Курс «Параллельное программирование»

Лабораторная работа №5. Параллельное вычисление произведения двух матриц на графическом процессоре.

Юлдашев Артур Владимирович
art@ugatu.su

Спеле Владимир Владимирович
spele.vova@ugatu.su

Кафедра высокопроизводительных вычислительных технологий и систем (ВВТиС)

Цель работы

На примере задачи параллельного вычисления произведения двух матриц научиться проводить декомпозицию на двумерные блоки, а также оптимизировать программы для GPU с использованием разделяемой памяти.

Задачи

- Разработка программы, реализующей перемножения двух квадратных матриц на CPU (последовательно и параллельно).
- Реализация простейшего алгоритма вычисления матричного произведения на GPU с использованием глобальной памяти.
- Реализация блочного алгоритма вычисления матричного произведения с использованием разделяемой памяти.
- Реализация различных вариантов вычисления AA^T
- Тестирование производительности и профилирование разработанных программ.

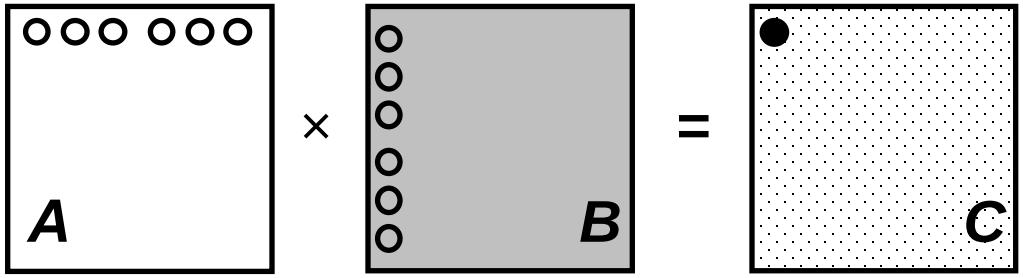
Вычисление произведения двух матриц

Требуется перемножить матрицы $A(N \times M)$ и $B(M \times L)$ и вычислить квадрат евклидовой нормы результирующей матрицы:

$$C = AB, \qquad \|C\|^2 = \sum_{i=1}^{N} \sum_{j}^{L} c_{ij}^2$$

Матричное умножение выражается формулой

$$c_{ij} = \sum_{k=1}^{M} a_{ik} b_{kj}, \qquad i = \overline{1 \dots N}, j = \overline{1 \dots L}$$



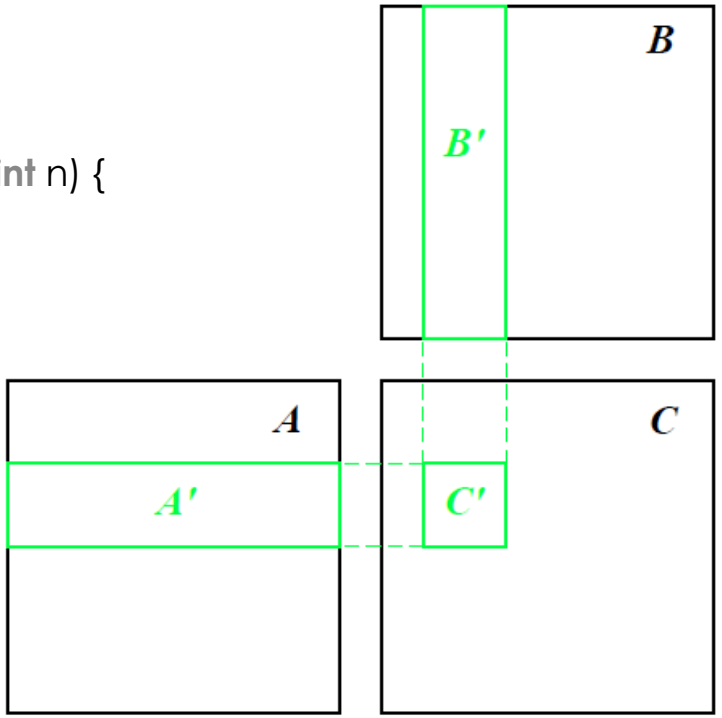
Переход от последовательной программы к параллельной (для GPU)

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        for (k = 0; k < n; k++)
            c[i * n + j] += a[i * n + k] * b[k * n + j];
```

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        for (k = 0; k < n; k++)
            c[i * n + j] += a[i * n + k] * b[k * n + j];
```

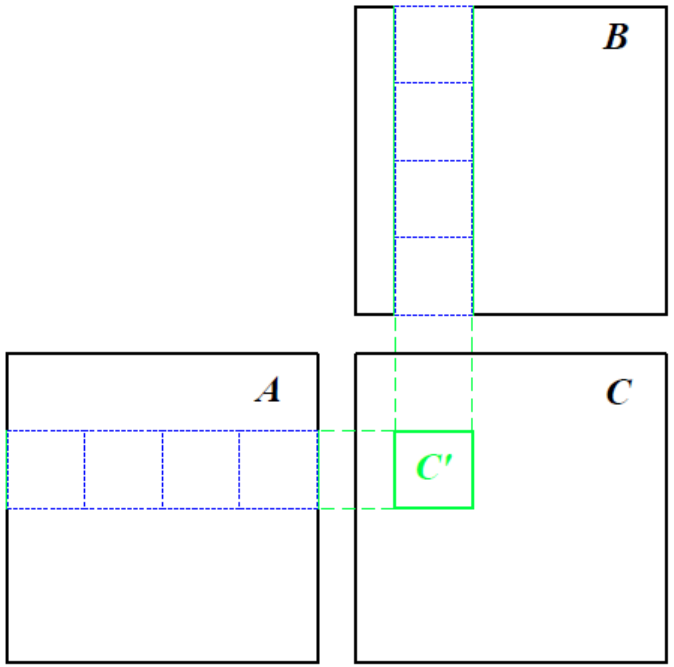
Реализация простейшего алгоритма с использованием глобальной памяти GPU

```
#define BS 32


__global__ void kernel (float *a, float *b, float *c, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float s = 0;
    for(int i = 0; i < n; i++)
        s += a[row * n+ i] * b[i * n + col];
    c[row * n + col] = s;
}

int main(void) {
    …
    dim3 threads( BS, BS );
    dim3 blocks(N / BS, N / BS);
    kernel<<<blocks, threads>>>(adev, bdev, cdev, N );
    …
}
```



Блочный алгоритм с использованием разделяемой памяти GPU

- Полосы, требуемые для вычисления C', проблематично разместить в разделяемой памяти
- Матрица C' может быть вычислена как сумма произведений квадратных блоков из A и B
- Это позволяет поэтапно подгружать блоки матриц A и B в разделяемую память
- При размерах блока 32 × 32 количество обращений к глобальной памяти можно снизить в 32 раза до 2N/32 на каждый вычисляемый элемент матрицы C'



$$C' = A'_1 * B'_1 + \dots + A'_{N/16} * B'_{N/16}$$

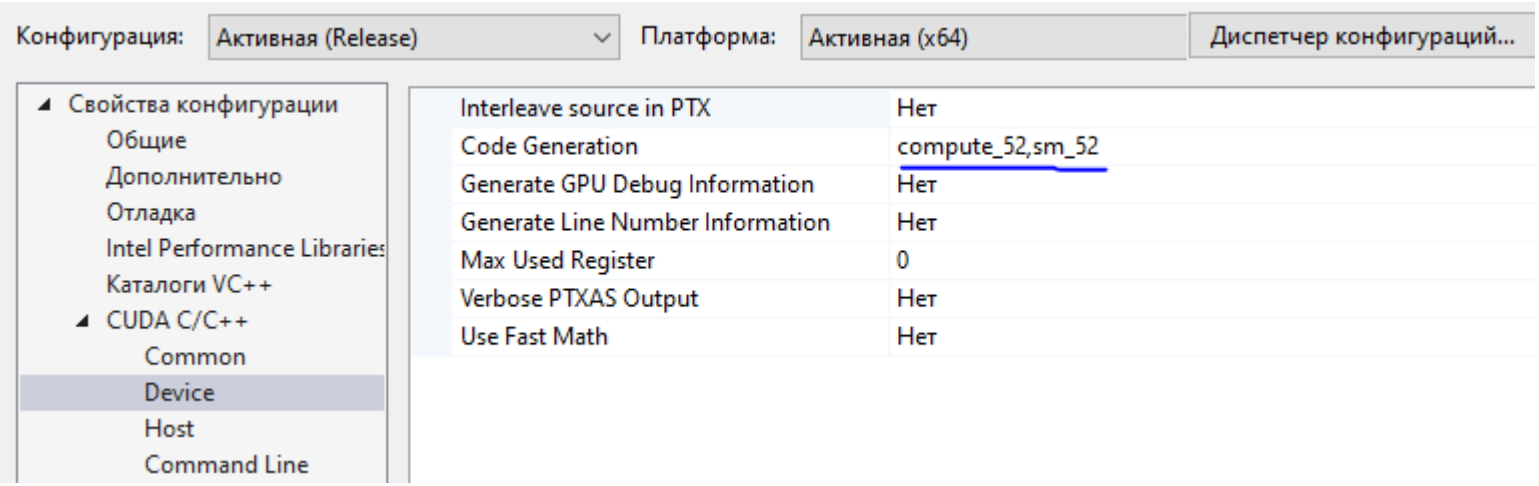
Реализация блочного алгоритма с использованием разделяемой памяти GPU

```
__global__ void mul_shared(float *a, float *b, float *c, int N) {
    int i, j,
                bx = blockIdx.x, by = blockIdx.y,
                tx = threadIdx.x, ty = threadIdx.y,
                ix = bx * blockDim.x + tx, iy = by * blockDim.y + ty;
    float s = 0;
    __shared__ float as[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float bs[BLOCK_SIZE][BLOCK_SIZE];

    for(i = 0; i < N / BLOCK_SIZE; i++) {
        as[ty][tx] = a[(by * BLOCK_SIZE + ty) * N + i * BLOCK_SIZE + tx];
        bs[ty][tx] = b[(i * BLOCK_SIZE + ty) * N + bx * BLOCK_SIZE + tx];

        __syncthreads();

        for(j = 0; j < BLOCK_SIZE; j++)
            s += as[ty][j] * bs[j][tx];

        __syncthreads();
    }
    c[iy * N + ix] = s;
}
```

Требования к программам (часть 1)

1. Размерности матриц предполагаются кратными 32 (если нет, программа завершается с сообщением об ошибке).
2. Перемножаются матрицы A и B с элементами вещественного типа одинарной точности, сгенерированные ранее на CPU.
3. Необходимо предусмотреть расчет нормы матрицы C.
4. Реализовать последовательную и многопоточную (OpenMP) версии для CPU, а также с использованием глобальной и разделяемой памяти для GPU.

При компиляции для CPU выставить ключ -O2
При компиляции для GPU выставить ключ -arch=sm_50 или выше



Часть1. Исследование эффективности реализаций C = *AB*

Таблица 1. Время выполнения программ.

Версия, платформа	Время	N = 256	N = 1024	N = 4096
Последовательная, CPU	Время расчета на CPU			
Параллельная, CPU	Время расчета на CPU			
С использованием глобальной памяти, GPU	Время обмена данными с GPU			
	Время расчета на GPU			
	Полное время работы на GPU			
С использованием разделяемой памяти, GPU	Время обмена данными с GPU			
	Время расчета на GPU			
	Полное время работы на GPU			

Часть1. Исследование эффективности реализаций C = *AB (продолжение)*

Таблица 2. Ускорение на GPU относительно параллельной версии для CPU.

Версия	N = 256	N = 1024	N = 4096
С использованием глобальной памяти			
С использованием разделяемой памяти			

Таблица 3. Производительность разработанных версий.

Версия, платформа	N = 256	N = 1024	N = 4096
Параллельная, CPU			
С использованием глобальной памяти, GPU			
С использованием разделяемой памяти, GPU			

Вычисление произведения C = AA^T

Требуется перемножить матрицу $A(N \times N)$ и транспонированную матрицу A^T:

$$C = AA^T$$

Реализация с использованием глобальной памяти:

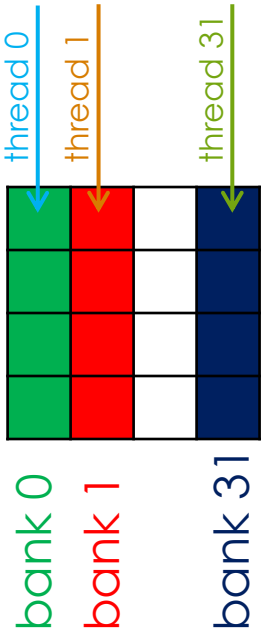
```
__global__ void simpleTransposeMultiply(float *a, float *c, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0;
    for(int i = 0; i < N; i++)
        sum += a[row * N+ i] * a[col * N + i];
    c[row * N + col] = sum;
}
```

Пример реализации $C = AA^T$ с использованием разделяемой памяти

```
__global__ void mul_transp_shared(float *a, float *c, int N) {
    int i, j,
                bx = blockIdx.x, by = blockIdx.y,
                tx = threadIdx.x, ty = threadIdx.y,
                ix = bx * blockDim.x + tx, iy = by * blockDim.y + ty;
    float s = 0;
    __shared__ float as[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float bs[BLOCK_SIZE][BLOCK_SIZE];

    for(i = 0; i < N / BLOCK_SIZE; i++) {
        as[ty][tx] = a[(by * BLOCK_SIZE + ty) * N + i * BLOCK_SIZE + tx];
        bs[ty][tx] = a[(bx * BLOCK_SIZE + tx) * N + i * BLOCK_SIZE + ty];

        __syncthreads();

        for(j = 0; j < BLOCK_SIZE; j++)
            s += as[ty][j] * bs[j][tx];

        __syncthreads();
    }
    c[iy * N + ix] = s;
}
```

Работа с памятью. Вариант A

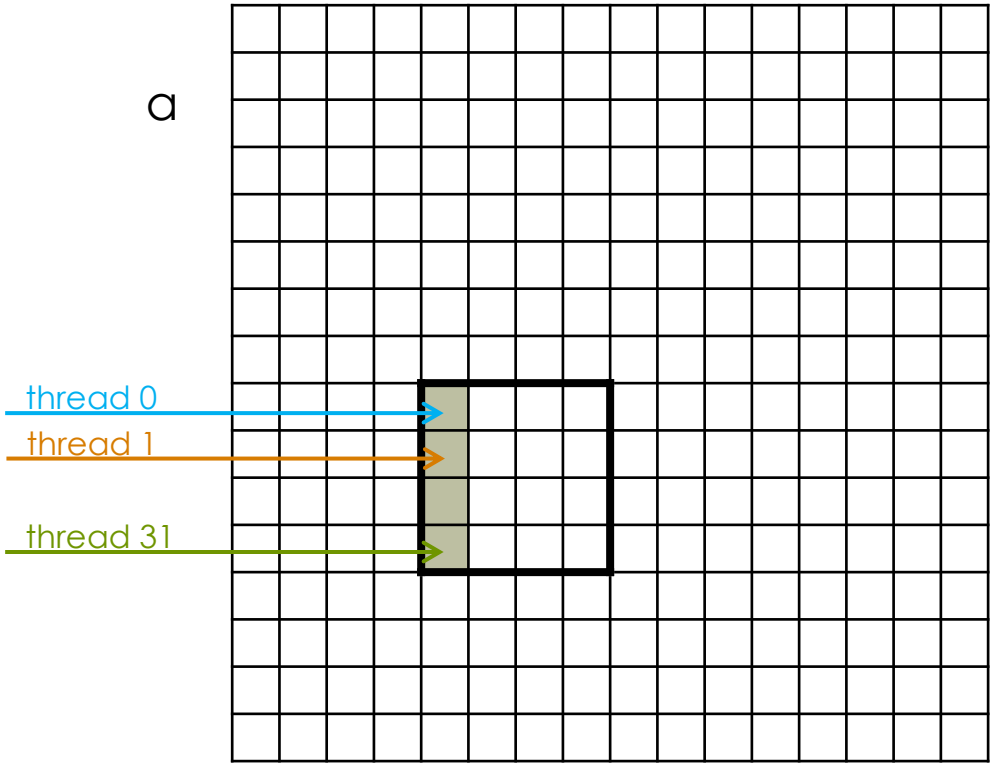
No bank conflict



bs

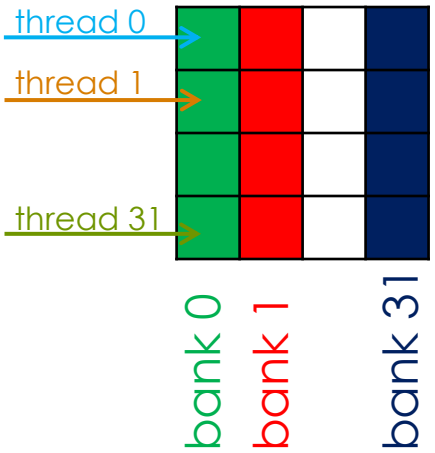
No coalescing

a



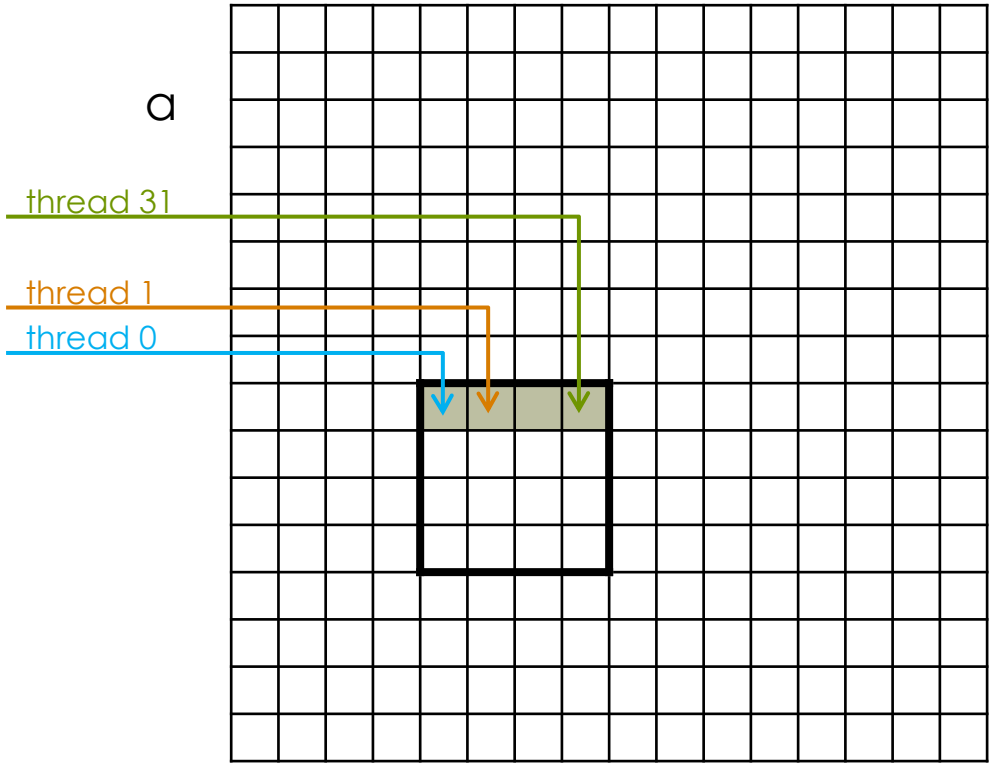
Работа с памятью. Вариант Б

Bank conflict



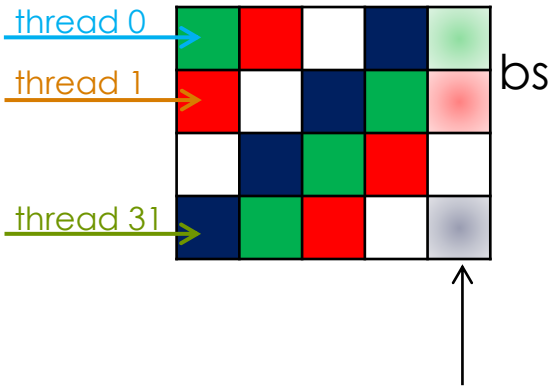
bs

Coalescing



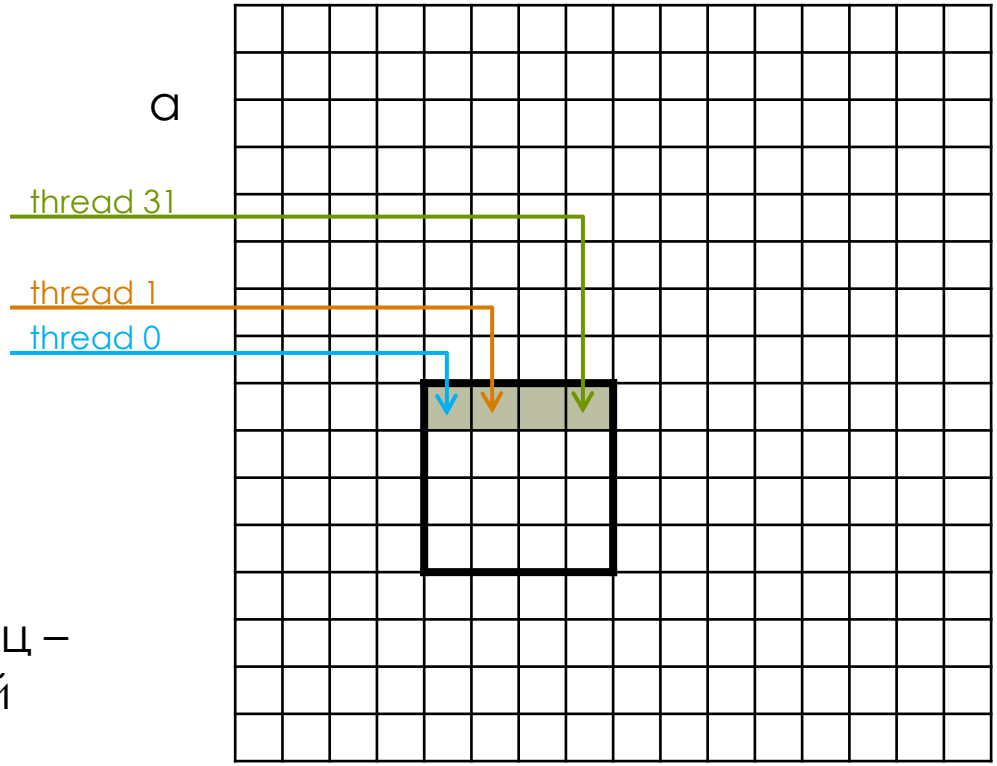
Работа с памятью. Вариант Б*

No bank conflict



Фиктивный столбец – не хранит никакой информации

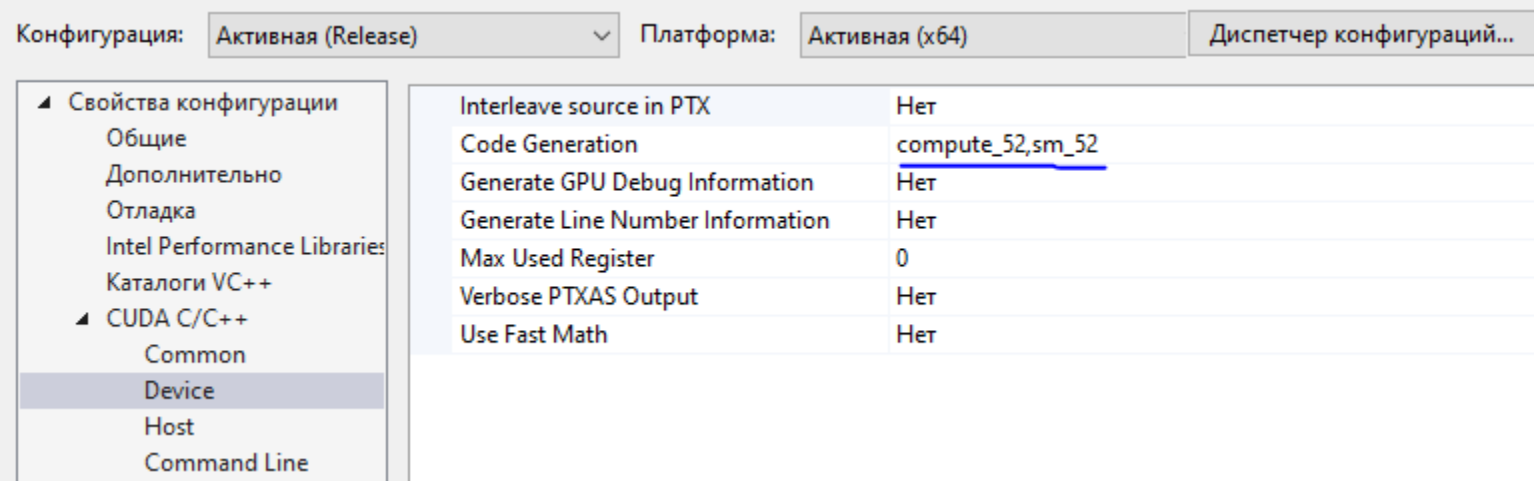
Coalescing



Требования к программам (часть 2)

1. Реализовать три версии ядра перемножения матрицы A на A^T с использованием разделяемой памяти.
   1. Без конфликтов банков и без коалесинга (Вариант А).
   2. С конфликтами банков и коалесингом (Вариант Б).
   3. Без конфликтов банков и с коалесингом (Вариант Б*).
2. Провести исследование полученных реализаций

При компиляции выставить ключ -arch=sm_50(не меньше, чем 50)



Часть 2. Исследование эффективности реализаций $C = AA^T$

Таблица 4. Время выполнения функции-ядра.

Версия	N = 256	N = 1024	N = 4096
Вариант А			
Вариант Б			
Вариант Б*			

Таблица 5. Производительность функции-ядра.

Версия	Размерность	N = 256	N = 1024	N = 4096
Вариант А	Производительность			
	% от пиковой			
Вариант Б	Производительность			
	% от пиковой			
Вариант Б*	Производительность			
	% от пиковой			

Требования к оформлению отчета

- В отчет по проделанной работе включить
  1) Характеристики CPU и GPU.
  2) По задаче реализации $C = AB$:
     1) заполненные таблицы 1-3;
     2) графики, построенные по таблицам 2, 3;
     3) выводы о полученных результатах;
     4) листинг разработанной программы;
     5) (строку компиляции).
  3) По задаче реализации $C = AA^T$:
     1) заполненные таблицы 4, 5;
     2) график, построенный по таблице 5;
     3) для матрицы 1024 ×1024 привести число конфликтов банков при реализации варианта А, Б и Б*;
     4) выводы о полученных результатах;
     5) листинг конечной оптимизированной программы (Б*);
     6) (строку компиляции).

Профилировка в консоли

nvprof – консольный профилировщик

Пример запуска:

nvprof ./a.out

- nvprof --query-events – вывод всех доступных для анализа событий
- nvprof --query-metrics – вывод всех доступных для анализа метрик

Профилировка в консоли

Нужные события (CC 5.x)

- shared_st_bank_conflict – конфликт банков при записи
- shared_ld_bank_conflict – конфликт банков при чтении

Нужные метрики

- flop_count_sp – число операций с плавающей точкой (в одинарной точности)
- flop_sp_efficiency – процент от пиковой производительности в одинарной точности

Примеры запуска:

```
nvprof -m flop_count_sp, flop_sp_efficiency ./a.out
nvprof -e shared_st_bank_conflict, shared_ld_bank_conflict ./a.out
```

Профилировка в Visual Profiler

В Visual Profiler после создания timeline нажимаем меню Run -> Configure Metrics and Events.
Во вкладке Metrics выбираем **Instruction** -> **FLOP Efficiency(Peak Single)** и **FP Instructions(Single).** Во вкладке **Events** выбираем **shared_st_bank_conflict** и **shared_ld_bank_conflict**. Нажимаем Apply and Run. После повторного генерирования timeline во вкладке GPU Details появляются нужные нам метрики.

Analysis | GPU Details (Summary) | CPU Details | OpenACC Details | OpenMP Details | Console | Settings

Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem	FLOP Efficiency(Peak Single)	FP Instructions(Single)	shared_st_bank_conflict	shared_ld_bar
kernel(float*, float*)	1	458,5808 ms	27	8192	0	8,9%	68719476736	2080374784	