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

local_load

local_store

branch

divergent_branch

instructions

warp_serialize

cta_launched

gputime

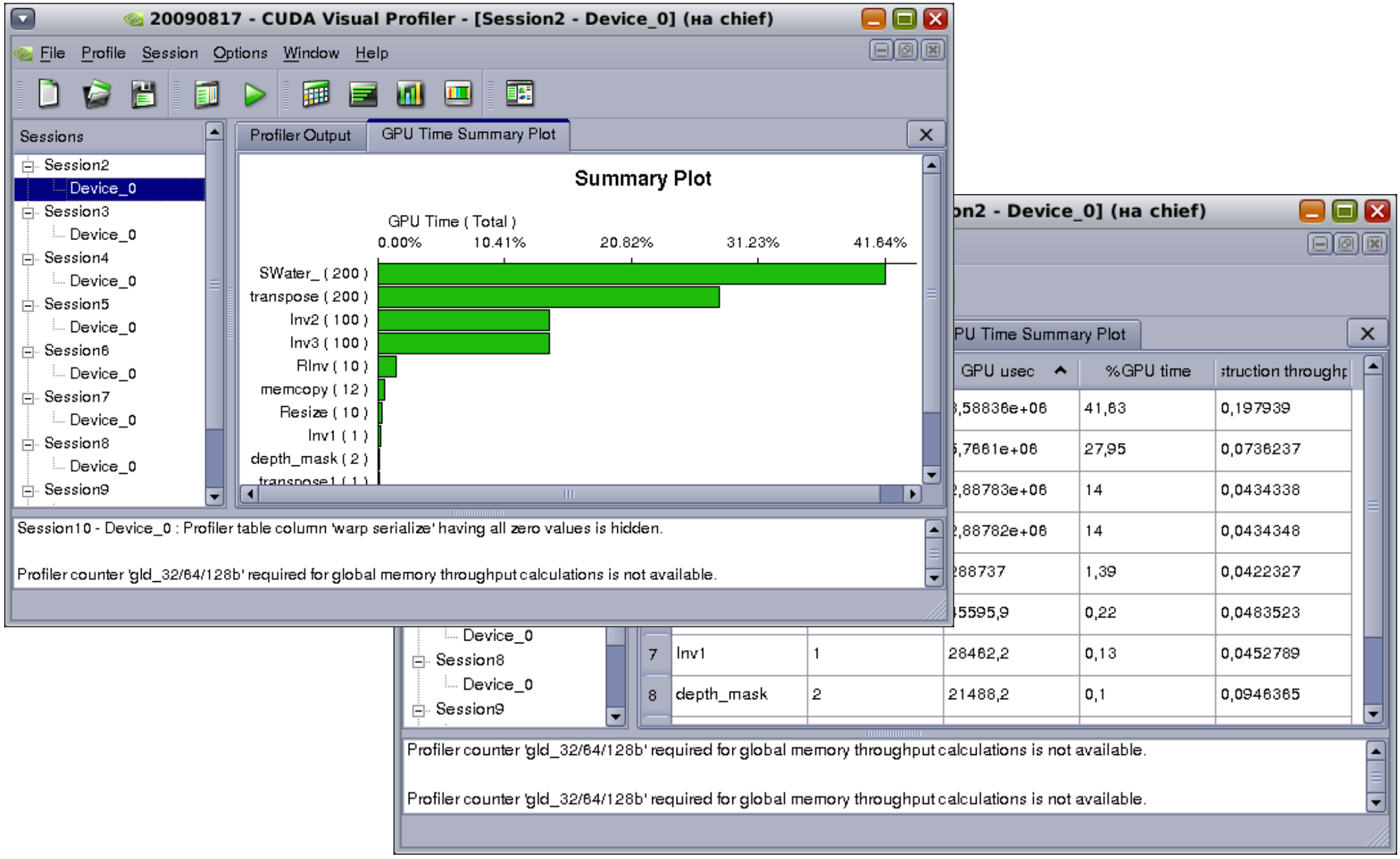
cputime

occupancy

Профилирование. Вывод

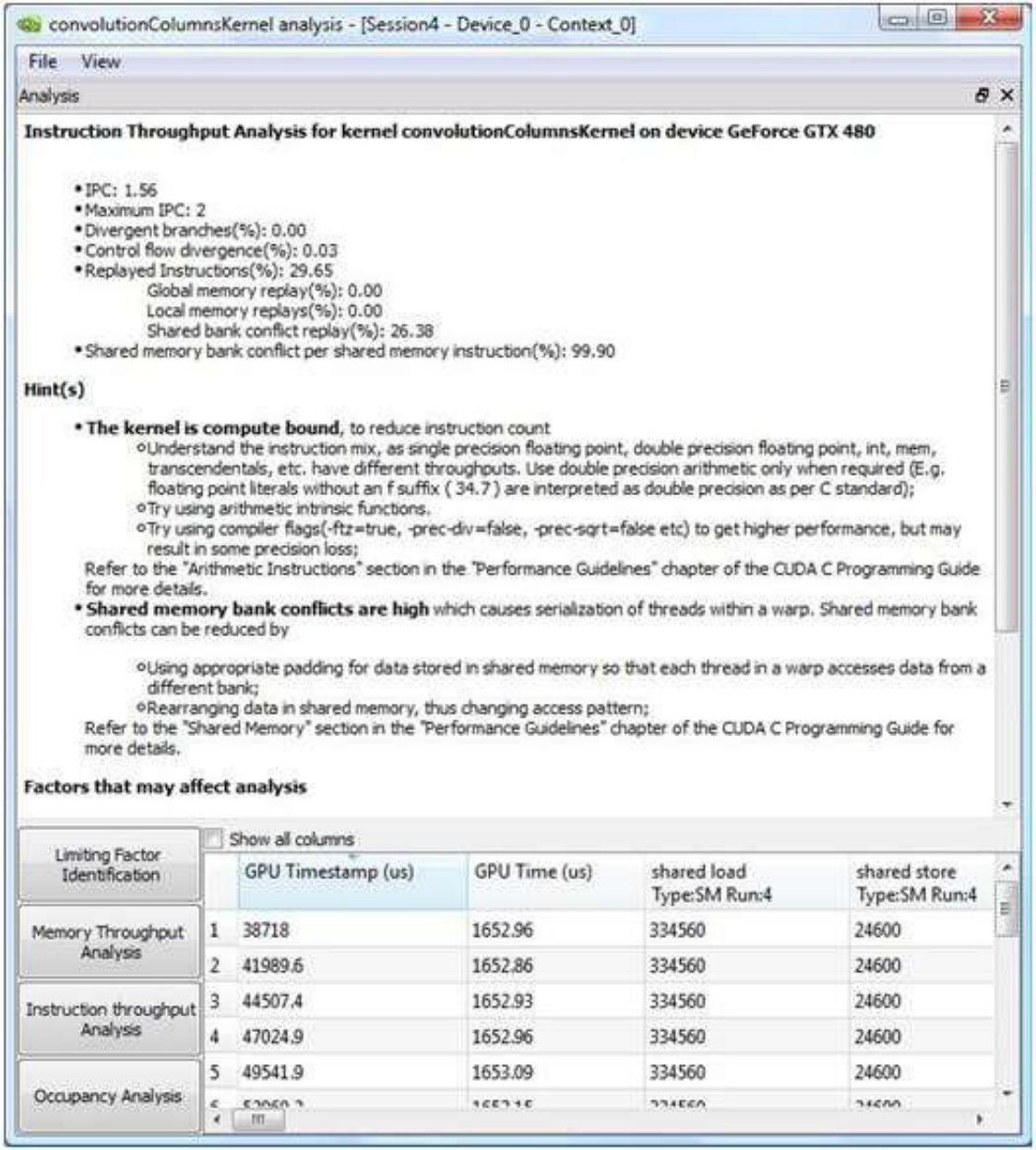
```
method,gputime,cputime,occupancy
method=[ memcopy ] gputime=[ 5519.680 ]
method=[ memcopy ] gputime=[ 5516.992 ]
method=[ memcopy ] gputime=[ 5517.728 ]
method=[ memcopy ] gputime=[ 5508.288 ]
method=[ __globfunc__Z10depth_maskPfS_fii ] gputime=[ 1791.424 ] cputime=[ 14.000 ]

          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 ]
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 cudaprof



Computerprof (CUDA 4.0)

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



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 ограничитель определяется автоматически

Анализ инструкций

- Счетчики профилировщика (на варп)
  - instructions executed: сколько инструкций исполнилось
  - instructions issued: включая сериализацию
- Разницамежду 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

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)

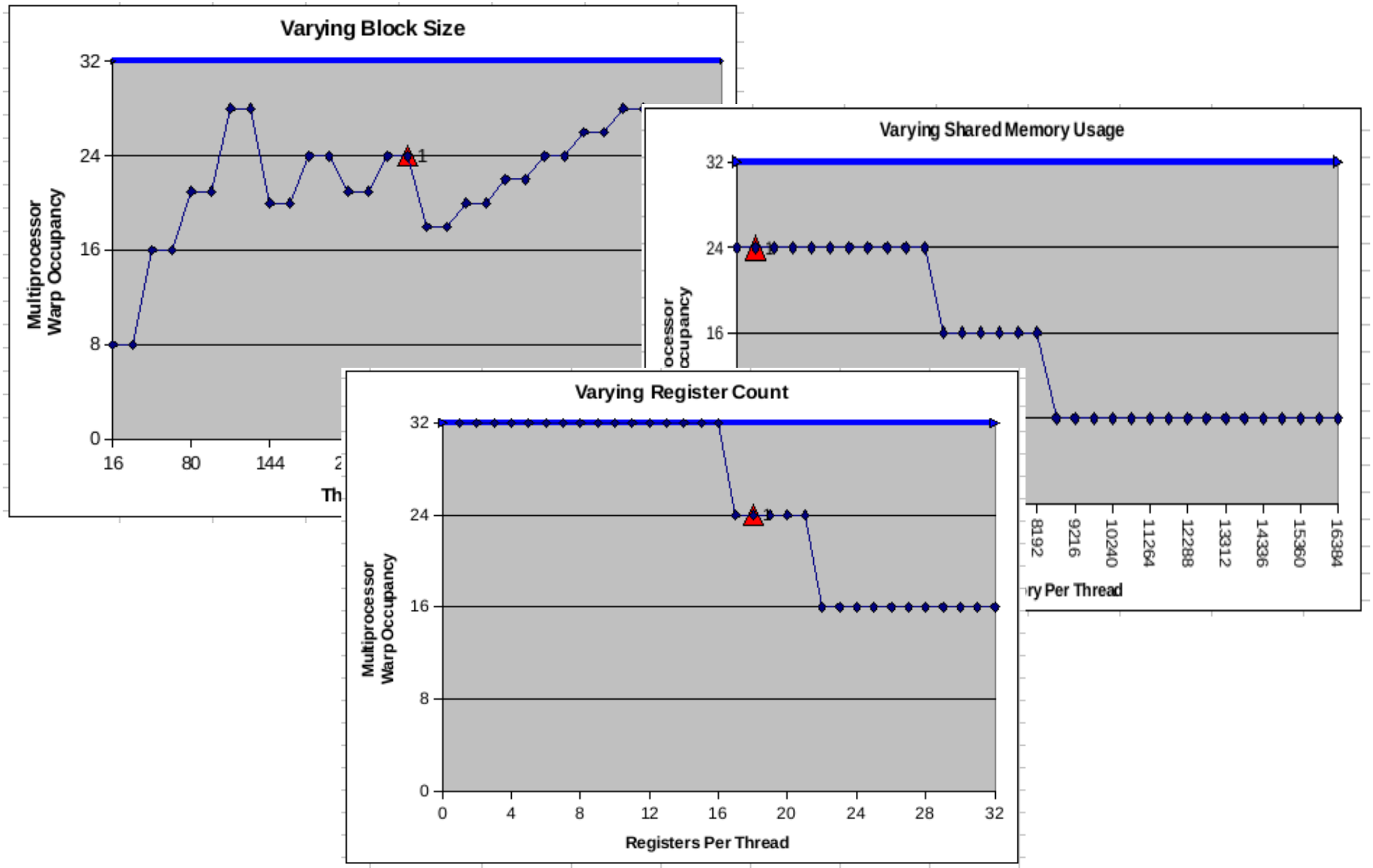
1.) Select Compute Capability (click):	1,2

Physical Limits for GPU:	2.) Enter your resource usage:	
Threads / Warp	Threads Per Block	256
Warps / Multiprocessor	Registers Per Thread	18
Threads / Multiprocessor	Shared Memory Per Block (bytes)	512
Thread Blocks / Multiprocessor		
Total # of 32-bit registers / Multiprocess		
Shared Memory / Multiprocessor (bytes)	(Don't edit anything below this line)	

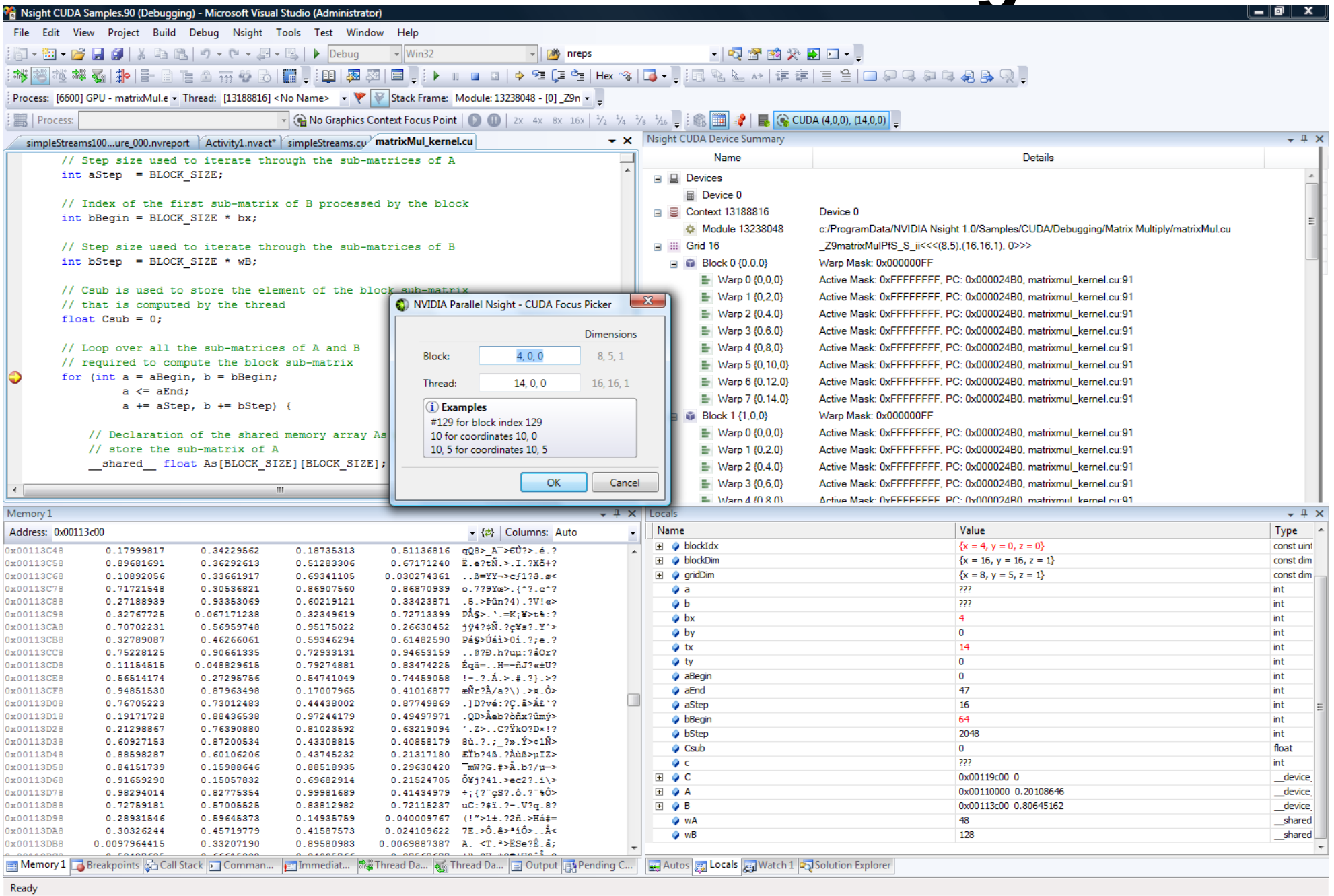
Allocation Per Thread Block	3.) GPU Occupancy Data is displayed here and in the graphs:	
Warps	Active Threads per Multiprocessor	768
Registers	Active Warps per Multiprocessor	24
Shared Memory	Active Thread Blocks per Multiprocessor	3
These data are used in computing the o	Occupancy of each Multiprocessor	75%

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



NVIDIA Parallel Nsight



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 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 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 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

Оптимизация

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