Realtime Computer Graphics on GPUs

GPGPU I

Jan Kolomazník

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

Computer
Graphics
Charles
University

Introduction

HARDWARE ADVANCES

- VLSI technology evolves very quickly
  - Frequency scaling slowed after 2005 (2-3GHz)
  - modern desktop parts now boost up to 6 GHz
- Current trend – multi-core approach
  - parallelizable tasks
  - changes in programming techniques (traditionally all the algorithms used to be sequential)
  - *High-performance computing (HPC)* community practices are becoming common in consumer segment

GPU ADVANCES

- graphic hardware evolution
  1. single-purpose
  2. configurable
  3. programmable
- currently almost arbitrary algorithm can run on a GPU
  - code size and memory are the only limits
- but GPU is efficient for special class of algorithms only

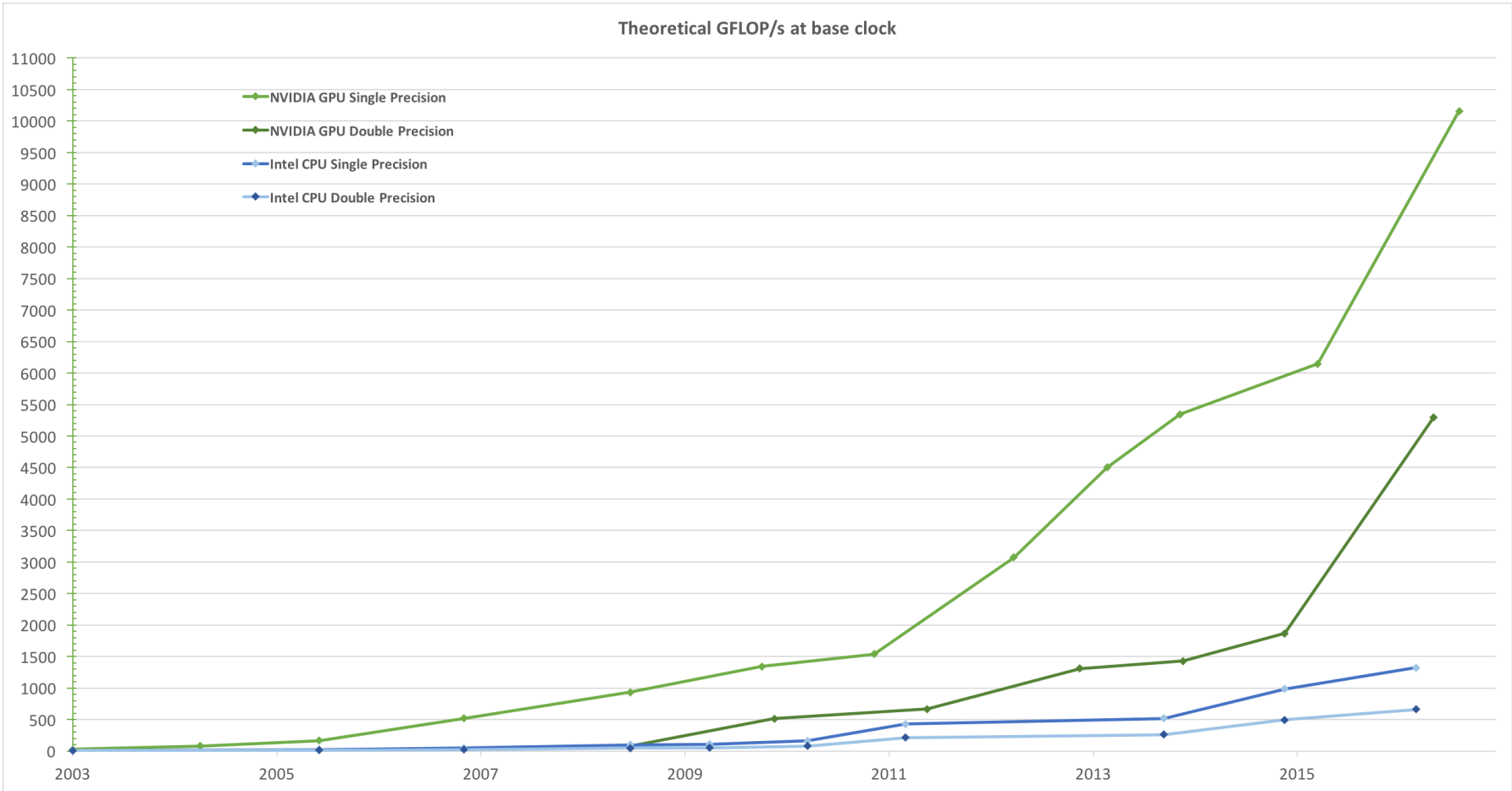
MULTI-CORE COMPUTING

- running many sequential algorithms
- two- to many-core CPUs
- full x86-64 instruction set, SSE, AVX extensions
- example 1: Intel Core i9 (Sky Lake X, Sep 2017)
  - 14 nm, 24MB L3 cache, TDP: 165W
  - 18 cores + hyperthreading = 36 computing threads
  - out-of-order execution
- example 2: Intel Xeon 6 “Sierra Forest” 6700E (2024)
  - Up to 288 Crestmont E-cores, 96MB L3 cahe, TDP: 330W

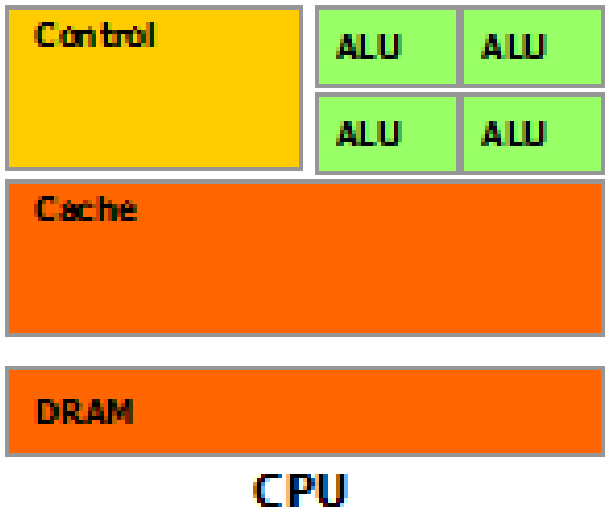
MANY-CORE COMPUTING

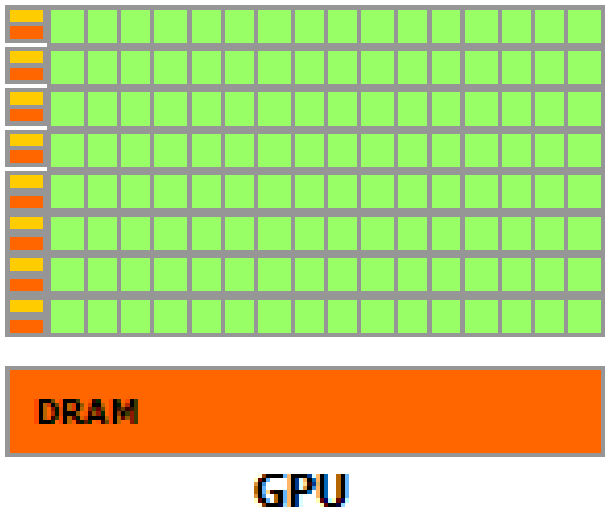
- emphasis on parallel algorithms
- graphic accelerators – programmable GPUs
- more simple instruction sets
- very high brute computing power
- example 1: NVIDIA Quadro GP100 (Pascal, 2017)
  - 16 nm, 3584 cores, 1476MHz, TDP: 235W
  - 16 GB HBM2 RAM (4096-bit bus)
- example 2: NVIDIA H100 (Hopper, 2023)
  - 132 SMs / 16 896 CUDA cores, 1.9GHz, TDP: 700W
  - 80 GB HBM3 RAM 3 TB/s

FLOPS

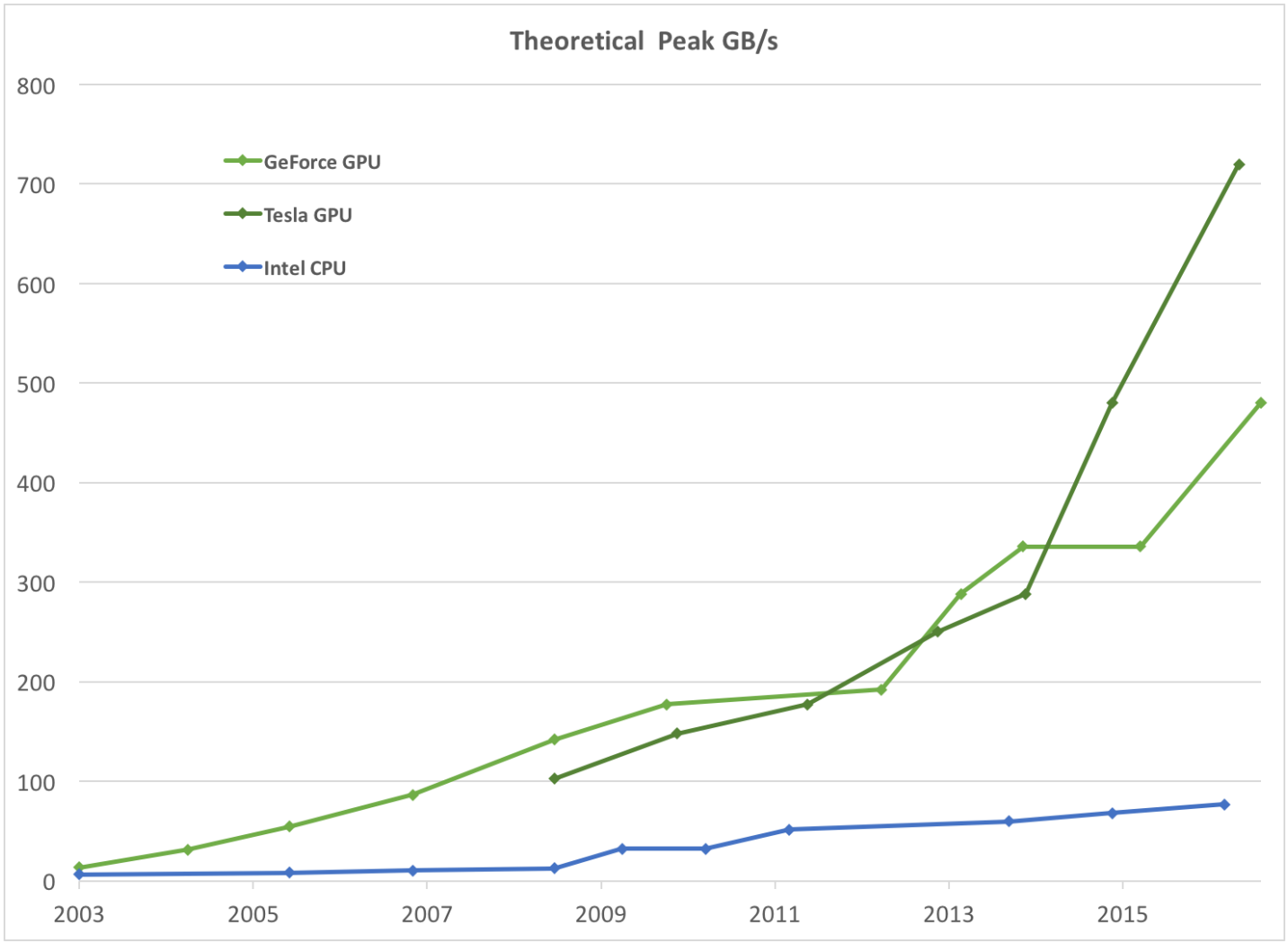

Theoretical GFLOP/s at base clock
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Single Precision
Intel CPU Double Precision
11000
10500
10000
9500
9000
8500
8000
7500
7000
6500
6000
5500
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
2003
2005
2007
2009
2011
2013
2015

CPU vs. GPU





Memory Bandwidth



CUDA

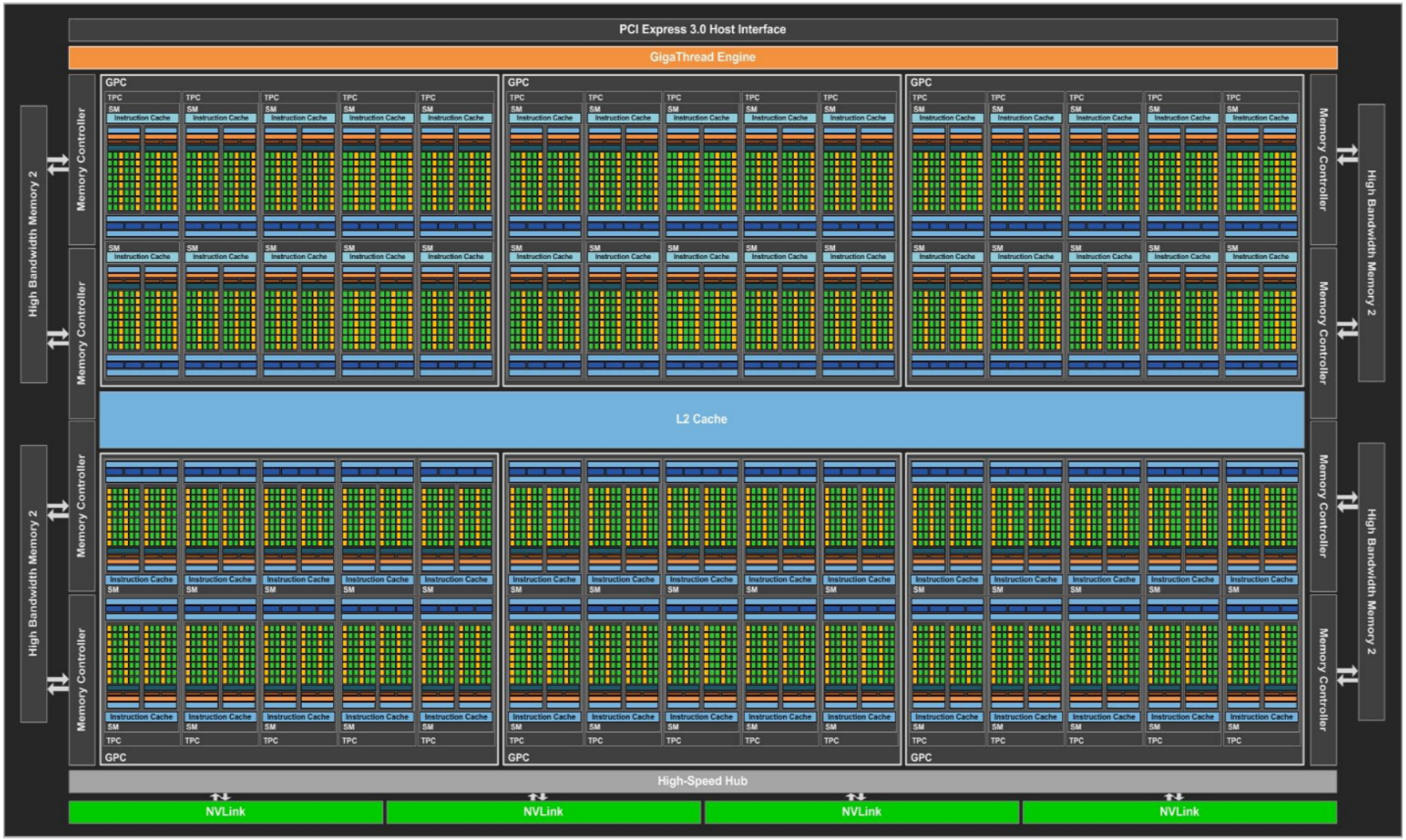
History

- GPGPU (history) – computing in shaders, data in textures
  - more elegant access to GPU resources was needed
- 2007: NVIDIA Tesla architecture (G80, GeForce 8800)
  - general model for parallel programming
  - computing thread hierarchy
  - barriers for synchronization
  - atomic operations
- CUDA: Compute Unified Device Architecture
  - C for CUDA: programming language

CUDA Architecture

- CUDA GPU: group of highly threaded streaming multiprocessors (SM)
- number of SM per GPU depends on GPU generation
- each SM has a set of streaming processors (SP) – CUDA cores
  - G80: 8 SP per SM
  - GF100: 32 SP per SM
  - GP100: 64 SP per SM
  - GA100: 64 SP per SM
  - GH100: 128 SP per SM
- SP within one SM share control circuits, instruction decoder and instruction cache
- SIMT (Single Instruction Multiple Threads) – warp

GPU Scheme


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Instruction Cache
Memory Controller
High Bandwidth Memory 2
L2 Cache
High-Speed Hub
NVLink

STREAMING MULTIPROCESSOR

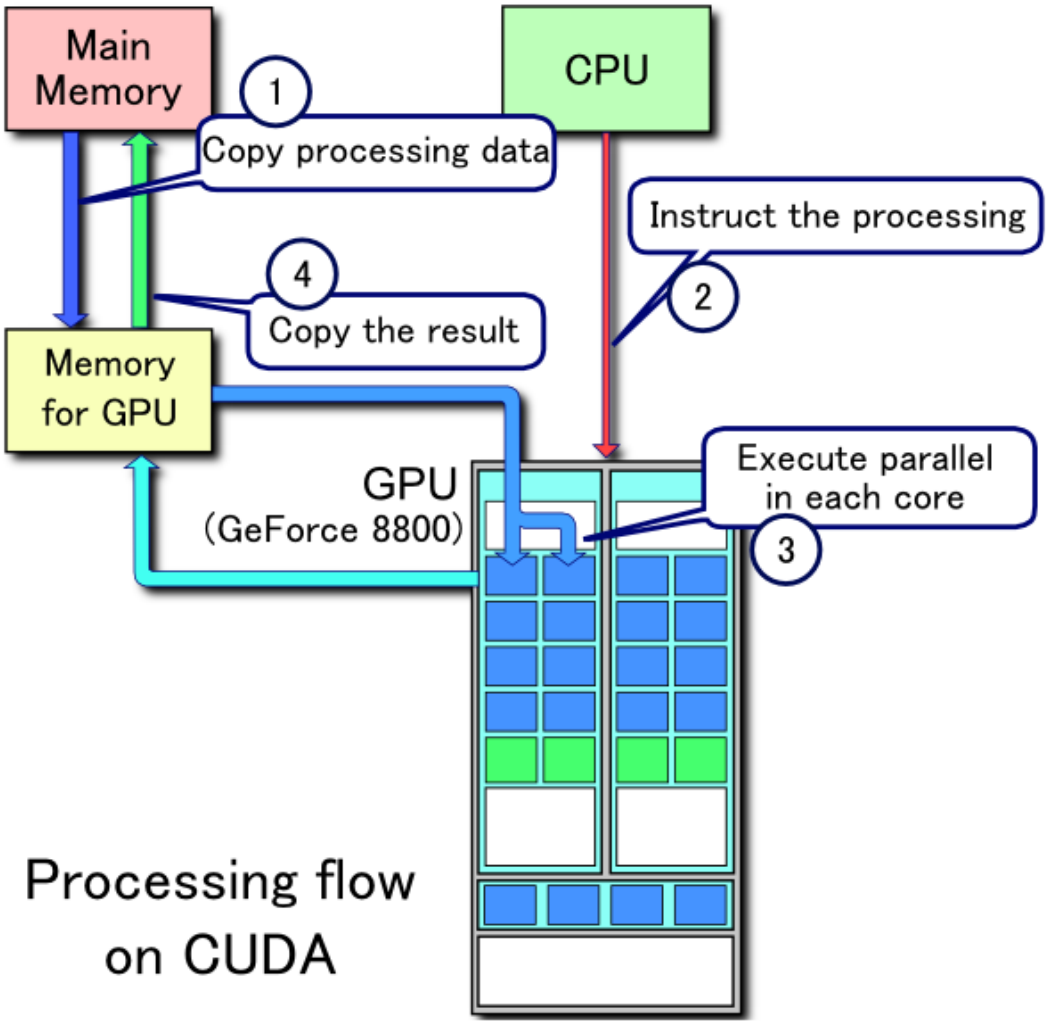
Compute Capability

- Specification of set of HW features
  - Instruction availability
  - Memory sizes
  - Synchronization options
- Mostly backward compatible
- Compilation must target specific feature set
- Each device supports specific Compute Capability version

Program Structure I

- one or more phases (one phase = one *kernel*)
- deployment on CPU or GPU
  - CUDA source code can contain both parts
- nvcc (NVIDIA C compiler) separates these parts during compilation
- language: from ANSI C to complete C++ support
- GPU code = **kernel**
  - deployed on thousands of threads
  - GPU threads are much more light-weighted (thread creation, ctxsw a couple of machine cycles)

PROGRAM STRUCTURE II



Program Structure III

- Keywords `__global__`, `__host__`, `__device__`
  - where the code can run
  - from where it can be called

Keyword	Runs on	Executable from
`__global__ float KernelFunc()`	GPU	host, GPU (CC 3.5)
`__device__ float DeviceFunc()`	GPU	GPU
`__host__ float HostFunc()`	host	host
`__host__ __device__ float HostDeviceFunc()`	host, GPU	host, GPU

Code Execution

Simple Example

```
// Kernel definition
__global__ void VecAdd(float *A, float *B, float *C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Run kernel in N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

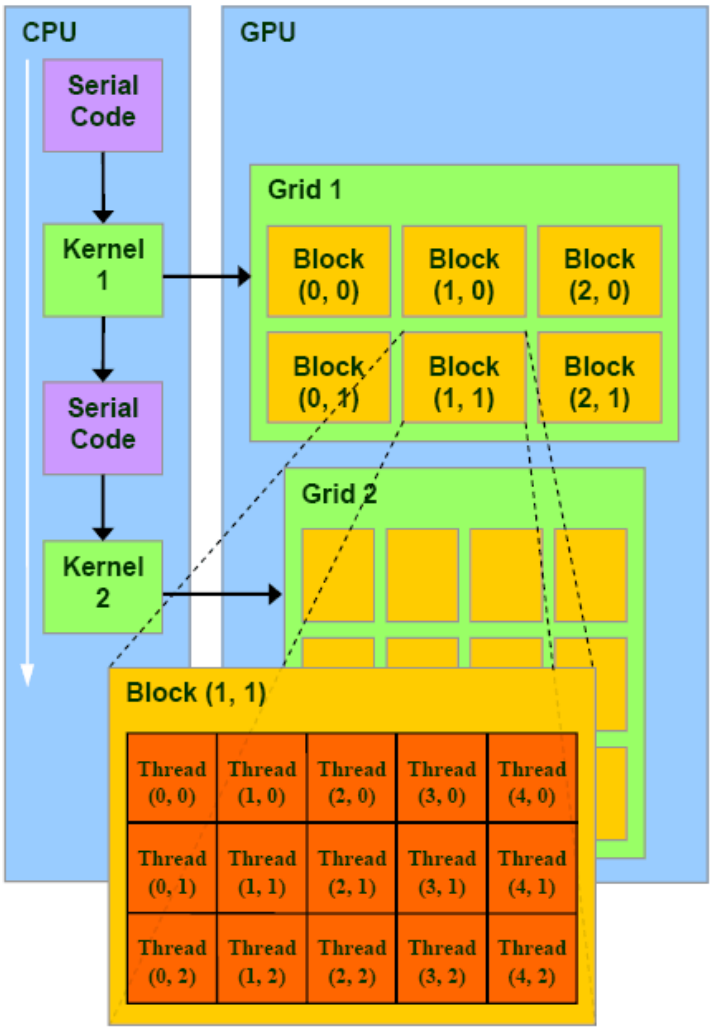
KERNEL EXECUTION

```
MyKernel<<<gridSize, blockSize, dynamicSharedMemorySize, streamID>>>(arg1, arg2, ...);
```

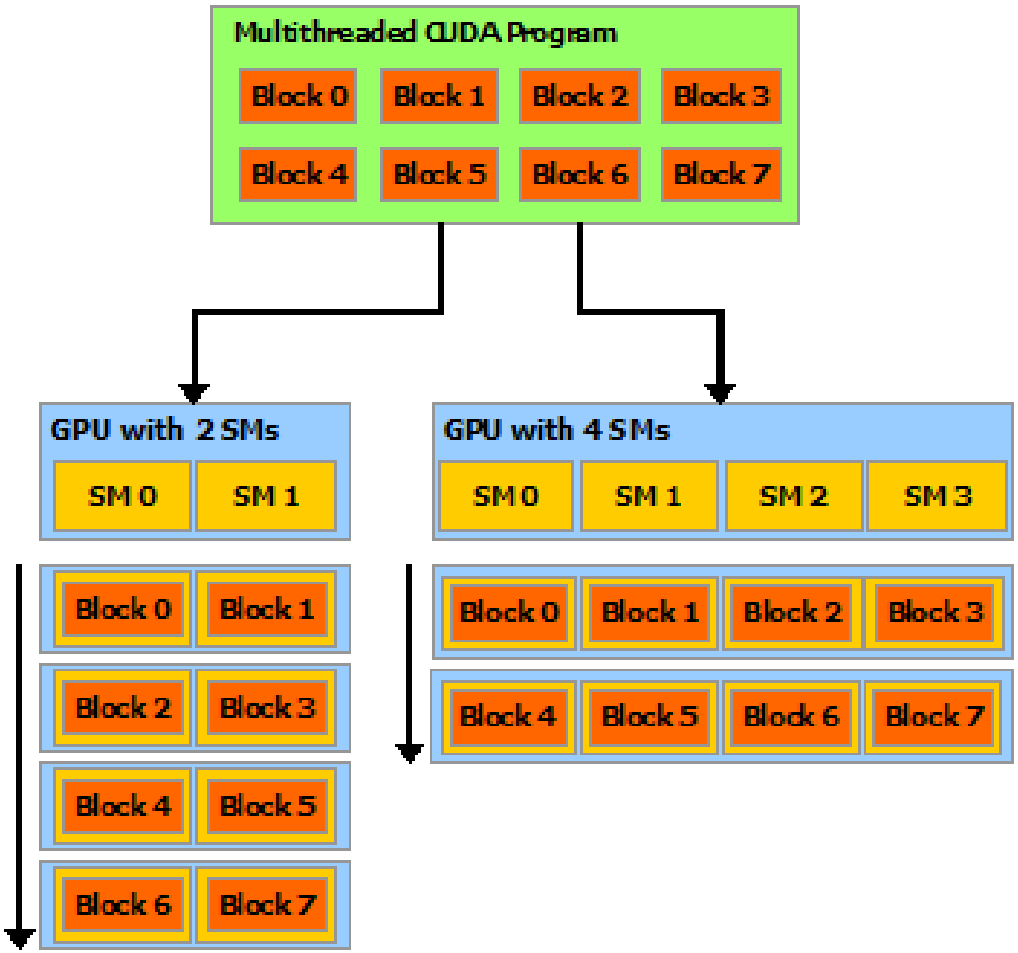
- gridSize
  - int or dim3
  - Specifies 3D structure of thread blocks
- blockSize
  - int or dim3
  - Specifies 3D structure of threads in block
- dynamicSharedMemorySize
  - amount of dynamically allocated shared memory
  - 0 is default
- streamID
  - Which stream is used for execution
  - 0 is default

GRID, BLOCK, WARP

- Usual approach – map grid and blocks on input data
- Block:
  - Executed on single SM
  - Cannot be removed until finished
- Threads in **warp**:
  - Set of threads that all share the same code
  - Follow the same execution path (masking execute on branching)



AUTOMATIC SCALABILITY



Simple Example Improved

```
// Kernel definition
__global__ void VecAdd(float *A, float *B, float *C, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        C[i] = A[i] + B[i];
    }
}

int main()
{
    ...
    constexpr int M = 256;
    // Run kernel in M threads per block
    VecAdd<<<(N + M - 1)/ M, M>>>(A, B, C, N);
    ...
}
```

STREAMS

- CUDA Stream – queue of commands (kernel execution, memory transfers, event)
- Commands in stream serialized
- Different streams – possible concurrency
- Default stream 0 always exists (can be per thread)
- `cudaStreamCreate()`
- Synchronization:
  - `cudaStreamSynchronize(stream)`
  - Event system

Memory Types

Host Memory (RAM)

- Normal memory RAM
- By default cannot be accessed from device
  - Must be copied to device memory

Global Memory

- Actual GPU memory
- Used as normal linear memory – pointer arithmetics
- Management:
  - `cudaMalloc()`, `cudaMallocPitch()`, `cudaMalloc3D()`
  - `cudaMemcpy()`, `cudaMemcpyToSymbol()`
- Read from kernel can take hundreds of cycles

Texture Memory

- Allow usage of texturing HW:
  - Spatial caching
  - Filtering
- Limited by predefined element types (colors)
  - No custom structures

Shared Memory

- Same space as L1 cache
  - Division is customizable
- `__shared__` keyword
- Shared by threads in block
- Use when same value accessed is multiple times in block execution (not necessarily by same thread)

Registers

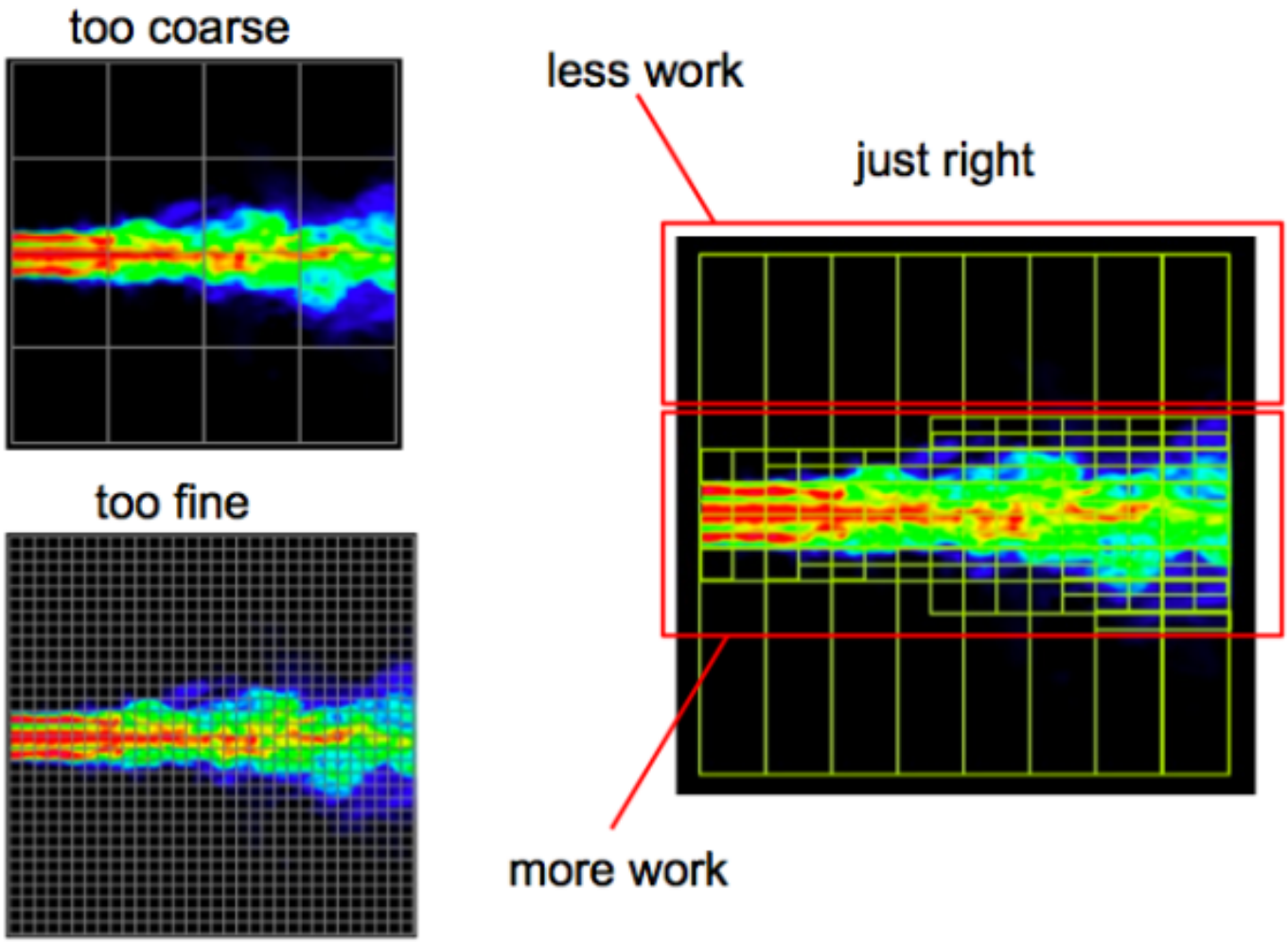
- 32-bit registers
- Divided between active warps
- Shared memory + registers limit occupancy:
  - Number of active warps vs. max possible warps on SM
- Limiting number of registers may lower performance
  - May be necessary to run at least 1 block on SM

Advanced Features

Unified Memory

- Single memory address space accessible from any processor in a system
- Replace malloc(), new with calls to cudaMallocManaged()
- New on Kepler architecture
- Pascal:
  - 49-bit virtual addressing
  - On-demand page migration
  - Oversubscription to GPU memory

Dynamic Parallelism I



Dynamic Parallelism II

- Kernels can be executed from kernels on device
- Parent kernel waits until children finishes
- Allows adaptive thread execution

GRAPHICAL APIS CONNECTION

- Acquire access to memory used by buffer objects in OpenGL, DX, Vulcan
  - `cudaGraphicsMapResources(...),...`
- Read or update the memory
- Unmap the resource

Parallel Algorithms

Introduction

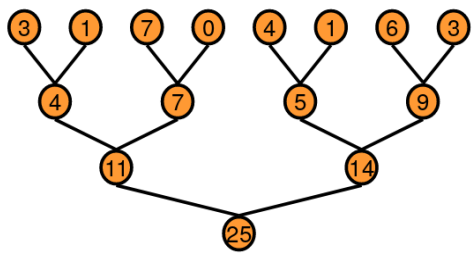
- Algorithms for massively parallel architectures:
  - Often bottom up design
  - Shallow datastructures
  - Memory access patterns considered first
- Try to make all operations local only
- Problem reformulation:
  - Search for possible constrains
  - Solve dual problem
  - Cellular automata
  - ...

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)

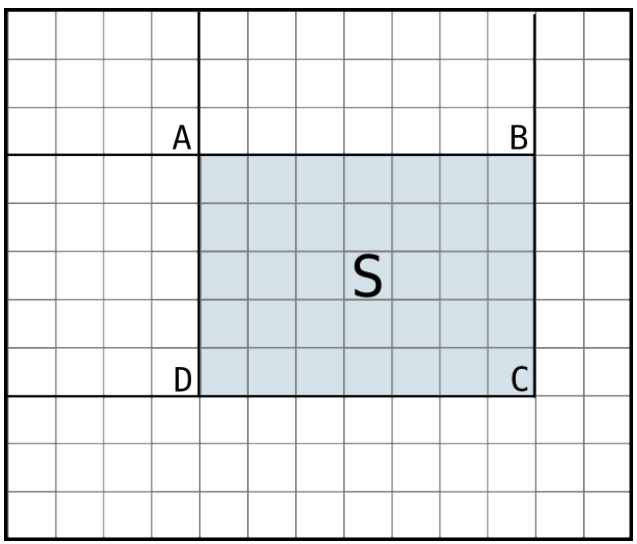
- Associative binary operation to combine input elements in front of each of the processed elements
- Inclusive vs. exclusive
- Similar implementation and optimizations as reduction

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

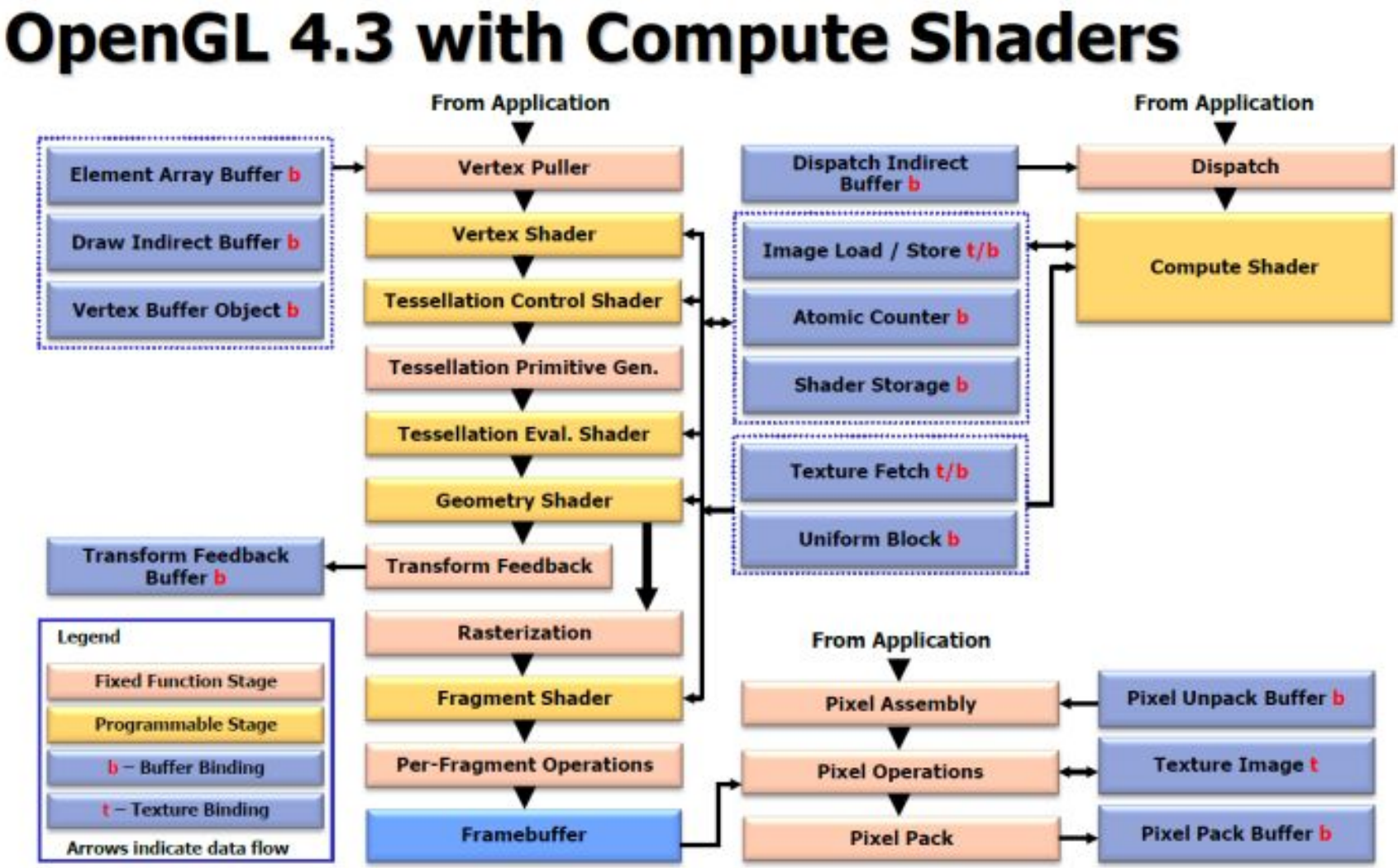
- Basic categories:
  - Dense arrays
  - Hash tables
  - Sparse structures
    - Matrices
    - Graphs
- Different criteria:
  - Read/write
  - Incremental changes vs. Rebuild from scratch
  - Space waste
- Two level design:
  - Local datastructure living in shared memory
  - Main datastructure living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

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;
```