

# CUDA

## Оптимизация программ

Романенко А.А.

[arom@ccfit.nsu.ru](mailto:arom@ccfit.nsu.ru)

Новосибирский государственный университет

# Процесс оптимизации

- Определяем что ограничивает производительность
  - Пропускная способность памяти (memory bandwidth)
  - Исполнение инструкций (instruction throughput)
  - Задержка (latency)
  - Комбинации
- Исследуем ограничители в порядке важности
  - Насколько эффективно работает
  - Анализ и нахождение возможных проблем
  - Применение оптимизаций

# Профилирование

- `clock_t clock()` - счетчик, который увеличивается с каждым тактом GPU
  - Разность значений в начале и конце ядра — количество тактов, затраченных GPU на выполнение потока. НЕ ПРОВЕДЕННОЕ в исполнении потока.
- Переменные окружения
  - `CUDA_PROFILE=1`
  - `CUDA_PROFILE_LOG`
  - `CUDA_PROFILE_CONFIG`

# Профилирование. Файл конфигурации

- \* timestamp
- \* gridsize
- \* threadblocksize
- \* dynsmemperblock
- \* stasmemperblock
- \* regperthread
- \* memtransferdir
- \* memtransfersize
- \* streamid

# Профилирование. Файл конфигурации (CUDA 2.3)

- \* gld\_incoherent
- \* gld\_coherent
- \* gld\_32b / gst\_32b
- \* gld\_64b / gst\_62b
- \* gld\_128b / gst\_128b
- \* gld\_request
- \* gst\_incoherent
- \* gst\_coherent
- \* gst\_request
- \* local\_load
- \* local\_store
- \* branch
- \* divergent\_branch
- \* instructions
- \* warp\_serialize
- \* cta\_launched
- \* gputime
- \* cputime
- \* occupancy





# Computerprof (CUDA 4.0)

- \* Пересмотрен интерфейс пользователя
- \* Добавлено
  - \* Аналитика и рекомендации по коду
  - \* Определение ограничивающих факторов

convolutionColumnsKernel analysis - [Session4 - Device\_0 - Context\_0]

File View

Analysis

### Instruction Throughput Analysis for kernel convolutionColumnsKernel on device GeForce GTX 480

- IPC: 1.56
- Maximum IPC: 2
- Divergent branches(%): 0.00
- Control flow divergence(%): 0.03
- Replayed Instructions(%): 29.65
  - Global memory replay(%): 0.00
  - Local memory replays(%): 0.00
  - Shared bank conflict replay(%): 26.38
- Shared memory bank conflict per shared memory instruction(%): 99.90

### Hint(s)

- **The kernel is compute bound**, to reduce instruction count
  - Understand the instruction mix, as single precision floating point, double precision floating point, int, mem, transcendentals, etc. have different throughputs. Use double precision arithmetic only when required (E.g. floating point literals without an f suffix ( 34.7 ) are interpreted as double precision as per C standard);
  - Try using arithmetic intrinsic functions.
  - Try using compiler flags (-ftz=true, -prec-div=false, -prec-sqrt=false etc) to get higher performance, but may result in some precision loss;Refer to the "Arithmetic Instructions" section in the "Performance Guidelines" chapter of the CUDA C Programming Guide for more details.
- **Shared memory bank conflicts are high** which causes serialization of threads within a warp. Shared memory bank conflicts can be reduced by
  - Using appropriate padding for data stored in shared memory so that each thread in a warp accesses data from a different bank;
  - Rearranging data in shared memory, thus changing access pattern;Refer to the "Shared Memory" section in the "Performance Guidelines" chapter of the CUDA C Programming Guide for more details.

### Factors that may affect analysis

Show all columns

Limiting Factor Identification	GPU Timestamp (us)	GPU Time (us)	shared load Type:SM Run:4	shared store Type:SM Run:4
Memory Throughput Analysis	1 38718	1652.96	334560	24600
	2 41989.6	1652.86	334560	24600
Instruction throughput Analysis	3 44507.4	1652.93	334560	24600
	4 47024.9	1652.96	334560	24600
Occupancy Analysis	5 49541.9	1653.09	334560	24600

# NVVP (CUDA 4.1)

The screenshot displays the NVIDIA Visual Profiler (NVVP) interface for CUDA 4.1. The main window shows a performance analysis of a kernel named `swlon_do(double*, double*, double*, double*, double*)`. The timeline view shows the execution of various operations, including memory copies and compute. The properties panel on the right provides details for the selected kernel, including its start time, duration, grid size, block size, registers per thread, shared memory per block, and occupancy.

**Properties Panel:**

Name	Value
Start	4.801 s
Duration	13.181 ms
Grid Size	[ 162,180,1 ]
Block Size	[ 16,16,1 ]
Registers/Thread	45
Shared Memory/Block	11.25 KB
Occupancy	
Theoretical	16.7%

**Analysis Results:**

- Low Compute Utilization [ 4.412 s / 10.653 s = 41.4% ]**  
The multiprocessors of one or more GPUs are mostly idle. [More...](#)
- Low Memcpy/Compute Overlap [ 0 ns / 1.888 s = 0% ]**  
The percentage of time when memcpy is being performed in parallel with compute is low. [More...](#)
- Low Memcpy Throughput [ 1.56 GB/s avg, for memcpys accounting for 99.9% of all memcpy time ]**  
The memory copies are not fully using the available host to device bandwidth. [More...](#)
- Low Memcpy Overlap [ 0 ns / 188.41 ms = 0% ]**  
The percentage of time when two memory copies are being performed in parallel is low. [More...](#)

# NVIDIA Parallel Nsight

The screenshot shows the NVIDIA Parallel Nsight interface within Microsoft Visual Studio. The main window displays the source code for a CUDA kernel, `matrixMul_kernel.cu`. The code includes comments and C++ code for a matrix multiplication kernel. A dialog box titled "NVIDIA Parallel Nsight - CUDA Focus Picker" is open, showing the dimensions for a block (4, 0, 0) and thread (14, 0, 0). The "Locals" window shows the current state of variables, including `blockIdx`, `blockDim`, `gridDim`, `a`, `b`, `bx`, `by`, `bx`, `by`, `ty`, `aBegin`, `aEnd`, `aStep`, `bBegin`, `bStep`, `Csub`, `c`, `C`, `A`, `B`, `wA`, and `wB`.

```
// Step size used to iterate through the sub-matrices of A
int aStep = BLOCK_SIZE;

// Index of the first sub-matrix of B processed by the block
int bBegin = BLOCK_SIZE * bx;

// Step size used to iterate through the sub-matrices of B
int bStep = BLOCK_SIZE * wB;

// Csub is used to store the element of the block sub-matrix
// that is computed by the thread
float Csub = 0;

// Loop over all the sub-matrices of A and B
// required to compute the block sub-matrix
for (int a = aBegin; a <= aEnd; a += aStep, b += bStep) {

    // Declaration of the shared memory array A
    // store the sub-matrix of A
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
```

**NVIDIA Parallel Nsight - CUDA Focus Picker**

Dimensions

Block: 4, 0, 0    8, 5, 1

Thread: 14, 0, 0    16, 16, 1

Examples

- #129 for block index 129  
10 for coordinates 10, 0
- 10, 5 for coordinates 10, 5

OK    Cancel

**Locals**

Name	Value	Type
blockIdx	{x = 4, y = 0, z = 0}	const uint
blockDim	{x = 16, y = 16, z = 1}	const dim
gridDim	{x = 8, y = 5, z = 1}	const dim
a	???	int
b	???	int
bx	4	int
by	0	int
bx	14	int
by	0	int
ty	0	int
aBegin	0	int
aEnd	47	int
aStep	16	int
bBegin	64	int
bStep	2048	int
Csub	0	float
c	???	int
C	0x00119c00 0	__device_
A	0x00110000 0.20108646	__device_
B	0x00113c00 0.80645162	__device_
wA	48	__shared
wB	128	__shared

# NSight Eclipse Edition

The screenshot displays the NSight Eclipse Edition interface for analyzing a CUDA application. The main window shows a timeline from 0.045 s to 0.06 s. The process is identified as 31954, and the thread is -1314150624. The GPU is GeForce GTX 480, and the context is Context 1 (CUDA). The analysis shows various memory copy operations (MemCpy) and compute kernels. A specific kernel, `VecThen(int*, int*, int*, int)`, is highlighted in the timeline and its properties are shown in the right-hand panel.

**Kernel Properties: `VecThen(int*, int*, int*, int)`**

Name	Value
Start	51.274 ms
End	53.615 ms
Duration	2.342 ms
Grid Size	[ 256,1,1 ]
Block Size	[ 256,1,1 ]
Registers/Thread	11
Shared Memory/Block	0 bytes
<b>Memory</b>	
Global Load Efficiency	99.1%
Global Store Efficiency	100%
<b>Occupancy</b>	
Theoretical	100%
<b>L1 Cache Configuration</b>	
Shared Memory Requested	48 KB
Shared Memory Executed	48 KB

**Analysis Results:**

- Low Global Memory Load Efficiency [ 9% avg, for kernels accounting for 75.6% of compute ]**  
Global memory loads may have a poor access pattern, leading to inefficient use of global memory bandwidth. [More...](#)
- Low Global Memory Store Efficiency [ 21.3% avg, for kernels accounting for 73.9% of compute ]**  
Global memory stores may have a poor access pattern, leading to inefficient use of global memory bandwidth. [More...](#)

The interface also includes a console, analysis scope (Analyze Entire Application), and stages (Timeline, Multiprocessor, Kernel Memory).

# Анализ отчета о профилировании

- \* Значение имеет не цифры, а их приращение и отношение.
- \* Для ядер надо стремиться чтобы стремились к нулю  
непоследовательное обращение к памяти  
(gld\_incoherent, gld\_coherent, gst\_incoherent,  
gst\_coherent)

# Арифметика или память

- \* Оптимальное соотношение инструкции:байты для Tesla C2050:
  - \*  $\sim 3.6 : 1$ , float, при включенном ECC
  - \*  $\sim 4.5 : 1$ , float, при выключенном ECC
- \* Использование счетчиков
  - \*  $32 * \text{instructions\_issued}$  (+1 на варп)
  - \*  $128B * (\text{global\_store\_transaction} + \text{l1\_global\_load\_miss})$  (+1 на одну линию L1)
- \* В версии 4.0 и выше ограничитель определяется автоматически

# Анализ инструкций

- \* Счетчики профилировщика (на варп)
  - \* `instructions executed`: сколько инструкций исполнилось
  - \* `instructions issued`: включая сериализацию
- \* Разница между ними –проблемы с сериализацией, промахи кэша
- \* Сравниваем с характеристиками устройства
  - \* См. максимум в Programming Guide или Visual Profiler
  - \* Fermi: IPC (инструкций в такт)

# Сериализация

- \* Дивергенция варпов
  - \* Счетчики: `divergent_branch`, `branch`
  - \* Определяем сколько процентов дивергентных
- \* Конфликты банков разделяемой памяти
  - \* Счетчики
    - \* `l1_shared_bank_conflict`,
    - \* `shared_load`, `shared_store`
- \* Конфликты банков существенны, если оба условия выполнены
  - \* `l1_shared_bank_conflict >> (shared_load+ shared_store)`
  - \* `l1_shared_bank_conflict >> instructions_issued`
- \* Автоматический анализ в 4.0 и выше

# Спиллинг регистров

- \* Когда достигнут предел доступных регистров, компилятор начинает использовать локальную память (спиллинг)
  - \* На Fermi предел 63 регистра
  - \* Предел можно указать вручную
  - \* Локальная память работает как глобальная, только запись кэшируется в L1
    - \* Попадание в L1 – почти бесплатно
    - \* Промисс в L1 – запрос из глобальной, 128B за промах
  - \* Флаг `-ptxas-options=-v` показывает использование регистров и локальной памяти на одну нить
- \* Возможное влияние на производительность
  - \* Дополнительный трафик через шину памяти
  - \* Дополнительные инструкции
  - \* Не всегда проблема

# Спиллинг регистров

- \* Счетчики: `l1_local_load_hit`, `l1_local_load_miss`
  - \* Влияние на число инструкций
  - \* Сравнить с общим числом инструкций
- \* Влияние на пропускную способность памяти
  - \* Промахи добавляют 128В за варп
  - \* Сравнить  $2 * l1\_local\_load\_miss$  с обращениями к глобальной памяти (чтение + запись)
    - \* Умножаем на 2, т.к. каждый промах вытесняет кэш линию – дополнительная запись через шину
    - \* Если кэш L1 включен – сравниваем с промахами L1
    - \* Если кэш L1 выключен – сравниваем со всеми чтениями

# CUDA Occupancy calculator

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

1.) Select Compute Capability (click): 1,2

2.) Enter your resource usage:

Threads Per Block	256
Registers Per Thread	18
Shared Memory Per Block (bytes)	512

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:

Active Threads per Multiprocessor	768
Active Warps per Multiprocessor	24
Active Thread Blocks per Multiprocessor	3
Occupancy of each Multiprocessor	75%

Physical Limits for GPU:

Threads / Warp
Warps / Multiprocessor
Threads / Multiprocessor
Thread Blocks / Multiprocessor
Total # of 32-bit registers / Multiprocessor
Shared Memory / Multiprocessor (bytes)

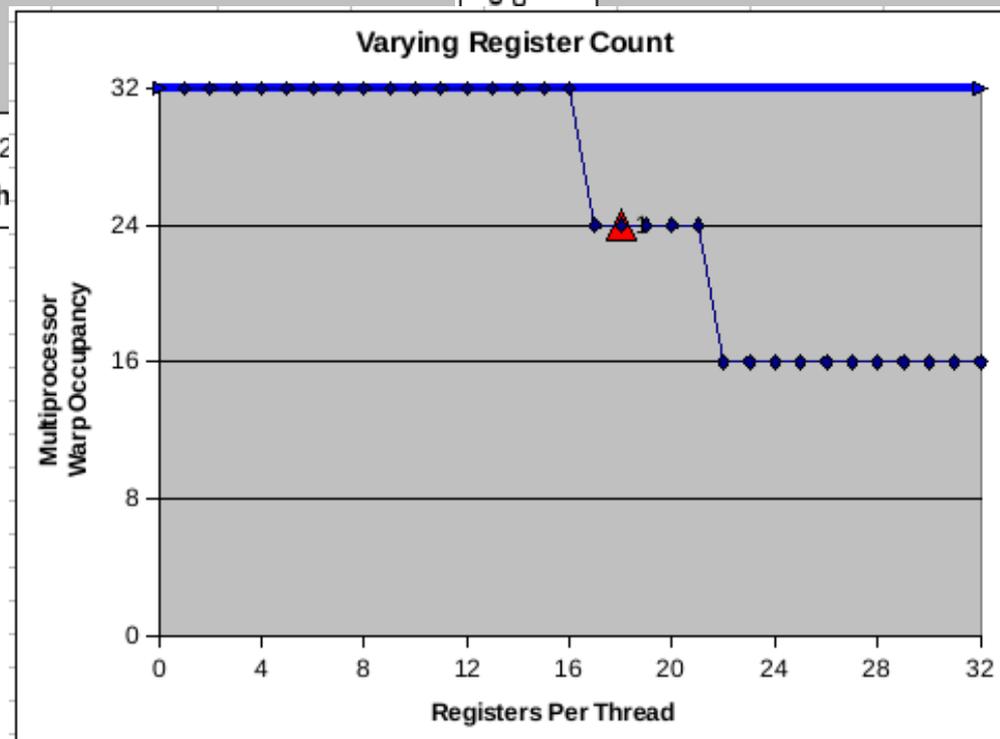
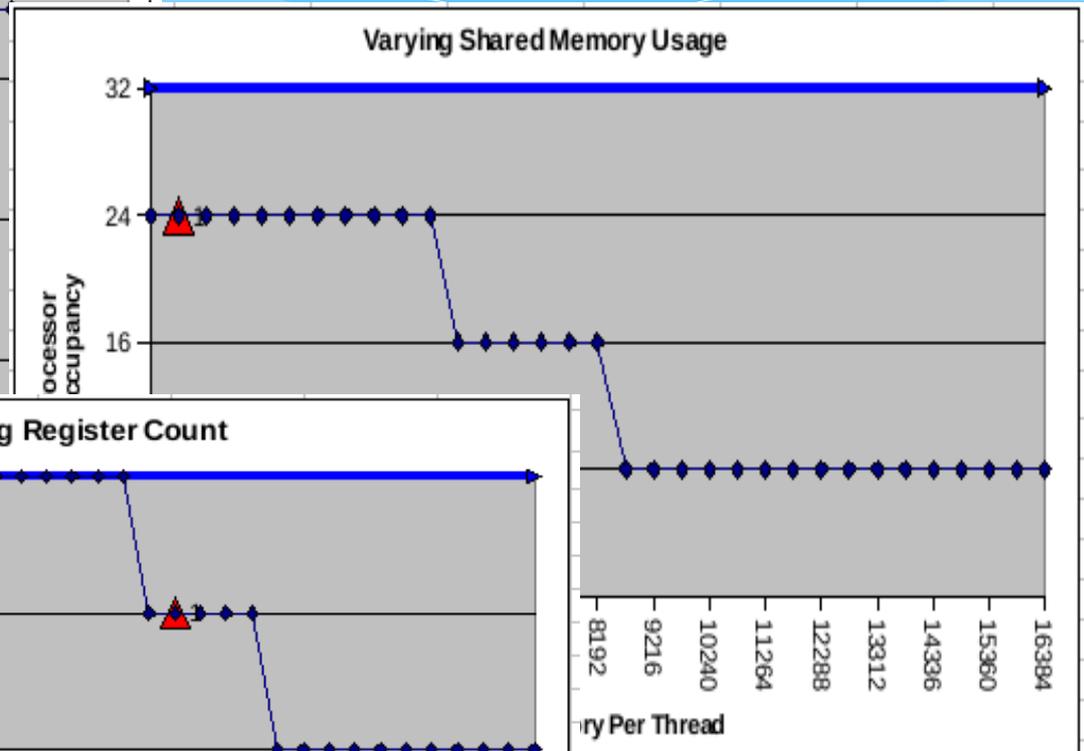
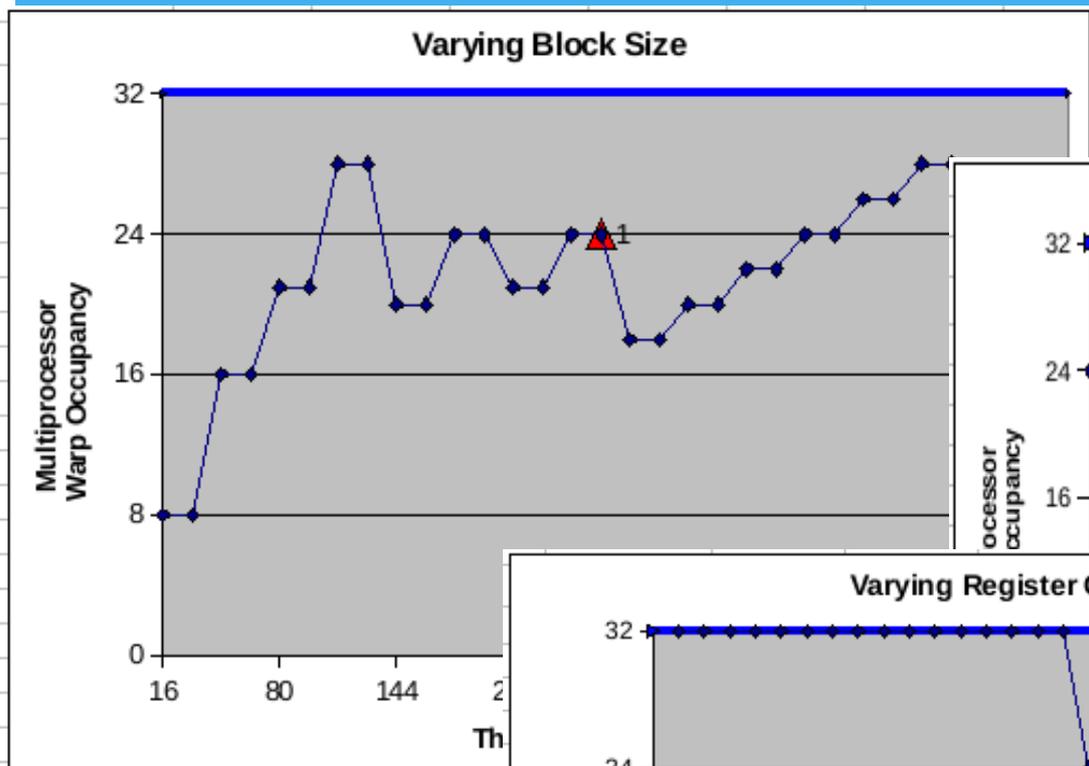
Allocation Per Thread Block

Warps
Registers
Shared Memory

These data are used in computing the occupancy

<u>Maximum Thread Blocks Per Multiprocessor</u>	<u>Blocks</u>
Limited by Max Warps / Multiprocessor	4
Limited by Registers / Multiprocessor	3
Limited by Shared Memory / Multiprocessor	32
Thread Block Limit Per Multiprocessor highlighted	RED

# CUDA Occupancy calculator



# Оптимизация

- \* Обработка инструкций
  - \* Чтение операндов
  - \* Выполнение инструкции
  - \* Сохранение результата
- \* Для оптимизации
  - \* Использовать более быстрые инструкции
  - \* Минимизировать задержки на обращение к памяти
  - \* По максимуму использовать пропускную способность шин данных

# Исполнение инструкций

- \* Арифметические операции
  - \* 4 такта для FMUL, FADD, FMAD IADD, бинарные операции, сравнение, MIN, MAX, приведение типов
  - \* 16 тактов для `__log`, `1/sqrt`, IMUL, `1/(float)x`
  - \* 32 такта для `sqrt`, `__sin`, `__cos`, `__exp`
  - \* 36 такта для FDIV
  - \* 20 тактов для `__fdivdef(x, y)`
- \* Дополнительные опции компилятора:
  - \* `-ftz=true`
  - \* `-prec-div=false`
  - \* `-prec-sqrt=false`

# Условные переходы

- \* Если в варпе есть две ветви исполнения (условный переход), то сначала исполняются потоки, которые проходят одну ветвь, затем потоки, которые проходят вторую.
- \* Минимизировать количество ветвлений. В частности внутри варпа. Например за счет предвычислений или переупорядочивания нитей.

# Доступ к памяти

- \* 4 такта на обработку одной инструкции по работе с памятью (разделяемая, константная\*, текстуры\*).
- \* 400-600 тактов задержка по доступу к глобальной памяти
- \* Метод работы:
  - \* Загрузить данные из глобальной памяти в разделяемую (через текстуры)
  - \* Обработать данные
  - \* Выгрузить в глобальную

\* - если нет промахов по кэшу

# Доступ к памяти

- \* Используйте `cudaMallocPitch` вместо `cudaMalloc`, если двумерный массив
- \* Используйте `cudaMallocArray` для 2D и 3D массивов
- \* Используйте `page-locked` память
  - \* `cudaHostAlloc()`, `cudaFreeHost()`, `cudaHostRegister()`,
- \* Используйте текстуры
- \* Используйте поверхности (CUDA 4.0)

# L1 –кэширование и размер (Fermi)

- \* Два варианта:
  - \* L1 включен
    - \* По-умолчанию (опция -Xptxas -dlcm=ca)
    - \* Размер транзакции с памятью 128B
  - \* L1 выключен
    - \* Опция -Xptxas -dlcm=cg
    - \* Размер транзакции с памятью 32B
- \* Выбора размера L1/SMEM
  - \* 16KB L1, 48KB SMEM или 48KB L1, 16KB SMEM
- \* Вызов CUDA
- \* Как использовать
  - \* Попробовать все 4 варианта (CA, CG) x(16, 48)

# Параллельное исполнение ядер

- \* Возможно если:
  - \* Устройство с вычислительными возможностями (computer compatibility) выше или равно 2.0
  - \* Свойство concurrentKernels устройства = 1
  - \* Ядра из одного контекста
- \* Максимальное количество параллельно исполняемых ядер 16

# Передача данных на/с устройства

- \* Скорость копирования на устройстве выше, чем между Хостом и Устройством
- \* Рекомендуется не выгружать данные, а запустить ядро с малым уровнем параллелизма, если это возможно.
- \* На Хосте выделять память с помощью `cudaMallocHost()`
  - \* Page-locked memory
- \* Совмещать передачу данных с вычислениями.
- \* Совмещать передачу с устройства с передачей данных на устройство.

# Параллельная передача данных

- \* Одновременное двунаправленное копирование данных возможно если:
- \* Используется page-locked память
- \* Устройство с вычислительными возможностями (computer compatibility) выше или равно 2.0
- \* Свойство **asyncEngineCount** устройства = 2

# Копирование данных между GPU

## CUDA 3.2

```
cudaMemcpy(Host, GPU1);  
cudaMemcpy(GPU2, Host);
```

## CUDA 4.0

```
cudaMemcpy(GPU1, GPU2);
```

Можно как читать так и  
писать в память.

Поддерживается только  
на Tesla 20xx (Fermi)

64-битные приложения

# Совмещение передачи данных с вычислениями

- \* Возможно если:
  - \* Устройство с вычислительными возможностями (computer compatibility) выше 1.1
  - \* Свойство `asyncEngineCount` устройства  $> 0$
- \* Не поддерживается, если в копирование вовлечены массивы (CUDA Arrays) или 2D массивы, выделенные через `cudaMallocPitch`
- \* Может блокироваться переменной окружения `CUDA_LAUNCH_BLOCKING`, установленной в 1

# Диспетчеризация потоков

- \* **Fermi имеет 3 очереди исполнения**
  - \* 1 вычислительная
  - \* 2 очереди копирования – одна для H2D и одна для D2H
- \* **Операции CUDA отправляются на исполнение в той последовательности в которой объявлены**
  - \* Помещаются в соответствующие очереди
  - \* Операции в одном потоке зависимы и независимы между очередями
- \* **CUDA операция ставится на выполнение в очередь если:**
  - \* Предшествующая операция в той же очереди завершена,
  - \* Завершена операция в соответствующей очереди исполнения
  - \* Есть ресурсы
- \* **CUDA ядра исполняются параллельно только из разных потоков**
  - \* Поставленные последовательно в очередь ядра блокируют другие очереди

# Пример 1

```
for (int i = 0; i < 3; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size,
                    hostPtr + i * size, size,
                    cudaMemcpyHostToDevice, stream[i]);

for (int i = 0; i < 3; ++i)
    MyKernel<<<size/512, 512, 0, stream[i]>>>
        (outputDevPtr + i * size,
         inputDevPtr + i * size, size);

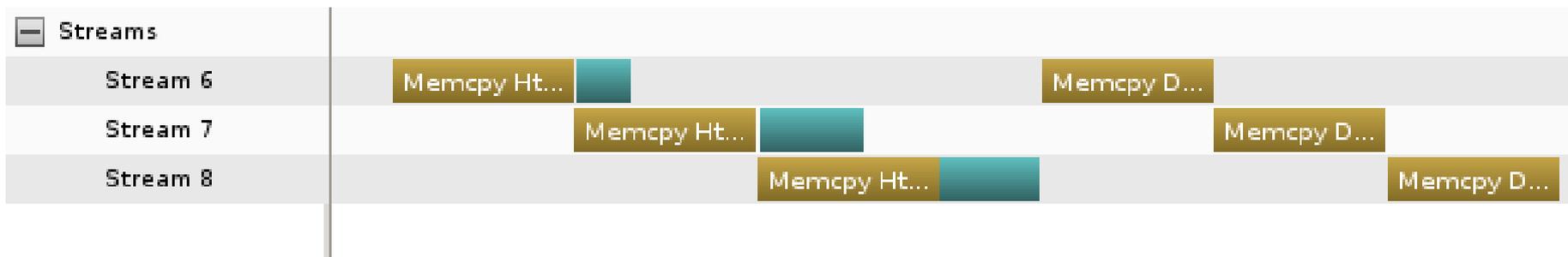
for (int i = 0; i < 3; ++i)
    cudaMemcpyAsync(hostPtr + i * size,
                    outputDevPtr + i * size, size,
                    cudaMemcpyDeviceToHost, stream[i]);
```

# Пример 2

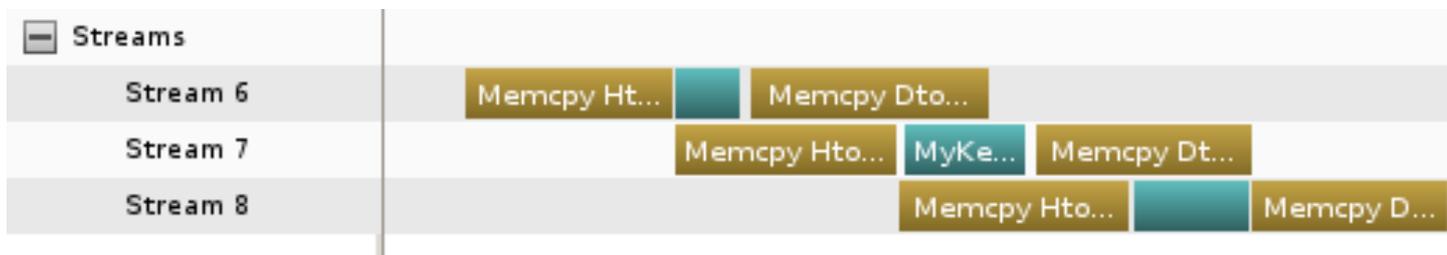
```
for(int i=0; i < 3; ++i) {  
    cudaMemcpyAsync (inputDevPtr + i * size,  
                    hostPtr + i * size, size,  
                    cudaMemcpyHostToDevice, stream[i]);  
  
    MyKernel<<<size/512, 512, 0, stream[i]>>>  
        (outputDevPtr + i * size,  
         inputDevPtr + i * size, size);  
  
    cudaMemcpyAsync (hostPtr + i * size,  
                    outputDevPtr + i * size, size,  
                    cudaMemcpyDeviceToHost, stream[i]);  
}
```

# Timeline

## Пример 1



## Пример 2



# Вычислительная совместимость

## Compute capability

- \* Compute Capability 1.0+
  - \* Асинхронный запуск ядер
- \* Compute Capability 1.1+ ( например, C1060, cc1.3 )
  - \* Добавлена поддержка асинхронного копирования (одно устройство). Свойство `asyncEngineCount`
- \* Compute Capability 2.0+ ( например, C2050 )
  - \* Добавлена возможность параллельного исполнения ядер на GPU (свойство `concurrentKernels`)
  - \* Добавлено второе устройство для двунаправленного асинхронного копирования (свойство `asyncEngineCount`)

# Спиллинг регистров

- \* Увеличить максимальный предел регистров

- \* Использовать `__launch_bounds__`

```
__global__ void  
__launch_bounds__(maxThreadsPerBlock, minBlocksPerMS)  
MyKernel(...){...}
```

- \* Скорей всего уменьшит загруженность, запросы в глобальную память будут менее эффективными
    - \* Но может быть лучше в целом
- \* Выключить L1 для запросов в глобальную память
- \* Меньше коллизий с кэшированием локальной памяти
- \* Увеличить размер L1 до 48KB

# Спиллинг регистров

```
arom@cuda:~/cuda/edison$ /usr/local/cuda/bin/nvcc -
gencode=arch=compute_20,code=\"sm_20,compute_20\" -m32 --compiler-
options -fno-strict-aliasing -I. -I /usr/local/cuda/include -I
/usr/local/cudasdk/C/common/inc -I/usr/local/cudasdk/shared/inc -
DUNIX -O2 -g --ptxas-options=-v,-abi=no -c most_cuda.cu -
maxrregcount=32
```

```
ptxas info      : Compiling entry function
'_Z8swlon_doPdS_S_S_S_S_jiiddi' for 'sm_20'
```

```
ptxas info      : Used 32 registers, 52+0 bytes lmem, 11520+0 bytes
smem, 92 bytes cmem[0], 32 bytes cmem[14], 48 bytes cmem[16]
```

```
ptxas info      : Compiling entry function
'_Z4Inv1PdS_S_S_S_S_jiid' for 'sm_20'
```

```
ptxas info      : Used 19 registers, 80 bytes cmem[0], 32 bytes
cmem[14], 16 bytes cmem[16]
```

# Оптимизация

- \* Количество регистров на поток и разделяемой памяти на блок
  - \* `--ptxas-options=-v`
  - \* CUDA Occupancy calculator