

# CUDA

## Оптимизация программ

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

# 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

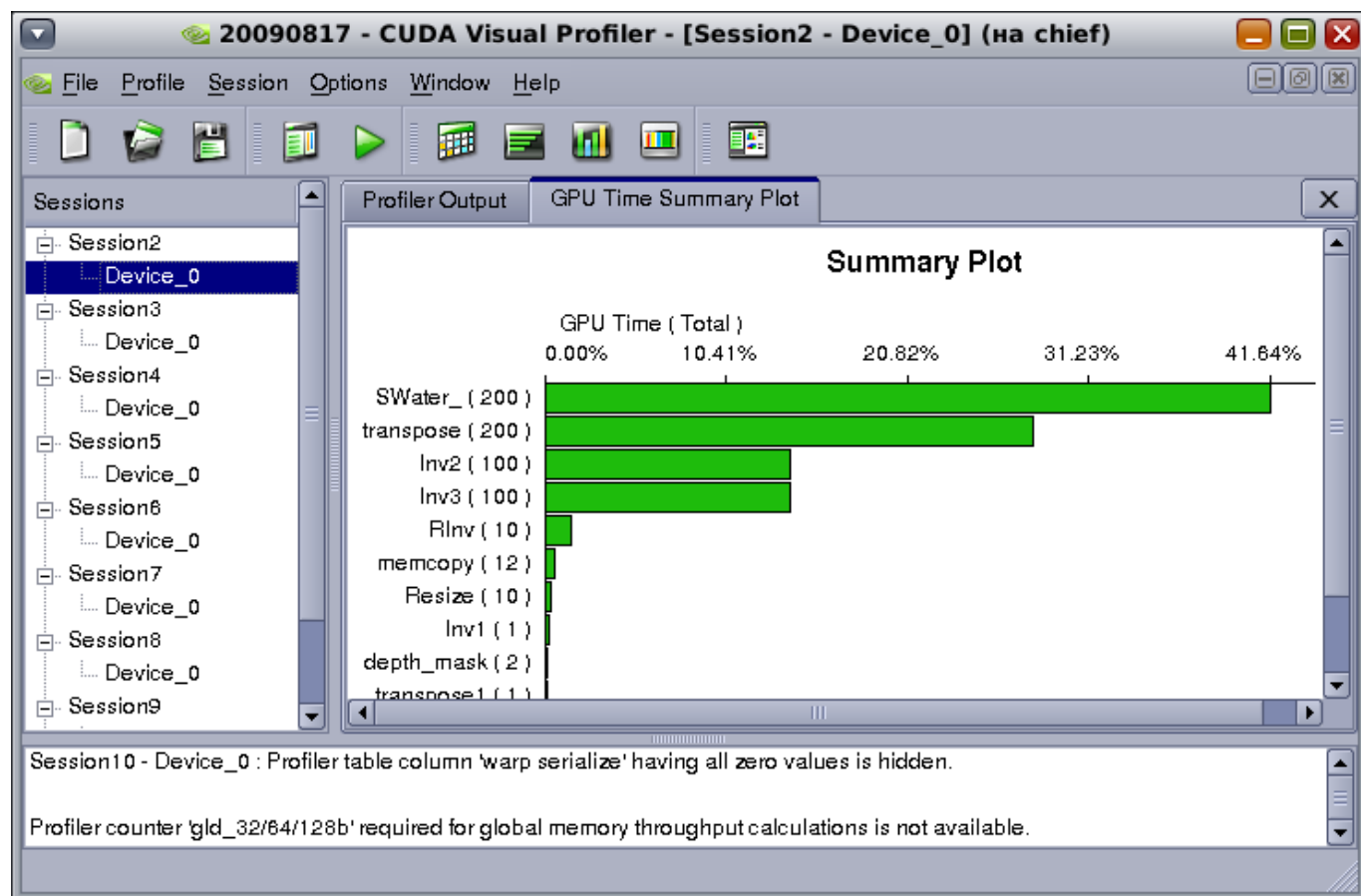
# Профилирование. Файл конфигурации (CUDA 2.3)

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

# Профилирование. Вывод

```
method,gputime,cputime,occupancy
method=[ memcopy ] gputime=[ 5519.680 ]
method=[ memcopy ] gputime=[ 5516.992 ]
method=[ memcopy ] gputime=[ 5517.728 ]
method=[ memcopy ] gputime=[ 5508.288 ]
method=[ __globfunc__Z10depth_maskPfS_fii ] gputime=[ 1791.424 ] cputime=[ 14.000 ]
        occupancy=[ 0.812 ]
method=[ __globfunc__Z10depth_maskPfS_fii ] gputime=[ 1769.920 ] cputime=[ 2.000 ]
        occupancy=[ 0.812 ]
method=[ __globfunc__Z9transposePfS_ii ] gputime=[ 1598.336 ] cputime=[ 2.000 ] occupancy=[ 0.812 ]
method=[ __globfunc__Z12Invariants_XPfS_S_S_S_S_S_ii ] gputime=[ 5061.920 ] cputime=[ 3.000 ]
        occupancy=[ 0.812 ]
method=[ __globfunc__Z7SWater_PfS_S_S_S_S_fS_S_S_ii ] gputime=[ 10506.752 ] cputime=[ 2.000 ]
        occupancy=[ 0.406 ]
method=[ __globfunc__Z9transposePfS_ii ] gputime=[ 1599.776 ] cputime=[ 2.000 ] occupancy=[ 0.812 ]
method=[ __globfunc__Z9transposePfS_ii ] gputime=[ 1613.472 ] cputime=[ 1.000 ] occupancy=[ 0.812 ]
method=[ __globfunc__Z9transposePfS_ii ] gputime=[ 1609.056 ] cputime=[ 2.000 ] occupancy=[ 0.812 ]
method=[ __globfunc__Z7SWater_PfS_S_S_S_S_fS_S_S_ii ] gputime=[ 10758.784 ] cputime=[ 2.000 ]
        occupancy=[ 0.406 ]
method=[ __globfunc__Z13RInvariants_YPfS_S_S_S_S_S_ii ] gputime=[ 5423.456 ] cputime=[ 3.000 ]
        occupancy=[ 0.812 ]
```

# Профилировщик cuda prof



on2 - Device\_0] (на chief)

GPU Time Summary Plot

GPU usec	%GPU time	instruction throughput
3,58836e+06	41,63	0,197939
5,7661e+06	27,95	0,0736237
2,88783e+06	14	0,0434338
2,88782e+06	14	0,0434348
288737	1,39	0,0422327
15595,9	0,22	0,0483523
7 Inv1	0,13	0,0452789
8 depth_mask	0,1	0,0946365

Profiler counter 'gld\_32/64/128b' required for global memory throughput calculations is not available.

Profiler counter 'gld\_32/64/128b' required for global memory throughput calculations is not available.



# 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



# 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:
  - ~3.6 : 1, float, при включенном ECC
  - ~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 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
  - $l1\_shared\_bank\_conflict \gg (shared\_load + shared\_store)$
  - $l1\_shared\_bank\_conflict \gg instructions\_issued$
- Автоматический анализ в 4.0

# 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
    - Промах в L1 – запрос из глобальной, 128В 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: `I1_local_load_hit`, `I1_local_load_miss`
- Влияние на число инструкций
  - Сравнить с общим числом инструкций
- Влияние на пропускную способность памяти
  - Промахи добавляют 128В за варп
  - Сравнить  $2 * I1\_local\_load\_miss$  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)

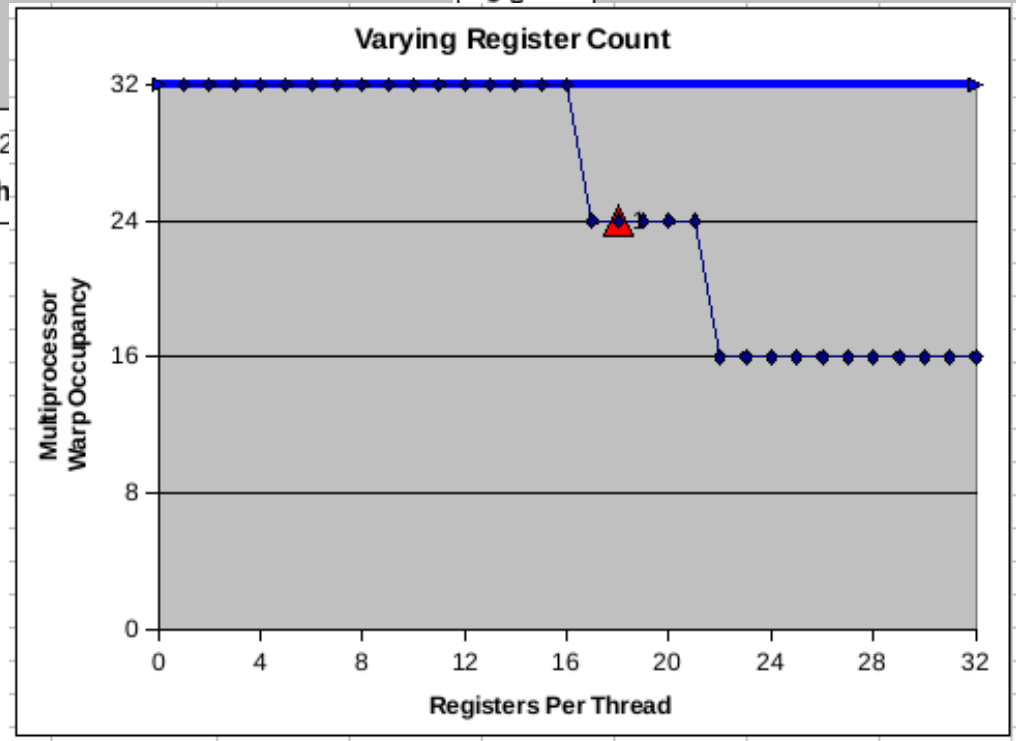
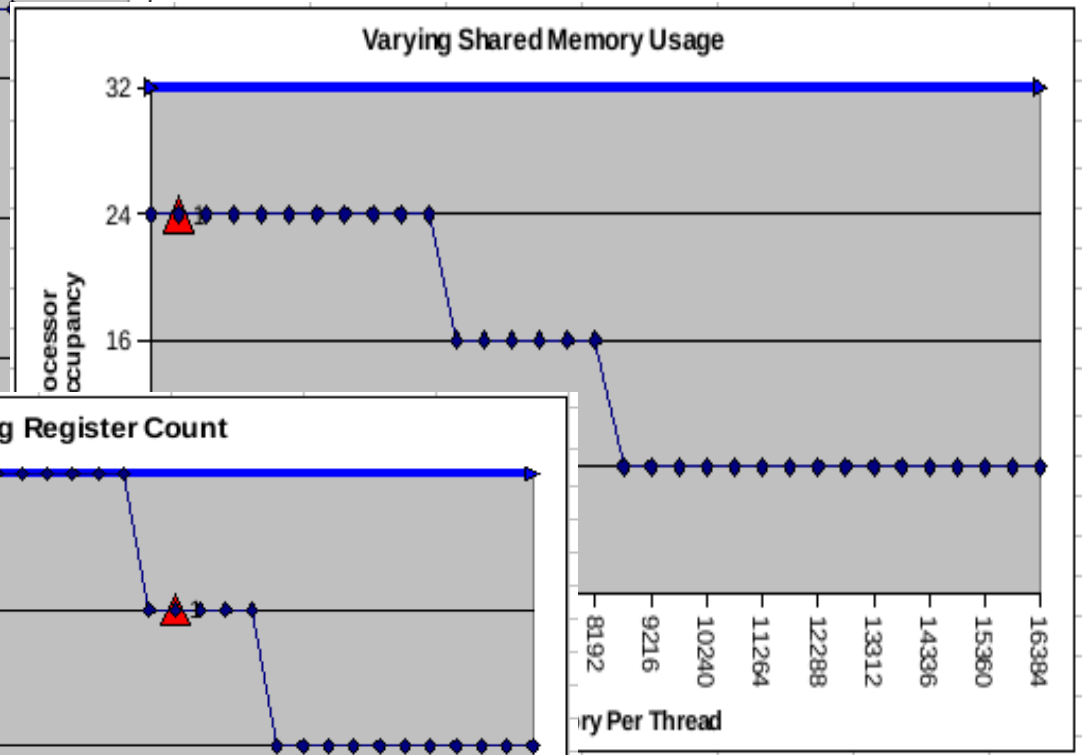
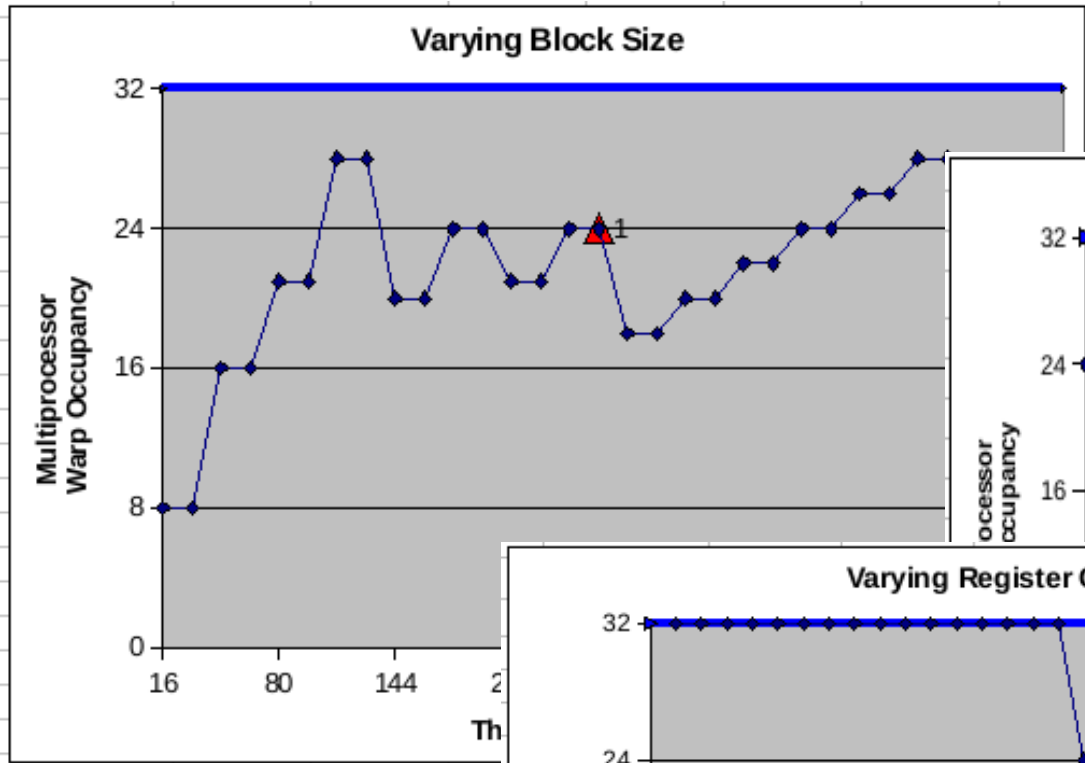
## Allocation Per Thread Block

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



# NVIDIA Parallel Nsight

The screenshot displays the Visual Studio IDE with the NVIDIA Parallel Nsight extension. The main window shows the source code for `matrixMul_kernel.cu`. A dialog box titled "NVIDIA Parallel Nsight - CUDA Focus Picker" is open, allowing the user to select a block and thread. The selected block is `4, 0, 0` and the thread is `14, 0, 0`. The "Examples" section provides instructions for selecting a block and thread for debugging.

The "NVIDIA Parallel Nsight - CUDA Focus Picker" dialog box contains the following information:

- Block:  Dimensions: 8, 5, 1
- Thread:  Dimensions: 16, 16, 1
- Examples:
  - #129 for block index 129
  - 10 for coordinates 10, 0
  - 10, 5 for coordinates 10, 5

The "NVIDIA Parallel Nsight - CUDA Device Summary" window shows the following details:

Name	Details
Devices	
Device 0	Device 0
Context 13188816	
Module 13238048	c:/ProgramData/NVIDIA Nsight 1.0/Samples/CUDA/Debugging/Matrix Multiply/matrixMul.cu
Grid 16	Module 13238048
Block 0 {0,0,0}	Warp Mask: 0x000000FF
Warp 0 {0,0,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixMul_kernel.cu:91
Warp 1 {0,2,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixMul_kernel.cu:91
Warp 2 {0,4,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixMul_kernel.cu:91
Warp 3 {0,6,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixMul_kernel.cu:91
Warp 4 {0,8,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixMul_kernel.cu:91
Warp 5 {0,10,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixMul_kernel.cu:91
Warp 6 {0,12,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixMul_kernel.cu:91
Warp 7 {0,14,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixMul_kernel.cu:91
Block 1 {1,0,0}	Warp Mask: 0x000000FF
Warp 0 {0,0,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixMul_kernel.cu:91
Warp 1 {0,2,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixMul_kernel.cu:91
Warp 2 {0,4,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixMul_kernel.cu:91
Warp 3 {0,6,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixMul_kernel.cu:91
Warp 4 {0,8,0}	Active Mask: 0xFFFFFFFF, PC: 0x000024B0, matrixMul_kernel.cu:91

The "Memory 1" window shows a memory dump for address `0x00113c00`. The "Locals" window shows the following variables:

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
tx	14	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_

# 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 `__fdividef(x, y)`
- Дополнительные опции компилятора:
  - `-ftz=true`
  - `-prec-div=false`
  - `-prec-sqrt=false`



# Условные переходы

- Если в ворпе 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 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 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
- При запуске `cuda-gdb`, CUDA Visual Profiler, Parallel Nsight исполнение синхронное

# Пример 1

```
for(int i=0; i < 2; ++i) {  
    cudaMemcpyAsync(inputDevPtr + i * size,  
                    hostPtr + i * size, size,  
                    cudaMemcpyHostToDevice, stream[i]);  
    MyKernel<<<100, 512, 0, stream[i]>>>  
        (outputDevPtr + i * size,  
         inputDevPtr + i * size, size);  
    cudaMemcpyAsync(hostPtr + i * size,  
                    outputDevPtr + i * size, size,  
                    cudaMemcpyDeviceToHost, stream[i]);  
}
```

# Пример 2

```
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size,
                    hostPtr + i * size, size,
                    cudaMemcpyHostToDevice, stream[i]);
```

```
for (int i = 0; i < 2; ++i)
    MyKernel<<<100, 512, 0, stream[i]>>>
        (outputDevPtr + i * size,
         inputDevPtr + i * size, size);
```

```
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size,
                    outputDevPtr + i * size, size,
                    cudaMemcpyDeviceToHost, stream[i]);
```

# Спиллинг регистров

- Увеличить максимальный предел регистров
  - Использовать `__launch_bounds__`

```
__global__ void  
__launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor)  
MyKernel(...){...}
```
  - Скорей всего уменьшит загрузенность, 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

# Оптимизация

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