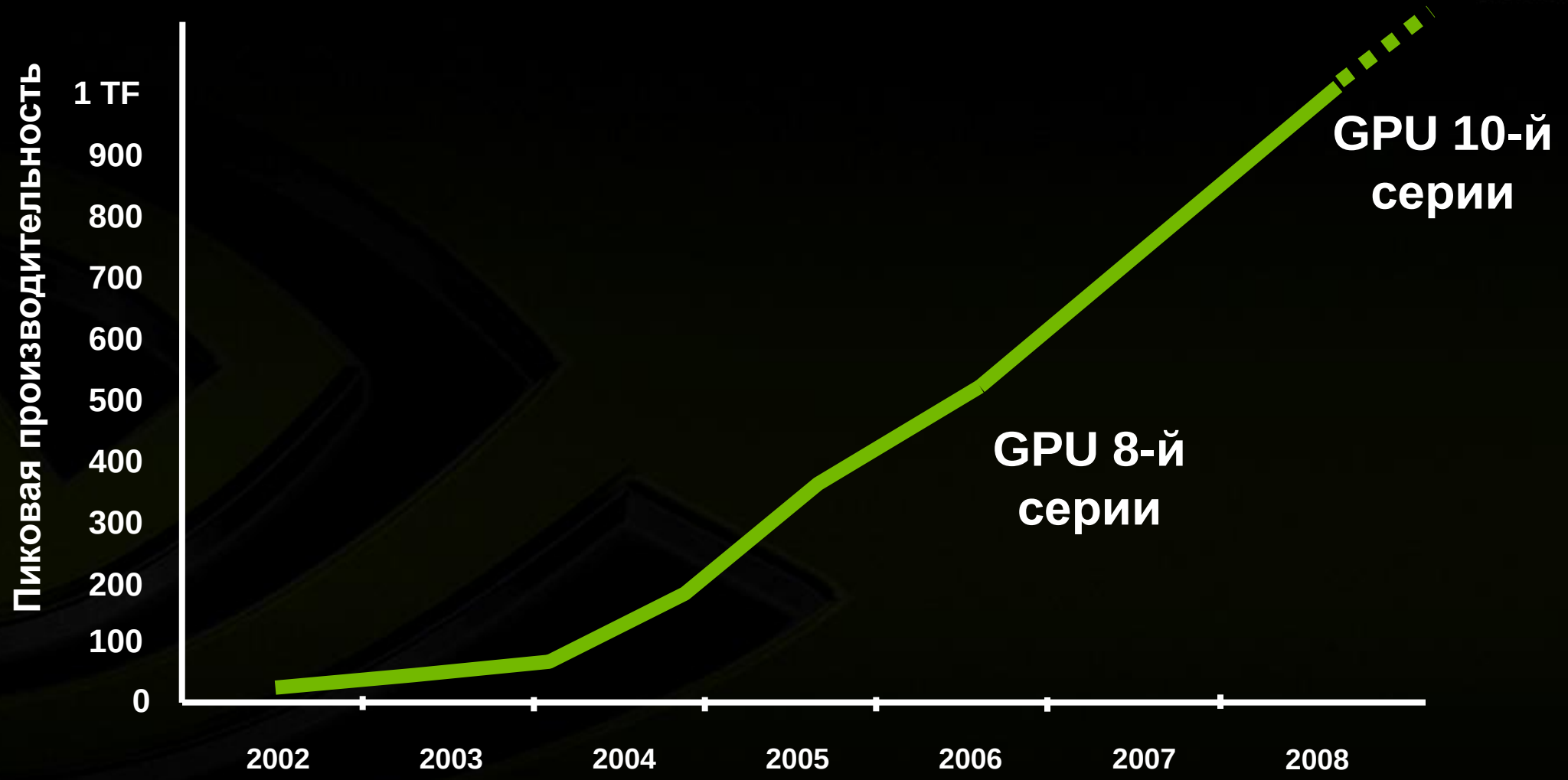

NVIDIA CUDA
для параллельных
вычислений на GPU
Александр Харламов
NVIDIA
NVIDIA

Вычисления на GPU

- Открывают новые возможности
  - Значительно сокращая время до получения результата
  - Цикл обработки данных уменьшается от дней до минут, от недель до дней
- Новый взгляд на вычислительные архитектуры
  - Активизируя исследования в области параллельных алгоритмов, новых программных моделей и вычислительных архитектур

Рост производительности GPU



Исследователи всего мира строят станции на основе GPU

3 GPUs

YONSEI UNIVERSITY

Korea

2 GPUs

University of Cambridge, UK

THEORETICAL and COMPUTATIONAL BIOPHYSICS GROUP

NIH RESOURCE FOR MACROMOLECULAR MODELING AND BIOINFORMATICS

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

3 GPUs

University of Illinois

MIT Graduates Build 16-GPU Monster

POST FROM UBERGIZMO ON 28 JULY 2008 01:08:38 PM. © UBERGIZMO

16 GPUs

The Visual Neuroscience Group

@ The Rowland Institute at Harvard

8 GPUs

FASTRA

GPU SuperPC

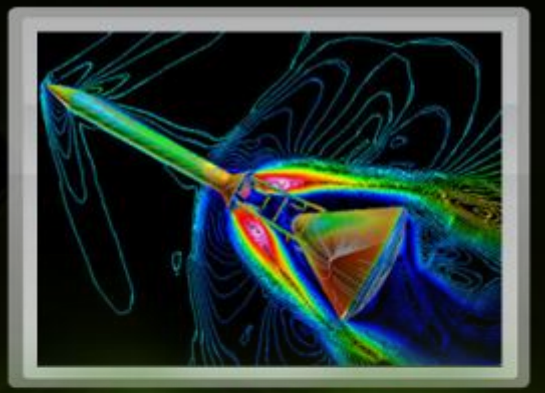
University of Antwerp, Belgium

Параллельные вычисления на GPU

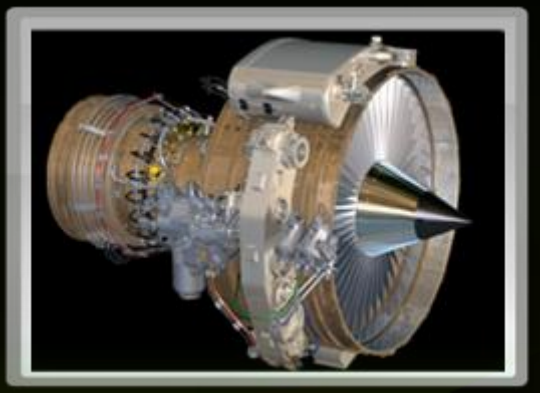
100+ млн. GPU в мире поддерживают CUDA

GeForce®
Развлечения

Tesla™
Высокопроизводительные вычисления



Quadro®
Дизайн, разработка



Выбор CUDA платформы

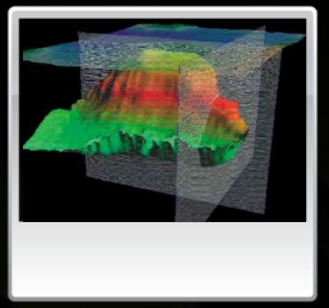
	Tesla	Quadro	GeForce
Стресс-тест с проверкой точности вычислений	X		
Произведено NVIDIA из высококачественных комплектующих	X	X	
3-х летняя гарантия, корпоративная поддержка	X	X	
4 Гб оперативной памяти для работы с большими объемами данных	X	X	
Единое профессиональное решение для вычислений и графики		X	
Пользовательские приложения: PhysX, Video, Imaging			X
Короткий жизненный цикл пользовательского продукта			X
Производится и сопровождается партнерами NVIDIA			X
Поддержка осуществляется через партнеров NVIDIA			X

Более 250 заказчиков / разработчиков ПО

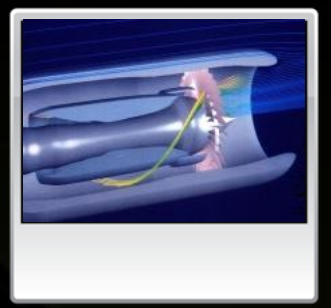
Life Sciences & Medical Equipment		Productivity / Misc	Oil and Gas	EDA	Manufacturing	Finance	CAE / Numerics	Communication
Max Planck	GE Healthcare	CEA	Hess	Synopsys	Renault	Symcor	The Mathworks	Nokia
FDA	Siemens	WRF Weather Modeling	TOTAL	Nascentric	Boeing	Level 3	Wolfram	RIM
Robarts Research	Techniscan	OptiTex	CGG/Veritas	Gauda		SciComp	National Instruments	Philips
Medtronic	Boston Scientific	Tech-X	Chevron	CST		Hanweck	Access Analytics	Samsung
AGC	Eli Lilly	Elemental Technologies	Headwave	Agilent		Quant Catalyst	Tech-x	LG
Evolved machines	Silicon Informatics	Dimensional Imaging	Acceleware			RogueWave	RIKEN	Sony Ericsson
Smith-Waterman DNA sequencing	Stockholm Research	Manifold	Seismic City			BNP Paribas	SOFA	NTT DoCoMo
AutoDock	Harvard	Digisens	P-Wave Seismic Imaging					Mitsubishi
NAMD/VMD	Delaware	General Mills	Mercury Computer					Hitachi
Folding@Home	Pittsburg	Rapidmind	ffA					Radio Research Laboratory
Howard Huges Medical	ETH Zurich	MS Visual Studio						US Air Force
CRIBI Genomics	Institute Atomic Physics	Rhythm & Hues						
		xNormal						
		Elcomsoft						
		LINZIK						

CUDA

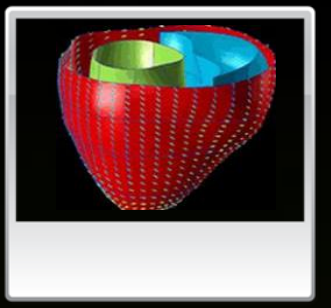
Будущие прорывы в индустрии опираются на компьютерное моделирование



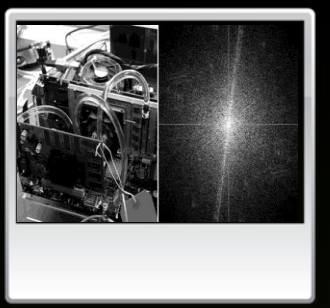
Вычислительная геология



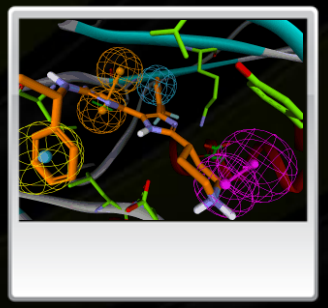
Вычислительное моделирование



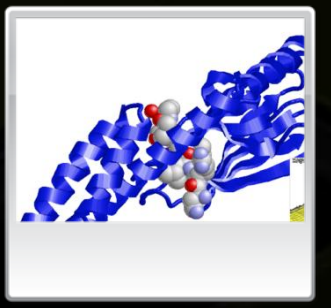
Вычислительная медицина



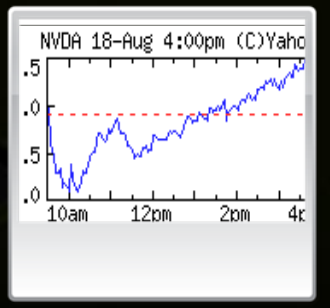
Вычислительная физика



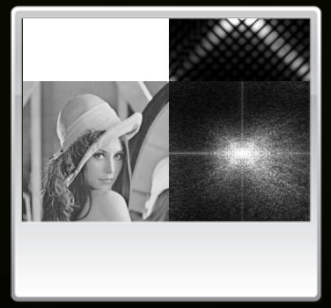
Вычислительная химия



Вычислительная биология



Вычислительная экономика

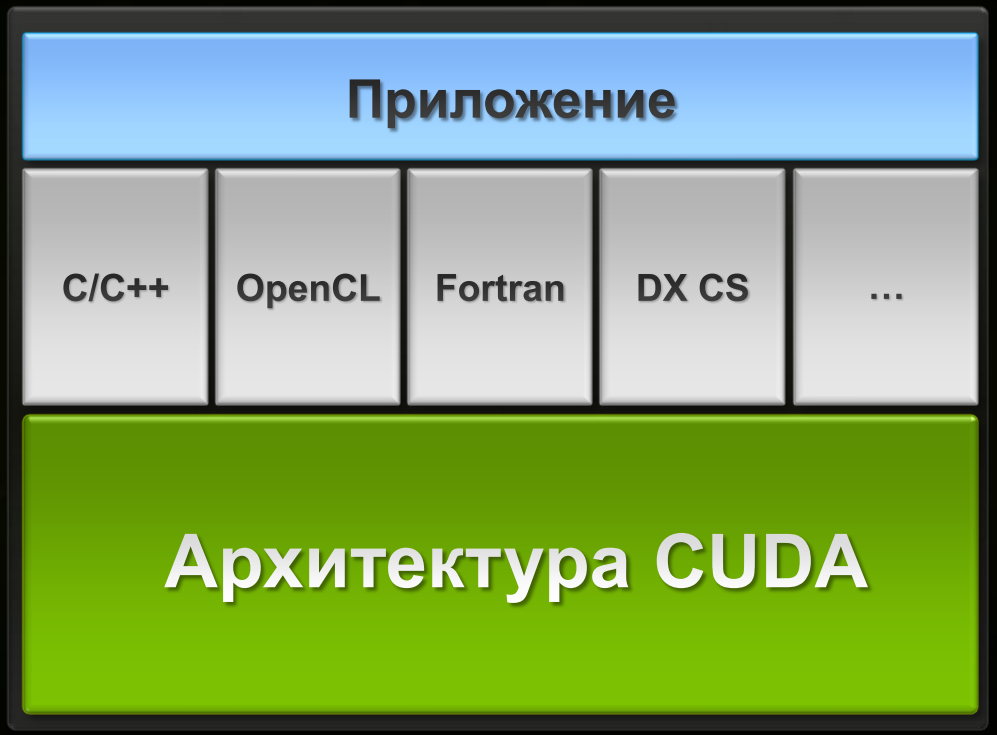


Обработка сигналов

Масштабируемая программно-аппаратная архитектура для параллельных вычислений

CUDA для параллельных вычислений

- Программно-аппаратный стек для процессоров NVIDIA
- Раскрывает потенциал GPU для вычислений общего назначения
- Спроектирован для поддержки любого вычислительного интерфейса
  - OpenCL, C/C++, и т.д.



Программная модель

Программная модель CUDA

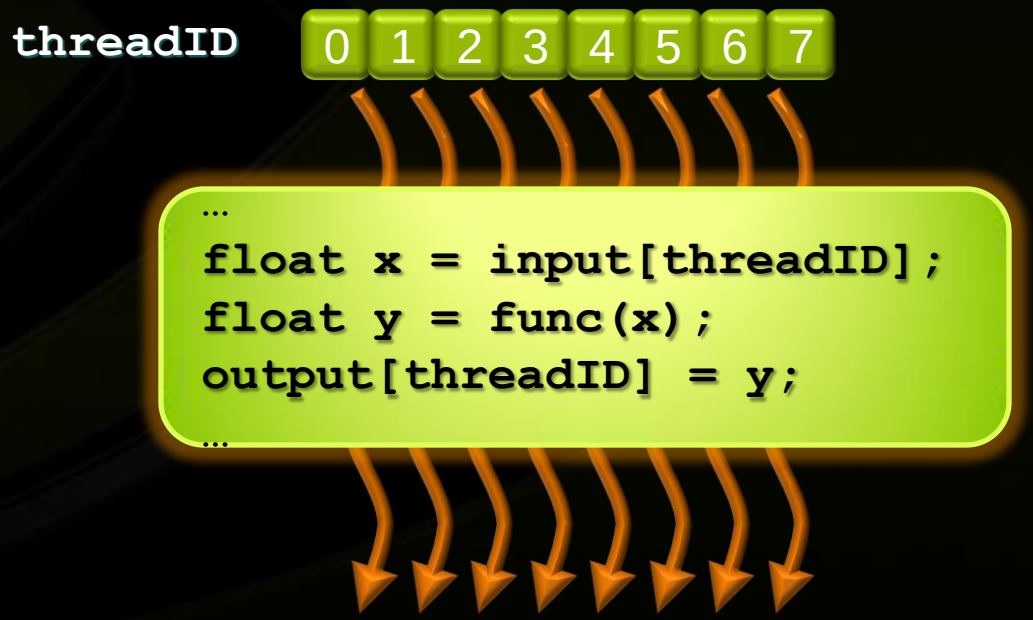
- Масштабируемость на десятки ядер, сотни параллельных потоков
- Позволяет сфокусироваться на разработке параллельных алгоритмов
  - А не внутренней механике языка программирования
- Поддерживает гетерогенные вычисления
  - CPU для последовательного кода, GPU – для параллельного

Ключевые абстракции CUDA

- **Потоки** CUDA выражают мелкозернистый параллелизм данных
  - Упрощают параллельную декомпозицию задач
  - Виртуализируют физические процессоры
- **Блоки потоков** CUDA выражают крупнозернистый параллелизм
  - Простая модель исполнения
  - Обеспечивают прозрачную масштабируемость
- Легковесные **примитивы синхронизации**
  - Простая модель синхронизации
- **Разделяемая память** для обмена данными между потоками
  - Кооперативное исполнение потоков

Потоки CUDA

- Ядро CUDA исполняется массивом потоков
  - Все потоки исполняют одну программу
  - Каждый поток использует свой индекс для вычислений и управления исполнением

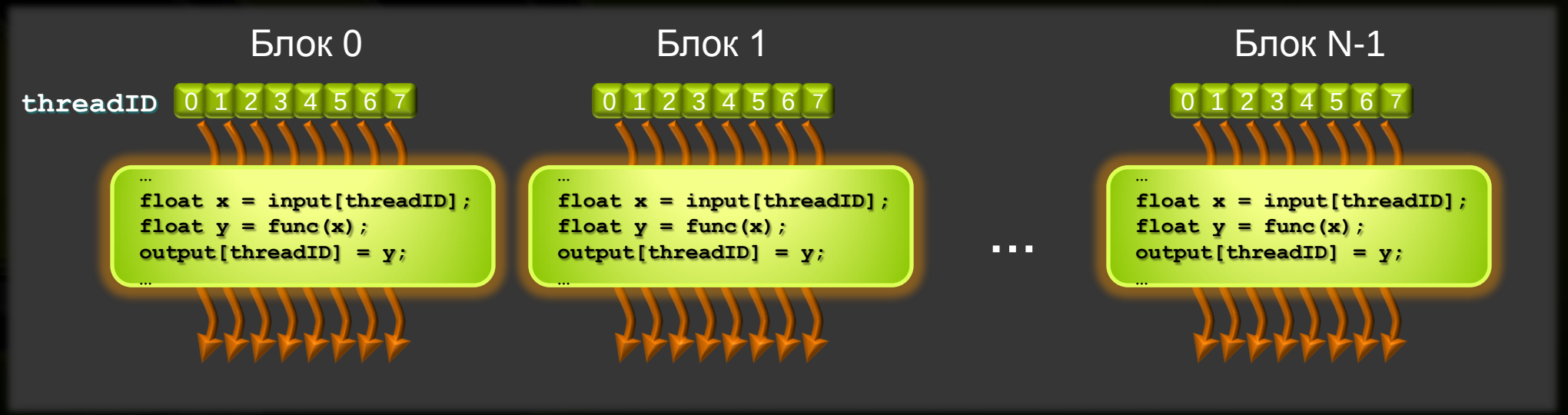


Взаимодействие потоков

- Потоки могут быть не полностью независимы
  - Обмениваются результатами вычислений
  - Разделяют доступ к внешней памяти
- Возможность взаимодействия потоков – ключевая особенность программной модели CUDA
  - Потоки кооперируют, используя разделяемую память и примитивы синхронизации

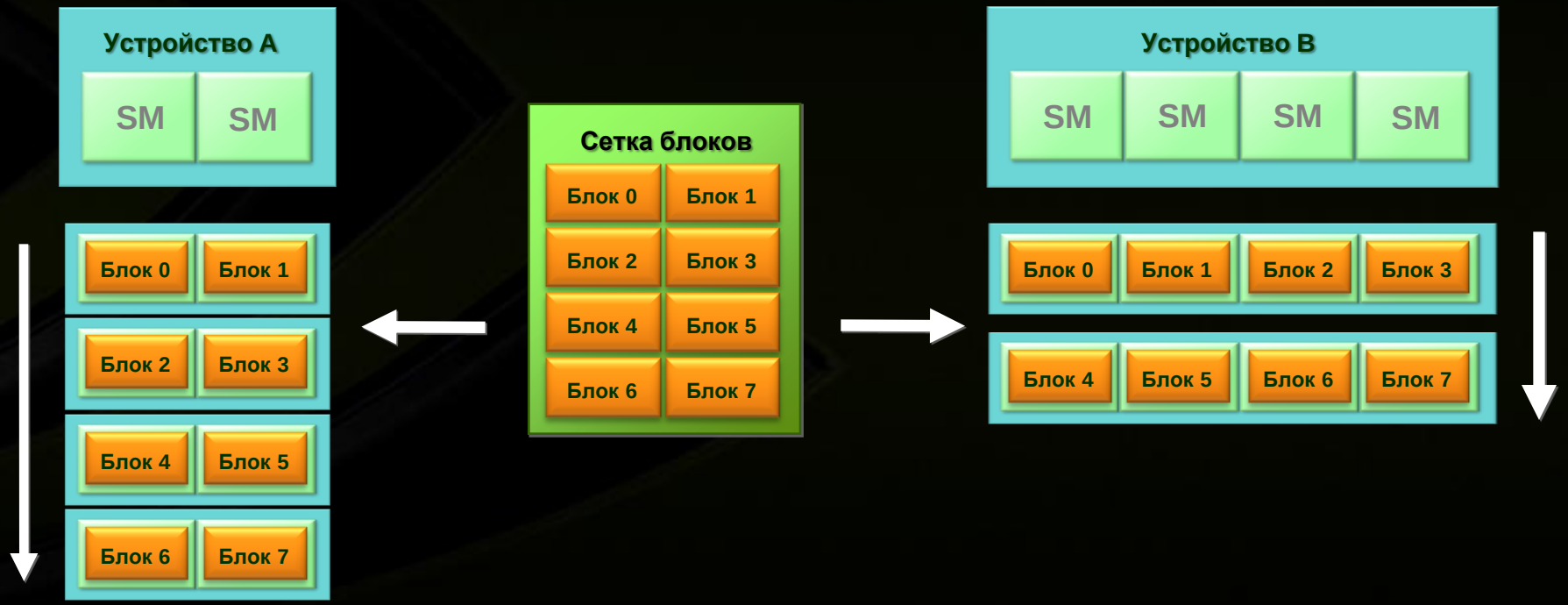
Блоки потоков: масштабируемость

- Монолитный массив потоков разделяется на блоки
  - Потоки внутри блока взаимодействуют через разделяемую память
  - Потоки в разных блоках не могут синхронизироваться
- Позволяет программам прозрачно масштабироваться



Прозрачная масштабируемость

- Блоки могут быть распределены на любой процессор
  - Код на CUDA масштабируется на любое количество ядер



Гетерогенная модель вычислений

- Последовательная программа, с параллельными секциями
  - Последовательный код исполняется на CPU
  - Параллельный код исполняется блоками потоков на множестве вычислительных модулей GPU

Расширения C для CUDA

Стандартный C код

```
void saxpy_serial(int i, float a,
                  float *x, float *y)
{
    y[i] = a*x[i] + y[i];
}


// Вызов последовательного
// ядра SAXPY

for (int i = 0; i < n; ++i)
  saxpy_serial(i, 2.0, x, y);
```

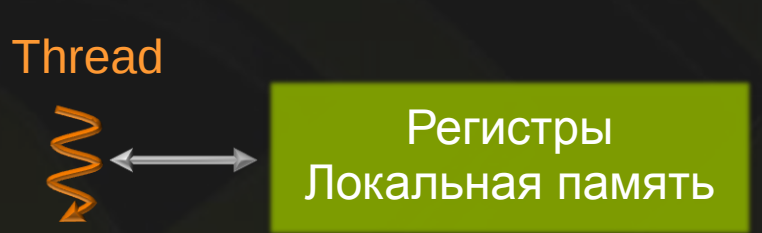
Код на CUDA C

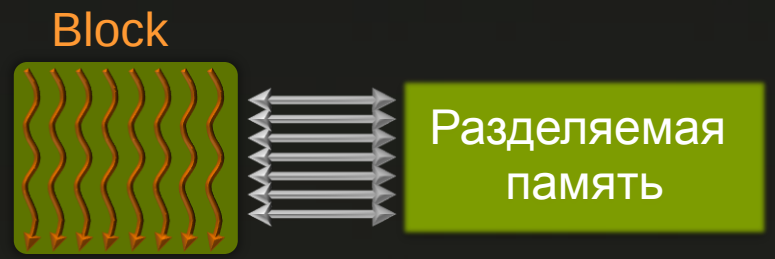
```
__global__
void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}

// Вызов параллельного ядра SAXPY с размером
// блока в 256 потоков
int nblocks = (n + 255) / 256;

saxpy_parallel<<<nblocks, 256>>>(n, 2.0f, x, y);
```

Уровни параллелизма в CUDA

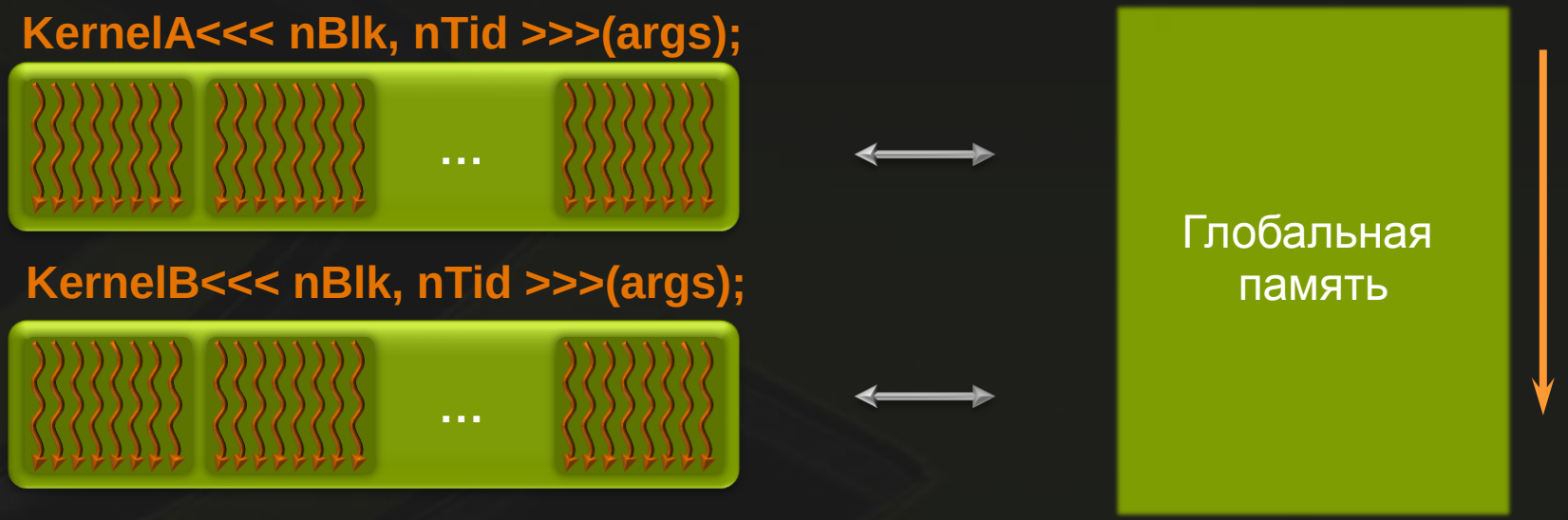
- **Параллелизм на уровне потоков**
  - Каждый поток – независимая нить исполнения
- **Параллелизм на уровне данных**
  - По потокам внутри блока
  - По блокам внутри ядра
- **Параллелизм на уровне задач**
  - Блоки исполняются независимо друг от друга

Пространство памяти CUDA





Пространство памяти CUDA

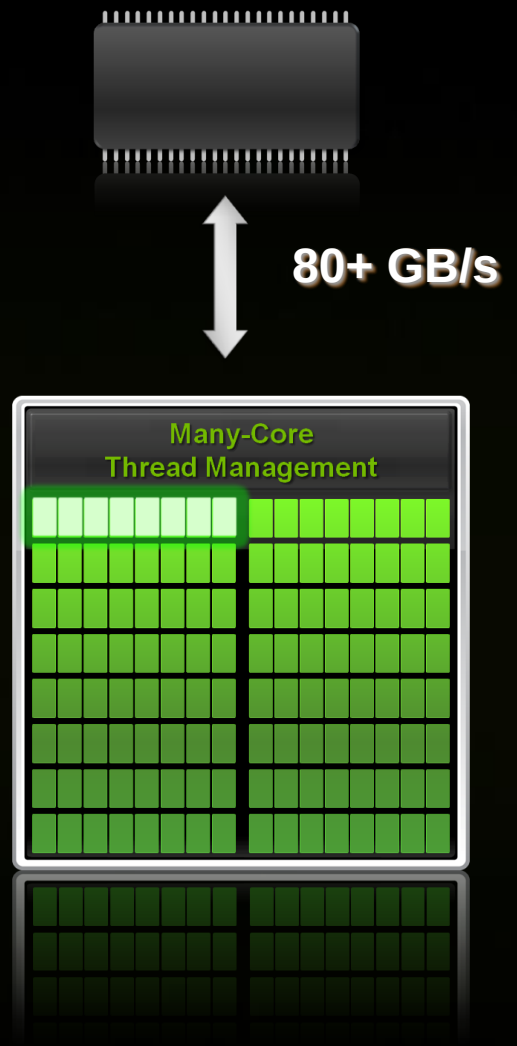


Пространство памяти CUDA


NVIDIA
CUDA
Архитектура

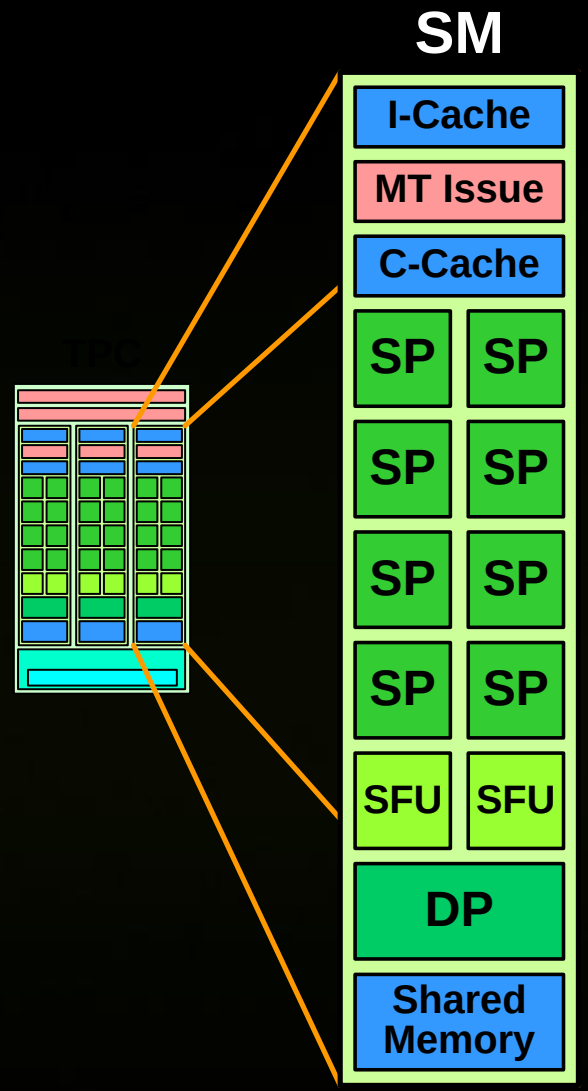
GPU как параллельный процессор

- GPU обладает массивно-параллельной вычислительной архитектурой
  - Выступает в роли сопроцессора для CPU
  - Имеет доступ к выделенной памяти с широкой полосой пропускания
  - Исполняет множество потоков одновременно
- Параллельные секции приложения исполняются как вычислительные ядра
  - Каждое ядро исполняет множество потоков
- Потоки GPU легковесны
  - Очень низкие накладные расходны на создание
  - Мгновенное переключение контекстов
  - GPU использует тысячи потоков для эффективности



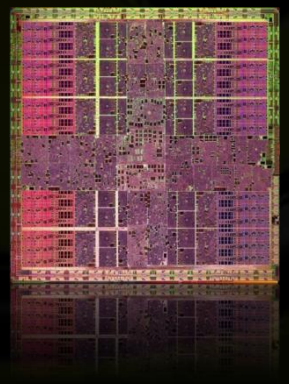
Мультипроцессор NVIDIA Tesla 10

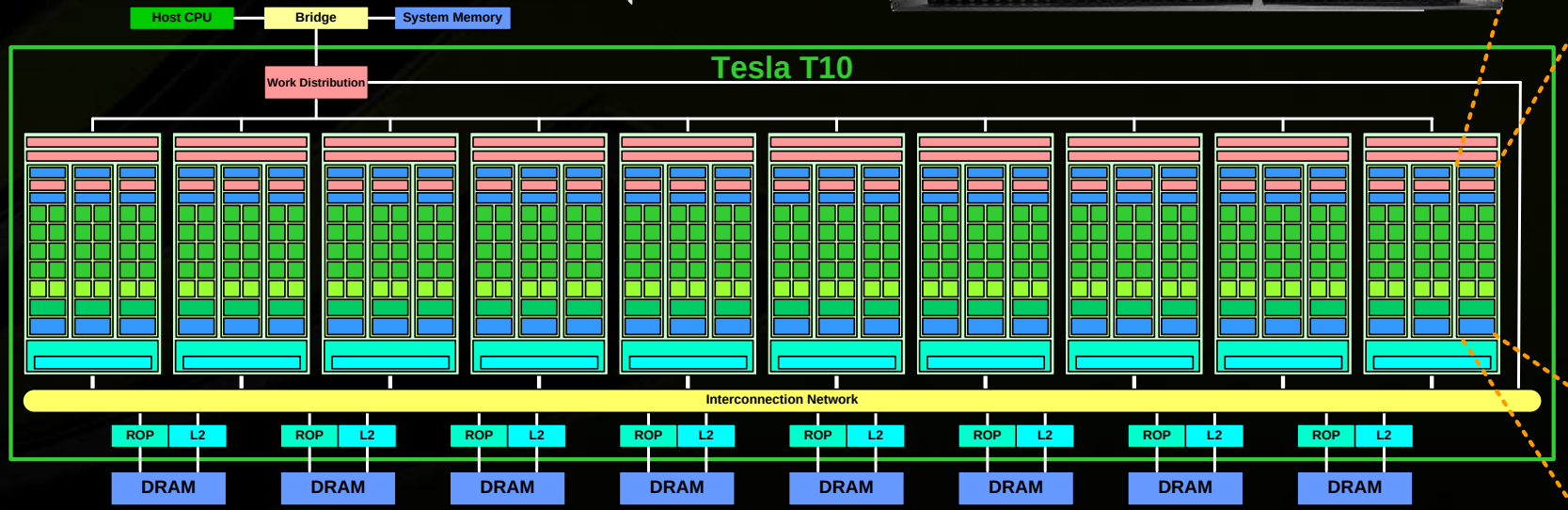
- RISC-подобная скалярная система команд
- Многопоточное исполнение инструкций
  - Поддержка до 1024 потоков
  - Аппаратный менеджер потоков
- 8 скалярных АЛУ
  - Операции с одинарной точностью IEEE-754
  - Целочисленные операции
  - 16K 32-х битных регистров
- 2 АЛУ для специальных операций
  - sin/cos/rcp/rsq
- 1 АЛУ для операций с двойной точностью
  - Поддерживает FMA
- 16KB разделяемой памяти

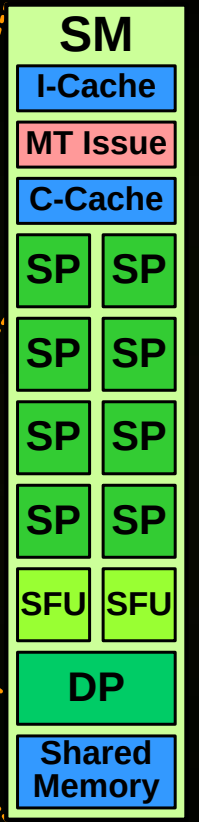


Архитектура NVIDIA Tesla 10

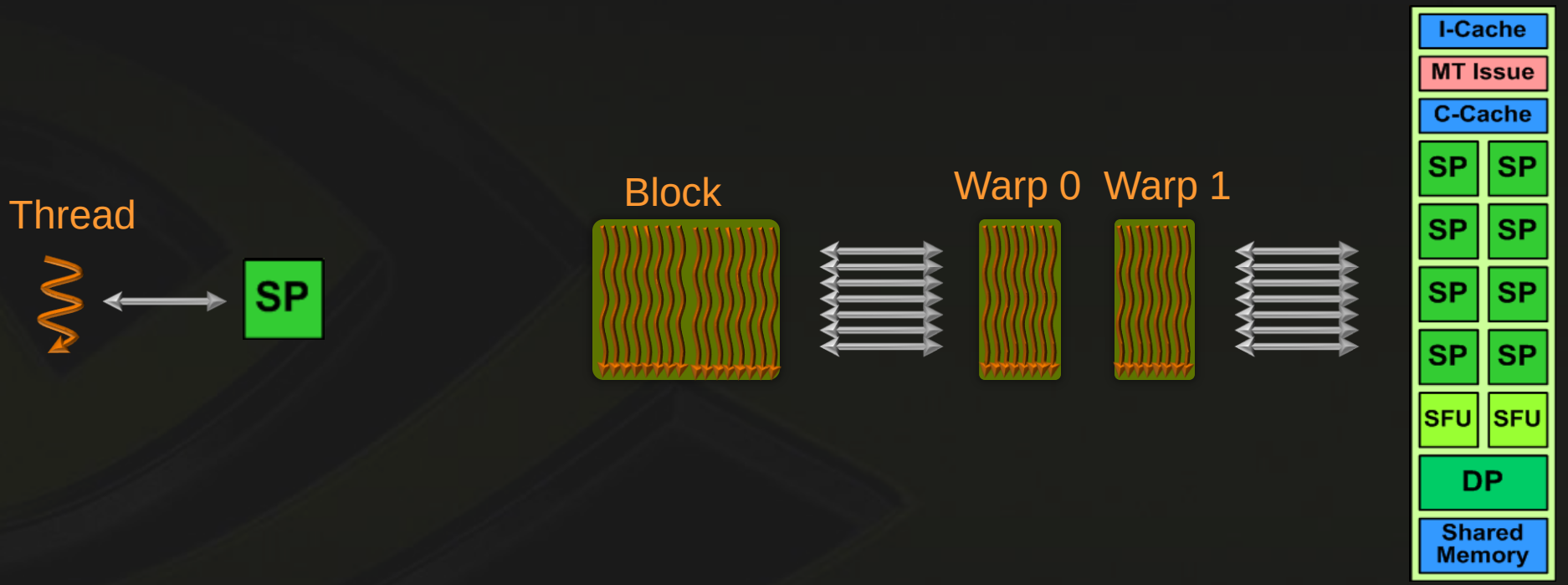
- 240 процессоров работают на частоте 1.5 GHz: 1 TFLOPS в пике
- 128 потоков на процессор: 30,720 потоков на чипе



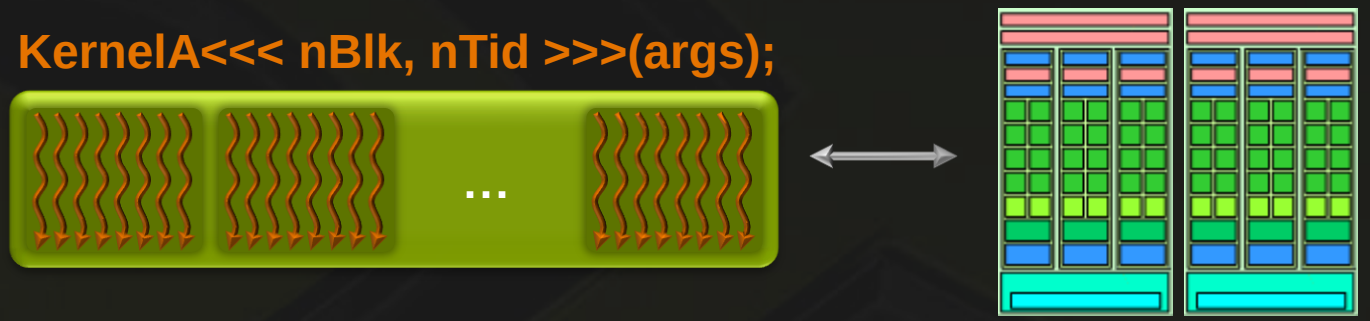




Программная модель & архитектура

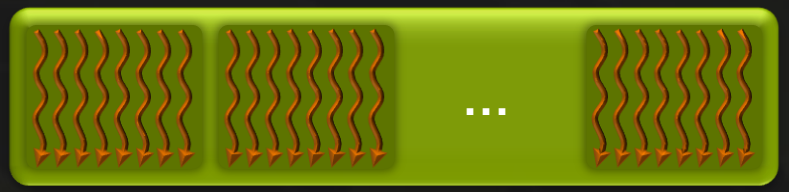


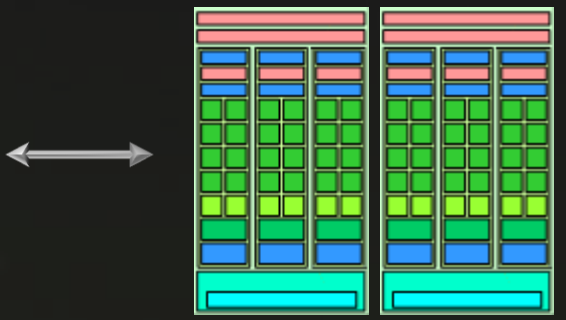
Программная модель & архитектура



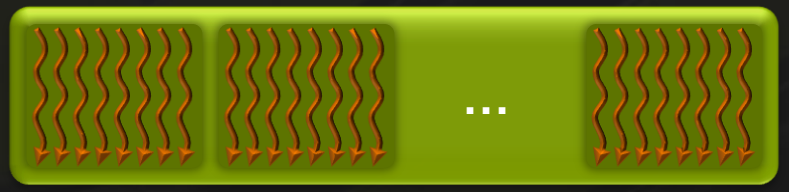
Программная модель & архитектура

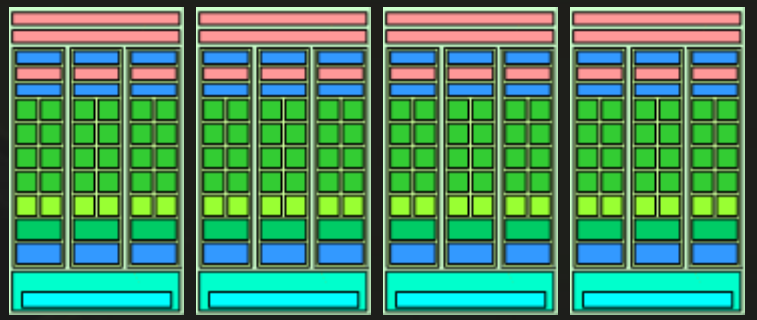
KernelA<<< nBlk, nTid >>>(args);





KernelB<<< nBlk, nTid >>>(args);





Fermi

- 8x производительность double вычислений
- Error Correcting Codes
- Однородное адресное пространство
  - Указатели на функции
- Исполнение разных ядер
- Иерархия кэшей
- Warp-коалесинг

Средства разработки

Средства разработки

- CUDA C
  - Runtime API
  - Driver API
- OpenCL

CUDA C (Runtime API)

- Расширение языка C
- CUDA API:
  - Расширения языка C
    - Затрагивает те части кода, которые исполняются на GPU
  - Runtime библиотека состоит из:
    - Общие компоненты (типы и функции)
    - Управление GPU и взаимодействие с графическими API
    - Функции доступные на GPU
      - __syncthreads
      - «быстрые» функции

Расширение языка: Спецификаторы функций

	Исполняется	Вызывается
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` определяет ядро
  - Всегда возвращает `void`
- `__device__` и `__host__` можно использовать вместе
- `__device__` недоступен указатель на функции
- Для `__device__` функций
  - Нет рекурсии
  - Нет статических переменных
  - Нет переменного числа аргументов

Расширение языка: Спецификаторы переменных

	Память	Видимость	Время жизни
`__shared__ int SharedVar;`	shared	thread block	thread block
`__device__ int GlobalVar;`	global	grid	Приложение
`__constant__ int ConstantVar;`	constant	grid	Приложение

- Переменные без спецификатора попадают в регистры
  - Кроме больших структур, которые попадают в локальную память
- Указатели могут указывать на память в разделяемой или глобальной памяти
  - Глобальная память:
    - Память выделенная на CPU и переданная в ядро
  - Разделяемая память:
    - Статически выделенная внутри ядра
    - Статически выделенная при вызове

CUDA C Runtime

- NVCC (cudart.lib)
- Спецификаторы функций
- Встроенные переменные
- C / C++ интерфейс
- Библиотека функций
  - __device__

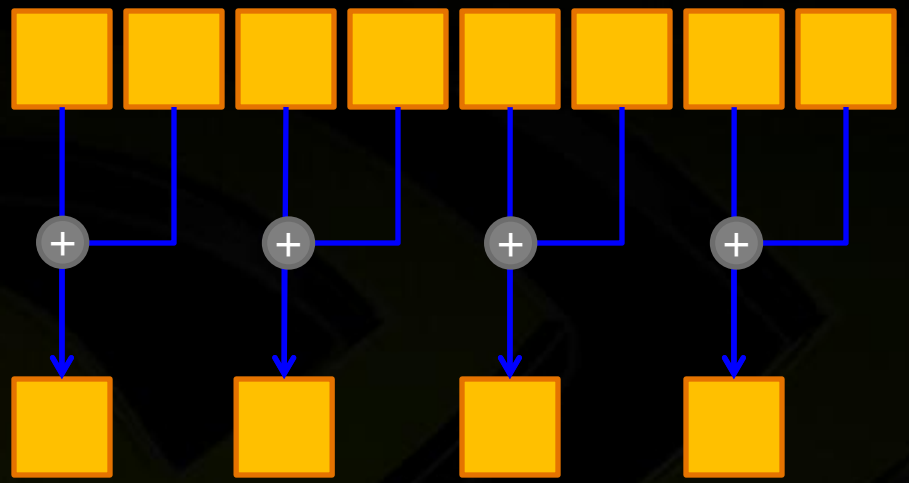
```
#define  N          (256*256)     // array size
#define  PI         3.1415926f

__global__ void kernel ( float * data )
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x ;

    data [idx] = idx;
}
```

CUDA C Runtime

- NVCC (cudart.lib)
- Спецификаторы функций
- Встроенные переменные
- C / C++ интерфейс
- Библиотека функций
  - __host__

```
int main ( int argc, char *  argv [] )
{
    float * a   = new float [N]; // CPU память
    float * dev = NULL;          // GPU память

        // выделение памяти
    cudaMalloc ( (void**)&dev, N * sizeof ( float ) );

    dim3 threads = dim3( 512, 1 );
    dim3 blocks  = dim3( N / threads.x, 1 );

        // запуск ядра
    kernel<<<blocks, threads>>> ( dev );
    cudaThreadSynchronize();

    cudaMemcpy ( a, dev, N * sizeof ( float ),
                 cudaMemcpyDeviceToHost
    cudaFree   ( dev   );

    delete [] a;

    return 0;
}
```

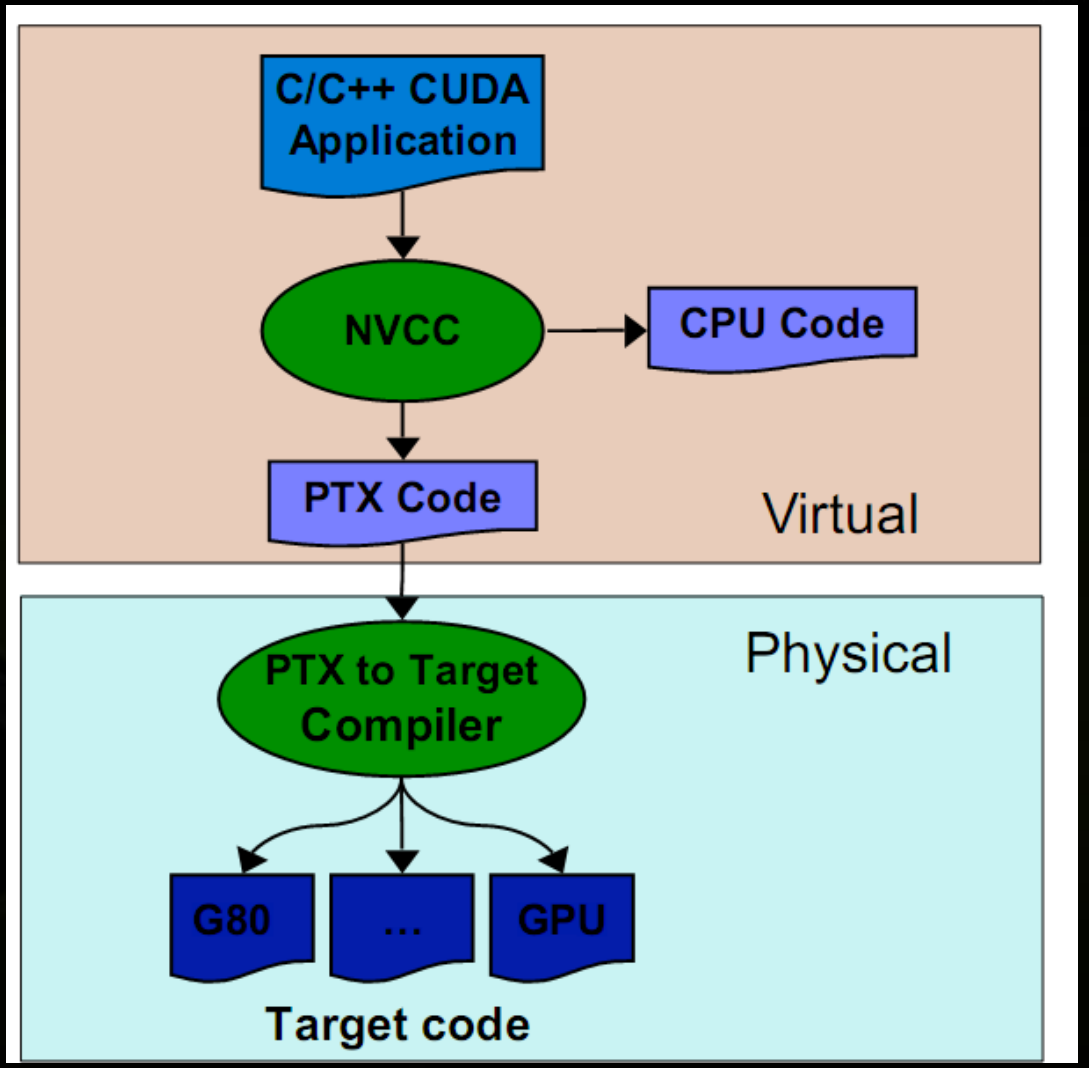
CUDA C Runtime



```
#define  N          (256*256)     // array size
#define  PI         3.1415926f

__global__ void linear( float * pDst, float * pSrc,
                        int wDst, int wSrc )
{
   int idx = blockIdx.x * blockDim.x + threadIdx.x ;

   float factor = (float) wDst / (float) wSrc;

   float center = idx / factor;
   int start = (int) center;
   int stop = start + 1;
   float t = center - start;

   float a = pSrc[start];
   float b = pSrc[stop];
   float r = a * (1 - t) + b * t;


   pDst[idx] = r;
}
```

CUDA C Runtime

- NVCC
- -keep
  - .ptx
  - .cubin



CUDA C Runtime

- NVCC
- -keep
  - .ptx
  - .cubin

```
__global__ void kernel ( float * data )
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x ;

    data [idx] = idx;
}
```

CUDA C Runtime

- NVCC
- -keep
  - .ptx
  - .cubin

```
.entry _Z6kernelPf (
                .param .u32 __cudaparm__Z6kernelPf_data)
{
  .reg .u16 %rh<4>;
  .reg .u32 %r<8>;
  .reg .f32 %f<3>;
  .loc   14       6       0
  $LBB1__Z6kernelPf:
  .loc   14       10      0
  mov.u16         %rh1, %ctaid.x;  //
  mov.u16         %rh2, %ntid.x;   //
  mul.wide.u16    %r1, %rh1, %rh2; //
  cvt.u32.u16     %r2, %tid.x;     //
  add.u32         %r3, %r2, %r1;   //
  cvt.rn.f32.s32  %f1, %r3;        //
  ld.param.u32    %r4, [__cudaparm__Z6kernelPf_data]; //
  mul.lo.u32      %r5, %r3, 4;     //
  add.u32         %r6, %r4, %r5;   //
  st.global.f32   [%r6+0], %f1;    // id:15
  .loc   14       11      0
  exit;
  $LDWend__Z6kernelPf:
} // _Z6kernelPf
```

CUDA C Driver

- Низкоуровневый API
  - Инициализация cuInit()
  - Поиск устройства CUdevice
  - Создание контекста CUcontext
  - Загрузка модуля CUmodule
  - Загрузка функции CUfunction
  - Выделение памяти CUdeviceptr
  - // ...

```
CUdevice    device;
CUcontext   context;
CUmodule    module;
CUfunction  function;
CUdeviceptr pData;

float * pHostData = new float[N];

cuInit(0);

cuDeviceGetCount(&device_count);
cuDeviceGet( &device, 0 );

cuCtxCreate( &context, 0, device );

cuModuleLoad( &module, "hello.cuda_runtime.ptx" );

cuModuleGetFunction( &function, module, "_Z6kernelPf" );

cuMemAlloc( &pData, N * sizeof(float) );

// ...
```

CUDA C Driver

- Низкоуровневый API
  - // …
  - Установка размеров блока
  - Установка аргументов функции ядра
  - Установка параметров ядра
  - Запуск ядра cuLaunch()
  - Копирование памяти
  - Освобождение ресурсов

```
// ...
cuFuncSetBlockShape( function, N, 1, 1 );
cuParamSeti( function, 0, pData );
cuParamSetSize( function, sizeof(void *) );
cuLaunchGrid( function, 1, 1 );
cuMemcpyDtoH( pHostData, pData, N * sizeof( float) );
cuMemFree( pData );
```

OpenCL

- Кроссплатформенный стандарт
  - GPU, CPU, Cell, …
- Проблема: функциональность, но не производительность
  - Разный код для разных платформ
  - Разные расширения openGL-style

CUDA vs OpenCL Терминология

- CUDA C
  - Поток (thread)
  - Блок потоков (thread block)
  - Сеть (grid)
  - Ядро
- OpenCL
  - Элемент работы (work-item)
  - Группа работы (work-group)
  - N-мерное пространство индексов (ND-Range index space)
  - Ядро

CUDA vs OpenCL
Спецификаторы функций

- CUDA C
  - __global__
  - __host__
  - __device__
- OpenCL
  - __kernel
  - `n/a`
  - `n/a`

CUDA vs OpenCL
Пространство памяти

- CUDA C
  - __device__
  - __shared__
  - __constant__
  - local
- OpenCL
  - __global
  - __local
  - __constant
  - __private

OpenCL

- Низкоуровневый API
  - Создание контекста cl_context
  - Поиск устройства cl_device_id
  - Создание очереди комманд
  - Создание ид. программы и компиляция
  - Создание ид. ядра
  - // ...

```
cl_context ctx;
cl_command_queue cmd_q;
cl_program program;
cl_kernel kernel;
cl_mem    mem;
cl_device_id * pDevId = NULL;
```

```
ctx = clCreateContextFromType(0,CL_DEVICE_TYPE_GPU,0,0,0)
```

```
clGetContextInfo(ctx,CL_CONTEXT_DEVICES,0,0,&dev_cnt);

clGetContextInfo(ctx,CL_CONTEXT_DEVICES,dev_cnt,pDevId,0)
```

```
cmd_q= clCreateCommandQueue(ctx,pDevId[0],0,0);
```

```
program = clCreateProgramWithSource(ctx,1,pText,0,0);

clBuildProgram(program, 0,0,0,0,0);
```

```
kernel = clCreateKernel(program, "simple", 0);

// ...
```

OpenCL

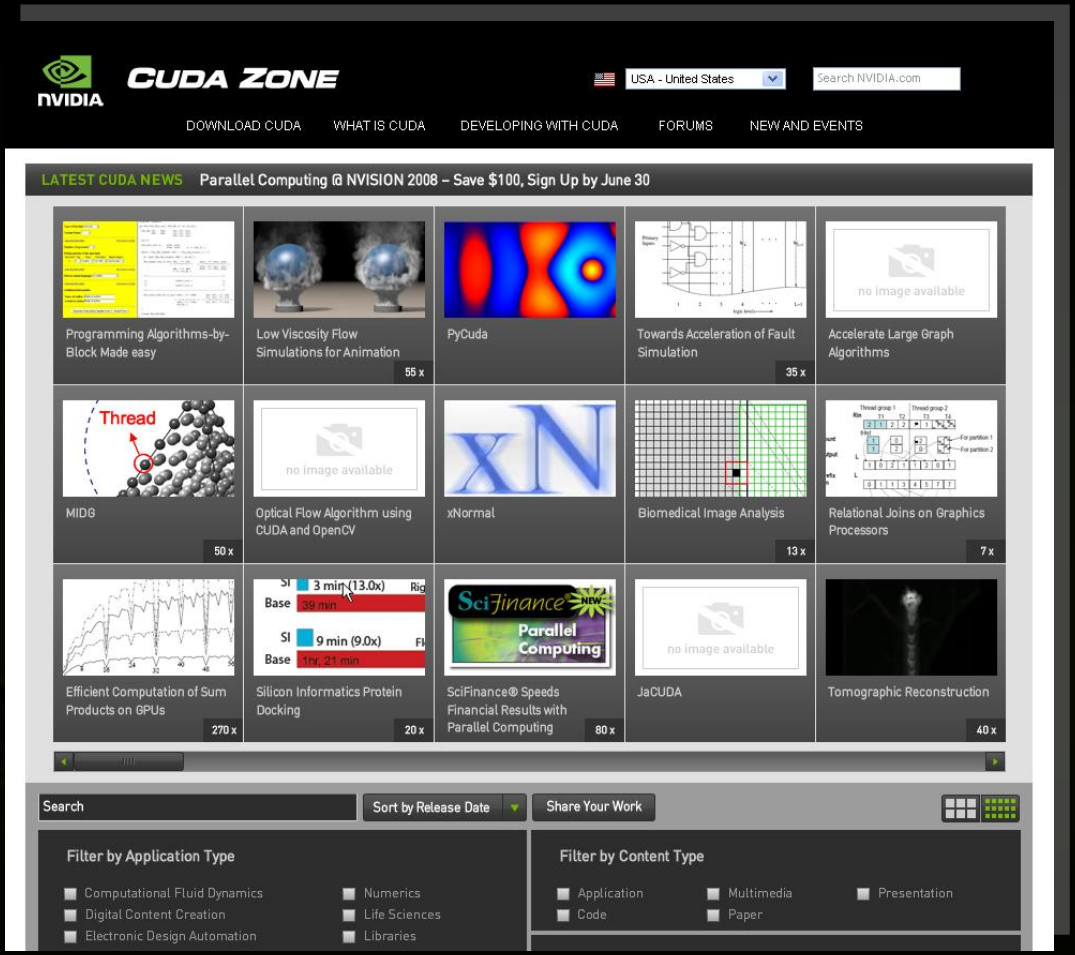
- Низкоуровневый API
  - // ...
  - Выделение памяти
  - Установка аргументов функции ядра
  - Добавление в очередь ядра с заданным рамером рабочей группы и NDRange
  - Добавление в очередь копирование памяти
  - Освобождение ресурсов

```
// ...

mem = clCreateBuffer(ctx,CL_MEM_WRITE_ONLY,
                     N*sizeof(float),0,0);

clSetKernelArg(kernel, 0, sizeof(cl_mem),(void*) &mem);
clSetKernelArg(kernel, 1, sizeof(int),   (void*) &N);

clEnqueueNDRangeKernel(cmd_q,kernel,1,0,&N,&N,0,0,0);

clEnqueueReadBuffer(cmd_q, mem, CL_TRUE, 0,
                    N*sizeof(float), pData,0,0,0);

clReleaseMemObject(mem);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmd_q);
clReleaseContext(ctx);
```

Оптимизации

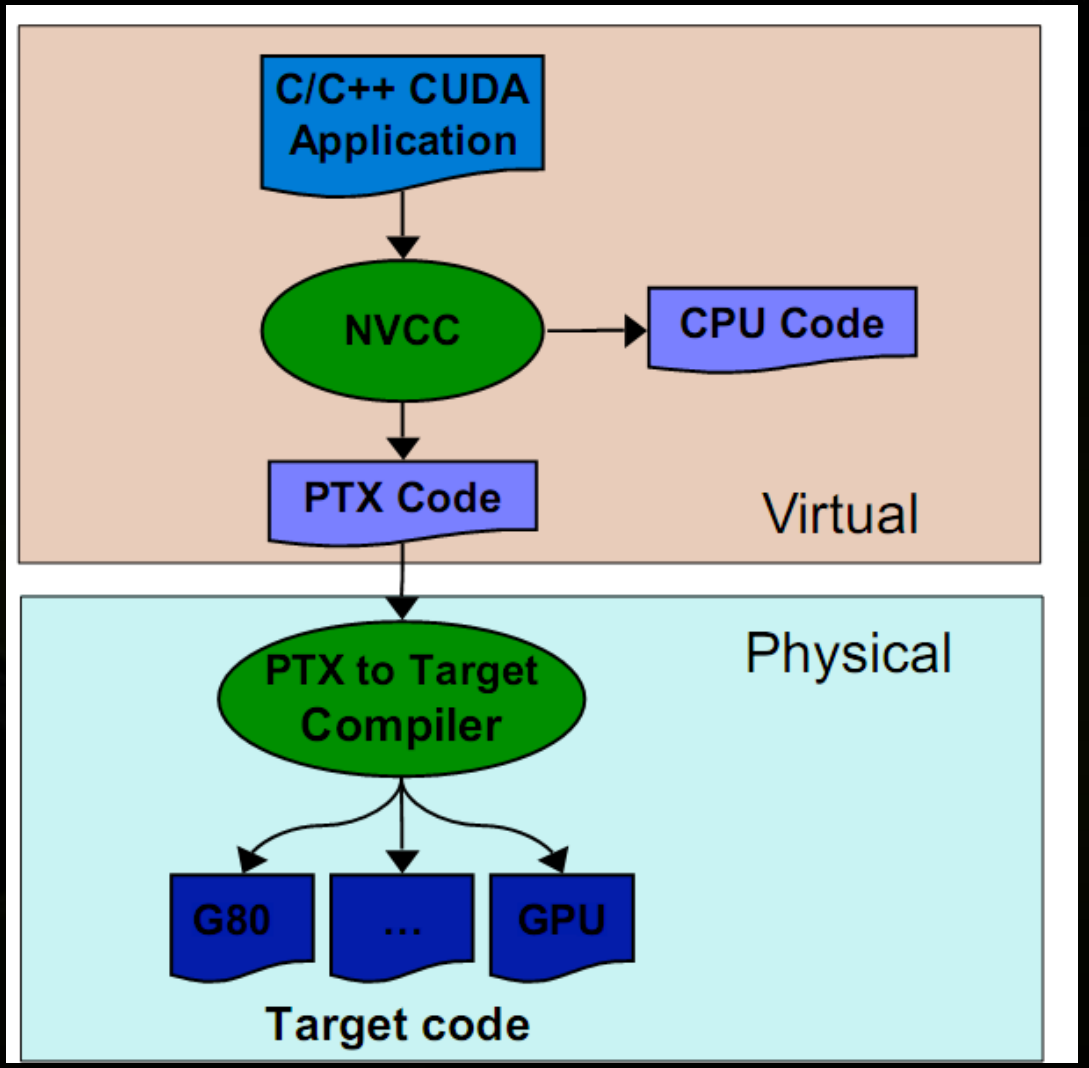
Ресурсы для разработчиков

- Инструментарий разработчика CUDA (CUDA toolkit)
  - Компилятор и библиотеки
  - Версии для Win32/64, MacOS, Linux доступны для загрузки
- Набор примеров и средств разработки (CUDA SDK)
  - Готовые решения на CUDA
  - Примеры использования
- Библиотеки
  - CUFFT, CUBLAS, CUDPP
- Инструменты для отладки и оптимизации
  - Nexus



Оптимизация

PTX



PTX

- Промежуточный ассемблер может показать много интересного
  - `--ptxas-options=-v`

```
__global__ void kernel(float *pData)
{
  float id = (float) threadIdx.x;
  pData[threadIdx.x] = _sinf( id );
}
```

```
__global__ void kernel(float *pData)
{
  float id = (float) threadIdx.x;
  pData[threadIdx.x] = sinf( id );
}
```

PTX

- Промежуточный ассемблер может показать много интересного
  - `--ptxas-options=-v`

```
__global__ void kernel(float *pData)
{
  float id = (float) threadIdx.x;
  pData[threadIdx.x] = _sinf( id );
}
```

```
2  .reg
```

```
__global__ void kernel(float *pData)
{
  float id = (float) threadIdx.x;
  pData[threadIdx.x] = sinf( id );
}
```

```
10  .reg        !
28 bytes lmem !
```

PTX

- Промежуточный ассемблер может показать много интересного
  - `--keep`

```
__global__ void kernel(float3 *pData)
{
  pData[threadIdx.x] = f3();
}
```

```
__global__ void kernel(float4 *pData)
{
  pData[threadIdx.x] = f4();
}
```

PTX

- Промежуточный ассемблер может показать много интересного
  - `--keep`

```
{ // ...
  mov.u16         %rh1, %tid.x;     //
  mul.wide.u16    %r1, %rh1, 12;    //
  ld.param.u32 %r2, [__cudaparm__kernel_f3_pD]
  add.u32         %r3, %r2, %r1;    //
  mov.f32         %f1, 0f00000000; // 0
  st.global.f32   [%r3+0], %f1;     // id:14
  mov.f32         %f2, 0f00000000; // 0
  st.global.f32   [%r3+4], %f2;     // id:15
  mov.f32         %f3, 0f00000000; // 0
  st.global.f32   [%r3+8], %f3;     // id:16
  .loc   14       18       0
  exit;                             //
  $LDWend__Z9kernel_P6float3:
} // _Z9kernel_P6float3
```

```
{ // ...
  mov.u16       %rh1, %tid.x;
  mul.wide.u16 %r1, %rh1, 16;
  ld.param.u32 %r2,[__cudaparm__kernel_f4_pD
  add.u32       %r3, %r2, %r1;                  //
  mov.f32       %f1, 0f00000000;                // 0
  mov.f32       %f2, 0f00000000;                // 0
  mov.f32       %f3, 0f00000000;                // 0
  mov.f32       %f4, 0f00000000;                // 0
  st.global.v4.f32 [%r3+0], {%f1,%f2,%f3,%f4};
  .loc   14       23       0
  exit;                                 //
  $LDWend__Z9kernel_P6float4:
} // _Z9kernel_P6float4
```

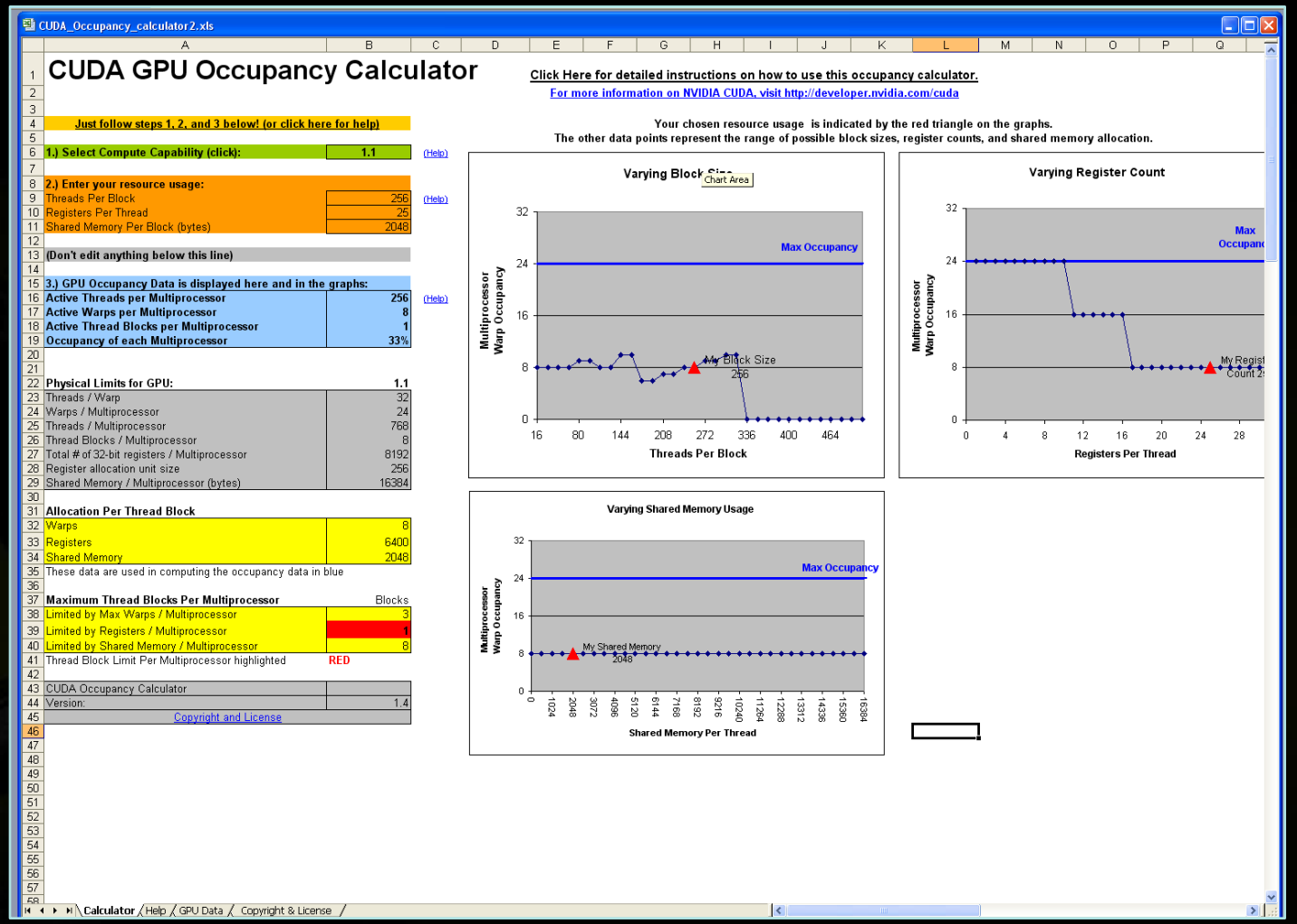
Occupancy

- Покрытие латентностей: инструкции потока выполняются последовательно
- Исполнение других потоков необходимо для покрытия латентностей
- Занятость: отношение активных варпов к максимально возможному
  - В архитектуре Tesla 32 варпа на SM

Occupancy

- Увеличение занятости приводит к лучшему покрытию латентностей
- После определенной точки (~50%), происходит насыщение
- Занятость ограничена достыпными ресурсами:
  - Регистры
  - Разделяемая память

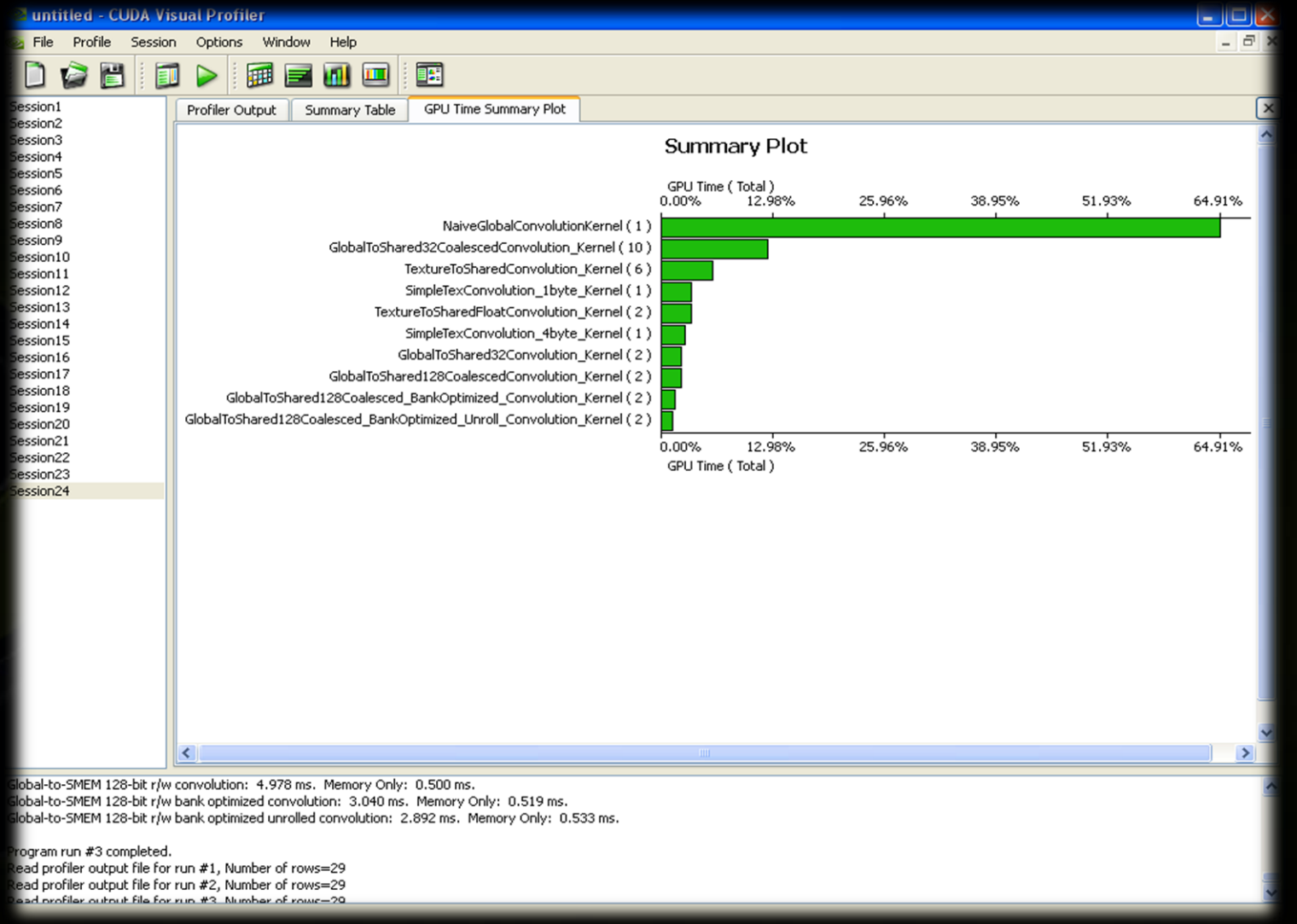
Оптимизация



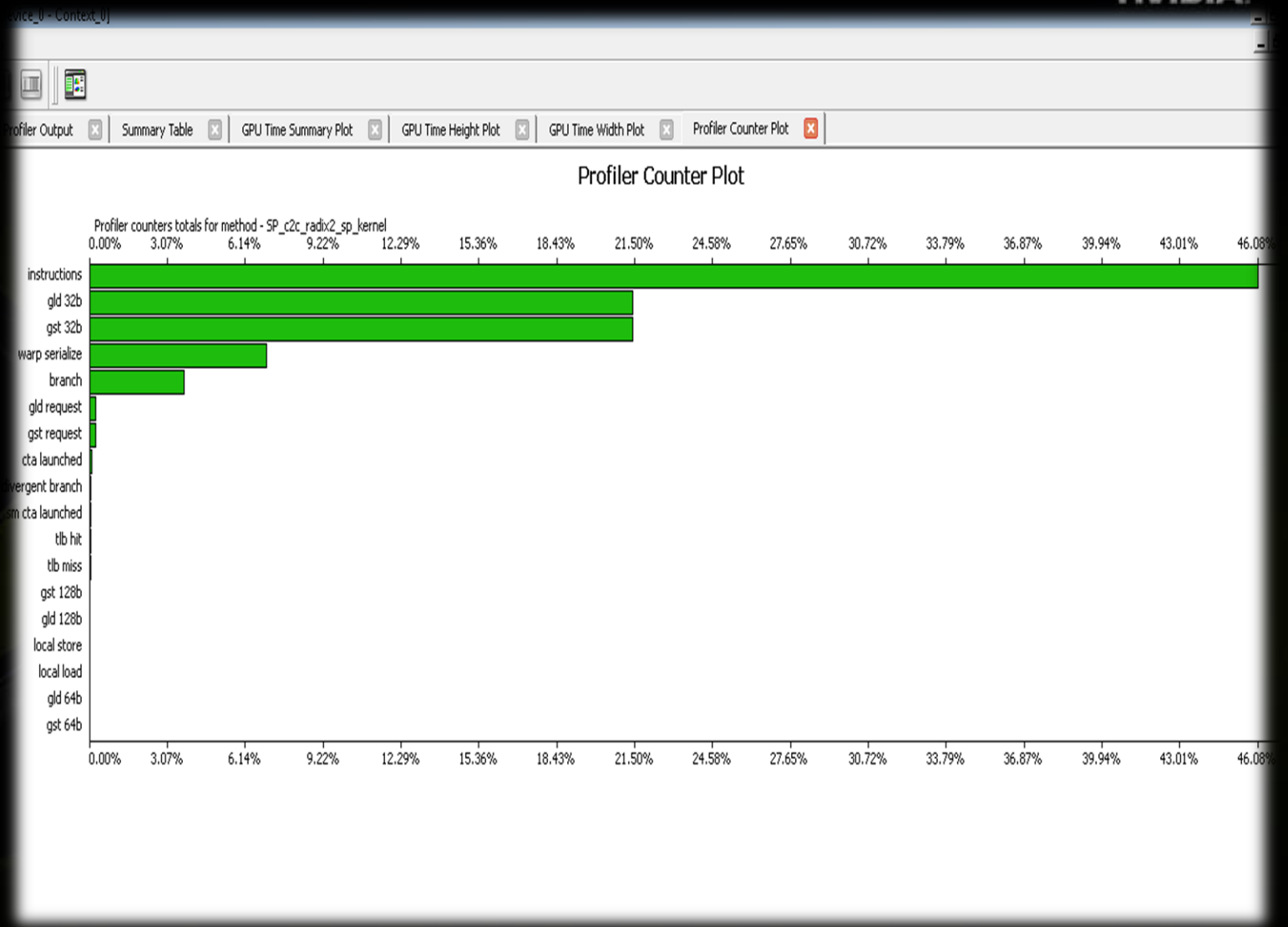
Occupancy Calculator Spreadsheet

Оптимизация

Visual Profiler



Оптимизация



Profiler Counter Plot

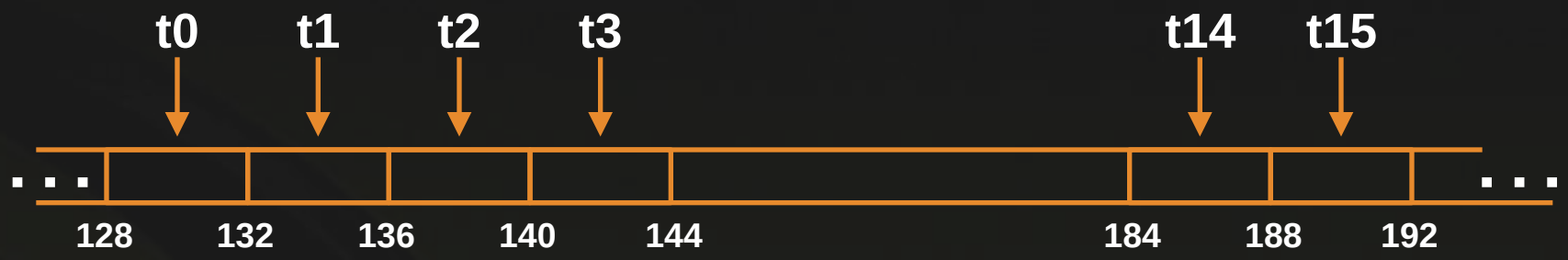
Особенности памяти

- Константная: обращение в один адрес всем варпом
- Разделяемая: обращение в
  - Разные банки
  - В один адрес (broadcast)
- Текстурная: обращение локализованно
- Глобальная: обращения объеденены в одну транзакцию
- Host: минимизировать трансферы

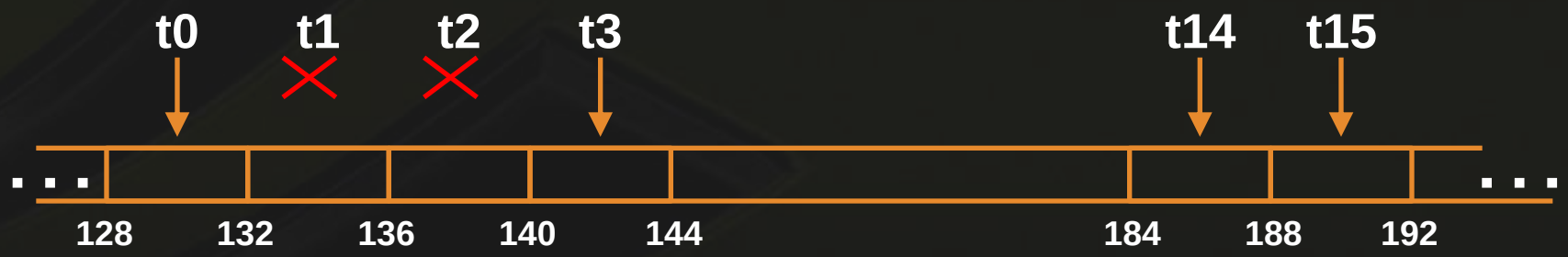
Coalescing GMEM
Наиболее важная оптимизация

- Координированное чтение / запись (16 потоков)
- Непрерывный участок памяти:
  - 64 bytes—каждый поток читает 32-битное слово: int, float, …
  - 128 bytes— каждый поток читает 64-битное слово: int2, float2, double…
  - 256 bytes— каждый поток читает 128-битное слово: int4, float4, …
- Ограничения:
  - Начальный адрес должен быть выровнен по размеру
  - *k*ый поток полу-варпа должен обращаться к *k*ому элемету
- Исключение: Потоки могут быть неактивными
  - Например если выбрана другая ветка исполнения

Coalesced Доступ:
Чтение float (32-bit)

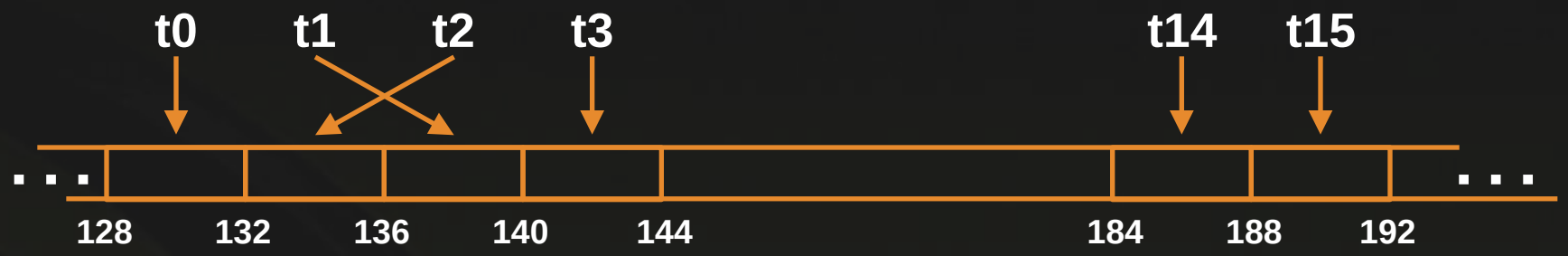


Все потоки активны

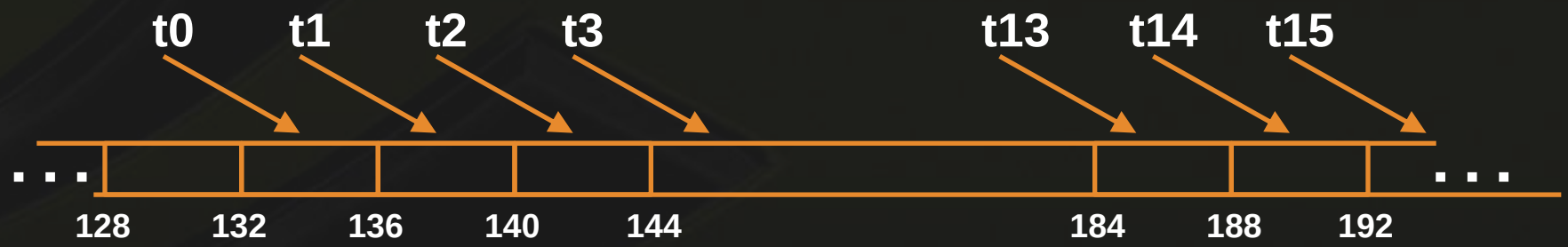


Потоки могут не учавствовать

Uncoalesced Доступ:
Чтение float (32-bit)



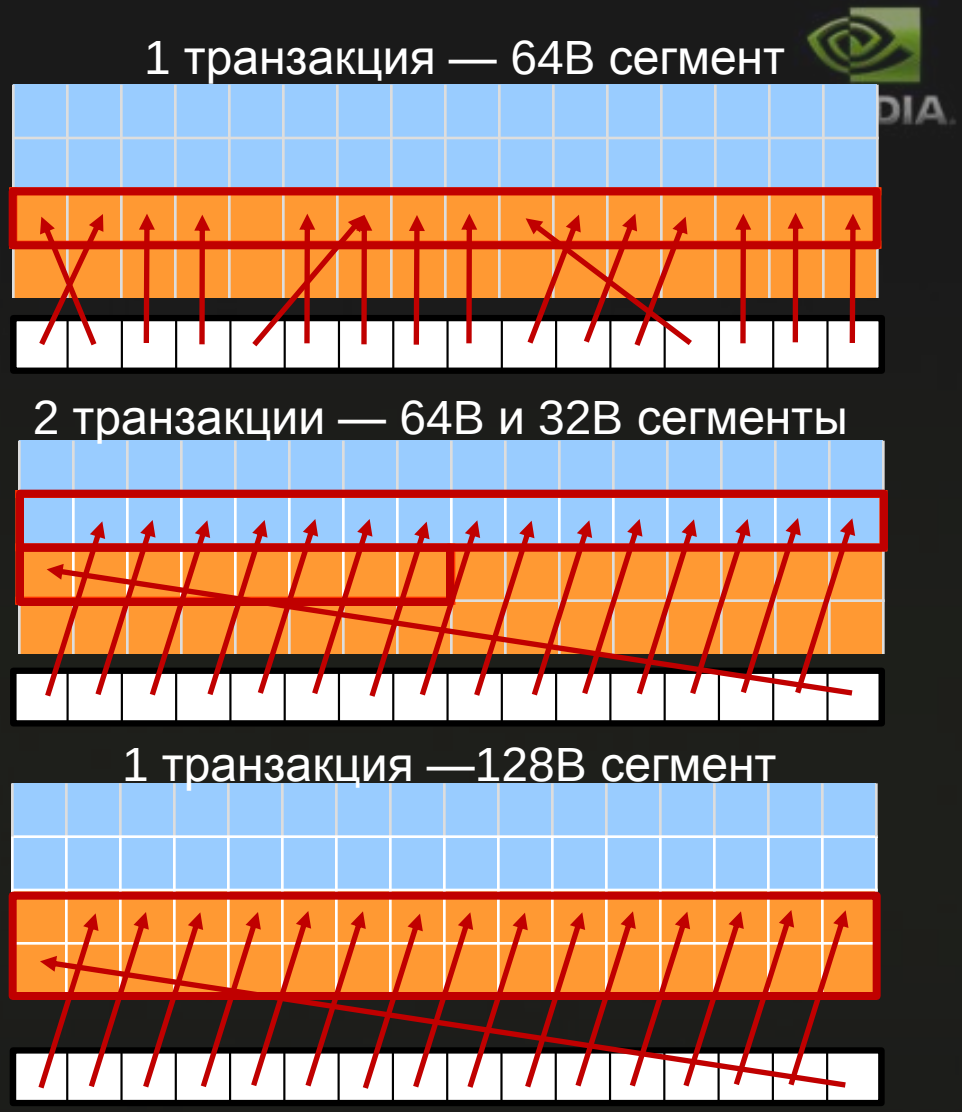
Произвольный доступ



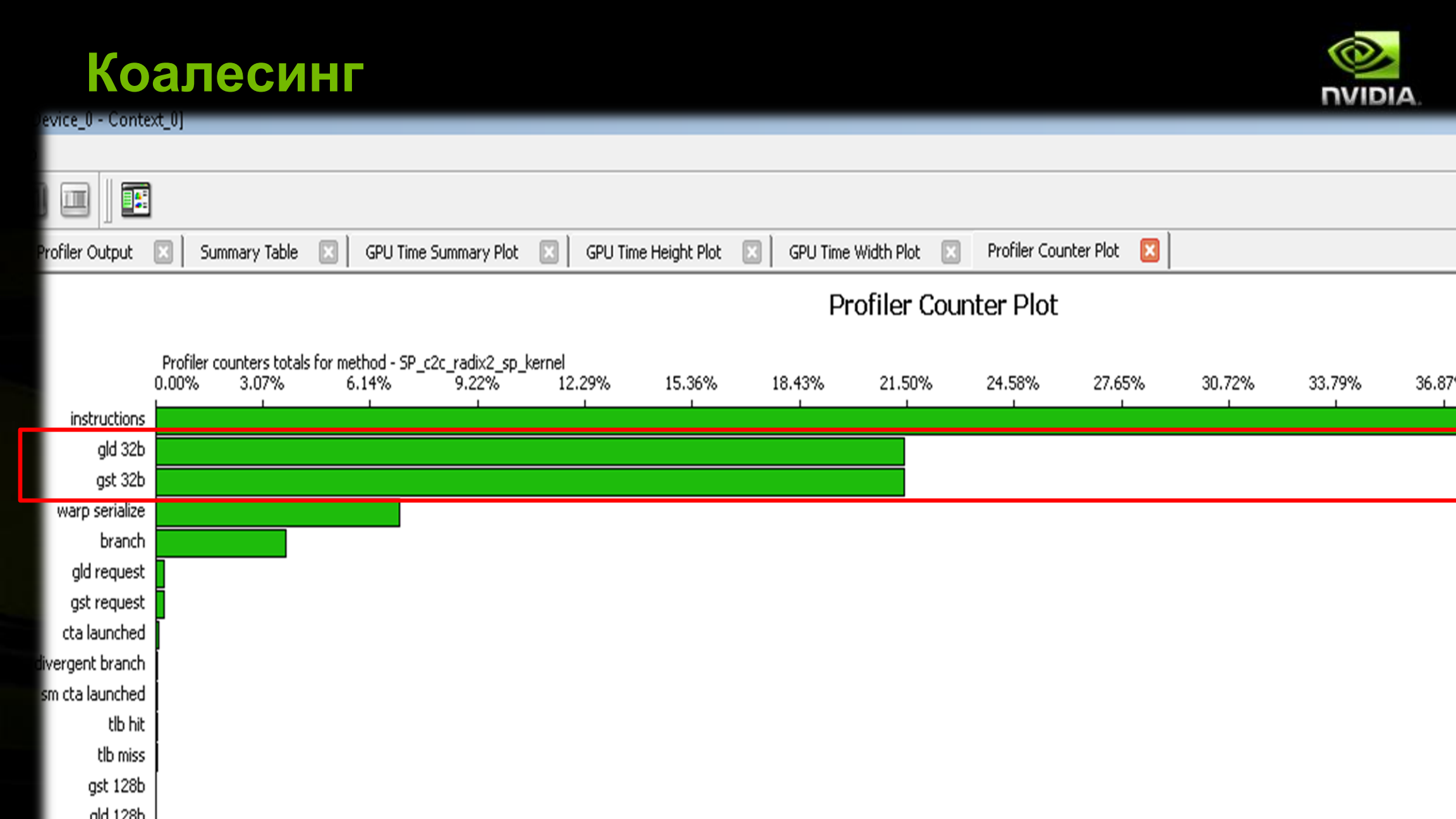
Не выровненный адрес (не кратный 64b)

Coalescing SM 1.2

- Объединение транзакций происходит при любом шаблоне доступа который ложится в сегмент размера: 32B для 8-bit слов, 64B для 16-bit слов, 128B для 32- и 64-bit слов
- Выравниевание более не играет такой роли но сильно хаотичные чтение / запись все еще медленные

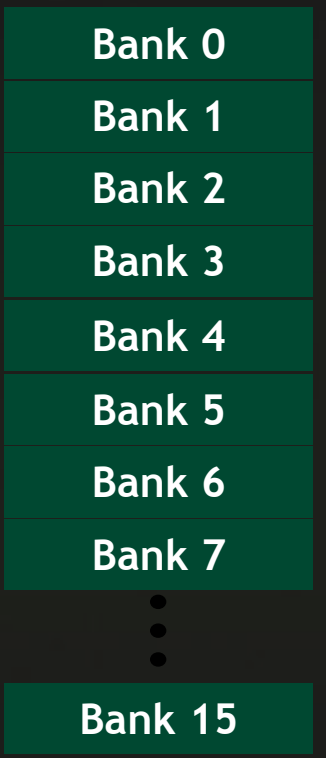


Коалесинг


evice_0 - Context_0]
Profiler Output
Summary Table
GPU Time Summary Plot
GPU Time Height Plot
GPU Time Width Plot
Profiler Counter Plot
Profiler Counter Plot
Profiler counters totals for method - SP_c2c_radix2_sp_kernel
0.00%
3.07%
6.14%
9.22%
12.29%
15.36%
18.43%
21.50%
24.58%
27.65%
30.72%
33.79%
instructions
gld 32b
gst 32b
warp serialize
branch
gld request
gst request
cta launched
divergent branch
sm cta launched
tlb hit
tlb miss
gst 128b

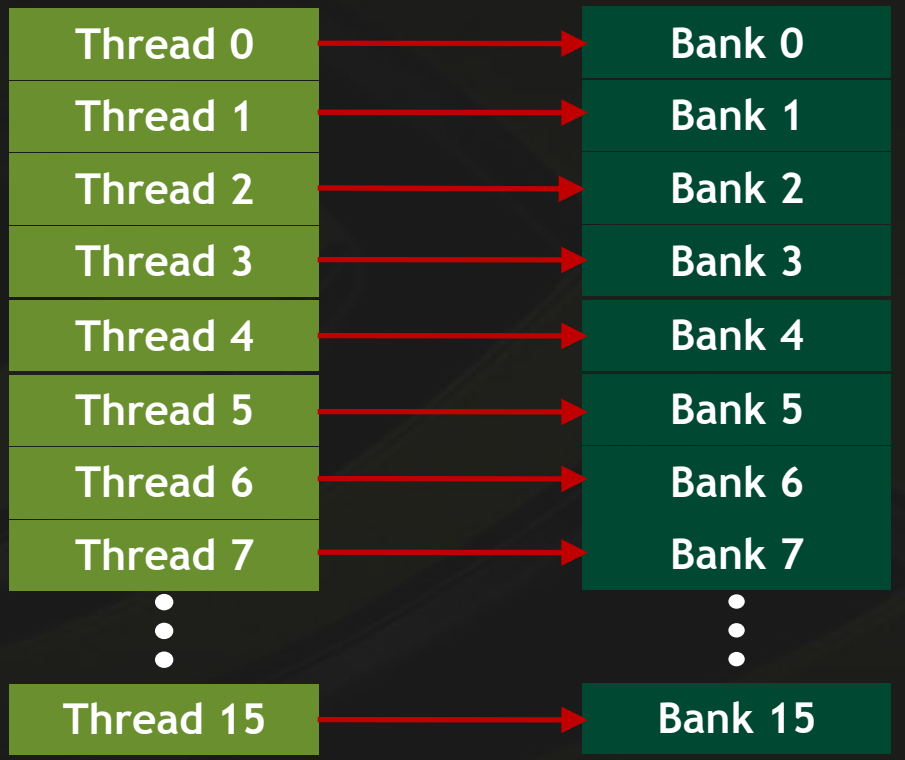
Банки разделяемой памяти

- Одновременное обращение потоков к разделяемой памяти
  - Память разбивается на банки
  - Необходимо для обеспечения высокой пропускной способности
- Каждый банк работает с одним адресом одновременно
- Одновременное обращение в один банк приводит к конфликтам
  - Доступ сериализуется

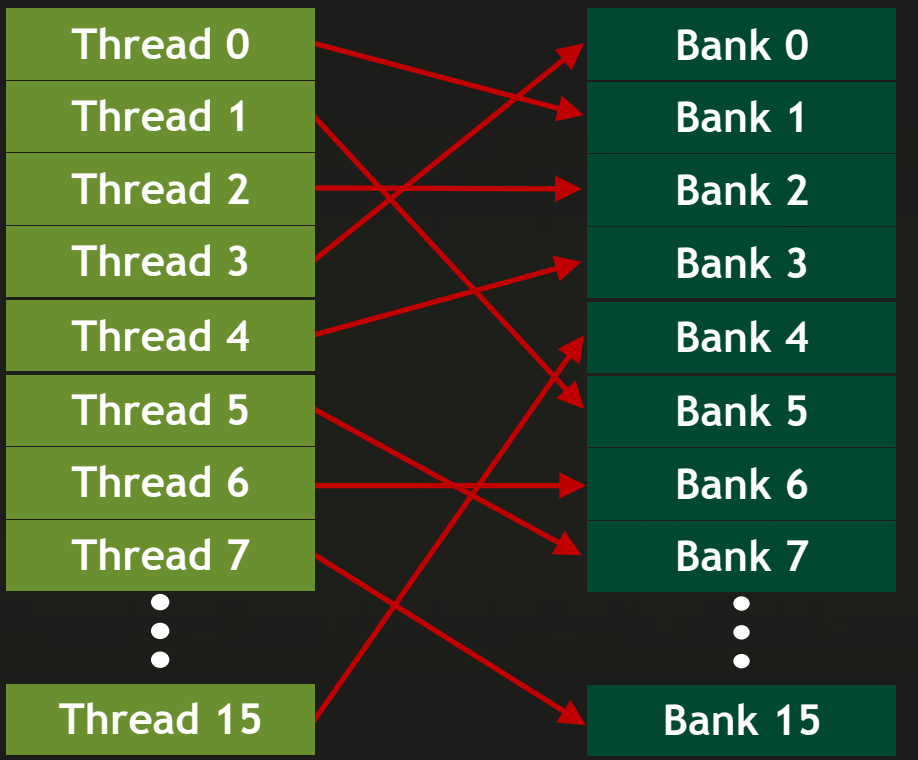


Конфликты банков

Нет конфликтов
- Прямая адресация, с шагом == 1

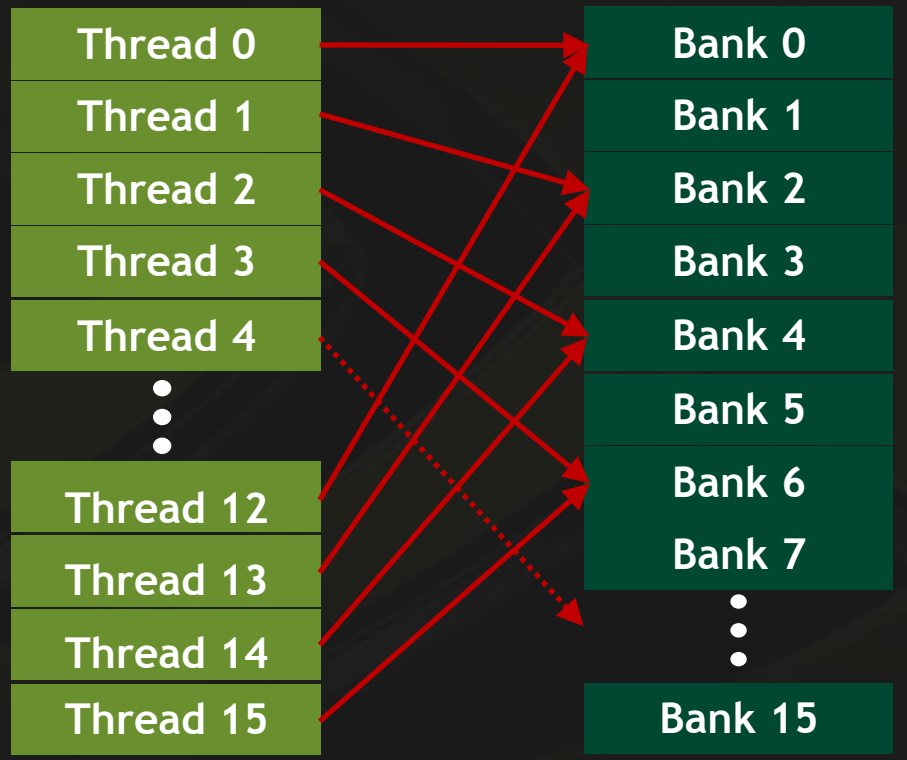


Нет конфликтов
- **Хаотичная адресация**

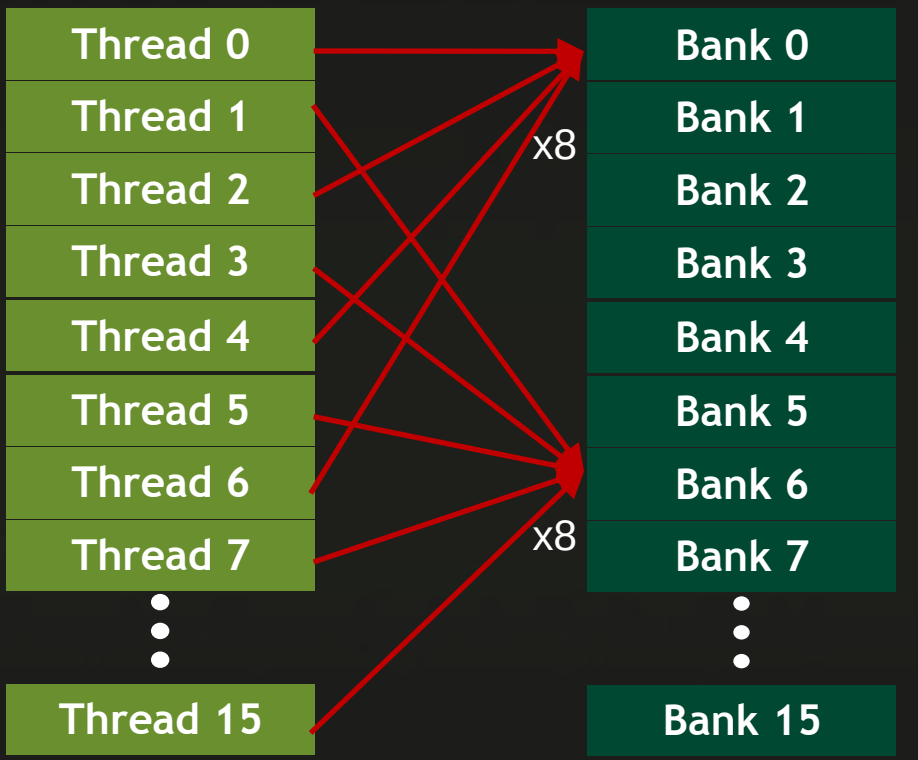


Конфликты банков

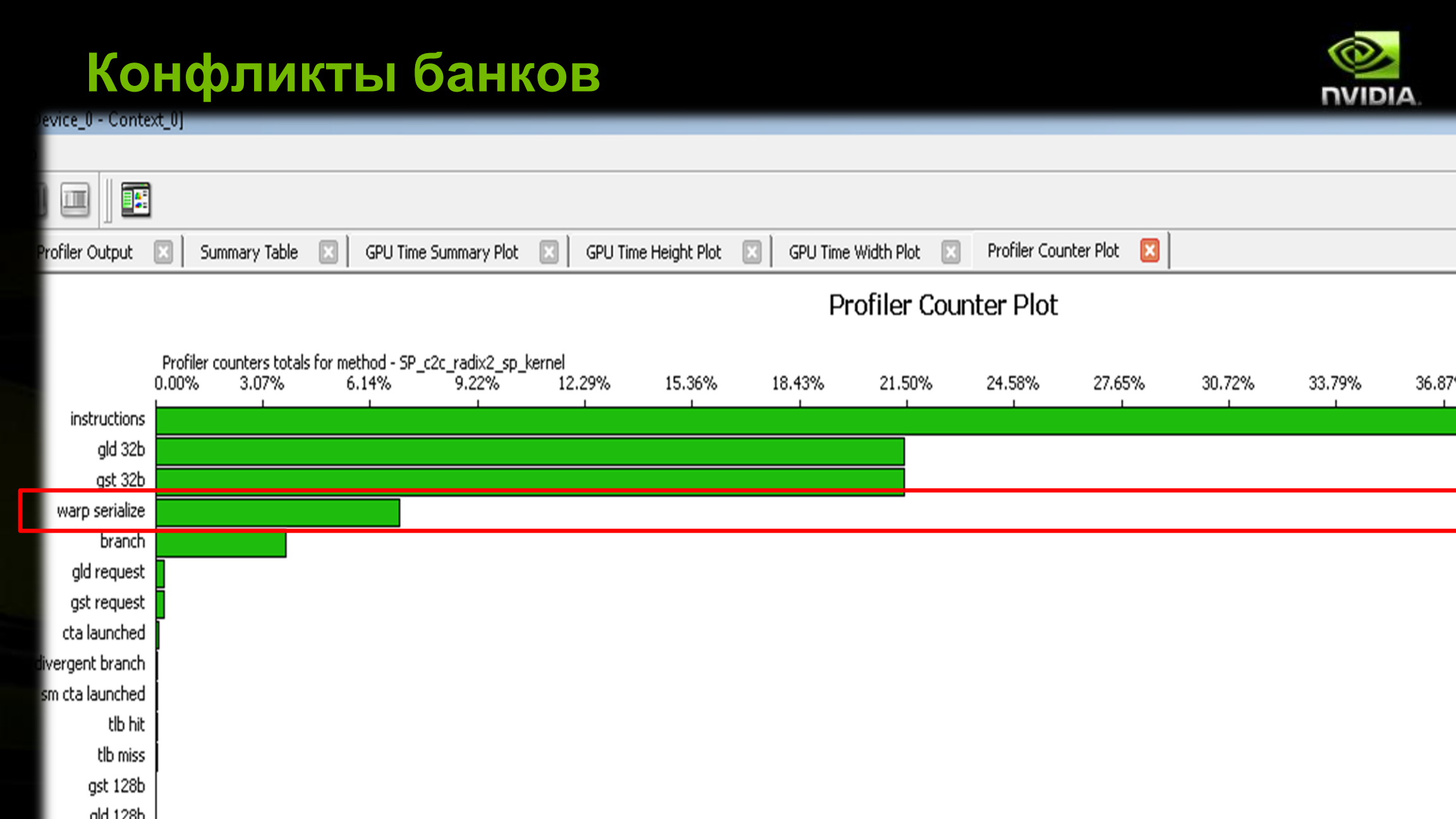
2x Конфликты



8x Конфликты



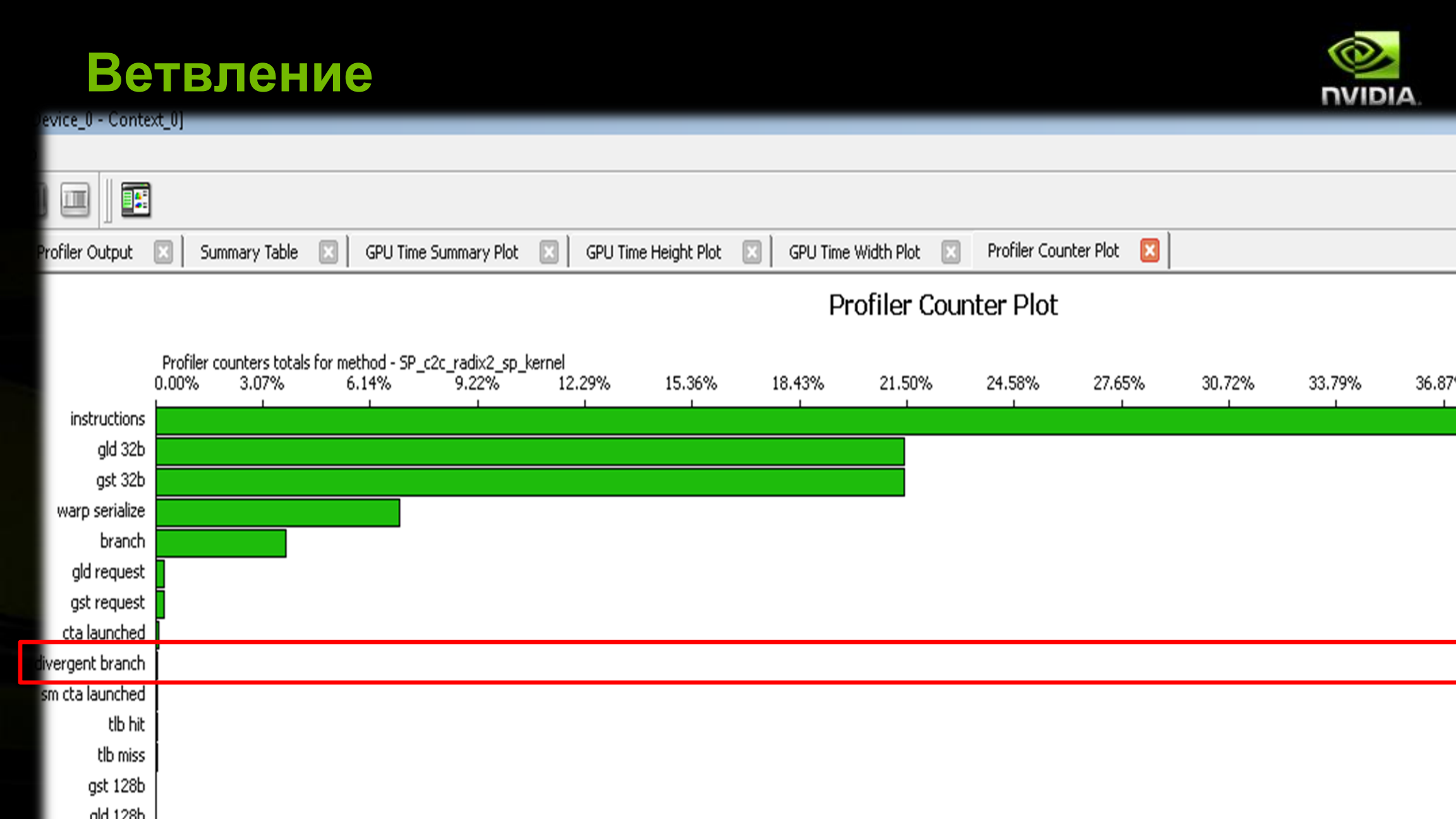
Конфликты банков



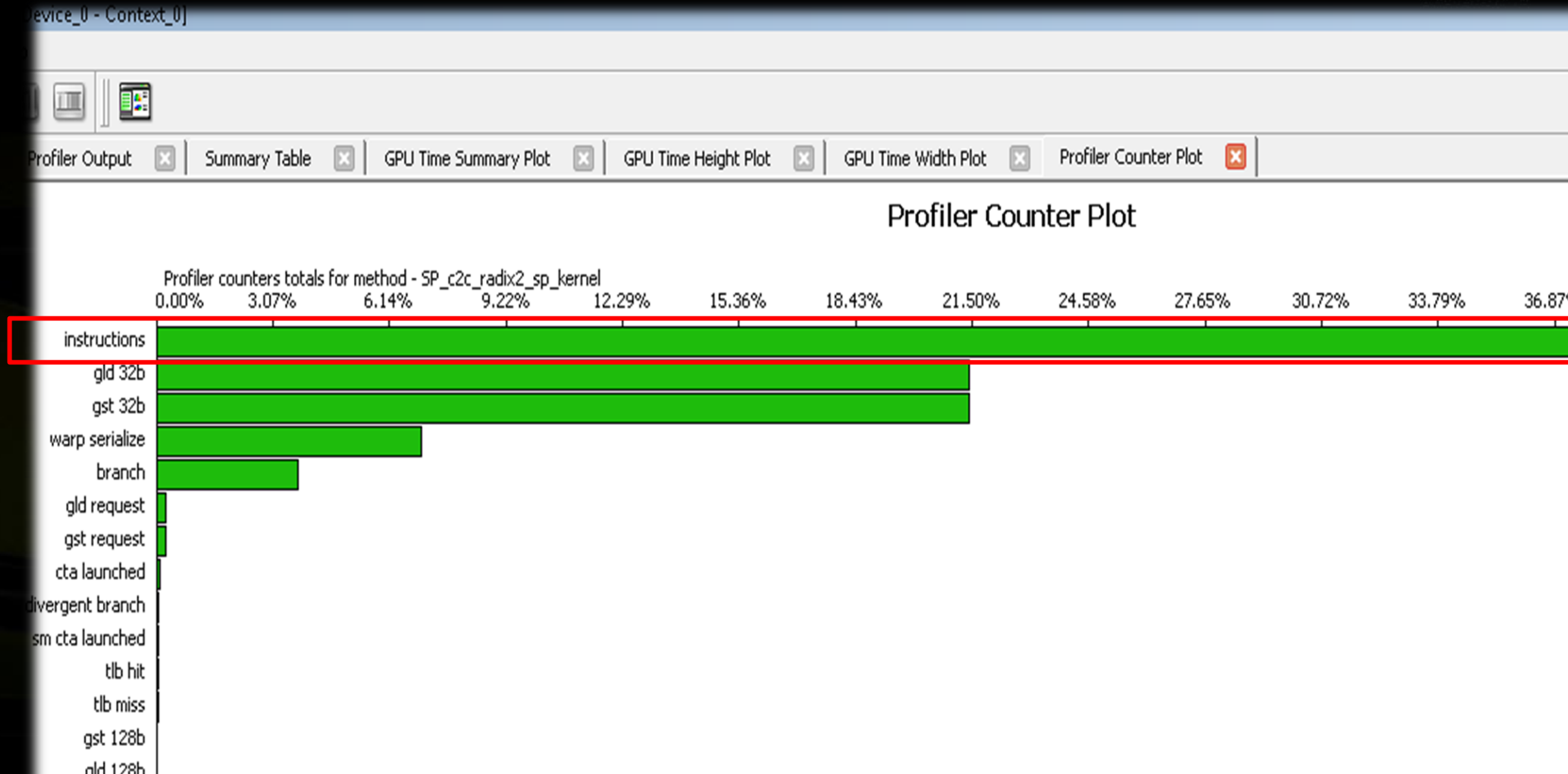
Ветвление

- Если происходит ветвление внутри варпа, то разные ветви исполнения сериализуются
- Увеличивается общее количество инструкций
- Если ветвление происходит между варпами, то штраф минимальный

Ветвление


evice_0 - Context_0]
Profiler Output
Summary Table
GPU Time Summary Plot
GPU Time Height Plot
GPU Time Width Plot
Profiler Counter Plot
Profiler Counter Plot
Profiler counters totals for method - SP_c2c_radix2_sp_kernel
0.00%
3.07%
6.14%
9.22%
12.29%
15.36%
18.43%
21.50%
24.58%
27.65%
30.72%
33.79%
instructions
gld 32b
gst 32b
warp serialize
branch
gld request
gst request
cta launched
divergent branch
sm cta launched
tlb hit
tlb miss
gst 128b

Инструкции


evice_0 - Context_0]
Profiler Output
Summary Table
GPU Time Summary Plot
GPU Time Height Plot
GPU Time Width Plot
Profiler Counter Plot
Profiler Counter Plot
Profiler counters totals for method - SP_c2c_radix2_sp_kernel
0.00%
3.07%
6.14%
9.22%
12.29%
15.36%
18.43%
21.50%
24.58%
27.65%
30.72%
33.79%
instructions
gld 32b
gst 32b
warp serialize
branch
gld request
gst request
cta launched
divergent branch
sm cta launched
tlb hit
tlb miss
gst 128b

Инструкции

- Следить за ветвлением
- Заменить часть вычислений на look-up таблицу
- Интринсики
  - __sinf(); __cosf(); expf()
  - __[u]mul24()
  - __fdividef()
  - __[u]sad()

Мифы о вычислениях на GPU

Мифы о вычислениях на GPU

- **Миф 1:** ***GPU исполняет программы как графические шейдеры***

Мифы о вычислениях на GPU

- ~~*Миф 1: GPU исполняет программы как графические шейдеры*~~

- **Это не так – CUDA использует вычислительные ресурсы GPU напрямую**

Мифы о вычислениях на GPU

- **Миф 2:** ***Архитектурно, GPU представляют из себя***
  - ***Очень широкие SIMD-машины (1000) ...***
  - ***… с крайне неэффективной поддержкой ветвления ...***
  - ***… которые ориентированые на обработку 4-х мерных векторов***

Мифы о вычислениях на GPU

- ~~Миф 2: Архитектурно, GPU представляют из себя~~
  - ~~Очень широкие SIMD-машины (1000) ...~~
  - ~~... с крайне неэффективной поддержкой ветвления ...~~
  - ~~... которые ориентированые на обработку 4-х мерных векторов~~

- Современные GPU используют SIMT исполнение, с размером блока (warp) в 32 элемента и скалярными потоками

Мифы о вычислениях на GPU

- **Миф 3:** ***GPU неэффективны с точки зрения энергопотребления***

Мифы о вычислениях на GPU

- ~~*Миф 3: GPU неэффективны с точки зрения энергопотребления*~~

- Показатель производительность/ватт для GPU от 4 до 20 раз выше чем у CPU
  - до 89 раз для некоторых применений

Мифы о вычислениях на GPU

- **Миф 4:** ***GPU не используют “настоящую” арифметику с плавающей точкой***

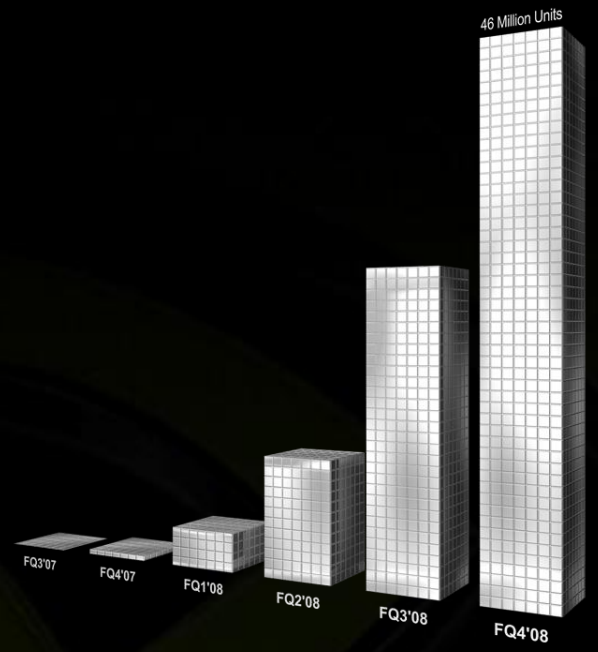
Мифы о вычислениях на GPU

- ~~*Миф 4: GPU не используют “настоящую” арифметику с плавающей точкой*~~

- GPU поддерживают стандарт IEEE-754
  - Сравнимы с другими вычислительными устройствами
  - Поддерживаются вычисления с двойной точностью

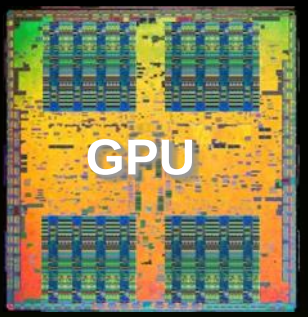
Поддержка операций с двойной точностью

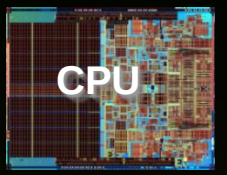
	NVIDIA GPU	SSE2	Cell SPE
Поддержка стандарта	IEEE 754	IEEE 754	IEEE 754
Режимы округления для FADD и FMUL	Все 4 режима IEEE: rtne, zero, inf, -inf	Все 4 режима IEEE: rtne, zero, inf, -inf	Только zero/truncate
Обработка денормализованных чисел	С полной скоростью	Поддерживается, 1000+ циклов	Не поддерживается
Поддержка NaN	Да	Да	Нет
Поддержка переполнений и бесконечностей	Да	Да	Бесконечности отсутствуют
Флаги	Нет	Да	Некоторые
FMA	Да	Нет	Да
Извлечение квадратного корня	Эмуляция на основе FMA	Аппаратная поддержка	Программная эмуляция
Деление	Эмуляция на основе FMA	Аппаратная поддержка	Программная эмуляция
Точность операции 1/x	24 bit	12 бит	12 бит
Точность операции 1/sqrt(x)	23 bit	12 бит	12 бит
Точность операций log2(x) и 2^x	23 bit	Нет	Нет

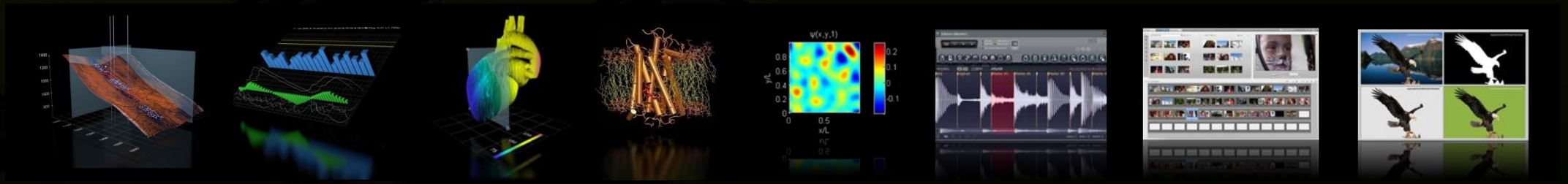


Вопросы ?

CUDA



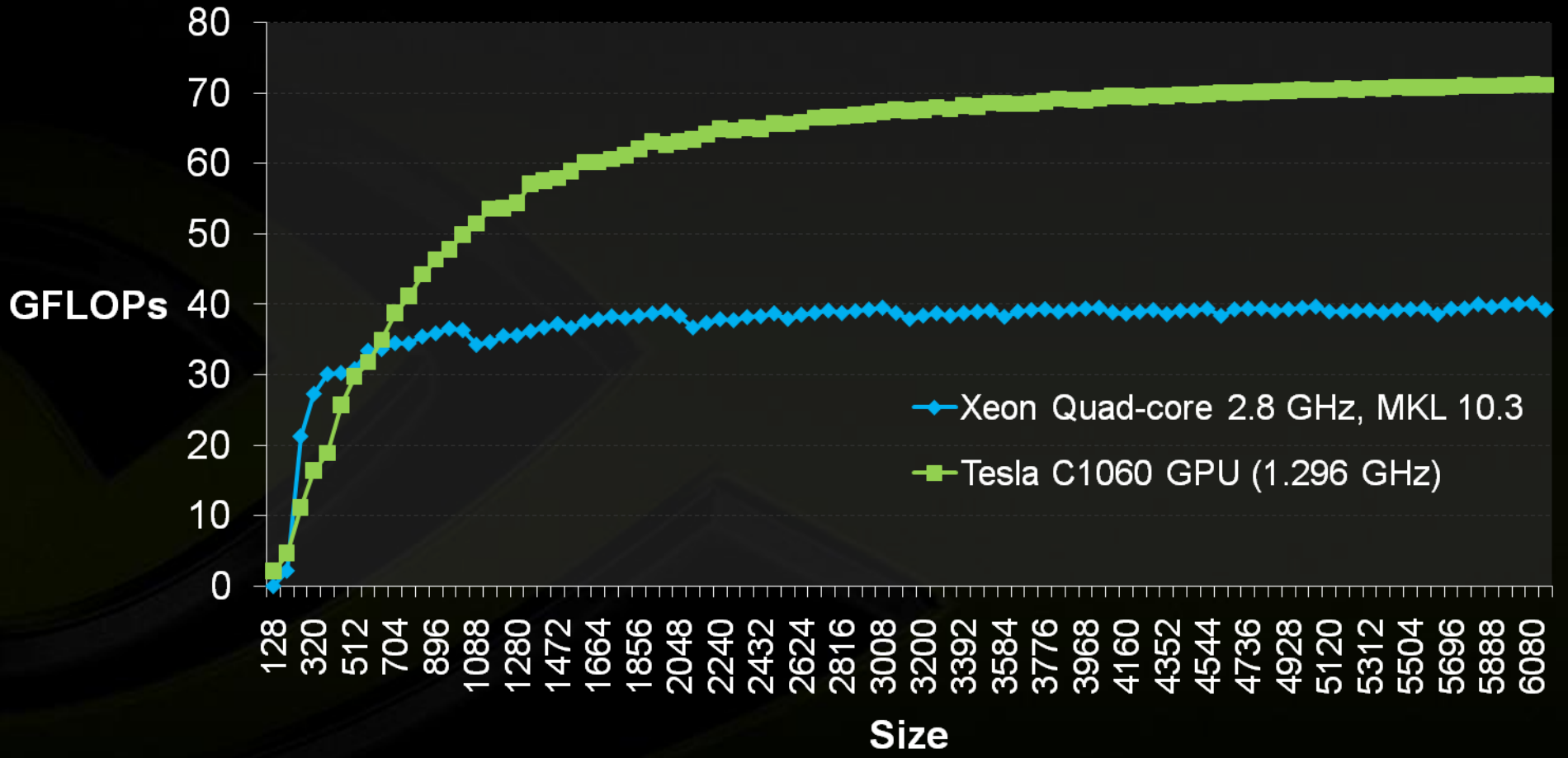




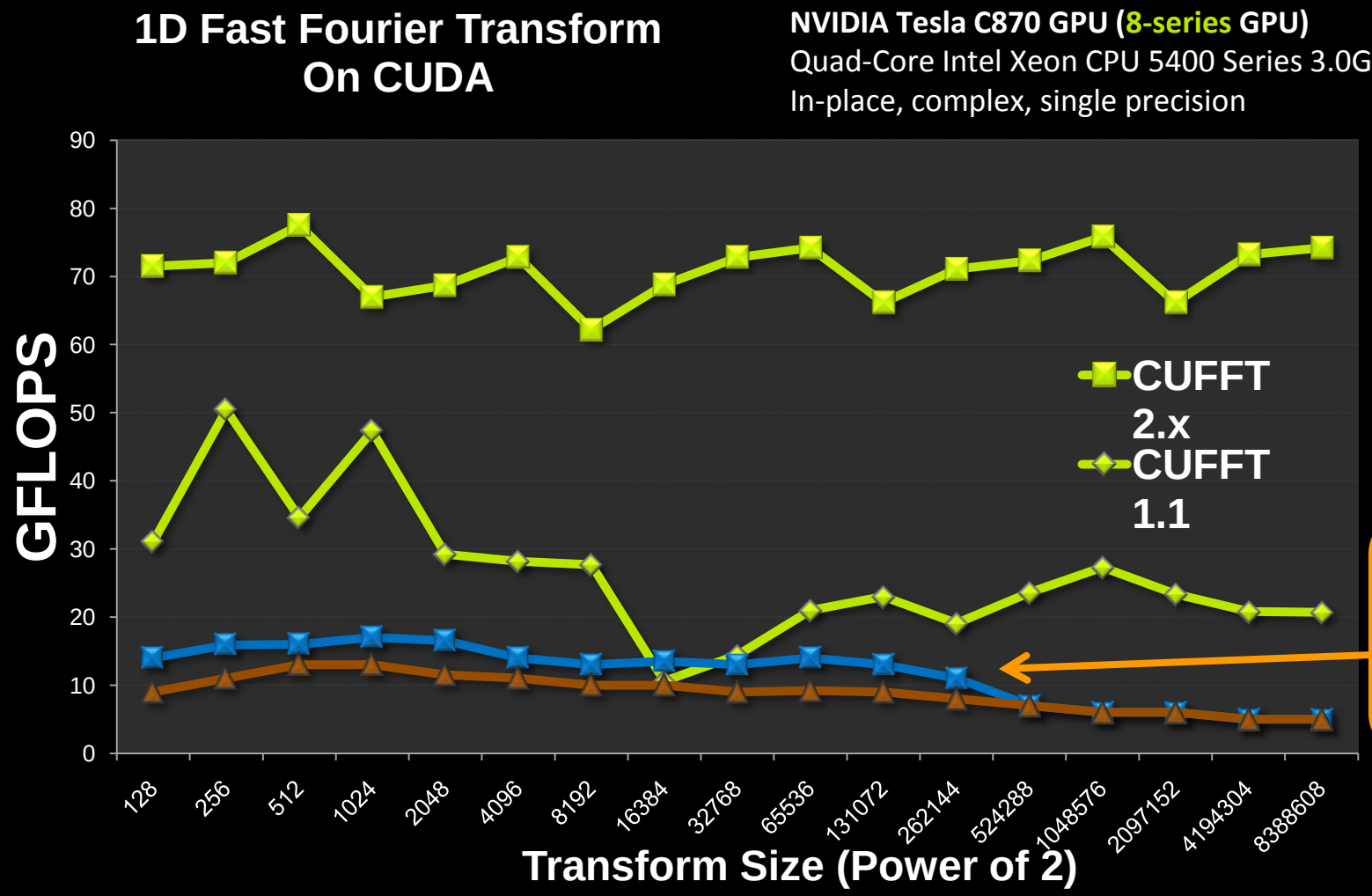
Oil & Gas | Finance | Medical | Biophysics | Numerics | Audio | Video | Imaging

Backup slides

Производительность DGEMM



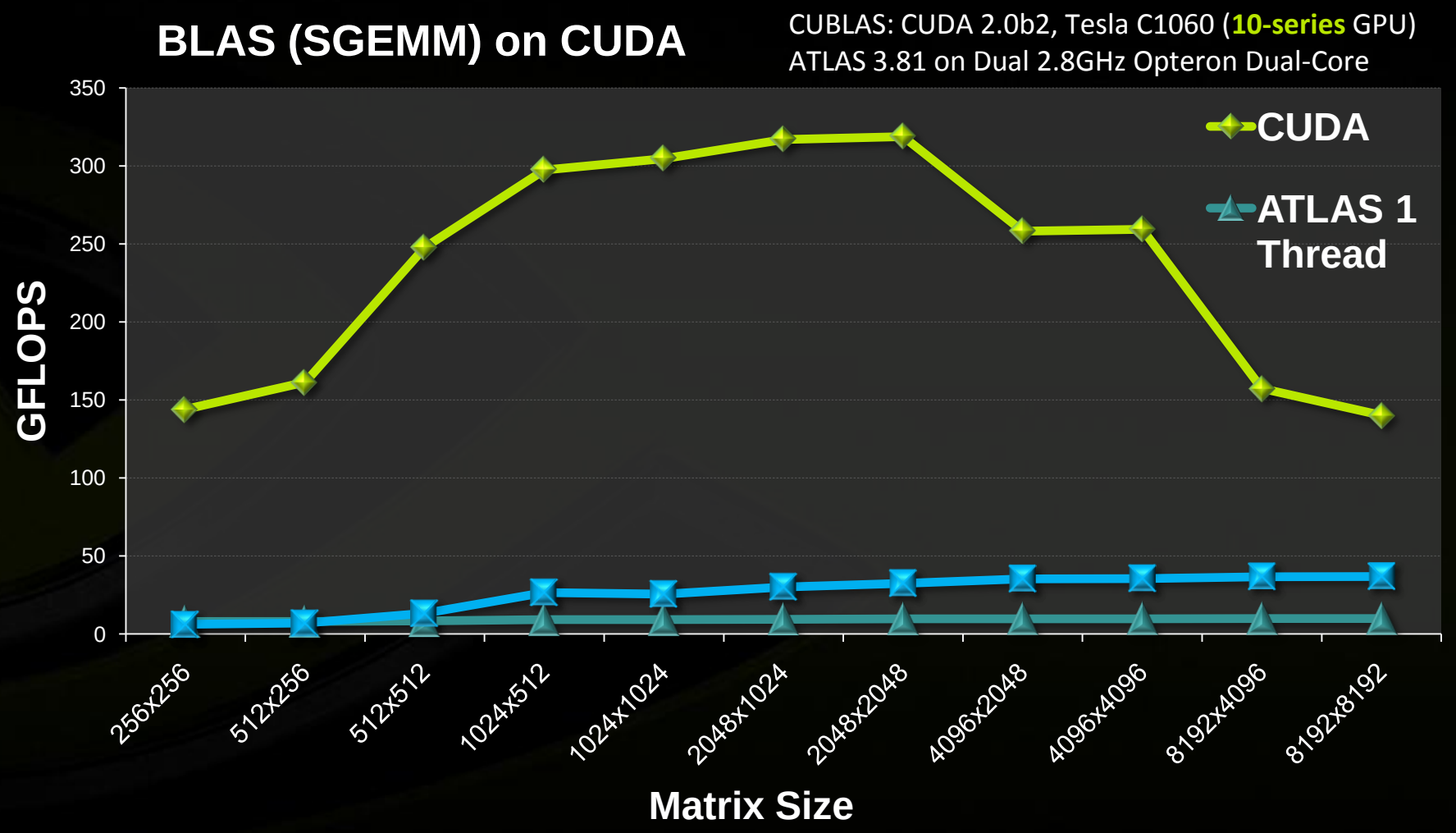
FFT Performance: CPU vs GPU (8-Series)



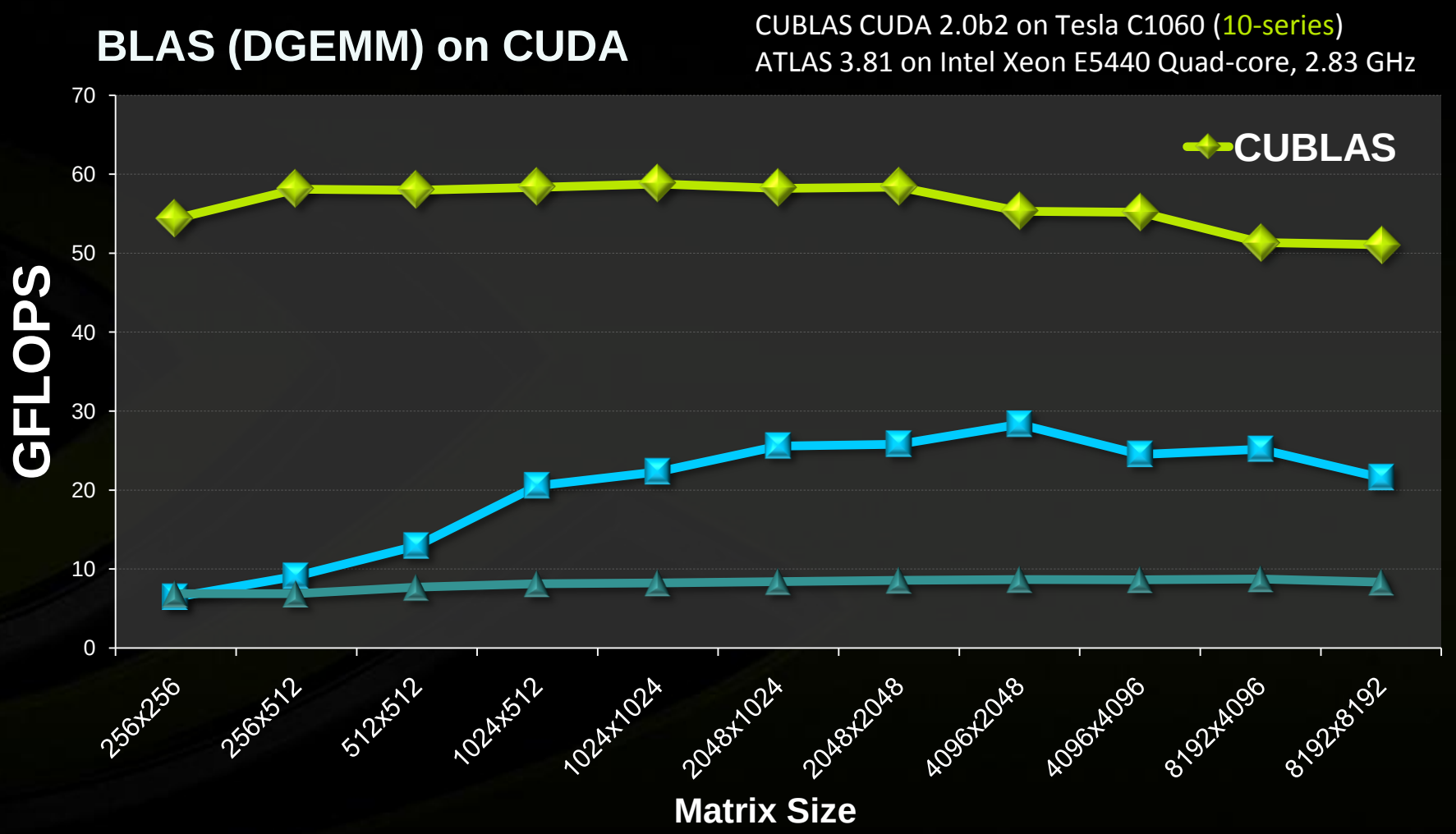
- *Intel FFT numbers calculated by repeating same FFT plan*
- *Real FFT performance is ~10 GFlops*

Source for Intel data : http://www.intel.com/cd/software/products/asmo-na/eng/266852.htm

Single Precision BLAS: CPU vs GPU (10-series)

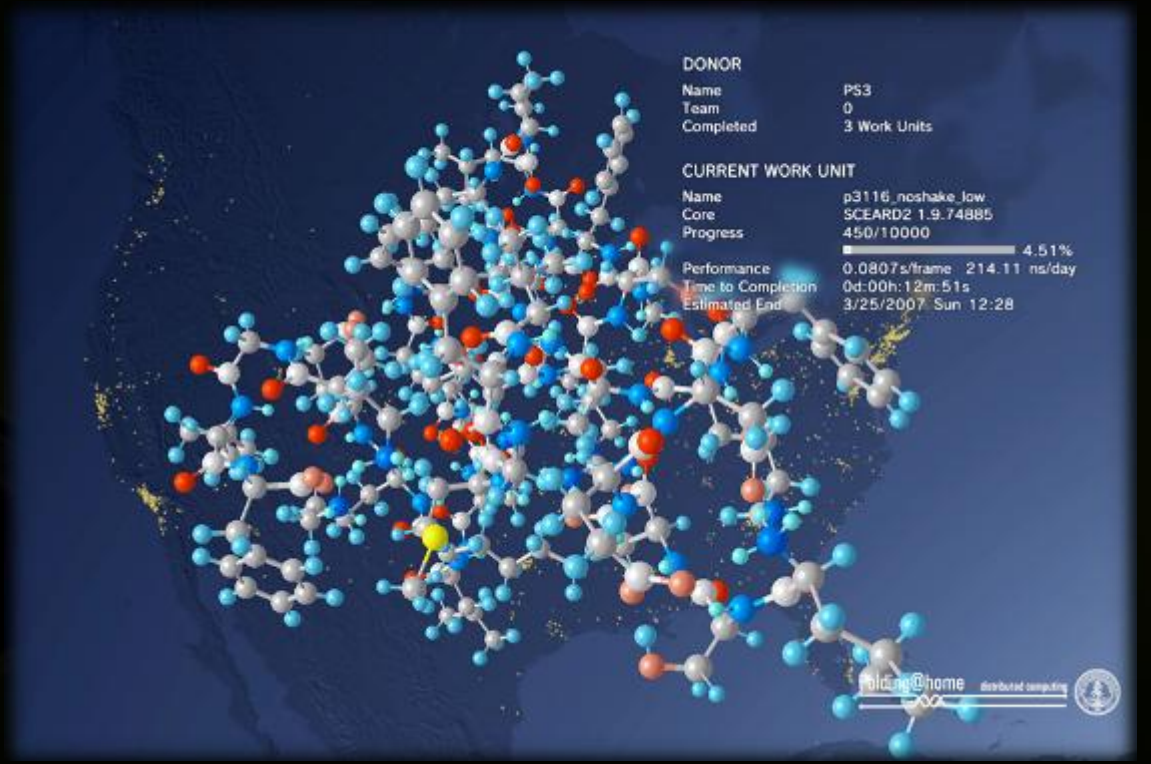


Double Precision BLAS: CPU vs GPU (10-series)



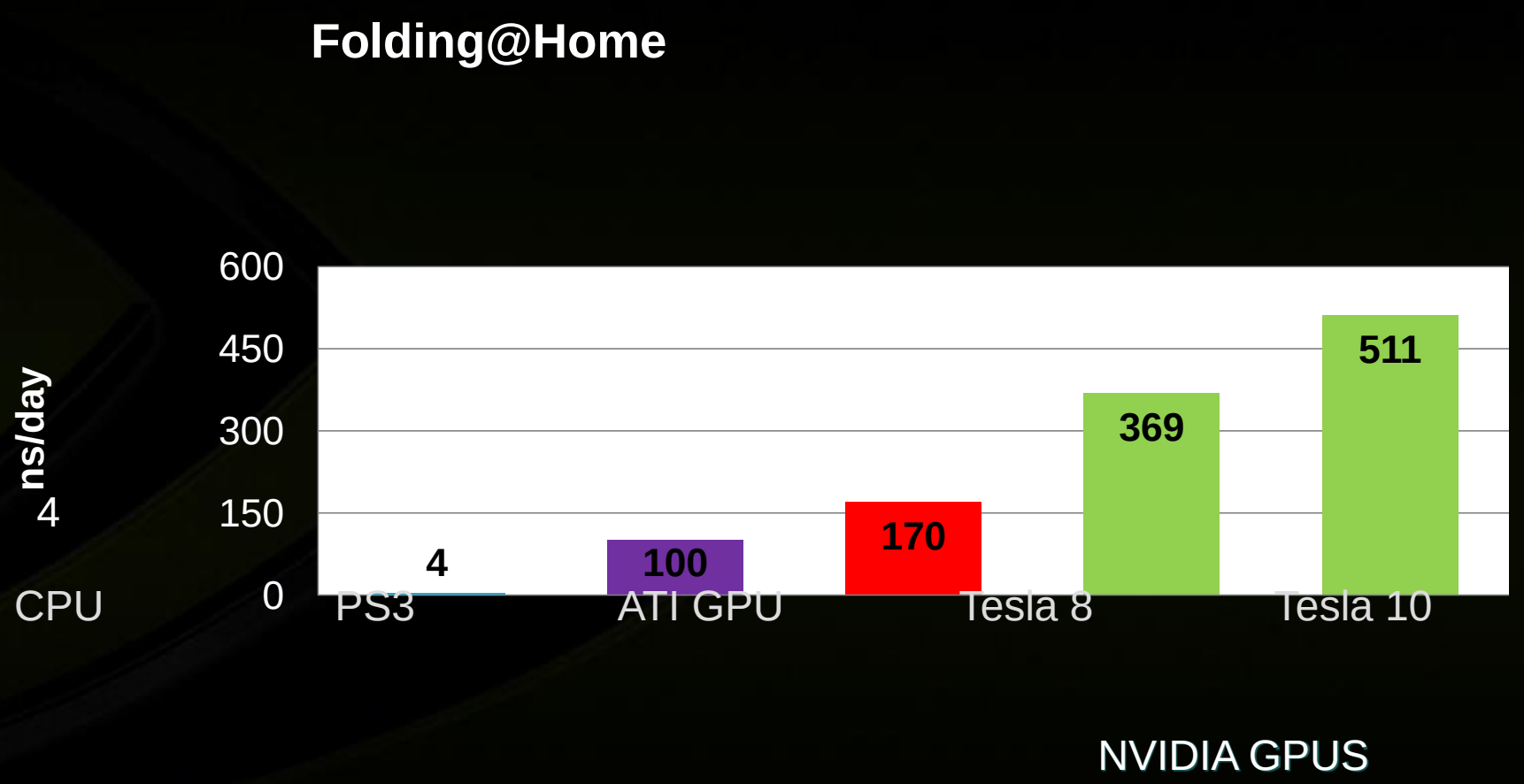
Folding@Home

- Distributed computing to study protein folding
  - Alzheimer's Disease
  - Huntington's Disease
  - Cancer
  - Osteogenesis imperfecta
  - Parkinson's Disease
  - Antibiotics



- CUDA client available now!

Molecular Dynamics : GROMACS

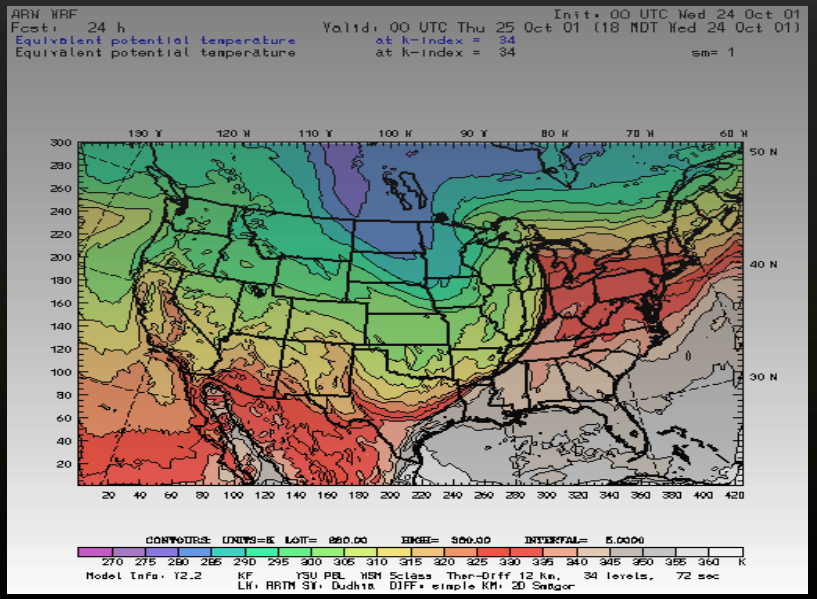


National Center for Atmospheric Research

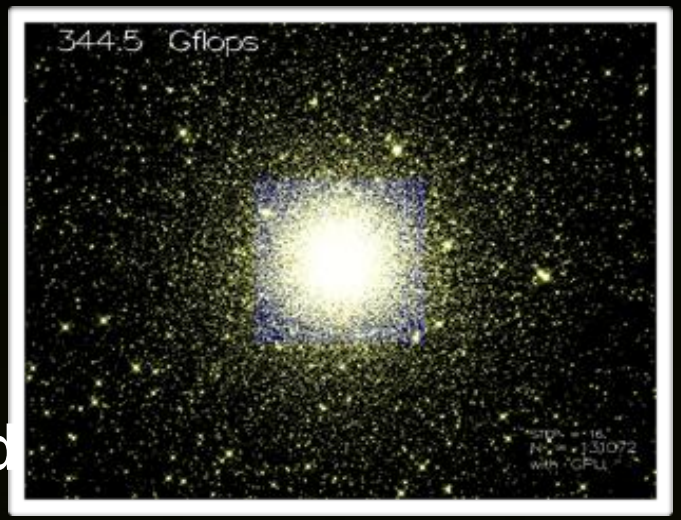
- Weather Research and Forecast (WRF) model
  - 4000+ registered users worldwide
  - First-ever release with GPU acceleration

12km CONUS WRF benchmark
Running on NCSA CUDA cluster

- Adapted 1% of WRF code to CUDA
- Resulted in 20% overall speedup
- Ongoing work to adapt more of WRF to CUDA



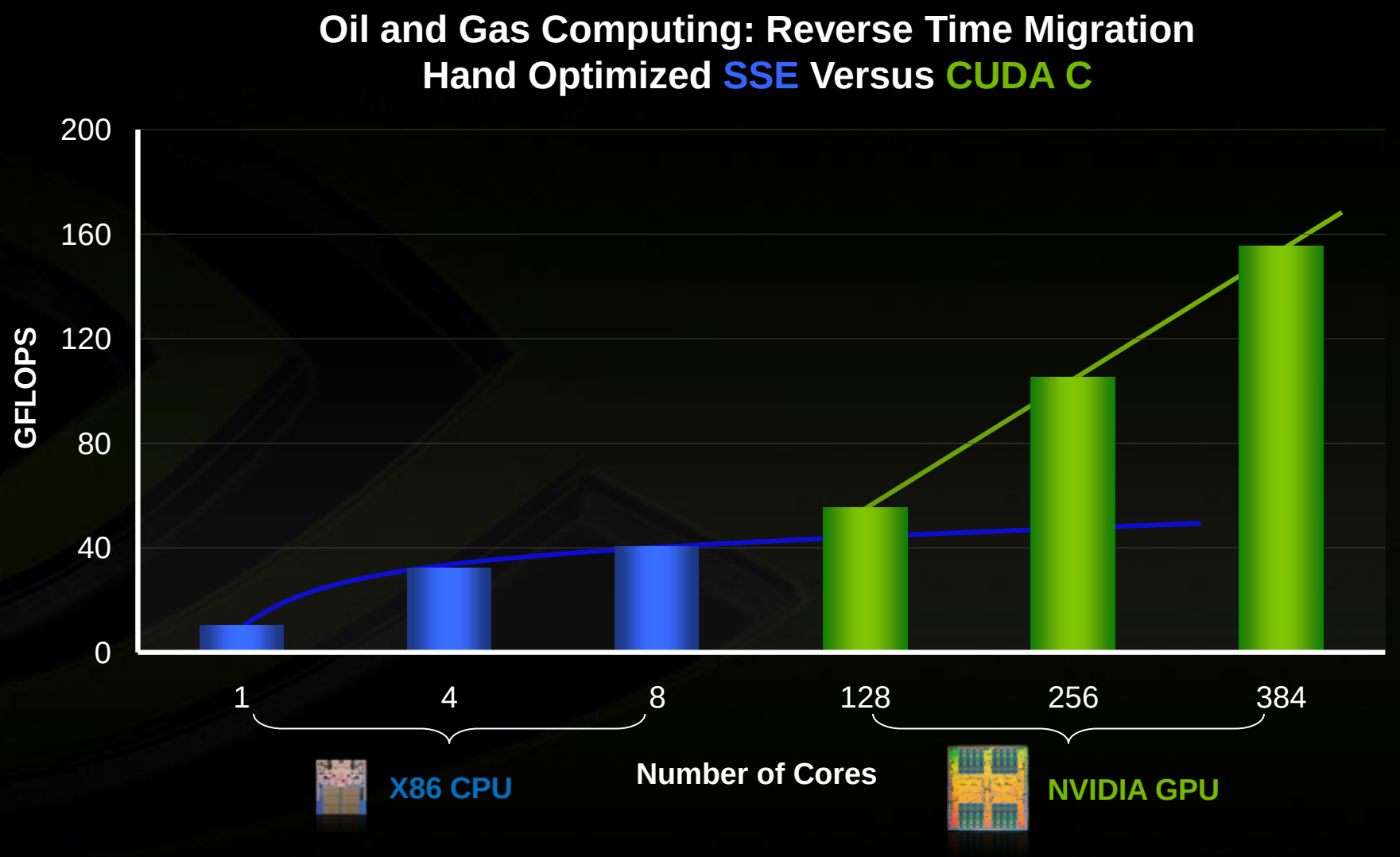
Astrophysics N-Body Simulation



http://progrape.jp/cs/

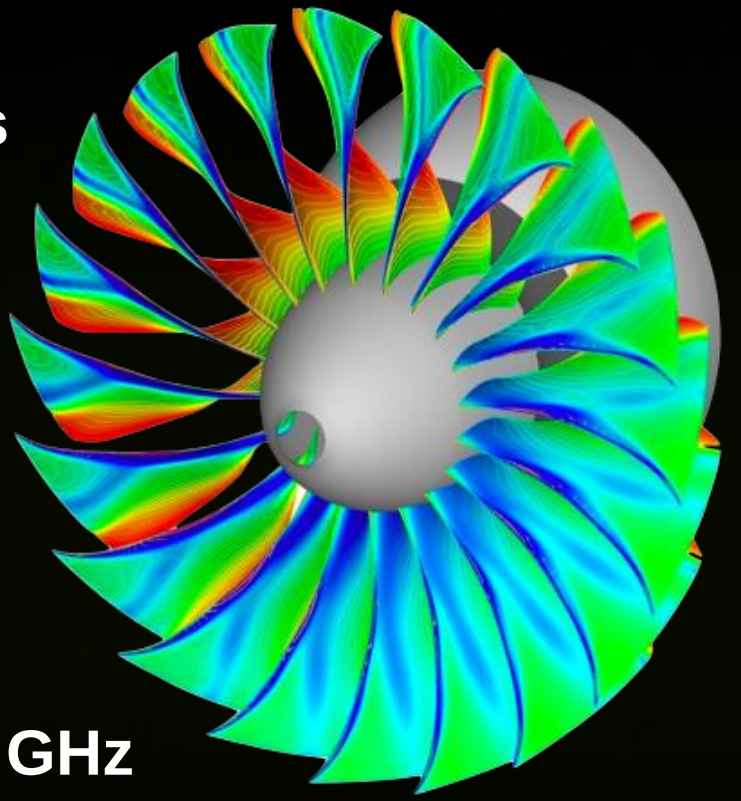
- 12+ billion body-body interactions per second
- 300 GFLOP/s+ on GeForce 8800 Ultra
  - 1-5 GFLOP/s on single-core CPU
  - Faster than custom GRAPE-6Af n-body computer
- http://www.astrogpu.org/

Linear Scaling with Multiple GPUs



Cambridge University Turbomachinery CFD

- Partial differential equations on structured grids
- NVIDIA GPUs enable faster design cycles and new science
- Up to 10 million points on a single GPU
- Scales to many GPUs using MPI
- 10x – 20x speedup Intel Core2 Quad 2.33 GHz



UNIVERSITY OF CAMBRIDGE

WHITTLE LABORATORY

CFD analysis of a jet engine fan, courtesy of Vicente Jerez Fidalgo, Whittle Lab

Slide courtesy of Tobias Brandvik and Graham Pullan, Whittle Lab