

# Оптимизация CUDA программ

Лекция 7  
(повторение)

# Kepler GK110

Архитектура

7.1 миллиарда  
транзисторов

15 SMX

мультимикроспроцессоров

6 GB памяти

> 1 TFLOP FP64

1.5 MB L2 кэш

384-bit GDDR5

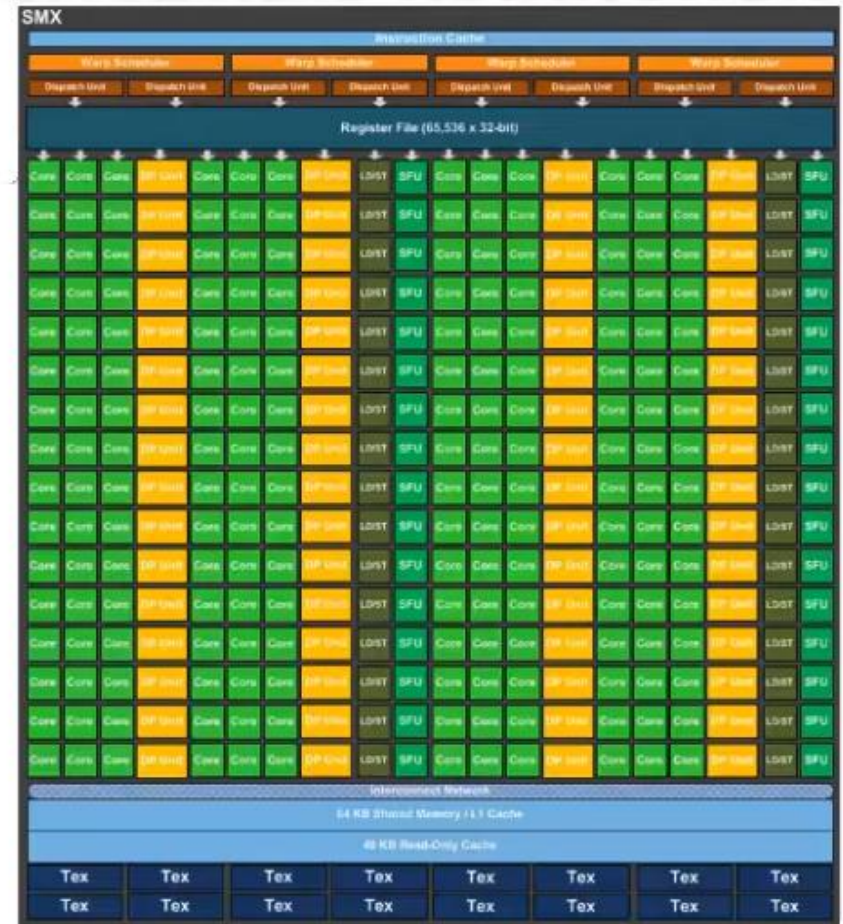
PCI Express 3



# SMX: основные компоненты

## Состав SMX:

- 192 ядра
- 64 DP Unit
- 32 LD/ST Unit
- 32 SFU
- 64 КБ L1 кэш/ разделяемая память
- 16 текстурных процессоров
- 48 КБ read-only кэша
- 65536 32-битных регистра
- 4 warp scheduler
- 8 dispatch unit
- Кэш инструкций



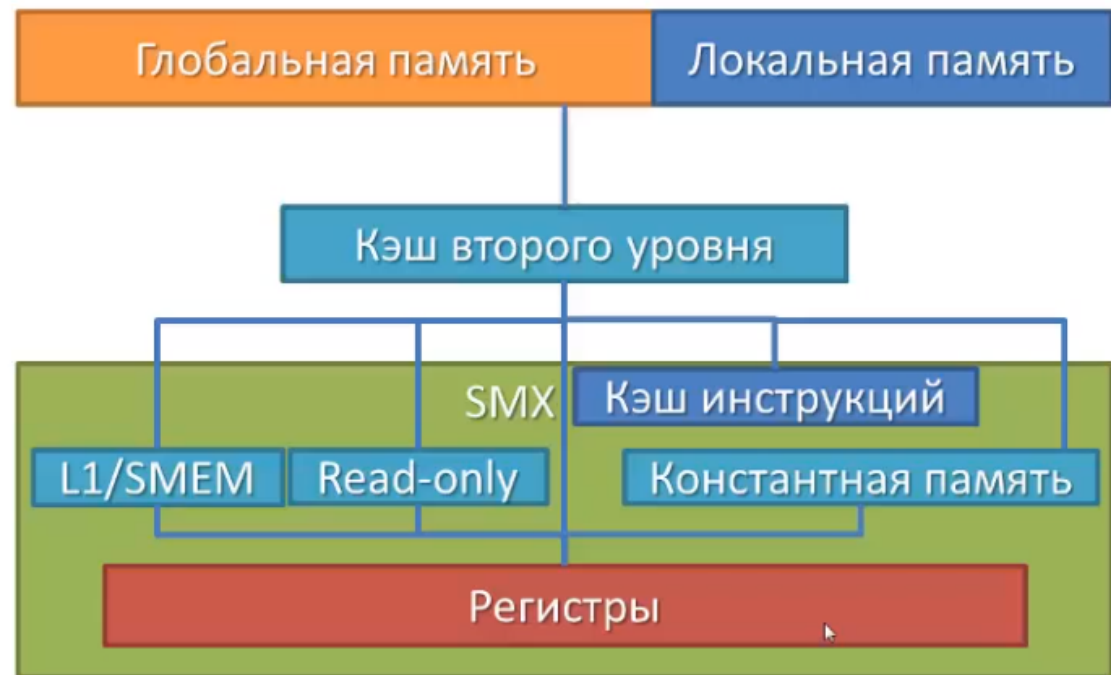
# Иерархия памяти

Чем можно управлять:

- Глобальная память
- Разделяемая память

Косвенно или частично:

- Локальная память
- Регистры
- L1 кэш
- Read-only кэш



# Конфигурация сетки

Сетка – распределение нитей в блоках

- Две основных причины:
  - ✓ Дать достаточно параллельных задач мультипроцессору
  - ✓ Распределить работу между мультипроцессорами
- Что необходимо помнить:
  - ✓ Количество нитей на блок
  - ✓ Количество блоков
  - ✓ Количество работы на блок



# Занятость устройства

Занятость устройства (Occupancy): количество конкурентно-исполняемых нитей на SMX

- Может быть выражено как количество нитей (варпов)
- Или как процент от максимального количества нитей

Определяется несколькими факторами:

- Количество регистров на одну нить
  - ✓ Регистры SMX делятся среди нитей
- Объем разделяемой памяти на блок
  - ✓ Разделяемая память SMX делится среди блоков

## Ресурсы Kepler SMX:

- 64k 32-х битных регистров
- До 48 КБ разделяемой памяти
- До 2048 конкурентных нитей
- До 16 конкурентных блоков

# Размеры блоков

Размер блока кратен размеру варпа

- Даже, если вы запрашиваете меньше нитей, он все-равно аппаратно округляется

Блоки могут быть слишком маленькими

- SMX может выполнять одновременно до 16 блоков
  - ✓ Лимит блоков может ограничивать производительность

Ресурсы SMX:  
-Регистры  
-Разделяемая память

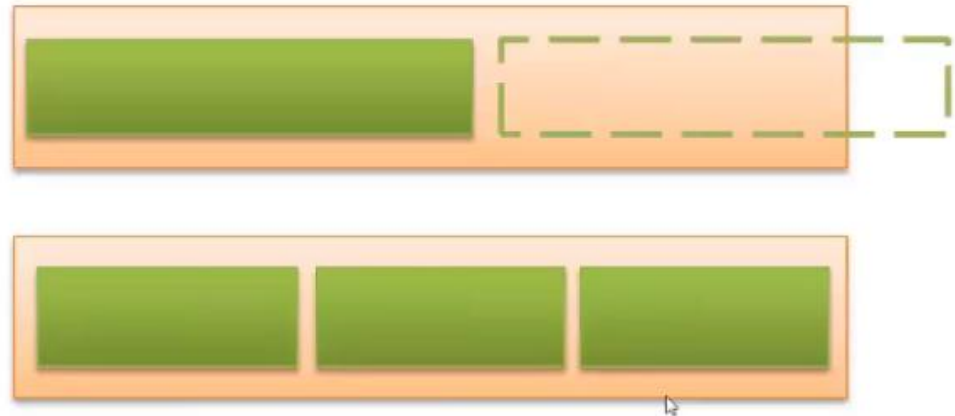


# Размеры блоков

Блоки могут быть слишком большими

- Эффект квантования:
  - ✓ Достаточно ресурсов SMX для дополнительных нитей, но недостаточно для еще одного большого блока
  - ✓ Блок не начинает выполняться, пока нет достаточного количества ресурсов для всех его нитей

**Ресурсы SMX:**  
-Регистры  
-Разделяемая память



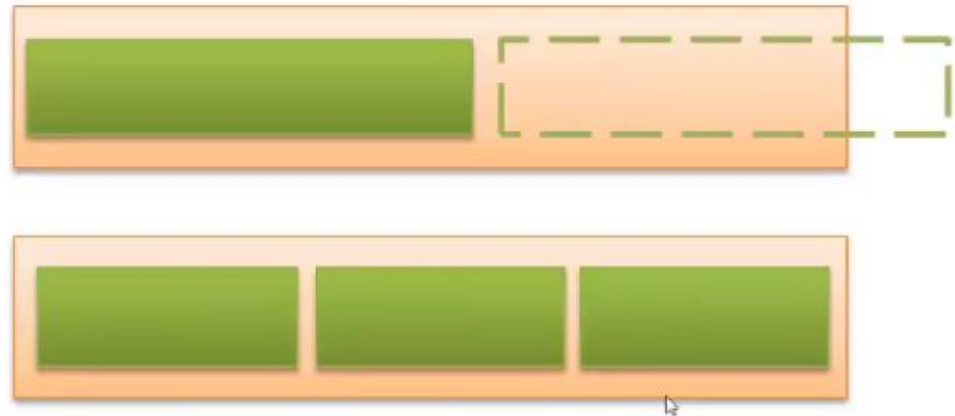


# Размеры блоков

Блоки могут быть слишком большими

- Эффект квантования:
  - ✓ Достаточно ресурсов SMX для дополнительных нитей, но недостаточно для еще одного большого блока
  - ✓ Блок не начинает выполняться, пока нет достаточного количества ресурсов для всех его нитей

**Ресурсы SMX:**  
-Регистры  
-Разделяемая память



# Occupancy calculator

Поставляется с SDK

- **CUDA Samples -> documentation -> CUDA Occupancy Calculator**

Помогает рассчитать теоретическую занятость устройства исходя из указанных параметров:

- Архитектура
- Конфигурация SMEM/L1
- Количество нитей в блоке
- Количество регистров на нить
- Количество разделяемой памяти на блок

# Occupancy calculator

## CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):  (Help)  
1.b) Select Shared Memory Size Config (bytes)

2.) Enter your resource usage:  
Threads Per Block  (Help)  
Registers Per Thread   
Shared Memory Per Block (bytes)

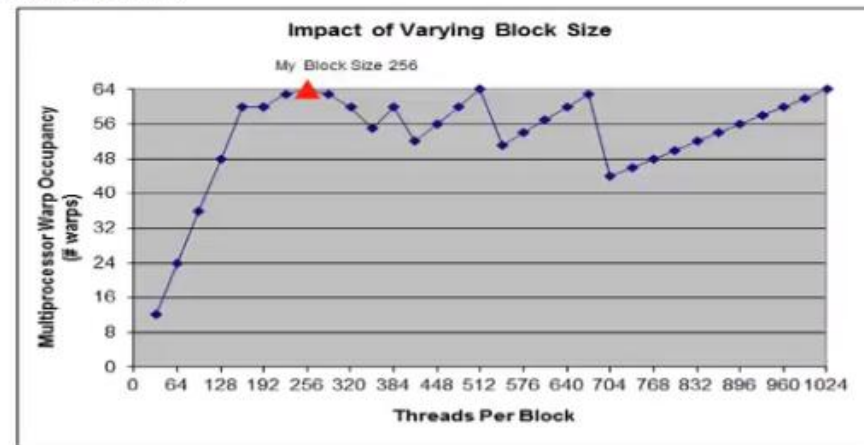
(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:  
Active Threads per Multiprocessor  (Help)  
Active Warps per Multiprocessor   
Active Thread Blocks per Multiprocessor   
Occupancy of each Multiprocessor

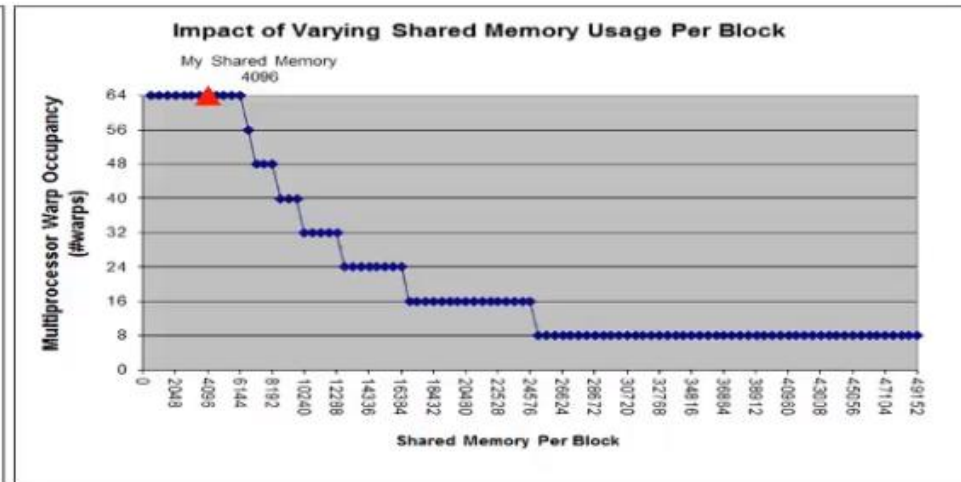
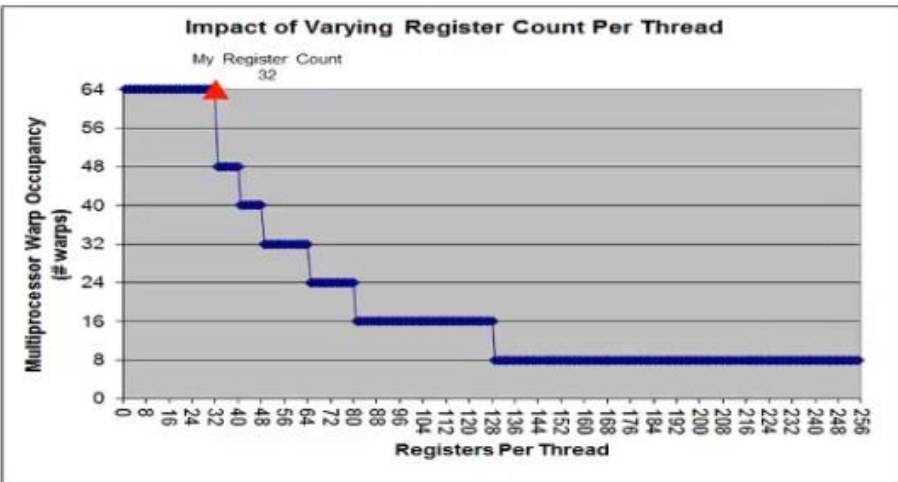
Physical Limits for GPU Compute Capability: 3.5	
Threads per Warp	32
Warps per Multiprocessor	64
Threads per Multiprocessor	2048
Thread Blocks per Multiprocessor	16
Total # of 32-bit registers per Multiprocessor	65536
Register allocation unit size	256
Register allocation granularity	warp
Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	256
Warp allocation granularity	4
Maximum Thread Block Size	1024

[Click Here for detailed instructions on how to use this occupancy calculator.](#)  
[For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda](http://developer.nvidia.com/cuda)

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.



# Occupancy calculator



# Рекомендации

## Размер блоков:

- Начинайте с 128-256 нитей на блок
  - ✓ Увеличивайте, или уменьшайте в зависимости от вашей функции
- Кратны размеру варпа
- Если занятость устройства критична:
  - ✓ Проверьте, количество требуемых ресурсов – регистры и разделяемая память

## Размер сетки

- 1000 или больше блоков
  - ✓ Равномерное распределение работы по всему GPU
  - ✓ Код будет готов для исполнения на разных поколениях GPU (в том числе и будущих)



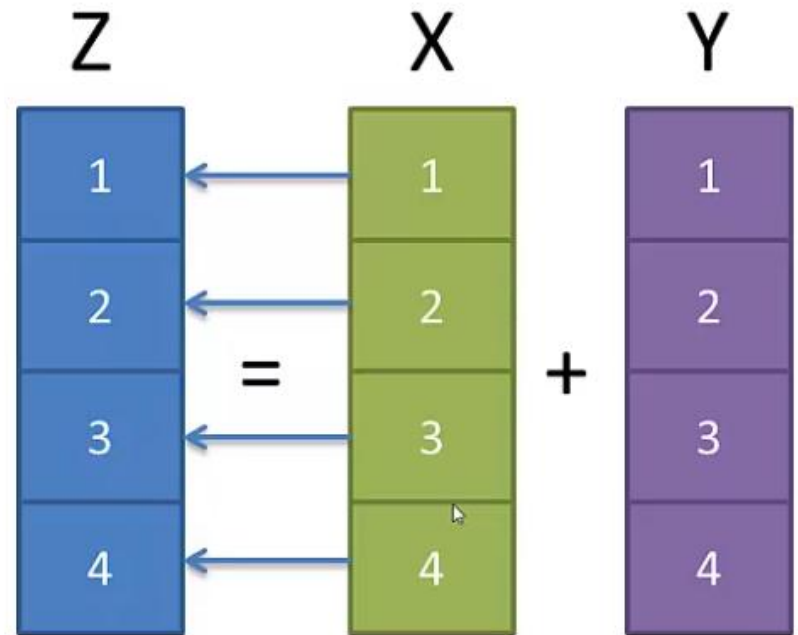
# Сложение векторов

Загрузить элемент массива X  
из глобальной памяти

Загрузить элемент массива Y  
из глобальной памяти

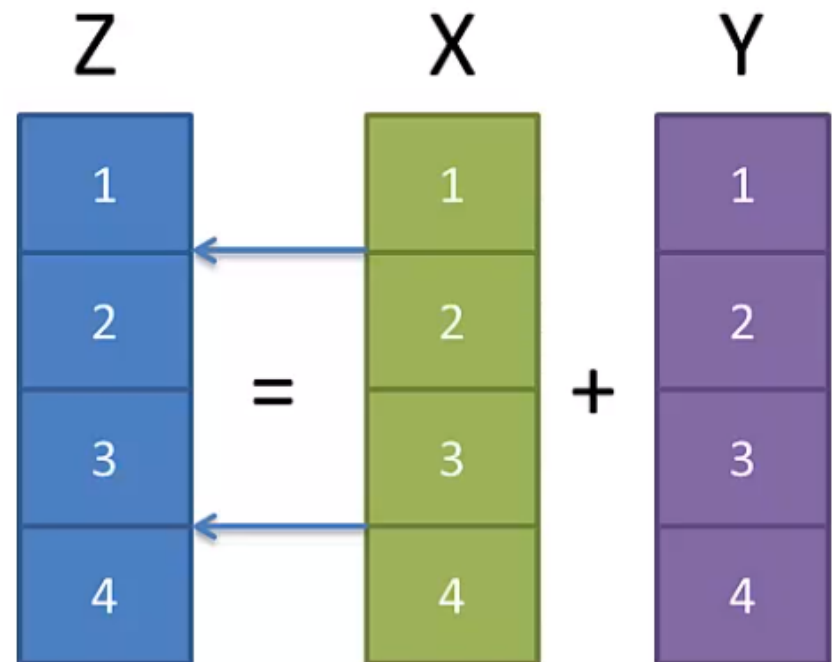
Сложить элементы

Выгрузить результат в  
глобальную память



# Сложение векторов

Загрузить первый элемент массива X из глобальной памяти  
Загрузить второй элемент массива X из глобальной памяти  
Загрузить первый элемент массива Y из глобальной памяти  
Загрузить второй элемент массива Y из глобальной памяти  
Сложить первые элементы  
Сложить вторые элементы  
Выгрузить результат первого сложения в глобальную память  
Выгрузить результат второго сложения в глобальную память



# Параллелизм инструкций

Латентность инструкций  $\approx 11$  тактов

- Требуется 44 варпа для покрытия латентности

На 192 ядра есть всего 4 warp scheduler'a

- Каждый из них исполняет сразу 2 инструкции для своего варпа (если возможно)
  - ✓ В случае если нет параллелизма на уровне инструкций (ILP) загружены только 128 ядер

ILP позволяет

- выиграть дополнительные 10-30% производительности
- снизить количество используемых ресурсов

# Параллелизм инструкций

Как обеспечить параллелизм?

- Нити должны рассчитывать значения нескольких выходных переменных одновременно
- Использовать `#pragma unroll` для циклов внутри ядра
- Разделять инструкции загрузки (сохранения) данных из глобальной памяти и вычислений

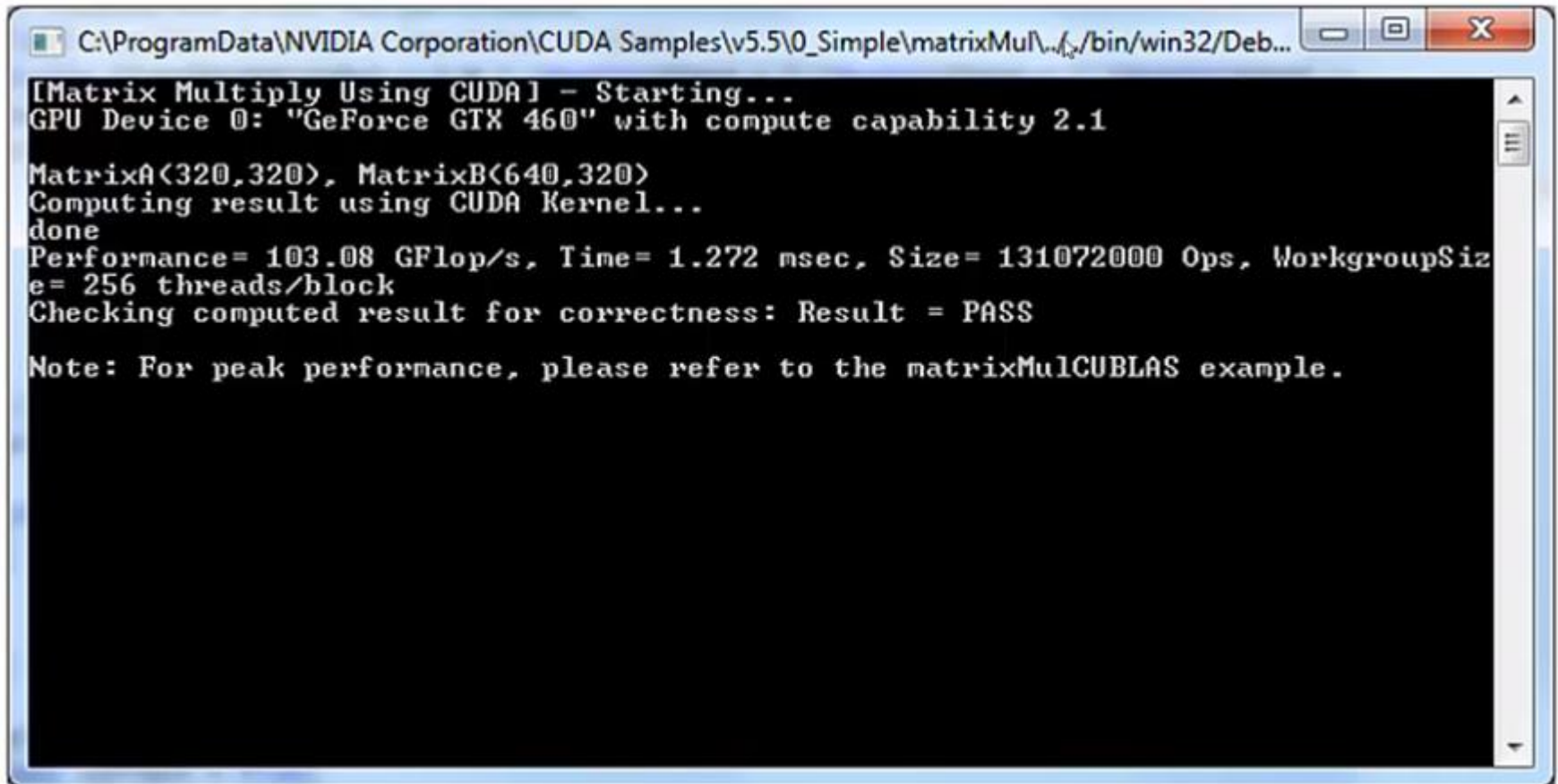
# Пример ILP

```
for (int a = aBegin, b = bBegin; a <= aEnd; a +=  
    aStep, b += bStep) {  
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];  
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];  
    As[ty][tx] = A[a + wA * ty + tx];  
    Bs[ty][tx] = B[b + wB * ty + tx];  
    __syncthreads();  
    #pragma unroll  
    for (int k = 0; k < BLOCK_SIZE; ++k)  
    {  
        Csub += As[ty][k] * Bs[k][tx];  
    }  
    __syncthreads();  
}  
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;  
C[c + wB * ty + tx] = Csub;
```

Каждая нить  
рассчитывает  
значение одного  
элемента




# Пример ILP



```
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v5.5\0_Simple\matrixMul\./bin/win32/Deb...  
[Matrix Multiply Using CUDA] - Starting...  
GPU Device 0: "GeForce GTX 460" with compute capability 2.1  
MatrixA<320,320>, MatrixB<640,320>  
Computing result using CUDA Kernel...  
done  
Performance= 103.08 GFlop/s, Time= 1.272 msec, Size= 131072000 Ops, WorkgroupSize= 256 threads/block  
Checking computed result for correctness: Result = PASS  
Note: For peak performance, please refer to the matrixMulCUBLAS example.
```

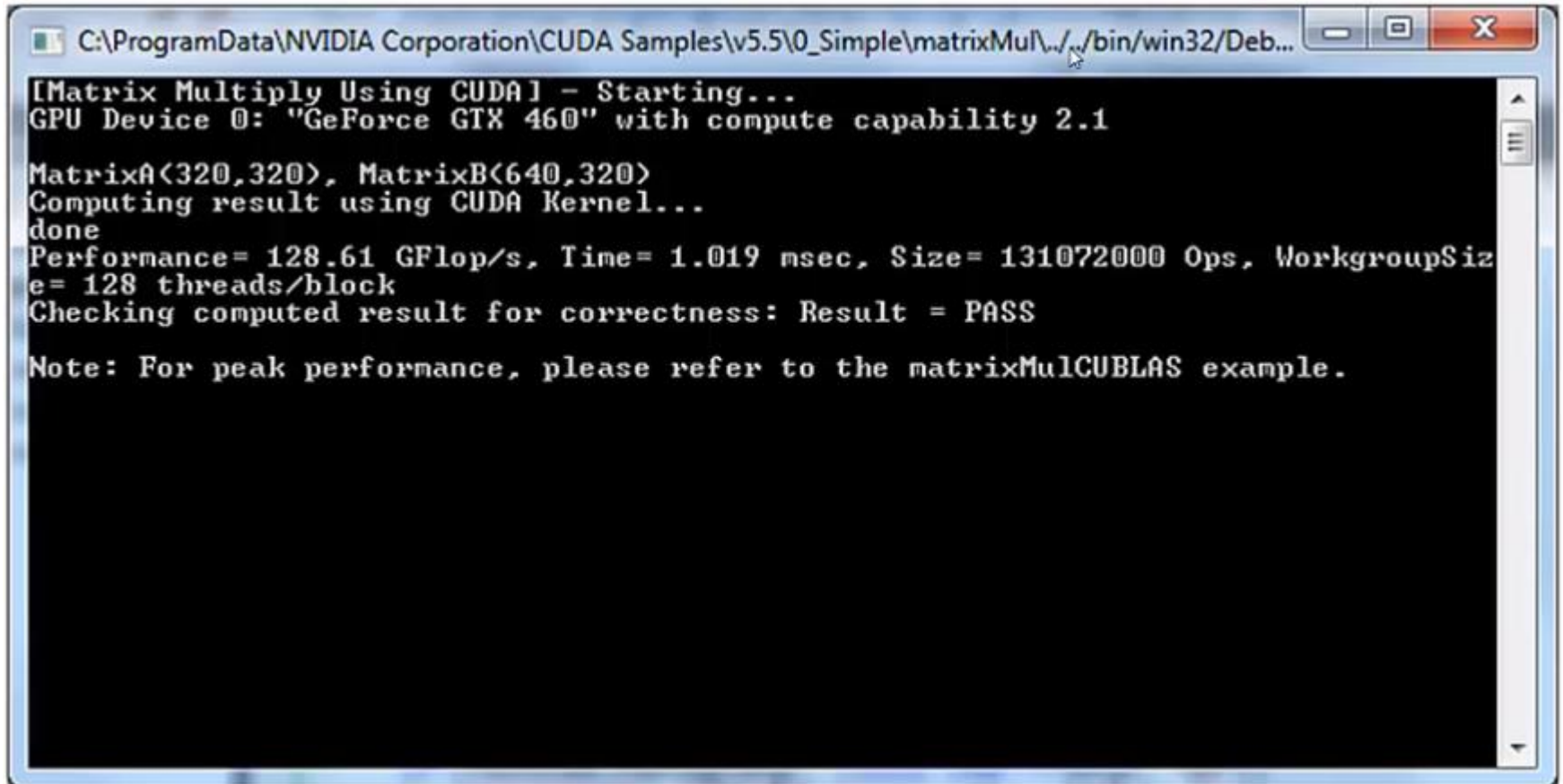
# Пример ILP

```
for (int a = aBegin, b = bBegin; a <= aEnd; a +=
aStep, b += bStep){
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];
    As[ty+8][tx] = A[a + wA * (ty+8) + tx];
    Bs[ty+8][tx] = B[b + wB * (ty+8) + tx];
    __syncthreads();
    #pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k){
        Csub[0] += As[ty][k] * Bs[k][tx];
        Csub[1] += As[ty+8][k] * Bs[k][tx];
    }
    __syncthreads();
}
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub[0];
C[c + wB * (ty+8) + tx] = Csub[1];
```

 Каждая нить  
рассчитывает  
значение двух  
элементов

 Прирост  
производительности – 24%

# Пример ILP



```
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v5.5\0_Simple\matrixMul\././bin/win32/Deb...  
[Matrix Multiply Using CUDA] - Starting...  
GPU Device 0: "GeForce GTX 460" with compute capability 2.1  
  
MatrixA<320,320>, MatrixB<640,320>  
Computing result using CUDA Kernel...  
done  
Performance= 128.61 GFlop/s, Time= 1.019 msec, Size= 131072000 Ops, WorkgroupSize= 128 threads/block  
Checking computed result for correctness: Result = PASS  
  
Note: For peak performance, please refer to the matrixMulCUBLAS example.
```