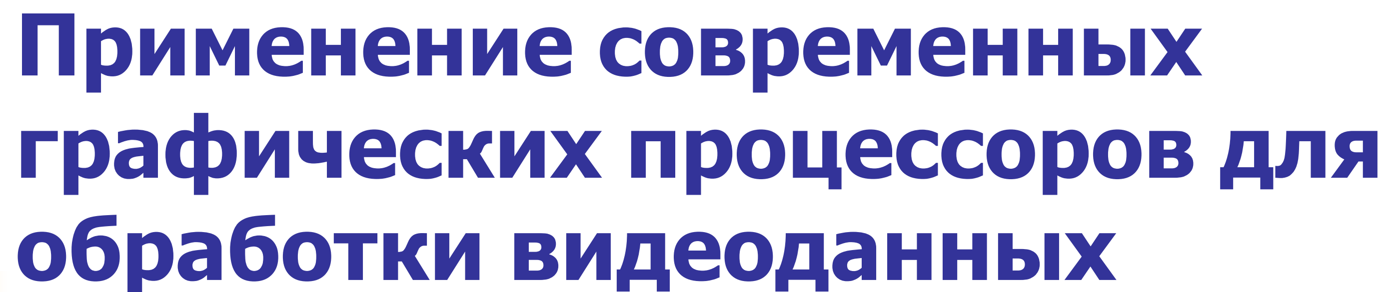
GRAPHICS & MEDIA LAB
VIDEO GROUP



Применение современных графических процессоров для обработки видеоданных

Илья Цветков

Video Group
CS MSU Graphics & Media Lab

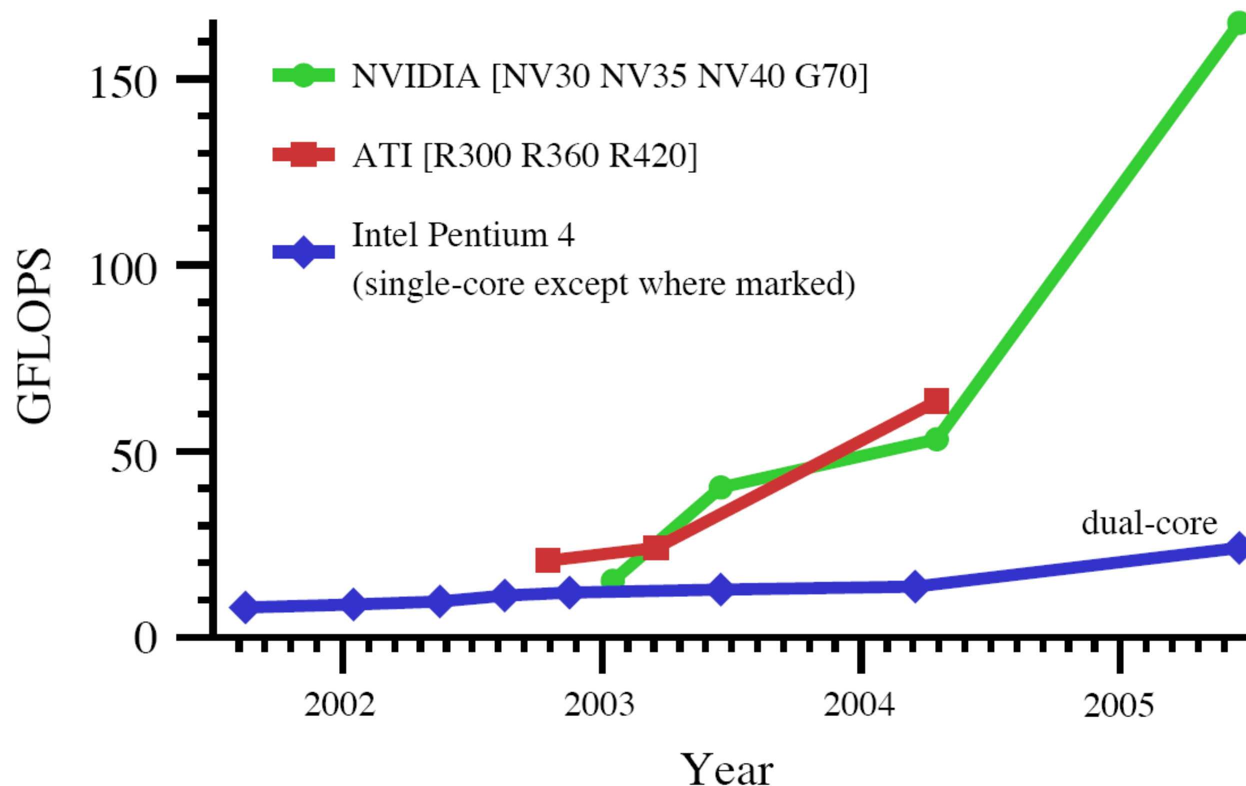
Содержание доклада

- Введение
- Устройство современного графического процессора
  - Графический конвейер
  - Вершинные и пиксельные шейдеры
- Базовые концепции программирования на GPU
  - Оптимизация
- Алгоритмы обработки видео на GPU
  - Компенсация движения
  - Фильтрация
- Результаты

Почему GPU?

- Пиковая производительность GPU последнего поколения — около 600 GFLOPS
- Скорость обмена данными с видеопамятью составляет 60 Гбайт/с
- Даже low-end видеокарты позволяют обрабатывать большие изображения быстрее чем CPU.
- Очень широко распространены среди обычных пользователей компьютеров в отличие от FPGA или DSP.

Рост производительности GPU



J. D. Owens, D. Luebke et al., “A Survey of General-Purpose Computation on Graphics Hardware,” *EUROGRAPHICS 2005, State of the Art reports*, 2005

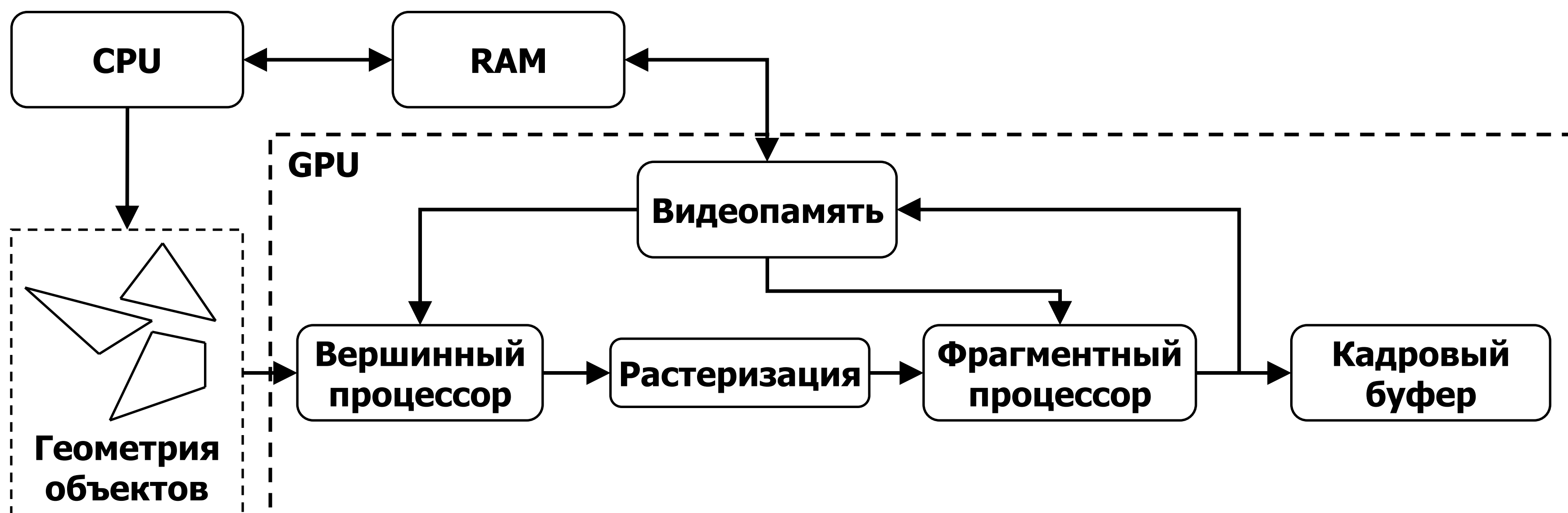
Почему GPU так быстро работают

- CPU
  - Оптимизирован для исполнения последовательных операций
  - Ориентирован на обработку скалярных данных
- GPU
  - Графическому процессору необходимо производить однотипные операции над большими объемами данных
  - Изначально проектируется как векторный процессор

Устройство современного графического процессора

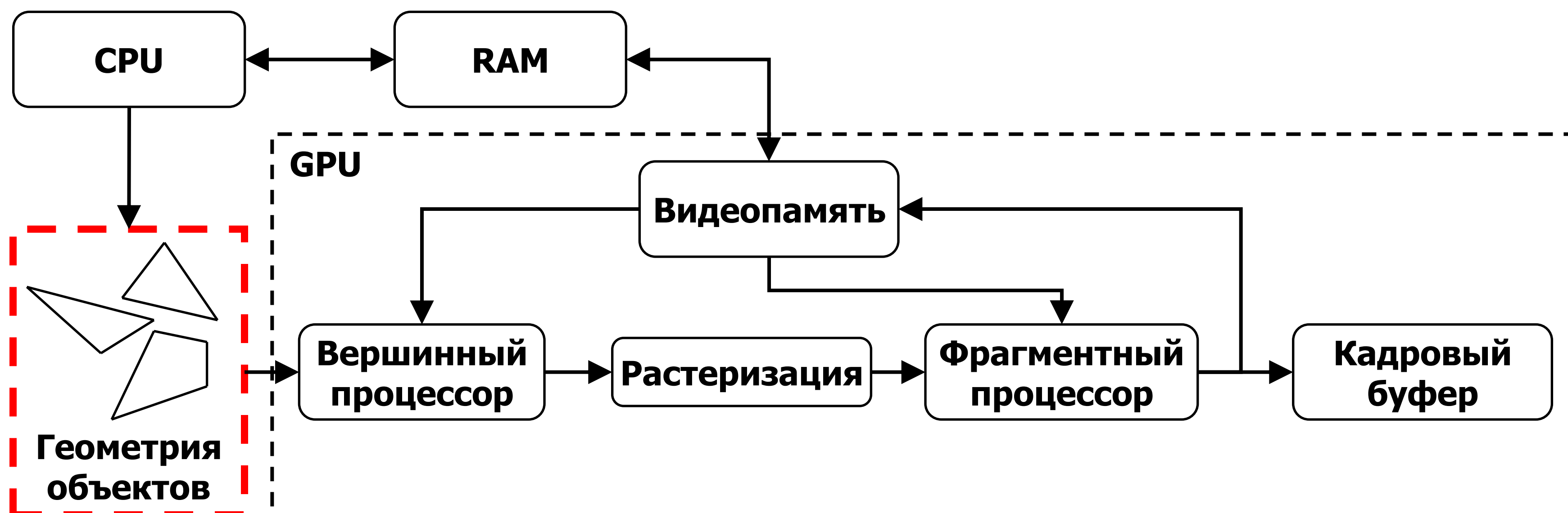
- Графический конвейер
- Программируемый вершинный процессор
- Растеризация
- Текстуры
- Программируемый фрагментный процессор
- Рендеринг в текстуру

Графический конвейер



- Каждый этап конвейера может быть настроен с помощью графического API (например OpenGL или Direct3D)
- Некоторые этапы полностью программируемы, другие обладают фиксированной функциональностью

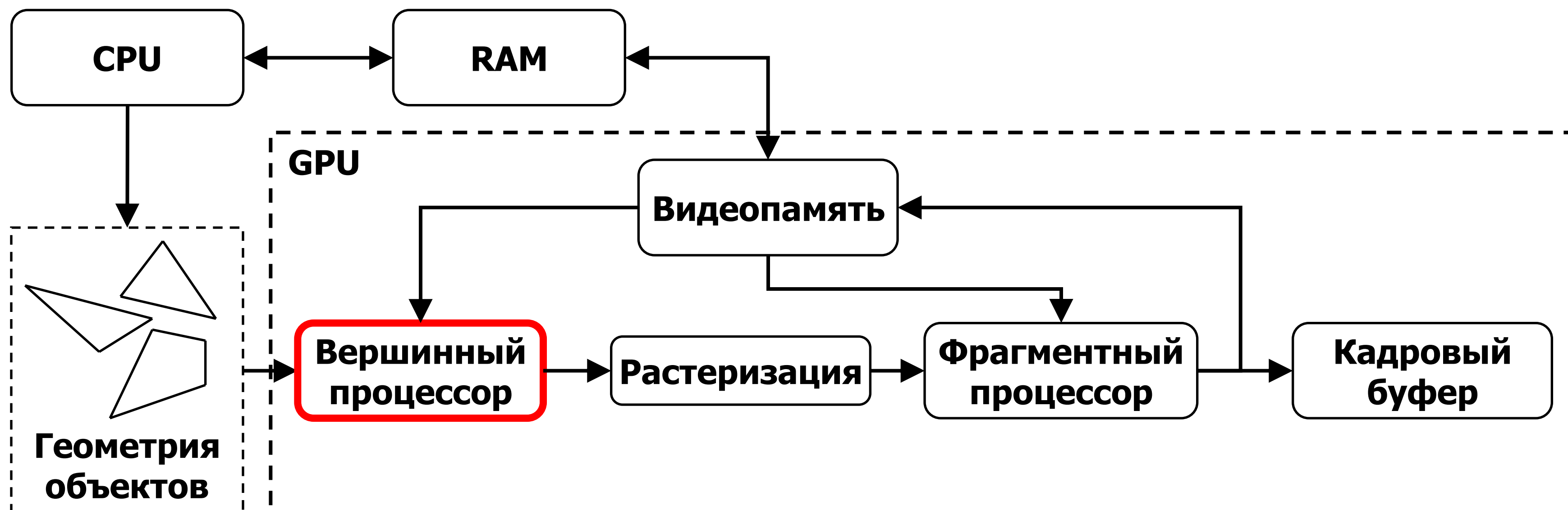
Геометрия объектов



Геометрия объектов

- Является массивом сгруппированных точек, обладающих атрибутами
- Виды группировки:
  - отдельные точки;
  - пары точек — линии;
  - треугольники;
  - многоугольники
- Атрибуты точек:
  - координаты точки в пространстве;
  - цвет;
  - координаты текстуры (до 8 различных векторов);
  - произвольные пользовательские атрибуты

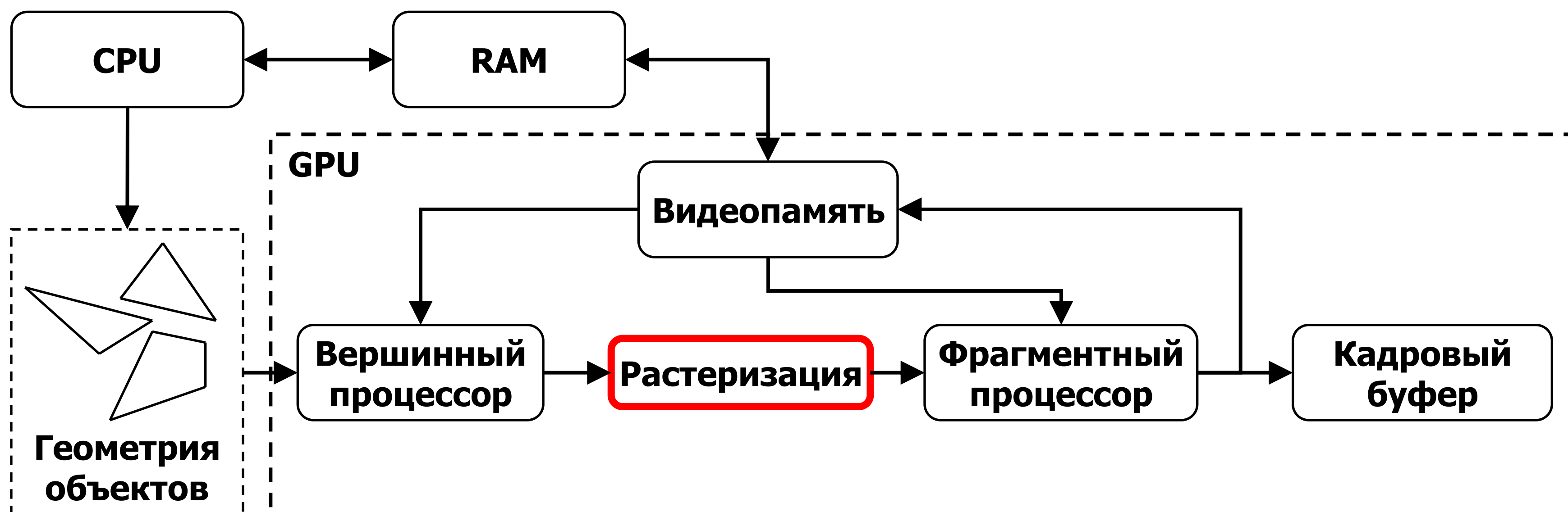
Вершинный процессор



Вершинный процессор

- Полностью программируемый этап графического конвейера
- Входные данные — последовательность точек (и их атрибутов), задающих геометрию объектов
- Каждая точка обрабатывается независимо
- Точки обрабатываются параллельно на нескольких вершинных процессорах (например, на 12)
- Работает над векторами (координаты, цвет — вектора)
- Может изменять произвольным образом координаты и атрибуты точек
- Выходные данные — последовательность точек с преобразованными координатами и атрибутами
- Может добавить новые или убрать некоторые атрибуты

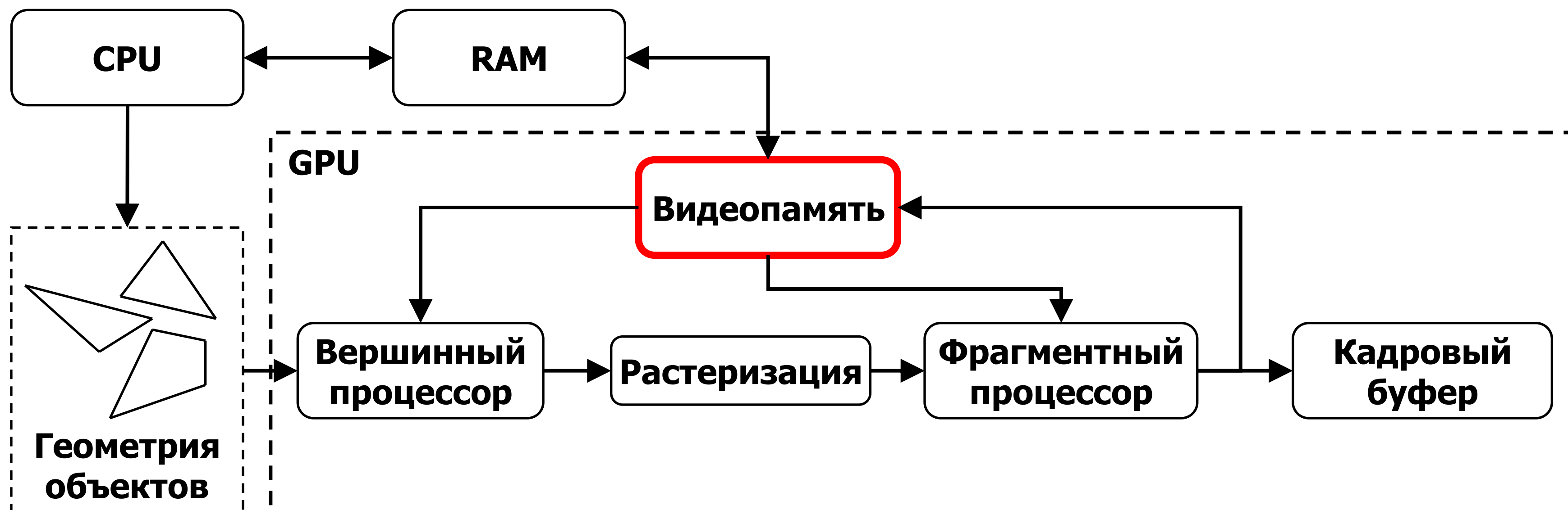
Растеризация



GRAPHICS & MEDIA LAB
VIDEO GROUP

- Этап с фиксированной функциональностью
- Входные данные — набор преобразованных вершинным процессором точек
- Группирует точки в полигоны
- Проектирует полигоны на экран
- Определяются пиксели экрана, покрываемые полигоном
- Для каждого пикселя линейно интерполируются все атрибуты вершин (кроме координат)
- Выходные данные — поток пикселей (фрагментов), обладающих атрибутами

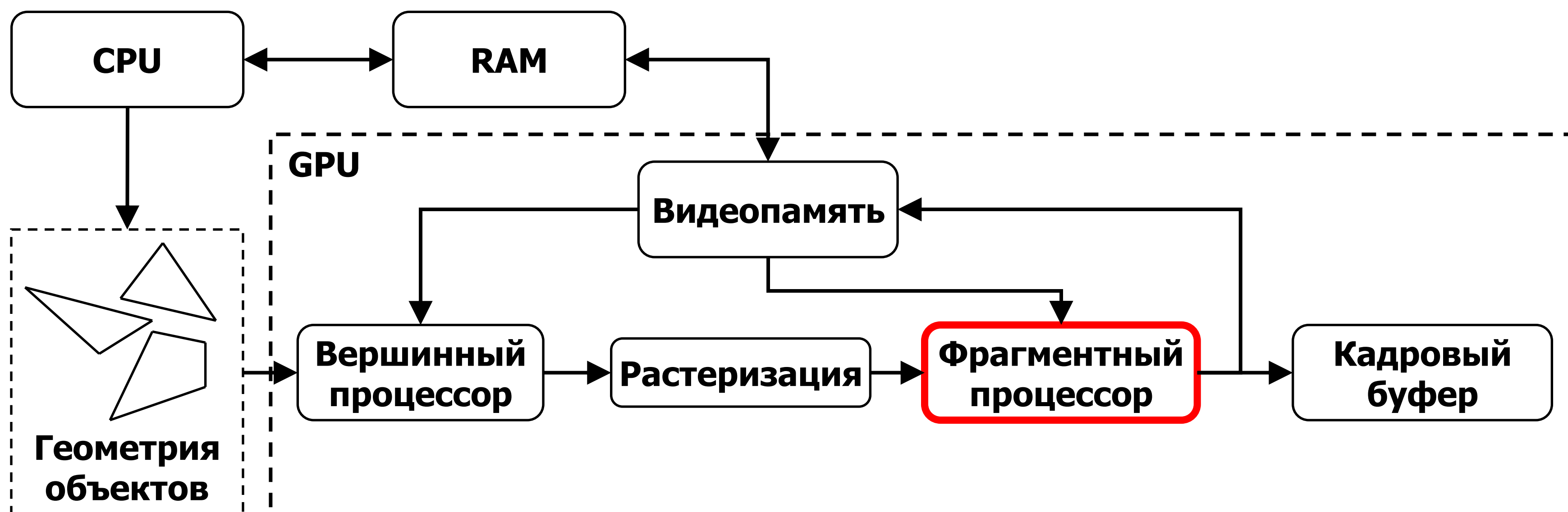
Текстуры



Текстуры

- Текстуры — массивы
- Поддерживаются 1D-, 2D- и 3D-текстуры
- Ограничение на размер: максимум 4096×4096 пикселей для 2D-текстуры
- Изначально поддерживаются только квадратные POT-текстуры
- Существуют расширения для поддержки прямоугольных NPOT-текстур
- Форматы пикселей: от однокомпонентных (только яркость) до четырехкомпонентных (ARGB)
- Значения могут храниться в виде байта или числа с плавающей точкой

Фрагментный процессор

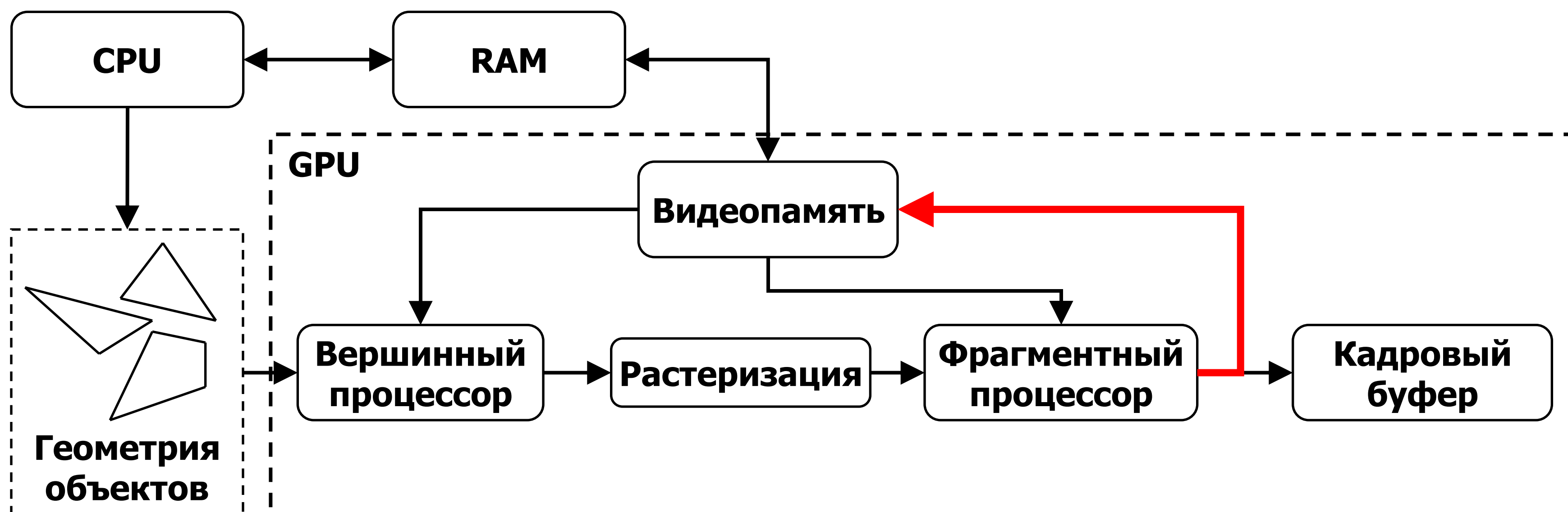


GRAPHICS & MEDIA LAB
VIDEO GROUP

Фрагментный процессор

- Полностью программируемый этап графического конвейера
- Входные данные — поток пикселей (фрагментов) с интерполированными атрибутами
- Каждый фрагмент обрабатывается независимо
- Пиксели обрабатываются параллельно на нескольких фрагментных процессорах (например, на 24)
- Имеет доступ к видеопамяти и может получить по заданным координатам пиксель из текстуры (возможна интерполяция на лету)
- Работает с векторами без потери в производительности
- Является средством основных вычисление для GPGPU
- Выходные данные — пиксели конечного изображения

Рендеринг в текстуру



Рендеринг в текстуру

- Текстуры могут быть использованы для сохранения результатов рендеринга
- Текстура может находится либо в режиме только для чтения либо в режиме только для записи
- Текстуры могут быть использованы для хранения промежуточных результатов вычислений
- Текстуры могут быть выгружены обратно в оперативную память

Multiple Rendering Target (MRT)

- Фрагментный шейдер может возвращать несколько результатов
- Рендеринг может производиться в несколько буферов одновременно
- На данный момент максимальное число «целей рендеринга» — четыре
- Использование MRT позволяет при вычислениях добиться оптимальной арифметической интенсивности алгоритмов

Базовые концепции программирования на GPU

GRAPHICS & MEDIA LAB VIDEO GROUP

- Концепции программирования на GPU
  - Массивы — текстуры
  - Тело цикла — шейдер
  - Вычисление — рисование
- Метод пинг-понга
- Редукция
- Перенос вычислений на вершинный процессор

Программа для CPU

```
coef1 = 0.76;
coef2 = 1.0 - coef1;

for (int i = 0; i < imageHeight; ++i) {
    for (int j = 0; j < imageWidth; ++j) {
           imgA[i, j].red =
                    coef1 * imgA[i, j].red + coef2 * imgB[i, j].red;

           imgA[i, j].green =
                    coef1 * imgA[i, j].green + coef2 * imgB[i, j].green;

           imgA[i, j].blue =
                    coef1 * imgA[i, j].blue + coef2 * imgB[i, j].blue;

    }
}
```

Массивы — текстуры

- Двумерный массив — 2D-текстура
- Структура текстуры: RGB
- Формат данных: байты
- Необходимы 3 текстуры, т. к. к `imgA` нужен доступ в режиме чтения и записи одновременно
- Создание необходимой текстуры средствами OpenGL:

```
glTexImage2D(
        GL_TEXTURE_RECTANGLE_ARB,          // Прямоугольная текстура
        GL_RGB8,                           // RGB, 8 бит на канал
        imageWidth,                        // Ширина текстуры
        imageHeight,                       // Высота текстуры
        0,
        GL_BGR,                            // Структура данных в ОЗУ
        GL_UNSIGNED_BYTE,                  // Формат данных в ОЗУ
        imgA                               // Изображение
);
```

Тело цикла — шейдер

- Вычисления в теле цикла можно производить параллельно
- Значение каждого пикселя — вектор
- Пример кода на Cg, реализующего «ядро» вычислений:

```
float3 main(float2 coords : TEXCOORD0,
            uniform samplerRECT imgA, uniform samplerRECT imgB,
            uniform float coef1, uniform float coef2) : COLOR
{
   float3 valA = texRECT(imgA, coords).rgb;
   float3 valB = texRECT(imgB, coords).rgb;

   return valA * coef1 + valB * coef2;
}
```

Векторные типы данных

```
float3 main(float2 coords : TEXCOORD0,
            uniform samplerRECT imgA, uniform samplerRECT imgB,
            uniform float coef1, uniform float coef2) : COLOR
{
   float3 valA = texRECT(imgA, coords).rgb;
   float3 valB = texRECT(imgB, coords).rgb;

   return valA * coef1 + valB * coef2;
}
```

- Поддерживаются векторные величины (до четырех компонент в векторе)
- Матрицы размера до 4×4 (например `float3x4`, `float4x2`)
- Соответствующие операции: умножение вектора на число, скалярное произведение векторов и т. п.

Оператор перестановки

```
float3 main(float2 coords : TEXCOORD0,
            uniform samplerRECT imgA, uniform samplerRECT imgB,
            uniform float coef1, uniform float coef2) : COLOR
{
   float3 valA = texRECT(imgA, coords).rgb;
   float3 valB = texRECT(imgB, coords).rgb;

   return valA * coef1 + valB * coef2;
}
```

- Компоненты в векторе могут быть переставлены
- Примеры:

```
float3 a = valA.gbr + valB.yzx;
float2 b = valA.xy + valB.yz;
float4 c = valA.xxyy + valB.yzzx;
```

Обращение к текстурам

```
float3 main(float2 coords : TEXCOORD0,
            uniform samplerRECT imgA, uniform samplerRECT imgB,
            uniform float coef1, uniform float coef2) : COLOR
{
   float3 valA = texRECT(imgA, coords).rgb;
   float3 valB = texRECT(imgB, coords).rgb;

   return valA * coef1 + valB * coef2;
}
```

- Различные типы текстур используют различные типы адресации пикселей
- Текстуры задаются в виде параметров шейдера

Постоянные параметры шейдера

```
float3 main(float2 coords : TEXCOORD0,
            uniform samplerRECT imgA, uniform samplerRECT imgB,
            uniform float coef1, uniform float coef2) : COLOR
{
   float3 valA = texRECT(imgA, coords).rgb;
   float3 valB = texRECT(imgB, coords).rgb;

   return valA * coef1 + valB * coef2;
}
```

- Задаются функциями графической библиотеки
- Одинаковы для всех обрабатываемых элементов

Переменные параметры шейдера

```
float3 main(float2 coords : TEXCOORD0,
            uniform samplerRECT imgA, uniform samplerRECT imgB,
            uniform float coef1, uniform float coef2) : COLOR
{
   float3 valA = texRECT(imgA, coords).rgb;
   float3 valB = texRECT(imgB, coords).rgb;

   return valA * coef1 + valB * coef2;
}
```

- Являются атрибутами фрагментов, полученных после растеризации
- Тип атрибута, соответствующий данному параметру, указывается после двоеточия

Возвращаемое значение

```
float3 main(float2 coords : TEXCOORD0,
            uniform samplerRECT imgA, uniform samplerRECT imgB,
            uniform float coef1, uniform float coef2) : COLOR;
```

- Возвращение результата аналогично функциям в C
- Возвращаемому значению также приписывается определенная семантика
- При использовании MRT:

```
struct output {
    float4 c0 : COLOR0;
    float4 c1 : COLOR1;
};

output main(float2 coords : TEXCOORD0, uniform samplerRECT texture);
```

Вычисление — рисование

- Устанавливается текстура, в которую будет производиться рендеринг
- Задается ортогональная проекция полигонов на экран
- Рисуется прямоугольник соответствующего размера с необходимыми атрибутами вершин
- Вершинный процессор может провести предварительные вычисления
- В процессе растеризации для каждого пикселя генерируется фрагмент с некоторыми атрибутами
- Пиксельный шейдер выполняется над каждым фрагментом
- Результаты могут быть выгружены обратно в оперативную память или использоваться далее

Ограничения на шейдеры

- Количество инструкций в шейдере ограничено
  - Разделение шейдера на несколько частей и последовательное их исполнение — метод пинг-понга
- Количество обращений к текстурам также ограничено
  - Метод редукции
- Несмотря на наличие кеша, обращение к текстуре занимает больше времени, чем выполнение арифметической операции
  - Проектирование шейдера, в котором количество арифметических операций больше количества обращений к текстурам
- Ветвление доступно, но является «дорогой» операцией
  - Использование доступных на GPU функций вычисления максимума, минимума и т. п.

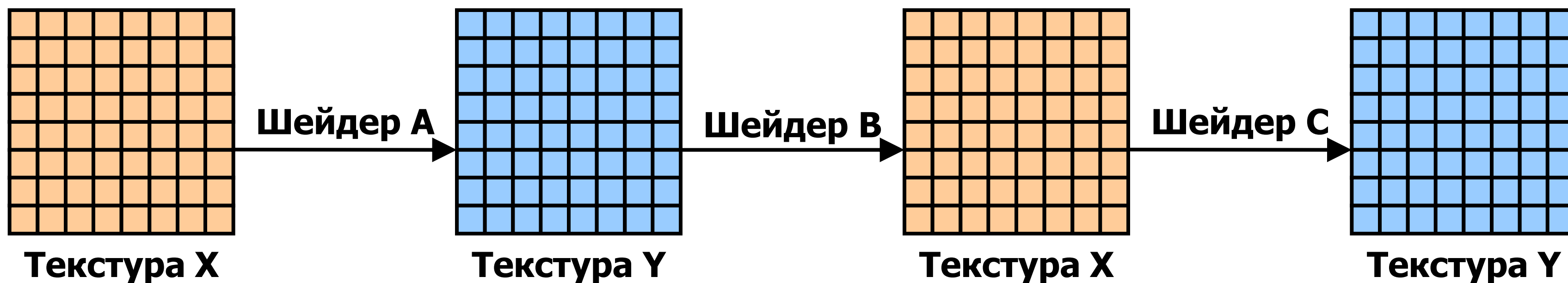
Ограничения на шейдеры

Ограничения		ATI 9600	ATI X800
Вершинный шейдер	Количество атрибутов	16	16
	Количество инструкций	256	256
	Количество регистров	32	32
Пиксельный шейдер	Количество атрибутов	10	10
	Количество инструкций	64	512
	Количество обращений к текстурам	32	512
	Количество регистров	32	64

- Количество инструкций измеряется в машинный командах вершинного и фрагментного процессоров
- Количество доступных регистров важно, т. к. при сложных вычислениях нет возможности сохранить промежуточные результаты в основную память (в отличие от CPU)

Метод пинг-понга

- Используются только две текстуры: X и Y
- В текстуру X загружаются данные
- На первом шаге шейдер берет данные из текстуры X, а рендеринг происходит в текстуру Y
- Изменяется шейдер, текстуры меняются ролями



Редукция

- Применяется для нахождения суммы значений, минимума, максимума в области
- Рендеринг производится в два раза меньшую область
- Шейдер находит локальный максимум в области 2×2

47	2	3	57	5	12	7	8
10	20	6	13	14	15	16	17
19	11	21	22	23	68	25	26
38	29	64	31	32	33	35	34
37	28	39	49	53	42	41	52
46	1	48	40	61	51	44	43
55	71	4	58	69	62	50	60
30	65	66	67	24	59	70	56

47	57	15	17
38	64	68	35
46	49	61	52
71	67	69	70

64	68
71	70

71

Использование min и max

- Пример — поиск медианы из трех значений a, b и c
- Обычный код с использование условных операторов:

```
if (a < b)
{
    if (b < c)
        return b;
    else
        return (a > c) ? a : c;
}
else
{
    if (b > c)
        return b;
    else
        return (a < c) ? a : c;
}
```

Использование min и max

- Определим функцию `clamp`:

$$\forall (a \le b)\ \mathrm{clamp}(x,\ a,\ b) = \max(a,\ \min(b,\ x)) = \begin{cases} a, & x < a \\ x, & a \le x \le b \\ b, & x > b \end{cases}$$

- Тогда медиану можно вычислить:

$$\mathrm{median}(a,\ b,\ c) = \mathrm{clamp}(c,\ \min(a,\ b),\ \max(a,\ b))$$

- Код вычисления медианы на Cg с использованием встроенной функции `clamp`:

```
float4 a = texRECT(texA, coords);
float4 b = texRECT(texB, coords);
float4 c = texRECT(texC, coords);

return clamp(c, min(a, b), max(a, b));
```

Перенос вычислений на вершинный процессор

- Пример шейдера обращающегося к двум соседним текселям:

```
float4 main(float2 coords : TEXCOORD0,
            uniform samplerRECT texture) : COLOR
{
   float4 a = texRECT(texture, coords);
   float4 b = texRECT(texture, coords + float2(1, 0));

   return abs(a - b);
}
```

- Выделенная операция сложения не является частью «полезных» вычислений, но будет выполняться для каждого обрабатываемого пикселя
- Такие вычисления могут быть перенесены на вершинный процессор

Перенос вычислений на вершинный процессор

- Вершинный шейдер:

```
void main(in  float4 in_pos      : POSITION,
          in  float2 in_coords   : TEXCOORD0,
          out float4 out_pos     : POSITION,
          out float2 out_coords0 : TEXCOORD0,
          out float2 out_coords1 : TEXCOORD1)
{
   out_pos = in_pos;
   out_coords0 = in_coords;
   out_coords1 = in_coords + float2(1, 0);
}
```

- На этапе растеризации TEXCOORD0 и TEXCOORD1 будут линейно интерполированы для каждого пикселя

Перенос вычислений на вершинный процессор

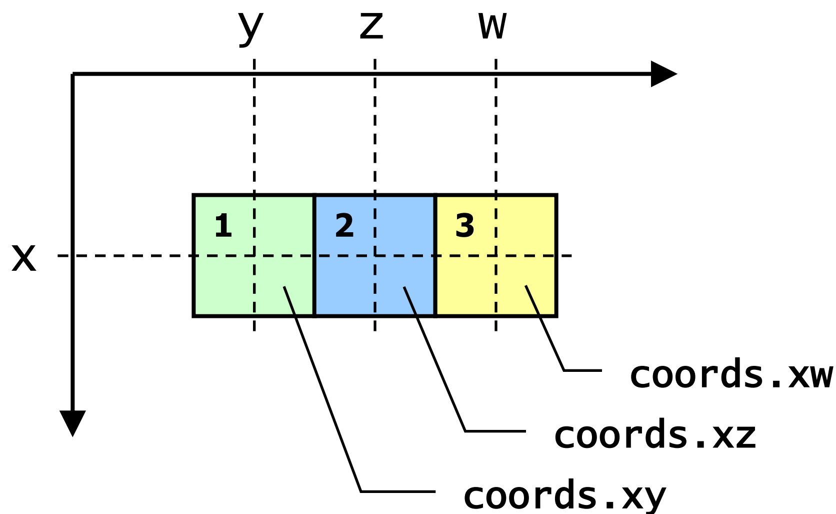
- Тогда пиксельный шейдер можно упростить:

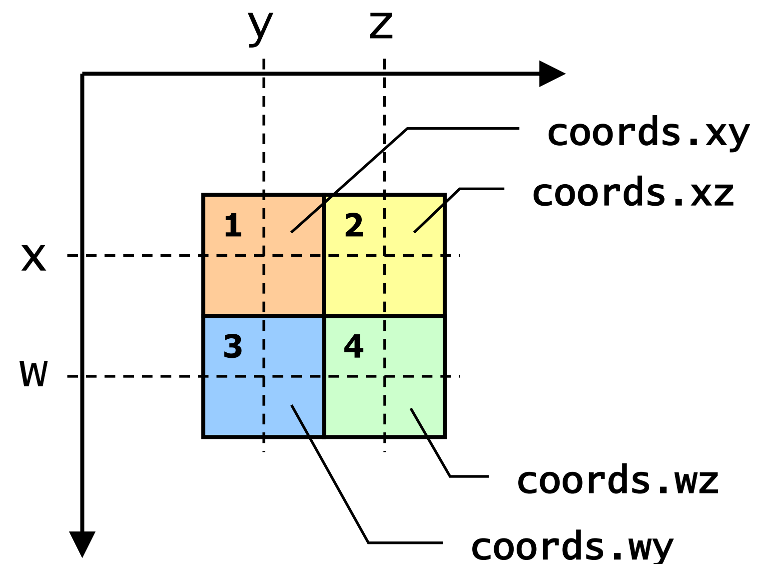
```
float4 main(float2 coords0 : TEXCOORD0,
            float2 coords1 : TEXCOORD1,
            uniform samplerRECT texture) : COLOR
{
   float4 a = texRECT(texture, coords0);
   float4 b = texRECT(texture, coords1);

   return abs(a - b);
}
```

- Но число текстурных координат ограничено восемью и уже при реализации на GPU свертки с ядром 3×3 аналогичный способ применить не удастся
- Для решение этой проблемы используют упаковку координат

Упаковка текстурных координат

- TEXCOORD0—TEXCOORD7 могут быть векторами из четырёх компонент
- При выборке из 2D-текстуры нужно задать только две координаты
- Вектором из четырех компонент можно задать до четырех точек в двумерном пространстве:





Упаковка текстурных координат

- Пример — координаты текстур для свёртки 5×5 (c0—c6 определяют координаты текстур TEXCOORD0—TEXCOORD6 соответственно; c — исходные координаты TEXCOORD0):

OUT.TEXCOORD0 .xy .zy .xw .zw
OUT.TEXCOORD1 .xy .zy .xw .zw
OUT.TEXCOORD2 .xy .xz .xw
OUT.TEXCOORD3 .xy .zy .xw .zw
OUT.TEXCOORD4 .xy .zy .xw
OUT.TEXCOORD5 .xy .zy .xw .zw
OUT.TEXCOORD6 .xw .yw .zw

```
c0 = c.xyxy + float4(-2, 2,-1, 1);
c1 = c.xyxy + float4( 0, 2, 1, 1);
c2 = c.xyyy + float4( 2, 2, 1, 0);
c3 = c.xyxy + float4(-2, 0,-1,-1);
c4 = c.xyxy + float4( 0, 0, 1,-1);
c5 = c.xyxy + float4( 1,-1, 2,-2);
c6 = c.xxxy + float4(-2,-1, 0,-2);
```

Viola, I., Kanitsar, A., Gröller, M.E., "Hardware-Based Nonlinear Filtering and Segmentation using High-Level Shading Languages," *Proceedings of IEEE Visualization*, 2003

Алгоритмы обработки видео на GPU

- Motion Estimation (ME) на GPU
  - Постановка задачи ME
  - Реализация алгоритма на GPU
  - Sub-pixel Motion Estimation
  - Результаты
- Фильтрация видео на GPU
  - Схема работы фильтра шумоподавления
  - Производительность
  - Результаты

Постановка задачи ME

GRAPHICS & MEDIA LAB
VIDEO GROUP

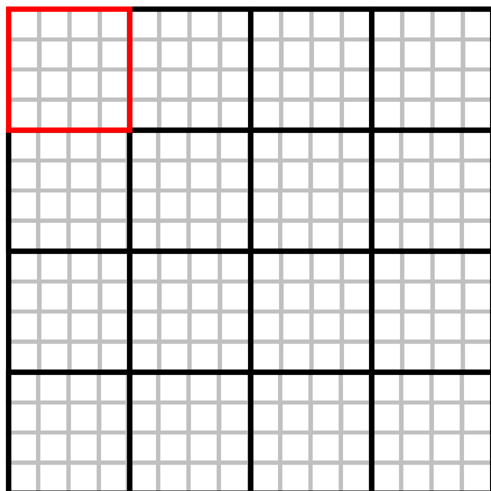
- Текущий кадр разбивается на прямоугольные блоки
- Для каждого блока текущего кадра в предыдущем кадре ищется наиболее «похожий» блок
- В качестве меры схожести двух блоков B_1 и B_2 размера $M \times N$ обычно выступает SAD:

$$\mathrm{SAD}(B_1, B_2) = \sum_{x=1}^{M} \sum_{y=1}^{N} \left| B_1(x, y) - B_2(x, y) \right|$$

- Часто поиск векторов движения осуществляется только для яркостной составляющей изображения
- Самым простым методом поиска векторов движения является полный перебор в некоторой области

Внутреннее представление макро-блока

- Макро-блок разбивается на 16 матриц 4×4
- Каждая из матриц передается в шейдер как постоянный параметр. Это позволяет значительно уменьшить число обращений к текстурам

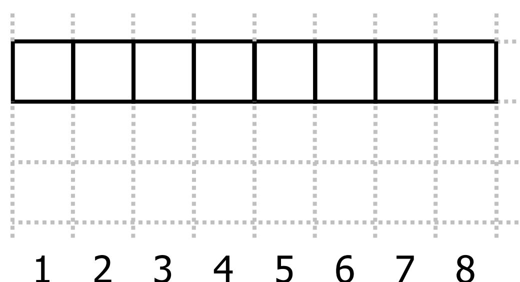


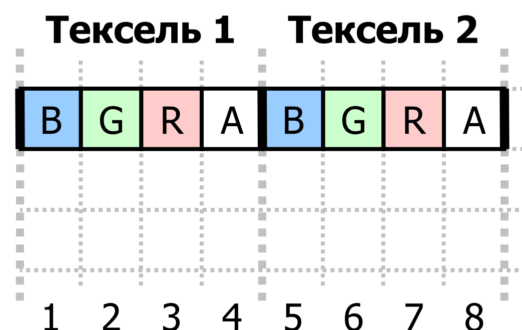
```
// Установка параметра шейдера
float matrix[16];
cgSetMatrixParameterfr(param, matrix);

float3 main(float2 coords : TEXCOORD0,
            uniform samplerRECT texture,
            uniform float4x4 block) : COLOR
{
    // Вычисления
}
```

C.-W. Ho, O. Au, S.-H. Chan, S.-K. Yip, and H.-M. Wong, “Motion Estimation for H.264/AVC using Programmable Graphics Hardware,” *Proceedings IEEE International Conference on Multimedia Expo*, 2006.

Формат данных

- ME выполняется только для яркостной компоненты кадра
- Используются текстуры BGRA, 8 бит на канал
- При загрузке изображения в текстуру исходные значения яркости упаковываются по 4 подряд идущих пикселя в один тексель

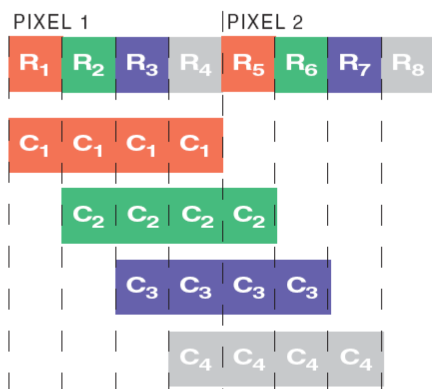




C.-W. Ho, O. Au, S.-H. Chan, S.-K. Yip, and H.-M. Wong, "Motion Estimation for H.264/AVC using Programmable Graphics Hardware," *Proceedings IEEE International Conference on Multimedia Expo*, 2006.

Вычисление SAD

- Пиксельный шейдер загружает из текстуры два соседних текселя (8 значений яркости)
- Вычисление SAD выполняется сразу для 4 соседних позиций
- На примере одной строки блока 4×4:

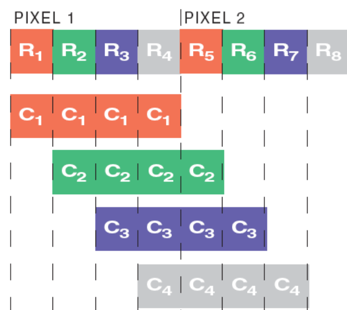


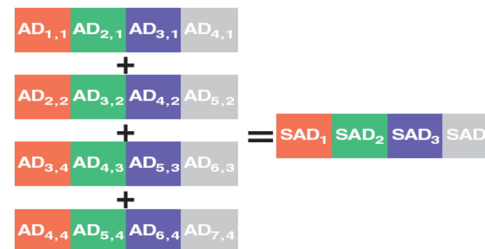
Строка: $\{C_1, C_2, C_3, C_4\}$

$$AD_{i,j} = \left| R_i - C_j \right|$$

C.-W. Ho, O. Au, S.-H. Chan, S.-K. Yip, and H.-M. Wong, "Motion Estimation for H.264/AVC using Programmable Graphics Hardware," *Proceedings IEEE International Conference on Multimedia Expo*, 2006.

Вычисление SAD






```
float4 pix1, pix2;                // Два соседних текселя
float4 c;                         // Строка блока {C1, C2, C3, C4}
float4 ad1, ad2, ad3, ad4;

ad1 = abs(c.xxxx - pix1.xyzw);                 // {AD1,1, AD2,1, AD3,1, AD4,1}
ad2 = abs(c.yyyy - float4(pix1.yzw, pix2.x)); // {AD2,2, AD3,2, AD4,2, AD5,2}
ad3 = abs(c.zzzz - float4(pix1.zw, pix2.xy)); // {AD3,3, AD4,3, AD5,3, AD6,3}
ad4 = abs(c.wwww - float4(pix1.w, pix2.xyz)); // {AD4,4, AD5,4, AD6,4, AD7,4}

float4 sad = ad1 + ad2 + ad3 + ad4;           // {SAD1, SAD2, SAD3, SAD4}
```

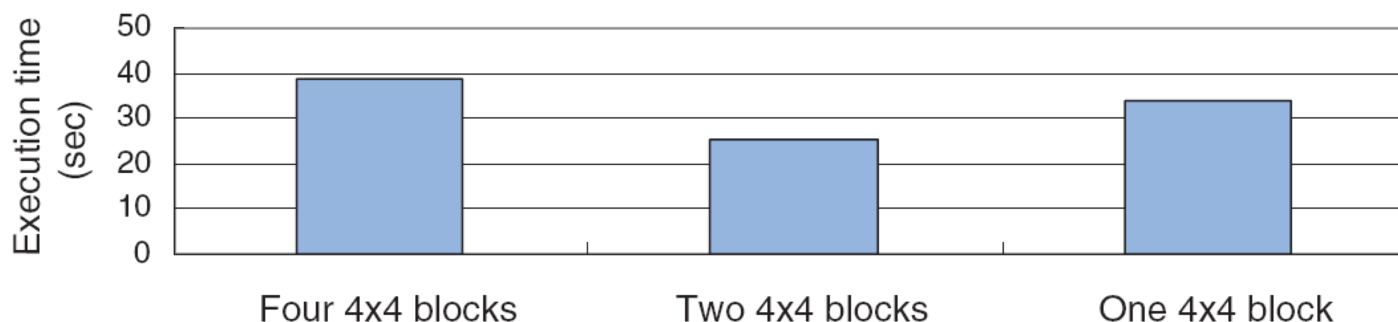
C.-W. Ho, O. Au, S.-H. Chan, S.-K. Yip, and H.-M. Wong, "Motion Estimation for H.264/AVC using Programmable Graphics Hardware," *Proceedings IEEE International Conference on Multimedia Expo*, 2006.

Arithmetic Intensity

- Арифметическая интенсивность шейдера (arithmetic intensity) — отношение числа операций АЛУ к числу выборок из текстур
- Необходимо, чтобы арифметическая интенсивность шейдеров была достаточно высока — около 8 операций АЛУ на один доступ к текстуре
  - Причина — «медленный» доступ к памяти
  - Во время получения текселя АЛУ продолжает работу
- Для различных GPU оптимальная арифметическая интенсивность различна
  - Решение — реализация нескольких версий шейдера для различных GPU

Arithmetic Intensity

- Решением проблемы малой арифметической интенсивности может быть использование MRT
- Для одних и тех же текселей одновременно вычисляются сразу несколько результатов
- В предлагаемой реализации ME SAD можно вычислять более чем для одного блока за один проход
- Число выборок из текстуры при этом не изменяется, т. к. исходный блок задается в качестве параметра



Сравнение времени выполнения ME для видео формата 1080p и областью перебора ±256 пикселей при различном числе обрабатываемых блоков за один проход на GeForce 7800GT PCIe

C.-W. Ho, O. Au, S.-H. Chan, S.-K. Yip, and H.-M. Wong, "Motion Estimation for H.264/AVC using Programmable Graphics Hardware," *Proceedings IEEE International Conference on Multimedia Expo*, 2006.

Производительность предлагаемого метода ME

P4 1.8GHz SIMD-optimized
P4 3.2GHz SIMD-optimized
GF6600GT Readback
GF7800GT Readback
GF6600GT w/o readback
GF7800GT w/o readback

Relative Performance

0 2 4 6 8 10 12 14 16 18 20 22 24

720p±32 720p±64 720p±128 720p±256 1080p±64 1080p±128 1080p±256

- Накладные расходы вызовов API не позволяют полностью раскрыть потенциал GPU при малых областях перебора
- «Узким местом» при вычислениях является выгрузка результатов

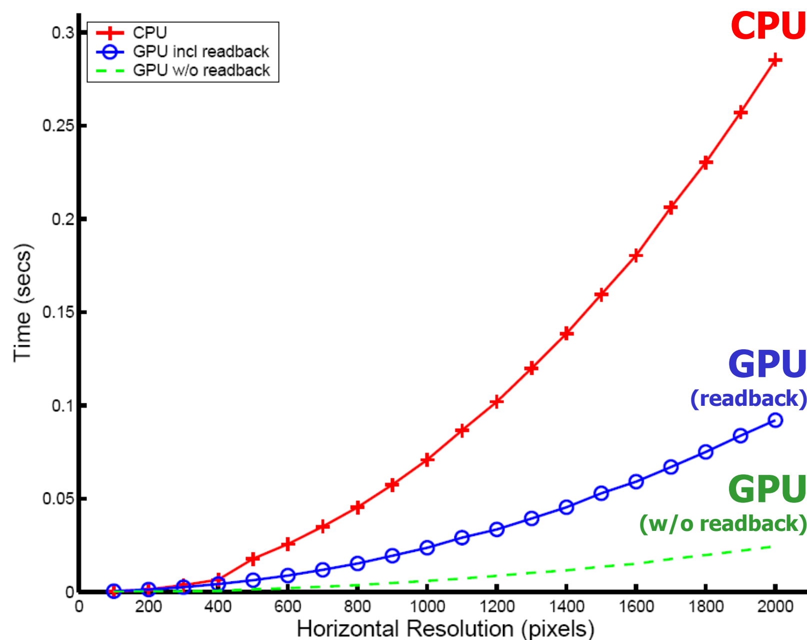
C.-W. Ho, O. Au, S.-H. Chan, S.-K. Yip, and H.-M. Wong, "Motion Estimation for H.264/AVC using Programmable Graphics Hardware," *Proceedings IEEE International Conference on Multimedia Expo*, 2006.

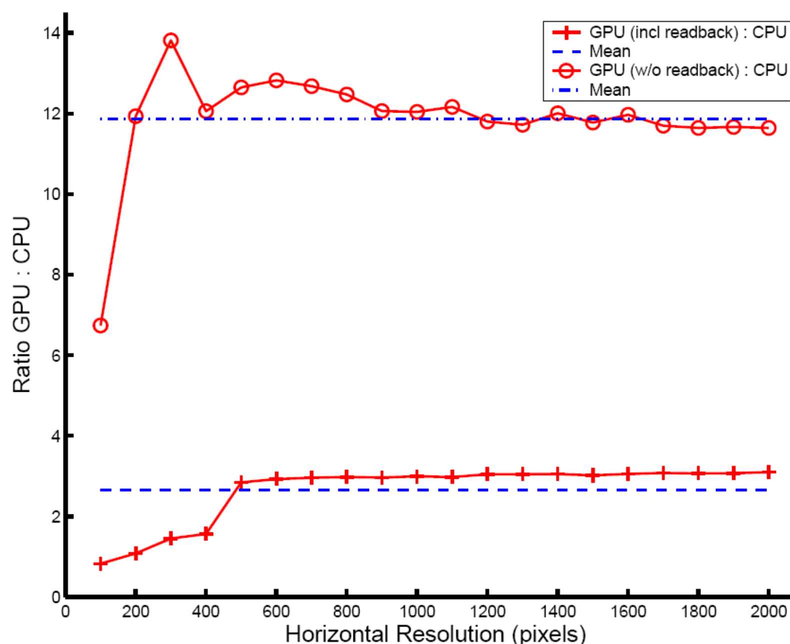
Sub-pixel Motion Estimation

GRAPHICS & MEDIA LAB VIDEO GROUP

- Пиксельная точность обычно недостаточна для качественной компенсации движения
- Для повышения качества используется полупиксельная, четверть-пиксельная или бо́льшая точность
- Необходима интерполяция
- GPU позволяет проводить билинейную интерполяцию текстур во время обращения к ним
- Скорость интерполяции очень высока, т. к. эта функция GPU полностью реализована на аппаратном уровне

Скорость интерполяции на GPU

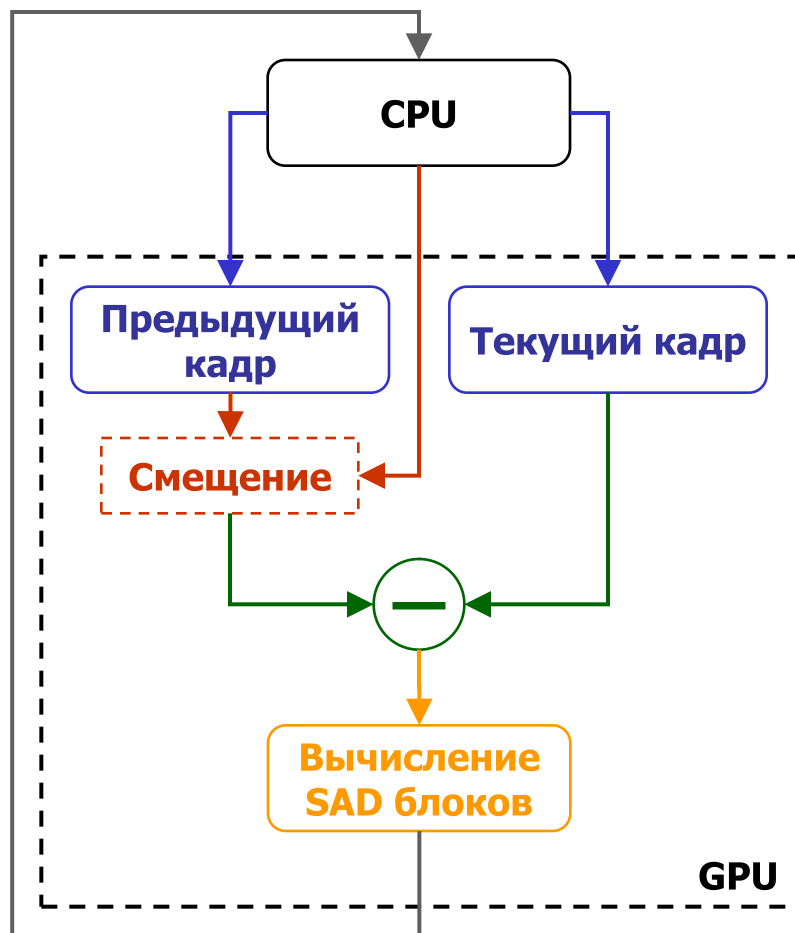




Сравнение времени выполнения билинейной интерполяции на Pentium 4 1,6 ГГц и GeForce FX 5600 с учетом времени выгрузки результатов обратно в ОЗУ и без него

Francis Kelly and Anil Kokaram, "Fast Image Interpolation for Motion Estimation using Graphics Hardware," *IS&T/SPIE Electronic Imaging — Real-Time Imaging VIII*, 2004

Схема sub-pixel ME на GPU

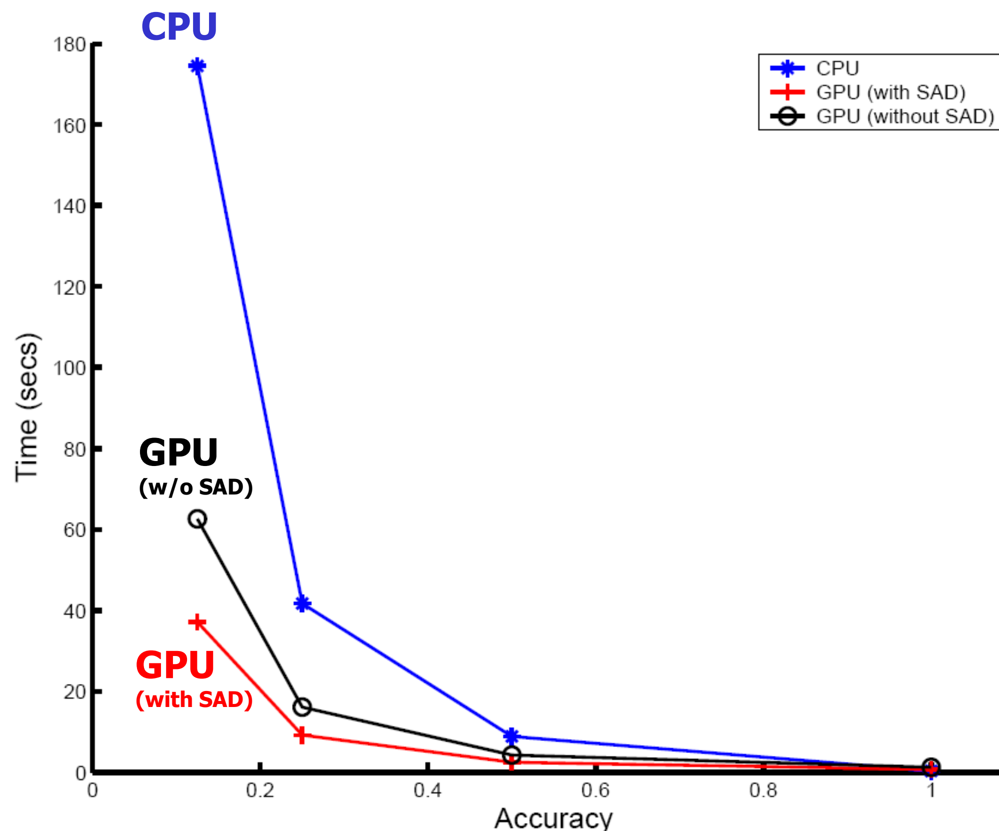


- В видеопамять загружаются текущий и предыдущий кадры

  Для каждого вектора движения из области поиска:
- Задаются два вектора текстурных координат: для текущего кадра и предыдущего с учетом вектора движения
- Находится разность текущего и предыдущего кадров с учетом вектора движения
- Методом редукции для всех блоков вычисляется SAD
- Результаты выгружаются в ОЗУ

Francis Kelly and Anil Kokaram, "Fast Image Interpolation for Motion Estimation using Graphics Hardware," *IS&T/SPIE Electronic Imaging — Real-Time Imaging VIII*, 2004

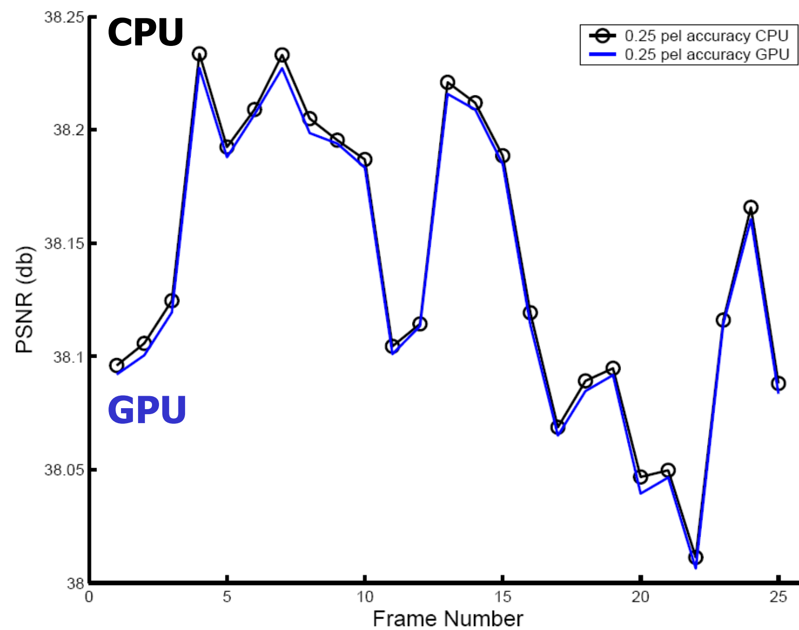
Производительность предлагаемого метода



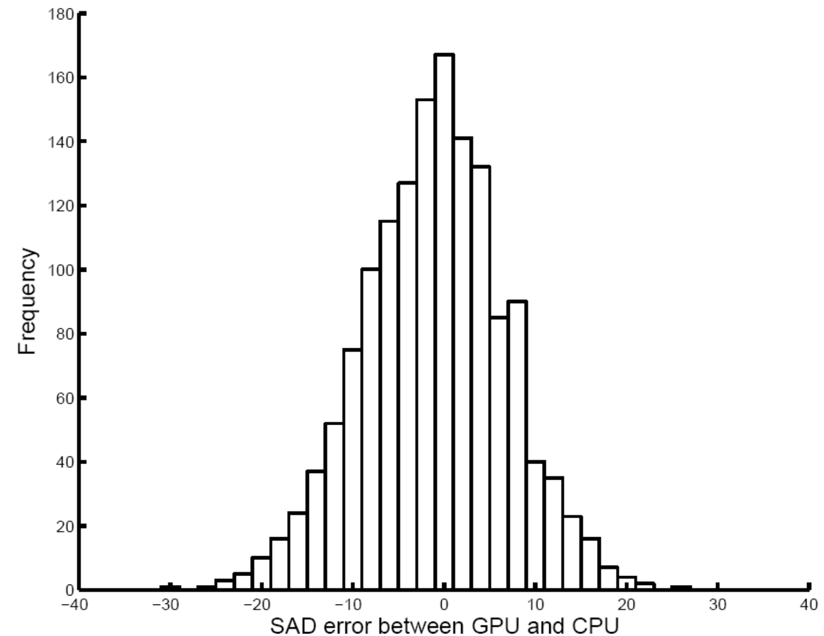
Время работы ME при различной точности и реализации без использования GPU, с использованием GPU и с использованием GPU в том числе и для вычисления SAD блоков

Francis Kelly and Anil Kokaram, "Fast Image Interpolation for Motion Estimation using Graphics Hardware," *IS&T/SPIE Electronic Imaging — Real-Time Imaging VIII*, 2004

Точность предлагаемого метода



Качество кадров, скомпенсированных с четверть-пиксельной точностью на CPU и GPU, по сравнению с исходным кадром

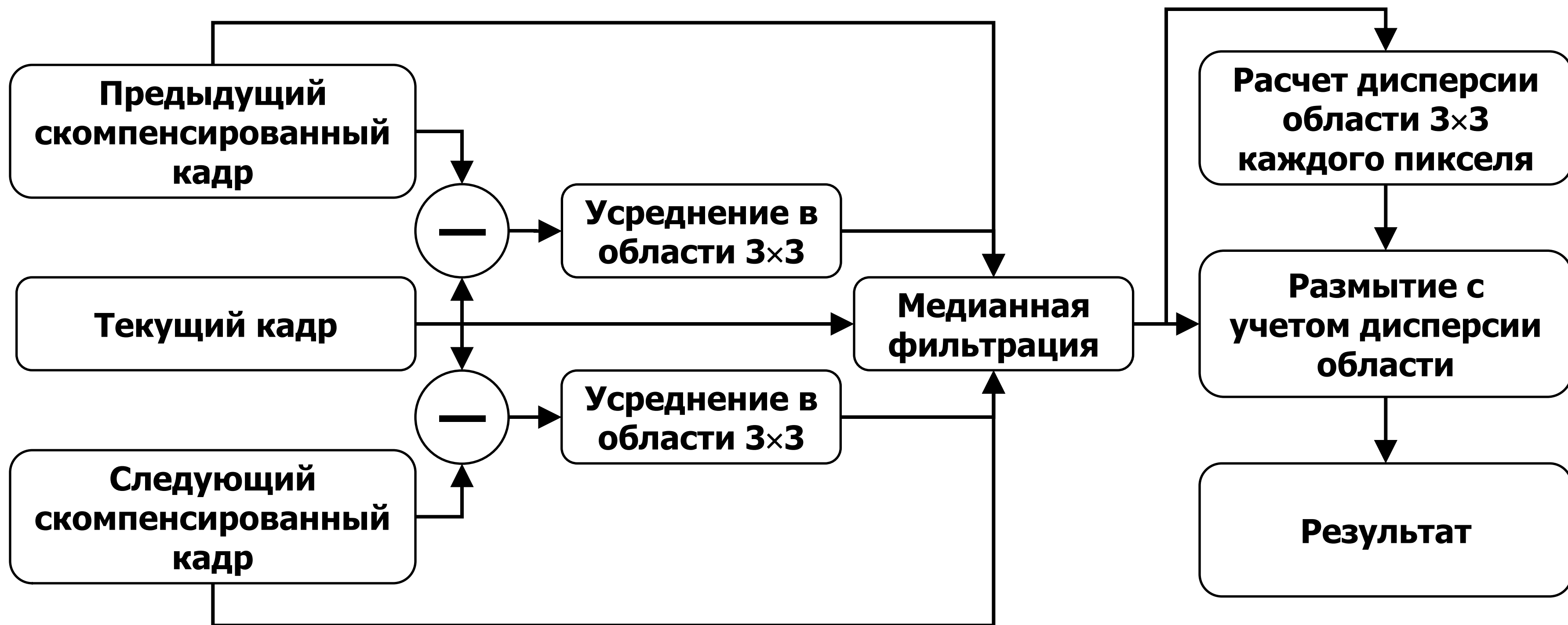


Гистограмма ошибок вычисления SAD полученных на GPU из-за погрешностей при интерполяции

Francis Kelly and Anil Kokaram, "Fast Image Interpolation for Motion Estimation using Graphics Hardware," *IS&T/SPIE Electronic Imaging — Real-Time Imaging VIII*, 2004

Схема работы фильтра шумоподавления

- Схема фильтрации, полностью реализованная на GPU:



Реализация на GPU

- Формат данных
  - Фильтрация проводится отдельно для яркостной и цветностных компонент
  - Изображения находятся в текстурах в упакованном виде — по четыре точки в одном текселе
- Усреднение в области 3×3 реализовано в два прохода: суммирование проводится сначала по горизонтали, затем по вертикали
- Медианная фильтрация
  - Проводится для пикселя только, если разности между скомпенсированными и исходным кадром в данной точке меньше заданного порога
  - Для реализации порога условный оператор не используется. Применяется функция `step`:

$$\text{step}(a, x) = \begin{cases} 0, & x < a \\ 1, & x \geq a \end{cases}$$

Реализация на GPU

- Для области 3×3 каждого пикселя вычисляется дисперсия
  - Расчет проводится по формуле (S — множество пикселей области; N_S — число пикселей в S; $f(q)$ — значение пикселя q):

$$\mathbf{D}(S) = \frac{1}{N_S^2}\left(N_S \sum_{q\in S} f^2(q) - \left(\sum_{q\in S} f(q) \right)^2 \right)$$

  - Вычисление проходит за 6 проходов
    - Вычисляется $f^2(q)$ для каждого пикселя
    - Аналогично усреднению находятся суммы
    - Вычисляется результирующее значение
- В зависимости от дисперсии, рассчитанной для области каждого пикселя, происходит размытие изображения полученного после медианной фильтрации

Размытие на GPU

- Так как сила размытия должна быть различна в разных областях изображения, то фильтр не может быть реализован как сепарабельный
- Коэффициенты нормированы так, чтобы сумма элементов ядра свёртки равнялась 1
- Для области 3×3 в ядре свёртки присутствуют только 3 различных значения
- Все коэффициенты хранятся в видеопамяти в виде одномерной RGB-текстуры
  - Каждый тексель задает матрицу размытия
  - Различным точкам текстуры соответствуют матрицы размытия различной интенсивности
  - Элементы RGB-векторов — байты

Размытие на GPU

- Получение коэффициентов матриц размытия для четырёх соседних пикселей:

```
float4 disp = texRECT(dispersion, c);          // Получение дисперсии

float4x3 kern;
kern[0] = tex1D(kernel, disp.x).zyx;           // Выборка коэффициентов
kern[1] = tex1D(kernel, disp.y).zyx;
kern[2] = tex1D(kernel, disp.z).zyx;
kern[3] = tex1D(kernel, disp.w).zyx;
float3x4 coef = transpose(kern);               // Транспонирование

float4 value = texRECT(texture, c) * coef[0]; // Вычисления
```

- Транспонирование проводится для получения вектора, состоящего из четырех коэффициентов, отвечающих одной позиции в матрицах размытия четырех соседних пикселей

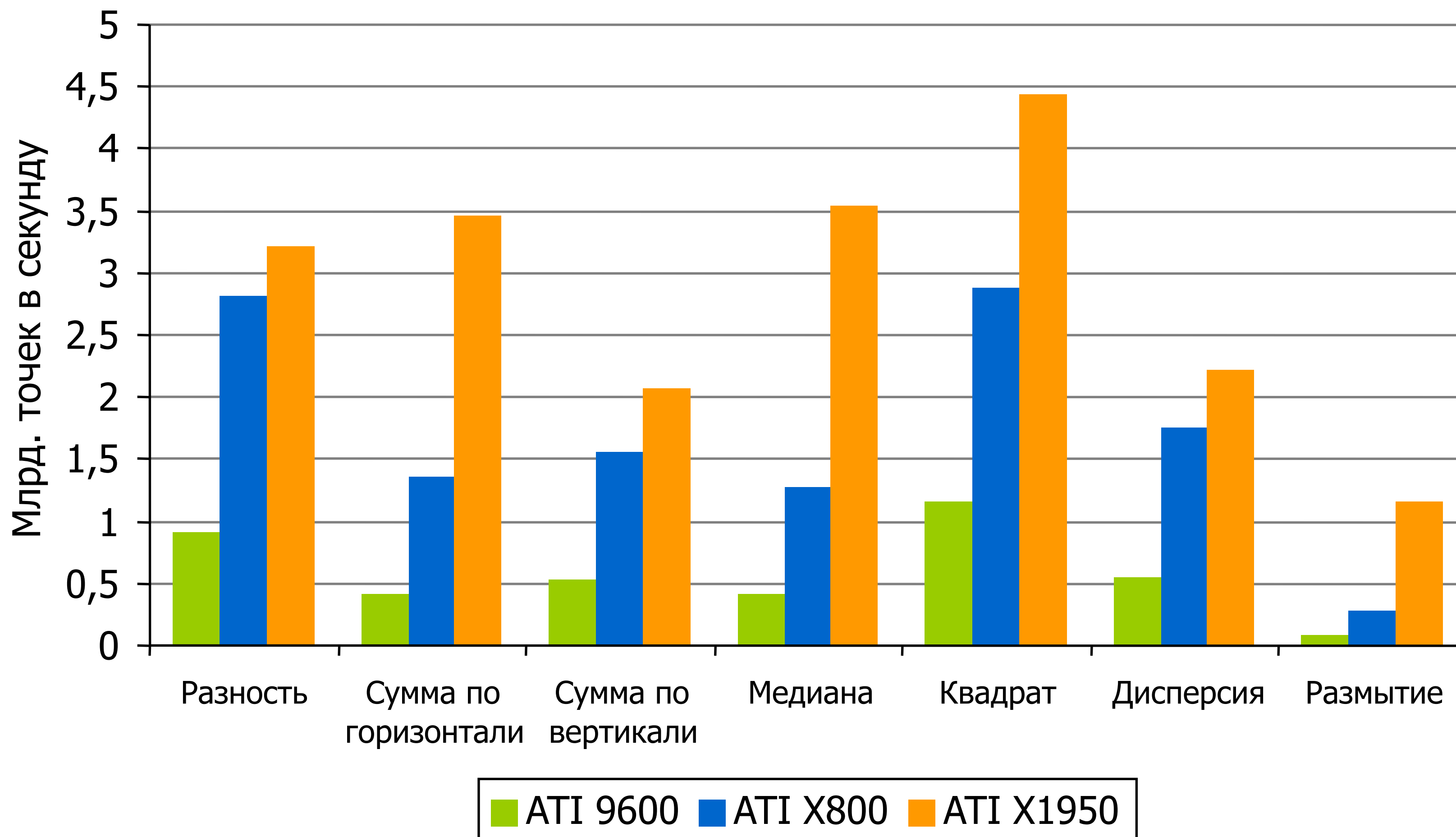
Производительность шейдеров, млрд. точек в секунду

GRAPHICS & MEDIA LAB VIDEO GROUP

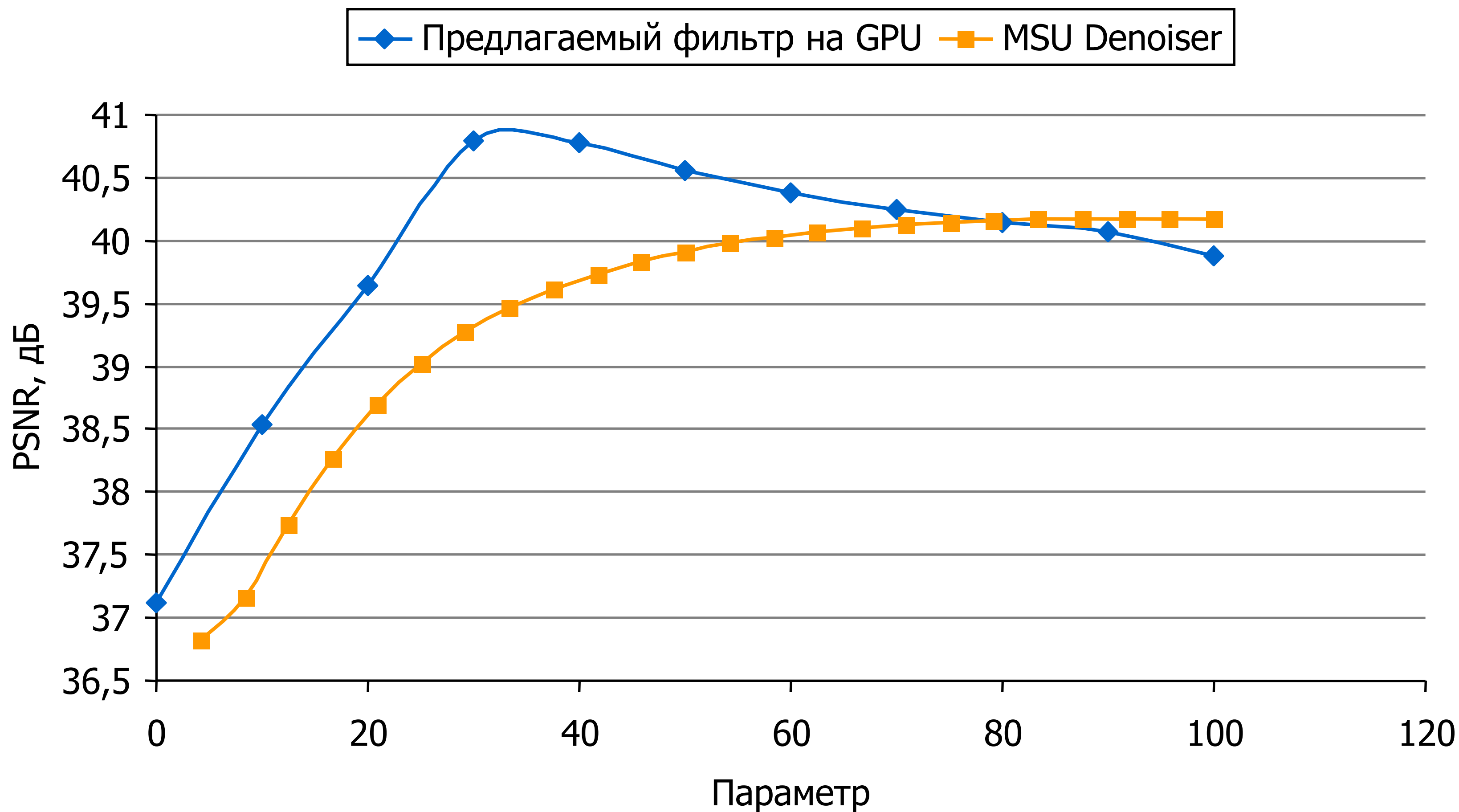
Шейдер	ATI 9600	ATI X800	ATI X1950
Вычисление разности	0,909	2,807	3,213
Суммирование по горизонтали	0,417	1,351	3,457
Суммирование по вертикали	0,522	1,550	2,076
Медианная фильтрация	0,412	1,272	3,546
Возведение в квадрат	1,162	2,886	4,430
Расчет дисперсии	0,546	1,763	2,225
Размытие	0,084[2]	0,288	1,152
Размытие (этап 1)[1]	0,273	0,859	—[3]
Размытие (этап 2)[1]	0,124	0,407	—[3]

[1] — из-за нехватки ресурсов ATI 9600 шейдер размытия пришлось разделить на два;
[2] — вычислено по результатов двух этапов размытия;
[3] — измерения не проводились

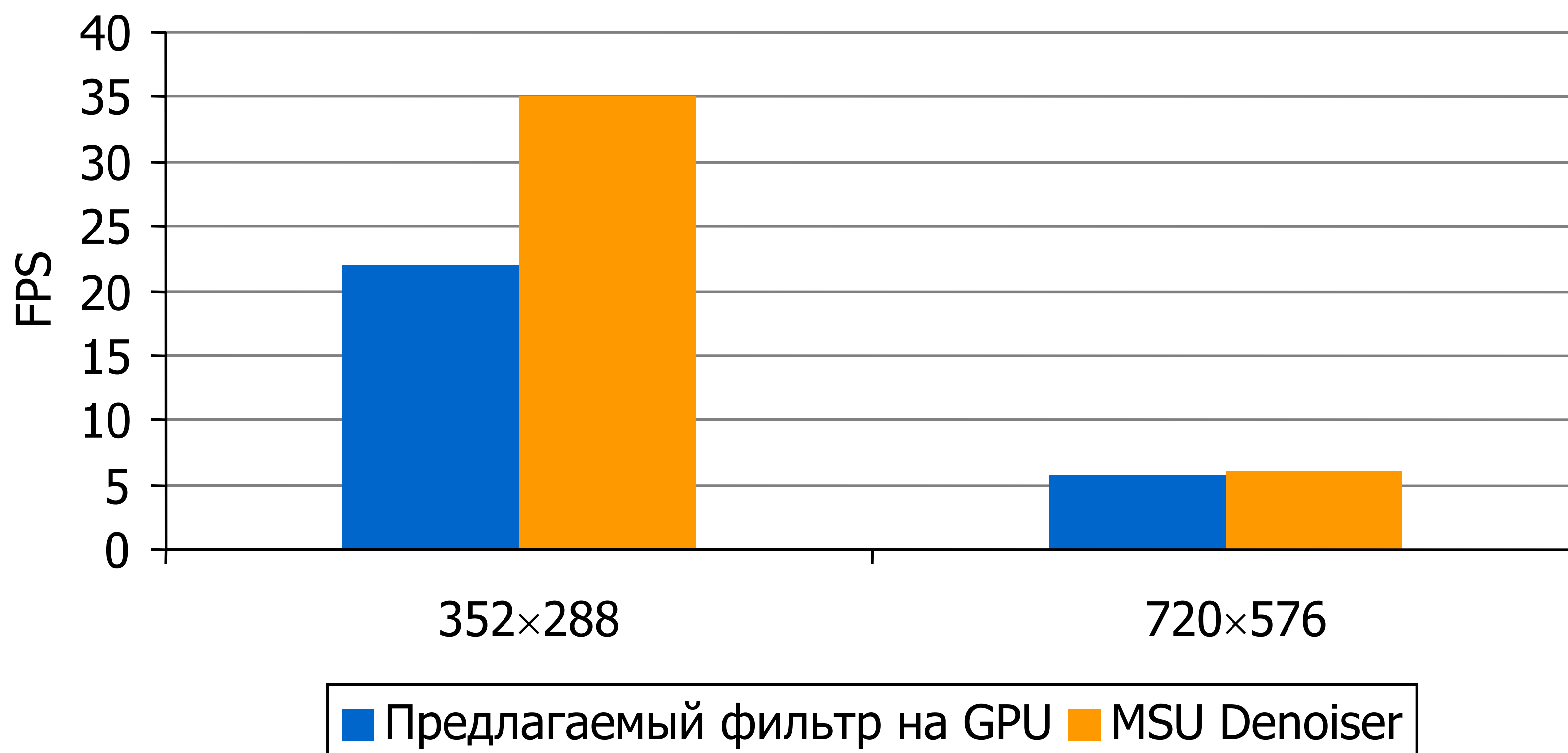
Производительность шейдеров



Качество шумоподавления



Производительность



Дальнейшая работа

- Реализация на GPU других методов шумоподавления
- Перенос компенсации движения на GPU
- Создание фильтра шумоподавления для VirtualDub, выполняющего всю обработку исключительно на GPU
- Перенос большинства алгоритмов обработки видео на GPU

Список литературы

- GPGPU
  - http://www.gpgpu.org
  - J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kuger, A. E. Lefohn, and T. J. Purcell, "A survey of General-Purpose Computation on Graphics Hardware". In *EUROGRAPHICS 2005, State of the Art reports*, pp. 21–51, 2005
- Motion Estimation
  - Francis Kelly and Anil Kokaram, "General Purpose Graphics Hardware for Accelerating Motion Estimation," in *Irish Machine Vision and Image Processing Conference (IMVIP)*, Sept. 2003
  - Francis Kelly and Anil Kokaram, "Fast Image Interpolation for Motion Estimation using Graphics Hardware," in IS&T/SPIE Electronic Imaging — Real-Time Imaging VIII, May 2004, vol. 5297, pp. 184–194
  - Sébastien Mazaré, Renaud Pacalet, Jean-Luc Dugelay, "Using GPU for Fast Block-matching," *EUSIPCO 2006, 14th European Signal Processing Conference*, September 4–8, 2006, Florence, Italy
  - C.-W. Ho, O. Au, S.-H. Chan, S.-K. Yip, and H.-M. Wong, "Motion Estimation for H.264/AVC using Programmable Graphics Hardware," *Proceedings IEEE International Conference on Multimedia Expo (ICME)* , pp. 2049–2052, Toronto, Canada, 9–12 July, 2006
- Шумоподавление
  - Andreas Langs, Matthias Biedermann, "Filtering Video Volumes using the Graphics Hardware," *15th Scandinavian Conference on Image Analysis, SCIA 2007, LNCS 4522*, p. 878–887, June 10–14, 2007, Aalborg, Denmark
  - Ivan Viola, Armin Kanitsar, Meister Eduard Gröller, "Hardware-Based Nonlinear Filtering and Segmentation using High-Level Shading Languages," *Proceedings of the 14th IEEE Visualization 2003*, 2003

Список литературы

- Декодирование видео
  - Guobin Shen, Guang-Ping Gao, Shipeng Li, Heung-Yeung Shum, and Ya-Qin Zhang, “Accelerate Video Decoding with Generic GPU,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 5, pp. 685–693, May 2005
  - Bart Pieters, Dieter Van Rijsselbergen, Wesley De Neve, Rik Van de Walle, “Motion Compensation and Reconstruction of H.264/AVC Video Bitstreams using the GPU,” *Image Analysis for Multimedia Interactive Services*, 6–8 June, 2007 pp. 69–69
- Вейвлеты
  - Carsten Stocklöw, Stefan Noll, “Realtime Wavelet Video Encoding with Generic Graphics Processing Unit,” *Proceedings Eurographics Italian Chapter Conference*, 2007, pp. 73–78
  - T. T. Wong, C. S. Leung, P. A. Heng and J. Wang, “Discrete Wavelet Transform on Consumer-Level Graphics Hardware,” *IEEE Transactions on Multimedia*, Vol. 9, No. 3, April 2007, pp. 668–673
- DCT/IDCT
  - Bo Fang, Guobin Shen, Shipeng Li, and Huifang Chen, “Techniques for Efficient DCT/IDCT Implementation on Generic GPU,” in *Proceedings of the IEEE International Symposium on Circuits and Systems*, May 2005, vol. 2, pp. 1126–1129

Лаборатория компьютерной графики и мультимедиа

Видеогруппа это:

- Выпускники в аспирантурах Англии, Франции, Швейцарии (в России в МГУ и ИПМ им. Келдыша)
- Выпускниками защищено 5 диссертаций
- Наиболее популярные в мире сравнения видеокодеков
- Более 3 миллионов скачанных фильтров обработки видео