Applied Parallel Computing LLC
http://parallel-computing.pro

Оптимизации на основе архитектуры и управление потоками данных

Алексей Ивахненко

4 марта, 2017

Обзор архитектур

Программная модель CUDA:

- Архитектура определяется по CC
- Определяет поддерживаемые аппаратные возможности

Обзор архитектур

Программная модель CUDA:

- Архитектура определяется по CC
- Определяет поддерживаемые аппаратные возможности
- Процесс компиляции
  - Бинарный код vs. PTX

```
nvcc src.cu
-gencode arch=compute_20,code=sm_20
-gencode arch=compute_30,code=sm_30
```

CC	Architecture	Products Ex.
1.0	Tesla	Tesla C870, Tesla D870
1.3		Tesla C1060
2.0	Fermi	Tesla C2075, Tesla C2050
2.1		GeForce GT 720M
3.0	Kepler	Tesla K10
3.5		Tesla K40, Tesla K20x/K20
3.7		Tesla K80
5.0	Maxwell	GeForce GTX 850M
5.2		Tesla M40, Tesla M60
6.0	Pascal	Tesla P100
6.1		GeForce GTX 1080, NVIDIA Titan X

Обзор архитектур

Программная модель CUDA:

- Архитектура определяется по CC
- Определяет поддерживаемые аппаратные возможности
- Процесс компиляции
  - Бинарный код vs. PTX

```
nvcc src.cu
-gencode arch=compute_20,code=sm_20
-gencode arch=compute_30,code=sm_30
```

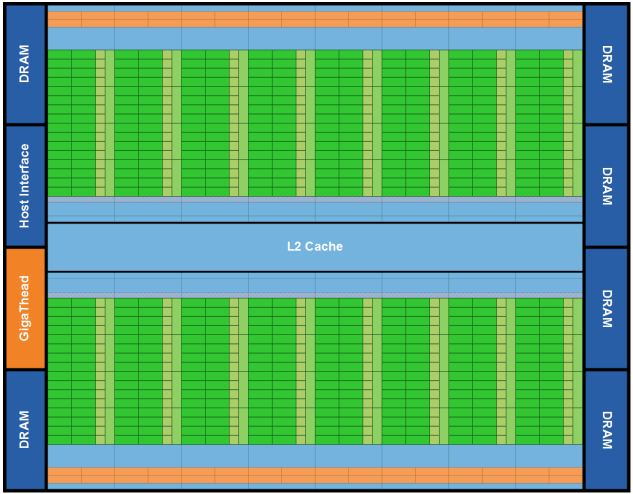
CC	Architecture	Products Ex.
1.0	Tesla	Tesla C870, Tesla D870
1.3		Tesla C1060
2.0	Fermi	Tesla C2075, Tesla C2050
2.1		GeForce GT 720M
3.0	Kepler	Tesla K10
3.5		Tesla K40, Tesla K20x/K20
3.7		Tesla K80
5.0	Maxwell	GeForce GTX 850M
5.2		Tesla M40, Tesla M60
6.0	Pascal	Tesla P100
6.1		GeForce GTX 1080, NVIDIA Titan X

CUDA compute capability table: https://developer.nvidia.com/cuda-gpus

Архитектура Fermi

Спецификации Fermi:

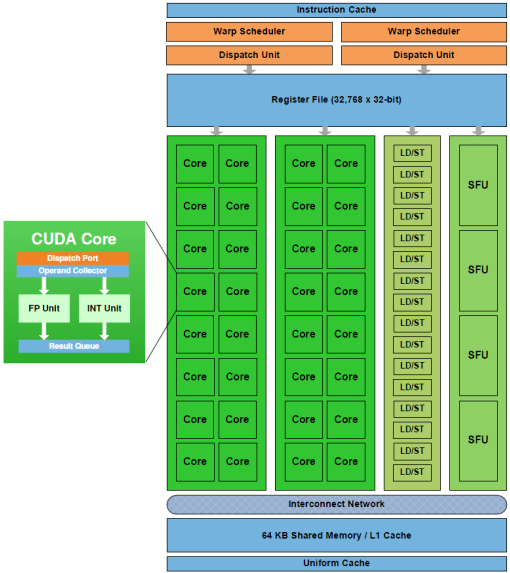
- 16 SM
- Пиковая производительность: 1.5 TFlops
  2 DP GFLOPS / W
- До 6GB памяти GDDR5
- Пропускная способность DRAM: 192GB/s
- 768 KB L2 кэш
- PCI Express 2 (8 GB/s)



Архитектура Fermi

Спецификации Fermi SM:

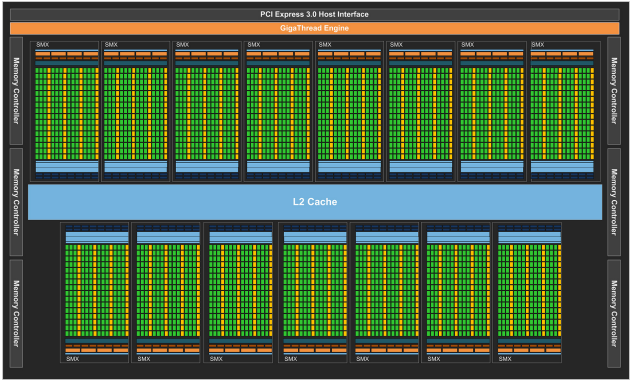
- 32 ядра CUDA
- 16 LD/ST юнита
- 4 SFU
- Двойной планировщик варпов
- 64 KB L1 кэш/ разделяемая память
- 32768 32-битных регистра



Архитектура Kepler

Спецификации Kepler:

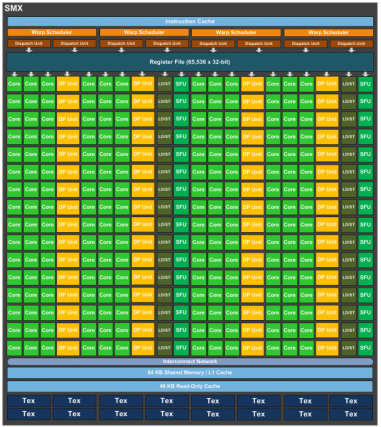
- 15 SMX
- 6 DP GFLOPS / W
- 1536 kB L2 кэш (GK110)
- PCI Express 3 (15 GB/s)
- Hyper Q - 32 аппаратных очереди
  GPU Direct - прямое копирование
  Динамический параллелизм – вложенные вызовы ядер



Архитектура Kepler

Спецификации Kepler SMX:

- 192 ядра CUDA
- 64 DP юнита
- 32 SFU
- Четыре планировщика варпов
- 64 KB L1 кэш/ разделяемая память



Архитектура Maxwell

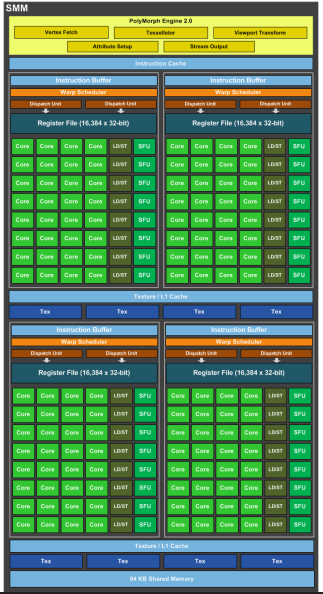
Спецификации Maxwell:

- 16 SMM
- 12 DP GFLOPS / W
- PCI Express 3 (15 GB/s)
- 2048 kB L2 кэш (GM204)
- Больше активных блоков на SMM (32)
- Атомарные операции в разделяемой памяти
- Сниженная латентность инструкций

Архитектура Maxwell

Спецификаци Maxwell SMM:

- 128 ядра CUDA
- Выделенные 96 KB разделяемой памяти
- 64 KB L1 кэш/ текстурный кэш
- Четрые планировщика варпов



Maxwell: Атомарные операции в разделяемой памяти

```
__shared__ int l_n;
//...
atomicAdd(&l_n, 1);
```

Kepler (nvcc -arch=sm_35 ...):

```
/*00c8*/    LDSLK P0, R2, [RZ];
/*00d0*/    @P0 IADD R2, R2, 0x1;
/*00d8*/    @P0 STSCUL P1, [RZ], R2;
/*00e0*/    @!P1 BRA 0xc0;
```

Maxwell (nvcc -arch=sm_50 ...):

```
/*00f0*/    @!P0 ATOMS.ADD RZ, [RZ], R4;
```

Архитектура Pascal

Спецификации Pascal SMM:

- 56 SMs
- 4096-битный интерфейс памяти HBM2
- 4096 KB L2 кэш
- Поддержка NVLink
- 16-nm FinFET
- 15,3 миллиардов транзисторов
- 5,3 TFLOPS of double precision
- 10,6 TFLOPS of single precision
- 21,2 TFLOPS of half-precision

Архитектура Pascal

Спецификации Pascal SMM:

- До 3840 ядер CUDA
- Размер регистрового файла: 14336 KB/GPU

Архитектура

	Fermi	Kepler GK110	Maxwell GM100	Maxwell GM200	Pascal P100
Streaming Multiprocessors *	16	15	16	24	56
PCI-e Version	v2	v3	v3	v3	v3, NVLink
L2 cache, KB *	768	1536	2048	3072	4096
CUDA Cores (per SM)	32	192	128	128	64
LD/ST Units (per SM)	16	32	32	32	16
SFUs (per SM)	4	32	32	32	16
Warp schedulers (per SM)	2	4	4	4	2
Shared Memory (per SM), KB	16/48	16/32/48	96	96	64
Registers (per SM)	32K	65K	65K	65K	65K
Active Blocks per SM	8	16	32	32	32
Memory Bandwidth, GB/s	144	250	336	288	720

PCI-e bandwidth: v2 - 8 GB/s, v3 - 15.75 GB/s

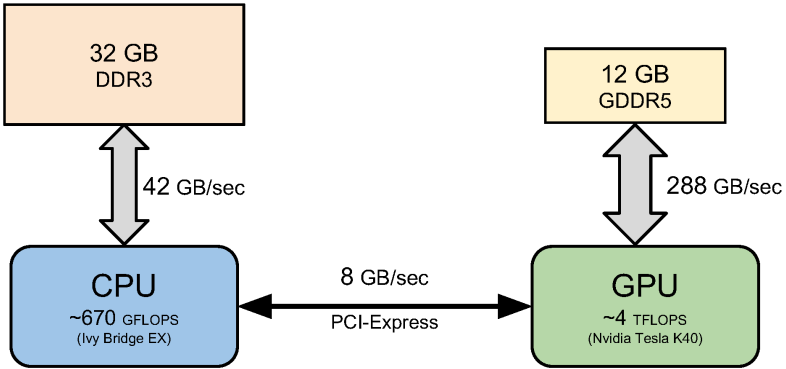
Note: (*) parameters are specific amongst different GPU series and models.

Эффективная передача данных

- Шина расширения PCI-E
- Соединения через PCI-E
  - Pageable и pinned память
  - Явные и скрытые передачи данных
  - Peer to peer & GPU Direct
  - Асинхронная передача данных
- Пример работы с UVM
- CUDA streams

Мотивация

- Влияние **перемещения данных** на общую производительность приложений



Когда происходят передачи данных GPU $\leftrightarrow$ CPU?

Когда происходят передачи данных GPU $\leftrightarrow$ CPU?

- При передаче **входных/выходных массивов**

Когда происходят передачи данных GPU $\leftrightarrow$ CPU?

- При передаче **входных/выходных массивов**

Когда еще??

Когда происходят передачи данных GPU $\leftrightarrow$ CPU?

- При передаче **входных/выходных массивов**

Когда еще??

- Загрузка **бинарного кода ядра** (неявно, под управлением драйвера)

Когда происходят передачи данных GPU $\leftrightarrow$ CPU?

- При передаче **входных/выходных массивов**

Когда еще??

- Загрузка **бинарного кода ядра** (неявно, под управлением драйвера)
- Загрузка **аргументов ядра** (передаются в константную память GPU перед запуском ядра, неявно, под управлением драйвера)

Когда происходят передачи данных GPU $\leftrightarrow$ CPU?

- При передаче **входных/выходных массивов**

Когда еще??

- Загрузка **бинарного кода ядра** (неявно, под управлением драйвера)
- Загрузка **аргументов ядра** (передаются в константную память GPU перед запуском ядра, неявно, под управлением драйвера)
- Передача результирующего значения на хост, например результат редукции (`__global__` функции всегда `void`)

Когда происходят передачи данных GPU $\leftrightarrow$ CPU?

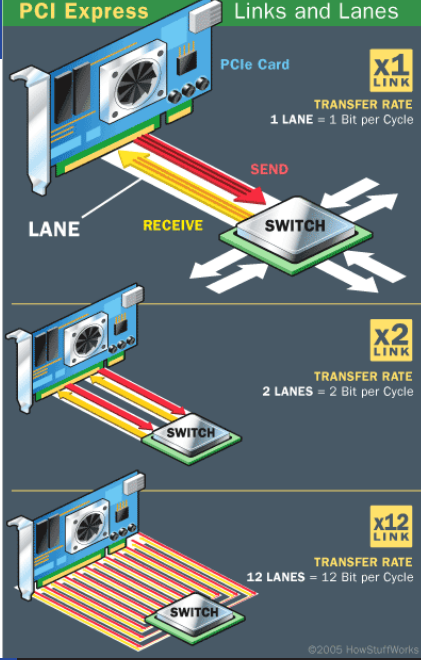
- При передаче **входных/выходных массивов**

Когда еще??

- Загрузка **бинарного кода ядра** (неявно, под управлением драйвера)
- Загрузка **аргументов ядра** (передаются в константную память GPU перед запуском ядра, неявно, под управлением драйвера)
- Передача результирующего значения на хост, например результат редукции (`__global__` функции всегда `void`)
- Инициализация `__device__` переменных

Обзор PCI Express

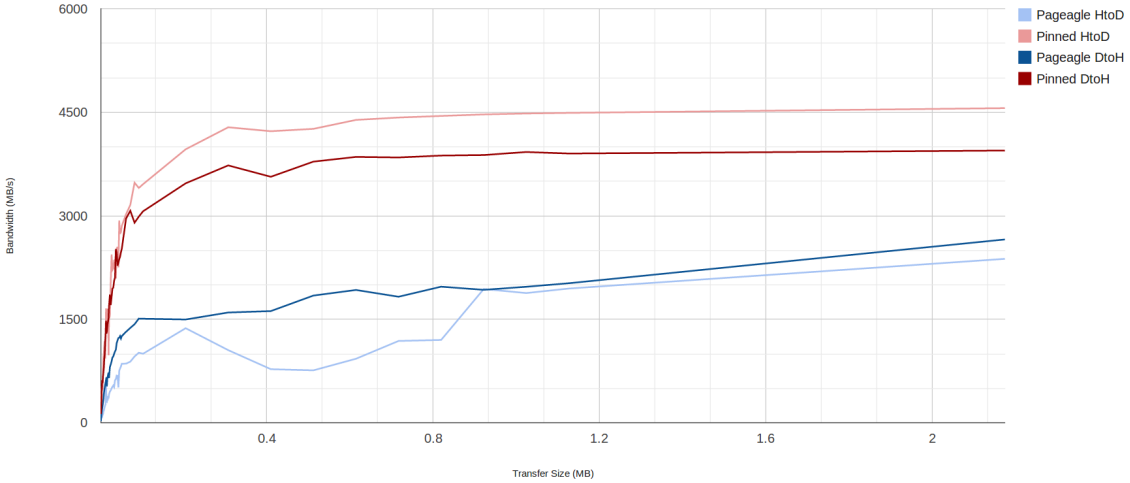
- Шина расширения компьютера
- Point-to-point соединение
- Разделение линий
- Одна линия (x1)
  - 500 MB/s на линию lane (PCI-e v2)
- Множество линий (x2, x4, x8, x16, x32)
  - 8 GB/s для шини в 16 линий



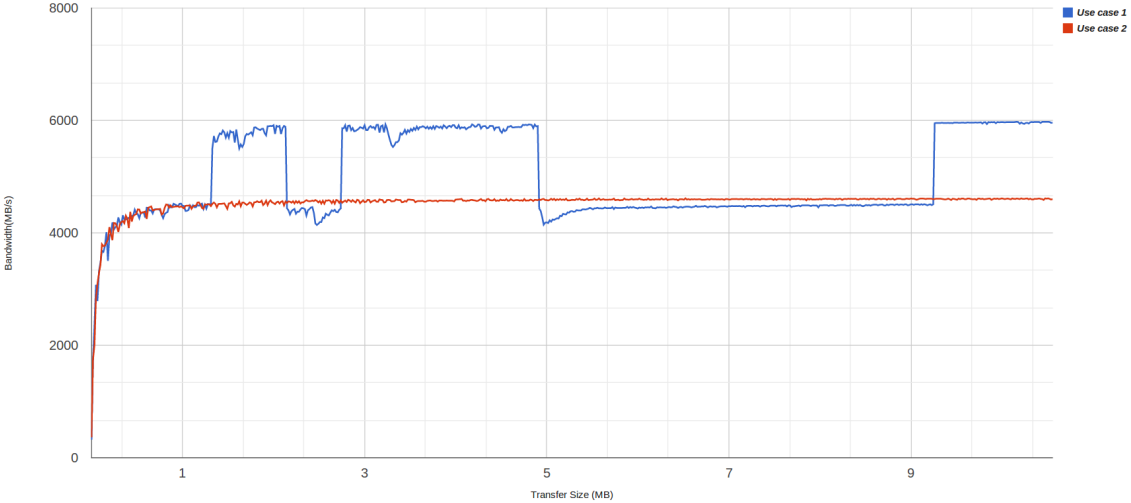
Поколения PCI-Express

PCI Express version	Per Lane Bandwidth	x16 Bandwidth
1.0 (2003)	250 MB/s	4 GB/s
2.0 (2007)	500 MB/s	8 GB/s
3.0 (2010)	984 MB/s	15 GB/s
4.0 (2017)	1969 MB/s	31 GB/s
5.0 (?)	3125-4000 MB/s	50-64 GB/s

Тест пропускной способности PCI-E



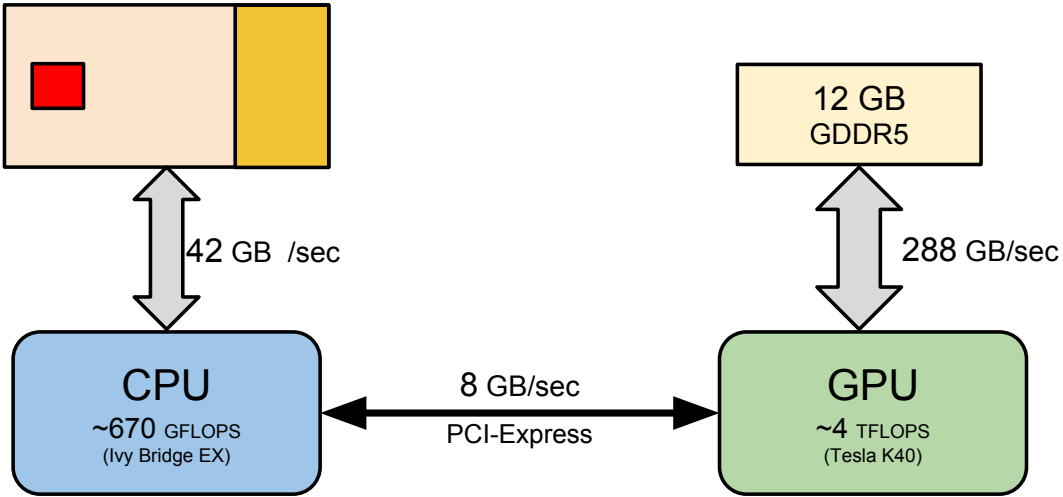
Линии PCI-E


8000
6000
4000
2000
0
Bandwidth(MB/s)
1
3
5
7
9
Transfer Size (MB)
Use case 1
Use case 2

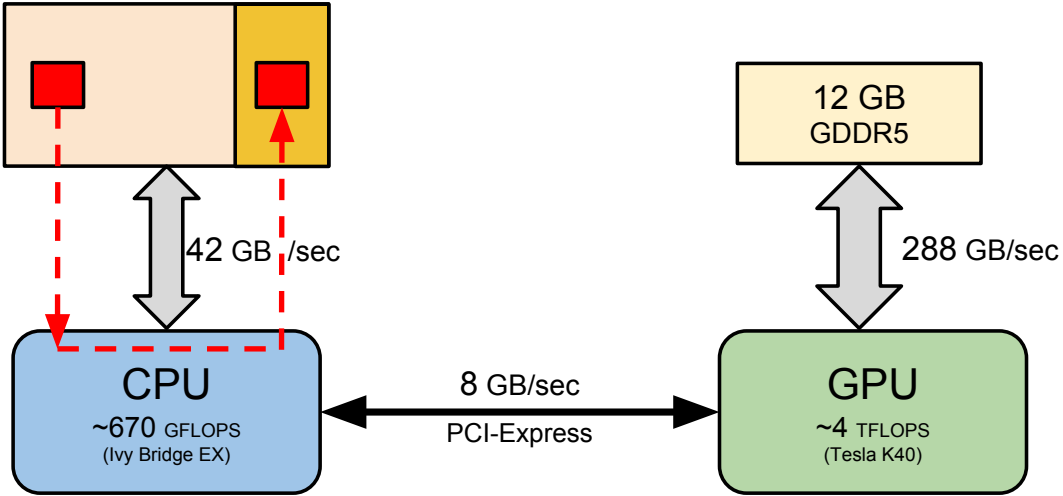
Виды передач данных в CUDA

- Pageable и pinned
- Явные и неявные (UVM)
- Peer to peer между GPU внутри одного узла)
- GPUDirect (между GPU через сетевой интерфейс)
- Синхронные и асинхронные

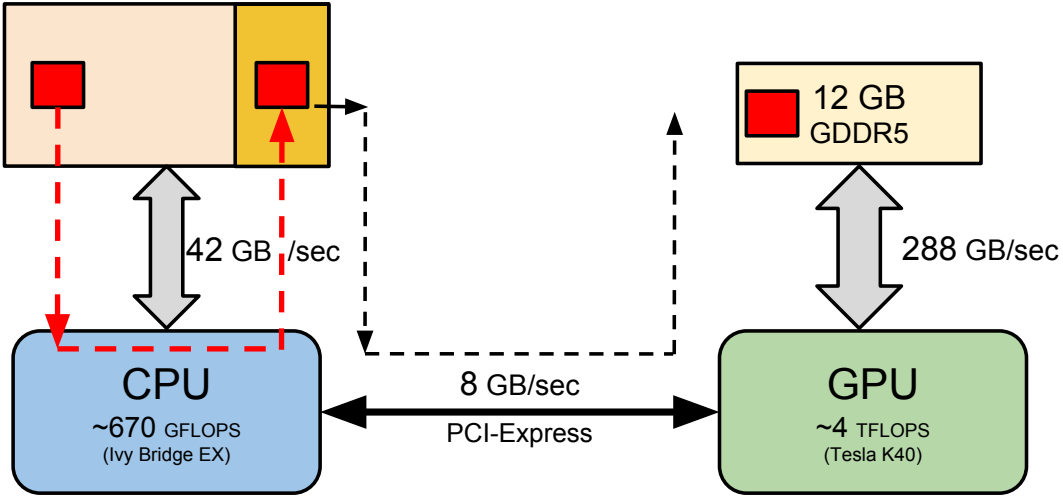
Pageable и pinned передачи данных



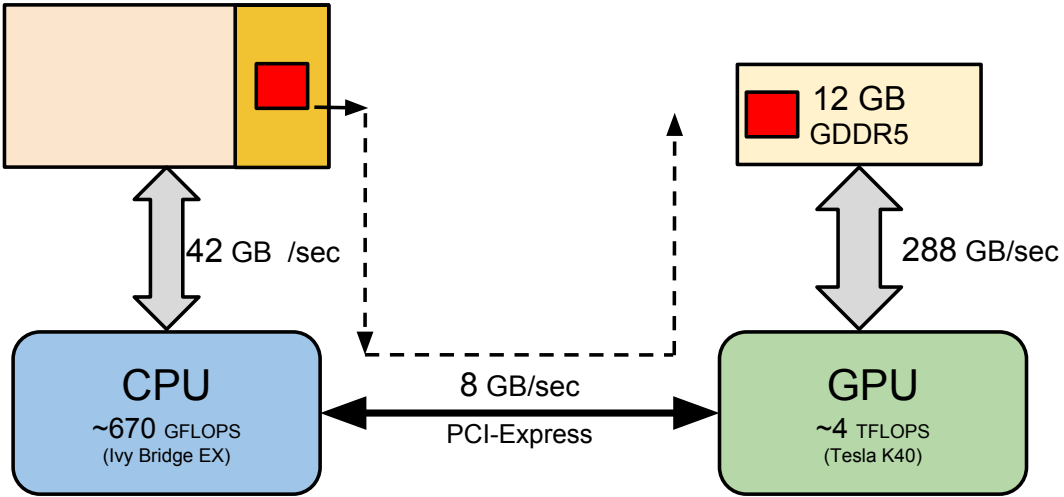
Pageable и pinned передачи данных



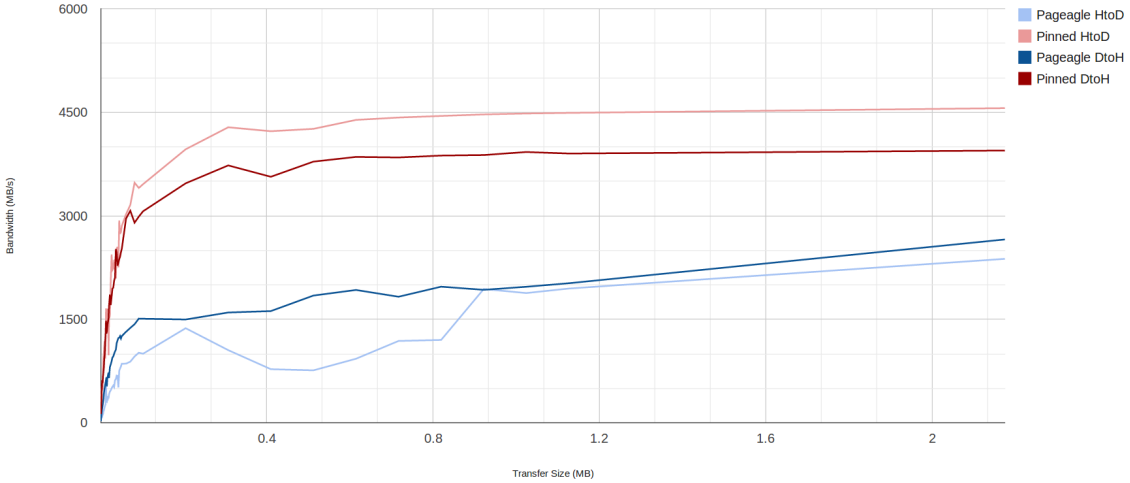
Pageable и pinned передачи данных



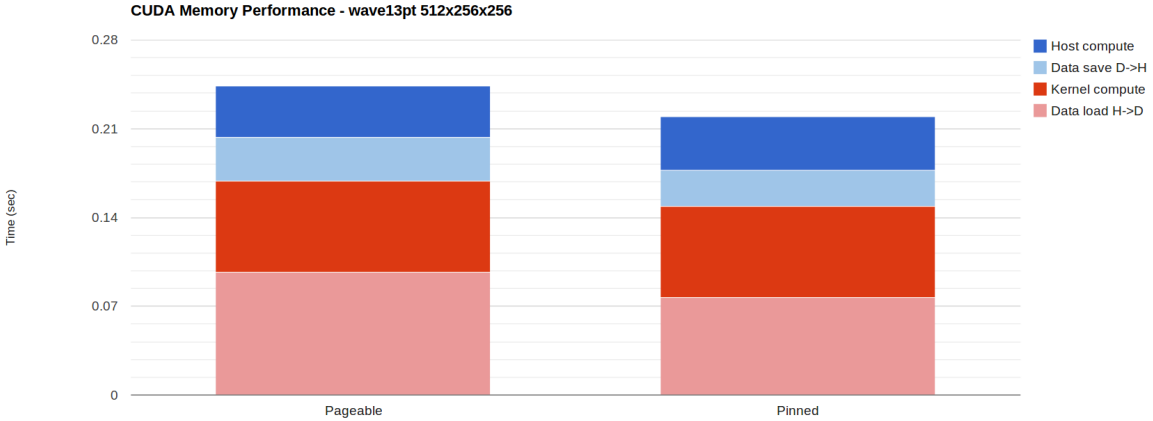
Pageable и **pinned** передачи данных



Pageable и pinned передачи данных



wave13pt – Pageable vs Pinned память



Pageable и pinned передачи данных

```
//allocate memory
w0 = (real*)malloc(szarrayb);
cudaMalloc(&w0_dev, szarrayb);

//memcopy
cudaMemcpy(w0_dev, w0, szarrayb, cudaMemcpyHostToDevice);

//kernel compute
wave13pt_d<<<...>>>(..., w0_dev, ...);

//memcopy
cudaMemcpy(w0, w0_dev, szarrayb, cudaMemcpyDeviceToHost);
```

1: Pageable

```
//allocate memory
cudaMallocHost(&w0, szarrayb);
cudaMalloc(&w0_dev, szarrayb);

//memcopy
cudaMemcpy(w0_dev, w0, szarrayb, cudaMemcpyHostToDevice);

//kernel compute
wave13pt_d<<<...>>>(..., w0_dev, ...);

//memcopy
cudaMemcpy(w0, w0_dev, szarrayb, cudaMemcpyDeviceToHost);
```

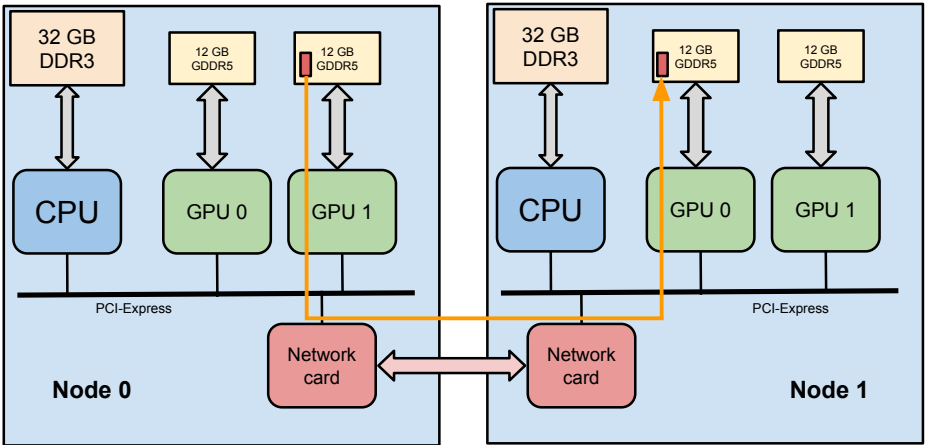
2: Pinned

Pageable и pinned передачи данных

- Pageable память – пользовательская память, требует дополнительного копирования
- Pinned память – память ядра
- Pinned память дает большую производительность (выше пропускная способность)
- Не злоупотребляйте pinned памятью – сокращает объем памяти физически доступный ОС

GPU direct

- Исключает ненужные копирования через CPU с его низкой пропускной способностью и дополнительной латентностью, используя Remote Direct Memory Access передачи между GPU и другими устройствами PCI

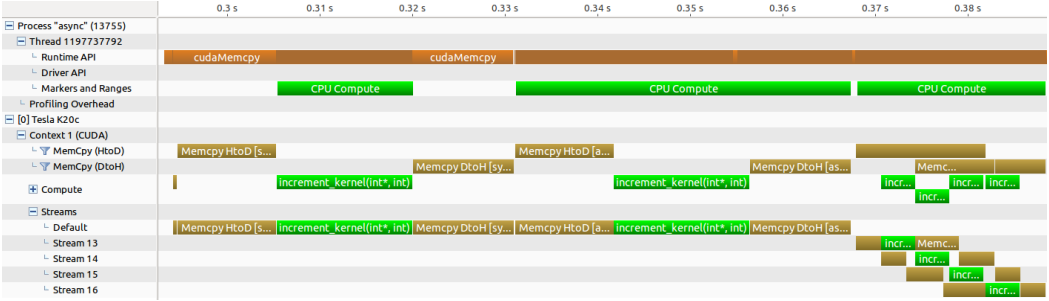


Асинхронная передача данных

- `cudaMemcpy()` блокирует исполнение до тех пор, пока не окончится передача
- `cudaMemcpyAsync()` не блокирует исполнение CPU команд, можно использовать CPU для каких-нибудь полезных вычислений
- Асинхронные копирования имеют два дополнительных требования:
  - Pinned память
  - Stream id

Асинхронная передача данных

- Для чего использовать?
  - Перекрытие CPU и GPU вычислений
  - Перекрытие вычислений и передач данных



CUDA Streams

```
// create cuda streams
cudaStream_t stream[2];
cudaStreamCreate(&stream[0]);
cudaStreamCreate(&stream[1]);

// asynchronously copy data, run kernel and copy back
cudaMemcpyAsync(d_a, a, nbytes/4, cudaMemcpyHostToDevice, stream[0]);
kernel<<<blocks, threads, 0, stream[0]>>>(d_a, value);
cudaMemcpyAsync(a, d_a, nbytes/4, cudaMemcpyDeviceToHost, stream[0]);

cudaMemcpyAsync(d_a+offset, a+offset, nbytes/4, cudaMemcpyHostToDevice, stream[1]);
kernel<<<blocks, threads, 0, stream[1]>>>(d_a+offset, value);
cudaMemcpyAsync(a+offset, d_a+offset, nbytes/4, cudaMemcpyDeviceToHost, stream[1]);

// run some CPU code
f();

// wait for GPU operations to finish and destroy streams
cudaDeviceSynchronize();
cudaStreamDestroy(stream[0]);
cudaStreamDestroy(stream[1]);
```

Синхронизация CUDA Streams

Явная:

- `cudaDeviceSynchronize()`
  - Блокирует исполнение до окончания всех CUDA вызовов
- `cudaStreamSynchronize(stream)`
  - Блокирует исполнение до окончания всех CUDA вызовов внутри заданного потока
- `cudaEventRecord(event,stream1)`, `cudaStreamWaitEvent(stream2,event)`
  - Fine-grained синхронизация

Неявная:

- Выделение Page-locked памяти
  - `cudaMallocHost`, `cudaHostAlloc`
- Выделение памяти на устройсте
  - `cudaMalloc`
- Блокирующие версии операций с памятью
  - `cudaMemcpy`, `cudaMemset`
- Неявно синхронизируют все CUDA операции

Ограничения CUDA Streams

Ограничения в параллельном исполнении:

- До 16(Fermi, <3.5) / 32(Kepler 3.5) ядра на GPU
- 1 H2D memcopy
- 1 D2H memcopy

Общие рекомендации

- PCI-E транзакции эффективны, начиная с некоторого, достаточно большого, объема передаваемых данных
- UVM упрощает программную модель, но может снизить производительность
- Лучше остлеживать, когда программа использует промежуточные буферы копирования и обходить ускорять исполнение с помощью pinned памяти и GPUDirect
- Передача данных не является целью вычислительной программы и должна как можно чаще происходить параллельно с вычислениями (асинхронно)

Оптимизация GPU кода

- Общие техники на основе знаний об архитектуре:
  - Выбор оптимального размера block & grid
  - Coalescing, AoS to SoA
  - Ветвление
- Оптимизации на основе профилирования:
  - Использование вычислений & памяти
  - Пропускная способность памяти

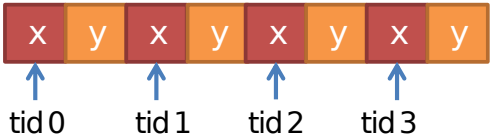
Выбор оптимального размера CUDA block & grid

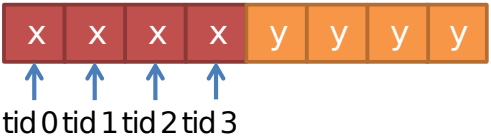
- Размер блока:
  - Начните с 128-256 нитей на блок
  - Увеличивайте или уменьшайте, в соответствии с особенностями вашей функции–ядра
  - Кратен размеру варпа (32)
- Размер сетки:
  - 1000 и больше блоков
  - Развномерное распределение работы по всему GPU
  - Код готов к исполнению на различных архитектурах GPU

Оптимизация глобальной памяти: AoS vs SoA

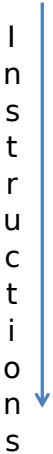
```
struct A
{
    float x;
    float y;
};
struct A myArray[n];
```

```
struct A
{
    float x [n];
    float y [n];
};
struct A myArray;
```



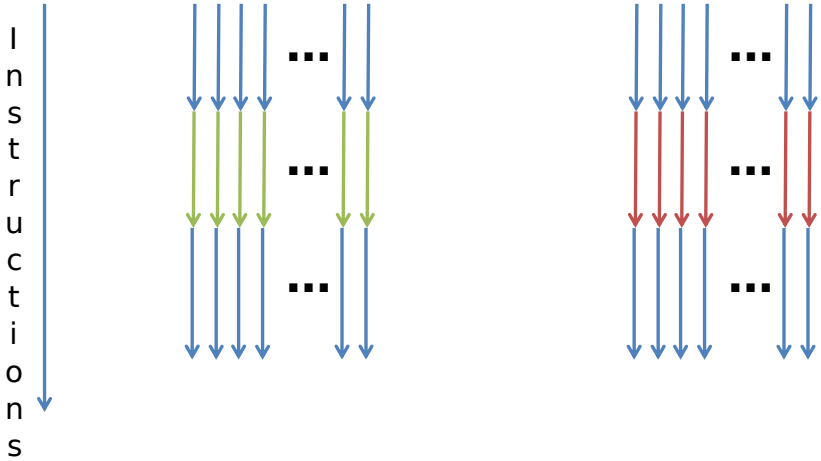


Ход исполнения

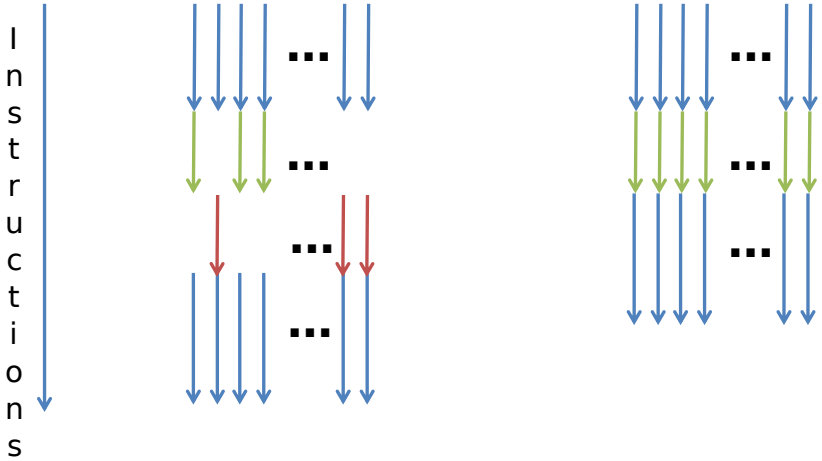


```
if ( )
{
  //then-clause
}
else
{
  //else-clause
}
```

исполнение варпами

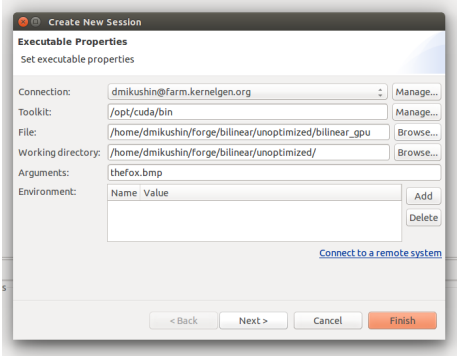


Дивергентное ветвление



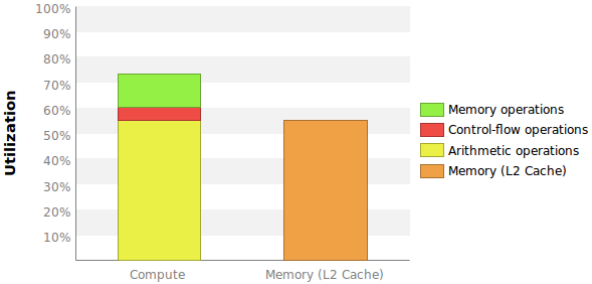
Настройки профайлера

- Подготовьте скомпилированный бинарный файл на удаленном GPU узле (`-arch=sm_35` для Tesla K40)
- Откройте NVIDIA Visual Profiler → File → New Session
- Подключитесь к серверу через меню
- Отправьте приложение на исполнение и профилирование



Профилирование: общий анализ ядра

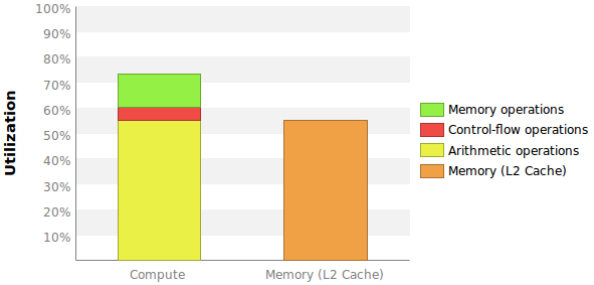
- Analysis → CUDA Application Analysis → Examine Individual Kernels → Select `bilinear` → Perform Kernel Analysis



- Профиль выглядит как-будто приложение compute-bound а не memory-bound, так ли это?

Профилирование: общий анализ ядра

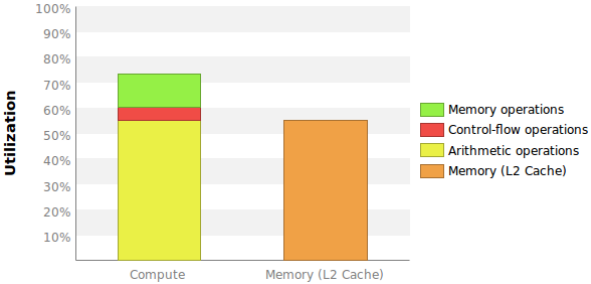
- Analysis → CUDA Application Analysis → Examine Individual Kernels → Select `bilinear` → Perform Kernel Analysis



- Профиль выглядит как-будто приложение compute-bound а не memory-bound, так ли это?
- $4 \times$ 4-байтовые чтения, $4 \times$ 1-байтовые записи **VS** 44 FLOPS $\Rightarrow$ 20:44 $\approx$ 1:2.2

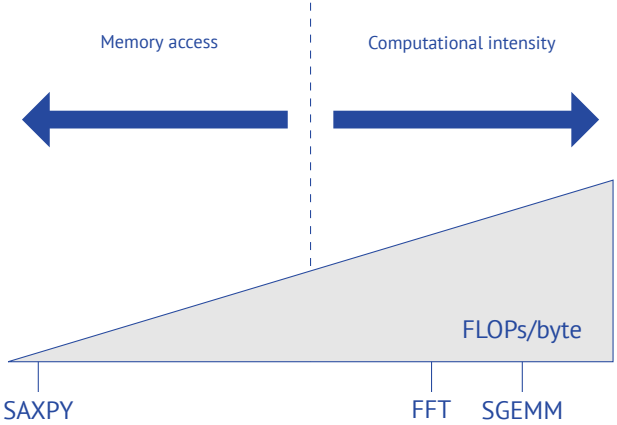
Профилирование: общий анализ ядра

- Analysis → CUDA Application Analysis → Examine Individual Kernels → Select `bilinear` → Perform Kernel Analysis



- Профиль выглядит как-будто приложение compute-bound а не memory-bound, так ли это?
- $4 \times$ 4-байтовые чтения, $4 \times$ 1-байтовые записи **VS** 44 FLOPS $\Rightarrow$ 20:44 $\approx$ 1:2.2
- Почему memory-bound приложение выглядит как compute-bound? Давайте разбираться!

Compute-bound **VS** memory-bound



Оптимальная вычислительная интенсивность для GPU (FLOPS : byte):

Tesla C1060	9.1 : 1
Tesla C2050	7.1 : 1
Tesla K20	16.9 : 1
Tesla K40	17.5 : 1
Tesla M40	23.7 : 1
Tesla P100	12.9 : 1
NVIDIA Titan X	22.8 : 1

Профилирование: compute-bound **VS** memory-bound

Как memory-bound приложение может выглядеть как compute-bound?

Профилирование: compute-bound **VS** memory-bound

Как memory-bound приложение может выглядеть как compute-bound?

- Вспомогательные вычислителдьные операции превалируют над floating-point операциями (например, слишком много integer вычислений)

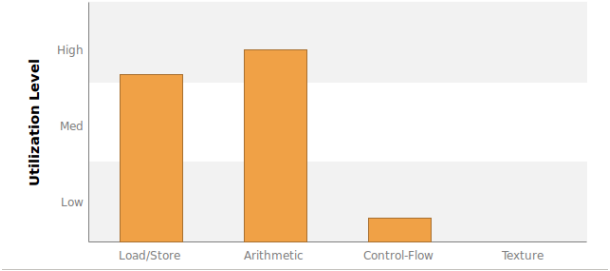
Профилирование: compute-bound **VS** memory-bound

Как memory-bound приложение может выглядеть как compute-bound?

- Вспомогательные вычислителдьные операции превалируют над floating-point операциями (например, слишком много integer вычислений)
- В таком случае, FLOPS на байт – не лучший способ определить compute/memory bounding

Профилирование: инструкции

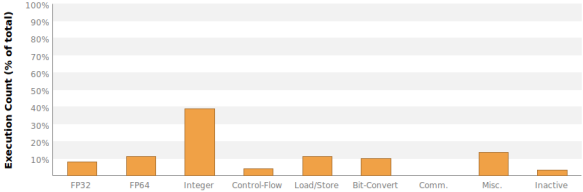
- Compute, Bandwidth, or Latency Bound $\longrightarrow$ Perform Compute Analysis



- Еще один взгляд на memory & compute показывает, что оба используются с высокой загрузкой
- Основная цель дальнейшего профилирования и оптимизации: точно ли память и вычисления используются эффективно? Можем ли мы снизить нагрузку, или оптимизировать?

Профилирование: инструкции

- Compute, Bandwidth, or Latency Bound $\longrightarrow$ Perform Compute Analysis (and scroll down to second chat)



- Большинство вычислений – Integer
- Так же есть FP64, хотя мы используем перменные float и int (почему?)

Откуда FP64

```
$ cuobjdump -sass bilinear_gpu | grep DFMA
 /*04d0*/ DFMA R8, -R2, R6, c[0x2][0x8];
 /*04d8*/ DFMA R8, R2, R8, R2;
 /*04e0*/ DFMA R10, -R8, R6, c[0x2][0x8];
 /*04f0*/ DFMA R8, R8, R10, R8;
 /*04f8*/ DFMA R12, -R2, R6, R4;
 /*0508*/ DFMA R2, R8, R12, R2;
 /*0510*/ DFMA R8, -R2, R6, R4;
 /*07b8*/ DFMA R10, -R2, R22, c[0x2][0x8];
 /*07c8*/ DFMA R8, R10, R22, R22;
 /*07e0*/ DFMA R8, R6, R8, R8;
 /*07e8*/ DFMA R10, -R2, R22, R4;
 /*07f0*/ DFMA R6, -R2, R8, c[0x2][0x8];
 /*07f8*/ DFMA R22, R10, R8, R22;
 /*0808*/ DFMA R8, R6, R8, R8;
 /*0810*/ DFMA R2, -R2, R22, R4;
 /*0818*/ DFMA R4, R2, R8, R22;
 /*08f0*/ DFMA.RZ R10, R2, R8, R22;
 /*0908*/ DFMA.RP R6, R2, R8, R22;
 /*0910*/ DFMA.RM R8, R2, R8, R22;
$
```

```
float x = width * (i - 0.5) / (float)(2 * width);
float y = height * (j - 0.5) / (float)(2 * height);
```

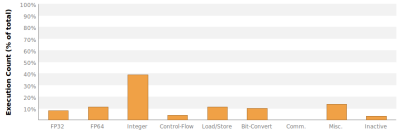
⇓

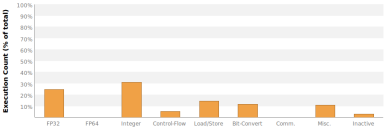
```
float x = width * (i - 0.5f) / (float)(2 * width);
float y = height * (j - 0.5f) / (float)(2 * height);
```

перекомпилируем:

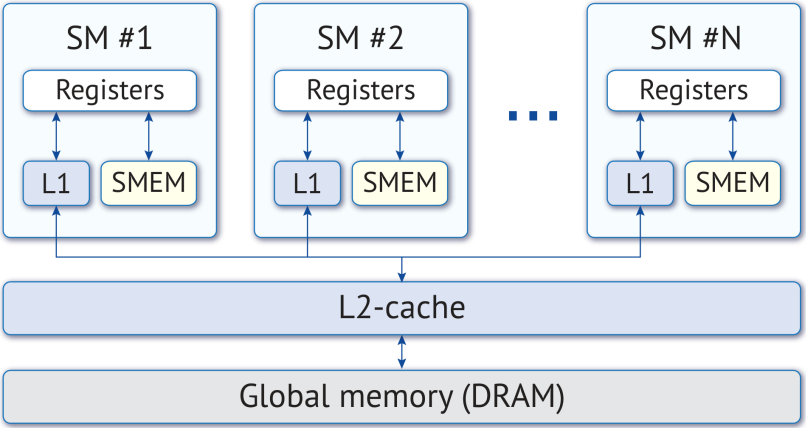
```
$ cuobjdump -sass bilinear_gpu | grep DFMA
$
```

FP64 пропали! Следует избегать FP64 на потребительских картах (например, sm_30), т.к. они имеют пропускную способность FP64 в 8× раз меньше чем Tesla GPU





Иерархия памяти GPU



Профилирование: кэши

- Compute, Bandwidth, or Latency Bound $\longrightarrow$ Perform Memory Bandwidth Analysis (and scroll down)

L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	69659312	278.369 GB/s	
Global Stores	21764592	104.374 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	91423904	382.743 GB/s	Idle Low Medium High Max
L2 Cache			
L1 Reads	139250652	278.369 GB/s	
L1 Writes	52211716	104.374 GB/s	
Texture Reads	0	0 B/s	
Atomic	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Total	191462368	382.743 GB/s	Idle Low Medium High Max

- Современные GPU имеют кэши размером: 32 KB L1 на мультипроцессор, 1536 KB L2 на GPU Tesla K40
- L1 имеет большую пропускную способность,но меньший размер $\Rightarrow$ ограничено L2

Профилирование: память

- Compute, Bandwidth, or Latency Bound $\longrightarrow$ Perform Memory Bandwidth Analysis (and scroll down)

Device Memory

Reads	2900891	5.799 GB/s	
Writes	31222936	62.416 GB/s	
Total	34123827	68.215 GB/s	Idle · Low · Medium · High · Max

- Для memory-bound приложений, использование глобальной памяти должно быть как можно выше. Если нет:
  - Устройство может быть недозагружено (слишком маленькая задача)
  - Плохое взаимное расположение инструкций вычисления и чтения/записи
- Для compute-bound приложений низкая загрузка глобальной памяти не является проблемой, снижение может даже ускорить программу

Оптимизации эффективности работы памяти

- Исходный GPU код загружает один `RGBApixel` с помощью `LD.U8` инструкции
- `RGBApixel` состоит из 4 байт, соседние нити используют соседние значения ⇒ coalescing возможен
- Вместо этого, компилятор генерирует load/store 1-байтных компонентов ⇒ coalescing не используется
- Интенсивная нагрузка на L1/L2 кэш минимизирует потери производительности, но некоторая просадка все же наблюдается

⇒ Оптимизация памяти: переписываем функцию-ядро

Оптимизации эффективности работы памяти

Векторные чтения/записи - вариант №1: inline assembly

```
static __device__ __inline__ RGBApixel __ld(const RGBApixel *ptr)
{
    RGBApixel ret;
    int4 tmp;
    asm volatile ("ld.global.v4.u8 {%0,%1,%2,%3}, [%4];"  : "=r"(tmp.x), "=r"(tmp.y), "=r"(tmp.z), "=r"(tmp.w) : __RET_PTR (ptr));
    ret.Red = (char)tmp.x; ret.Green = (char)tmp.y; ret.Blue = (char)tmp.z; ret.Alpha = (char)tmp.w; return ret;
}

static __device__ __inline__ void __stcs(const RGBApixel *ptr, const RGBApixel& val)
{
    int4 tmp;
    tmp.x = val.Red; tmp.y = val.Green; tmp.z = val.Blue; tmp.w = val.Alpha;
    asm volatile ("st.global.cs.v4.u8 [%0], {%1,%2,%3,%4};"  : : __RET_PTR (ptr), "r"(tmp.x), "r"(tmp.y), "r"(tmp.z), "r"(tmp.w));
}
```

```
// Load the four neighboring pixels
const RGBApixel p1 = __ld(&p0[0 + 0 * stride]);
const RGBApixel p2 = __ld(&p0[1 + 0 * stride]);
const RGBApixel p3 = __ld(&p0[0 + 1 * stride]);
const RGBApixel p4 = __ld(&p0[1 + 1 * stride]);
```

```
RGBApixel ret;
...
__stcs(output, ret);
```

Оптимизации эффективности работы памяти

Векторные чтения/записи - вариант №2: union

```
union RGBApixel_
{
    RGBApixel p;
    int i;
};
```

```
const RGBApixel* p0 = &pixels[0] + px + py * stride; // pointer to first pixel

// Load the four neighboring pixels
RGBApixel_ p1_; p1_.i = *(int*)&p0[0 + 0 * stride];
RGBApixel_ p2_; p2_.i = *(int*)&p0[1 + 0 * stride];
RGBApixel_ p3_; p3_.i = *(int*)&p0[0 + 1 * stride];
RGBApixel_ p4_; p4_.i = *(int*)&p0[1 + 1 * stride];

const RGBApixel& p1 = p1_.p;
const RGBApixel& p2 = p2_.p;
const RGBApixel& p3 = p3_.p;
const RGBApixel& p4 = p4_.p;
```

```
RGBApixel_* output_ = (RGBApixel_*)output;
output_->p = ret;
```

Результат оптимизации

- Выровненный доступ к памяти ⇒ меньше нагрузки на кэш:

L1/Shared Memory		
Local Loads	0	0 B/s
Local Stores	0	0 B/s
Shared Loads	0	0 B/s
Shared Stores	0	0 B/s
Global Loads	17414828	81.66 GB/s
Global Stores	5441148	30.618 GB/s
Atomic	0	0 B/s
L1/Shared Total	22855976	112.278 GB/s

L2 Cache		
L1 Reads	34812663	81.66 GB/s
L1 Writes	13052928	30.618 GB/s
Texture Reads	0	0 B/s
Atomic	0	0 B/s
Noncoherent Reads	0	0 B/s
Total	47865591	112.278 GB/s

Idle Low Medium High Max

- Выровненный доступ к памяти ⇒ меньше операций доступа к глобальной памяти:

Device Memory		
Reads	2900859	6.805 GB/s
Writes	12047537	28.26 GB/s
Total	14948396	35.064 GB/s

Idle Low Medium High Max

Результаты тестов производительности (Tesla K40)

version	time, sec
GPU unoptimized	0.016322
GPU no FP64	0.017247
GPU vector LD/ST	0.013728
GPU texture	0.014012
CPU OpenMP	1.128108

Результаты тестов производительности (Tesla K40)

version	time, sec
GPU unoptimized	0.016322
GPU no FP64	0.017247
GPU vector LD/ST	0.013728
GPU texture	0.014012
CPU OpenMP	1.128108

Q: Почему версия без FP64 хуже чем с FP64?

Результаты тестов производительности (Tesla K40)

version	time, sec
GPU unoptimized	0.016322
GPU no FP64	0.017247
GPU vector LD/ST	0.013728
GPU texture	0.014012
CPU OpenMP	1.128108

Q: Почему версия без FP64 хуже чем с FP64?

A: На Tesla GPU достаточно FP64 юнитов. На потребительских картах ситуация поменяется: недостаток FP64 юнитов замедлит программу