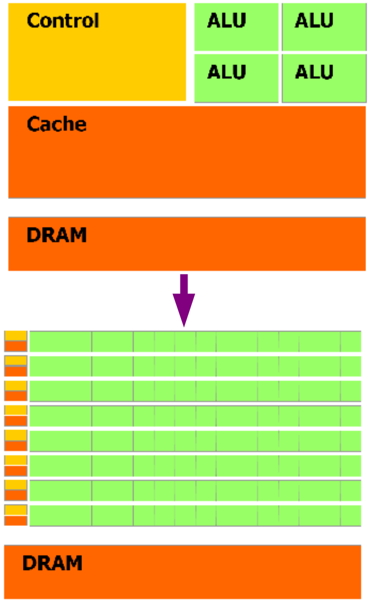
Optimization Strategies

- **Complex memory hierarchies with drastically varied speeds**
  - Only 6 – 12 GB/s to the system memory
  - ~ 500 clock cycle latency to access global memory
  - And it will be worse if optimal access patterns are not followeed
  - Very low Bandwidth-per-flop ratio (50 GB/s per Tflop)
- **Varying architectures**
  - Amount of registers, sizes of caches vary drastically and hence optimal grid configuration and accepted kernel complexity
  - Balance of operation performances changes between devices as well
- **GPUs optimized for FP additions and multiplications**
  - Branching and many other operations are very expensive



What are this NVIDIA GFlops

GTX Titan is able to execute 2688 FMA (A*B + C) instructions (counted as 2 instructions) per clock cycle if not stuck on memory access: 2688 * 2 * 837 MHz = 4,499,712 MFlops

Instructions per CU per clock	**Fermi**	**Kepler**
FP FMA, ADD, MULTIPLY	32	192
FP Reciprocal	4	32
Integer ADD	1	~ 1
Integer MULTIPLY	16	32
Integer Compare	16	8
Type Conversions	16	8

Type conversions on GTX Titan will be slower than on Fermi!

Special instructions

Allow IEEE 754 incompatibility and get faster fp performance but lower precision

```
err = clBuildProgram(app, num_devices, devices,  “-cl-fast-relaxed-math”, NULL, NULL);
```

There is advanced NVIDIA instructions which will be not used by OpenCL optimizing compiler.

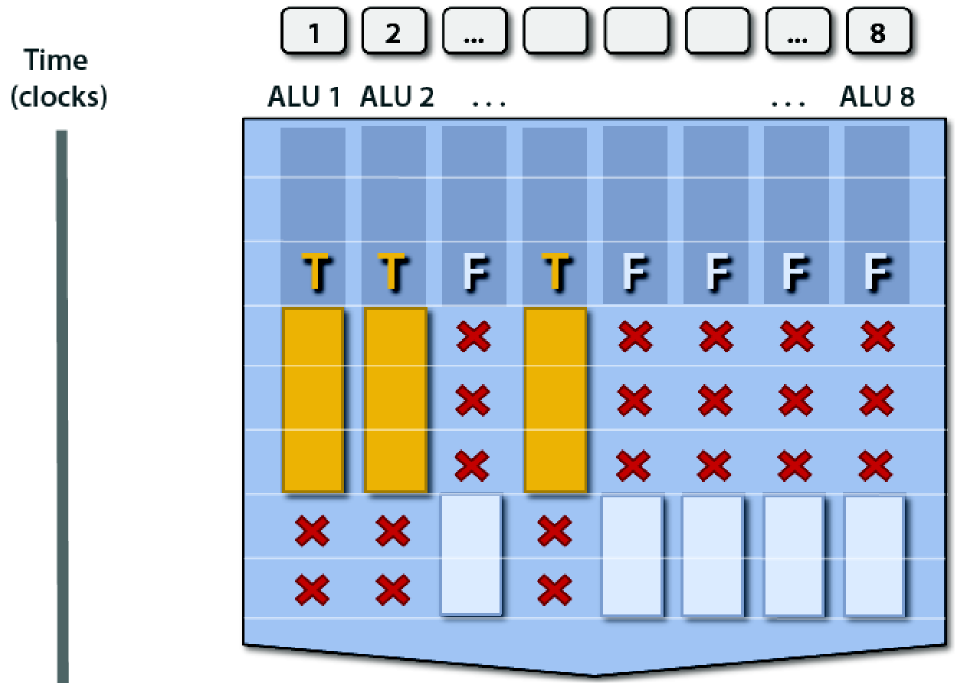
- Math functions with reduced precision: **__sinf**, **__cosf**, **__expf**, …
- SIMD Video Instructions **vabsdiff**, **vadd**, **vsub**, **vmin**, **vmax**, **vset**, **vabsdiff2** (so on…), **vabsdiff4** (so on...) operating on 1-2-4 byte integer arguments.
- Kepler **shfl** instruction intended to exchange data between warp work-items.

Compute absolute difference of 2 byte vectors

```
__kernel void multiply(__global unsigned char *res, *a, *b) {
   res[get_global_id(0)] = abs(a[get_global_id(0)] - b[get_global_id(0)]);
}
```

```
__kernel void multiply(__global unsigned int *res, *a, *b) {  // times less work-items
   unsigned res0, a0 = *(__global unsigned*)&a[i], b0 = *(__global unsigned*)&b[i];
   asm("vabsdiff4.u32.u32.u32 %0, %1, %2, %0;" : "=r"(res0) : "r"(a0), "r"(b0));
   *(__global unsigned*)&res[i * size + j] = res0;
}
```

Conditionals



Not all ALUs do useful work!
Worst case: 1/8 performance

```
<unconditional
shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
shader code>
```

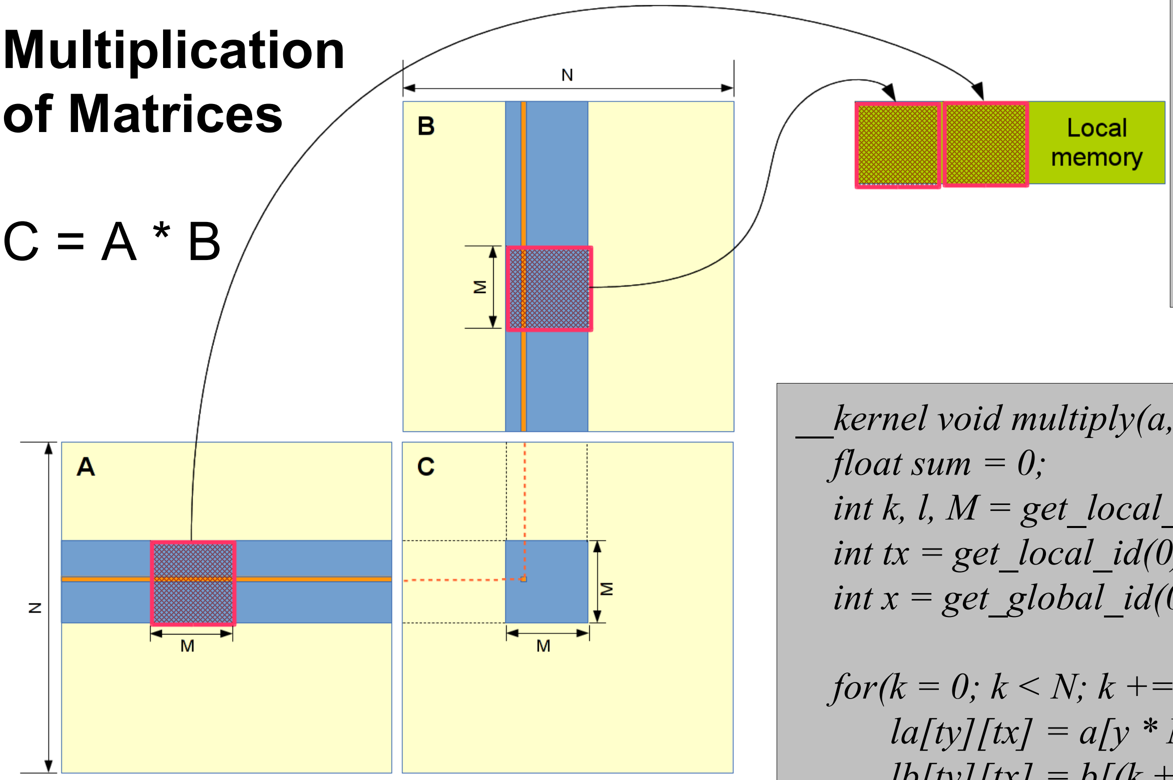
Warp (32 work-items on all NVIDIA devices) is minimal unit of executions. GPU will execute both branches if conditional evaluates differently within warp. On other hand, there is no performance penalty if different wraps select different branches of if-clause

Local memory

Local memory is about 10 times faster than global

Multiplication of Matrices

C = A * B



```
__kernel void multiply(a, b, c, N) {
   float sum = 0;
   int x= get_global_id(0);  int y = get_global_id(1);
   for(int k = 0; k < N; k++)
        sum  += a[y * N + k]  * b[k * N + x];
   c[y * N + x] = sum;
}
```

```
__kernel void multiply(a, b, c, N, __local float *la, __local float *lb) {
   float sum = 0;
   int k, l, M = get_local_size(0);
   int tx = get_local_id(0), ty = get_local_id(1);
   int x = get_global_id(0), y = get_global_id(0);

   for(k = 0; k < N; k += M) {
        la[ty][tx] = a[y * N + (k + tx)];
        lb[ty][tx] = b[(k + ty) * N + x];
        barrier(CLK_LOCAL_MEM_FENCE);

        for (l = 0; l < M; ++l) sum  += la[ty][l]  * lb[l][tx]
        barrier(CLK_LOCAL_MEM_FENCE);
   }
   c[y * N + x] = sum;
}
```

Synchronizing work group

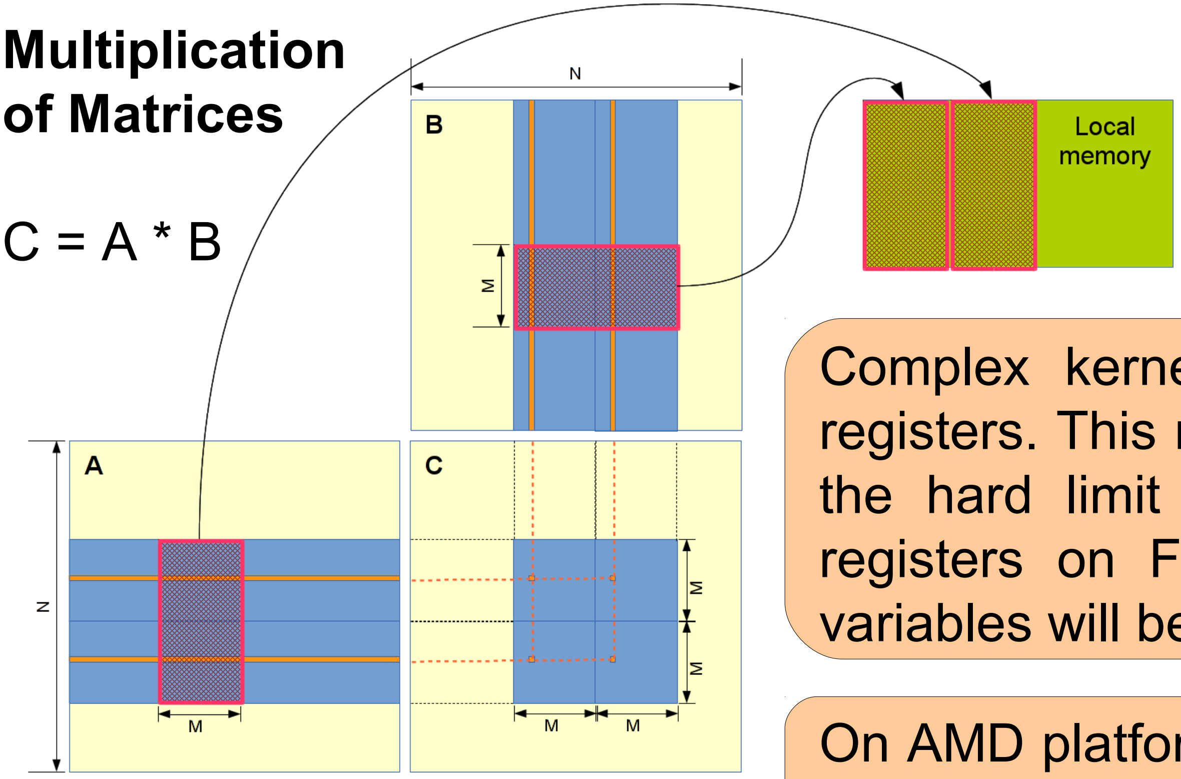
To compute a block of C we will need 2*M*M*N reads from memory, but using local memory we can limit global memory reads to 2*M*N

Complex kernels

Some times it is efficient that a single work-item process several points of output space.

Multiplication of Matrices

C = A * B



We can overcome work-group size limit, by computing multiple items per work-item!

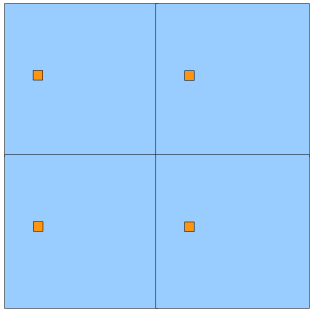
Complex kernels may require big number of registers. This reduces device occupancy and, if the hard limit of registers per work item (63 registers on Fermi) is surpassed, some local variables will be allocated in global memory!

On AMD platform local arrays (*int a[6]*) will be always allocated in the global memory.

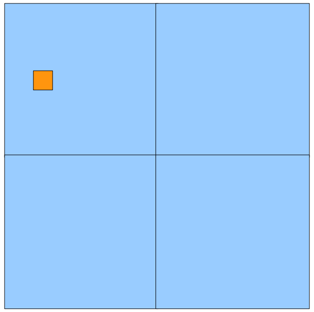
To compute 4 times bigger block, we need only 2 times more global memory reads

Coalescing memory accesses

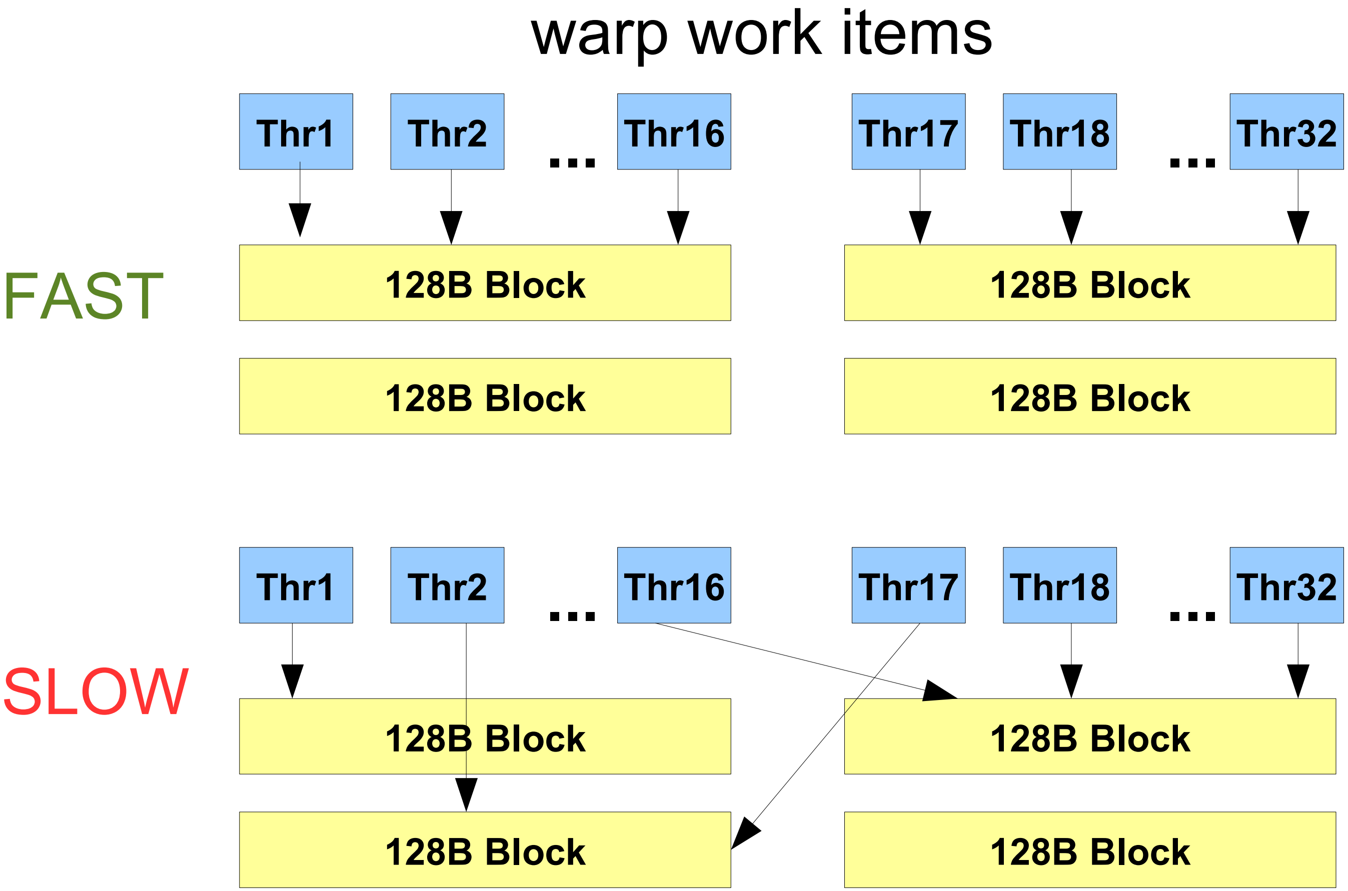
Coalescing accesses to global memory will significantly increase data throughput.



vs.



Why we go first way?



warp work items

FAST

Thr1 Thr2 ... Thr16 Thr17 Thr18 ... Thr32

128B Block 128B Block

128B Block 128B Block

SLOW

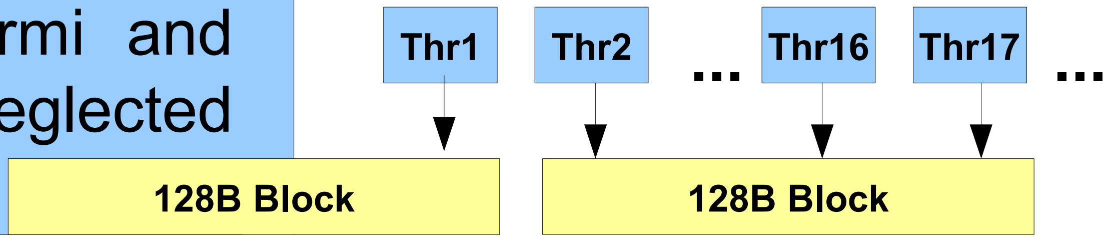
Thr1 Thr2 ... Thr16 Thr17 Thr18 ... Thr32

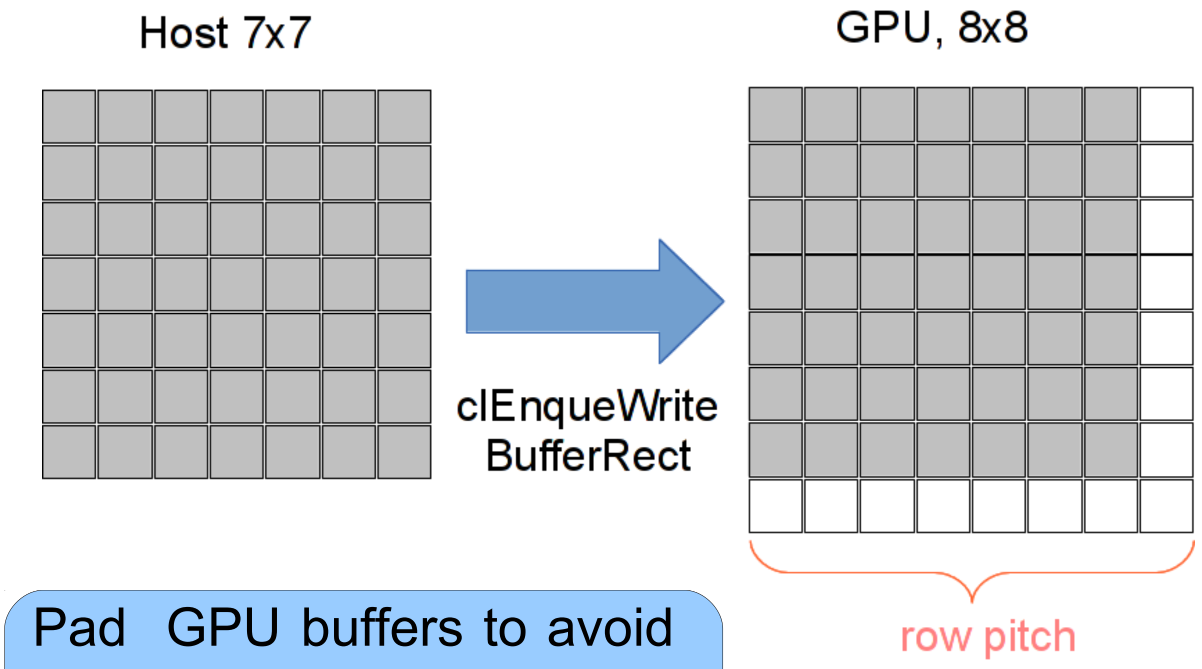
128B Block 128B Block

128B Block 128B Block

Padding 2D arrays to avoid unaligned accesses

Non aligned accesses may also harm performance, though on Fermi and Kepler the effect is mostly neglected by L2 cache





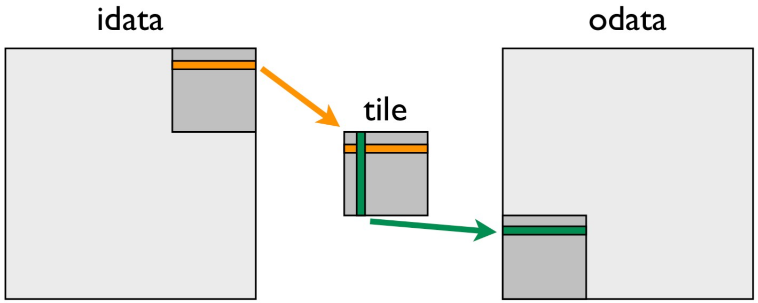
```
const size_t origin[3] = {0,0,0}
const size_t region[3] = {7, 7, 0};
const size_t pitch_gpu = 8 * sizeof(float);
Const size_t pitch_host = 0; // auto
err = clEnqueueWriteBufferRect(queue, dev_in,
            CL_TRUE,
            origin, origin,
            region, pitch_gpu, 0, pitch_host, 0,
            in,
            0, NULL, NULL);
```

Pad GPU buffers to avoid unaligned access and skip edge checking conditionals

Using local memory to optimize global memory performance

```
__kernel void  transpose(__global float *c, __global float *a, int size) {
         int x = get_global_id(0), y = get_global_id(1);
        c[y + x*size] = a[x + y*size];
}
```

Not coalesced



```
__kernel void  transpose(__global float *c, __global float *a, int size, __local float *tile) {
         int x = get_global_id(0), y = get_global_id(1);
        int tx = get_local_size(0), ty = get_local_size(1);

        tile[tx + ty * get_local_size(0)] = a[x + y*width];
        barrier(CLK_LOCAL_MEM_FENCE);

        x = tx + get_group_id(1) * get_local_size(1); y = ty + get_group_id(0) * get_local_size(0);
        c[x + y*height] = tile[ty + tx * get_local_size(1)];
}
```

Synchronizing work group

Fine

Providing local memory to kernel

```
clSetKernelArg(kernel, 3, 256 * sizeof(float), NULL);
```

Local memory banks

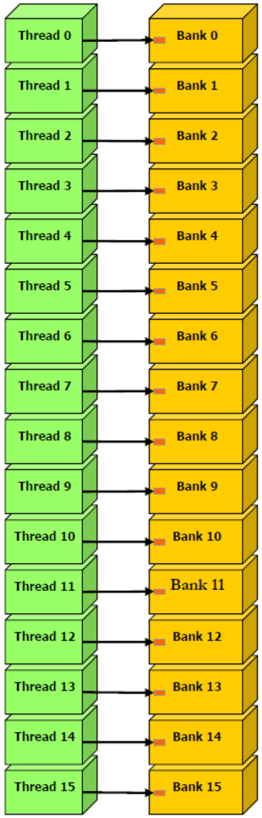
Local memory is divided into equally sized memory modules (banks) that can be accessed in parallel. Successive 32-bit words are assigned to successive banks and each bank has a bandwidth of 32 bits per two clock cycles

```
__kernel void  transpose() {
        ...
        c[x + y*height] = tile[ty + tx * get_local_size(1)];
}
```

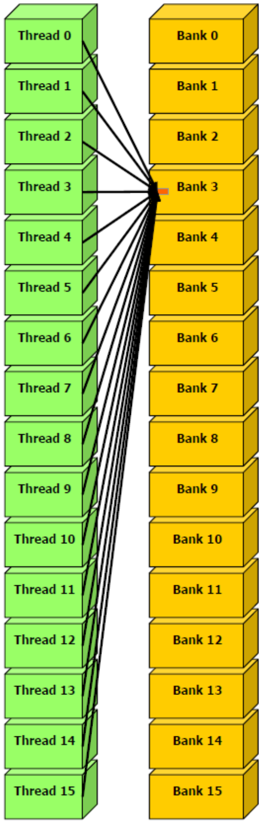
Bad

```
__kernel void  transpose(__global float *c, __global float *a, int size, __local float *tile)
{
         int x = get_global_id(0), y = get_global_id(1);
        int tx = get_local_size(0), ty = get_local_size(1);

        tile[tx + ty * (get_local_size(0) + 1)] = a[x + y*width];
        barrier(CLK_LOCAL_MEM_FENCE);

        x = tx + get_group_id(1) * get_local_size(1);
        y = ty + get_group_id(0) * get_local_size(0);
        c[x + y*height] = tile[ty + tx * (get_local_size(1) + 1)];
}
```

Good



Good

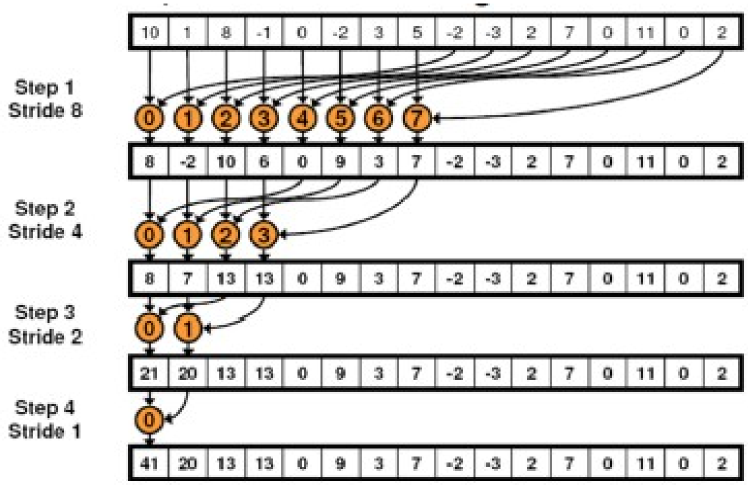


Bad

Reduction

How we sum elements of a vector with GPU?

1. While array is big enough just sum independent parts it with work-items
2. Reduct each work-group to a single value in local memory
3. Reduct to a sum in global memory



```
__kernel void multiply(__global float *res, __global float *a, int size,
                                          __local float *lmem, __local float *gmem) {
   int i;
   float sum = 0, full_sum = 0;
  int item_size = size / get_global_size(0);
 int tid = get_local_id(0);
Int groups = get_num_groups(0);

  for(i = 0; i < item_size; i++)
       sum += a[i * get_global_size(0) + get_global_id(0)]
  lmem[tid] = sum;

  barrier(CLK_LOCAL_MEM_FENCE);
  for(i = get_local_size(0)/2; i>0; i >>= 1) {
       if (tid < i) lmem[tid] += lmem[tid + i];
       barrier(CLK_LOCAL_MEM_FENCE);
  }

  if (tid == 0) gmem[group] = lmem[0];
  barrier(CLK_GLOBAL_MEM_FENCE);

  if (get_global_id(0) == 0) {
       for (i = 0; i < groups; i++) full_sum += gmem[i];
        *res = full_sum;
  }
}
```

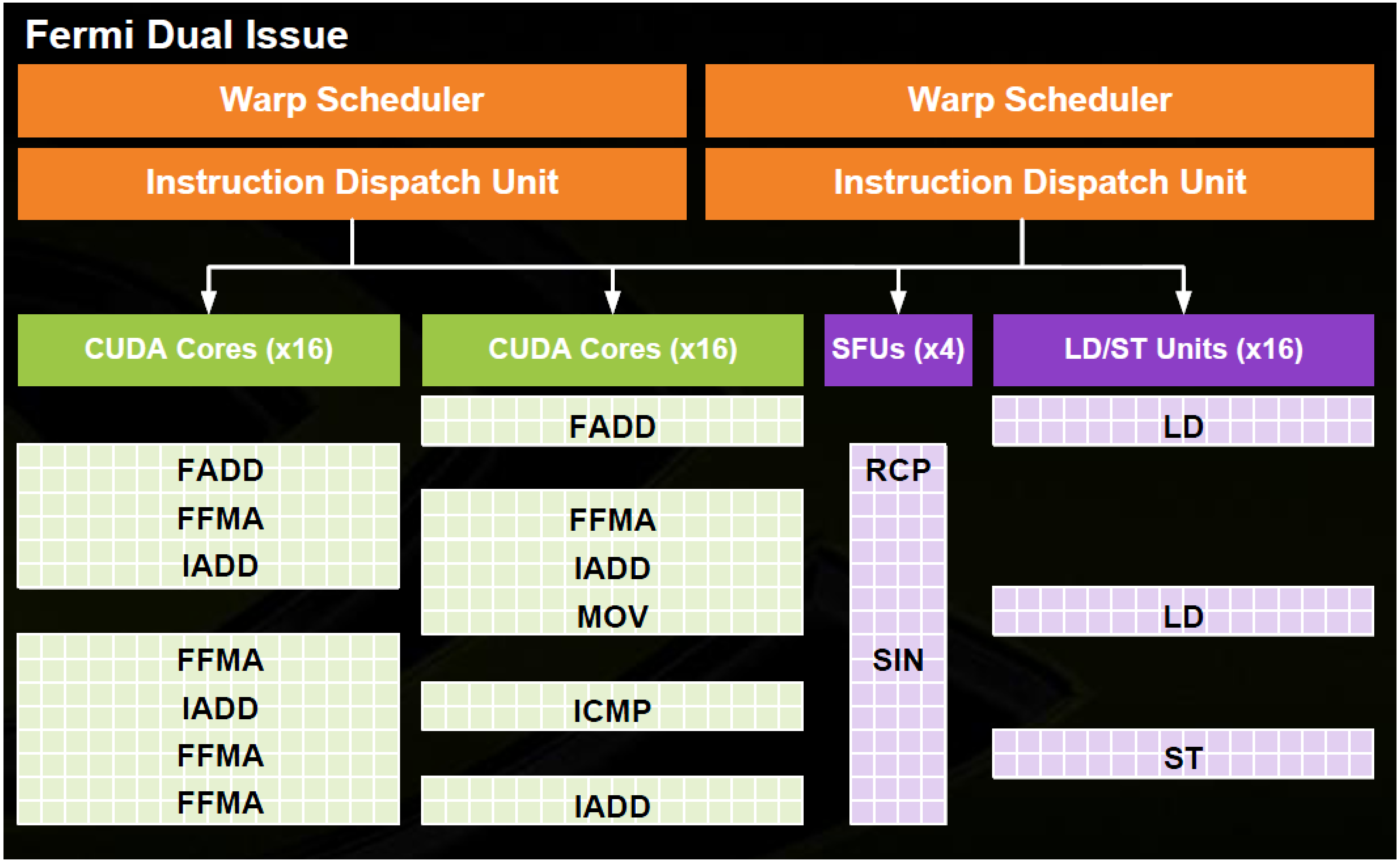
Coalesced access

Less bank conflicts

Atomics may be used for integer types

Scheduler of Fermi Compute Unit

There is even more parallelism when 32 cores per CU on Fermi



While one wrap is waiting for LD from global memory to complete other wraps may execute computations. This is used to hide memory access latencies.

Texture Engine

Features:

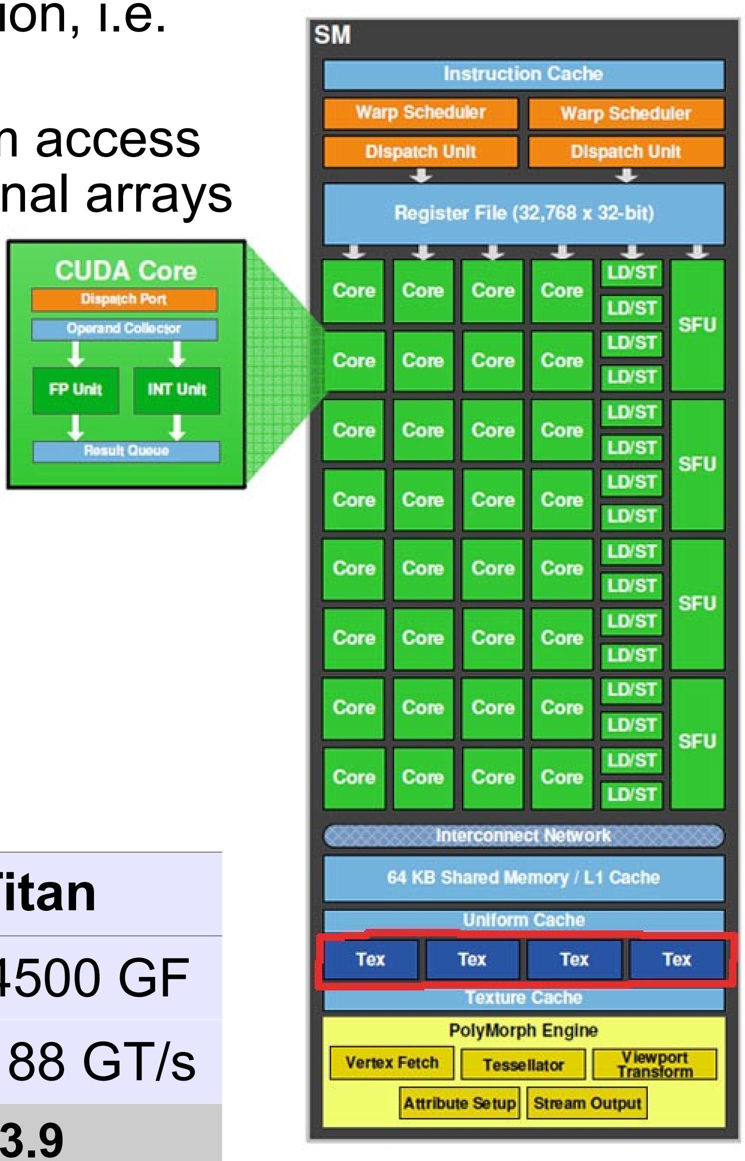
- Spatial-aware cache
- Bi/tri-linear interpolation
- Normalized coordinates
- Different clamping modes

Uses:

- Linear interpolation, i.e. image scaling
- Optimize random access to multidimensional arrays

Texture engine is accessed using LD/ST units, but it performs some computations as well (interpolation). On compute bound tasks this may be used to get extra performance.

	GT280	GTX580	Titan
Core Throughput	930 GF	1580 GF	4500 GF
Texture Fill Rate	48 GT/s	49 GT/s	188 GT/s
Ratio	**19.3**	**31.6**	**23.9**



Streaming Multiprocessor (SM)

Hiding the memory latencies

How to hide latencies?

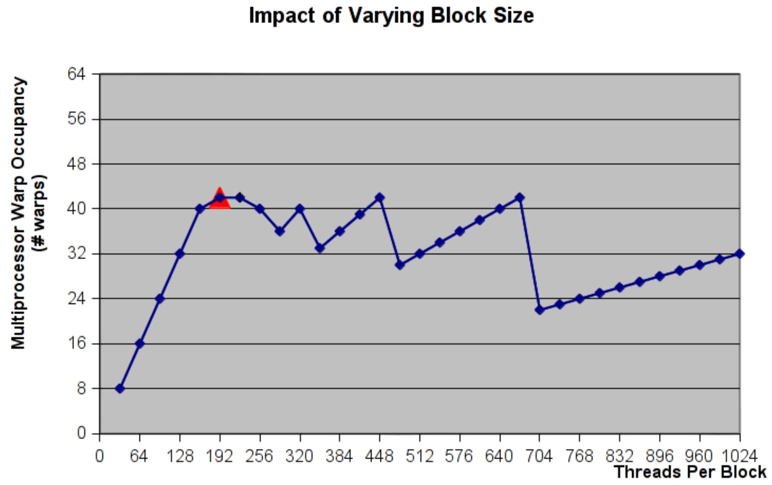
- **More active wraps per compute unit**
- **More independent instructions in the queue**
  - Some architectures (AMD VLIW) actually rely on flow of independent instructions to fully utilize hardware compute resources

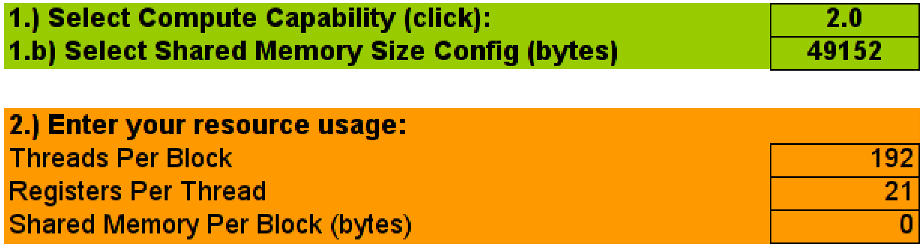
What limits number of active wraps?

- **The work-group size**
  - Fermi supports up to 48 active wraps per CU, but limited to 8 active work-groups. So, if there is less than 192 work-items in the group (i.e. 6 full wraps), the full occupancy will be impossible to achieve
- **Used local memory**
  - Fermi has up to 48 KB of shared memory per CU. High shared-memory usage (above 6 KB per group) will limit maximum number of active work-groups. However, this may be compensated by increased work-group size.
- **Used registers**
  - Fermi CU has 32k 4-byte registers per CU. High register usage (more than 20 registers) will limit maximum number of active wraps.

CUDA Occupancy Calculator

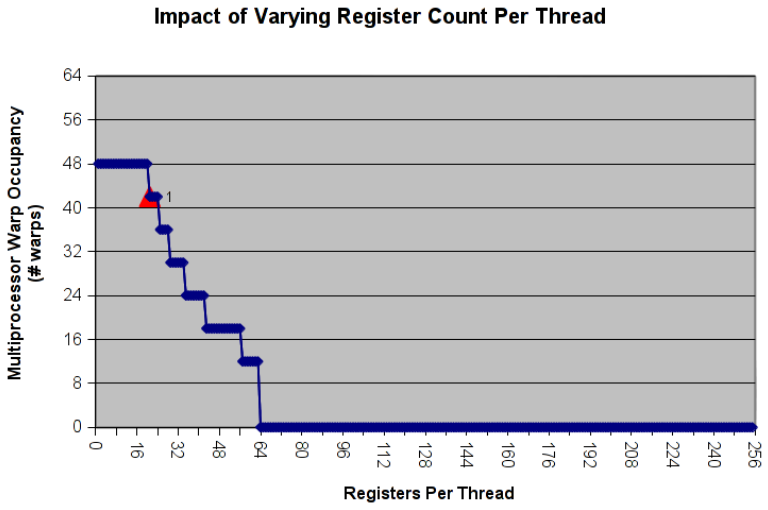
As well useful for OpenCL code run on NVIDIA hardware





1.) Select Compute Capability (click):	**2.0**
1.b) Select Shared Memory Size Config (bytes)	**49152**

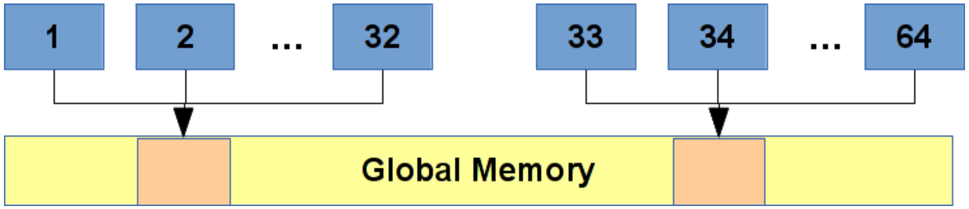
2.) Enter your resource usage:	
Threads Per Block	192
Registers Per Thread	21
Shared Memory Per Block (bytes)	0



Important to select optimum work-group size

Constant & Texture Memories

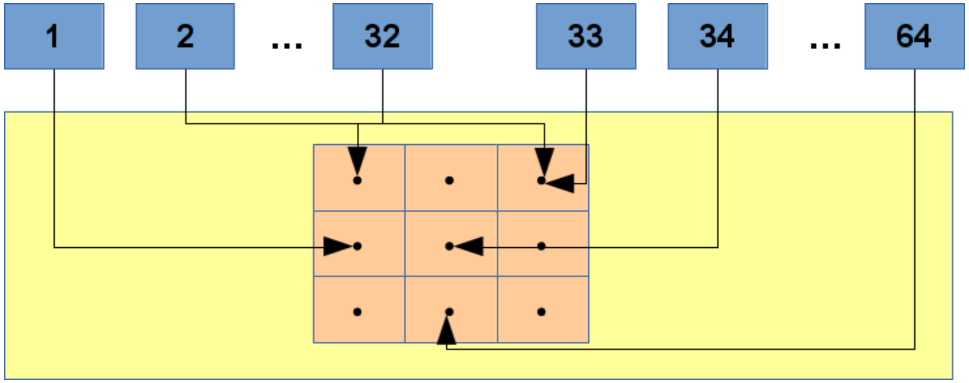
Constant Memory



Optimized for work-items reading from the same memory location

```
__kernel void  multiply(float *out,
        const float *in,  __constant float *params) {
        out[id] = in[id] * param[0];
}
```

Texture Memory



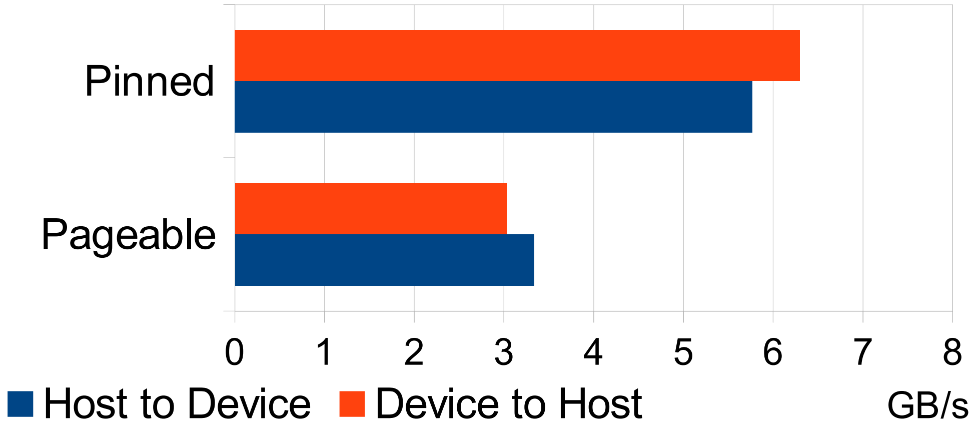
```
cl_image_format format;
format.image_channel_order = CL_R;
format.image_channel_data_type = CL_FLOAT;
cl_mem img = clCreateImage2D(ctx, CL_MEM_READ_ONLY,
                        &format, size, size, 0, NULL, &err);

size_t origin = {0, 0, 0}, region = {size, size, 1}, pitch = 0;
err = clEnqueueWriteImage(queue, img, CL_TRUE,
                origin, region, pitch, 0, data, 0, NULL, &event);
```

Optimized for 2D spatial locality

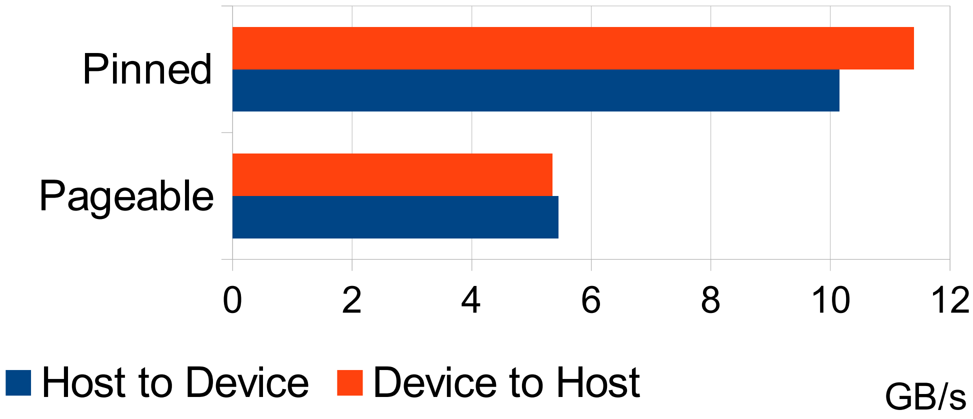
```
const sampler_t sampler = CLK_FILTER_LINEAR
                | CLK_NORMALIZED_COORDS_TRUE
                | CLK_ADDRESS_CLAMP_TO_EDGE;

__kernel void  scale(float *out,  __read_only image2d_t in) {
        int id = (get_global_id(1) * get_global_size(0)
                + get_global_size(0);

        float2 src = (float2)(
                1 . * get_global_id(0) / get_global_size(0),
                1 . * get_global_id(1) / get_global_size(1)
        )

        float4 val = reade_imagef(in, sampler, src);
        out[id] = val.x;
}
```

Page-locked vs. Page-able memory

GTX590 (gen2)



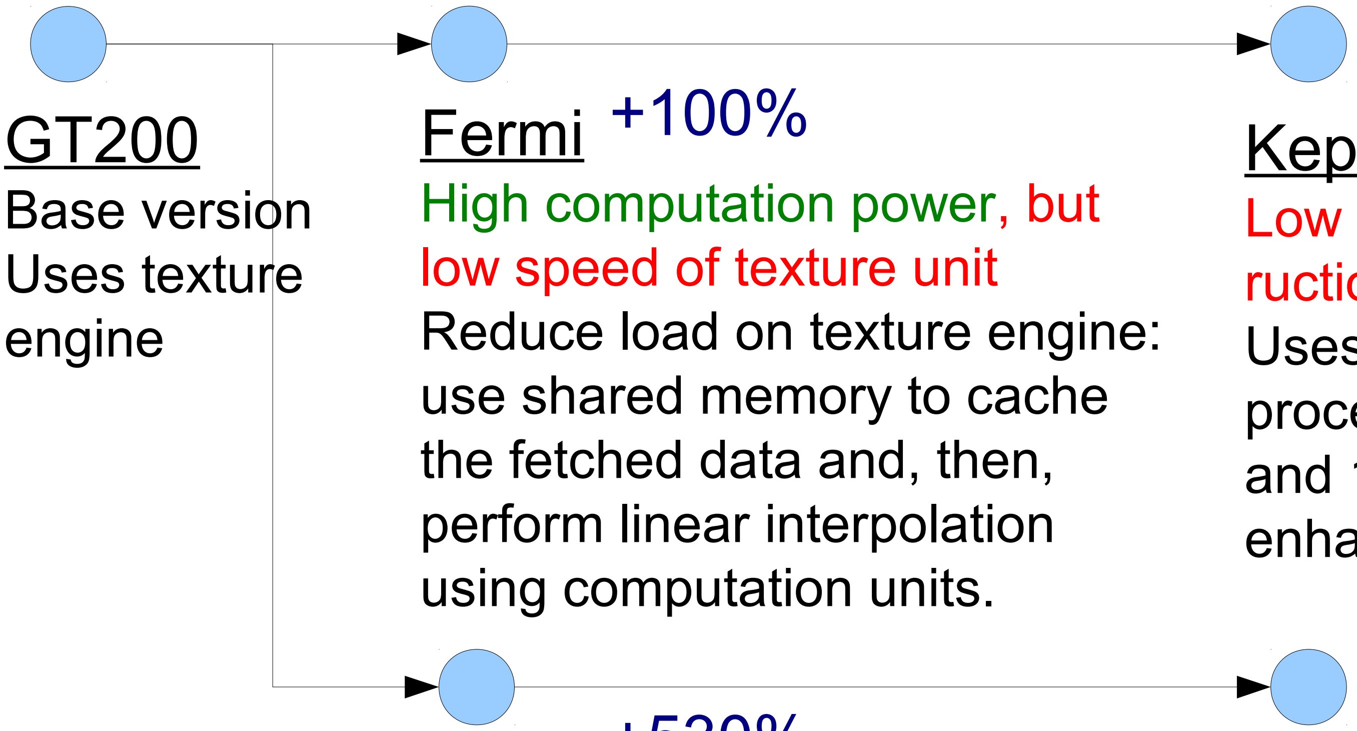
AMD HD7970 (gen3)



- Two times faster transfer rates between host and device
- Some devices support overlapping of data transfers from/to page-locked memory and kernel execution. For compute-bound problems you will not see the data transfers at all.
- There is no concept of page-locked memory in OpenCL. However, NVIDIA suggests to allocate using clCreateBuffer as below
- This also works on AMD, but on AMD platform such allocations reserve the memory on GPU devices and maximum possible allocation will be limited by the memory available on GPU.

```
cl_mem mem = clCreateBuffer(ctx, CL_MEM_READ_WRITE | CL_MEM_ALLOC_HOST_PTR, size, NULL, &err);
float *ptr = (cl_float*)clEnqueueMapBuffer(queue, mem, CL_TRUE, CL_MAP_WRITE, 0, size, 0, NULL, NULL, &err);
```

Tuning tomography for hardware architectures



GT200

Base version
Uses texture
engine

Fermi +100%

High computation power, but
low speed of texture unit

Reduce load on texture engine: use shared memory to cache the fetched data and, then, perform linear interpolation using computation units.

Kepler +75%

Low bandwidth of integer instructions, but high register count

Uses texture engine, but processes 16 projections at once and 16 points per thread to enhance cache hit rate

VLIW +530%

Executes 5 independent operations per thread

Computes 16 points per thread in order to provide sufficient flow of independent instructions to VLIW engine

GCN +95%

High performance of texture engine and computation nodes

Balance usage of texture engine and computation nodes to get highest performance

Summary

- Decide which precision is required. Do you really need double precision? Do you need IEEE 754 compliance?
- **First get a simple version working, than profile and start optimizing**
- Use page-locked memory and multiple command queues to allow parallel execution of multiple kernels and data transfer overlapping
- Estimate optimal work-group size and result-space per work-item
- Use local memory to optimize usage of global memory and think how usage of work items may be re-arranged during different stages of the kernel execution
- Remember about global memory coalescing and local memory banks
- Use texture engine to optimize caching of randomly accessed arrays
- Try to provide flow of independent instructions
- **Even if you plan to use OpenCL and AMD GPUs, read CUDA documentation CUDA Programming Guide / Best Practices. Understand the samples provided with NVIDIA and AMD SDKs.**