Realtime Computer Graphics on GPUs

GPGPU II

Jan Kolomazník

Department of Software and Computer Science Education
Faculty of Mathematics and Physics
Charles University in Prague

Parallel Algorithms

Introduction

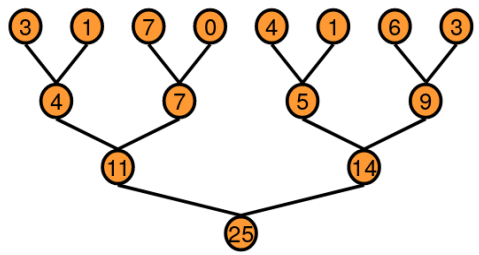
- Algorithms for massively parallel architectures:
  - Often bottom up design
  - Shallow datastructures
  - Memory access patterns considered first
- Try to make all operations local only
- Problem reformulation:
- Search for possible constrains
- Solve dual problem
- Cellular automata
- ...

Basic Meta-Algorithms

- Map :
  - ForEach (inplace?)
  - Transform
- Spatial filters with limited support:
  - Convolution
  - Morphological operations

Reduce (Fold)

- Associative binary operation to combine input elements into single value:
  - Sum
  - Multiplication
  - Min/Max
- On GPU:
  1. Tree-based approach used within each thread block
  2. Global sync (multiple kernels)
  3. Reduce block results



- Optimizations:
  - Prevent thread idling
  - Shared memory access patterns
  - Ref: Mark Harris – Optimizing Parallel Reduction in CUDA (30x speedup)

Scan (Prefix-sum)

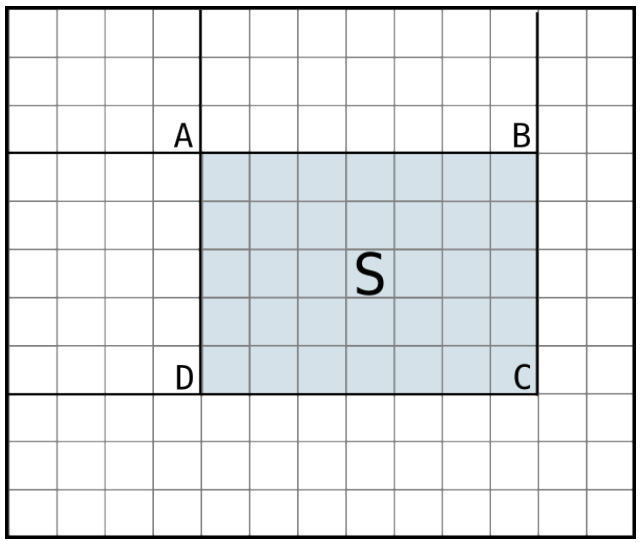
- Associative binary operation to combine input elements in front of each of the processed elements
- Inclusive vs. exclusive
- Similar implementation and optimizations as reduction

Example: Select Elements by Indicator

- Task:
  - Output elements selected by predicate
- Naive approach:
  - Adding Elements to output array directly
  - Bottleneck – size update
- Better solution:
  - Two level solution
  - Store local selection into shared memory
  - Update global output size once per block
- Advanced solution:
  - Run parallel prefix-sum on indicator set (0/1 for each element)
  - Computes indices for all selected elements and final size
  - Final step – store all selected elements on precomputed positions

Example: Integral Images

- Apply prefix-sum to rows and columns
- Sum/average queries for rectangular regions with constant complexity
- Used in feature detectors:
  - Haar features
  - Blur filter approximation



DATASTRUCTURES

- Basic categories:
  - Dense arrays
  - Hash tables
  - Sparse structures
    - Matrices
    - Graphs
- Different criteria:
  - Read/write
  - Incremental changes vs. Rebuild from scratch
  - Space waste
- Two level design:
  - Local datastructure living in shared memory
  - Main datastructure living in global memory
  - Write local instances at the end of computation

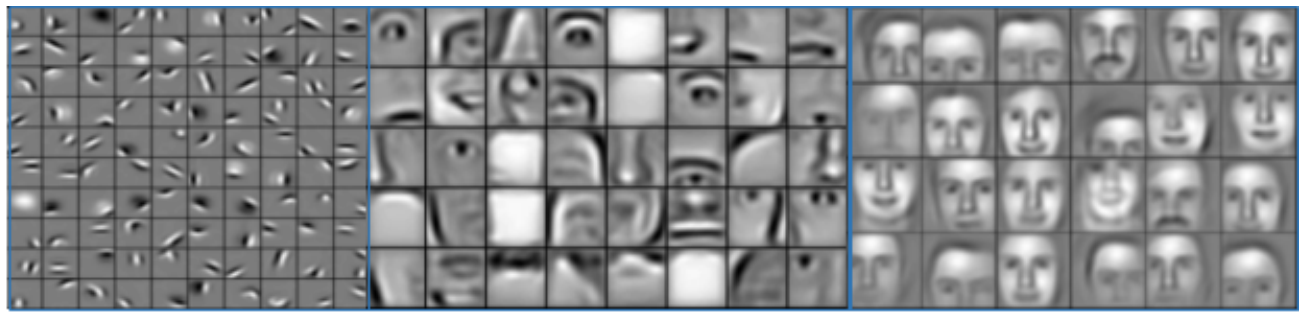
Deep Neural Networks

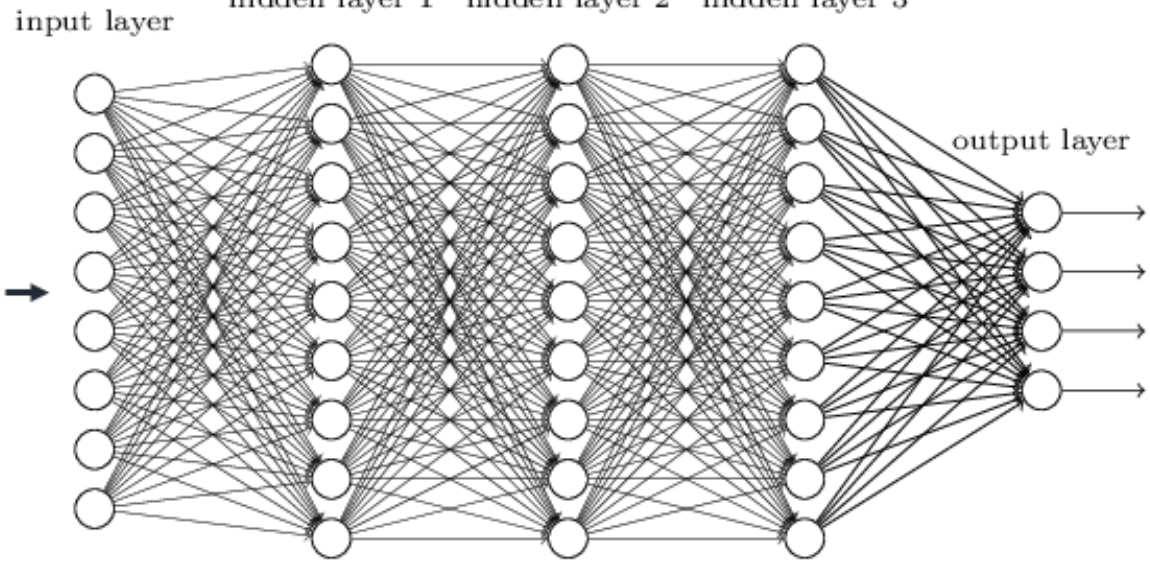
Deep Neural Networks

- Another neural networks renaissance
- Large neural networks with lots of layers
  - Convolutional networks
- Large numbers of identical neurons – highly parallel by nature
- Backpropagation
  - Millions of parameters
  - Large training set
- Training vs. inference

EXAMPLE

Deep neural networks learn hierarchical feature representations





cuDNN

- GPU-accelerated library of primitives for deep neural networks
- Used for speedup of DNN frameworks like:
  - TensorFlow
  - Caffe
  - Pytorch
  - Keras
  - Matlab
  - ...
- Special implementations for selected common cases – highly optimized

TensorRT

- Also by Nvidia
- Inference optimization
  - Significantly less computationally demanding than training
  - Deployed on embedded systems – memory constrains
- Change network topology without sacrificing inference precision
- Use lower numerical precision
- Less space occupied by weights
- Usage of tensor cores

Tensor Cores

- New in Volta architecture
- Accelerate matrix problems of the form $D = A * B + C$
  - Single core works on 4x4 matrices
  - fp16 precision for multiplication
  - fp16 or fp32 for accumulation
- Warp matrix functions
  - Special calls for specific matrix sizes
- Significant speedup of NN training and inference
- Denoising in raytracing APIs

OpenCL

OpenCL

- Alternative to *C for CUDA*
- Basic idea from *C for CUDA*, 1:1 equivalence in some parts
- Programming model for execution of massivily parallel tasks on **CPU, GPU, Cell,** ...
- Language:
  - Originally subset of C99
  - Subset of C++14 in OpenCL 2.1
  - Just-in-time compilation

Host Code

- Code structure similar to shader programming
- API:
  - clBuildProgram()
  - clCreateCommandQueue()
  - clCreateBuffer()
  - clEnqueueWriteBuffer()
  - ...

OpenCL Concepts

OpenCL	**CUDA equivalent**
kernel	kernel
host program	host program
NDRange (index space)	grid
work item	thread
work group	block

OpenCL threads

OpenCL	CUDA equivalent
`get_global_id(0)`	`blockIdx.x` · `blockDim.x` + `threadIdx.x`
`get_local_id(0)`	`threadIdx.x`
`get_global_size(0)`	`gridDim.x` · `blockDim.x`
`get_local_size(0)`	`blockDim.x`

OpenCL Memory

OpenCL	**CUDA equivalent**
global memory	global memory
constant memory	constant memory
local memory	shared memory
private memory	local memory

Sample OpenCL Kernel

```
// Kernel definition
__kernel void VecAdd(
    __global const float *A,
    __global const float *B,
    __global float *C)
{
    int id = get_global_id(0);
    C[id] = A[id] + B[id];
}
```

SPIR

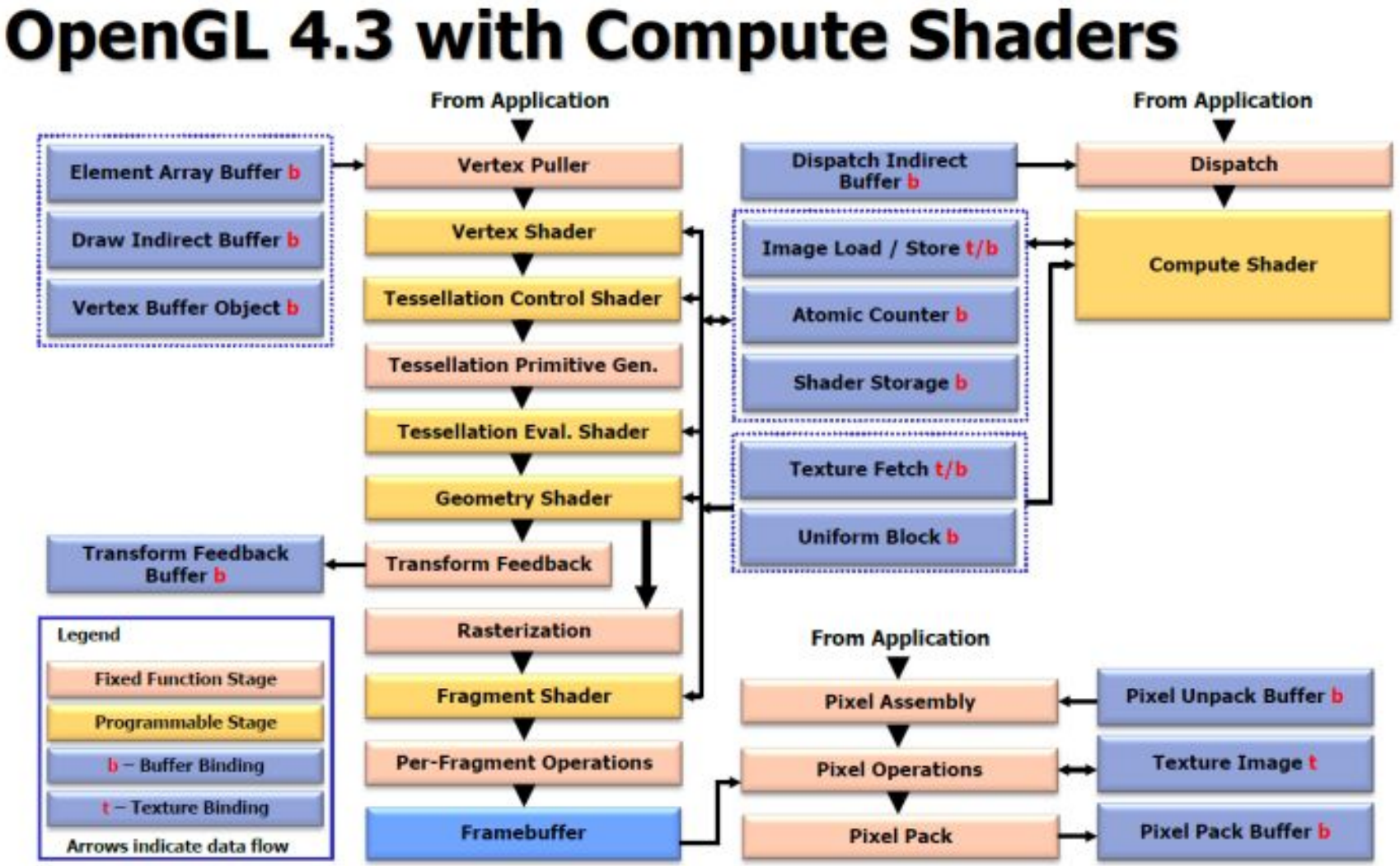
- Standard Portable Intermediate Representation
- Distribute device-specific pre-compiled binaries
- SPIR-V incorporated in the core specification of:
  - OpenCL 2.1
  - Vulkan API
  - OpenGL 4.6

Compute Shaders

Motivation

- Why not OpenCL or CUDA?
  - One API for graphics and GP processing
  - Avoid interop
  - Avoid context switches
  - You already know GLSL
- APIs:
  - Core since OpenGL 4.3
  - Part of OpenGL ES 3.1
  - Vulcan

WHERE IT BELONGS?



Usage

- Write compute shader in GLSL
  - Define memory resources
  - Write main()function
- Initialization
  - Allocate GPU memory (buffers, textures)
  - Compile shader, link program
- Run it
  - Bind buffers, textures, images, uniforms
  - Call glDispatchCompute(...)

Sample Compute Shader

```
#version 430
layout(local_size_x = 1, local_size_y = 1) in;
layout(rgba32f, binding = 0) uniform image2D img_output;

void main() {
  // base pixel colour for image
  vec4 pixel = vec4(0.0, 0.0, 0.0, 1.0);
  // get index in global work group i.e x,y position
  ivec2 pixel_coords = ivec2(gl_GlobalInvocationID.xy);


  // output to a specific pixel in the image
  imageStore(img_output, pixel_coords, pixel);
}
```

Builtin Variables

```
in uvec3 gl_NumWorkGroups;
in uvec3 gl_WorkGroupID;
in uvec3 gl_LocalInvocationID;
in uvec3 gl_GlobalInvocationID;
in uint  gl_LocalInvocationIndex;
```

Other APIs

C++ AMP, OpenACC

- C++ Accelerated Massive Parallelism:
  - Open specification from Microsoft
  - Builds on DX11
  - Language extensions, runtime library
  - Heterogenous computation
  - Deprecated since VS 2022
- OpenACC:
  - Similar to OpenMP
  - Compiler directives (pragmas) + runtime library

SYCL

- Standard by Khronos
- Cross-platform abstraction layer
- Builds on the underlying concepts, portability and efficiency of OpenCL
- **Single-source** style using completely standard C++

SAMPLE SYCL CODE

```
#include<CL/sycl.hpp>
#include<iostream>

int
main() {
   using namespace cl::sycl;
   int data[1024];
   // initialize data to be worked on
   // By including all the SYCL work in a {} block, we ensure
   // all SYCL tasks must complete before exiting the block
   {
      queue myQueue;
      bufferbuffer<int, 1> resultBuf(data, range<1>(1024));
      // create a command_group to issue commands to the queue
      myQueue.submit([&](handler& cgh) {
            // request access to the buffer
            auto writeResult = resultBuf.get_access<access::write>(cgh);
            // enqueuea parallel_for task
            cgh.parallel_for<class simple_test>(range<1>(1024), [=](id<1> idx) {
               writeResult[idx] = idx[0];});
         });
   } // end of scope, so we wait for the queued work to complete
   for(int i = 0; i < 1024; i++)
      std::cout<< "data[" << i << "] = " << data[i] << std::endl;
   return 0;
}
```

Thank you for your attention!