УДК 681.3
Э.И. Ватутин, канд. техн. наук, доцент, кафедра вычислительной техники, ЮЗГУ (e-mail: evatutin@rambler.ru)
И.А. Мартынов, аспирант кафедры вычислительной техники, ЮЗГУ (e-mail: morzee@inbox.ru)
В.С. Титов, д-р техн. наук, профессор, зав. кафедрой вычислительной техники, ЮЗГУ (e-mail: titov-kstu@rambler.ru)

ОЦЕНКА РЕАЛЬНОЙ ПРОИЗВОДИТЕЛЬНОСТИ СОВРЕМЕННЫХ ВИДЕОКАРТ С ПОДДЕРЖКОЙ ТЕХНОЛОГИИ CUDA В ЗАДАЧЕ УМНОЖЕНИЯ МАТРИЦ

Приведено описание подходов к выполнению операции умножения матриц на видеокартах с поддержкой технологии CUDA, показано, что «наивный» подход без оптимизации работы с глобальной памятью GPU характеризуется низкой производительностью обработки, в то время как блочное умножение позволяют более эффективно использовать разделяемую память, что в совокупности с применением раскрутки циклов, использованием конфигурации запуска CUDA-ядра с максимальным числом потоков в блоке и coalesced-доступом в глобальную память обеспечивает реальную производительность на уровне 100–300 GFLOP/s для большинства современных видеокарт NVidia.

Ключевые слова: умножение матриц, алгоритмическая оптимизация, CUDA

***

Одной из задач, имеющей важное значение для ряда научно-технических направлений (томография, компьютерная графика, проектирование роботизированных средств, классификация бинарных отношений [1] и др.), является задача умножения матриц. Время ее решения во многих случаях является бутылочным горлышком (англ. bottleneck), поэтому существует большое количество различных подходов, связанных с оптимизацией и распараллеливанием выполняемых действий. Существуют целый спектр программно-алгоритмического обеспечения для выполнения действий над матрицами в ряде частных случаев (например, для разряженных или ленточных матриц), а также базирующееся на нем аппаратно-алгоритмическое обеспечение (например, с использованием транспьютерных сетей, систолических или реконфигурируемых вычислительных структур). Несмотря на кажущуюся простоту и тривиальность решения, рассматриваемая задача характеризуется рядом особенностей, к которым в первую очередь можно отнести высокую степень параллелизма и однородности выполняемых операций, а также сильную зависимость времени вычисления от темпа поступления данных из памяти. В работе [2] описан ряд оптимизаций, оказывающих значительное влияние на время выполнения умножения для однопоточной программной реализации путем алгоритмической оптимизации, направленной на повышение эффективности работы кэш-памяти. В данной статье рассмотрены вопросы анализа эффективности различных подходов к решению задачи общего вида (умножение «плотных» квадратных матриц размера $N \times N$) с использованием видеокарт с поддержкой технологии CUDA [3] в рамках концепции GPGPU.

Учитывая специфику работы GPU, общая стратегия выполнения операции умножения матриц $C = A \times B$ сводится к выполнению следующих шагов:

1. Передача исходных матриц A и B из оперативной памяти в глобальную память GPU.
2. Выполнение умножения.
3. Передача результирующей матрицы C из глобальной памяти GPU в оперативную память.

В некоторых частных случаях действия 1 и 3 могут быть исключены из рассмотрения при условии, что остальные операции с матрицами, предшествующие и производящиеся после умножения, также выполняются на GPU, однако в общем случае необходимо учитывать время загрузки исходных данных и время возврата результата, для чего определяющим фактором является пропускная способность интерфейса PCI Express. Следуя работе [2], производительность обработки будем оценивать по формуле $P = \frac{V}{t}$, где $V = 2N^3$ – выполняемый объем вычислений, $t = t_1 + t_2 + t_3$ – суммарное время, включающее обмен данными и выполнения операции умножения матриц, t_i, $i = \overline{1,3}$ – время выполнения соответствующего шага.

Для параллельного выполнения умножения будем использовать понятие CUDA-ядра (англ. CUDA Kernel), запускаемого в соответствии с SIMD-принципом с различными начальными значениями – координатами блока в сетке и нити (потока) в блоке [3] в соответствии с заданной конфигурацией запуска. В данной задаче удобно использовать двумерные координаты блоков и нитей ввиду обработки данных, представленных двумерными массивами.

Наиболее простым вариантом умножения является широко известный параллельный вариант реализации умножения «в лоб» с использованием трех циклов, два (внешние) из которых фактически выполняются путем логической привязки соответствующих нитей к двумерным блокам, а блоков в свою очередь – к двумерной сетке.

```
__global__ void MatrixMulKernel(float *A, float *B, float *C, int n)
{
      int i = blockIdx.x * blockDim.x + threadIdx.x;
      int j = blockIdx.y * blockDim.y + threadIdx.y;

      float s = 0.0f;

      for (int k=0; k<n; k++)
            s += A[i*n+k]*B[k*n+j];

      C[i*n+j] = s;
}
```

Данное CUDA-ядро может быть выполнено в различной конфигурации запуска с различным числом потоков в блоке, которое определяется аппаратными ограничениями и получаемой скоростью обработки. Во всех приводимых ниже результатах тестов используется максимально возможное число потоков в блоке (обычно в конфигурации 32×32), если специально не оговорено иное:

```
const int BLOCK_SIZE = 32;
MatrixMulKernel_float<<<
    dim3(n/BLOCK_SIZE, n/BLOCK_SIZE),
    dim3(BLOCK_SIZE, BLOCK_SIZE)
>>>(A_gpu, B_gpu, C_gpu, n);
```

В таблицах 1 и 2 приведены результаты измерения производительности обработки на GPU с использованием двух машин (CPU Intel Core 2 Duo E6300 (Allendale) + GPU NVidia GeForce 450 GTS (Fermi), CPU Intel i7 4770 (Haswell) + GPU NVidia GeForce 770 GTX (Kepler)) и их сопоставление с соответствующими результатами однопоточной обработки на CPU с оптимизацией работы кэш-памяти [2].

Таблица 1. Результаты сопоставления производительности обработки на CPU и GPU для вариантов реализации «в лоб», CPU E6300 + GPU GTS 450

N	t_{CPU}, P	t_{GPU}, P	Выигрыш
256×256 (2×256 КБ)	0,135 с 0,25 GFLOP/s	0,140 с 0,24 GFLOP/s	~ 1x
512×512 (2×1 МБ)	1,1 с 0,24 GFLOP/s	0,19 с 1,41 GFLOP/s	6x
1024×1024 (2×4 МБ)	40,8 с 0,05 GFLOP/s	0,57 с 3,77 GFLOP/s	72x
2048×2048 (2×16 МБ)	340 с (5 мин 40 с) 0,05 GFLOP/s	3,7 с 4,64 GFLOP/s	92x

Таблица 2. Результаты сопоставления производительности обработки на CPU и GPU для вариантов реализации «в лоб», CPU i7 4770 + GPU GTX 770

N	t_{CPU}, P	t_{GPU}, P	Выигрыш
256×256 (2×256 КБ)	0,045 с 0,75 GFLOP/s	0,055 с 0,61 GFLOP/s	Проигрыш 18%
512×512 (2×1 МБ)	0,41 с 0,66 GFLOP/s	0,075 с 3,58 GFLOP/s	6x
1024×1024 (2×4 МБ)	3,3 с 0,65 GFLOP/s	0,21 с 10,2 GFLOP/s	16x
2048×2048 (2×16 МБ)	103 с (1 мин 43 с) 0,17 GFLOP/s	1,3 с 13,2 GFLOP/s	79x

Полученные результаты позволяют сделать вывод о том, что при обработке больших матриц ее целесообразно производить на GPU, при этом наблюдается выигрыш от единиц до десятков раз, в особенности в случае, когда размер обрабатываемых данных превышает размер кэш-памяти процессора.

Одним из вариантов оптимизации приведенного выше решения является использование буферизации обрабатываемого j-го столбца матрицы B, что при обработке на CPU позволяет добиться более эффективного использования подсистемы кэш-памяти [2]. При параллельной обработке на GPU данного ограничения нет, однако вызывает интерес исследование влияния данной алгоритмической оптимизации на время выполнения умножения на GPU, результаты чего приведены ниже.

CUDA-ядро:

```
__global__ void MatrixMulKernel_buffered(float *A, float *B, float *C, int n)
{
	int j = blockIdx.x;
	int i = threadIdx.x;

	__shared__ float t[N];

	t[i] = B[i*n+j];        // None coalesced доступ к глобальной памяти!!!

	__syncthreads();

	float s = 0.0f;

	for (int k = 0; k < n; k++)
		s += A[i*n+k]*t[k];

	C[i*n+j] = s;
}
```

Конфигурация запуска CUDA-ядра отличается от предыдущей и подразумевает наличие N параллельно работающих потоков, каждый из которых производит копирование в разделяемую память одного из элементов кэшируемого столбца, барьерную синхронизацию с последующим выполнением умножения:

```
MatrixMulKernel_buffered<<<dim3(n), dim3(n)>>>(A_gpu, B_gpu, C_gpu, n);
```

Таблица 3. Результаты буферизованного умножения с кэширование столбца на GPU, CPU i7 4770 + GPU GTX 770

N	GPU «в лоб»	GPU с буферизацией столбца	Выигрыш
256×256	0,055 с	0,054 с	1x
512×512	0,075 с	0,104 с	проигрыш 1,4x
1024×1024	0,21 с	0,777 с	проигрыш 3,7x

Вместо ожидаемого выигрыша наблюдается проигрыш, виной чему является none coalesced доступ к глобальной памяти GPU [3], который сводит на нет преимущество от уменьшения числа обращений к глобальной памяти. С целью обхода данного ограничения возможна разработка еще одного CUDA-ядра, во многом схожего с предыдущим и отличающимся тем, что буферизации подвергается не j-й столбец матрицы B, а i-я строка матрицы A.

```
__global__ void MatrixMulKernel_buff_row(float *A, float *B, float *C, int n)
{
	int i = blockIdx.x;
	int j = threadIdx.x;

	__shared__ float t[N];

	t[j] = A[i*n+j];        // Coalesced доступ!

	__syncthreads();

	float s = 0.0f;

	for (int k = 0; k < n; k++)
		s += t[k]*B[k*n+j];

	C[i*n+j] = s;
}
```

Результаты использования данного CUDA-ядра приведены в табл. 4.

Таблица 4. Время выполнения буферизованного умножения с кэшированием строки/столбца матрицы на GPU, CPU i7 4770 + GPU GTX 770

N	GPU «в лоб»	GPU с буферизацией столбца	GPU с буферизацией строки	Выигрыш
256×256	0,055 с 0,61 GFLOP/s	0,054 с 0,62 GFLOP/s	0,055 с 0,61 GFLOP/s	1x
512×512	0,075 с 3,58 GFLOP/s	0,104 с 2,58 GFLOP/s	0,061 с 4,40 GFLOP/s	+23%
1024×1024	0,21 с 10,2 GFLOP/s	0,777 с 2,76 GFLOP/s	0,12 с 17,9 GFLOP/s	1,75x

Данный вариант характеризуется coalesced-доступом в глобальную память при буферизации строки и имеет выигрыш во времени обработки по сравнению с двумя предыдущими.

Произведем теоретический анализ данного подхода с целью определения максимального теоретического выигрыша. Каждый поток в данной реализации требует 1 обращение к глобальной памяти (латентность t_G) при кэшировании элемента строки/столбца и *N* обращений при вычислении произведения в цикле, а также *N* обращений к разделяемой памяти (латентность t_S): $(1+N)t_G + Nt_S$ с (обращения на запись не рассматриваются). В используемой конфигурации запуска ядра присутствует *N* блоков, в каждом из которых запускается *N* потоков, соответственно общее время обработки составляет $t_{buf} = ((1+N)t_G + Nt_S)N^2 = N^3t_G + N^2t_G + N^3t_S$ с. С учетом того, что исходный алгоритм умножения «в лоб» характеризуется временем обработки $2N^3t_G$ с, выигрыш во времени обработки

$$\eta = \frac{2N^3 t_G}{N^3 t_G + N^2 t_G + N^3 t_S} < \frac{2N^3 t_G}{N^3 t_G} = 2$$

теоретически лимитирован двукратным значением (с точностью до констант, характеризующих дополнительные детали обработки и соответствующие дополнительные затраты вычислительного времени GPU), что соотносится с полученными экспериментальными данными.

Дополнительно повысить локальность обрабатываемых данных можно с использованием блочного умножения [2], код соответствующего CUDA-ядра приведен ниже.

```
__global__ void MatrixMulKernel_blocks(float *A, float *B, float *C, int n)
{
	int bx = blockIdx.x;
	int by = blockIdx.y;

	int tx = threadIdx.y;
	int ty = threadIdx.x;

	int x0 = bx*BS;
	int y0 = by*BS;

	int zb = 0;

	int ia = (x0+tx)*n + (0+ty);
	int a_step = BS;

	int ib = (0+tx)*n + (y0+ty);
	int b_step = BS*n;

	float sum = 0.0f;

	for (int z = 0; z < n/BS; z++, zb += BS, ia += a_step, ib += b_step)
	{
		__shared__ float ta[BS][BS];
		__shared__ float tb[BS][BS];

		ta[tx][ty] = A[ia];
		tb[tx][ty] = B[ib];

		__syncthreads();

		for (int k = 0; k < BS; k++)
			sum += ta[tx][k]*tb[k][ty];

		__syncthreads();
	}

	C[(x0+tx)*n + (y0+ty)] = sum;
}
```

Приведенное CUDA-ядро запускается в следующей конфигурации:

```
MatrixMulKernel_blocks<<<
    dim3(n/BS, n/BS),
    dim3(BS, BS)
>>>(A_gpu, B_gpu, C_gpu, n);
```

Данный код, также как и рассмотренное выше буферизованное умножение, может быть реализован с различным порядком загрузки элементов подматриц (по строкам и по столбцам), что приводит к различию во времени обработки около 2 раз (выше приведен код быстрого варианта с coalesced доступом в глобальную память). Для CPU-реализации была выявлена [2] сильная зависимость времени обработки от размера блока, являющаяся следствием работы подсистемы кэш-памяти с различной эффективностью. Для умножения матриц на GPU был организован схожий вычислительный эксперимент, результаты которого приведены в табл. 5.

Таблица 5. Зависимость времени обработки (в секундах) от размера блока, CPU i7 4770 + GPU GTX 770

N	GPU «в лоб»	Размер блока (блочное умножение, GPU)					
		1	2	4	8	16	32
256×256	0,055	0,099	0,060	0,059	0,052	0,057	0,055
512×512	0,075	0,397	0,108	0,064	0,055	0,059	0,056
1024×1024	0,21	–	0,471	0,129	0,075	0,065	0,064
2048×2048	1,3	–	–	0,647	0,218	0,141	0,137

Полученные результаты позволяют сделать вывод о том, что оптимальным размером блока является максимально допустимый в конфигурации запуска CUDA-ядра размер сетки потоков в блоке 32×32, при этом для кэширования подматриц используется 8 КБ разделяемой памяти на блок, имеющей объем 49 КБ.

При использовании блочного подхода из кода потока производится $\frac{2N}{B}$ обращений к глобальной памяти и $\frac{N}{B}\cdot 2B = 2N$ обращений к разделяемой памяти, что соответствует времени $\frac{2N}{B}t_G + 2Nt_S$ с. При наличии $B\times B = B^2$ потоков в блоке и $\frac{N}{B}\times\frac{N}{B} = \frac{N^2}{B^2}$ блоков в сетке общее время обработки составляет $\left(\frac{2N}{B}t_G + 2Nt_S\right)B^2\cdot\frac{N^2}{B^2} = \left(\frac{2N}{B}t_G + 2Nt_S\right)N^2 = \frac{2N^3}{B}t_G + 2N^3t_S$ с, а соответствующий теоретический выигрыш $\eta = \frac{2N^3t_G}{\frac{2N^3}{B}t_G + 2N^3t_S} = \frac{t_G}{\frac{t_G}{B} + t_S}$. Несложно заметить, что $\eta \to B$ при снижении времени обращения к разделяемой памяти $\left(t_S \to 0\right)$. С другой стороны, при $B\to\infty$ теоретический выигрыш определяется соотношением латентностей глобальной и разделяемой памяти: $\eta \to \frac{t_G}{t_S}$.
Таким образом, если в будущих поколениях GPU станет возможным запуск

конфигурации CUDA-ядра с большим числом потоков в блоке или более быстрая разделяемая память соответствующего объема, то это должно позитивно сказаться на реальной производительности в рассматриваемой задаче.

Еще одной оптимизацией, которая имеет положительное влияние на загрузку исполнительных устройств потоковых мультипроцессоров GPU (англ. Streaming Multiprocessor, SM), является раскрутка внутреннего цикла с целью повышения параллелизма на уровне инструкций (ILP). Часть кода CUDA-ядра (внутренний цикл по переменной z) с раскруткой на 4 итерации приведена ниже.

```
__global__ void MatrixMulKernel_blocks_unroll_to_4(
    float *A, float *B, float *C, int n
)
{
      [skip]

      for (int z = 0; z < n/BS; z++, zb += BS, ia += a_step, ib += b_step)
      {
            __shared__ float ta[BS][BS];
            __shared__ float tb[BS][BS];

            ta[tx][ty] = A[ia];
            tb[tx][ty] = B[ib];

            ta[tx+BS/4][ty] = A[ia+BS/4*n];
            tb[tx+BS/4][ty] = B[ib+BS/4*n];

            ta[tx+2*BS/4][ty] = A[ia+2*BS/4*n];
            tb[tx+2*BS/4][ty] = B[ib+2*BS/4*n];

            ta[tx+3*BS/4][ty] = A[ia+3*BS/4*n];
            tb[tx+3*BS/4][ty] = B[ib+3*BS/4*n];

            __syncthreads();

            for (int k = 0; k < BS; k++)
            {
                  sum1 += ta[tx][k]*tb[k][ty];
                  sum2 += ta[tx+BS/4][k]*tb[k][ty];
                  sum3 += ta[tx+2*BS/4][k]*tb[k][ty];
                  sum4 += ta[tx+3*BS/4][k]*tb[k][ty];
            }

            __syncthreads();
      }

      C[(x0+tx)*n + (y0+ty)] = sum1;
      C[(x0+tx+BS/4)*n + (y0+ty)] = sum2;
      C[(x0+tx+2*BS/4)*n + (y0+ty)] = sum3;
      C[(x0+tx+3*BS/4)*n + (y0+ty)] = sum4;
}
```

При этом соответствующая конфигурация запуска CUDA-ядра должна быть изменена в сторону уменьшения числа потоков в блоке в соответствующее число раз:

```
MatrixMulKernel_blocks_unroll_to_4<<<
```

```
    dim3(n/BS, n/BS),
    dim3(BS, BS/4)>>>
(A_gpu, B_gpu, C_gpu, n);
```

Результаты использования раскрутки циклов на 4 итерации приведены в табл. 6 (вычислительные эксперименты показали, что раскрутка на большее число итераций не приводит к дальнейшему сокращению времени обработки).

Таблица 6. Зависимость времени обработки от размера блока (в секундах) с учетом раскрутки цикла на 4 итерации, CPU i7 4770 + GPU GTX 770

N	GPU «в лоб»	GPU блочное умножение, 4 загрузки на поток					
		1	2	4	8	16	32
256×256	0,055	–	–	0,002	0,0008	0,0005	0,0005
512×512	0,075	–	–	0,01	0,003	0,002	0,002
1024×1024	0,21	–	–	0,07	0,02	0,011	0,01
2048×2048	1,3	–	–	0,58	0,14	0,071	0,058

Результирующее множество оптимизаций выполняемого действия приведено на рис. 1.

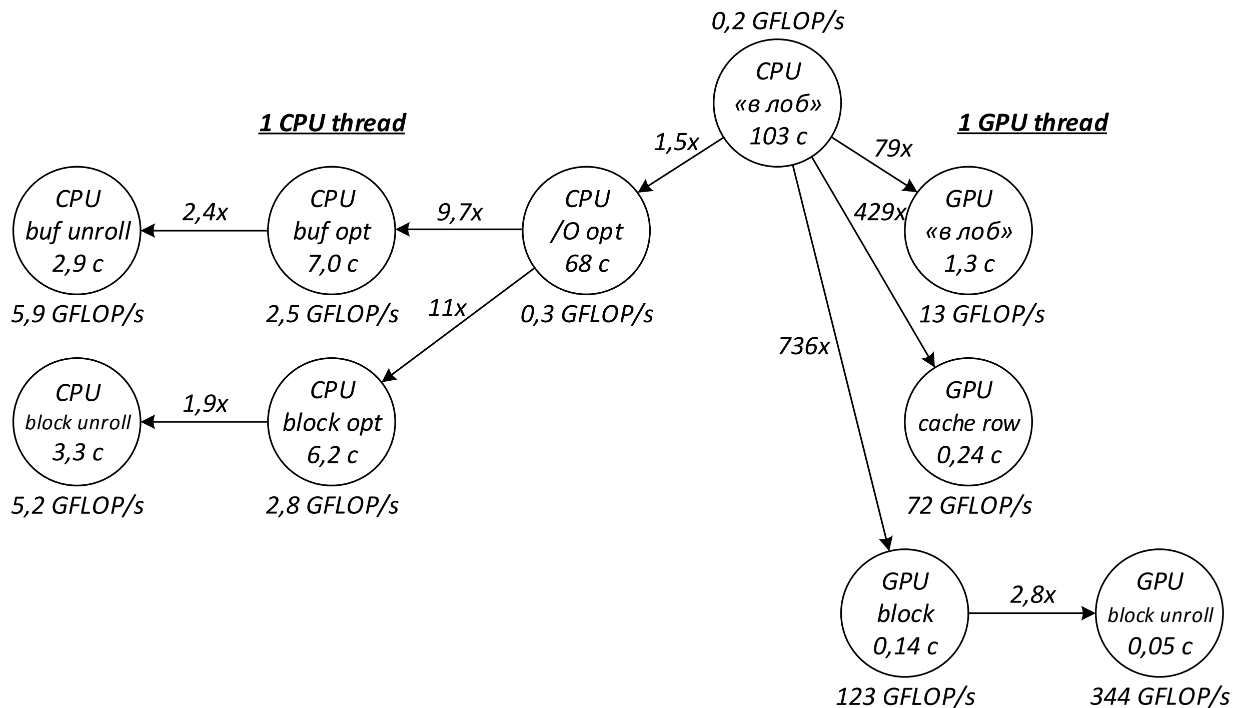

Рис. 1. Результирующее множество вариантов оптимизации исходного кода умножения квадратных матриц 2048×2048, CPU i7 4770 + GPU GTX 770

Оптимизации работы с кэш-памятью CPU позволяют достичь 35-кратного уменьшения времени однопоточной обработки и реальной производительности в пределах 5–6 GFLOP/s. Использование блочного подхода к выпол-

нению умножения на GPU позволяет получить более чем 2000-кратный выигрыш по сравнению с исходной неоптимизированной CPU-версией кода и почти 60-кратный по сравнению с соответствующей максимально оптимизированной на алгоритмическом уровне однопоточной версией, что для указанной машины соответствует производительности в 344 GFLOP/s. С целью выяснения эффективности выполненных оптимизаций на других аппаратных конфигурациях было проведено тестирование с использованием различных GPU, результаты которого приведены на рис. 2.

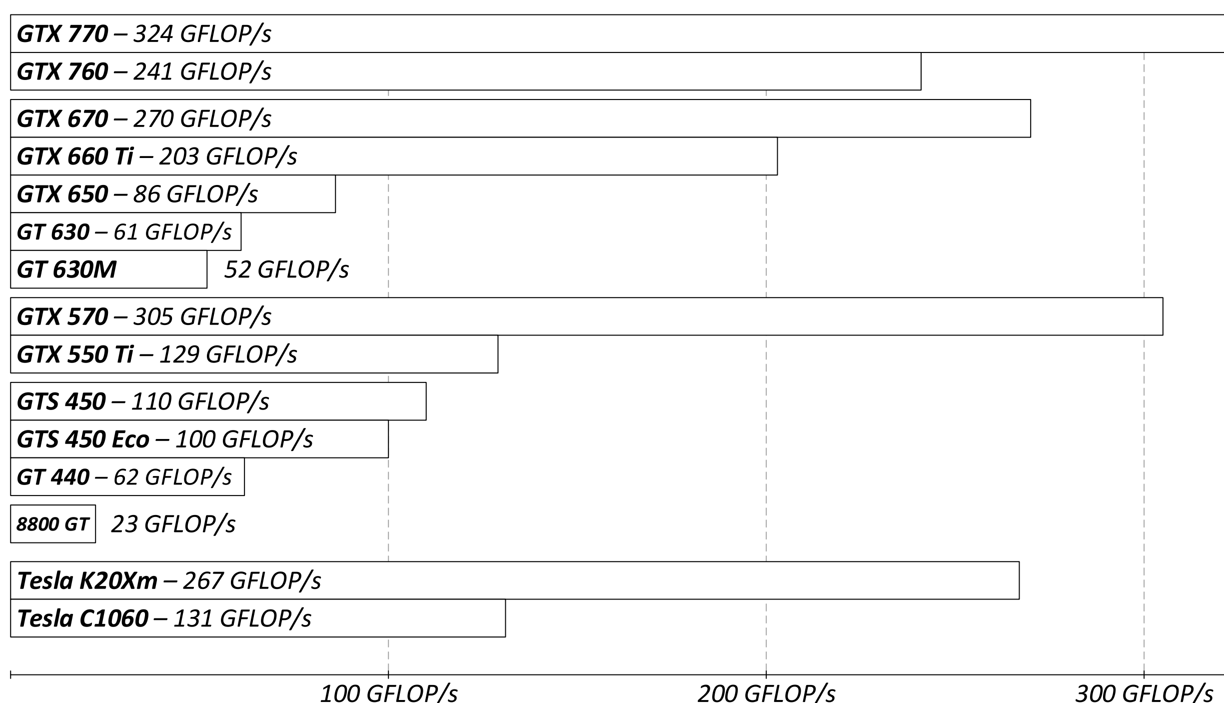

Рис. 2. Производительность различных GPU в задаче умножения матриц 2048×2048 с использованием блочного подхода с раскруткой цикла на 4 итерации

Полученные данные позволяют сделать вывод о том, что в рассматриваемой задаче определяющим является баланс между пропускной способностью глобальной памяти видеокарты (которая определяется как типом используемой памяти и ее параметрами, так и шириной шины) и итоговым числом FP-вычислителей (англ. CUDA Cores) в составе множества потоковых мультипроцессоров GPU. Приведенные результаты также наглядно демонстрируют, что профессиональные GPU Tesla, как минимум на порядок отличающиеся по цене от настольных аналогов, не демонстрируют существенного преимущества в производительности в рассматриваемой задаче. В то же время большинство современных GPU позволяют достичь в рассматриваемой задаче реальной производительности в диапазоне 100–300 GFLOP/s, что в десятки раз превосходит возможности CPU при их однопоточном использовании с оптимизацией работы кэш-памяти.

Список литературы

1. Ватутин Э.И., Зотов И.В. Построение матрицы отношений в задаче оптимального разбиения параллельных управляющих алгоритмов // Известия Курского государственного технического университета. Курск, 2004. № 2. С. 85–89.
2. Ватутин Э.И., Мартынов И.А., Титов В.С. Оценка реальной производительности современных процессоров в задаче умножения матриц для однопоточной программной реализации // Известия Юго-Западного государственного университета. Серия: Управление, вычислительная техника, информатика. Медицинское приборостроение. 2013. № 4. С. 11–20.
3. Параллельные вычисления на GPU. Архитектура и программная модель CUDA / Боресков А.В., Харламов А.А. Марковский Н.Д. и др. М.: изд-во Московского университета, 2012. 336 с.