CUDA
Оптимизация программ

Романенко А.А.
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

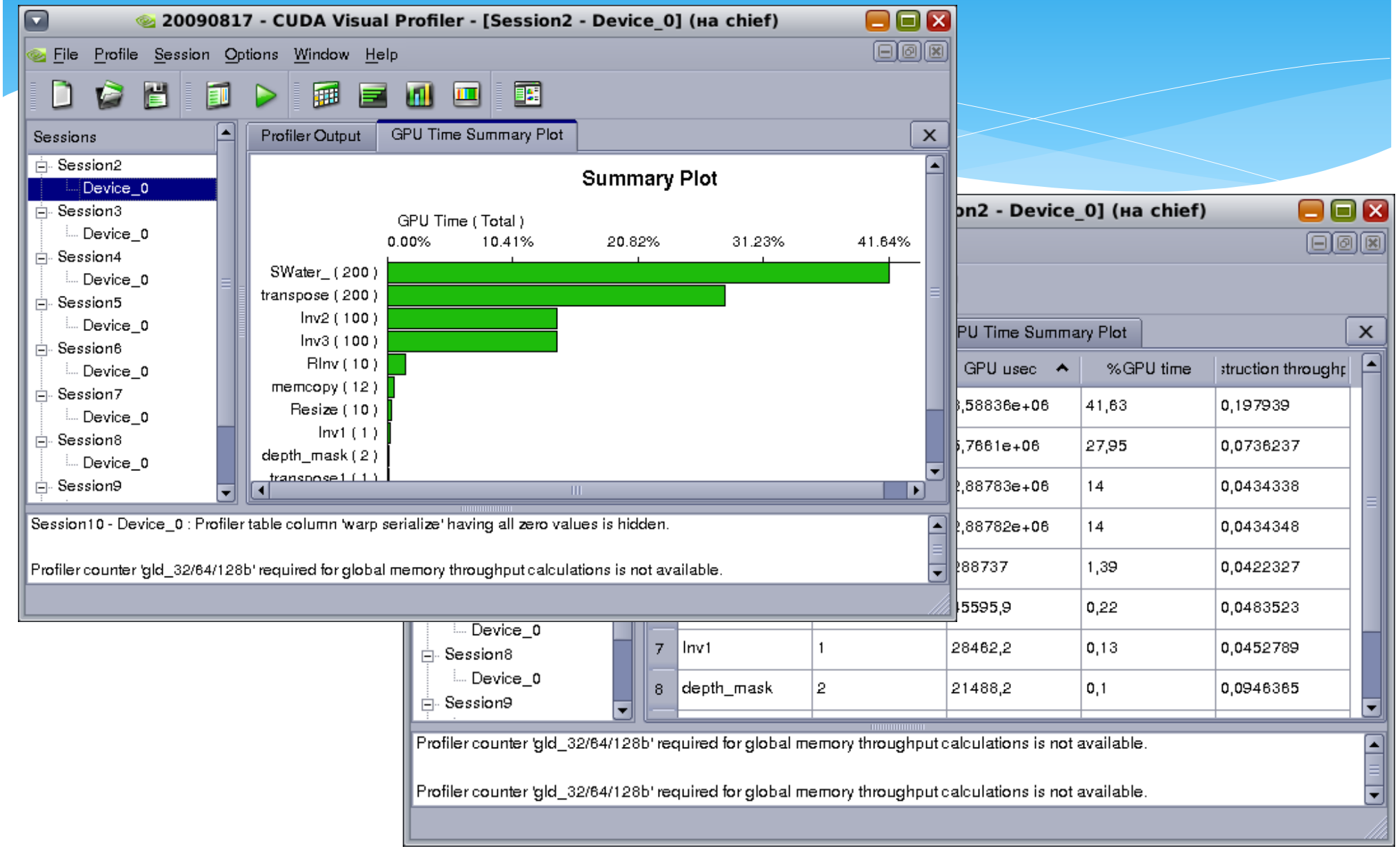
Профилирование. Файл конфигурации (CUDA 2.3)

* 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

Профилирование. Вывод

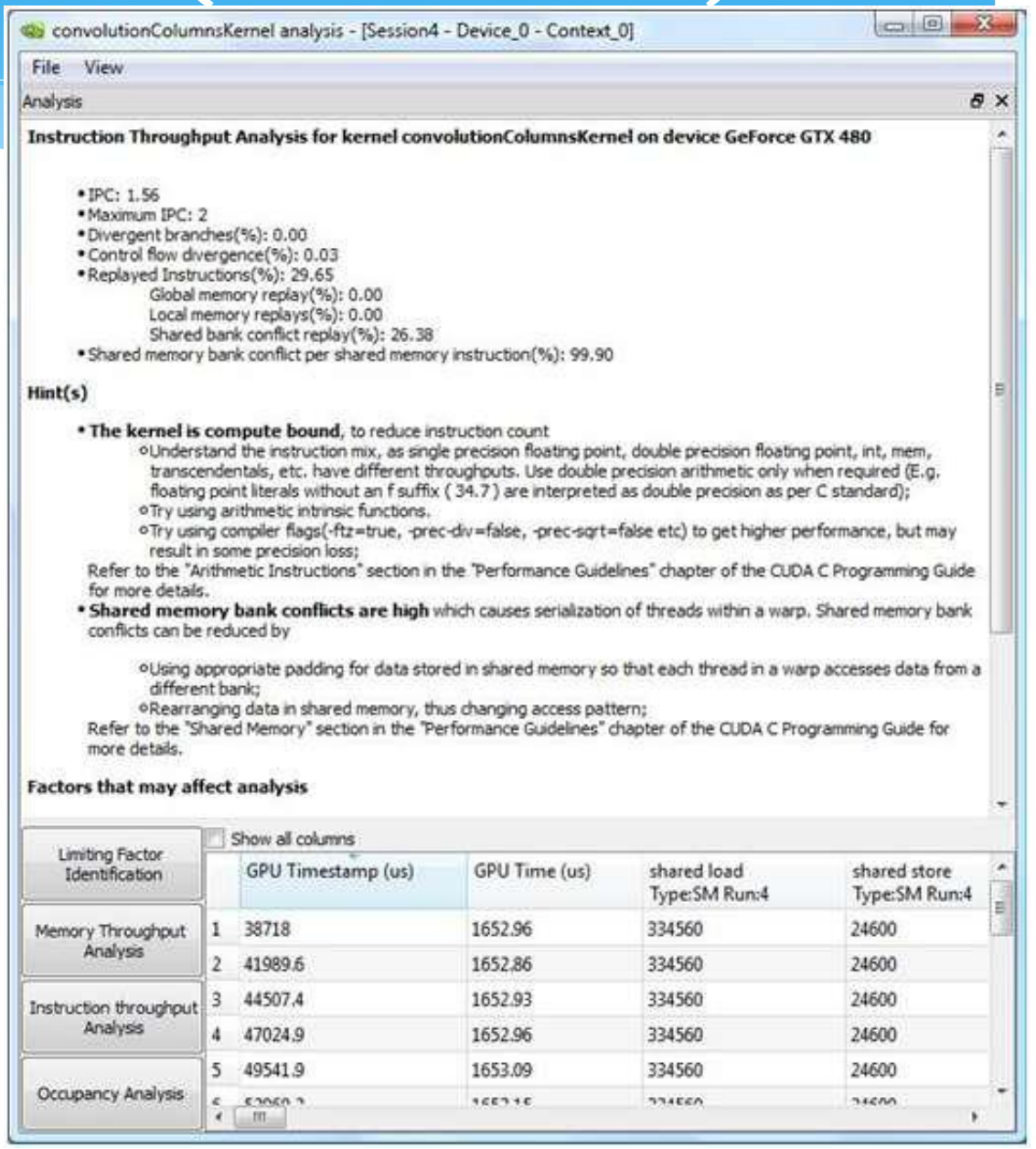
```
method,gputime,cputime,occupancy
method=[ memcopy ] gputime=[ 5519.680 ]
method=[ memcopy ] gputime=[ 5516.992 ]
method=[ memcopy ] gputime=[ 5517.728 ]
method=[ memcopy ] gputime=[ 5508.288 ]
method=[ __globfunc__Z10depth_maskPfS_fii ] gputime=[ 1791.424 ] cputime=[ 14.000 ]
                                                                  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 ]
```

Профилировщик cudaprof

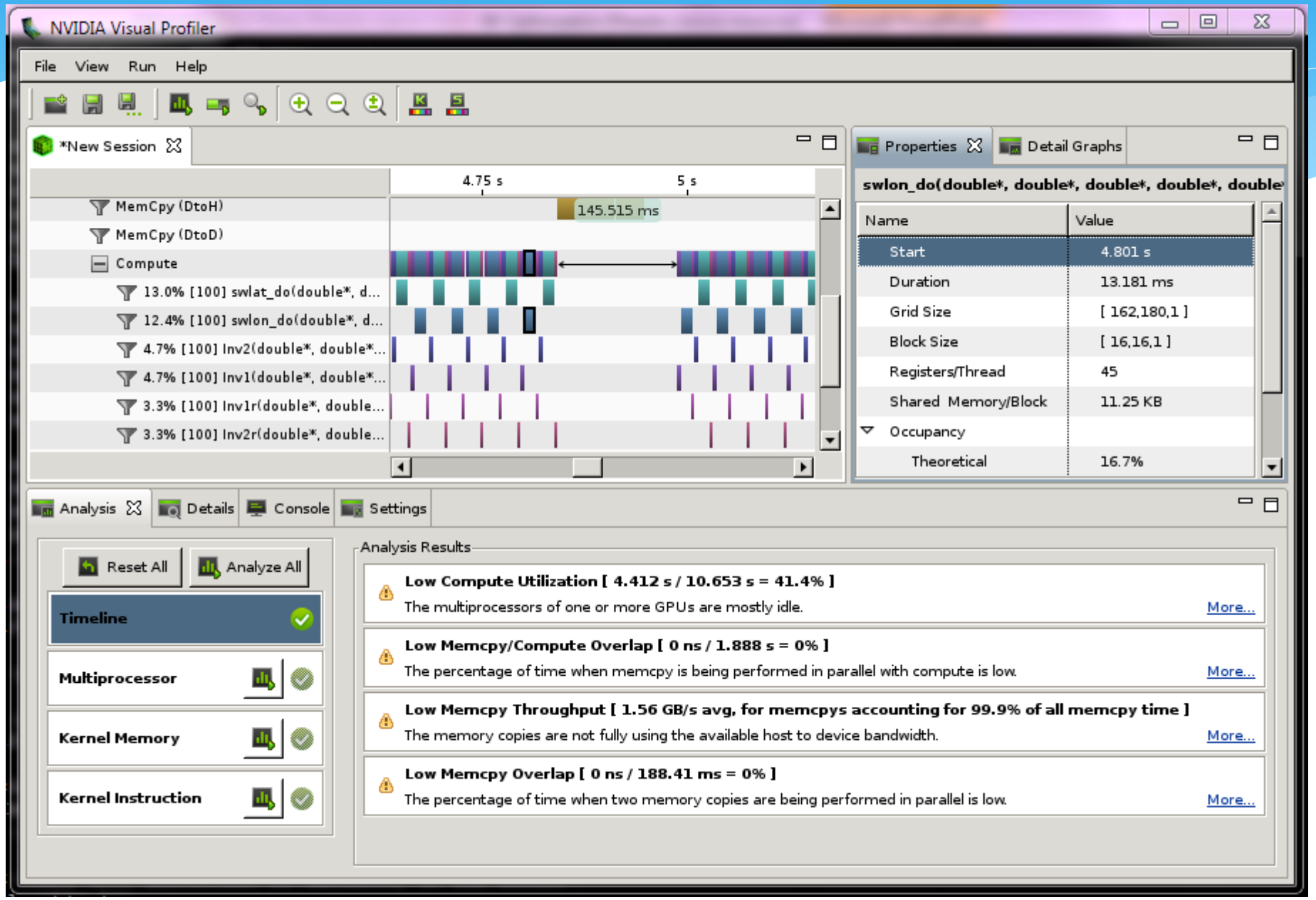


Computerprof (CUDA 4.0)

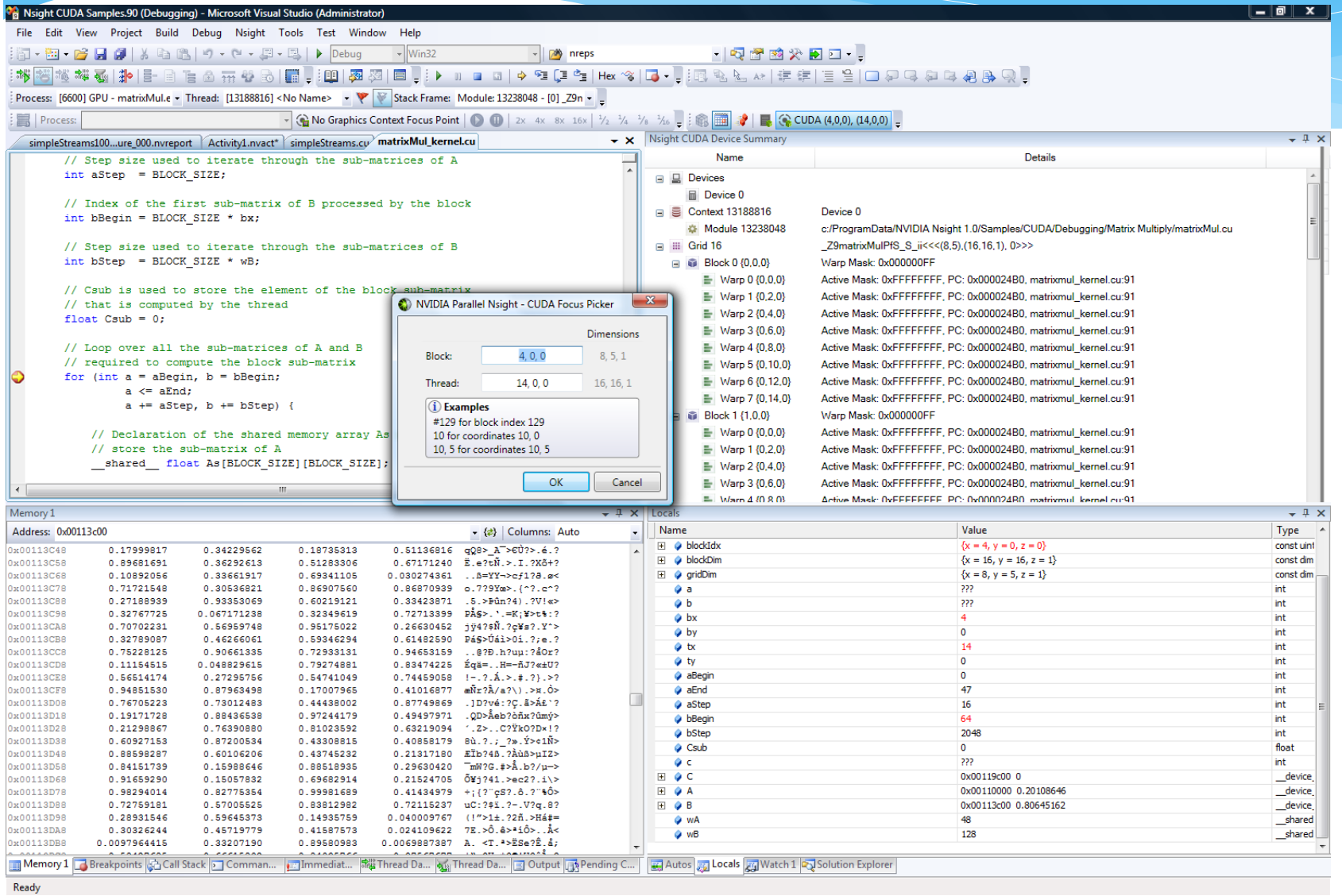
* Пересмотрен интерфейс пользователя
* Добавлено
  * Аналитика и рекомендации по коду
  * Определение ограничивающих факторов



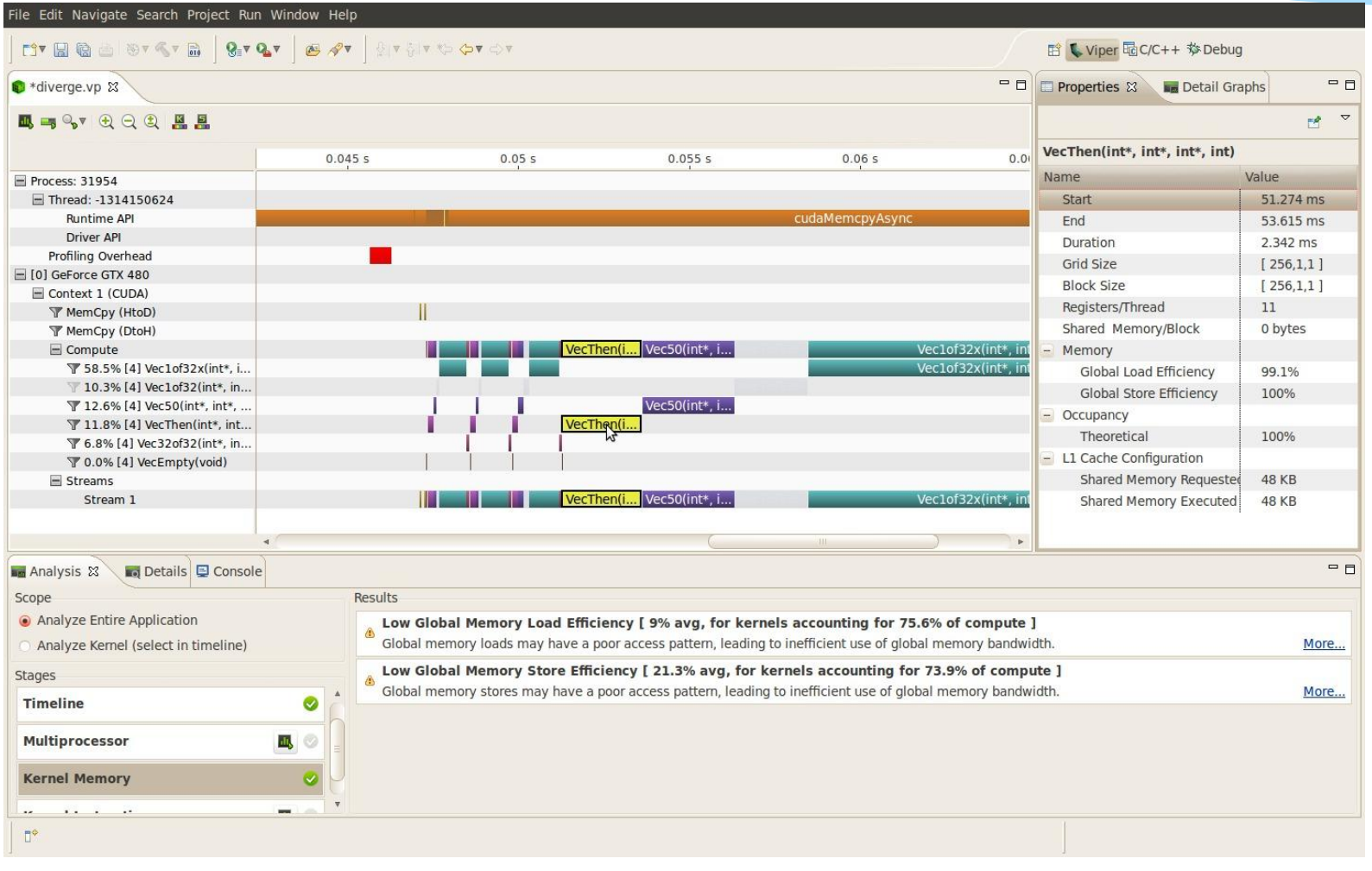
NVVP (CUDA 4.1)


NVIDIA Visual Profiler
File View Run Help
*New Session
4.75 s
5 s
145.515 ms
MemCpy (DtoH)
MemCpy (DtoD)
Compute
13.0% [100] swlat_do(double*, d...
12.4% [100] swlon_do(double*, d...
4.7% [100] Inv2(double*, double*...
4.7% [100] Inv1(double*, double*...
3.3% [100] Inv1r(double*, double...
3.3% [100] Inv2r(double*, double...
Properties
Detail Graphs
swlon_do(double*, double*, double*, double*, double
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
Details
Console
Settings
Reset All
Analyze All
Timeline
Multiprocessor
Kernel Memory
Kernel Instruction
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



NSight Eclipse Edition



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*instructions_issued (+1 на варп)
  * 128B*(global_store_transaction+ 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
    * Промах 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 128B 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

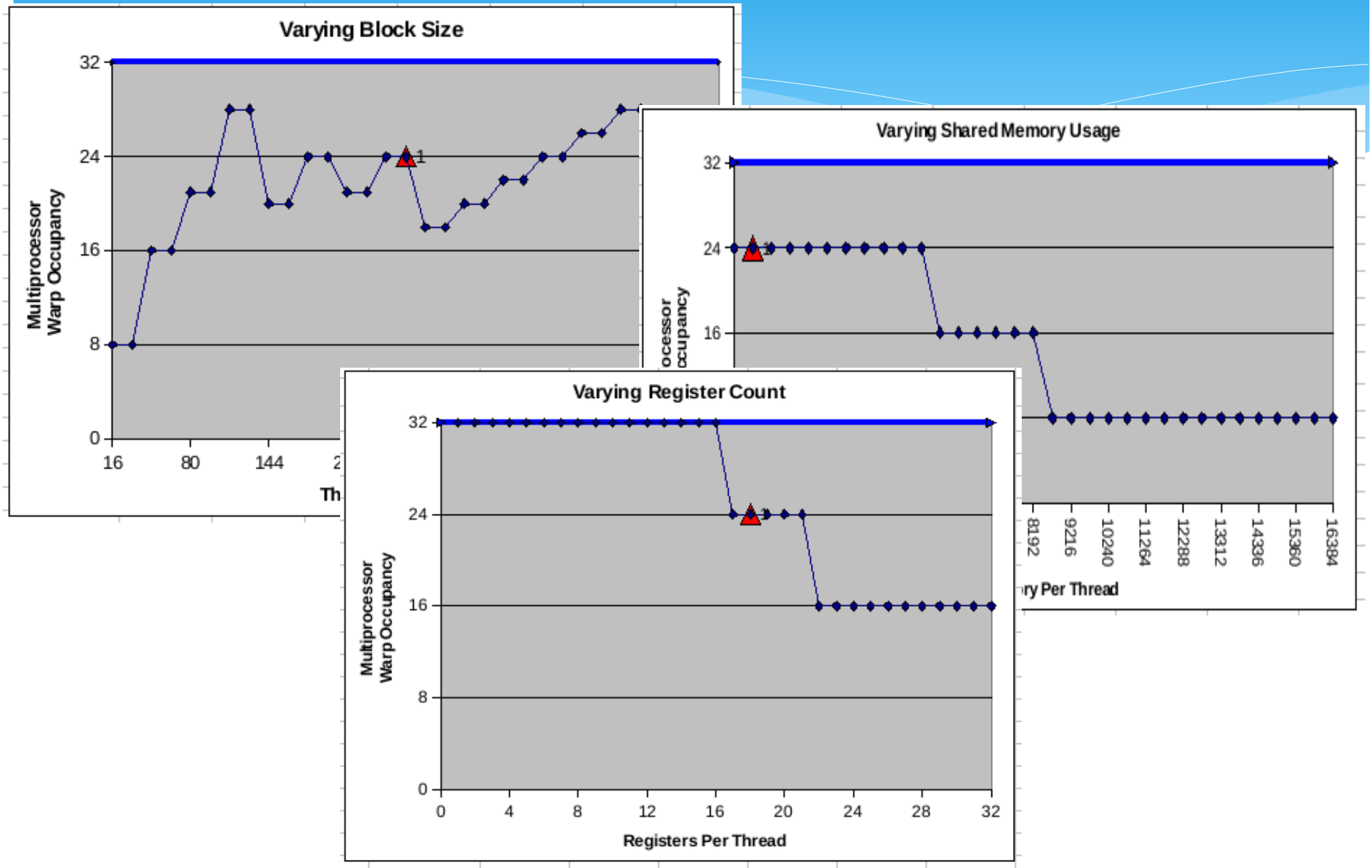
Just follow steps 1, 2, and 3 below! (or click here for help)	
1.) Select Compute Capability (click):	1,2
2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	18
Shared Memory Per Block (bytes)	512
(Don't edit anything below this line)	
3.) GPU Occupancy Data is displayed here and 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 / Multiprocess
Shared Memory / Multiprocessor (bytes)

Allocation Per Thread Block
Warps
Registers
Shared Memory
These data are used in computing the o

Maximum Thread Blocks Per Multiprocessor	Blocks
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 __fdividef(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 cudaMalloPitch 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

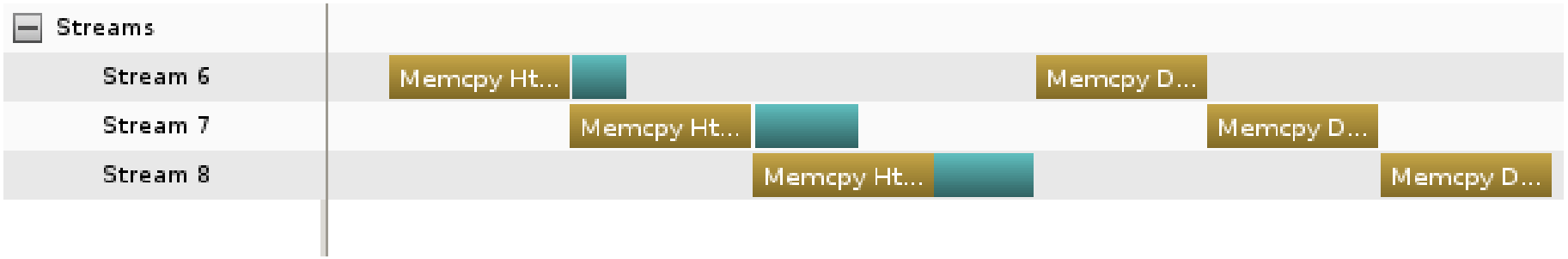
```
for (int i = 0; i < 3; ++i)
      cudaMemcpyAsync(inputDevPtr + i * size,
                      hostPtr + i * size, size,
                      cudaMemcpyHostToDevice, stream[i]);

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

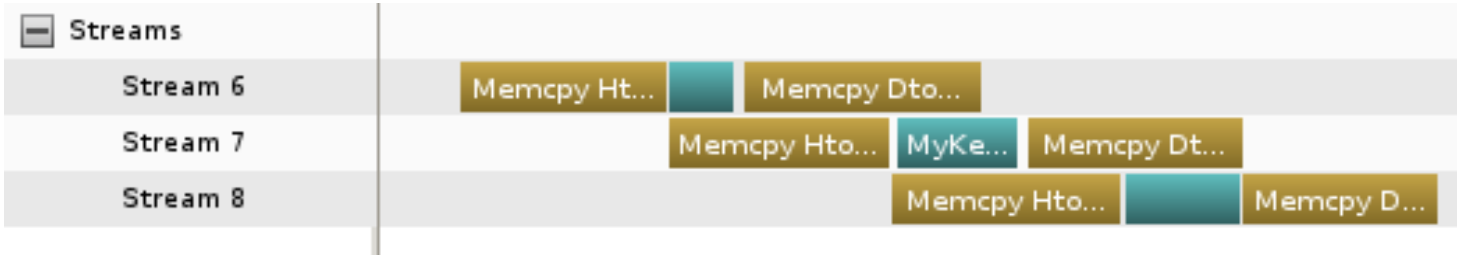
```
for(int i=0; i < 3; ++i) {
        cudaMemcpyAsync(inputDevPtr + i * size,
                        hostPtr + i * size, size,
                cudaMemcpyHostToDevice, stream[i]);

        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 capatibility

* 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_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