Технология CUDA для высокопроизводительных вычислений на кластерах с графическими процессорами

Колганов Александр
alexander.k.s@mail.ru

часть 3

Профилирование и отладка

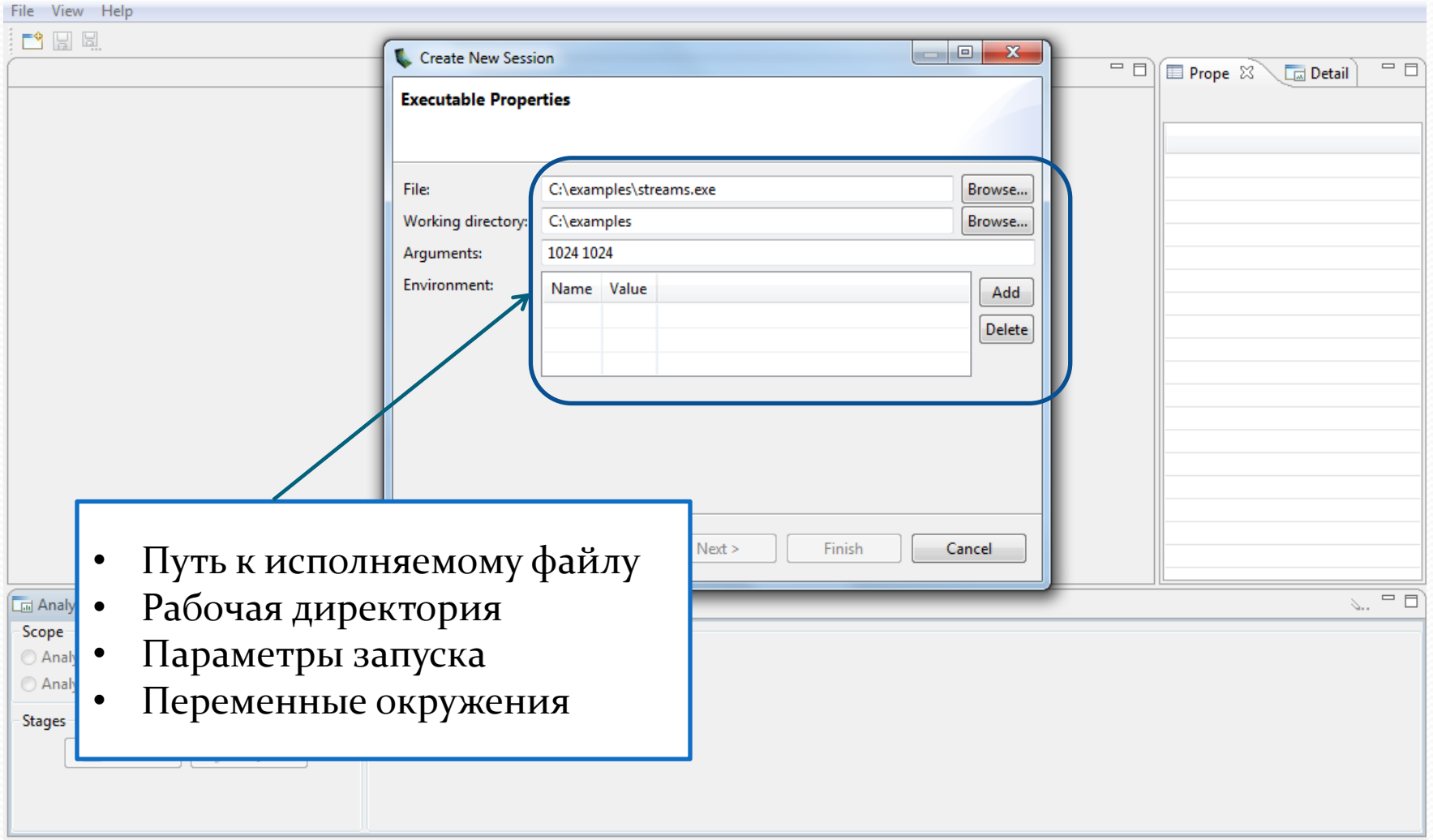
nvvp
nvprof
cuda-gdb
Удаленное профилирование

Visual Profiler

- NVidia Visual Profiler, NVVP
  - Поставляется вместе с toolkit-ом
  - Linux/Windows

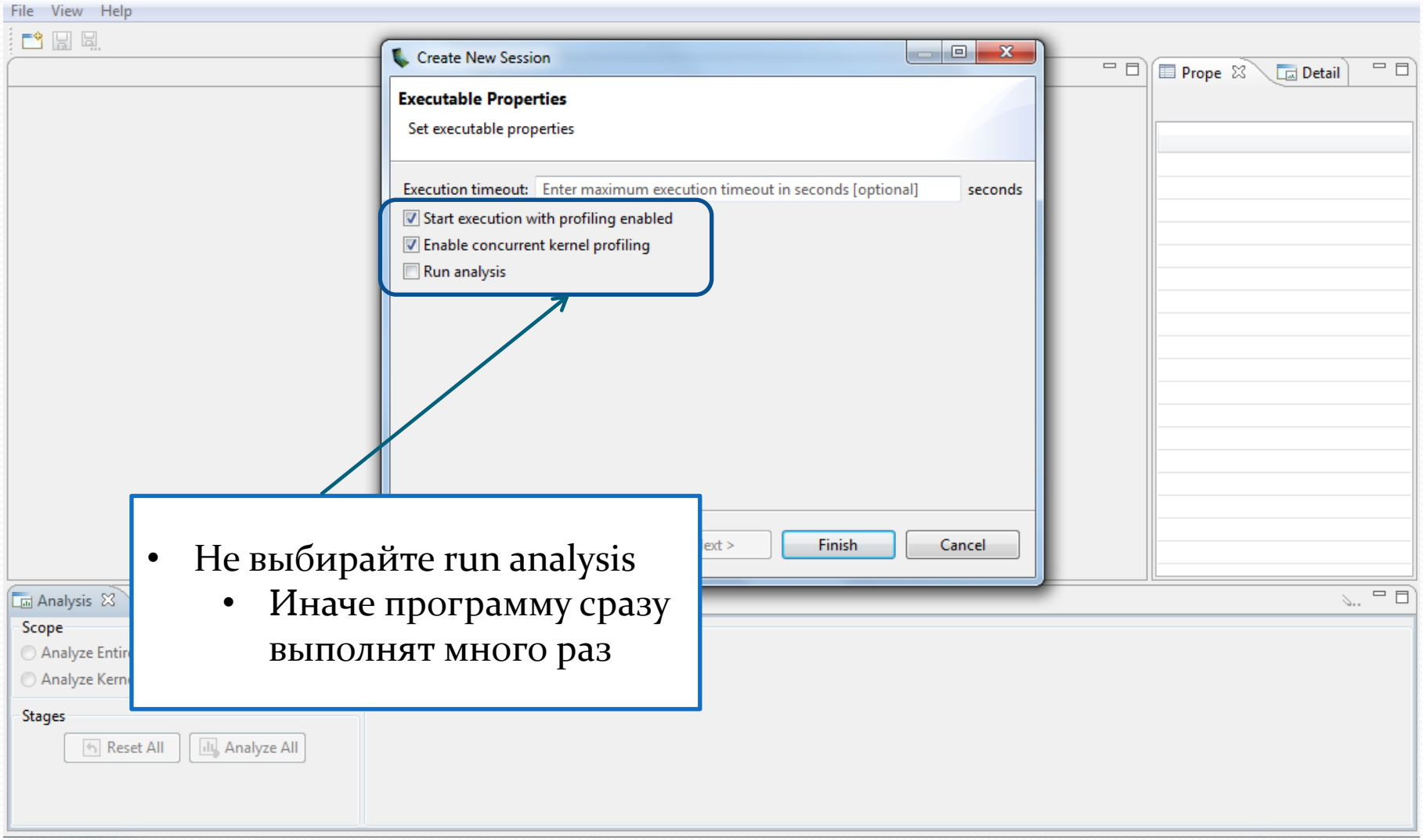
- timeline выполнения команд на GPU
- Показания счетчиков различных событий
- Расчёт метрик
- Автоматический анализ эффективности

File -> new Session



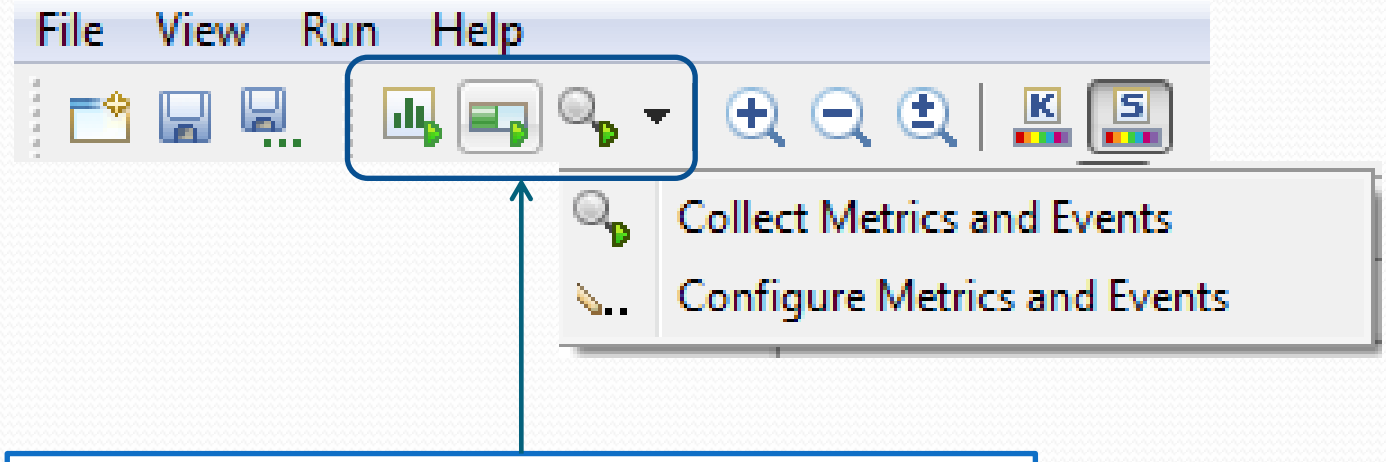
- Путь к исполняемому файлу
- Рабочая директория
- Параметры запуска
- Переменные окружения

File -> new Session -> next



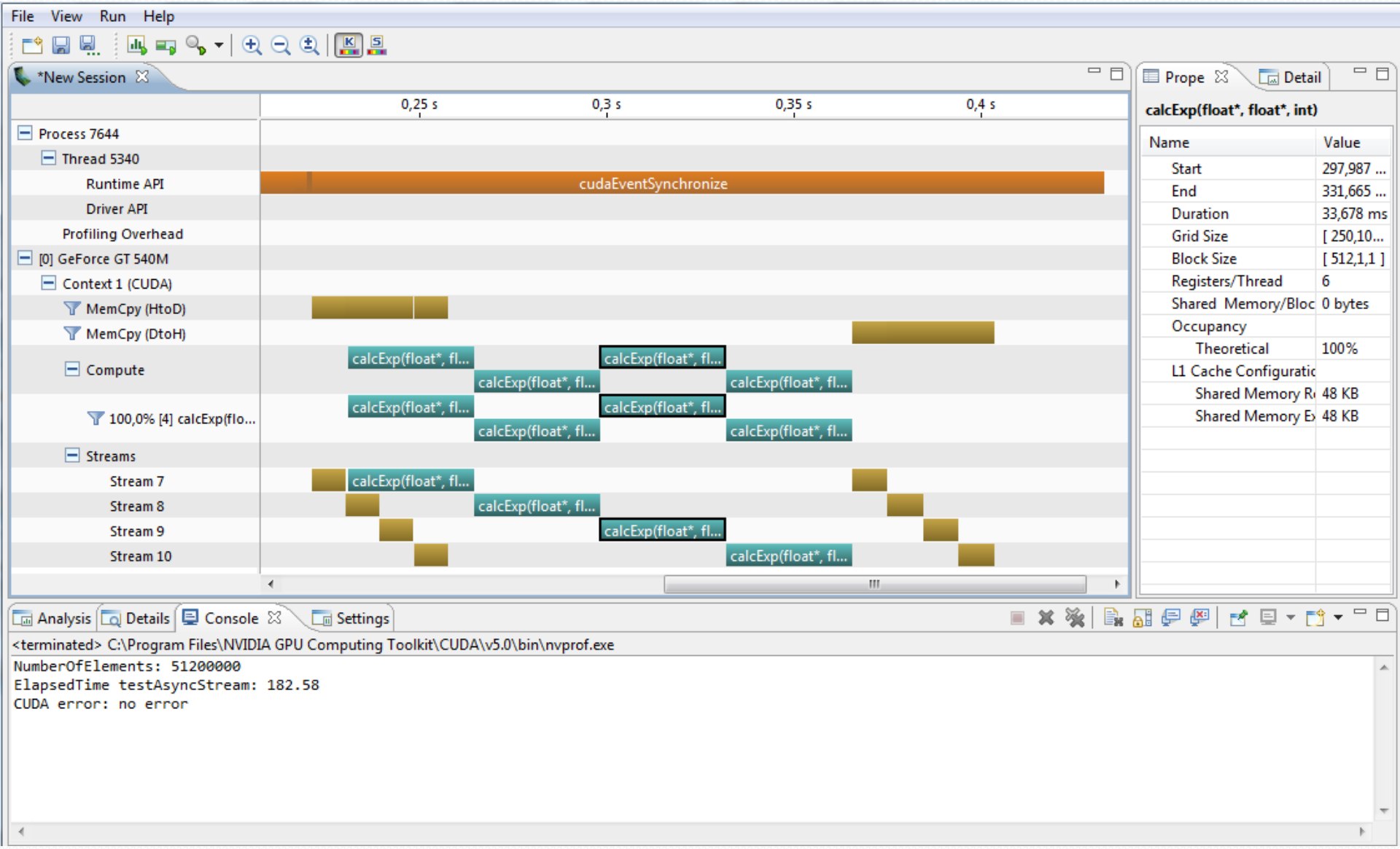
- Не выбирайте run analysis
  - Иначе программу сразу выполнят много раз

Toolbar

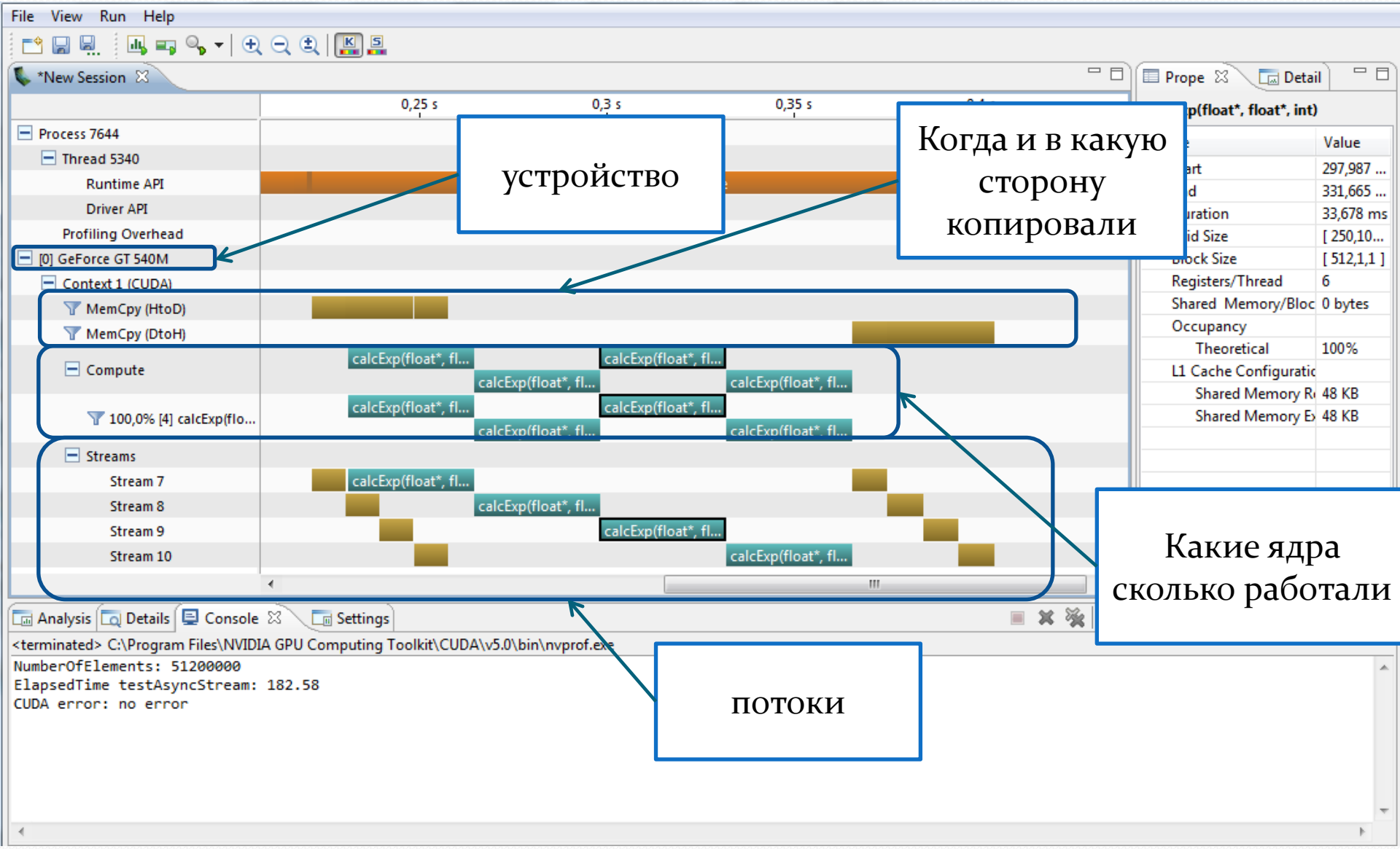


- Автоматический анализ
- Рассчитать timeline
- Собрать показания счетчиков и метрик
  - Выбрать интересующие счетчики и метрики

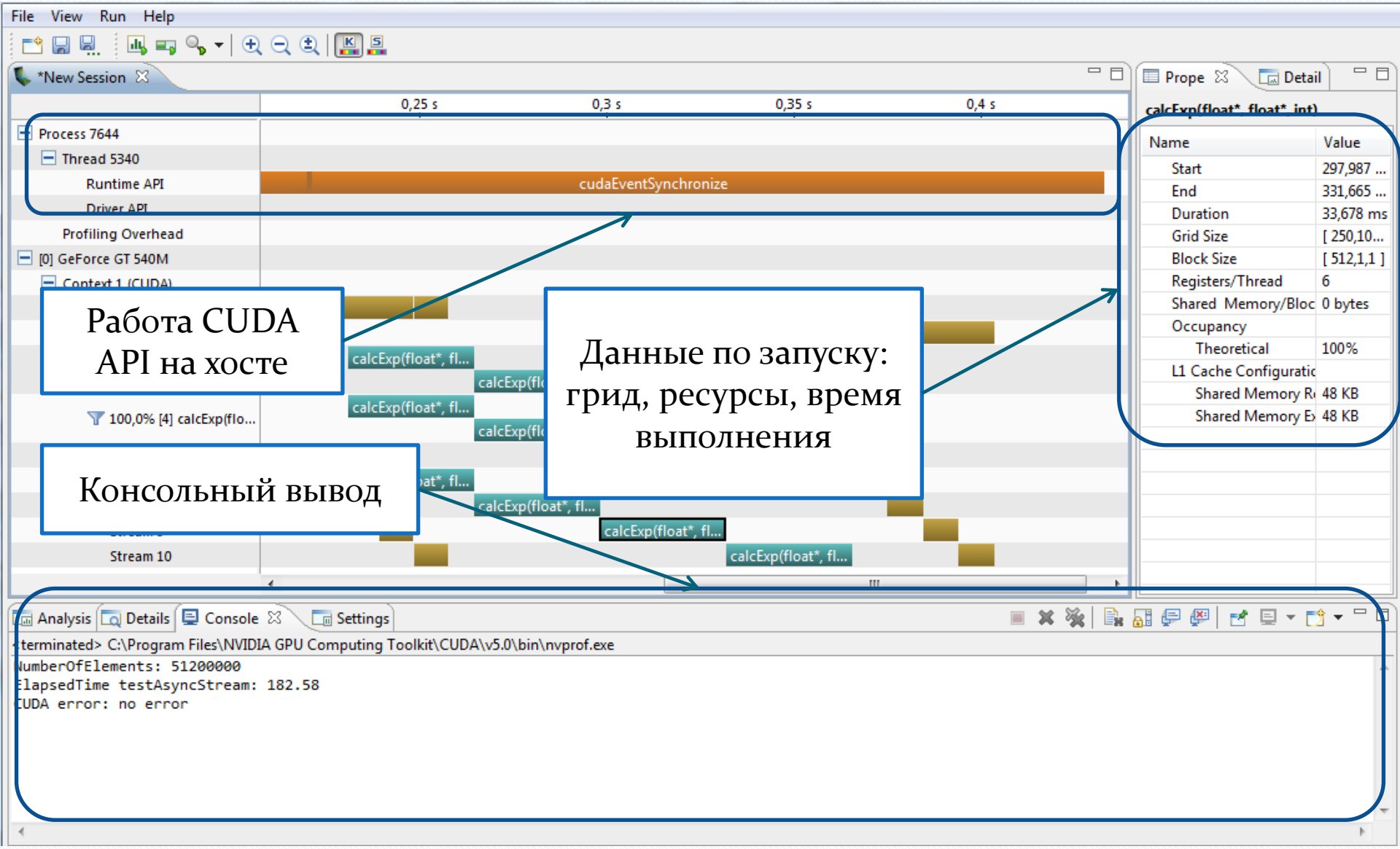
Основное окно



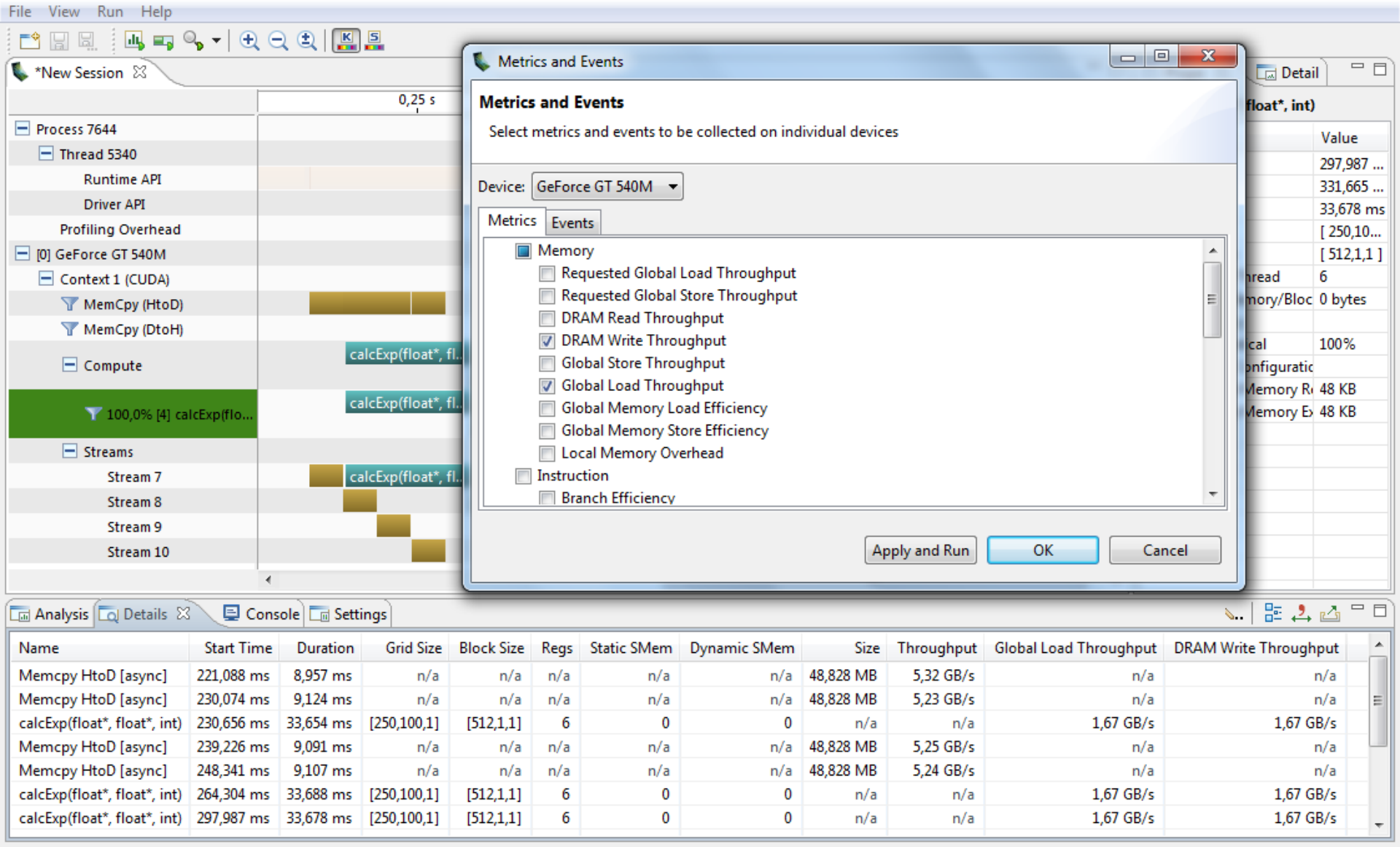
Основное окно



Основное окно

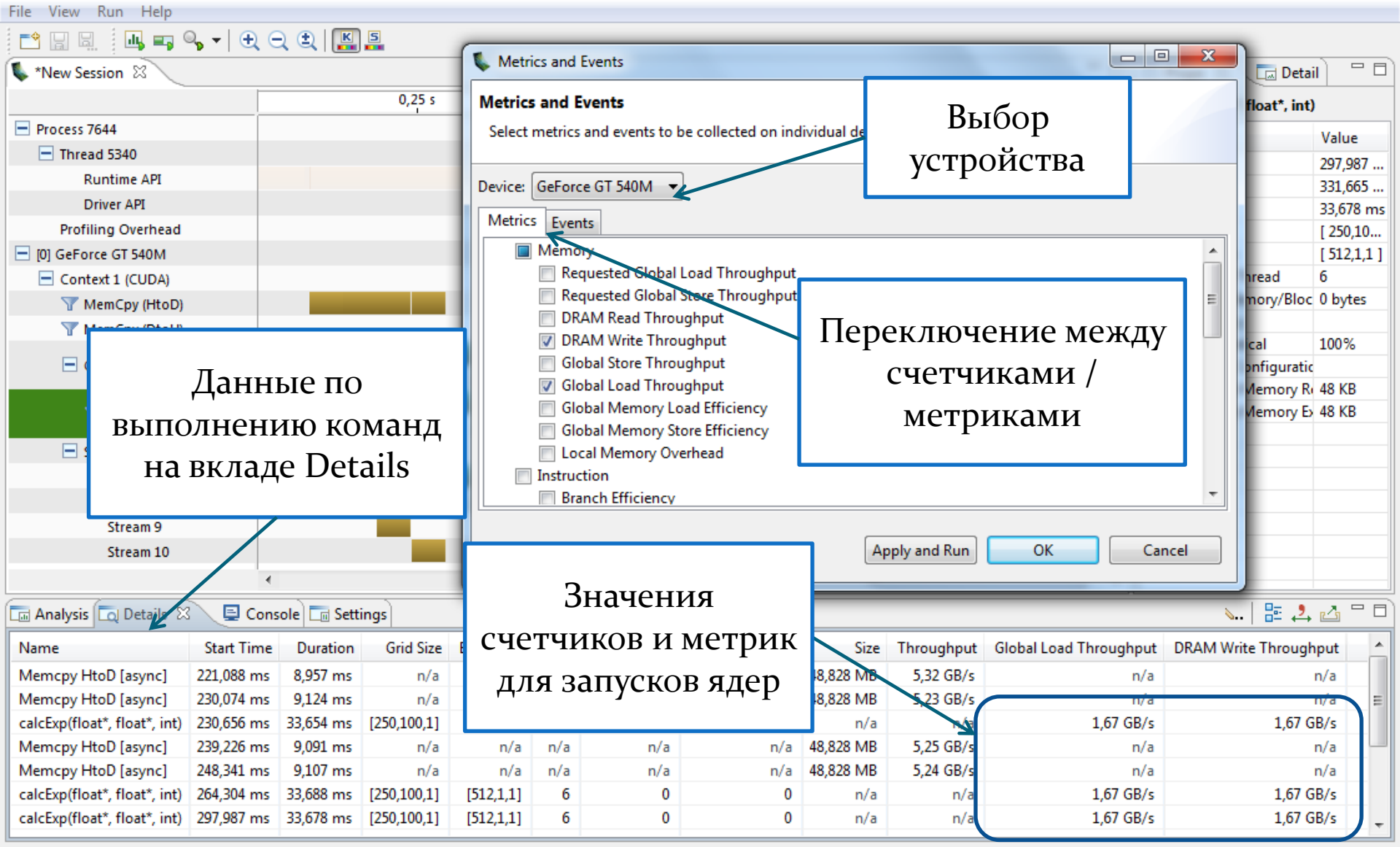


Счетчики и метрики



Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput	Global Load Throughput	DRAM Write Throughput
Memcpy HtoD [async]	221,088 ms	8,957 ms	n/a	n/a	n/a	n/a	n/a	48,828 MB	5,32 GB/s	n/a	n/a
Memcpy HtoD [async]	230,074 ms	9,124 ms	n/a	n/a	n/a	n/a	n/a	48,828 MB	5,23 GB/s	n/a	n/a
calcExp(float*, float*, int)	230,656 ms	33,654 ms	[250,100,1]	[512,1,1]	6	0	0	n/a	n/a	1,67 GB/s	1,67 GB/s
Memcpy HtoD [async]	239,226 ms	9,091 ms	n/a	n/a	n/a	n/a	n/a	48,828 MB	5,25 GB/s	n/a	n/a
Memcpy HtoD [async]	248,341 ms	9,107 ms	n/a	n/a	n/a	n/a	n/a	48,828 MB	5,24 GB/s	n/a	n/a
calcExp(float*, float*, int)	264,304 ms	33,688 ms	[250,100,1]	[512,1,1]	6	0	0	n/a	n/a	1,67 GB/s	1,67 GB/s
calcExp(float*, float*, int)	297,987 ms	33,678 ms	[250,100,1]	[512,1,1]	6	0	0	n/a	n/a	1,67 GB/s	1,67 GB/s

Счетчики и метрики



Счетчики и метрики

- Event
  - Некоторое низкоуровневое событие, например кеш-промах, банк-конфликт, бранчинг, завершение выполнения варпа и т.д.
  - В специальных регистрах-счетчиках на мультипроцессорах накапливаются количества таких событий
- Metric
  - Более высокоуровневая информация, рассчитываемая на основе счетчиков, например, % кеш промахов
- Примеры счетчиков и метрик в следующий раз

Полезные метрики

- Метрики рассчитываются на основе значений аппаратных счетчиков
- Описания метрик - http://docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference
- Точные формулы расчета можно найти в CUDA_Profiler_User_Guide.pdf

Полезные метрики

- **achieved_occupancy**
- **branch_efficiency** – доля варпов, выполненных без бранчей
- **warp_execution_efficiency** – средняя доля активных нитей в варпах
- **inst_replay_overhead** – повторы инструкций (из-за сериализации) / общее число выполненных устройством инструкций
- **shared_replay_overhead, global_cache_replay_overhead, local_replay_overhead** – разделение предыдущего по причинам

Полезные метрики

- **dram_read_throughput**– объем памяти, отданной ядру / время выполнения
- **gld_throughput** - объем глобальной памяти, отданной ядру / время выполнения
- **gld_requested_throughput** - объем глобальной памяти, запрошенный ядром / время выполнения
- **gld_efficiency** – gld_requested_throughput / gld_throughput
- **dram_write_throughput, gst_throughput, gst_requested_throughput, gst_efficiency** - аналогично для записи

Полезные метрики

- **l1_cache_global_hit_rate** – доля кеш-попаданий при обращении к глобальной памяти
- **l1_cache_local_hit_rate** – доле кеш-попаданий при обращении к локальной памяти
- **l2_l1_read_hit_rate –**доля кеш попаданий при обращении кеша l1 к кешу l2
- **local_memory_overhead** - доля локальной памяти в обменах между l1 и l2
- **IPC** – число инструкций, запускаемых за цикл работы warp scheduler

command-line profiler

Command-line profiler

Включается/отключается через переменную окружения COMPUTE_PROFILE:

```
$export COMPUTE_PROFILE=1
$./a.out
```

Или

```
$COMPUTE_PROFILE=1 ./a.out
```

Файл лога

- По умолчанию пишет лог профилировки в файлы cuda_profile_%d.log в текущей директории (pwd), где %d заменяется на номер контекста
  - Для каждого GPU свой лог профилировки - cuda_profile_0.log, cuda_profile_1.log и т.д.

- Можно изменить через переменную COMPUTE_PROFILE_LOG:

```
$export COMPUTE_PROFILE_LOG=example.%d.log
$COMPUTE_PROFILE=1 ./a.out
```

Стандартные счетчики

- Command-line profiler по умолчанию для каждой команды, выполненной на GPU, выводит в отдельной строке:
  - **method** – имя ядра | memcpyHtoD | memcpyDtoH
  - **gputime** – сколько времени команда выполнялась на GPU
  - **cputime** – сколько времени команда выполнялась на хосте
    - Для неблокирующих команд – время запуска, затраченное хостом на запуск
    - Для блокирующих – полное время, на которое заблокировалась хост-нить
  - **occupancy**

Файл конфигурации

- Файл, в котором указаны идентификаторы интересующих опций/счетчиков, по одному в строке + строки с комментариями, начинающиеся с #
- Доступные опции:

  http://docs.nvidia.com/cuda/profiler-users-guide/index.html#command-line-profiler-options

- Доступные счетчики:

```
$nvprof --query-events
```

Файл конфигурации

- Указываем файл в переменной окружения COMPUTE_PROFILE_CONFIG:

```
$ cat profiler.config
#some comment
gpustarttimestamp
streamid
$export COMPUTE_PROFILE_CONFIG=profiler.config
$COMPUTE_PROFILE=1 ./a.out
```

Несовместимость счетчиков

- Некоторые счетчики не могут быть собраны за один прогон
  - Например, gld_inst_32bit и sm_cta_launched

- В этом случае профилировщик выберет совместимый набор счетчиков, а в лог профилировки добавит «NV_Warning: Counter 'sm_cta_launched' is not compatible with other selected counters and it cannot be profiled in this run»

- Visual Profiler автоматически делает нужное число прогонов, но с command-line profiler это приходится делать вручную

Поддерживаемые форматы

- Key-Value-Pairs (по-умолчанию)

```
method,gputime,cputime,occupancy
_Z6kernelPfi,1724792.500,17.270,1.000
_Z6kernelPfi,1724766.000,5.320,1.000
_Z6kernelPfi,1724765.625,8.675,1.000
```

- Comma-Separated Values

```
method,gputime,cputime,occupancy
method=[ _Z6kernelPfi ] gputime=[ 1724745.250 ] cputime=[ 17.638 ] occupancy=[ 1.000 ]
method=[ _Z6kernelPfi ] gputime=[ 1724793.750 ] cputime=[ 5.287 ] occupancy=[ 1.000 ]
method=[ _Z6kernelPfi ] gputime=[ 1724740.500 ] cputime=[ 8.878 ] occupancy=[ 1.000 ]
```

Поддерживаемые форматы

- Comma-Separated Values включается через переменную COMPUTE_PROFILE_CSV:

```
$export COMPUTE_PROFILE_CSV=1
$COMPUTE_PROFILE=1 ./a.out
```

- Или строчкой в файле конфигурации:

```
profilelogformat CSV
```

Visual Profiler & Command-line

- Лог command-line профилировщика можно открыть в NVVP, если
  - В конфигурации указаны gpustarttimestamp и streamid - для построения timeline
  - Формат CSV
- Можно открыть сразу несколько логов – их timeline будут склеены
  - Если несколько логов получилось в результате прогонов с различными наборами счетчиков – лучше вручную объединить столбцы

Пример

```
$cat profiler.config
gpustarttimestamp
streamid
conckerneltrace
gridsize
$export COMPUTE_PROFILE_CSV=1
$export COMPUTE_PROFILE_CONFIG=profiler.config
$export COMPUTE_PROFILE_LOG=example.%d.log
$export COMPUTE_PROFILE=1
$./a.out
```

Пример

```
$cat example.0.log
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 3 Tesla C2075
# CUDA_CONTEXT 1
# CUDA_PROFILE_CSV 1
# TIMESTAMPFACTOR 13a27e35a487845e
gpustarttimestamp,method,gputime,cputime,gridsizeX,gridsizeY,occupancy,streamid
13a71b99010f2020,memcpyHtoD,397217.531,1501.800,,,,1
13a71b9930a33d40,memcpyHtoD,393306.625,1338.747,,,,1
13a71b99483ee600,_Z6matmul14cudaPitchedPtrS_S_,12667.392,5.871,32,32,0.667,1
13a71b99490c0bc0,memcpyDtoH,2406.752,4200.187,,,,1
```

- Файл example.0.log можно открыть в Visual Profiler

Command Line Profiler & метрики

Command-line profiler не поддерживает метрики!

nvprof

nvprof

- «Бэкенд» nvvp
- Позволяет быстро посмотреть что-сколько считается, собрать события и метрики
- Данные профилировки nvprof могут быть экспортированны в nvvp

Использование

- Компиляция остается без изменений
- Профилировка:

```
$nvprof [опции] [исполняемый файл] [параметры]
```

- Режимы:
  - Summary
  - Trace
  - Events/metrics

Режимы

- *Summary*

  Быстрый способ узнать на что ушла большая часть времени исполнения

- *Trace*

  Трасса вызовов CUDA API и GPU-операций

- *Events/metrics*

  События и метрики

Summary

```
$nvprof ./matmul 1024 1024 1024
```

- Вывести список использованных функций тулкита, ядер, пересылок данных.
- Для каждой операции:
  - Доля от общего времени выполнения, затраченное на все выполнения операции
  - Суммарное время всех выполнений операции
  - Число выполнений
  - Время выполнения (среднее, минимальное, максимальное)

Пример Summary

```
$nvprof ./matmul 1024 1024 1024
```

```
Elapsed time: 5.10317
CUDA error: no error
==12718== NVPROF is profiling process 12718, command: ./matmul 1024 1024 1024
==12718== Profiling application: ./matmul 1024 1024 1024
==12718== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 58.87%  5.0759ms         1  5.0759ms  5.0759ms  5.0759ms  matmul(cudaPitchedPtr, cudaPitchedPtr, cudaPitchedPtr)
 21.73%  1.8737ms         2  936.83us  923.18us  950.47us  [CUDA memcpy HtoD]
 19.40%  1.6731ms         1  1.6731ms  1.6731ms  1.6731ms  [CUDA memcpy DtoH]

==12718== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 59.33%  104.54ms         3  34.847ms  153.32us  104.23ms  cudaMallocPitch
 31.37%  55.274ms         1  55.274ms  55.274ms  55.274ms  cudaDeviceReset
  2.95%  5.2055ms         1  5.2055ms  5.2055ms  5.2055ms  cudaEventSynchronize
  2.84%  5.0000ms         3  1.6667ms  1.0558ms  2.7587ms  cudaMemcpy2D
  1.55%  2.7308ms       332  8.2250us     270ns  425.65us  cuDeviceGetAttribute
  1.29%  2.2730ms         4  568.24us  549.16us  579.66us  cudaGetDeviceProperties
  0.33%  577.77us         3  192.59us  161.06us  252.17us  cudaFree
  0.17%  308.30us         4  77.076us  64.779us  90.876us  cuDeviceTotalMem
  0.13%  221.07us         4  55.267us  52.939us  59.994us  cuDeviceGetName
…
```

Пример Summary

```
$nvprof ./matmul 1024 1024 1024
```

```
Elapsed time: 5.10317
CUDA error: no error
==12718== NVPROF is profiling process 12718, command: ./matmul 1024 1024 1024
==12718== Profiling application: ./matmul 1024 1024 1024
==12718== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 58.87%  5.0759ms         1  5.0759ms  5.0759ms  5.0759ms  matmul(cudaPitchedPtr, cudaPitchedPtr, cudaPitchedPtr)
 21.73%  1.8737ms         2  936.83us  923.18us  950.47us  [CUDA memcpy HtoD]
 19.40%  1.6731ms         1  1.6731ms  1.6731ms  1.6731ms  [CUDA memcpy DtoH]

==12718== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 59.33%  104.54ms         3  34.847ms  153.32us  104.23ms  cudaMallocPitch
 31.37%  55.274ms         1  55.274ms  55.274ms  55.274ms  cudaDeviceReset
  2.95%  5.2055ms         1  5.2055ms  5.2055ms  5.2055ms  cudaEventSynchronize
  2.84%  5.0000ms         3  1.6667ms  1.0558ms  2.7587ms  cudaMemcpy2D
  1.55%  2.7308ms       332  8.2250us     270ns  425.65us  cuDeviceGetAttribute
  1.29%  2.2730ms         4  568.24us  549.16us  579.66us  cudaGetDeviceProperties
  0.33%  577.77us         3  192.59us  161.06us  252.17us  cudaFree
  0.17%  308.30us         4  77.076us  64.779us  90.876us  cuDeviceTotalMem
  0.13%  221.07us         4  55.267us  52.939us  59.994us  cuDeviceGetName
…
```

Пример Summary

```
$nvprof ./matmul 1024 1024 1024
```

```
Elapsed time: 5.10317
CUDA error: no error
==12718== NVPROF is profiling process 12718, command: ./matmul 1024 1024 1024
==12718== Profiling application: ./matmul 1024 1024 1024
==12718== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 58.87%  5.0759ms         1  5.0759ms  5.0759ms  5.0759ms  matmul(cudaPitchedPtr, cudaPitchedPtr, cudaPitchedPtr)
 21.73%  1.8737ms         2  936.83us  923.18us  950.47us  [CUDA memcpy HtoD]
 19.40%  1.6731ms         1  1.6731ms  1.6731ms  1.6731ms  [CUDA memcpy DtoH]

==12718== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 59.33%  104.54ms         3  34.847ms  153.32us  104.23ms  cudaMallocPitch
 31.37%  55.274ms         1  55.274ms  55.274ms  55.274ms  cudaDeviceReset
  2.95%  5.2055ms         1  5.2055ms  5.2055ms  5.2055ms  cudaEventSynchronize
  2.84%  5.0000ms         3  1.6667ms  1.0558ms  2.7587ms  cudaMemcpy2D
  1.55%  2.7308ms       332  8.2250us     270ns  425.65us  cuDeviceGetAttribute
  1.29%  2.2730ms         4  568.24us  549.16us  579.66us  cudaGetDeviceProperties
  0.33%  577.77us         3  192.59us  161.06us  252.17us  cudaFree
  0.17%  308.30us         4  77.076us  64.779us  90.876us  cuDeviceTotalMem
  0.13%  221.07us         4  55.267us  52.939us  59.994us  cuDeviceGetName
  …
```

CPU Trace

```
$nvprof --print-api-trace ./matmul 1024 1024 1024
```

- Трасса вызовов CUDA API
- Для каждого вызова в отдельной строке время старта и время выполнения
- Не только CUDA driver API, но и CUDA runtime

CPU Trace

```
$nvprof --print-api-trace ./matmul 1024 1024 1024
```

==13358== NVPROF is profiling process 13358, command: ./a.out 32768 1 2 2

==13358== Profiling application: ./a.out 32768 1 2 2

==13358== Profiling result:

Start	Duration	Name
126.71ms	1.3140us	cuDeviceGetCount
126.72ms	402ns	cuDeviceGet
126.73ms	404ns	cuDeviceGet
126.74ms	285ns	cuDeviceGet

GPU Trace

```
$nvprof --print-gpu-trace ./matmul 1024 1024 1024
```

- Трасса GPU-операций – запуски ядер и копирования памяти
- Для каждой операции в отдельной строке
  - Старт, продолжительность
  - Для запусков ядер:
    - Грид
    - Ресурсы (регистры, общая память)
  - Для копирований памяти:
    - Объем
    - Скорость GB/s
  - Устройство
  - Поток

GPU Trace

```
$nvprof --print-gpu-trace ./matmul 1024 1024 1024

==13406== NVPROF is profiling process 13406, command: ./a.out 32768 1 2 2
==13406== Profiling application: ./a.out 32768 1 2 2
==13406== Profiling result:
   Start  Duration            Grid Size      Block Size     Regs*    SSMem*    DSMem*
Size  Throughput           Device   Context    Stream  Name
289.04ms  13.981ms                    -               -         -         -         -
67.109MB  4.8001GB/s  Tesla C2075 (2)         1        13  [CUDA memcpy HtoD]
303.02ms  15.931ms                    -               -         -         -         -
67.109MB  4.2124GB/s  Tesla C2075 (2)         1        14  [CUDA memcpy HtoD]
303.03ms  1.8371ms          (16384 1 1)       (512 1 1)        16        0B        0B
-           -  Tesla C2075 (2)         1        13  kernel(double*, int) [363]
304.87ms  15.973ms                    -               -         -         -         -
67.109MB  4.2013GB/s  Tesla C2075 (2)         1        13  [CUDA memcpy DtoH]
318.98ms  1.8346ms          (16384 1 1)       (512 1 1)        16        0B        0B
-           -  Tesla C2075 (2)         1        14  kernel(double*, int) [369]
320.85ms  11.025ms                    -               -         -         -         -
67.109MB  6.0872GB/s  Tesla C2075 (2)         1        14  [CUDA memcpy DtoH]
```

События/метрики

```
$nvprof --devices <индекс> --query-metrics
$nvprof --devices <индекс> --query-events
```

- Запросить доступные для устройства метрики/события

```
$nvprof -m <метрика>,[<метрика>] ./matmul 1024 1024 1024
$nvprof -e <событие>,[<событие>] ./matmul 1024 1024 1024
```

- Посчитать значения метрик/событий (среднее, максимальное, минимальное) для всех запусков каждого ядра

События/метрики

```
$nvprof  -m ipc,gld_efficiency,flop_sp_efficiency ./matmul 1024 1024 1024
==13638== NVPROF is profiling process 13638, command: ./matmul 1024 1024 1024
==13638== Warning: Some kernel(s) will be replayed on device 1 in order to collect all
events/metrics.
==13638== Profiling application: ./matmul 1024 1024 1024
==13638== Profiling result:
==13638== Metric result:
Invocations                               Metric Name                         Metric
Description         Min         Max         Avg
Device "Tesla C2075 (1)"
         Kernel: matmul(cudaPitchedPtr, cudaPitchedPtr, cudaPitchedPtr)
          1                                        ipc
Executed IPC    1.484029    1.484029    1.484029
          1                               gld_efficiency             Global Memory Load
Efficiency     100.20%     100.20%     100.20%
          1                          flop_sp_efficiency              FLOP Efficiency(Peak
Single)       5.87%       5.87%       5.87%
```

Перенаправление вывода

```
$nvprof --log-file nvprof.log ./matmul 1024 1024 1024
```

- Перенаправить вывод в файл

```
$nvprof --profile-api-trace none ./matmul 1024 1024 512
```

- Не профилировать вызовы CUDA API методов на хосте

```
$nvprof -o nvprof.trace ./matmul 1024 1024 512
```

- Записать результаты профилирования в файл со специальным форматом, которые можно экспортировать в nvvp

CUDA-потоки

Когда их стоит использовать?

cudaStream

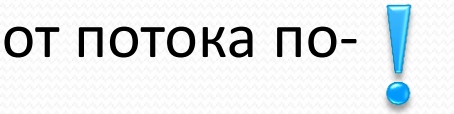
- Последовательность команд для GPU (запуски ядер, копирования памяти и т.д.), исполняемая строго последовательно
  - следующая команда выполняется после полного завершения предыдущей

cudaStream

- Пользователь сам создает потоки и распределяет команды по ним
- По-умолчанию, все команды помещаются в «Default Stream», равный нулю

cudaStream

- Только команды из разных потоков, отличных от потока по-умолчанию, могут выполняться параллельно



- Пользователь сам задет необходимую синхронизацию между командами из разных потоков (при наличии зависимостей)
  - В общем случае, порядок выполнения команд из разных потоков не определен

Создание и уничтожение

```
cudaStream_t stream;
cudaStreamCreate(&stream);
…
cudaStreamDestroy(stream);
```

- Поток привязывается к текущему активному устройству
  - Перед отправлением команды нужно переключаться на устройство, к которому привязан поток
  - Если попробовать отправить в него команду при другом активном устройстве, будет ошибка

Асинхронное копирование

Когда возвращается управление хостовой нити

	Host->device		Device->host		host-host	dev-dev
	pageable	**pinned**	pageable	**pinned**		
memcpy	После копирования в буфер*	После полного завершения	После полного завершения	После полного завершения	После полного завершения	сразу
memcpyAsync	После копирования в буфер	**сразу**	После полного завершения	**сразу**	После полного завершения	сразу

*В начале работы неявно вызывается cudaDeviceSynchronize

Параллельное выполнение команд

Параллельная работа хоста и устройства

- Ядра выполняются асинхронно
- Копирование между pinned-памятью и памятью устройства при помощи `cudaMemcpyAsync` также выполняется асинхронно

=> Добиться параллельной работы хоста и устройства достаточно просто!

Параллельная работа хоста и устройства

Пример:

```
cudaMallocHost(&aHost, size);
cudaMemcpyAsync( aDev, aHost, size,
          cudaMemcpyHostToDevice);
kernel<<<grid, block>>>(aDev, bDev);
cudaMemcpyAsync( bHost,
          cudaMemcpyHostToDevice);
doSomeWorkOnHost();
```

doSomeWorkOnHost будет выполняться параллельно с копированиями и выполнением ядра

Параллельное выполнение команд на GPU

- Команды из разных потоков, отличных от потока по-умолчанию, могут исполняться параллельно
  - В зависимости от аппаратных возможностей

- Возможные случаи:
  - Параллельные копирование и выполнение ядра
  - Параллельные выполнение ядер
  - Параллельные копирования с хоста на устройство и с устройства на хост

Копирование & выполнение ядра

- Если `cudaDeviceProp::asyncEngineCount > 0` устройство может выполнять параллельно копирование и счет ядра
  - Хостовая память долна быть page-locked

```
cudaMallocHost(&aHost, size);
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2); cudaMemcpyAsync(
aDev, aHost, size,
          cudaMemcpyHostToDevice, stream1);
kernel<<<grid, block, 0, stream2>>>(…);
```

Параллельное выполнение ядер

- Если `cudaDeviceProp::concurrentKernels` > 0 устройство может выполнять ядра параллельно

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2); kernel1<<<grid,
block, 0,
                              stream1>>>(data_1);
kernel2<<<grid, block, 0,
                              stream2>>>(data_2);
```

Копирование в обе стороны & выполнение ядра

- Если `cudaDeviceProp::asyncEngineCount == 2` устройство может выполнять параллельно копирование в обе стороны и счет ядра

```
cudaMallocHost(&aHost, size);
cudaMallocHost(&bHost, size);
// создать потоки
cudaMemcpyAsync( aDev, aHost, size,
         cudaMemcpyHostToDevice, stream1);
cudaMemcpyAsync( bHost, bDev, size,
         cudaMemcpyDeviceToHost, stream2);
kernel<<<grid, block, 0, stream3>>>(…);
```

Обозначения

- Ядро = код для GPU, __global void kernel(…) {}
- Размер ядра = длительность вычисления отдельной нитью
- Большое, сложное ядро = ядро большого размера
- Выполнение ядра = выполнение ядра на некотором гриде
- Запуск ядра = команда запуска ядра на гриде
- Длительный запуск ядра = запуск ядра, который долго выполняется
  - Время вычисления ~ размер ядра * размер грида

Примеры

- Рассмотрим классическую схему работы с GPU:
  - Копирование входных данных на GPU
  - Выполнение ядра
  - Копирование результатов обратно на хост
- Необходима синхронизация между командами, т.е. следующая выполняется после завершения предыдущей

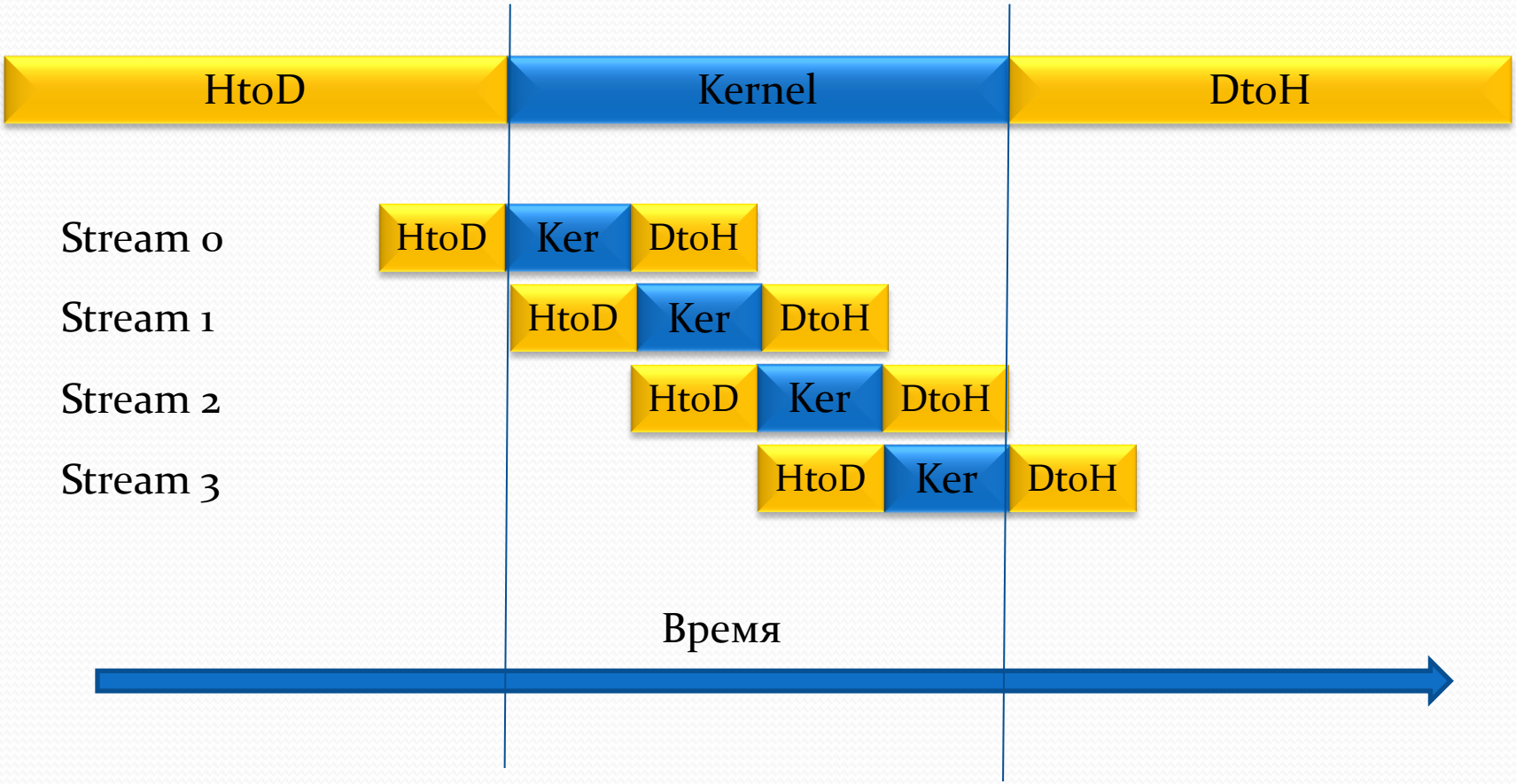
Идеальный случай

- Выполнения копирований сопоставимы по времени с выполнением ядра

- Разобьем задачу на подзадачи
  - Разделим грид на части и запустим то же ядро на подгридах – результат не изменится
  - Подзадаче нужна только часть данных для старта
  - Запустим подзадачи в разных потоках

Идеальный случай

- Запустим подзадачи в разных потоках



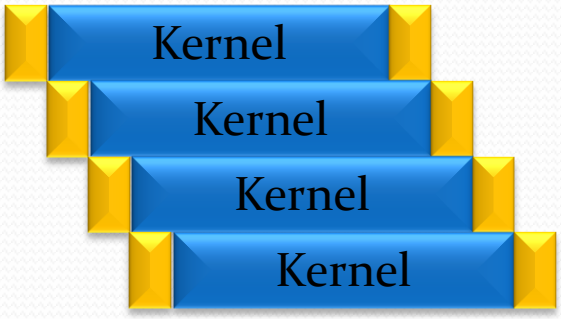
Идеальный случай

- Выполнение первой подзадачи начнется сразу после копирования нужной ей части данных
- Выполнение ядер будет происходить параллельно с копированиями
- Параллельное копирование в обе стороны, если аппаратура позволяет

Макимальное ускорение: $\dfrac{3}{1+\dfrac{2}{\text{число потоков}}}$

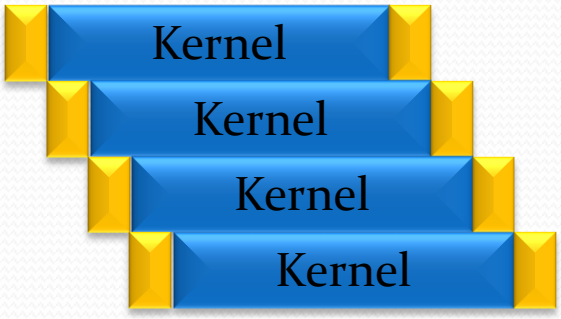
Небольшие копирования

Получится ли?



Небольшие копирования

Получится ли?



Нет!

Суммарное время выполнения ядра на подгридах никак не сделать меньше времени выполнения на целом гриде

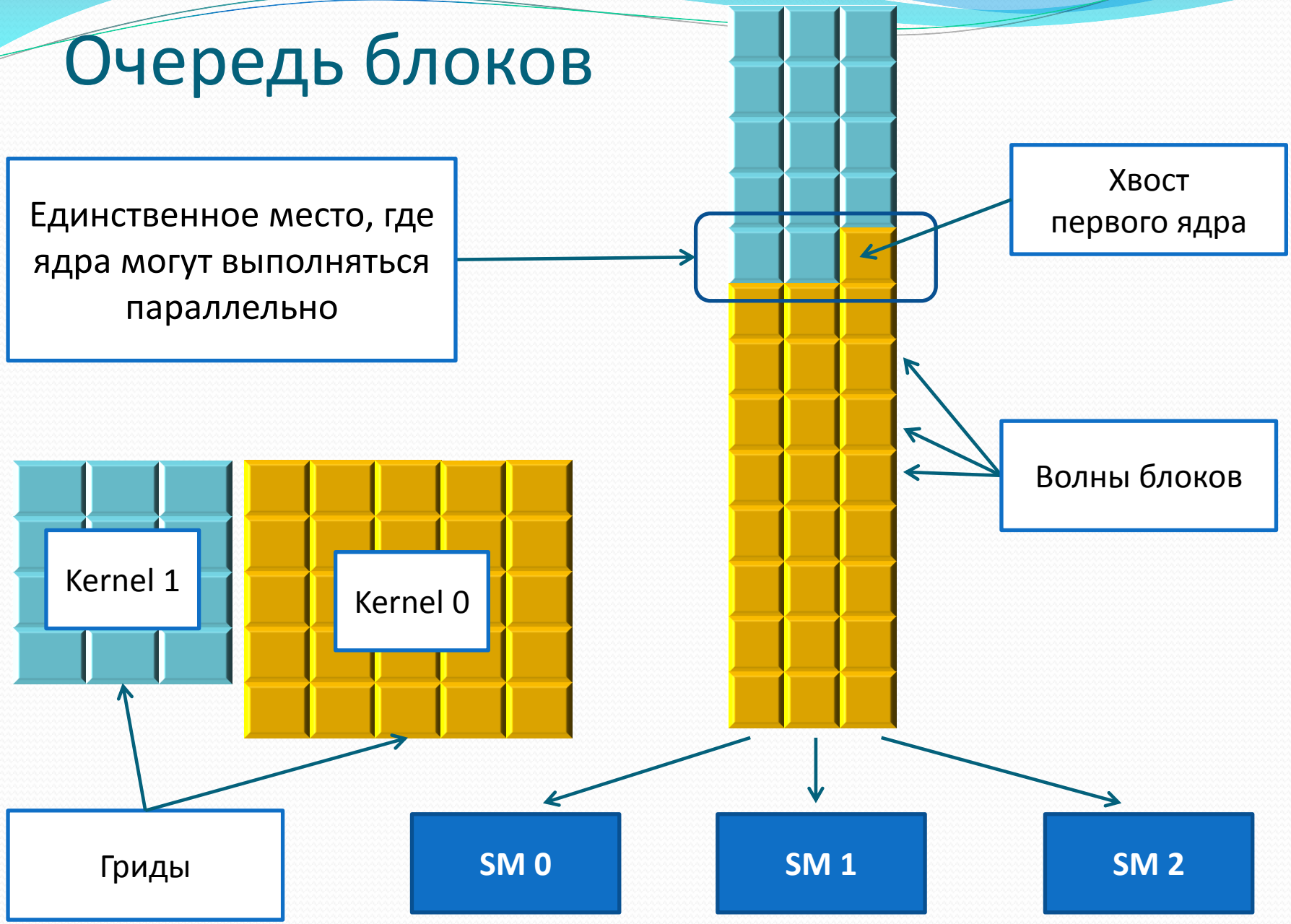
Параллельное выполнение ядер

- Ресурсы GPU ограничены – до 16 SM, до 1536 нитей на одном SM, до 8-ми блоков на SM

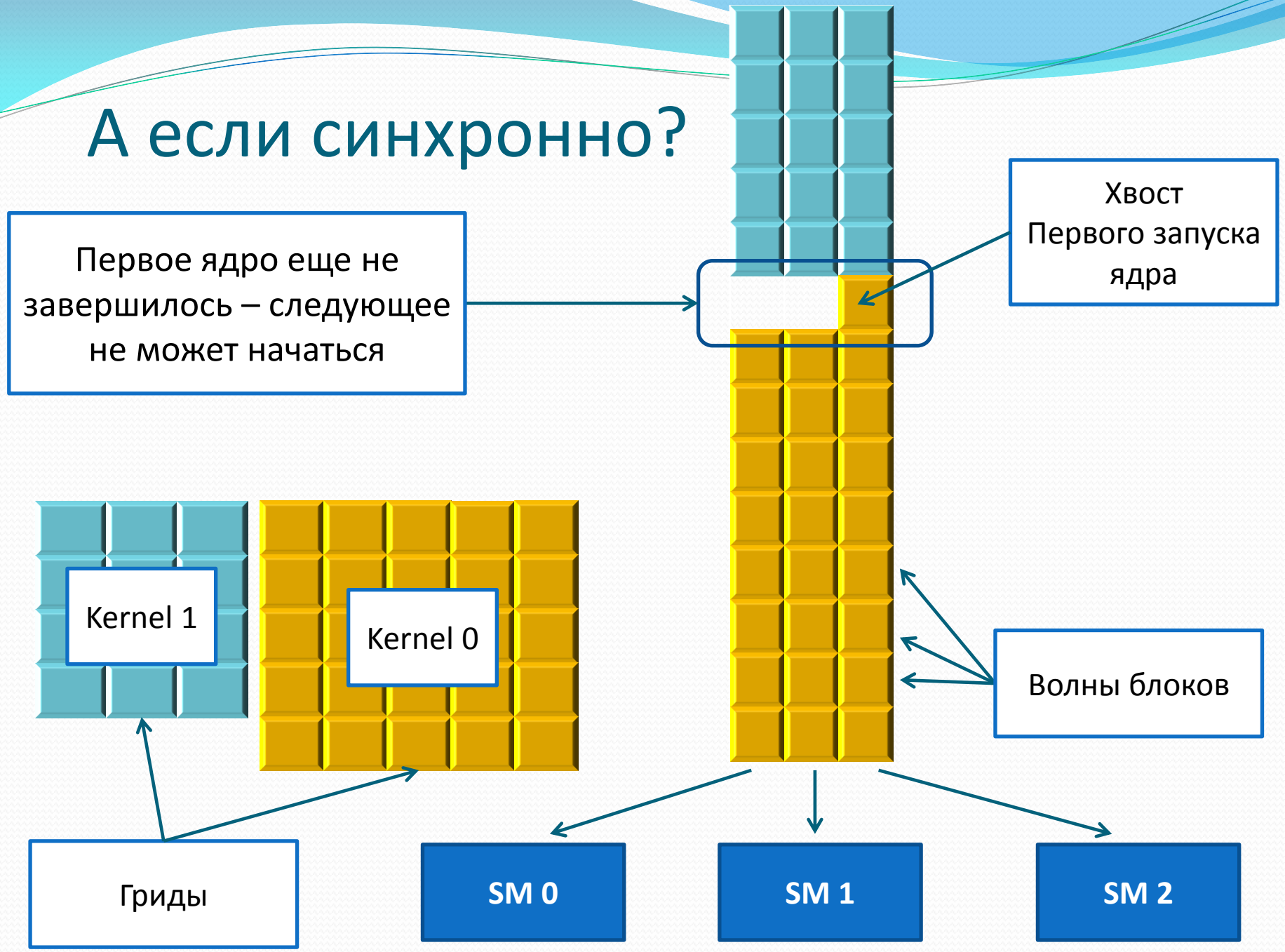
- Ядра запускаются на гридах из миллионов нитей = тысячи блоков
  - Блоки всех гридов попадают в одну общую очередь и выполняются по мере освобождения мультипроцессоров

Мультипроцессоры – «bottle neck» этой очереди

Очередь блоков

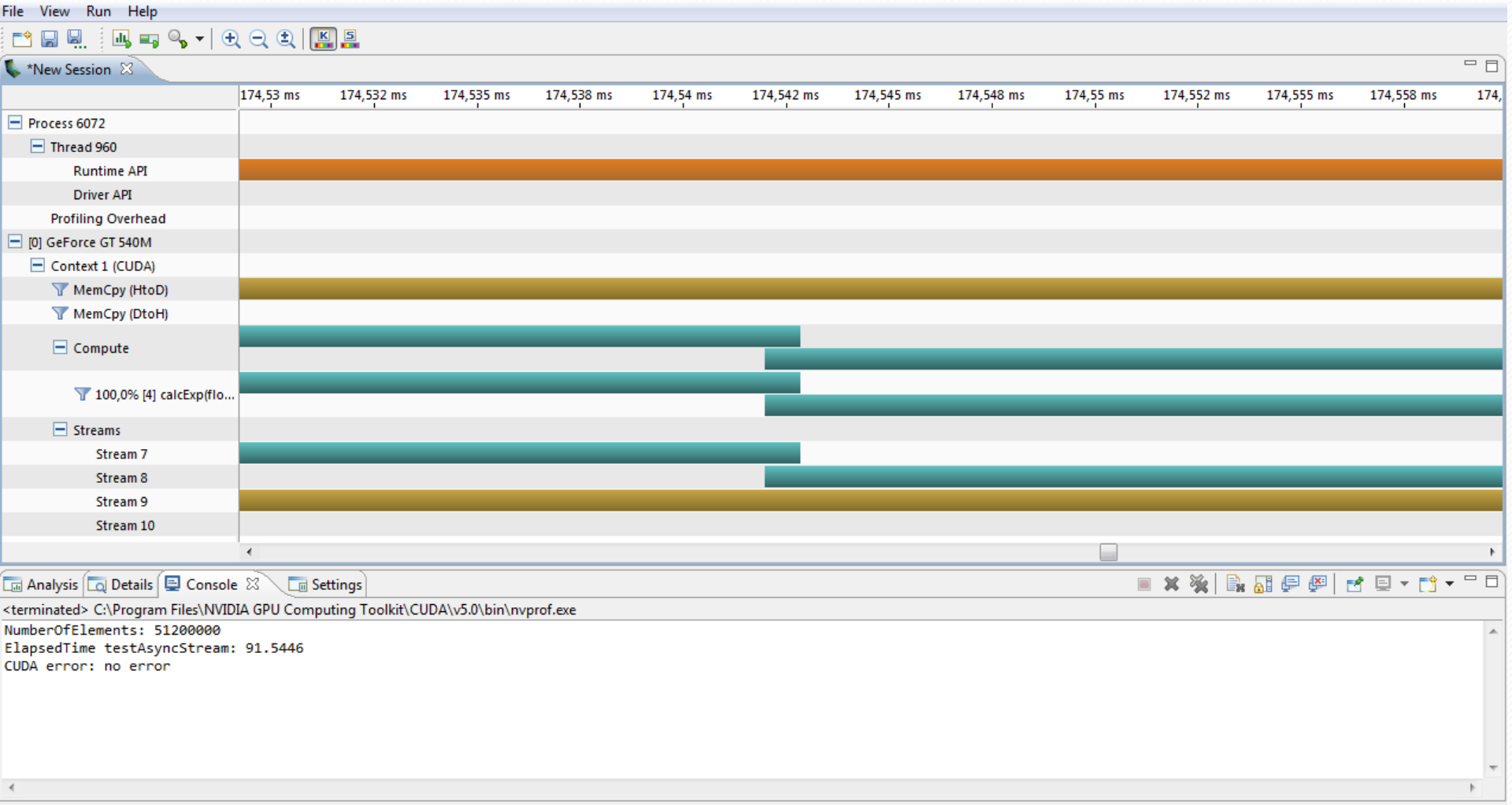


А если синхронно?



Пример в NVVP

- При большом приближении видно хвосты



Вывод

- При запуске двух ядер в разных потоках они могут выполняться параллельно только на границах
  - Когда последняя волна блоков (хвост) первого из запущенных ядер не полностью загружает устройство

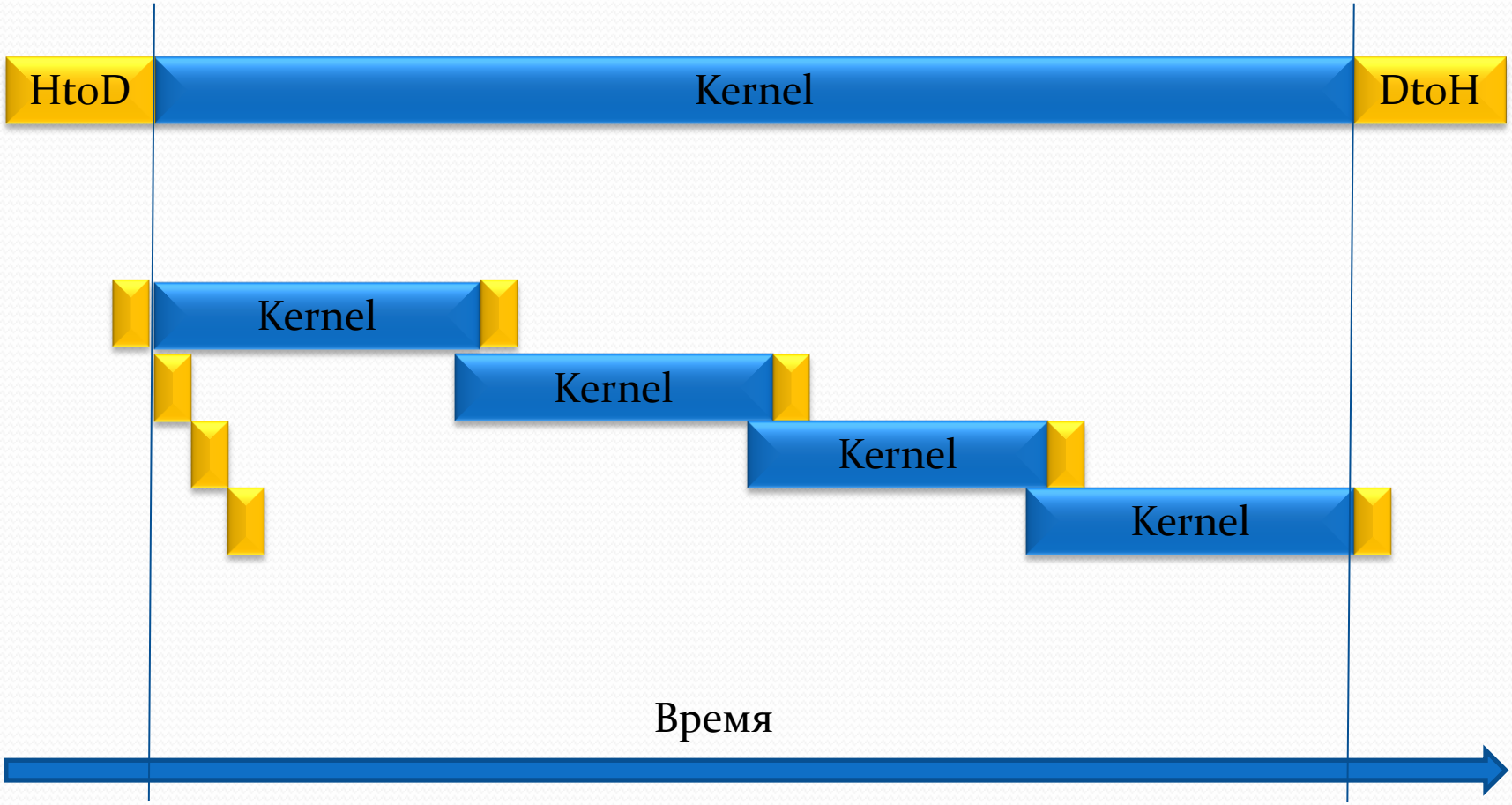
- Суммарное время выполнения одинаковых по сложности ядер, запущенных в разных потоках:

$$\frac{\text{суммарное число блоков} * \text{время выполнения блока}}{\text{число мультипроцессоров} * \left(\frac{1536}{\text{размер блока}} * occupancy\right)}$$

Вывод

- Суммарное время отдельных выполнений ядра на подгридах в разных потоках равно времени выполнения на целом гриде
  - Суммарное число блоков не меняется
- Суммарное время отдельных выполнений ядра на подгридах в одном потоке >= времени выполнения на целом гриде
  - Из-за простоя на границах

Небольшие копирования



Вывод

- При разбиении задачи на подзадачи выигрыш можем получить только за счет
  - Параллельного выполнения ядер и копирований
  - Параллельного выполнения копирований

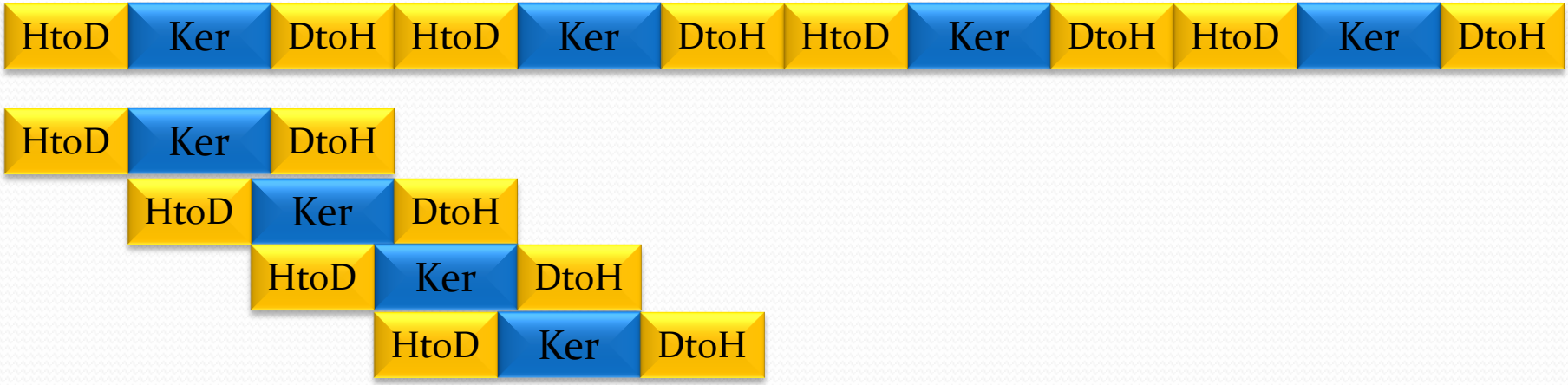
$\Rightarrow$При небольших копированиях

не имеет смысла возиться с потоками для подзадач

Множество задач

- При распределении по потокам результат зависит от
  - Соотношения копирования / счет
  - Размера гридов

Ядра запускаются на больших гридах



- Гриды большие => хвосты запусков ядер занимают малую долю в общем число блоков => доля параллельного выполнения невелика
  - Выигрываем в основном за счет параллельных копирований

Мало копирований – мало ускорения!

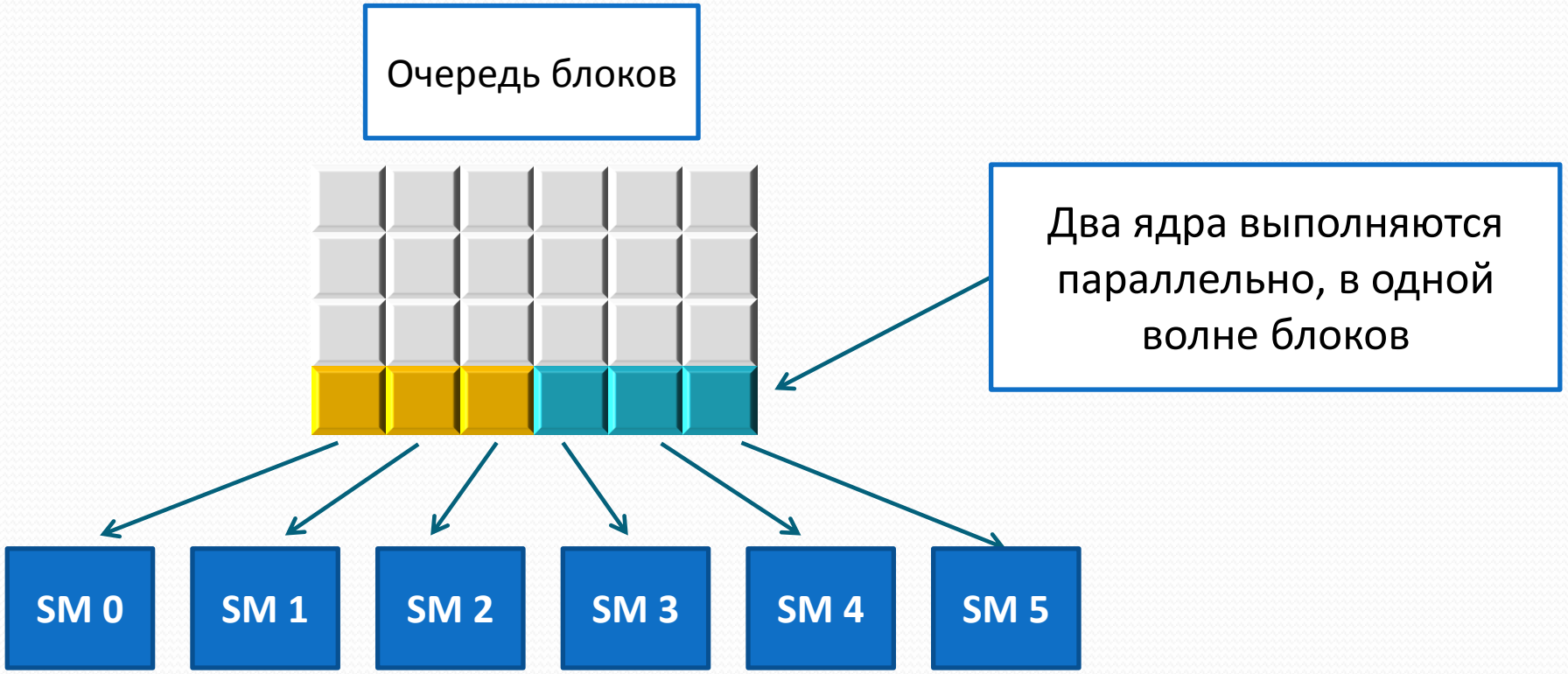
Ядра запускаются на малых гридах

- Пусть ядра запускаются на гридах такого размера, что ресурсов хватает для размещения всех блоков нескольких гридов
  - Например двух
- Пусть ядра примерно одинаковой сложности

Тогда два ядра, запущенные таких гридах, будут выполняться параллельно!

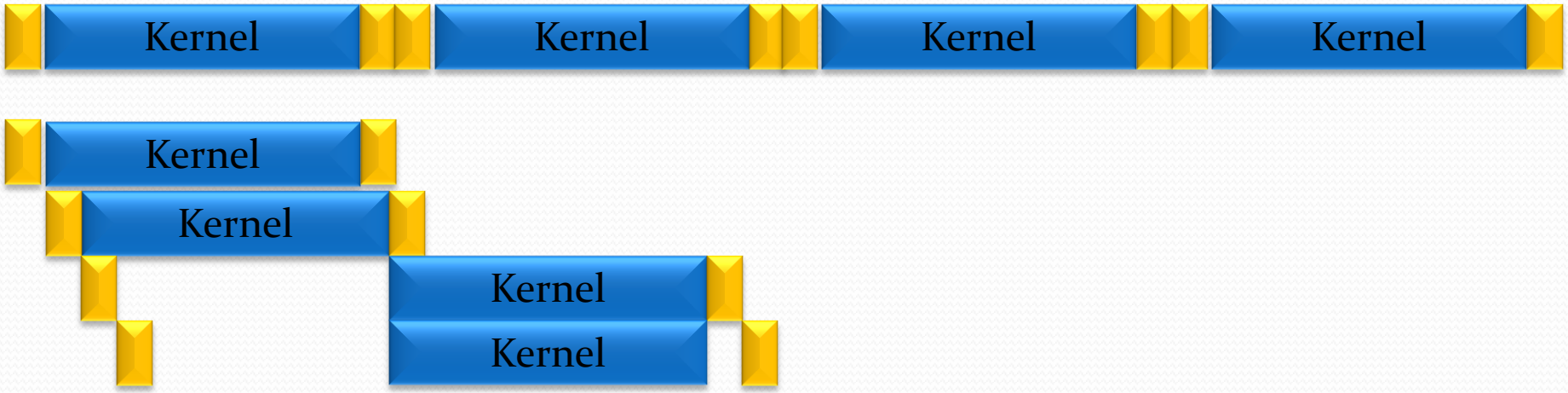
Ядра запускаются на малых гридах

- Ядра запускаются на трех блоках по 1024 нити, устройство состоит из 6 SM



Ядра запускаются на малых гридах

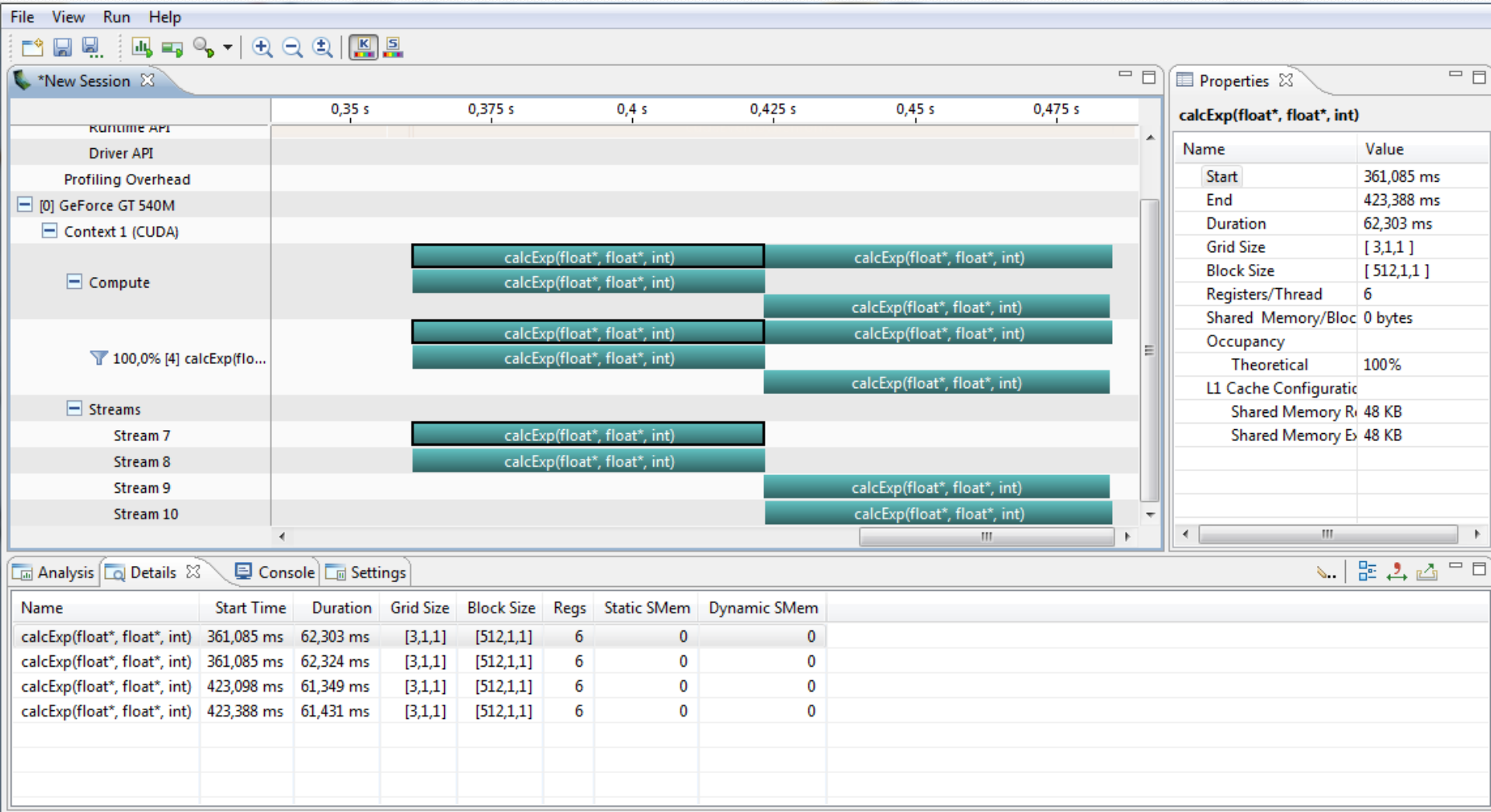
- Ядра запускаются на трех блоках по 1024 нити, устройство состоит из 6 SM



Можем получить ускорение даже при небольших копированиях!

Пример в NVVP

- Небольшие гриды



Дополнительные проблемы

- Устройство не поддерживает параллельное копирование в обе стороны
- В рассматриваемой архитектуре одна аппаратная очередь блоков

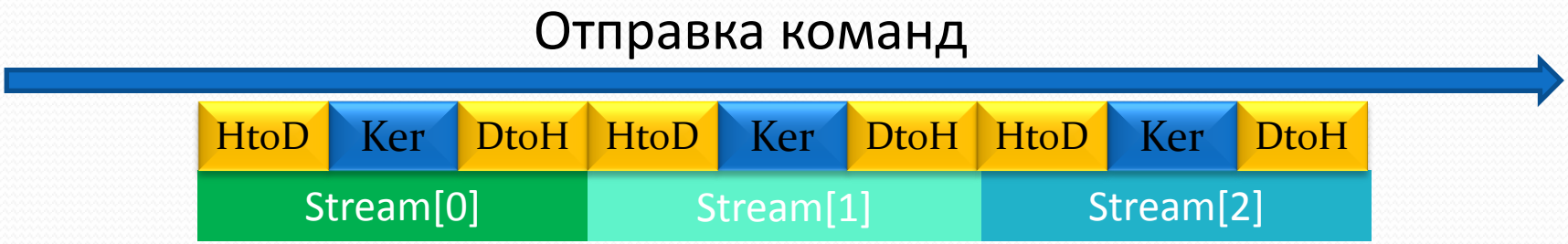
Важен порядок отправки команд!

Нет параллельных копирований

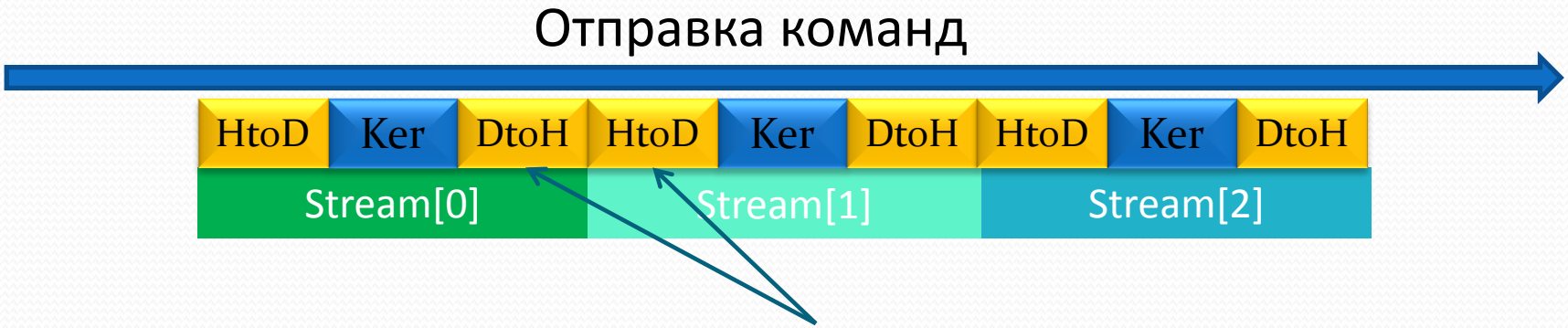
- Отправка команды копирования блокирует старт выполнения всех копирований в другую сторону, отправляемых после неё в любые потоки
- Ускорение только за счет параллельного копирования и выполнения ядер

Модельный пример

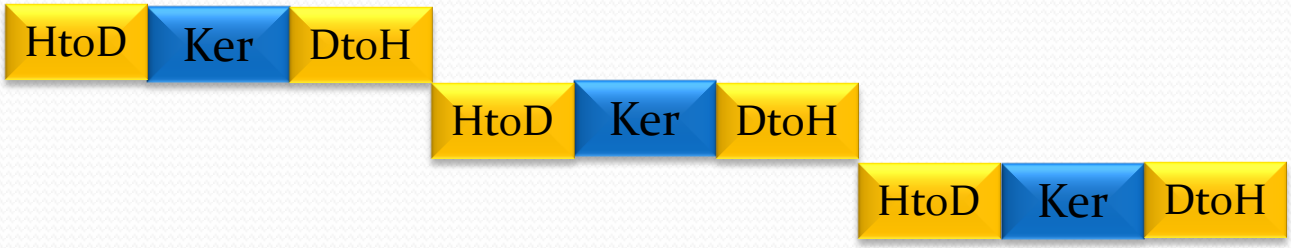
```
for (int i = 0; i < 3; ++i) {
   cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i *
   size, size, cudaMemcpyHostToDevice, stream[i]);
   MyKernel <<<100, 512, 0, stream[i]>>> (outputDevPtr
       + i * size, inputDevPtr + i * size, size);
   cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i
    * size, size, cudaMemcpyDeviceToHost, stream[i]);
}
```



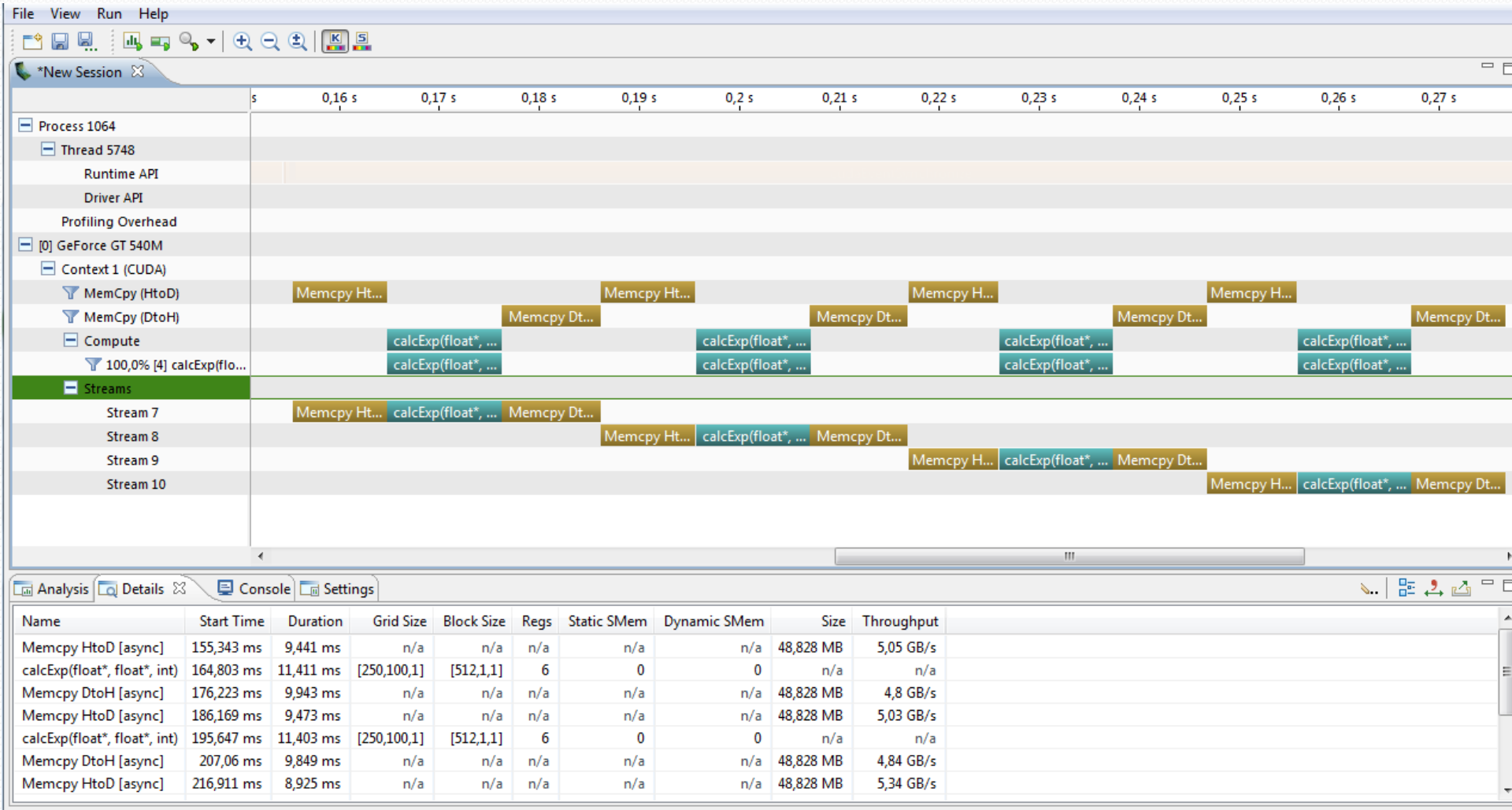
Нет параллельных копирований



Не будут выполняться параллельно

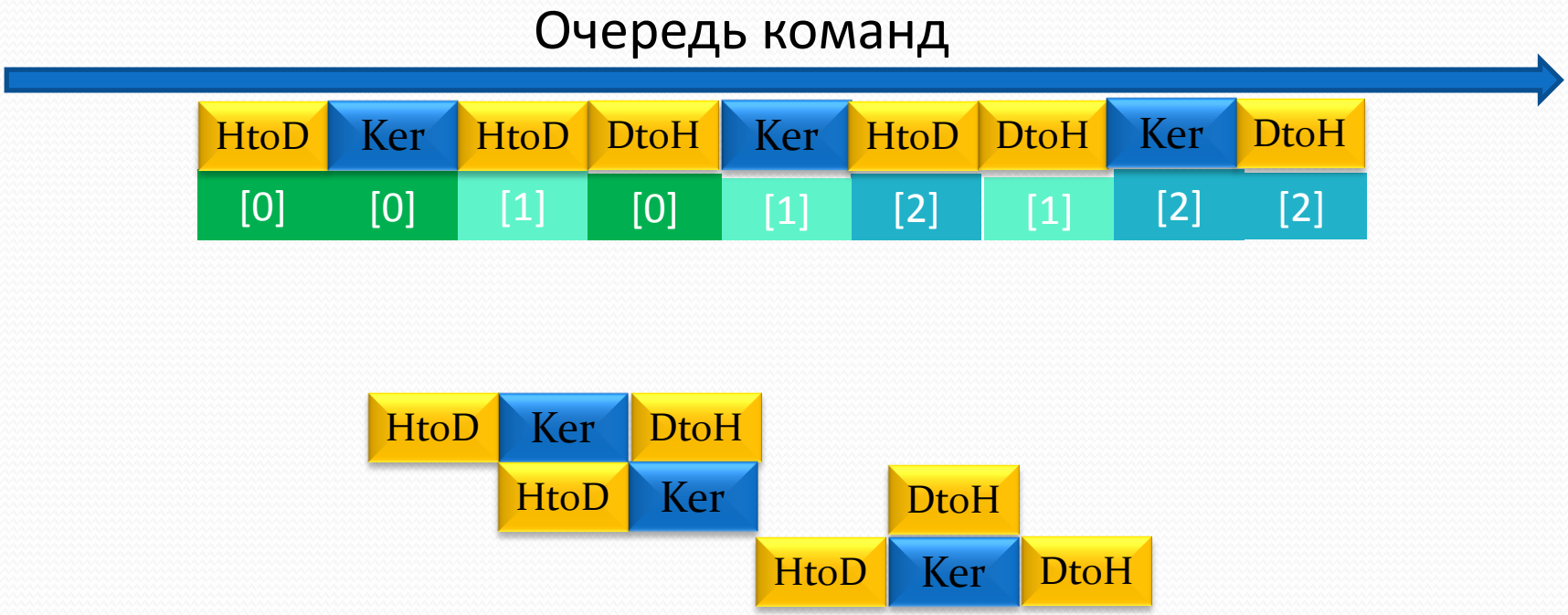


Пример в NVVP

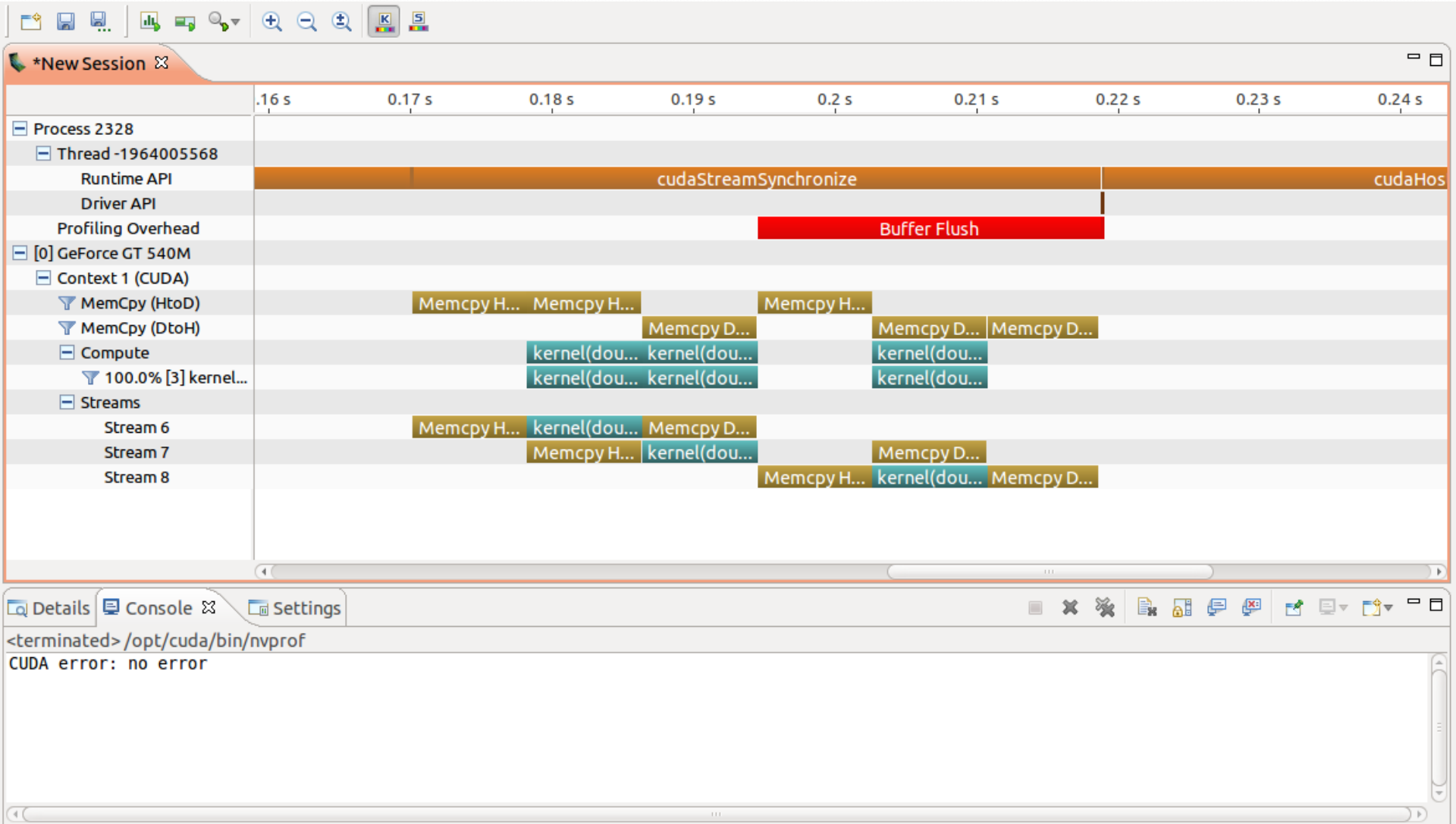


Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput
Memcpy HtoD [async]	155,343 ms	9,441 ms	n/a	n/a	n/a	n/a	n/a	48,828 MB	5,05 GB/s
calcExp(float*, float*, int)	164,803 ms	11,411 ms	[250,100,1]	[512,1,1]	6	0	0	n/a	n/a
Memcpy DtoH [async]	176,223 ms	9,943 ms	n/a	n/a	n/a	n/a	n/a	48,828 MB	4,8 GB/s
Memcpy HtoD [async]	186,169 ms	9,473 ms	n/a	n/a	n/a	n/a	n/a	48,828 MB	5,03 GB/s
calcExp(float*, float*, int)	195,647 ms	11,403 ms	[250,100,1]	[512,1,1]	6	0	0	n/a	n/a
Memcpy DtoH [async]	207,06 ms	9,849 ms	n/a	n/a	n/a	n/a	n/a	48,828 MB	4,84 GB/s
Memcpy HtoD [async]	216,911 ms	8,925 ms	n/a	n/a	n/a	n/a	n/a	48,828 MB	5,34 GB/s

Если поменять порядок запусков



Пример в NVVP



Единственная аппаратная очередь

- Если в поток отправлен запуск ядра, то все последующие запуски команд в тот же поток должны начать выполняться только после его полного завершения
- В устройствах с Compute Capability <= 3.0 одна аппаратная очередь блоков, в которую попадают блоки всех запусков ядер во всех потоках
  - Можно быть уверенным, что какой-либо запуск ядра полностью отработал, только когда в очереди больше нет блоков!

Единственная аппаратная очередь

Если в поток отправлен запуск ядра, то все последующие запуски команд в тот же поток должны начать выполняться только после его полного завершения

+

Можно быть уверенным, что какой-либо запуск ядра полностью отработал, только когда когда в очереди больше не блоков

=

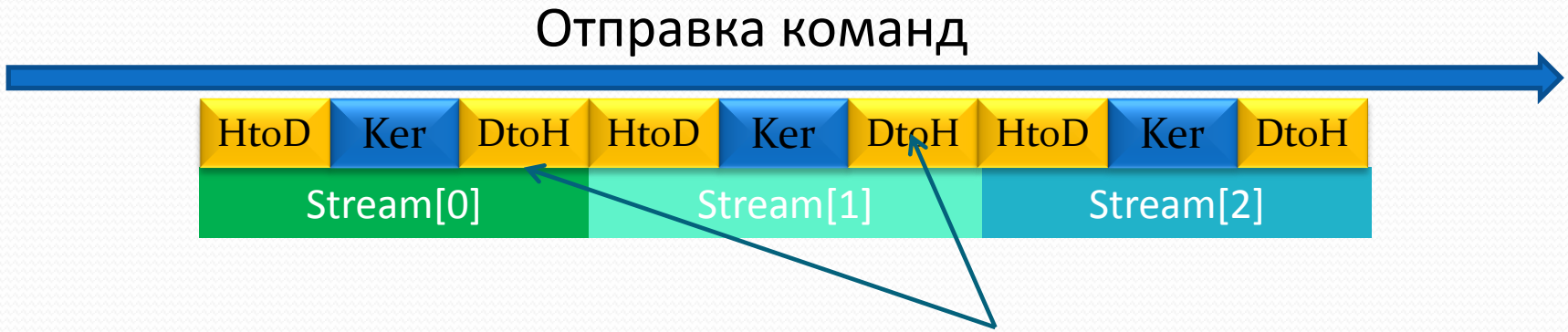
Неявная синхронизация перед стартом выполнения зависимой от запуска ядра команды:

- Начало выполнения команды откладывается до момента, когда в очереди не останется блоков
- Добавление новых блоков в очередь приостанавливается, пока команда не начнет выполняться

Единственная аппаратная очередь

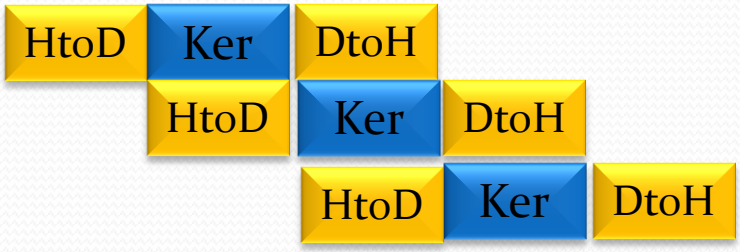
- Начало выполнения зависимой от запуска ядра команды откладывается до момента, когда в очереди не останется блоков
  - Зависимая от запуска ядра команда может параллельно выполняться только с последней волной запуска ядра в другом потоке
- Добавление новых блоков в очередь приостанавливается, пока зависимая от запуска ядра команда не начнет выполняться
  - Запуски всех ядер во всех потоках приостанавливаются до момента, когда полностью отработает запуск-зависимость.

Единственная аппаратная очередь

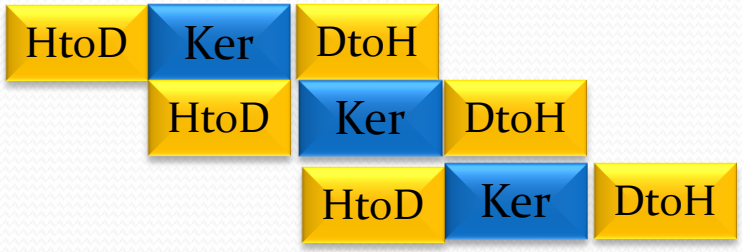


DtoH в блокирует все последующие запуски ядер

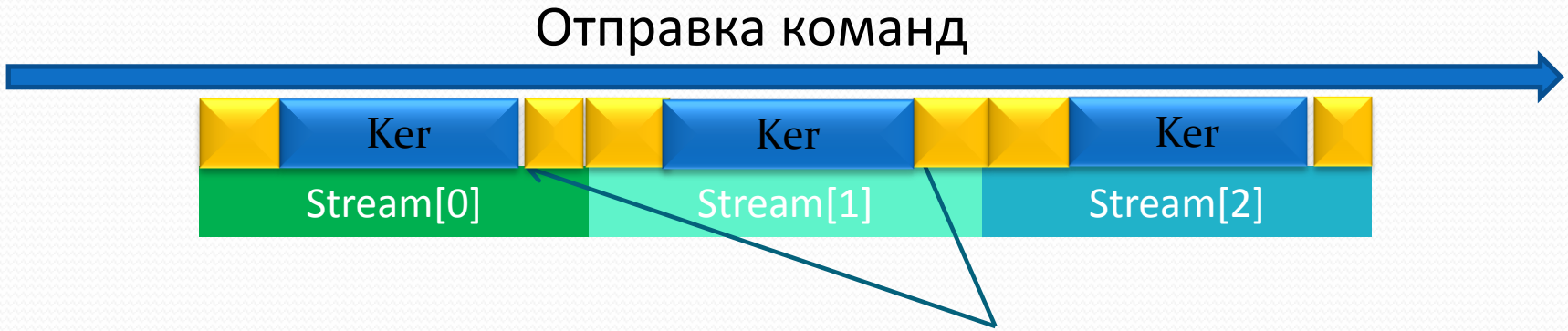
Ожидание



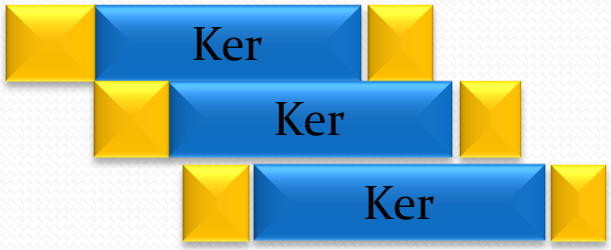
Реальность



Единственная аппаратная очередь

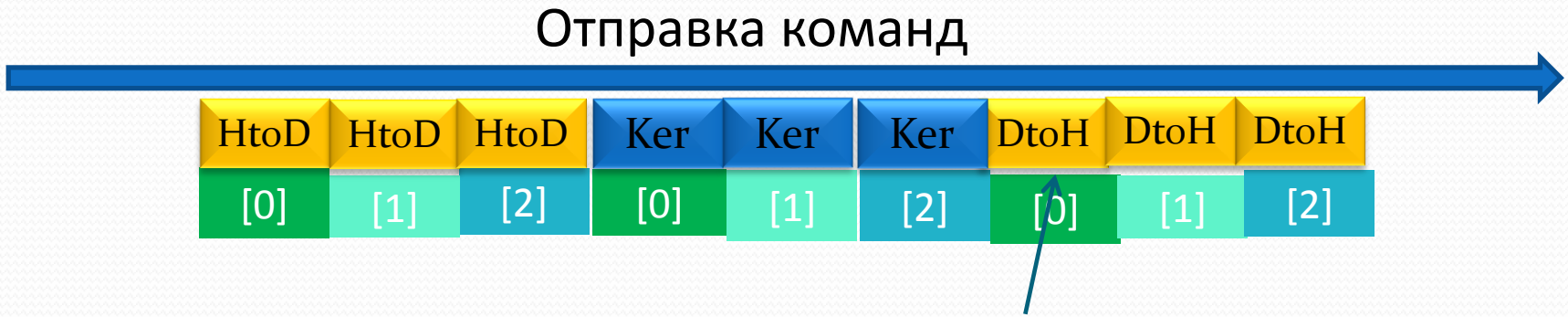


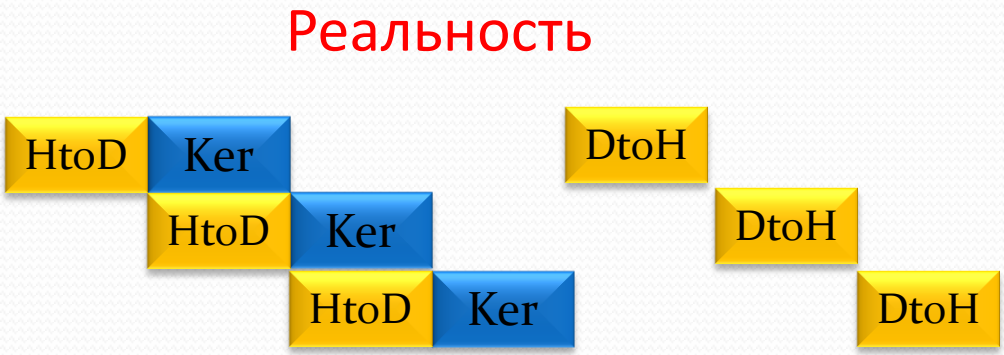
Ожидание



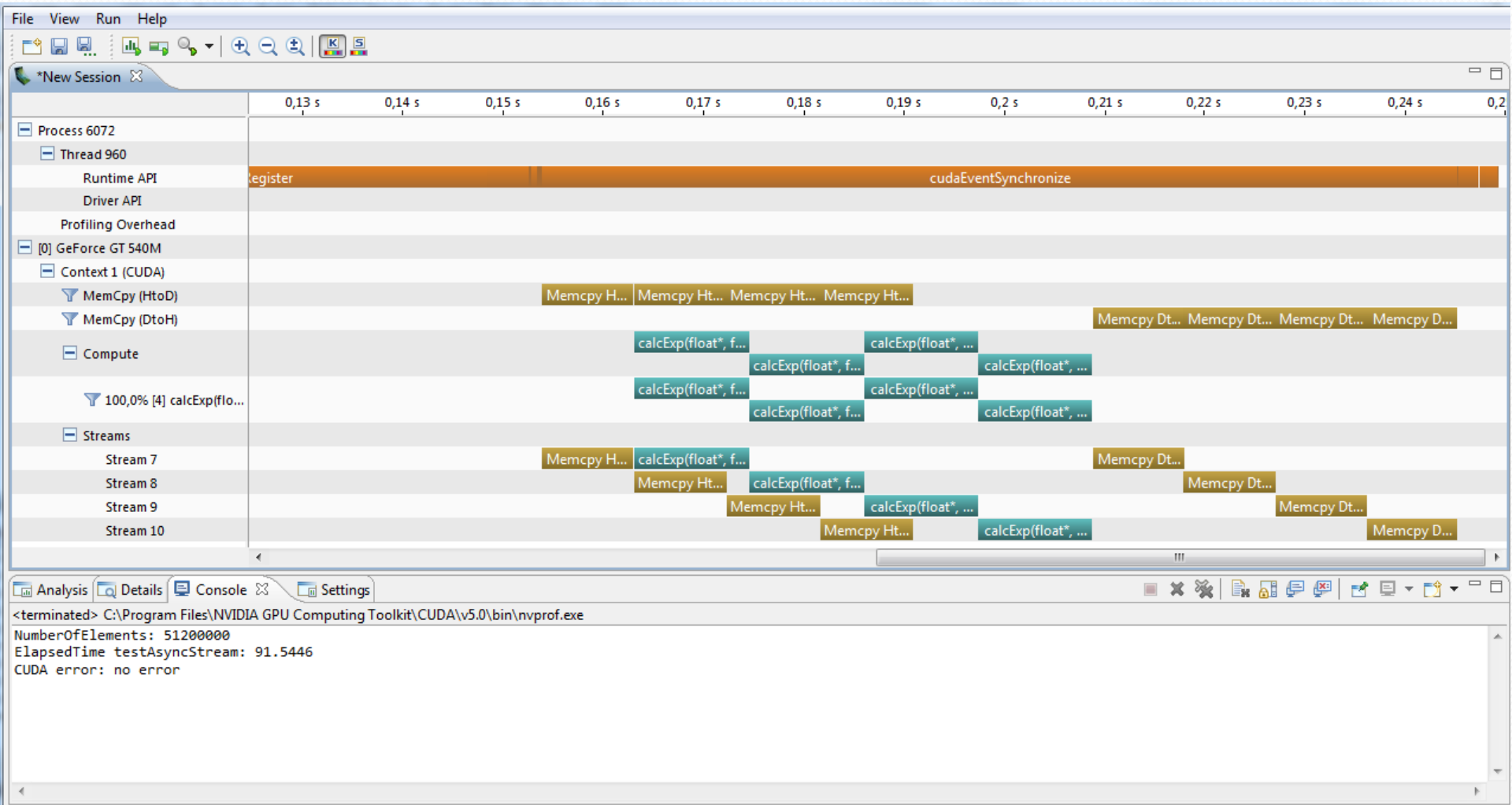
Реальность

Единственная аппаратная очередь





Пример в NVVP



Вывод

- Мало копирований с запусками ядер на больших гридах – с потоками лучше не возиться
- Аккуратно отправлять команды на GPU

Средства синхронизации CUDA-потоков

Неявная синхронизация

- Неявная синхронизация (ожидание завершения всех команд на устройтве) выполняется перед:
  - Выделением page-locked памяти / памяти на устройстве
  - cudaMemSet
  - Копированием между пересекающимися областями памяти на устройстве
  - Отправкой команды в поток по-умолчанию
  - Переключением режима кеша L1

- Если между отправкой двух команд в разные потоки стоит что-то из этого списка – параллельного выполнения не будет

События (cudaEvent)

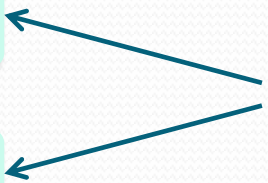
- Маркеры, приписываемые «точкам программы»
  - ✓ Можно проверить произошло событие или нет
  - ✓ Можно замерить время между двумя произошедшими событиями
  - ✓ Можно синхронизоваться по событию, т.е. заблокировать CPU-поток до момента его наступления
- «Точки программы» расположены между отправками команд на GPU

Запись события

```
cudaError_t cudaEventRecord (
                            cudaEvent_t event,
                            cudaStream_t stream = 0)
```

- Приписывает событие точке программы, в которой вызывается

```
cudaMemcpyAsync(…)
…;//код без запуска команд
kernel<<<….>>> (…);
cudaEventRecord(event, stream)
cudaMemcpyAsync(…)
```



Совершение события

- Событие происходит когда выполнение команд на GPU реально доходит до точки, к которой в последний раз было приписано событие

Совершение события

- Событие происходит когда завершаются все команды, помещённые в поток, к которому приписано событие, до последнего вызова cudaEventRecord для него

- Если событие приписано потоку по умолчанию (stream = 0), то оно происходит в момент завершения всех команд, помещённых во все потоки до последнего вызова cudaEventRecord для него

Синхронизация по событию

```
cudaError_t cudaEventQuery(cudaEvent_t event)
```

- Возвращает `cudaSuccess`, если событие уже произошло (вся работа до последнего `cudaEventRecord` выполнена): иначе **`cudaErrorNotReady`**

```
cudaError_t cudaEventSynchronize
                    (cudaEvent_t  event)
```

- Возвращает управление хостовой нити только после наступления события

Синхронизация на GPU

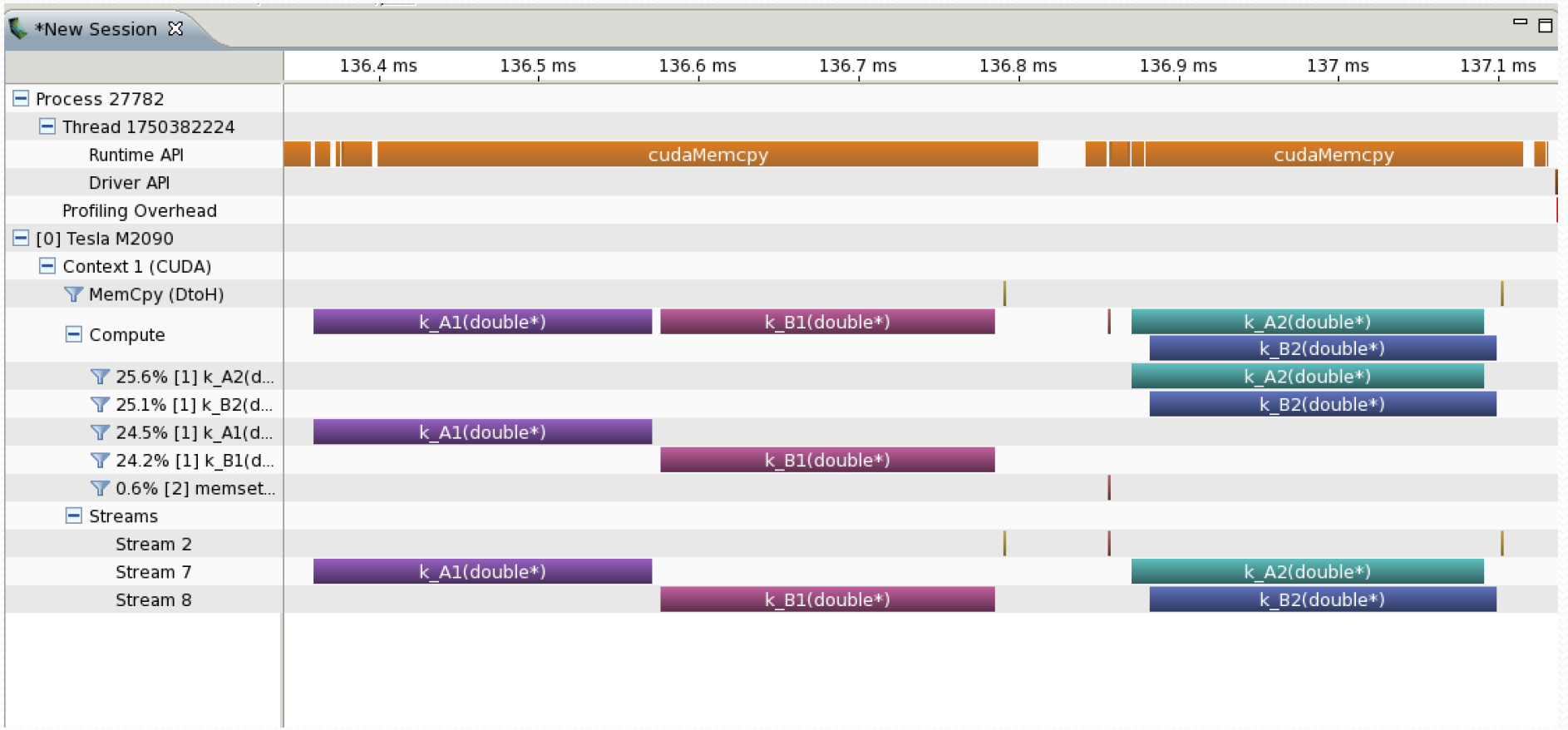
```
cudaError_t cudaStreamWaitEvent
      (cudaStream_t stream, cudaEvent_t event,
                  unsigned int  flags )
```

- Команды, отправленные в **stream** начнут выполняться после наступления события **event**
  - Синхронизация будет эффективно выполнена на GPU
  - При stream == NULL будут отложены все команды всех потоков
- Событие **event** может быть записано на другом GPU
  - Синхронизация между GPU

Синхронизация на GPU

```
k_A1<<<1,1,0,streamA>>>(d); // A1
cudaEventRecord(halfA,streamA);
cudaStreamWaitEvent(streamB,halfA,0);
k_B1<<<1,1,0,streamB>>>(d); // B1 начнется после
                               завершения A1
cudaEventRecord(halfB,streamB);
cudaStreamWaitEvent(streamA,halfB,0);
k_A2<<<1,1,0,streamA>>>(d); // A2 начнется после
                               завершения B1
k_B2<<<1,1,0,streamB>>>(d); // B2
```

Синхронизация на GPU



Синхронизация по потоку

```
cudaError_t cudaStreamQuery
                    (cudaStream_t stream)
```

```
cudaError_t cudaStreamSynchronize
                    (cudaStream_t event)
```

cudaStreamCallback

```
typedef void (*cudaStreamCallback_t)(
      cudaStream_t stream, cudaError_t status,
      void *userData )
```

```
cudaError_t cudaStreamAddCallback (
cudaStream_t stream, cudaStreamCallback_t callbac,
void *userData, unsigned int flags )
```

- **`callback`** будет вызван когда выполнятся все предшествующие команды, отправленные в поток
- В callback запрещены обращения к CUDA API

Multi-gpu

Использование нескольких GPU
CUDA+openmp
CUDA+MPI
P2P обмены между GPU

CUDA Context

CUDA Context

- Аналог процесса CPU
  - Выделения памяти, выполнение операций происходит в рамках некоторого контекста (=процесса)
  - Отдельное адресное пространство
- Выделенная память неявно освобождается при удалении контекста
- Операции из разных контекстов не могут выполнятся параллельно

CUDA context
- Адресное пространство
- Ресурсы
- Операции

CUDA Context

- Контексты устройств неявно создаются при инициализации CUDA-runtime
  - На каждом устройстве создается по одному контексту – «primary-контекст»
  - Все нити программы совместно их используют

- Инициализация CUDA-runtime происходит неявно, при первом вызове любой функции, не относящейся к Device / Version Management (см. Toolkit Reference Manual)

CUDA Context

- В каждой нити может быть только один активный контекст в каждый момент времени

`cudaSetDevice(n)` - переключение между устройствами (=между контекстами)

`cudaDeviceReset()` - уничтожает primary-контекст, активный в данный момент

  - При этом будет освобождена вся память, выделенная в контексте
  - При необходимости, новый контекст будет неявно создан в дальнейшем

Cuda Context & cudaStream/Event

- `cudaStream{Event}Create` создает соответствующий ресурс в активном контексте

- Если активный контекст отличен от того, в котором создан поток/событие:
  - Отправление команды в поток вызовет ошибку
  - `cudaEventRecord()` для события вызовет ошибку

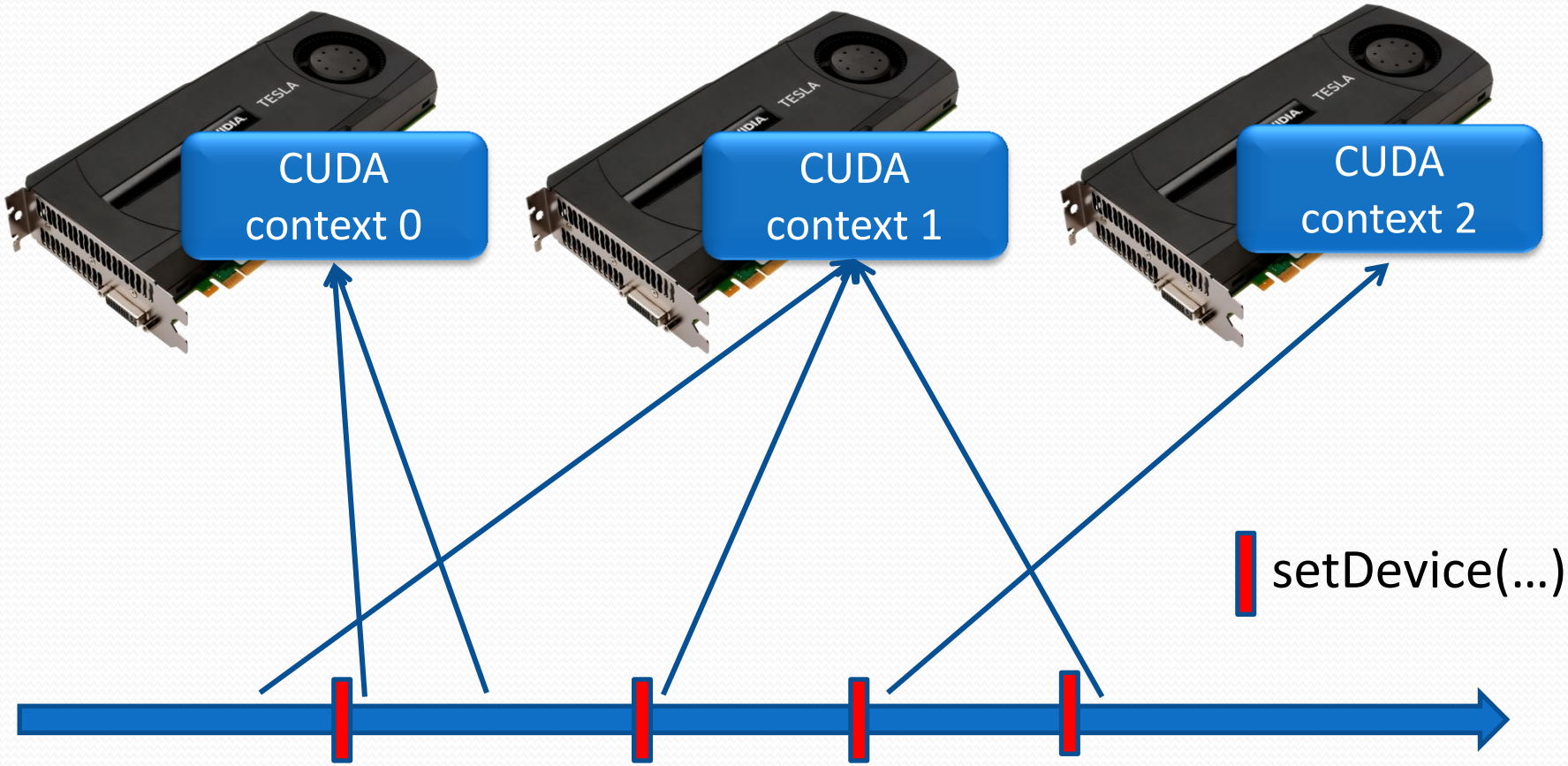
- `cudaEventElapsedTime()` вызовет ошибку, если события созданы в разных контекстах

Пример

```
cudaSetDevice(0);
cudaStream_t s0;
cudaStreamCreate(&s0); // создать поток на device 0
cudaSetDevice(1);// переключить контекст на device 1
cudaStream_t s1;
cudaStreamCreate(&s1); // создать поток на device 1
MyKernel<<<100, 64, 0, s1>>>();
MyKernel<<<100, 64, 0, s0>>>(); // ошибка
```

Multi-GPU
С одной хост-нитью

Multi-GPU & single CPU thread



Поток CPU переключается между контекстами

Модельная задача

```
float *devPtr = NULL, *hostPtr = NULL;
int n;
loadInputData(&n, &hostPtr);
cudaHostRegister(hostPtr, n*sizeof(float),
            cudaHostRegisterDefault);
cudaMalloc(&devPtr, n * sizeof(float));
cudaMemcpyAsync(devPtr, hostPtr, n*sizeof(float),
           cudaMemcpyHostToDevice, 0);
kernel<<<(n – 1) / 512 + 1, 512>>>(devPtr, n);
cudaMemcpyAsync(hostPtr, devPtr, n*sizeof(float),
                     cudaMemcpyDeviceToHost, 0);
cudaDeviceSynchronize();
```

Переписываем на multiGPU

```
float *hostPtr = NULL;
int n, deviceCount;
loadInputData(&n, &hostPtr);
cudaGetDeviceCount(&deviceCount);
float **devPtr = (float **)malloc(deviceCount *
                                  sizeof(float *));
```

- Получили число устройств
- Выделили массив указателей на GPU-память

Выделение памяти

- Выделение памяти через `cudaMalloc*` происходит на устройстве, к которому относится активный контекст
  - При определенных условиях память может быть доступна из ядер, работающих на других устройствах (**peer-to-peer**)

- `cudaHostRegister[Alloc](…)` лочит(выделяет) память в рамках активного контекста
  - Преимущества доступны другим контекстам только если pinned-память является **portable:**

```
cudaHostRegister(ptr, n, cudaHostRegisterMapped |
                         cudaHostRegisterPortable);
```

Выделение памяти

```
int elemsPerDevice = (n – 1) / deviceCount + 1;
for(int device = 0; device < deviceCount; device++) {
    cudaSetDevice(device);
    cudaMalloc(devPtr + device, elemsPerDevice *
                                        sizeof(float));
    cudaHostRegister(hostPtr + device * elemsPerDevice,
                     elemsPerDevice * sizeof(float),…);
}
```

Выделение памяти блокирует хост-нить!

- Рассчитали размер подзадач
- Выделили / залочили нужные объемы в каждом контексте

Отправка команд

```
for(int device = 0; device < deviceCount; device++) {
    int offset = device * elemsPerDevice;
    int elemCount = min(n – offset, elemsPerDevice);
    cudaSetDevice(device);
    cudaMemcpyAsync(devPtr[device], hostPtr + offset),
                    elemCount * sizeof(float),…, 0);
    kernel<<<(elemCount – 1)/512 + 1, 512>>>
                          (devPtr[device], elemCount);
    cudaMemcpyAsync(hostPtr + offset, devPtr[device],
                    elemCount * sizeof(float),…, 0);
}
```

- Асинхронно отправляем команды на устройства
  - Кажое GPU работает со своей порцией данных

Синхронизация

```
for(int device = 0; device < deviceCount; device++) {
    cudaSetDevice(device);
    cudaDeviceSynchronize();
}
```

- Ожидаем завершения всех команд на устройствах

Почему синхронизацию нужно делать в отдельном цикле?

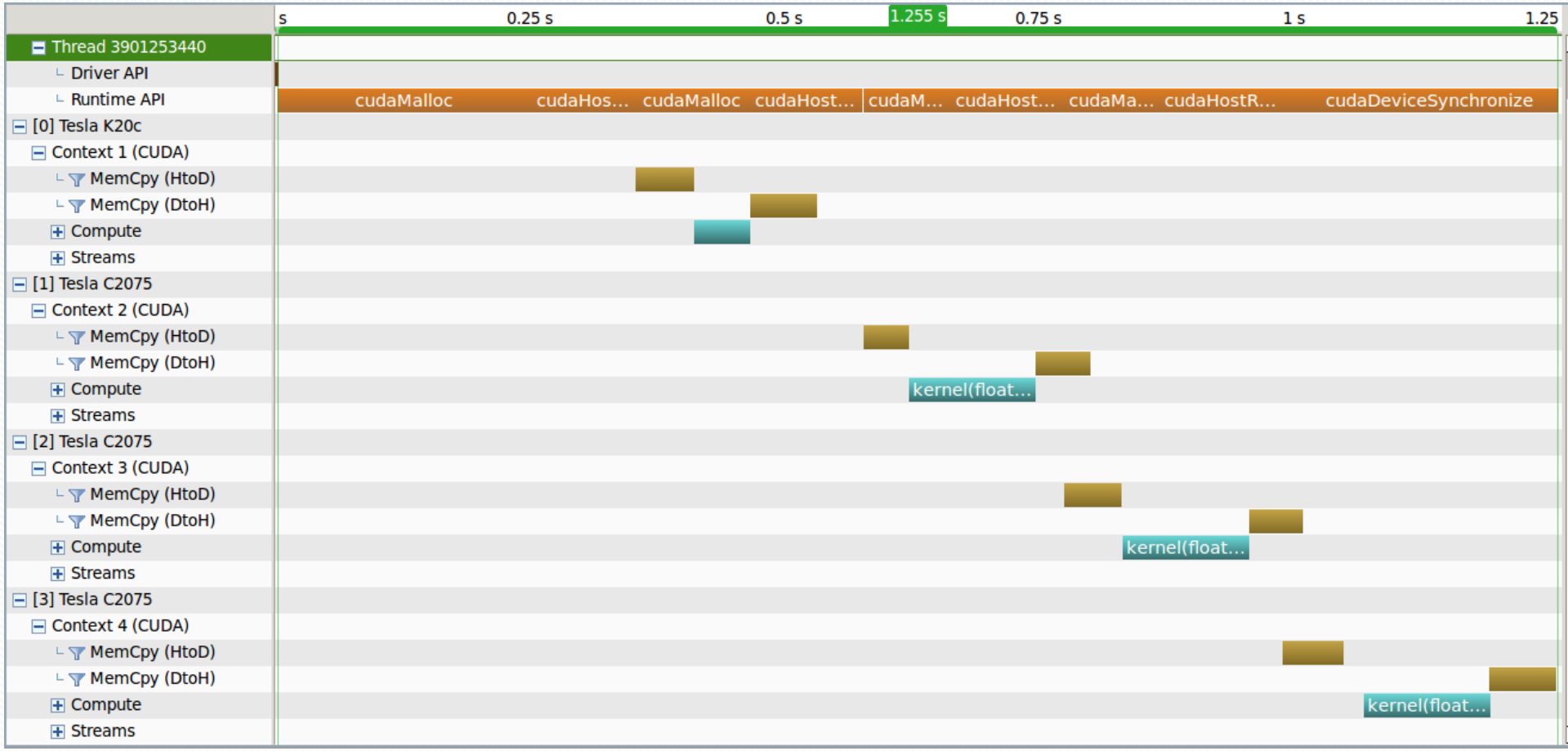
Результат (1.487 с)

Комментарий

- Неблокирующие запуски команд правильнее будет поместить в цикл с выделением памяти
  - Команды на первом GPU начнут выполняться пока на остальных выделяется память

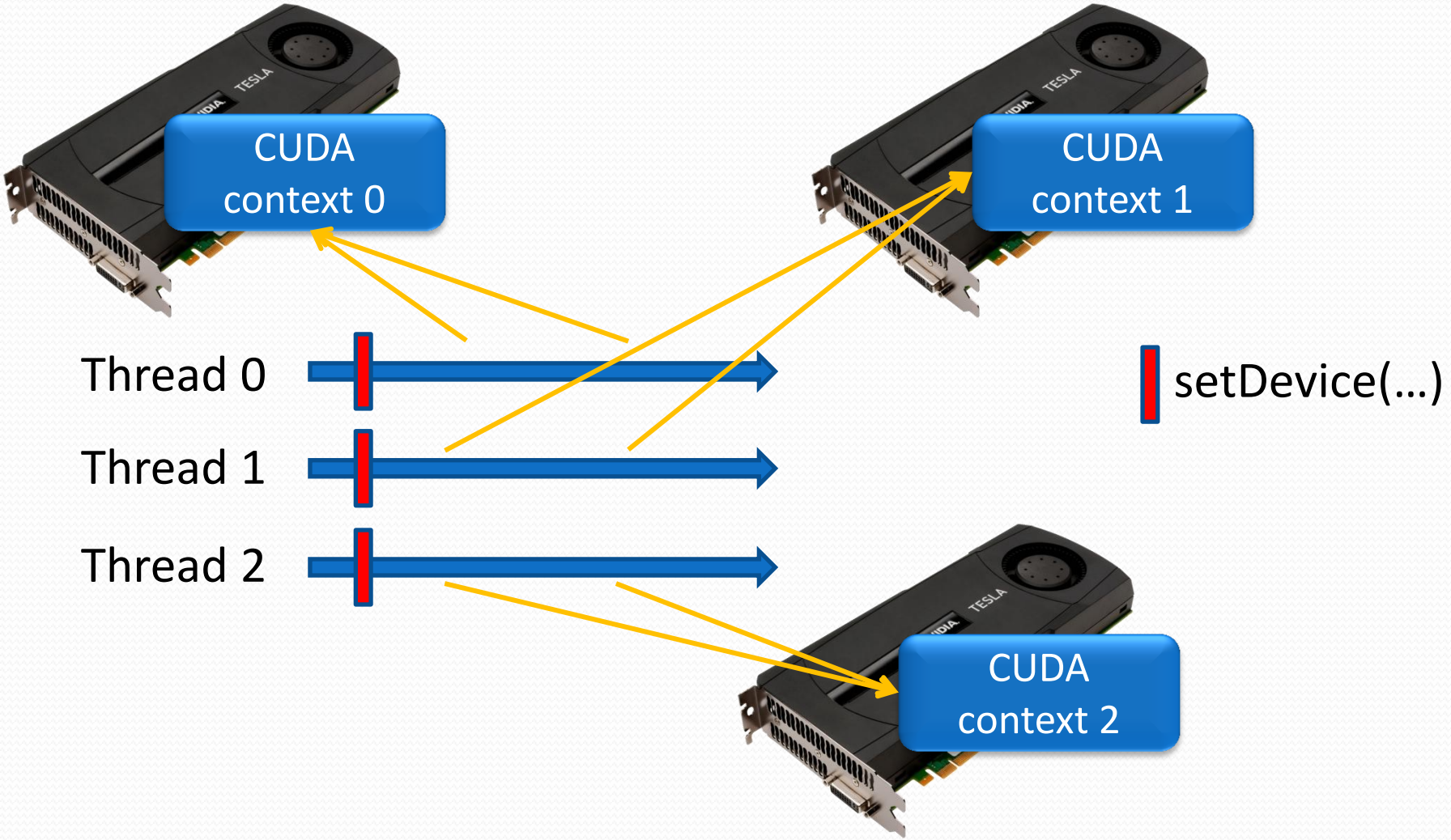
```
for(int device = 0; device < deviceCount; device++) {
    cudaSetDevice(device);
    cudaMalloc(…);
    cudaHostRegister(…);
    cudaMemcpyAsync(…); // pinned CPU <-> GPU
    kernel<<<…>>>(…);
    cudaMemcpyAsync(); // pinned CPU  <-> GPU
}
```

Результат (1.255 с)



Multi-GPU + OpenMP

Multi-GPU & multiple CPU threads



Компиляция

- Поддержка OpenMP встроена в популярные компиляторы
  - Intel icc/ifort, gcc/gfortran, MS cl, IBM xlc

- Обычный компилятор компилирует OpenMP директивы и функции при указании специального флага компиляции (для распознавания директив) и линковки (для линковки omp-функций)
  - icc –openmp
  - gcc –fopenmp
  - cl -/openmp
  - xlc -qsmp

Компиляция с NVCC

```
$nvcc –Xcompiler flag –arch=sm_20 main.cu
```

- Передает компилятору на стадию компиляции и на стадию линковки слово командной строки **flag**, следующее за –Xcompiler

```
$nvcc –Xcompiler –fopenmp –arch=sm_20 main.cu
```

- Компиляция CUDA+OpenMP на Linux (gcc)

Компиляция с NVCC

- Можно раздельно:

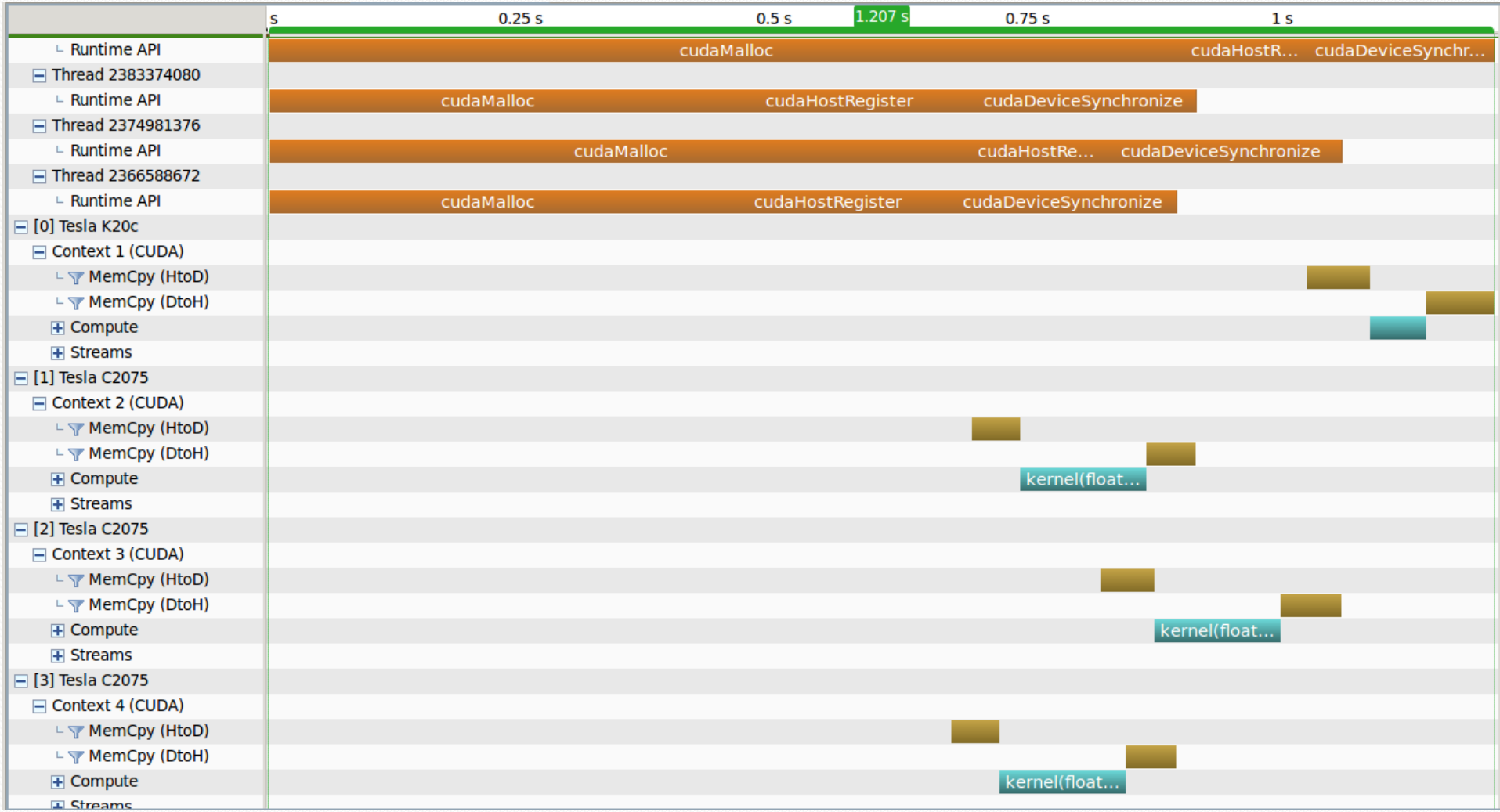
```
$nvcc –arch=sm_20 kernel.cu
$gcc -fopenmp –I/opt/cuda/include main.c
$gcc -fopenmp –L/opt/cuda/lib –lcudart
  main.o kernel.o
```

Переписываем под OpenMP

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
#pragma omp parallel num_threads(deviceCount)
{
    int device = omp_get_thread_num();
    cudaSetDevice(device);
    cudaMalloc(devPtr + device, elemsPerDevice *
                                sizeof(float));
    cudaHostRegister(…);
    … // прочие команды
    cudaDeviceSynchronize();
}
```

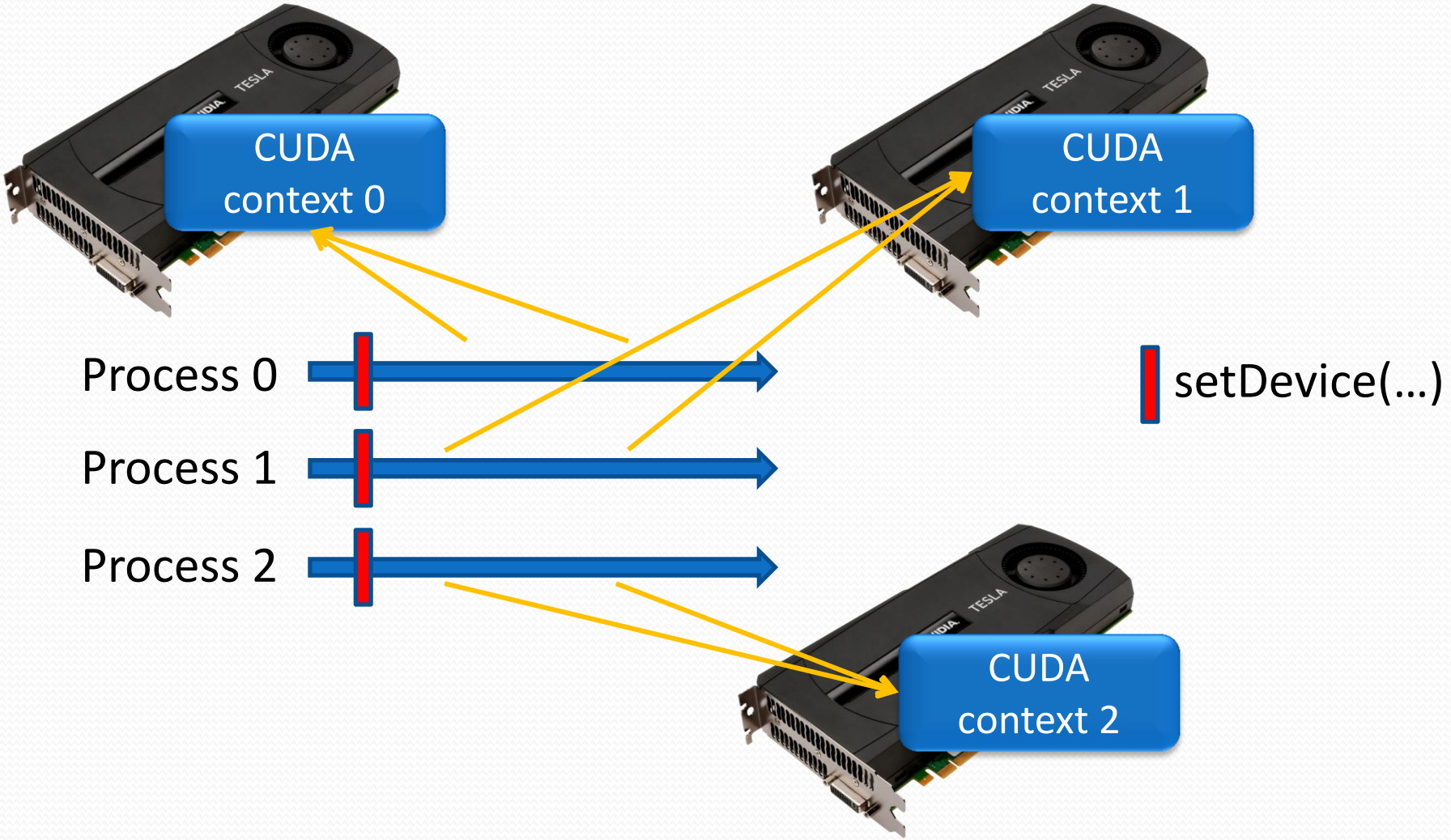
Запускаем параллельную секцию на нужном числе нитей

Результат (1.207 с)



Multi-GPU + MPI

Multi-GPU & multiple CPU processes



Компиляция

- **mpicc** – обертка над хостовым компилятором
  - Задача **mpicc** – подставить пути к инклюдам и слинковать объектные файлы с MPI-библиотеками

Больше ничего mpicc не делает!

- Наша цель:
  - скомпилировать MPI хост код с теми же флагами, с которыми это делает `mpicc`
  - Скомпилировать device-код при помощи `nvcc`
  - Слинковать все с нужными библиотеками MPI/CUDA

Как узнать флаги mpicc?

- При использовании OpenMPI:
  - Вывести флаги компиляции: `$mpicc -showme:compile`

    ```
    -I/usr/lib/openmpi/include -
    I/usr/lib/openmpi/include/openmpi –pthread
    ```

  - Вывести флаги линковки: `$mpicc -showme:link`

    ```
    -pthread -L/usr/lib/openmpi/lib -lmpi -lopen-rte -
    lopen-pal -ldl -Wl,--export-dynamic -lnsl -lutil -lm -
    ldl
    ```

Как узнать флаги mpicc?

- При использовании OpenMPI:
  - Вывести полную строку, вместе с именем используемого компилятора: `$mpicc –showme`

```
gcc -I/usr/lib/openmpi/include -
I/usr/lib/openmpi/include/openmpi -pthread -
L/usr/lib/openmpi/lib -lmpi -lopen-rte -lopen-pal
-ldl -Wl,--export-dynamic -lnsl -lutil -lm -ldl
```

Замена gcc в mpicc на nvcc

- При использовании OpenMPI компилятор можно задать через переменную окружения OMPI_CC, OMPI_F77, OMPI_CXX, OMPI_FC

  - **$OMP_CC=nvcc mpicc --showme**

    ```
    nvcc -I/usr/lib/openmpi/include -
    I/usr/lib/openmpi/include/openmpi -pthread -
    L/usr/lib/openmpi/lib -lmpi -lopen-rte -lopen-pal
    -ldl -Wl,--export-dynamic -lnsl -lutil -lm –ldl
    ```

Замена gcc в mpicc на nvcc

- Проблема:
  - nvcc сам парсит флаги

    Поддерживает только простые `-L, -l, -c,-g` и свои собственные `-v, -arch` и ничего не знает о `-Wl, -pthread`

- Специфические флаги компилятора нужно передавать через `-Xcompiler <флаг>,…`
- Специфические флаги линковщика ld нужно передавать через `-Xlinker <флаг>,…`

Раздельная компиляция

- Подставляем флаги компиляции `mpicxx` в `nvcc`
- Линкуем хостовым компилятором, явно подставляя флаги из `mpicxx` и `nvcc`

```
MPI_COMPILE_FLAGS = $(shell mpicxx --showme:compile)
MPI_LINK_FLAGS = $(shell mpicxx --showme:link)
NVCC_LINK_FLAGS = -L/opt/cuda/lib64 -lcudart
all: main
    nvcc -Xcompiler "\"$(MPI_COMPILE_FLAGS)\"" main.cu –o main.o
    g++ main.o -o main $(MPI_LINK_FLAGS) $(NVCC_LINK_FLAGS)
```

Замена gcc в nvcc на mpicxx

- Опция `-ccbin` позволяет задать используемый компилятор

```
$nvcc -ccbin /usr/bin/mpicxx main.cu -o main
```

- `cudafe` разделит код на host-код и device-код
  - Host-код будет скомпилирован в дальнейшем при помощи `/usr/bin/mpicxx`

- Объектники будут слинкованы через `/usr/bin/mpicxx` => все нужные MPI флаги подставятся

Пример main.cu

```
#include <mpi.h>
#include <iostream>
#include <stdio.h>

__global__ void kernel(int procnum, int device ) {
   printf("Hello from DEVICE %d process %d\n",device,
procnum);
}

int main (int argc, char* argv[])
{
    int rank, size;
    int numDevices = -1;
    cudaGetDeviceCount(&numDevices);
    …
```

Пример main.cu (продолжение)

```
    …
    cudaGetDeviceCount(&numDevices);
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    std::cout<<"Hello from HOST #"<< rank << " see "<<
numDevices << " devices" << std::endl;
    cudaSetDevice(rank % numDevices);
    kernel<<<1,1>>>(rank % numDevices, rank);
    cudaDeviceSynchronize();
    MPI_Finalize();
    return 0;
}
```

Компиляция & запуск

```
$nvcc –arch=sm_20 –ccbin mpicxx main.cu
$mpirun –n 6 ./a.out
```

```
Hello from HOST #3 see 4 devices
Hello from HOST #4 see 4 devices
Hello from HOST #2 see 4 devices
Hello from HOST #5 see 4 devices
Hello from HOST #1 see 4 devices
Hello from HOST #0 see 4 devices
Hello from DEVICE 1 process 1
Hello from DEVICE 3 process 3
Hello from DEVICE 5 process 1
Hello from DEVICE 2 process 2
Hello from DEVICE 4 process 0
Hello from DEVICE 0 process 0
```

peer-to-peer обмены между GPU

UVA & peer-to-peer

- При UVA peer-to-peer обмены между памятью разных GPU делаются неявно при использовании обычных функций `cudaMemcpy*`
  - `dst` и `src` указывают на память на разных устройствах
- Если UVA не поддерживается или нужно явно указать, что это peer-to-peer копирование, используются функции `cudaMemcpyPeer*`

peer-to-peer & non-UVA

- Нужно явно указать номера устройств, между которыми происходит обмен

```
cudaError_t cudaMemcpyPeer (void* dst,
int  dstDevice, const void* src, int  srcDevice,
size_t count )
```

```
cudaError_t cudaMemcpyPeerAsync (void* dst,
int  dstDevice, const void* src, int  srcDevice,
size_t count, cudaStream_t stream=0)
```

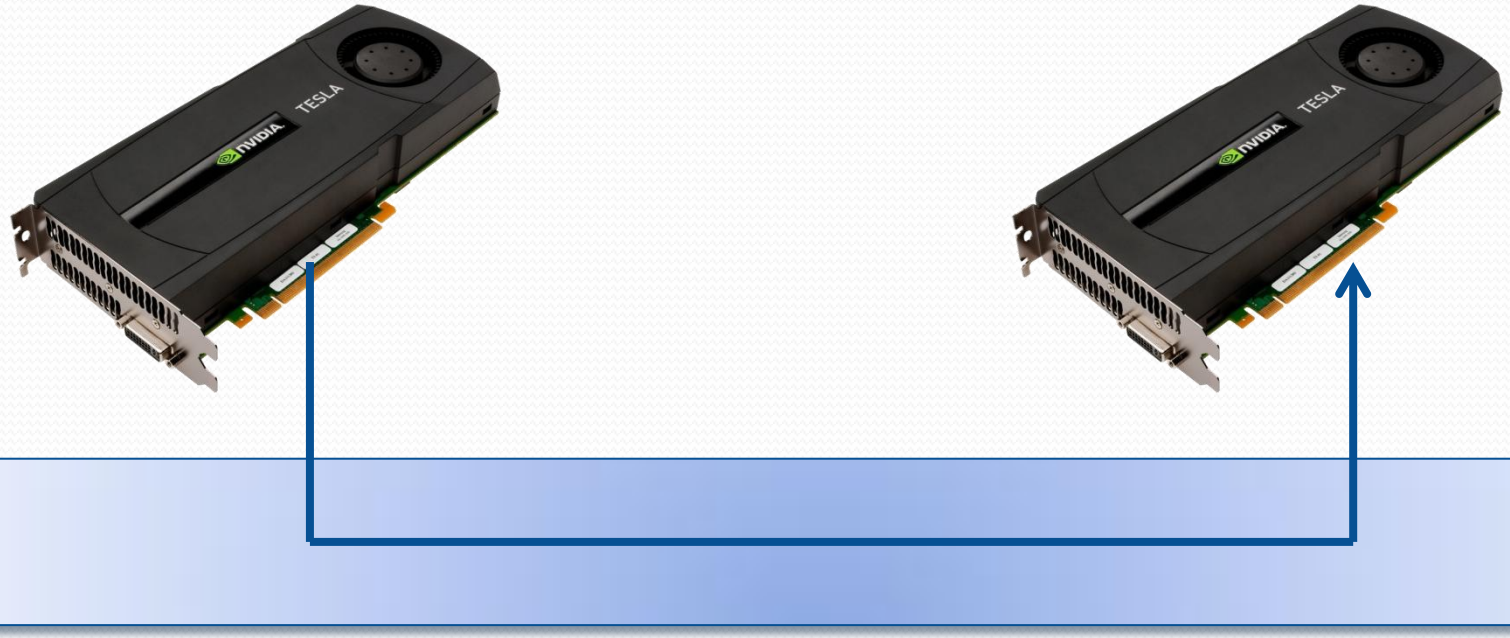
Peer vs PeerAsync

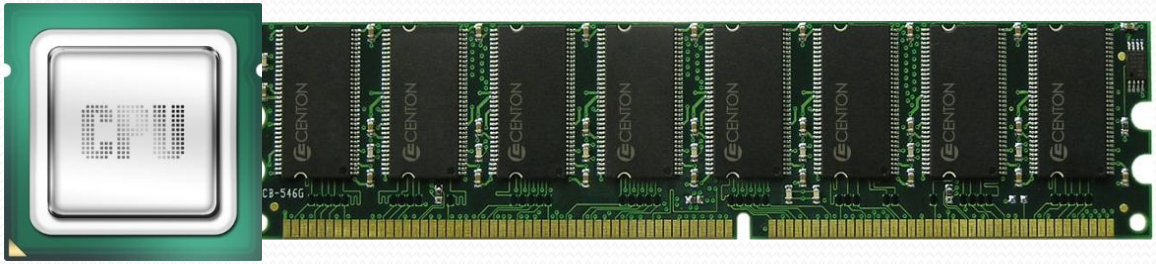
- Обе функции не блокируют хост
- cudaMemcpyPeer начнется только когда завершатся все команды на обоих устройствах (и на активном), отправленные до неё
- Параллельно с cudaMemcpyPeer не могут выполнятся другие команды на обоих устройствах (и на активном)
- cudaMemcpyPeerAsync лишена этих ограничений

P2P пример

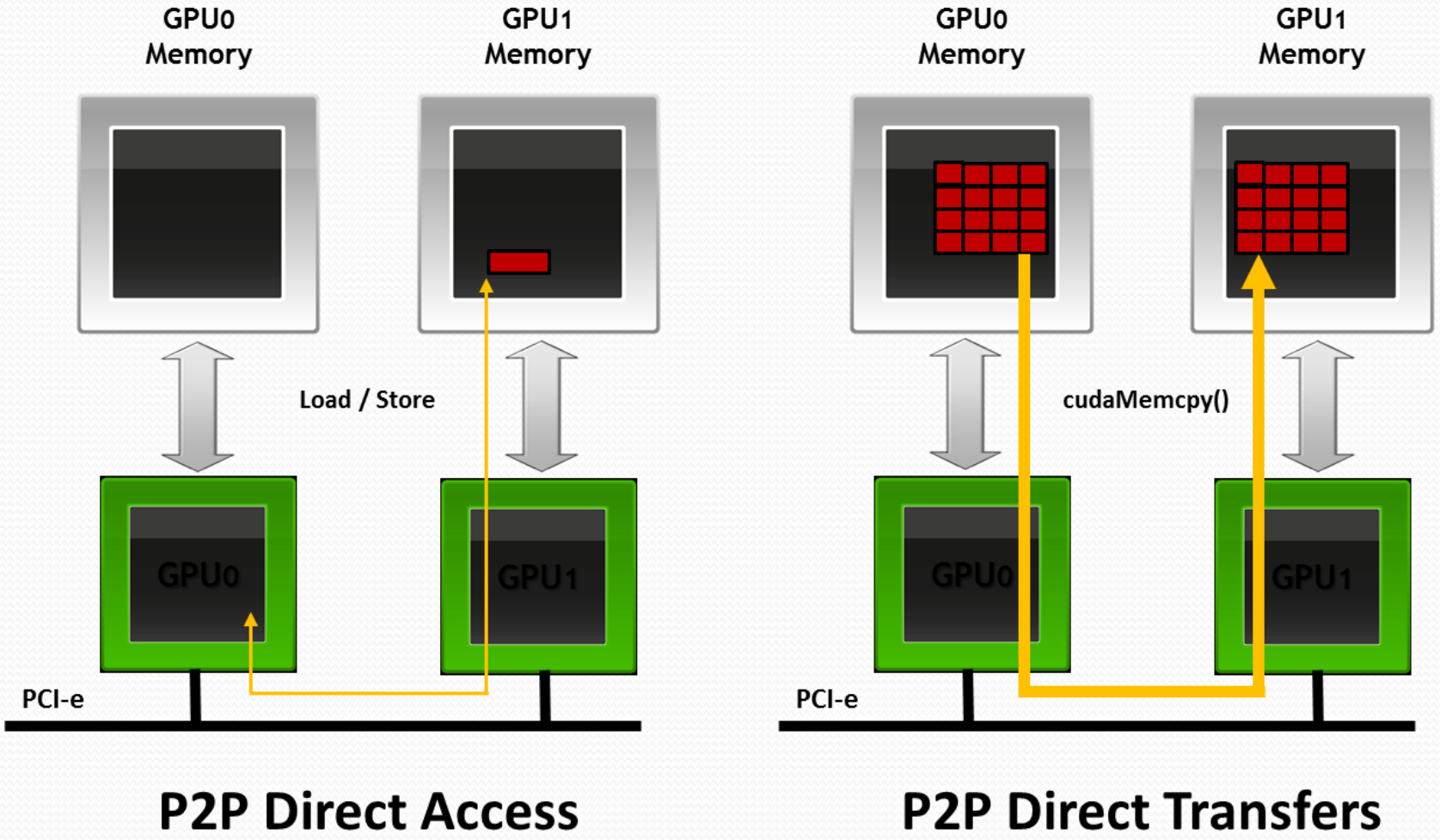
```
cudaSetDevice(0); // Переключились на device 0
float* p0, *p1;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size); // Выделили на device 0
cudaSetDevice(1); // Переключились на device 1
cudaMalloc(&p1, size); // Выделили на device 1
cudaSetDevice(0); // Переключились на device 0
MyKernel<<<1000, 128>>>(p0); // Запуск на device 0
cudaSetDevice(1); // Переключились на device 1
cudaMemcpyPeer(p1, 1, p0, 0, size); // Копировать p0 to p1
MyKernel<<<1000, 128>>>(p1); // Запуск на device 0
```

Прямые peer-to-peer обмены





Прямые peer-to-peer обмены



Прямой P2P доступ

```
cudaError_t cudaDeviceCanAccessPeer (
      int* canAccessPeer, int  device,
      int  peerDevice )
```

- Если в **canAccessPeer** записалось **«1»**, то
  - peer-to-peer копирования межу **peerDevice** и **device** могут выполняться без буферизации на хосте
  - Память выделенная на **peerDevice** может быть доступна напрямую из ядер, работающих на **device**

Прямой P2P доступ

```
cudaError_t cudaDeviceCanAccessPeer (
      int* canAccessPeer, int  device,
      int  peerDevice )
```

- Tesla series
- UVA
- Compute Capability > 2.0

Прямой доступ нужно явно включить!

```
cudaError_t cudaDeviceEnablePeerAccess (
      int  peerDevice, unsigned int  flags )
```

- Теперь память, выделяемая на **peerDevice** доступна напрямую из ядер, запускаемых на активном, а копирования выполняются без участия памяти хоста
- Вызов включает доступ только в одну сторону. Для доступа в обратную сторону нужен отдельный вызов `cudaDeviceEnablePeerAccess`

Прямой P2P доступ

```
cudaSetDevice(0); // Переключились на device 0
float* p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size); // Выделить память на device 0
cudaSetDevice(1); // Переключились на device 1
cudaDeviceEnablePeerAccess(0, 0); // Включить peer-to-peer
                                     доступ к 0

// Запуск на device 1
// p0 указывает на память, выделенную на device 0
MyKernel<<<1000, 128>>>(p0);
```

P2P на tesla-cmc

```
for (int device = 0..deviceCount) {
    for (int peerDevice = 0..deviceCount){
        if (device == peerDevice) {
            printf("- "); continue;
        }
        int canAccessPeer = -1;
        cudaDeviceCanAccessPeer(&canAccessPeer,
                                device, peerDevice);
        printf("%1d ", canAccessPeer);
    }
    printf("\n");
}
```

```
   0 1 2 3
0 - 0 0 0
1 0 - 1 0
2 0 1 - 0
3 0 0 0 -
```

Тест прямого P2P копирония

```
printf("\nCopying %d MB\n", sizeOfMem / (1024 * 1024));
cudaSetDevice(1);
cudaMalloc(&memoryOnDevice1, sizeOfMem);
cudaSetDevice(2);
cudaMalloc(&memoryOnDevice2, sizeOfMem);

...;/* замерить время cudaMemcpyPeer или cudaMemcpy */

printf("\nElapsed time %f \n", elapsedTime);
cudaDeviceEnablePeerAccess(1, 0);
printf("Enable peer access\n");

...;/* замерить время cudaMemcpyPeer или cudaMemcpy */

printf("Elapsed time %f\n", elapsedTime);
```

Тест прямого P2P копирония

```
$./a.out 402410240
Copying 383 MB
Elapsed time 475.087402
Enable peer access
Elapsed time 75.842880
```

Выводы

- Поддержка P2P-копирований упрощает хост-код при необходимости организации пересылок между устройствами
- Поддержка прямых P2P-копирований дает ускорение копирований в несколько раз
- Поддержка прямого P2P-доступа избавляет от необходимости организации пересылок между устройствами