Real-time Graphics

9. GPGPU

Juraj Starinský
Martin Samuelčík

Overview

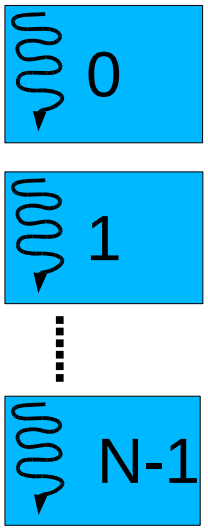
- Real-Time rendering motivation
- Parallel and SIMD processing
- Short GPU history
- GPU architecture
- CUDA & data processing
- Example

Real-Time rendering motivation

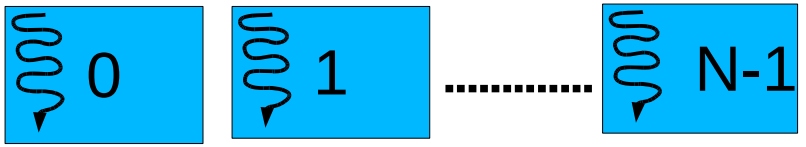
- Some rendering algorithms require special features
  - Various deformations
  - Correct order of dynamic geometry
    - Sorting
      - dynamic alpha-blended particles
  - Physics simulations
    - Collision detection
    - Cloth
- Can be done via shaders, but requires hacking

Serial vs Parallel processing

- void process()
- {
  - for (int i=0;i<N;i++)
    - calculate(i);
- }



- void process_in_parallel()
- {
  - start_threads(N);
- }
- void thread()
- {
  - calculate(thread_ID);
- }



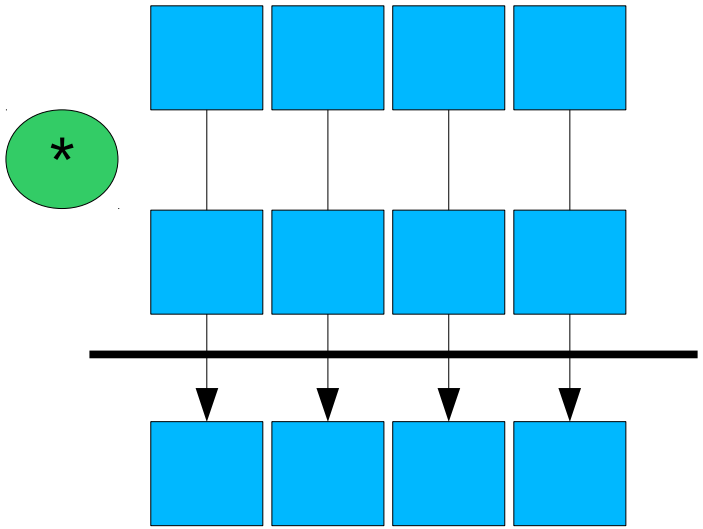
SIMD processing

- Single Instruction Multiple Data
- CPU x86 - special instructions + registers

–1997 Intel MMX
  - 8 x 8 bit Integer
  - 4 x 16 bit Integer
  - 2 x 32 bit Integer

–1998 AMD 3DNow!
  - 2 x 32bit FP

–1999 Intel SSE
  - 4x32 bit FP
  - 2x64 bit FP



Short GPU history

- primarily for rasterization and texturing
- data types – SIMD processing
  - vertex – 4 x 32bit FP
  - fragment (pixel) – 4 x 8bit FP/integer
  - texel – 4 x 8bit integer
- increasing number of processing units (pipelines)
  - separate pipelines for vertices and fragmets
- increasing programmability of pipelines (shader units)
  - conditional instructions
  - loops
- support for 32bit FP texels and fragments

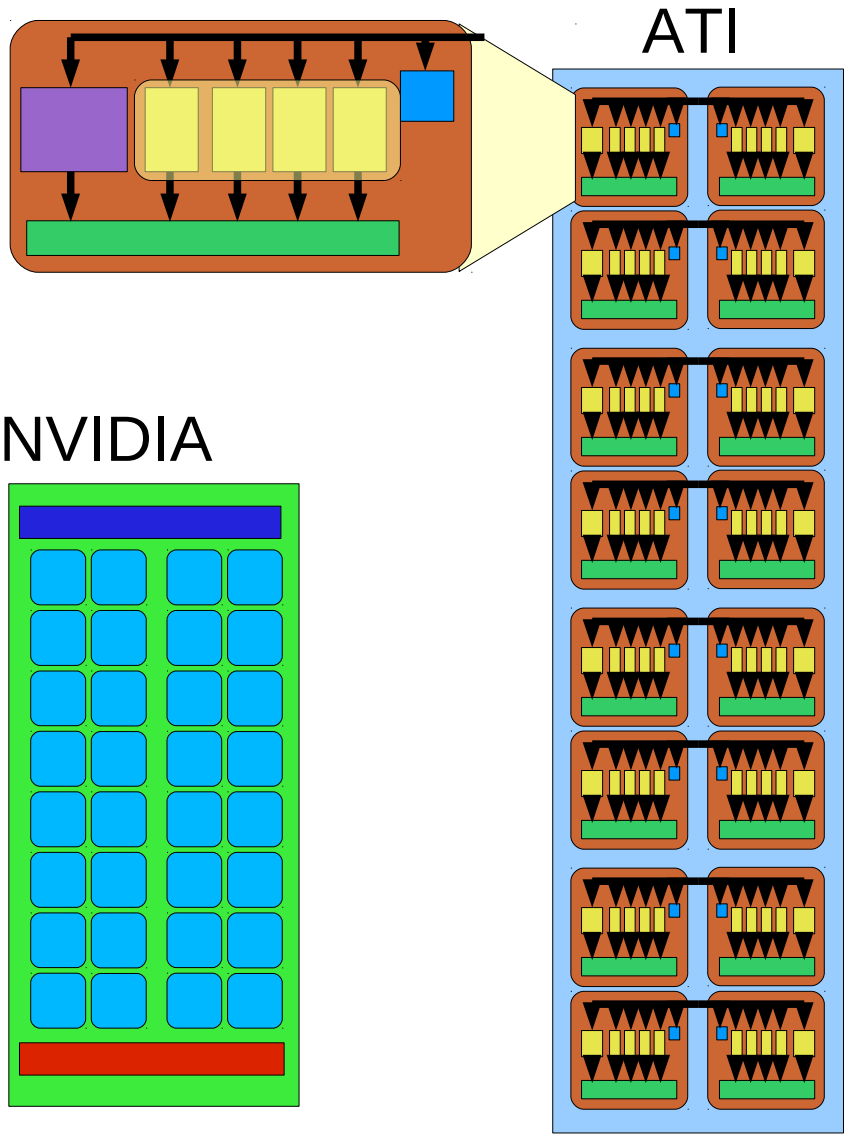
Unified shader
(since November 2006)

- unification of shader units
  - same instruction set for vertex and fragment processing
  - close to CPU arithmetic programmability
- #processing units (cores) – 60+
  - NVIDIA 8800 - 16 x (8 x 1 FP)
  - ATI HD 2000 - 4 x (16 x 4 FP)
- programmable via new technologies
  - NVIDIA – CUDA
  - ATI – ATI Stream

GPU architecture 1/4

Multiprocessor / SIMD Engine

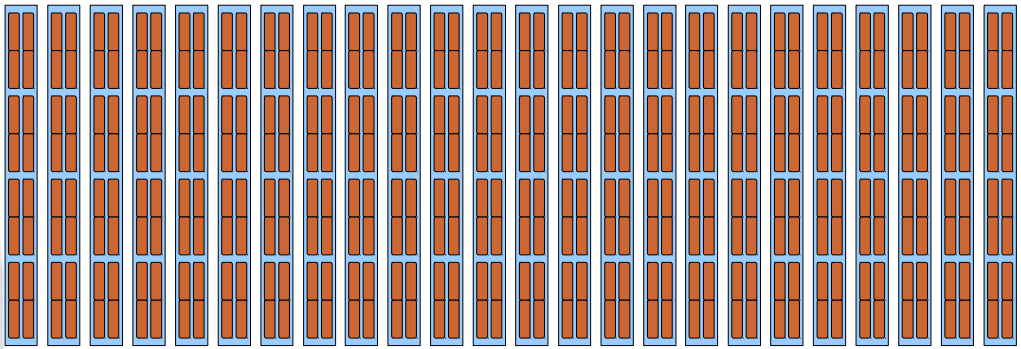
- thread
  - running computation
  - one thread on one core
- multiprocessor/SIMD engine
  - all cores execute the same instruction – SIMD
- threads may share data inside fast memory (with some restrictions)



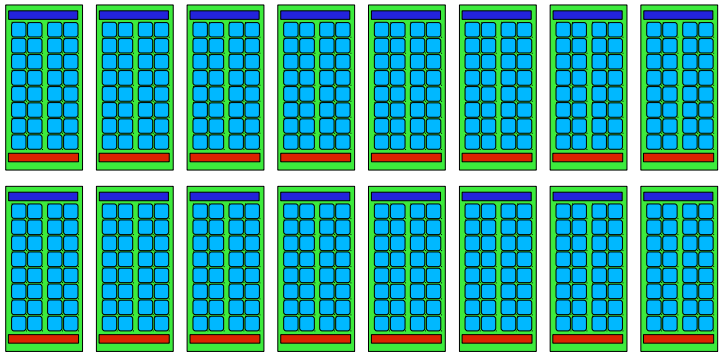
GPU architecture 2/4

Device

- ATI HD 6970
- 2048MB RAM
- 24x SIMD engines
  - independent
  - 24x16=384 FP64
  - 24x64=1536 FP32
- 300+ euro



- NVIDIA GT 580
- 1536MB RAM
- 16x Multiprocessors
  - independent
  - 16x16=256 FP64
  - 16x32=512 FP32
- 430+ euro



Cheapest pieces

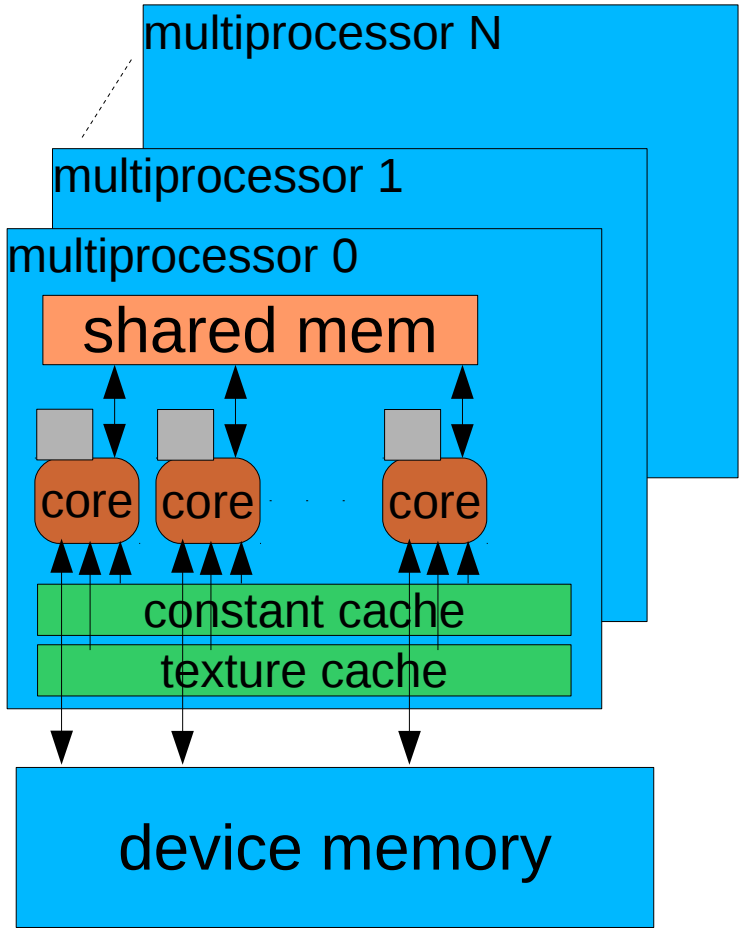
- ATI HD 5450
- 512 DDR3
- 2x SIMD engines
  - independent
  - 2x8=16 FP64
  - 2x8x4=64 FP32
- ~30 euro

- NVIDIA GeForce210
- 512 DDR2
- 2x Multiprocessors
  - independent
  - 0 FP64
  - 2x8=16 FP32
- ~30 euro

GPU Architecture 3/4

Memory Model

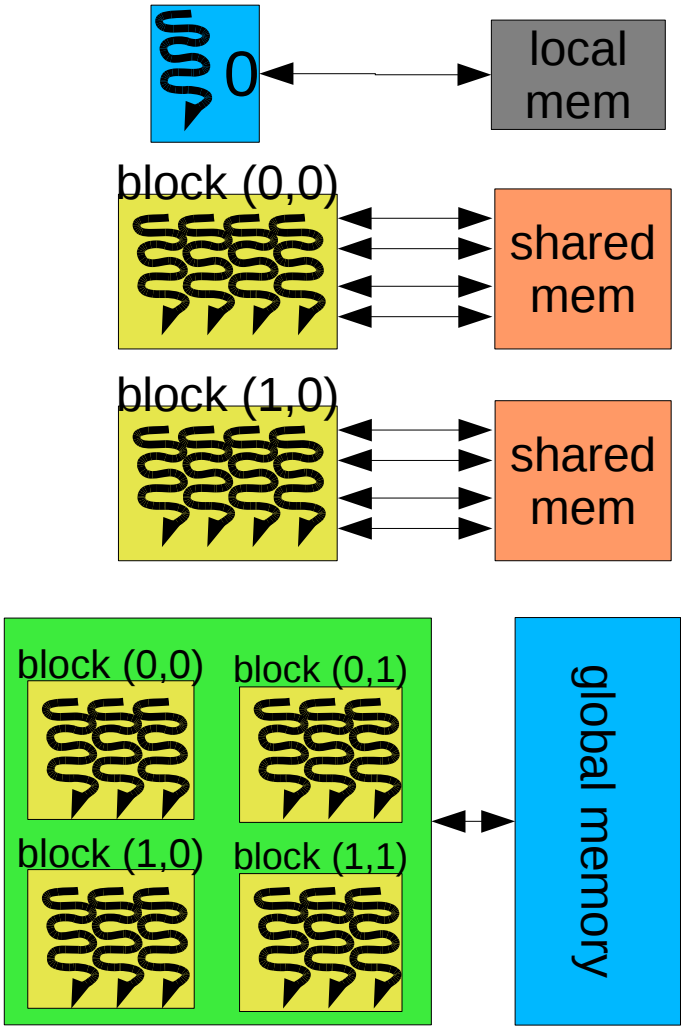
- Global memory
  - per device 512-2048 MB
- Constant memory
  - per device = 64 KB
  - cache per MP = 8 KB
- Shared memory
  - per MP = 16 KB – very fast
- Registers
  - per MP = 16384



GPU Architecture 4/4

Computation model

- kernel – thread program
- thread per core
  - threadID
  - acces to per-thread local memory
- block of threads per multiprocessor
  - blockID
  - blockDim - # of threads in block (3D)
  - access to per-block shared memory
- Grid of blocks per device
  - gridDim - # of blocks (2D)
  - access to global memory

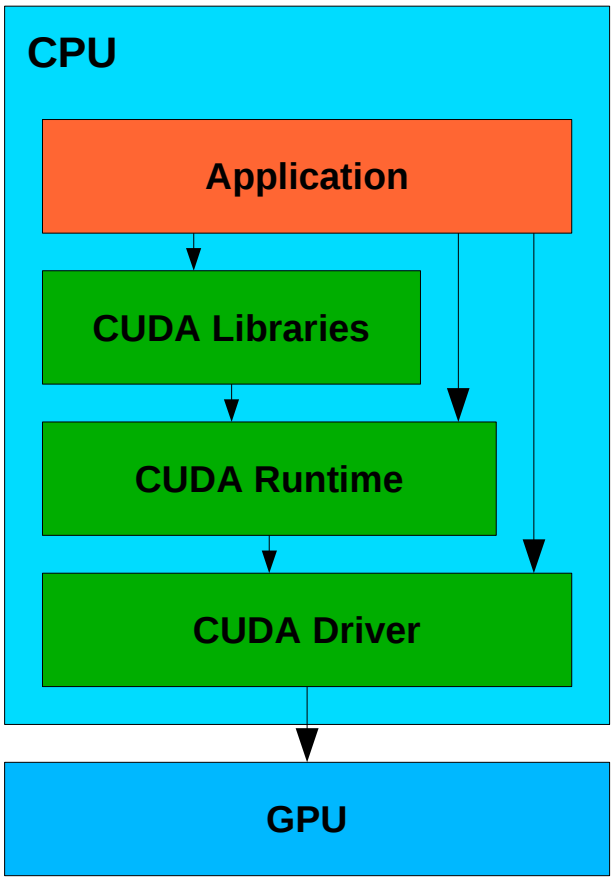


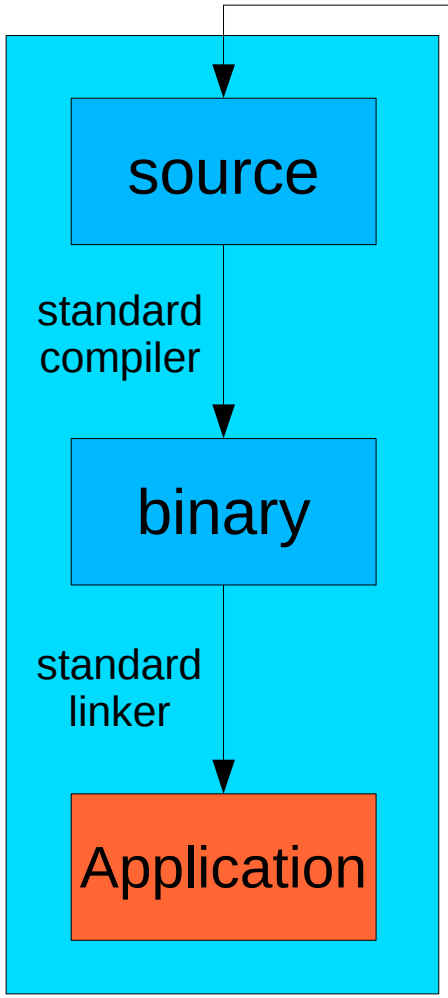
Programming possibilities

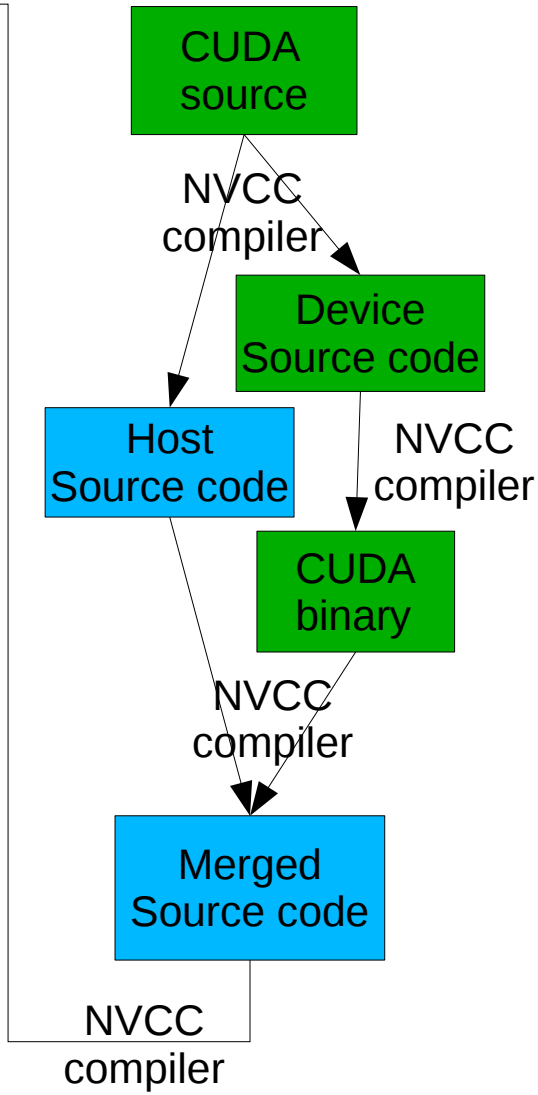
- graphics API
  - OpenGL (universal)
    - GLSL
    - Cg
    - ASM
  - DirectX (win32)
    - HLSL
    - Cg
    - ASM
- Computing languages
  - Nvidia CUDA (universal)
    - CUDA
    - PTX ISA
  - ATI Stream (universal)
    - Brook+
    - AMD IL
    - ISA
  - OpenCL (universal)
    - all devices
- DX11 Compute Shader (win7)
  - all GPUs

CUDA

Compute Unified Device Architecture







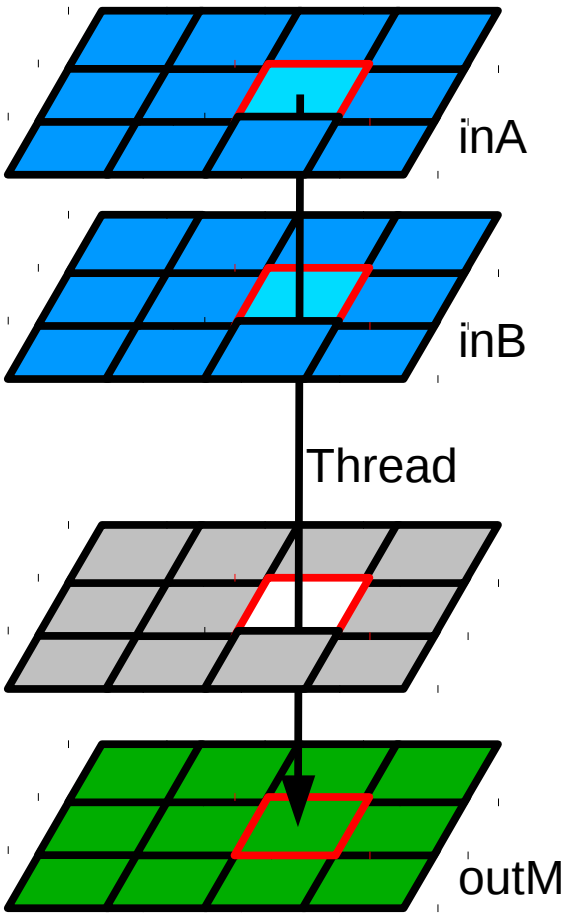
CUDA

C/C++ language extension

- device vs host code
  - function qualifiers
    - __device__, __host__, __global__
  - variable qualifiers
    - __device__, __constant__, __shared__
- built-in vars
  - threadIdx
  - blockDim - #of threads in block
  - blockIdx
  - gridDim - #of blocks in grid

Simple kernel
Matrix Add

- Kernel
  - __global__ void matAdd (float outMat[M][N],
    - float inMatA[M][N],
    - float inMatB[M][N])
  - {
    - int col = threadIdx.x;
    - int row = threadIdx.y;
    - outMat[row][col] = inMatA[row][col] +
      - inMatB[row][col];
  - }
- Execution
  - dim3 gridDim={1,1,1};
  - dim3 blockDim={M,N,1};
  - matAdd <<< blockDim, gridDim >>> (outM, inA, inB);

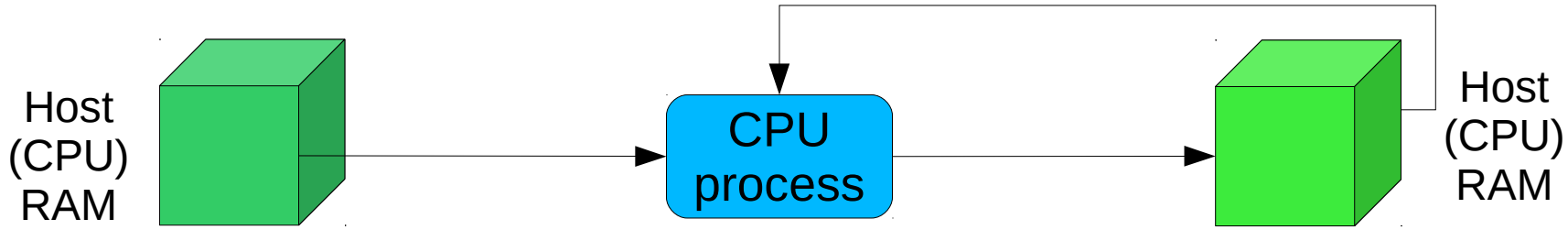


Graphics API vs CUDA

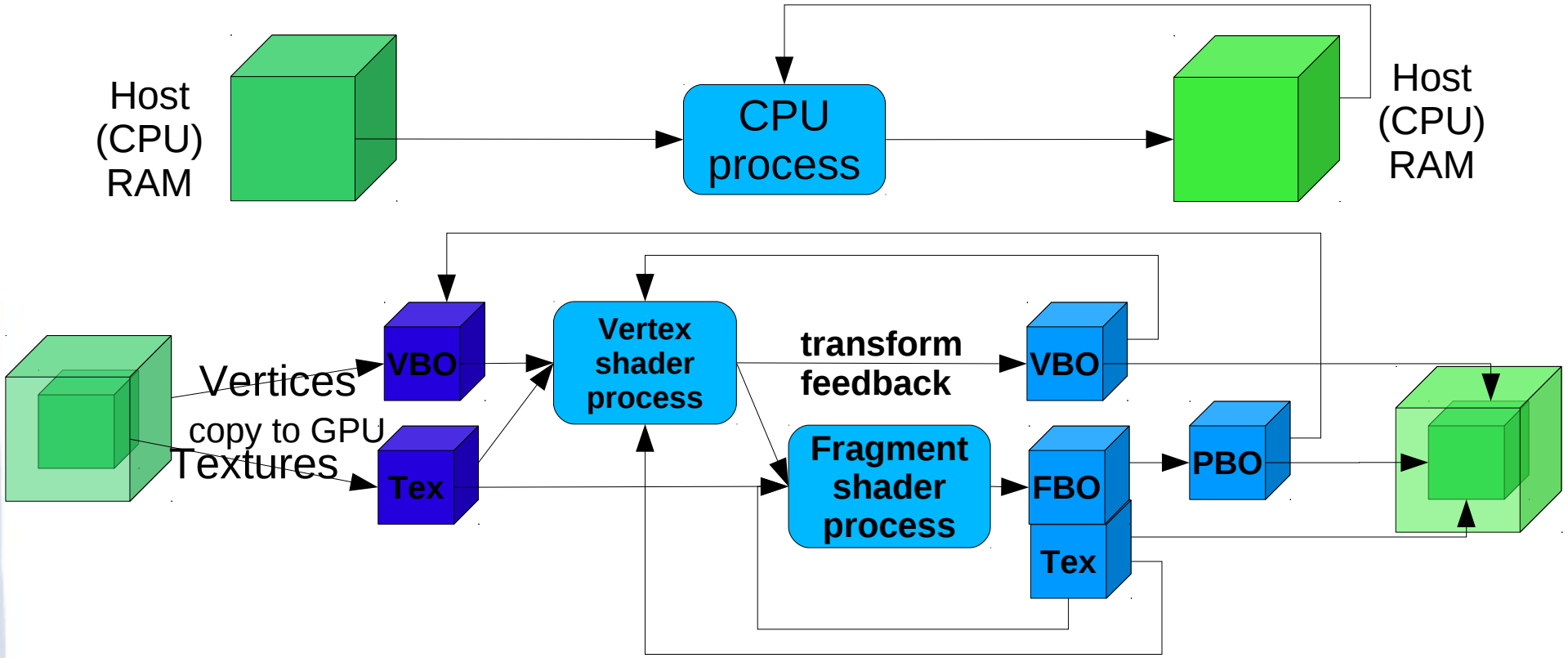
- graphics API + shading language
- limited output
- pre-specified location
  - output format
- data storage
  - Textures - HW limited resolution
- no shared memory
  - shader compiler may utilize it
- utilize non-programmable paths
  - earlyZ, rasterization, ...

- C/C++ like
- unlimited scatter write
- direct access to memory
- utilize shared memory
- non-programmable paths unavailable

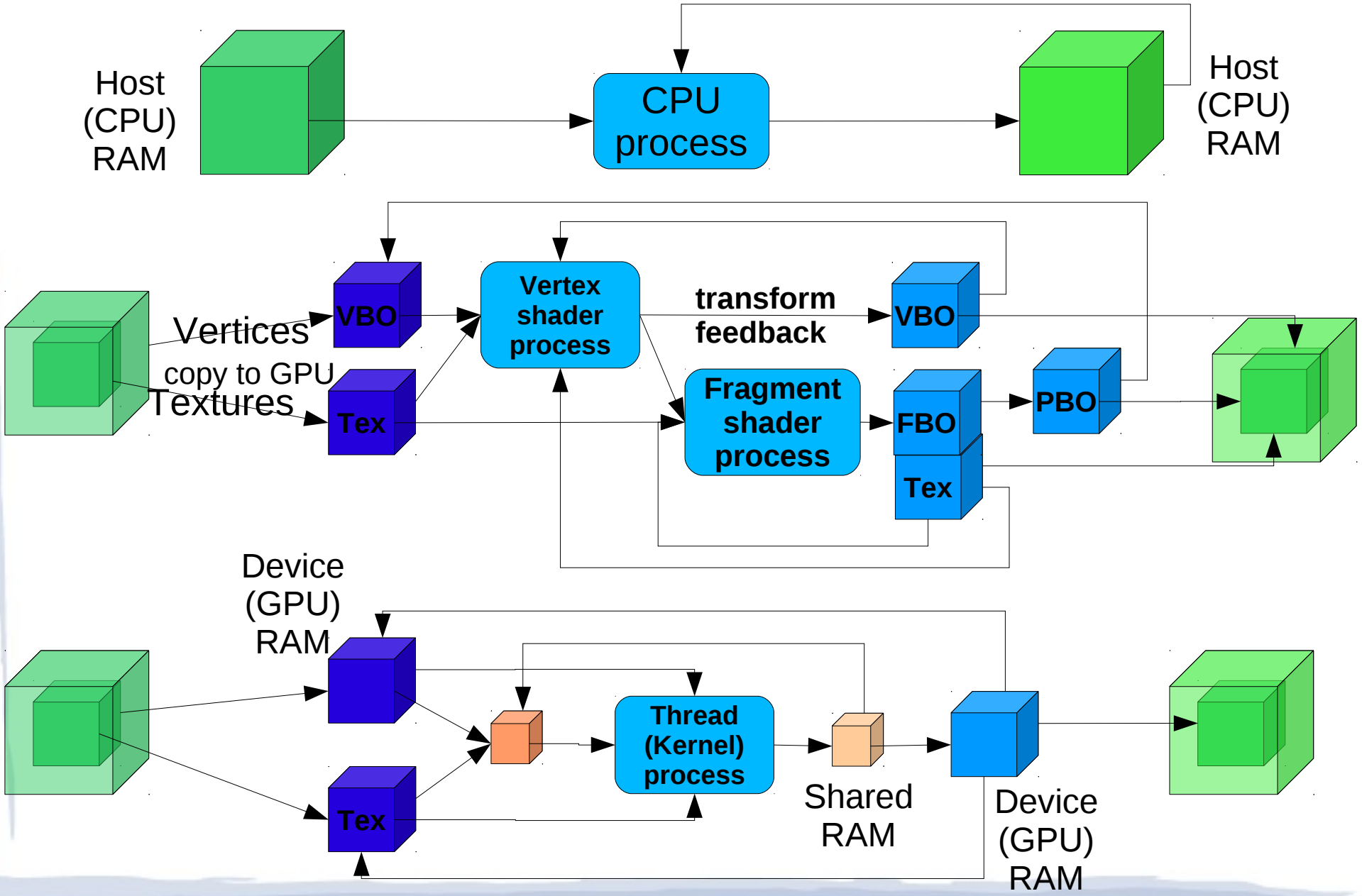
Data Processing



Data Processing

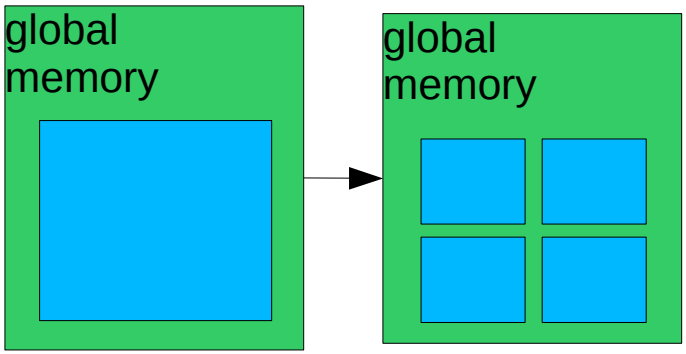


Data Processing



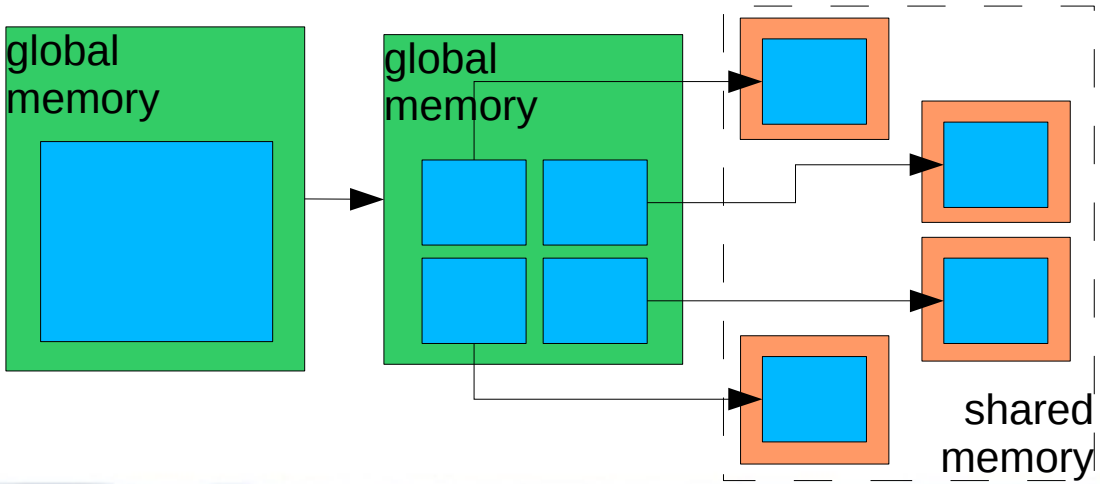
Data Processing

- divide data into blocks



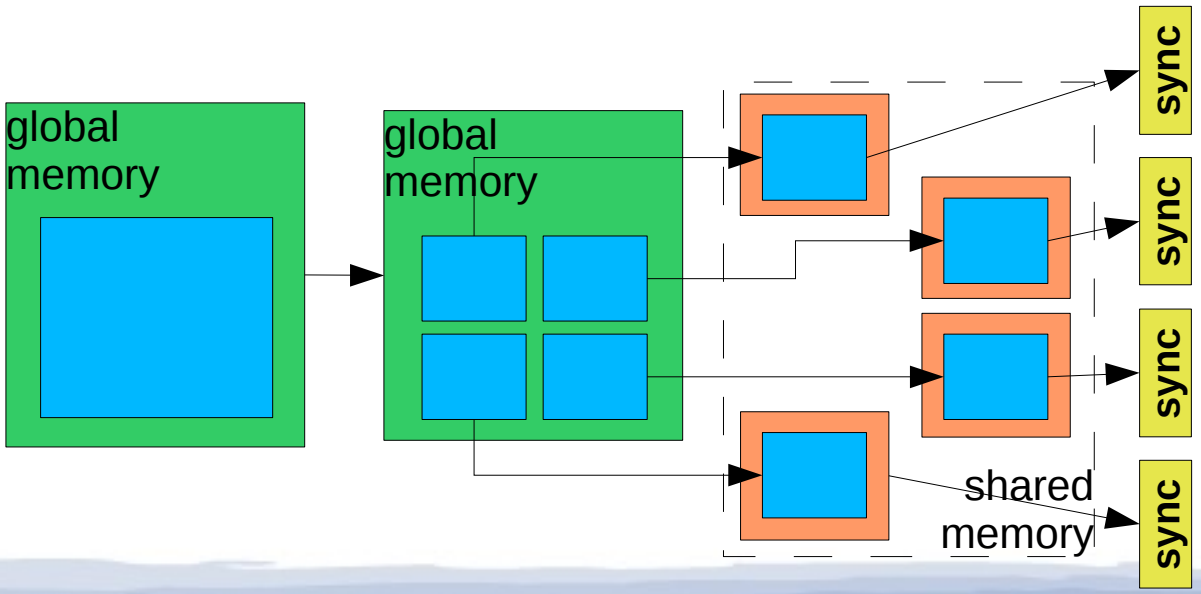
Data Processing

- divide data into blocks
- per-block
  - read into shared memory



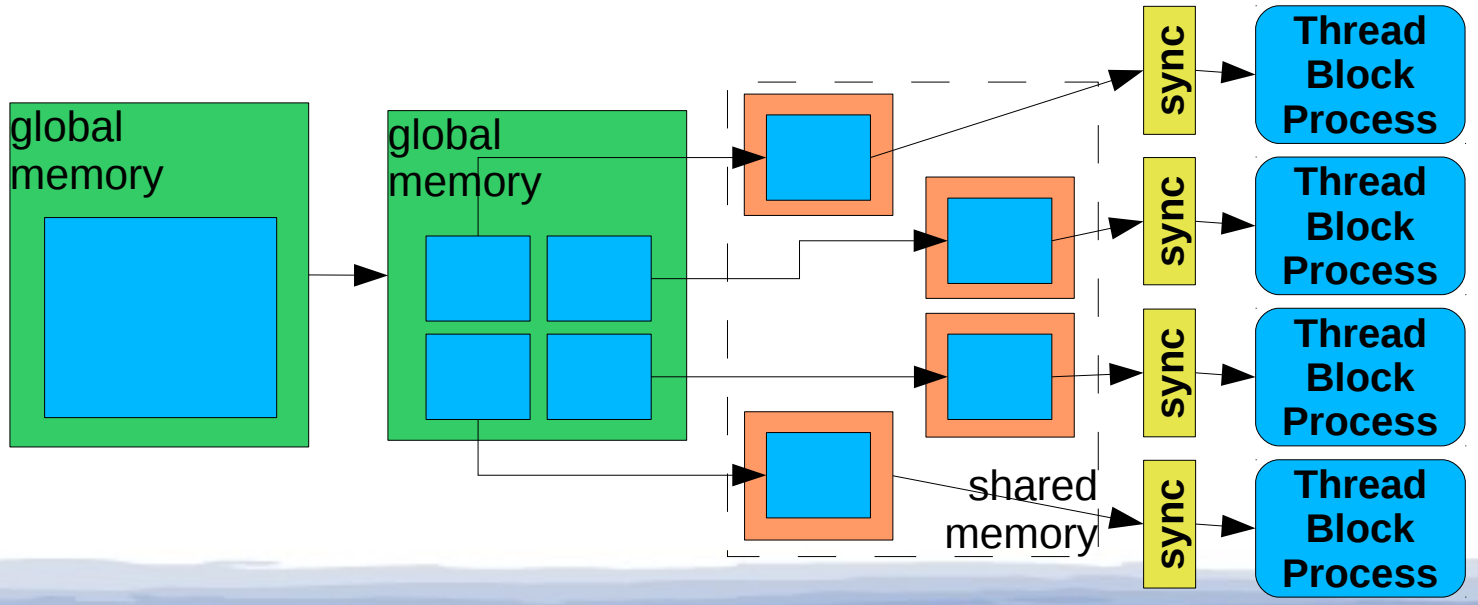
Data Processing

- divide data into blocks
- per-block
  - read into shared memory
  - synchronize to ensure all data in place



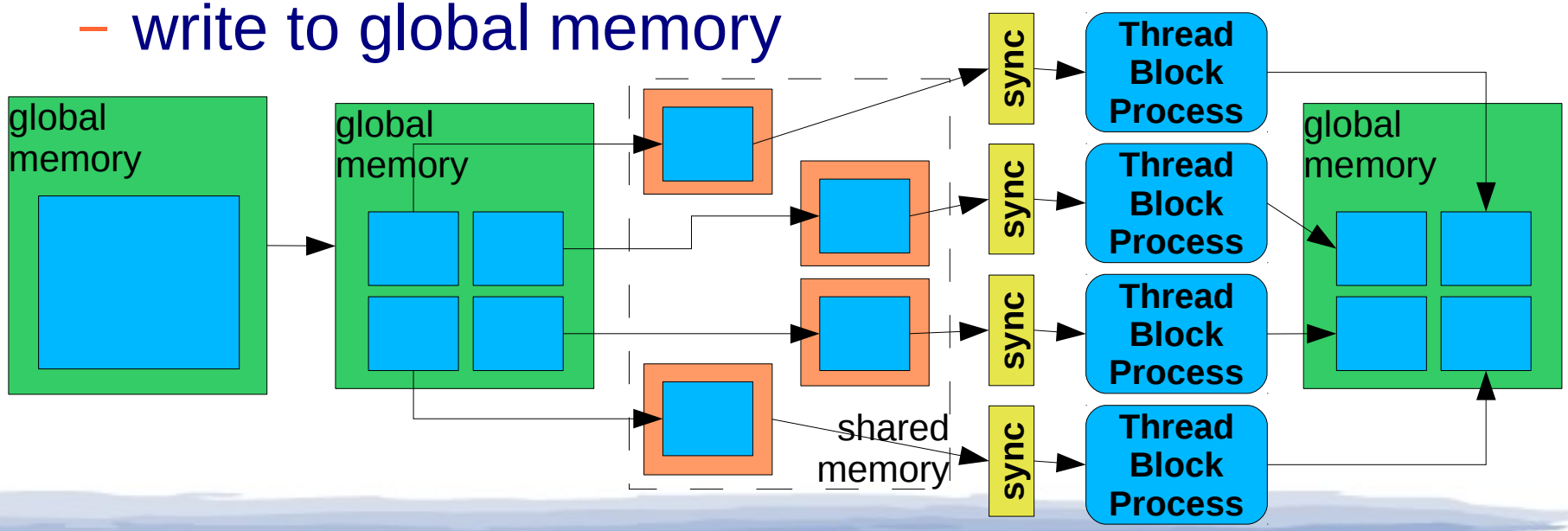
Data Processing

- divide data into blocks
- per-block
  - read into shared memory
  - synchronize to ensure all data in place
  - process



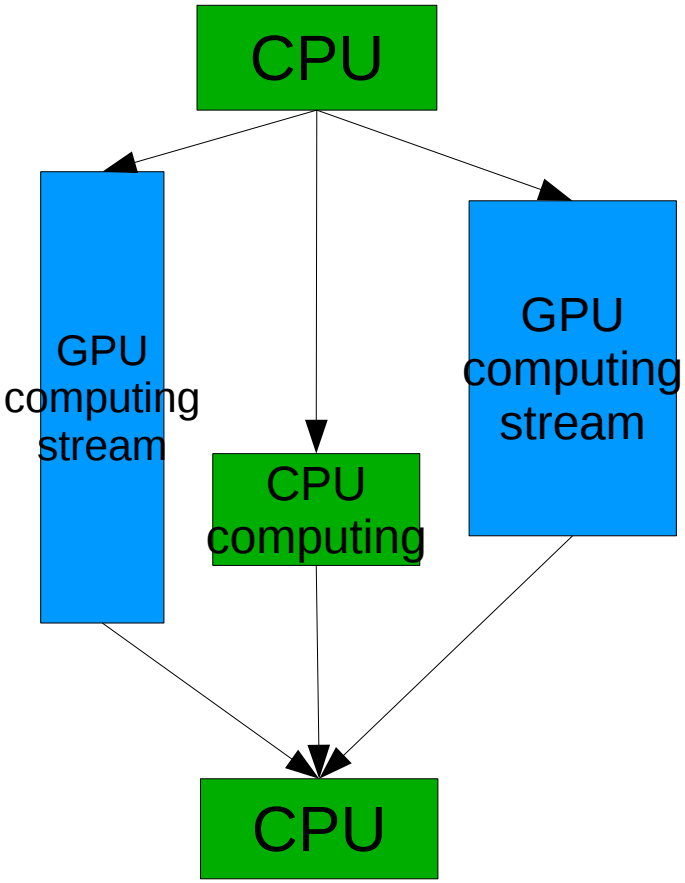
Data Processing

- divide data into blocks
- per-block
  - read into shared memory
  - synchronize to ensure all data in place
  - process
  - write to global memory



Multiple parallel computations
on one device

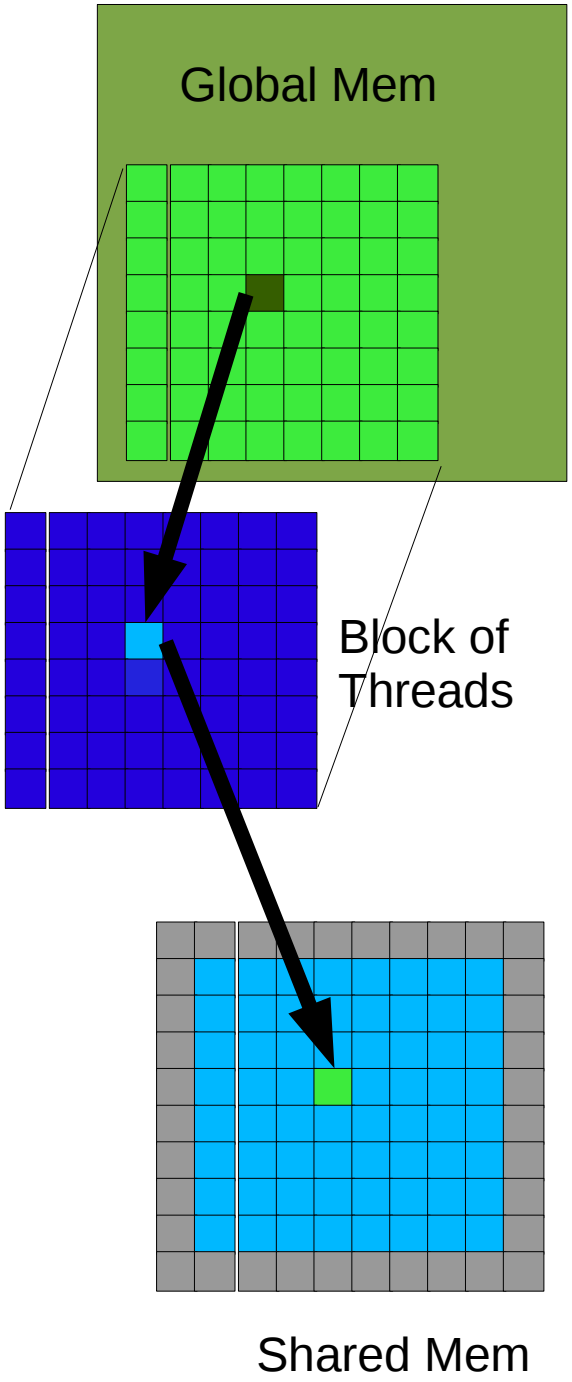
- asynchronous operations
  - copy CPU <-> GPU
  - GPU processing
  - CPU is free to work
- parallel async operations
  - multiple computing streams
  - own copying & processing



Example

normal map

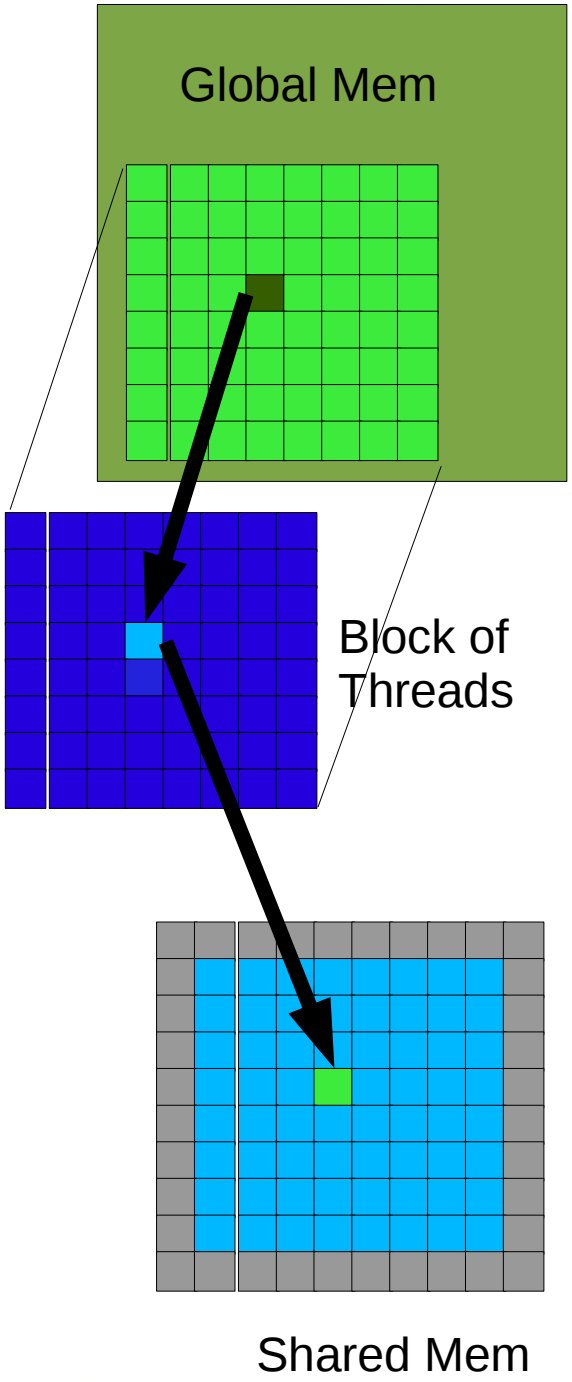
```
float* d_idata;                    // device (GPU) input data

cudaMalloc( (void**) &d_idata, mem_size);

float* d_odata;          // device (GPU) output data

cudaMalloc( (void**) &d_odata, mem_size);

loadImg( “input”, h_idata); // fill source host data by CPU

cudaMemcpy( d_idata, h_idata, mem_size,
                  cudaMemcpyHostToDevice);

dim3  threads( 8, 8, 1);          // threads per block

dim3  grid( 64, 64, 1);           // blocks in grid

kernel<<< grid, threads >>>( d_idata, d_odata);

cudaThreadSynchronize(); // wait for result

cudaMemcpy( h_odata, d_odata, mem_size,
                  cudaMemcpyDeviceToHost);

saveImg( “out.tga”, h_odata);
```



Example

normal map

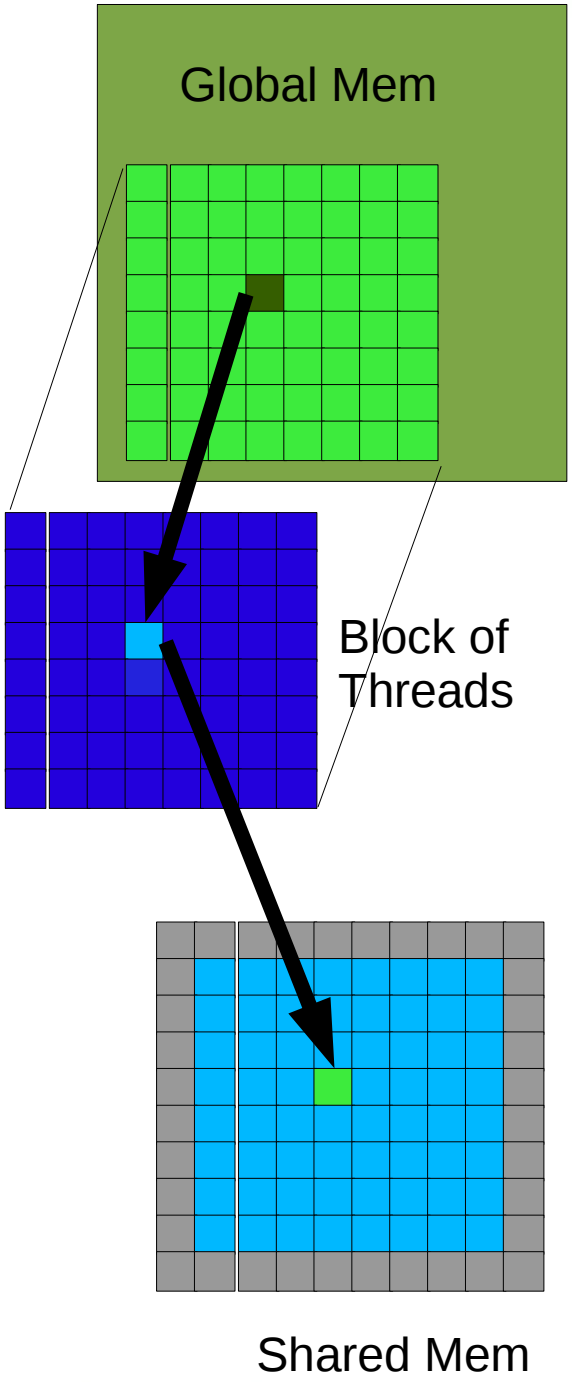
- __global__ void **kernel**(float* g_idata,
  float3* g_odata) {
- **__shared__ float s_data[10][10];**
- const unsigned int **gdid** = threadIdx.x +
  blockDim.x * (blockIdx.x + gridDim.x *
  (threadIdx.y + blockIdx.y * blockDim.y));
-
- **s_data [threadIdx.y+1]**
  - **[threadIdx.x+1] = g_idata[gdid];**
-
- **__syncthreads();**



Example

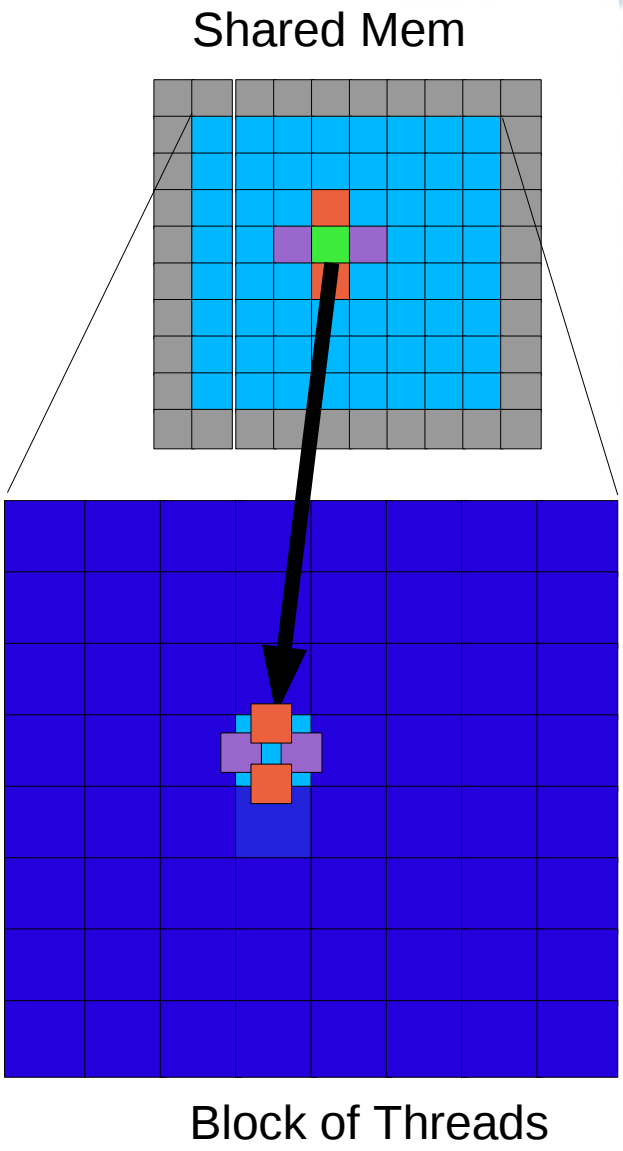
normal map

- __global__ void **kernel**(float* g_idata,
  float3* g_odata) {
- **__shared__ float s_data[10][10];**
- const unsigned int **gdid** = threadIdx.x +
  blockDim.x * (blockIdx.x + gridDim.x *
  (threadIdx.y + blockIdx.y * blockDim.y));
-
- **s_data [threadIdx.y+1]**
  - **[threadIdx.x+1] = g_idata[gdid];**
-
- **__syncthreads();**



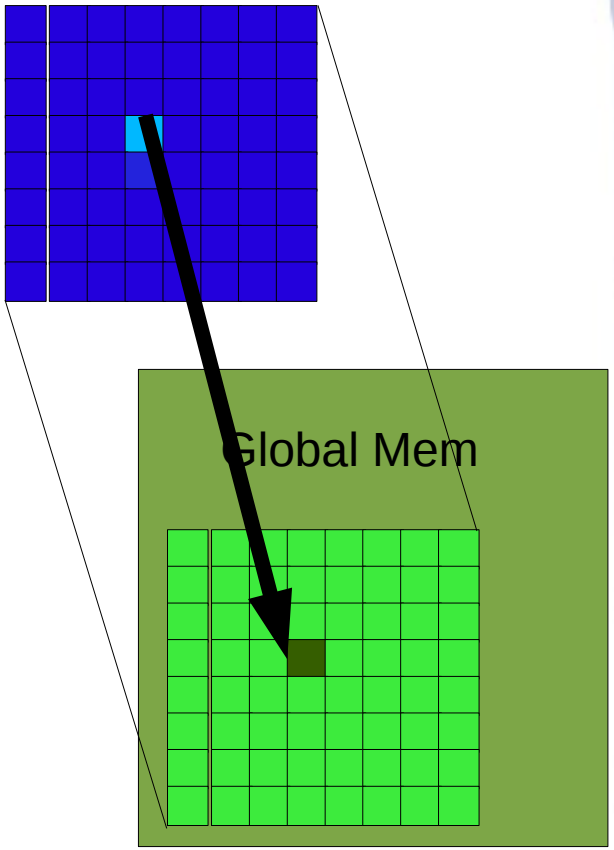
Example

- __global__ void **kernel**(float* g_idata,
  float3* g_odata) {
- __shared__ float s_data[10][10];
- const unsigned int gdid = threadIdx.x + blockDim.x * (blockIdx.x + gridDim.x * (threadIdx.y + blockIdx.y * blockDim.y));
- s_data [threadIdx.y+1] [threadIdx.x+1] = g_idata[gdid];
- __syncthreads();
- float **z_dx=**s_data[threadIdx.y+1] [threadIdx+1 **-1**]
  -s_data[threadIdx.y+1] [threadIdx+1 **-1**];
- float z_**dy=**s_data[threadIdx.y+1 **-1**] [threadIdx+1]
  -s_data[threadIdx.y+1 **+1**] [threadIdx+1];



Example

- __global__ void **kernel**(float* g_idata,
  float3* g_odata) {
- **__shared__ float s_data[10][10];**
- **const unsigned int gdid = threadIdx.x + blockDim.x * (blockIdx.x + gridDim.x * (threadIdx.y + blockIdx.y * blockDim.y));**
- **s_data [threadIdx.y+1] [threadIdx.x+1] = g_idata[gdid];**
- **__syncthreads();**
- **float z_dx=s_data[threadIdx.y+1] [threadIdx+1 -1] -s_data[threadIdx.y+1] [threadIdx+1 -1];**
- **float z_dy=s_data[threadIdx.y+1 -1] [threadIdx+1] -s_data[threadIdx.y+1 +1] [threadIdx+1];**
- **float3 n=cross (float3(2,0,z_dx), float3(0,2,z_dy));**
- **normalize(n);**
- **g_odata [gdid] = n;**
- **}**



References

- NVIDIA CUDA Programming Guide 1.0, 1.1, 2.0, 2.1
- The CUDA Compiler Driver NVCC
- Nvidia Geforce GTX 200 GPU architecture overview
- NVIDIA’s Next Generation CUDA™ Compute Architecture: Fermi™
- AMD Stream computing user guide
- AMD Entering the golden age of Heterogeneous Computing
- AMD HD 6900 Series Instruction Set Architecture