (cudaMemcpyDeviceToHost).

Вариант 2

Возьмем то же число порождаемых потоков и их отображение на элементы, как в Варианте 1. Потоки одного блока работают со следующими элементами матрицы A: {(x+tx,y+ty)| x =

Влияние размера блока потоков на время работы

GTX 8800	1	2	4	8	16	32	64	128	256
1							3.20	3.20	3.13
2						3.15	3.18	3.18	3.13
4					3.20	3.18	3.20	3.22	
8				3.55	3.18	3.20	3.26		
16			3.25	3.50	3.16	3.28			
32		3.25	3.24	3.41	3.15				
64	1.36	3.26	3.20	3.50					
128	1.81	2.68	2.90						
256	2.73	3.21							

Среднее время работы ядра transpose_1, в мс, на матрице 1024×1024 на графическом ускорителе Nvidia GTS 8800 для различных размеров блока потоков. По оси абсцисс — blockDim.x, по оси ординат — blockDim.y. Множество Мандельброта. Результат исполнения примера Mandelbrot из стандартного набора CUDA.

Quadro	1	2	4	8	16	32	64	128	256
1							2.75	2.73	2.60
2						2.78	2.75	2.73	2.60
4					1.94	2.76	2.75	2.75	
8				0.99	1.90	2.73	2.75		
16			0.60	1.02	1.87	2.70			
32		0.51	0.66	0.99	1.79				
64	0.82	0.56	0.64	0.95					
128	0.96	0.56	0.45						
256	1.32	0.72							

Среднее время работы ядра transpose_1 в мс на матрице 1024×1024 на графическом ускорителе Nvidia Quadro для различных размеров блока потоков. По оси абсцисс — blockDim.x, по оси ординат — blockDim.y.

В таблицах представлены средние времена работы ядра при различных размерах блока потоков. По результатам тестирования видно, что время работы ядра существенно зависит от размера блока потоков. Цветом выделены наилучшие результаты по таблице. Объяснить, почему именно эти размеры привели к наименьшим временам, трудно, поскольку в официальной документации открыта не вся информация об устройстве графических ускорителей. Можно лишь объяснить общую тенденцию: наименьшие результаты достигаются при blockDim.y > blockDim.x. Это связано с тем, что транзакция на запись исполняется дольше транзакции на чтение. В данном варианте при уменьшении blockDim.x уменьшается число транзакций на запись: для CUDA 1.3 число транзакций равно min(blockDim.x,32), для CUDA 1.1 – если blockDim.x = 1, то запись согласованная, иначе запись несогласованная.

blockDim.x*blockIdx.x, y = blockDim.y*blockIdx.y, 0 ≤ tx < blockDim.x, 0 ≤ ty < blockDim.y). Таким образом, потоки одного блока транспонируют подматрицу размера blockDim.x*blockDim.y и результат записывают в матрицу B, начиная с элемента (y,x). Подматрица достаточно небольшая и может быть размещена в разделяемой памяти, где стадия транспонирования будет исполняться быстрее. Там мы и будем проводить транспонирование подматрицы:

1 Загружаемый из глобальной памяти элемент (i,j) = (x+threadIdx.x, y+threadIdx.y) матрицы A записывается в элемент (threadIdx.x, threadIdx.y) дополнительного массива sh, расположенного в разделяемой памяти.

2 Происходит барьерная синхронизация потоков в блоке, чтобы быть уверенным, что все потоки загрузили свое значение в массив sh.

3 Поток (i,j) записывает значение элемента (threadIdx.y,threadIdx.x) массива sh в элемент (y+threadIdx.x,x+threadIdx.y) массива B.

Данный способ будет работать только с квадратными блоками потоков, например, 16×16 или 32×32. Вот новый листинг ядра:

```
__global__ void transpose_2(float* a, float* b, int N) {
  __shared__ float sh[ BSY ][ BSX ];
  int x = blockIdx.x * blockDim.x;
  int y = blockIdx.y * blockDim.y;
  int i = x + threadIdx.x;
  int j = y + threadIdx.y;
  sh[ threadIdx.y ][ threadIdx.x ] = a[ (y+threadIdx.y) * N +
(x+threadIdx.x) ];
  __syncthreads();
  b[ (x+threadIdx.y) * N + (y+threadIdx.x) ] = sh[ threadIdx.x ][
threadIdx.y ];
}
```

Вариант 3

За счет использования разделяемой памяти получилось минимизировать число транзакций с глобальной памятью. Однако в данной реализации есть небольшой недостаток: в последней строчке во время чтения из массива sh на всех версиях CUDA образуются конфликты банков памяти. Рассмотрим вариант с размером блока потоков 16×16 и 16 банками в разделяемой памяти (CUDA 1.x). Адрес элемента sh[threadIdx.x][threadIdx.y] равен (void*) sh+4*(threadIdx.x*16 + threadIdx.y), следовательно, элемент sh[threadIdx.x][threadIdx.y] размещается в банке, номер которого равен остатку от деления (threadIdx.x*16+threadIdx.y) на 16. В данном случае номер будет равен threadIdx.y. Поэтому все потоки каждой из половин варпа будут обращаться к одному и тому же банку. Такое обращение приведет к увеличению времени исполнения соответствующей инструкции в 16 раз. Чтобы избежать конфликтов по банкам, в этой реализации достаточно увеличить размер строк до 17 элементов:

```
__global__ void transpose_4(float* a, float* b, int N) {
  __shared__ sh[ BSY ][ BSX+1 ];
    ...
```

Это обеспечит отсутствие конфликтов, поскольку адреса элементов sh[threadIdx.x][threadIdx.y] будут равны (threadIdx.x*17 + threadIdx.y)%16, и для различных threadIdx.x из (0,1, ...,15) эти номера будут различаться.

	Время, мс		
	GTX 8800	Quadro	Core i7
transpose_1	1.36	0.45	
transpose_2	0.44	0.28	9.0
transpose_3	0.43	0.26	

Среднее время работы в мс различных вариантов реализации задачи транспонирования матрицы. Размер матрицы — 1024×1024.

	GTX 8800	Quadro
transpose_1	6.6	20
transpose_2	20	32
transpose_3	21	34

Ускорение различных вариантов реализации задачи транспонирования матрицы по отношению к последовательной реализации на процессоре Core i7. Размер матрицы равен 1024×1024.

Результаты

По результатам тестирования видно, что применение разделяемой памяти существенно уменьшает время работы ядра, т. к. минимизируется количество транзакций с памятью – как на чтение, так и на запись. Вариант 3, устранив конфликты по банкам, дает еще несколько процентов выигрыша по времени работы.

Эффективное программирование

В заключение перечислим по отдельности упомянутые ранее факторы, существенно влияющие на производительность.

Количество вычислительных потоков От размера сетки блоков потоков и самого блока потоков зависит степень загруженности планировщиков графического ускорителя. Чем больше варпов планирует планировщик, тем больше у него возможностей скрыть задержки, связанные с обращением в глобальную память. Кроме того, достаточно большие размеры сетки и блока обеспечат эффективное масштабирование на новых графических ускорителях без переписывания и перекомпилирования программы. Обычно рекомендуется порождать порядка 105 потоков за один запуск ядра.

Равномерность загрузки вычислительных потоков Неравномерность таковой – очень частое явление; например, вычисления значений во внутренних и граничных узлах сетки явной разностной схемы обычно различаются. Однако неравномерность загрузки потоков может привести к существенной деградации производительности. Во-первых, это связано с тем, что блок потоков завершается и освобождает мультипроцессор, когда все его потоки завершили исполнение. Во-вторых, запуск ядра завершается, когда все блоки потоков завершили исполнение.

Преобладание вычислений по отношению к загрузкам данных Для повышения производительности необходимо максимизировать количество производимых вычислений на единицу загружаемых данных из глобальной памяти. Минимизировать количество доступов в память помогает использование разделяемой памяти и кэшей, за счет переиспользования одним и тем же или другими потоками близко расположенных данных.

Локальность загружаемых данных Этот фактор напрямую следует из предыдущего. Увеличение локальности загружаемых данных на уровне варпа позволяет уменьшить количество и объем транзакций с памятью и повысить эффективность работы кэшей. Увеличение локальности загружаемых данных на уровне блока потоков позволяет увеличить эффективность использования разделяемой памяти и работы кэшей.

Деление варпов на условных переходах Когда разные потоки одного варпа разбиваются по разным ветвям условного перехода, время исполнения условного перехода складывается из времен исполнения его ветвей. Т. е. частое деление варпов по ветвям приводит к деградации производительности; ее степень зависит от числа делений варпов и от размера ветвей перехода. LXF

Обратная связь

Приглашаем высказаться потенциальных авторов статей по параллельным вычислениям – ценные предложения, критику и советы присылайте по электронной почте: kalgin@ssd.sscc.ru, E.M.Baldin@inp.nsk.su.