Ruini Xue

School of Computer Science and Engineering

University of Electronic Science and Technology of China

2016

Outline

- Introduction
  - GPU
  - GPGPU
- Programming Concepts and Mappings
  - Direct3D and OpenGL
  - NVIDIA CUDA
- Case Study: Decoding H.264/AVC

Accelerators

- No, not this ….

http://gizmodo.com/5032891/nissans-eco-gas-pedal-fights-back-to-help-you-save-gas

Accelerators

- In HPC, an accelerator is hardware component to speed up some aspect of the computing workload.
- In the olden days (1980s), supercomputers sometimes had ***array processors***, which did vector operations on arrays, and PCs sometimes had ***floating point accelerators***.
- More recently, ***Field Programmable Gate Arrays*** (FPGAs) allow reprogramming deep into the hardware.

Inspur and Altera Launch Speech Recognition FPGA Solution with OpenCL

Nov 17, 2015, 23:09 ET from Inspur Group Co., Ltd.

Why Accelerators are

Good

- they make your code run faster.

Bad

- they're expensive;
- they're hard to program;
- your code on them may not be portable to other accelerators, so the labor you invest in programming them has a very short half-life.

The King of the Accelerators

- The undisputed champion of accelerators is:
- the graphics processing unit.

Graphics Processing Unit (GPU)

- Programmable chip on graphics cards
- Developed in a gaming context
  - 3-D scenery by means of rasterization
- Programmable pipeline since DirectX 8.1
  - vertex, geometry, and pixel shaders
  - high-level language support

- Modern GPUs support high-precision
- Massive floating-point processing power
  - 933 gigaflops (NVIDIA GeForce 280GTX)
  - 141.7 GB/s peak memory bandwidth
  - fast PCI-Express bus, up to 2GB/sec transfer speed

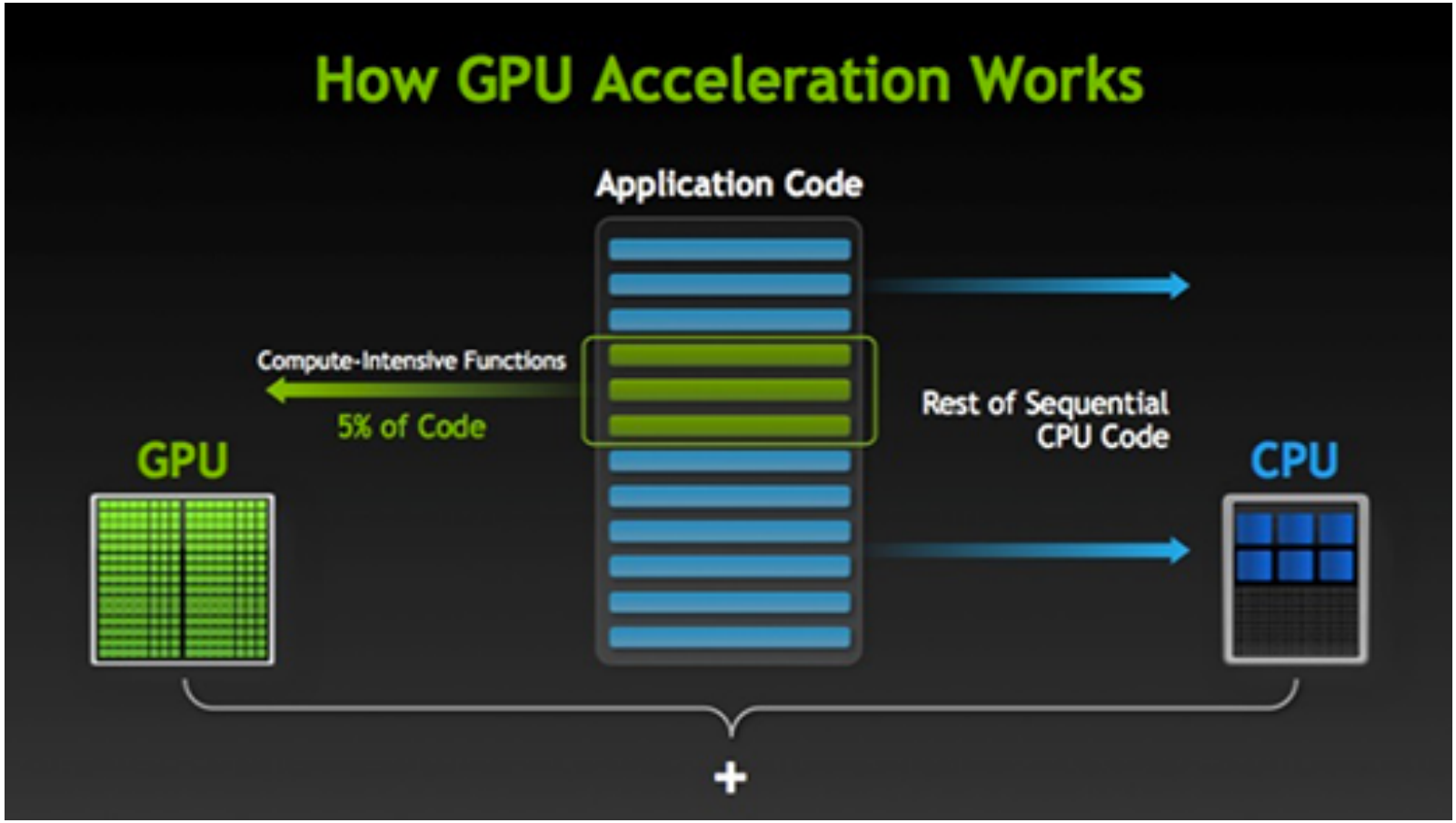
Why GPU?

- They ***became very very popular with videogamers***, because they've produced better and better images, and lightning fast.
- And, ***prices have been extremely good***, ranging from three figures at the low end to four figures at the high end.

CPU and GPU Comparison

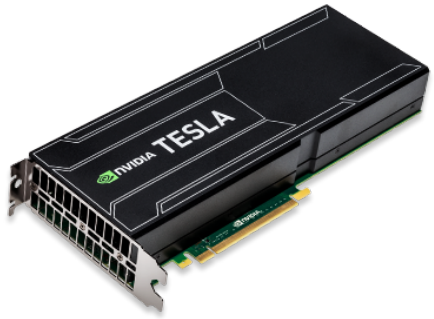
- Today's GPUs are yesterday's supercomputers

	Intel Xeon X5355	NVIDIA G80 (8800 GTX)
Clock Speed	2,66 GHz	575 MHz
#Cores / SPEs	4	128
Max. GFlop/s (float)	85	500
Typical Instr. Duration	1-2 cycles (SSE)	min. 4 cycles
Die Size (mm^2)	143	480
Typical Memory Speed	8GB/sec (DDR2-1066)	86 GB/sec (GDDR3-1800)
Power Usage (watt)	120	185
Price (€)	800	500

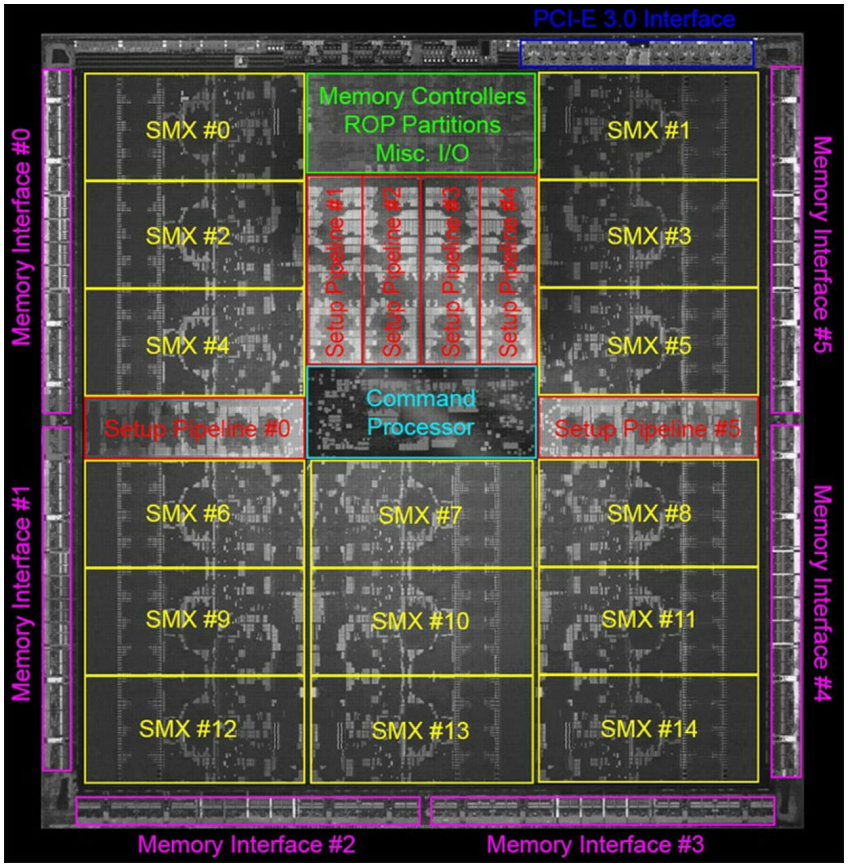

How GPU Acceleration Works
Application Code
Compute-Intensive Functions
5% of Code
Rest of Sequential
CPU Code
GPU
CPU
+

GPUs: massive data-parallelism for modest energy

- NVIDIA Tesla K40 discrete GPU: 4.3 TFLOPs, 235 Watts



Features	Tesla K40
Number and Type of GPU	1 Kepler GK110B
Peak double precision floating point performance	1.43 Tflops
Peak single precision floating point performance	4.29 Tflops
Memory bandwidth (ECC off)	288 GB/sec
Memory size (GDDR5)	12 GB
CUDA cores	2880

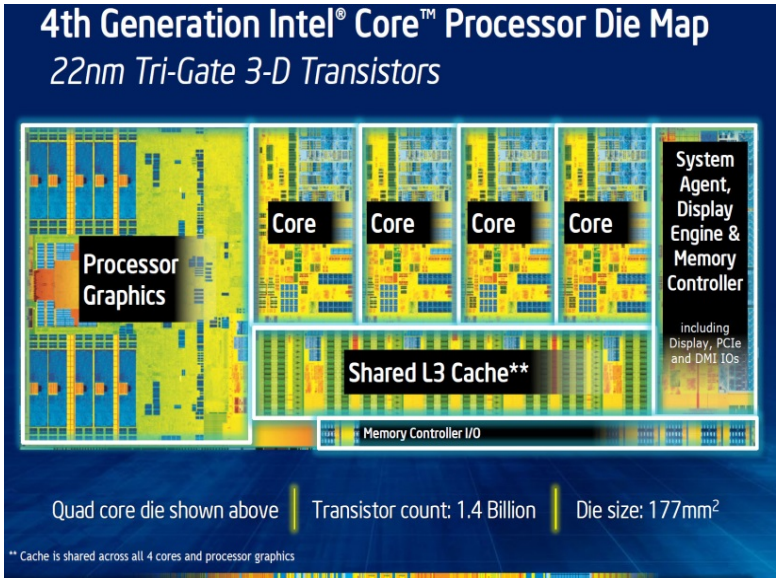


http://forum.beyond3d.com/showpost.php?p=1643034&postcount=107

Integrated CPU+GPU processors

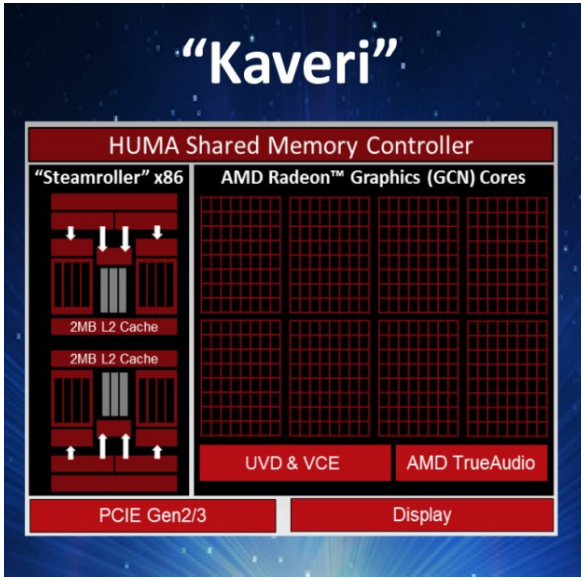
- More than 90% of processors shipping today include a GPU on die
- Low energy use is a key design goal

Intel 4th Generation Core Processor: "Haswell"



4-core GT2 Desktop: 35 W package
2-core GT2 Ultrabook: 11.5 W package

AMD Kaveri APU



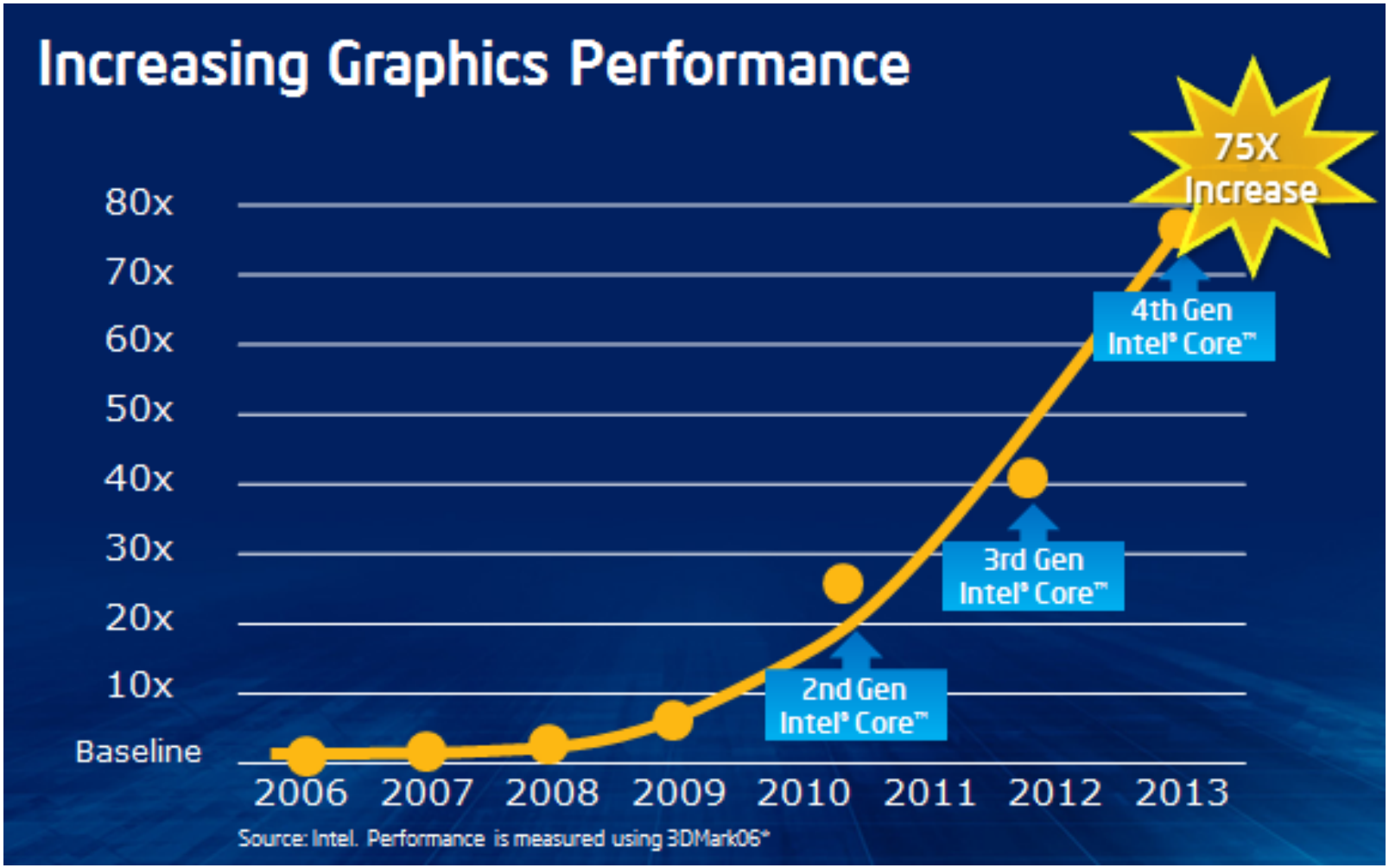
Desktop: 45-95 W package
Mobile, embedded: 15 W package

http://www.geeks3d.com/20140114/amd-kaveri-a10-7850k-a10-7700k-and-a8-7600-apus-announced/

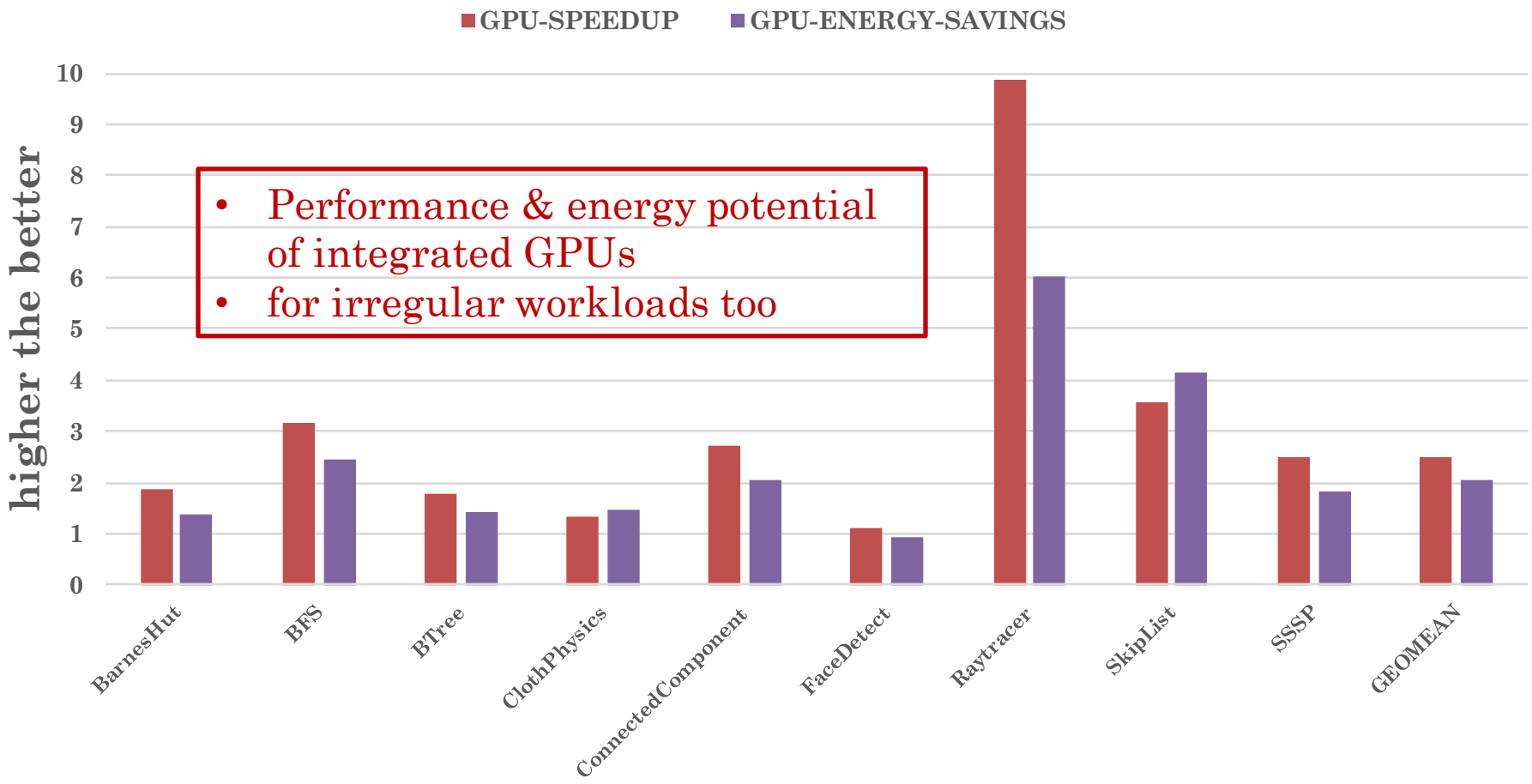
Discrete & integrated processors

- Discrete GPUs
  - Cost of PCIe transfers impacts granularity of offloading
- Integrated GPUs
  - The CPU and GPU share physical memory (DRAM)
    - Avoids cost of transferring data over a PCIe bus to a discrete GPU
  - May also share a common last-level cache
    - If so, data being offloaded is often in cache
- Same ISA but hybrid

Performance of integrated GPUs is increasing

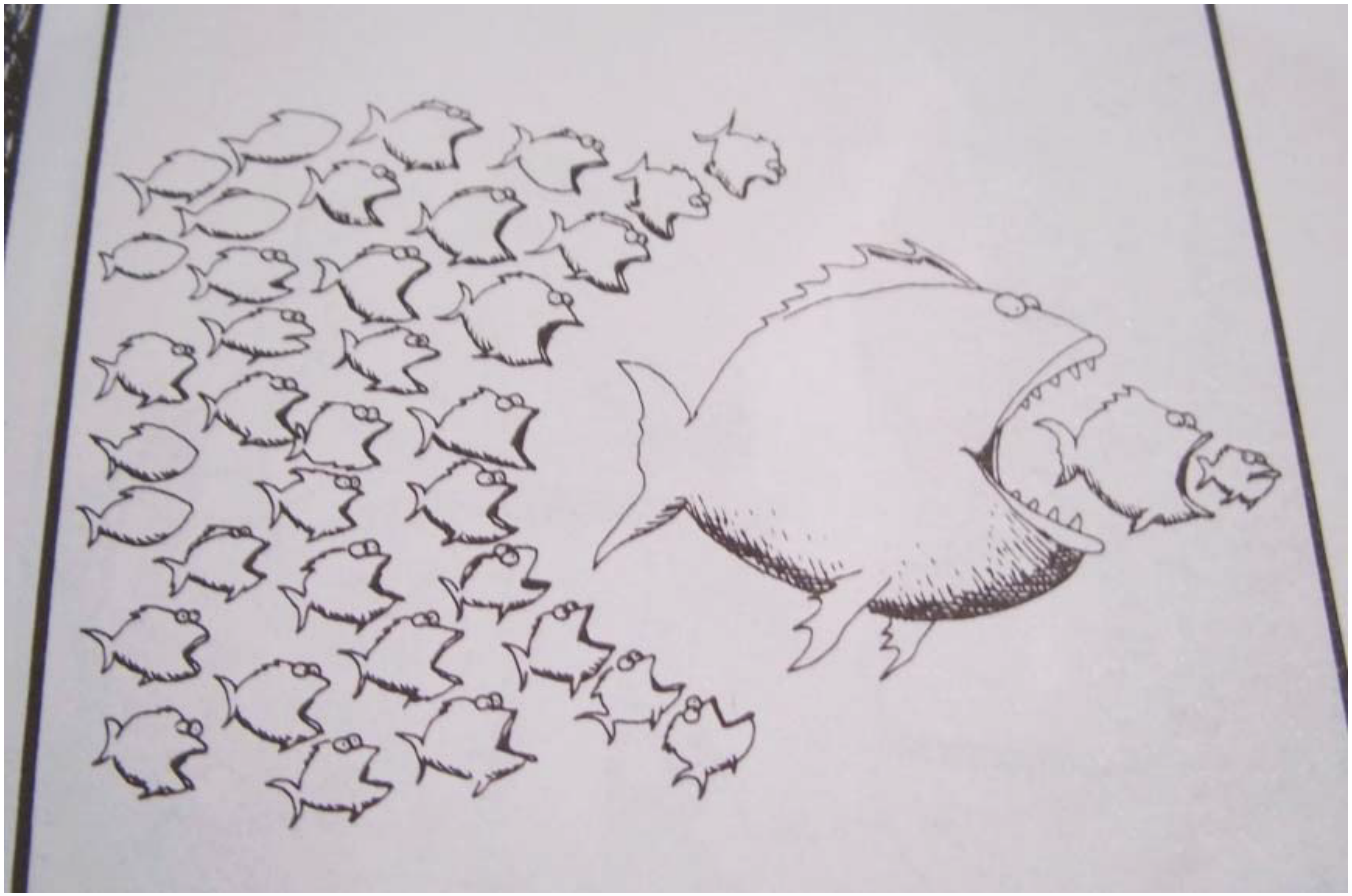


Ultrabook: Speedup & energy savings compared to multicore CPU



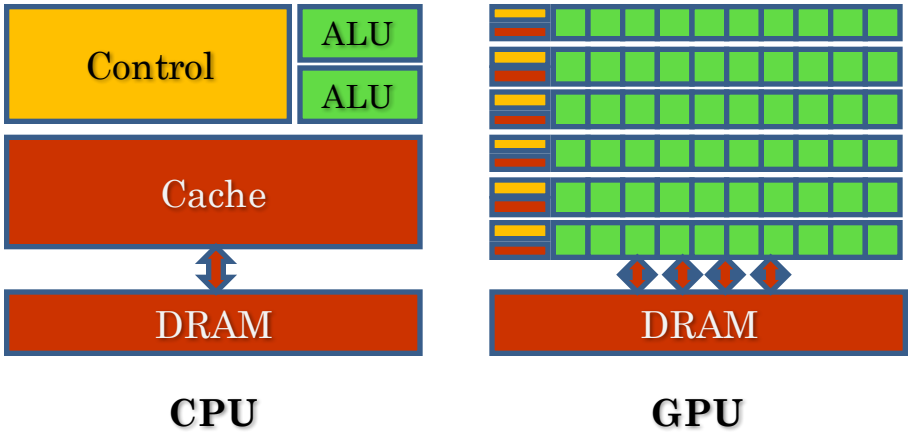
Average speedup of 2.5x and energy savings of 2x vs. multicore CPU

A powerful CPU or many less powerful CPUs?



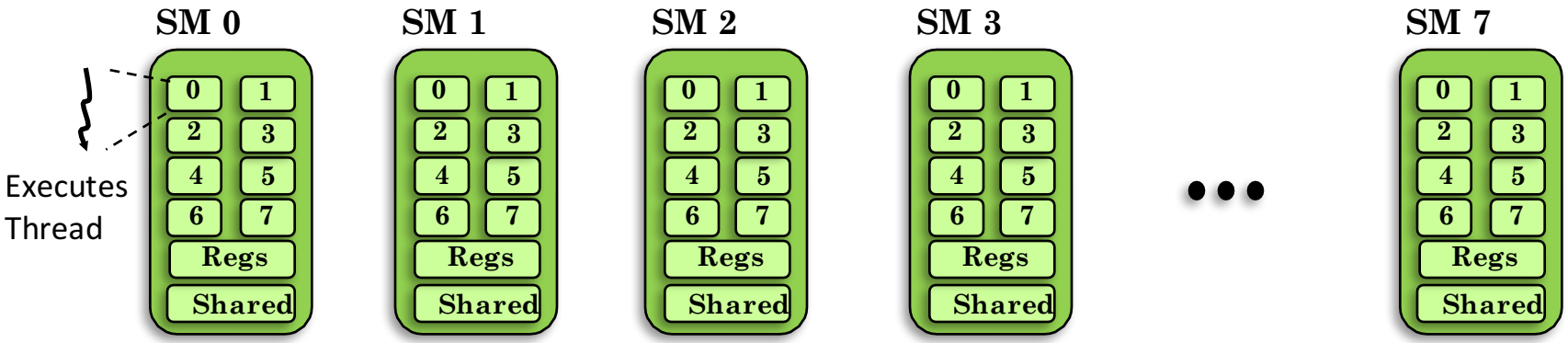
Why are GPUs so fast?

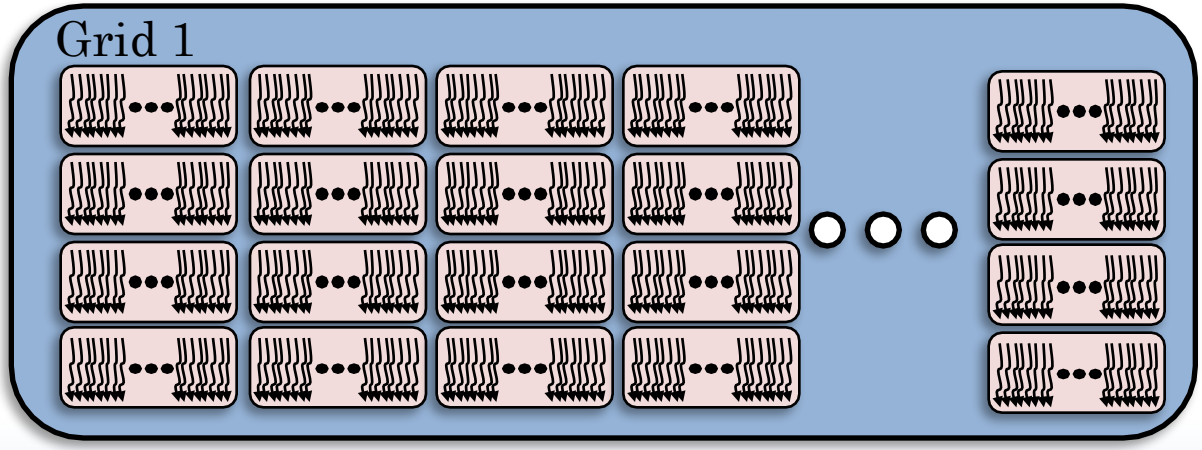
- Parallelism
  - massively-parallel/many-core architecture
    - needs a lot of work to be efficient
  - specialized hardware build for parallel tasks
  - more transistors mean more performance



- Multi-billion dollar gaming industry drives innovation

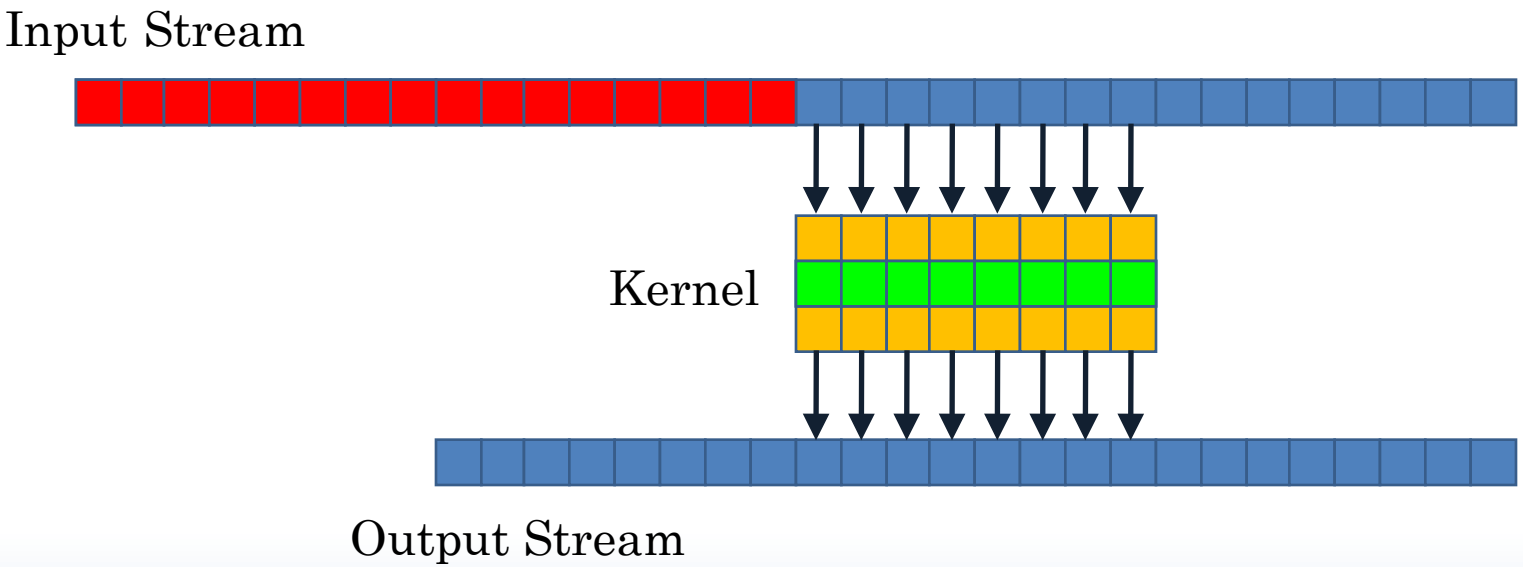
GPU Execution Model



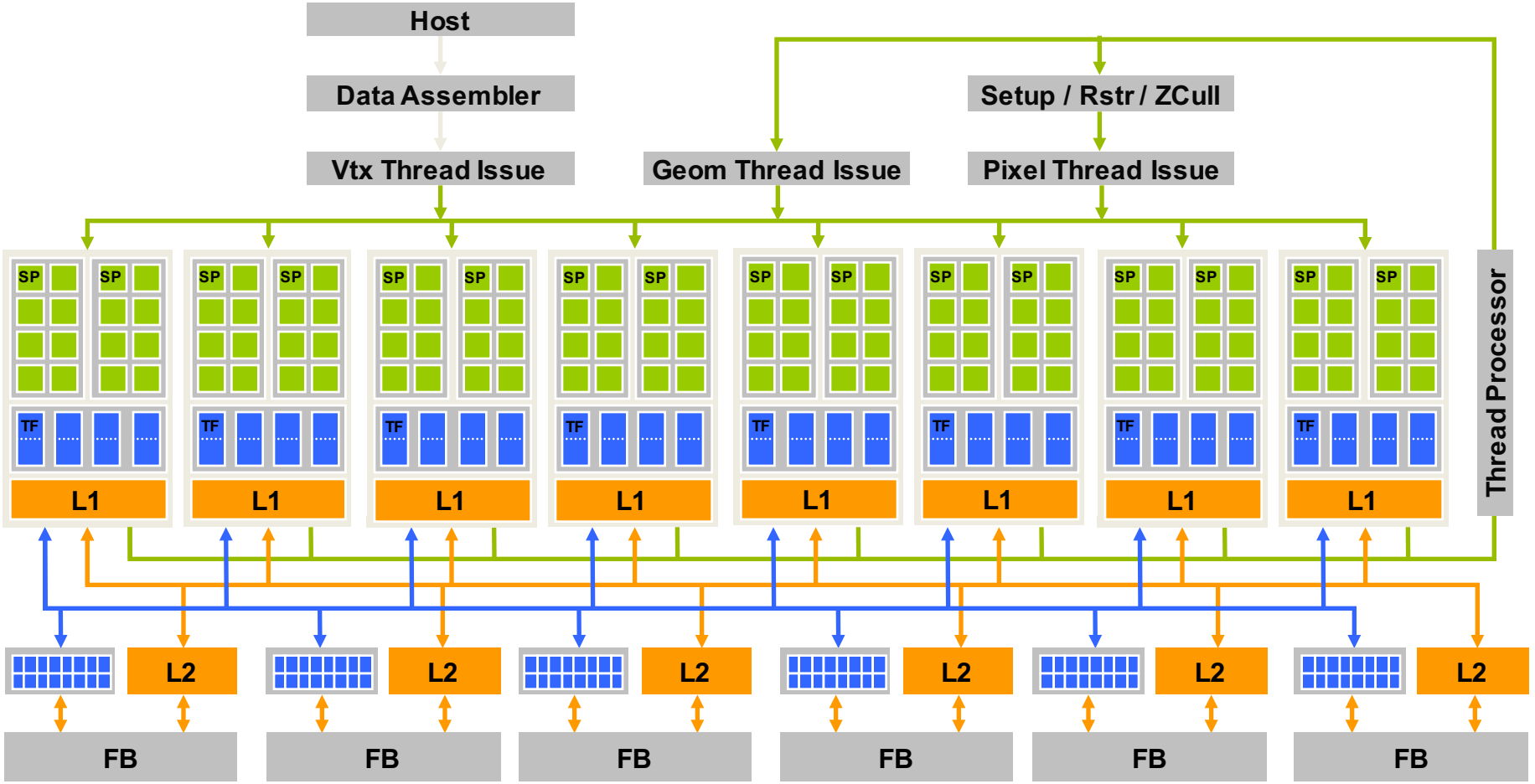


Computational Model: Stream Processing Model

- GPU is practically a stream processor
- Applications consist of streams and kernels
- Each kernel takes relatively long to process (PCIe, memory latency)
  - latency hidden by throughput



Inside a modern GPU



GPU differences from CPUs

- CPU cores optimized for latency, GPUs for throughput
  - CPUs: deep caches, sophisticated branch predictors
  - GPUs: transistors spent on many slim cores running in parallel
- SIMT execution
  - Work-items (logical threads) are partitioned into work-groups
  - The work-items of a work-group execute together in near lock-step
  - Allows several ALUs to share one instruction unit

Typically 256-1024 work-items per work-group

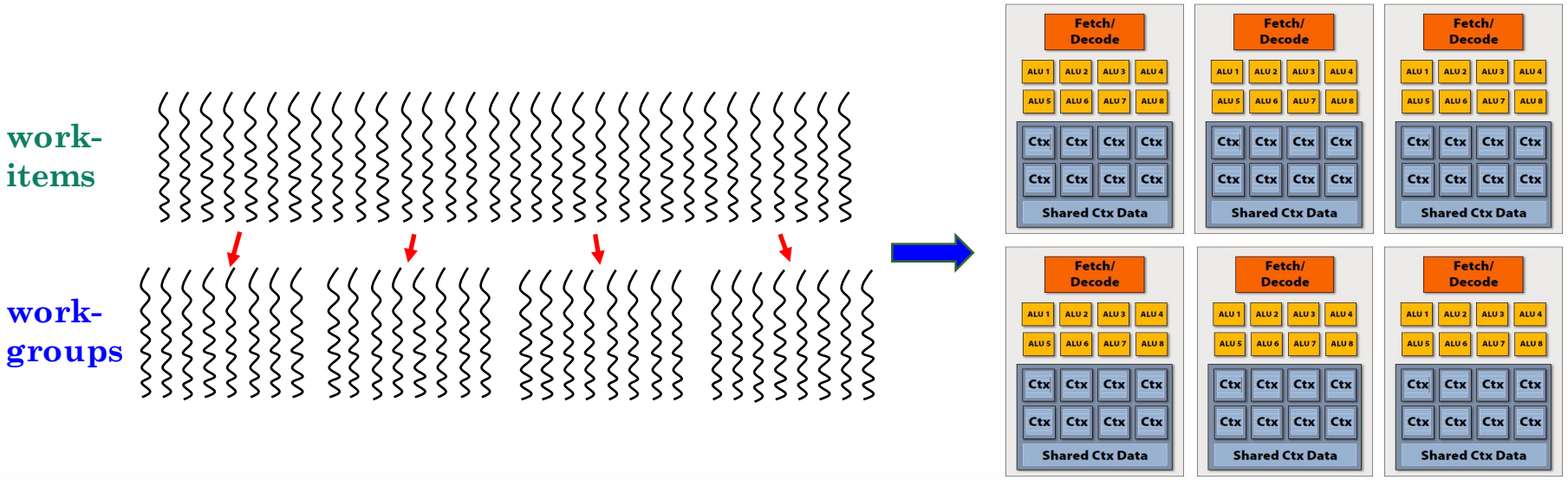


Figure by Kayvon Fatahalian, How Shader Cores Work – Beyond Programmable Shading

GPU differences from CPUs

- Shallow execution pipelines
- Low power consumption
- Highly multithreaded to hide memory latency
  - Assumes programs have a lot of parallelism
  - Switches execution to new work-group on a miss
- Separate high-speed local memory
  - Shared by work-items of an executing workgroup
  - Might, e.g., accumulate partial dot-products or reduction results
- Coalesced memory accesses
  - Reduces number of memory operations
- Execution barriers
  - Synchronize work-items in work-groups

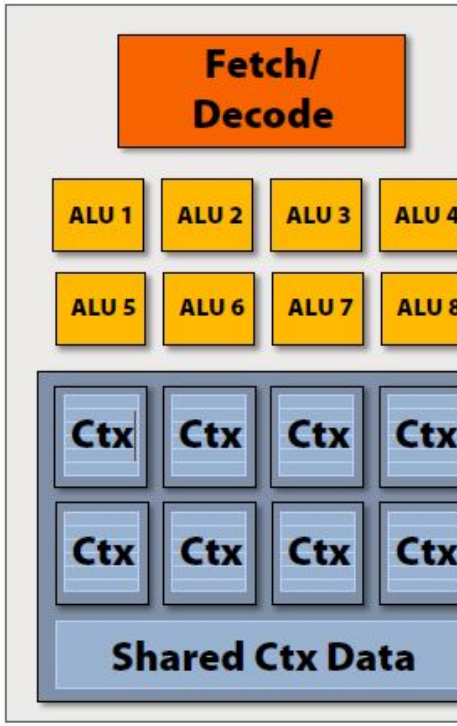


Figure by Kayvon Fatahalian, How Shader Cores Work – Beyond Programmable Shading

GPUs: but what about branches?

- Serially execute each branch path of a conditional branch
  - Too much branch divergence hurts performance

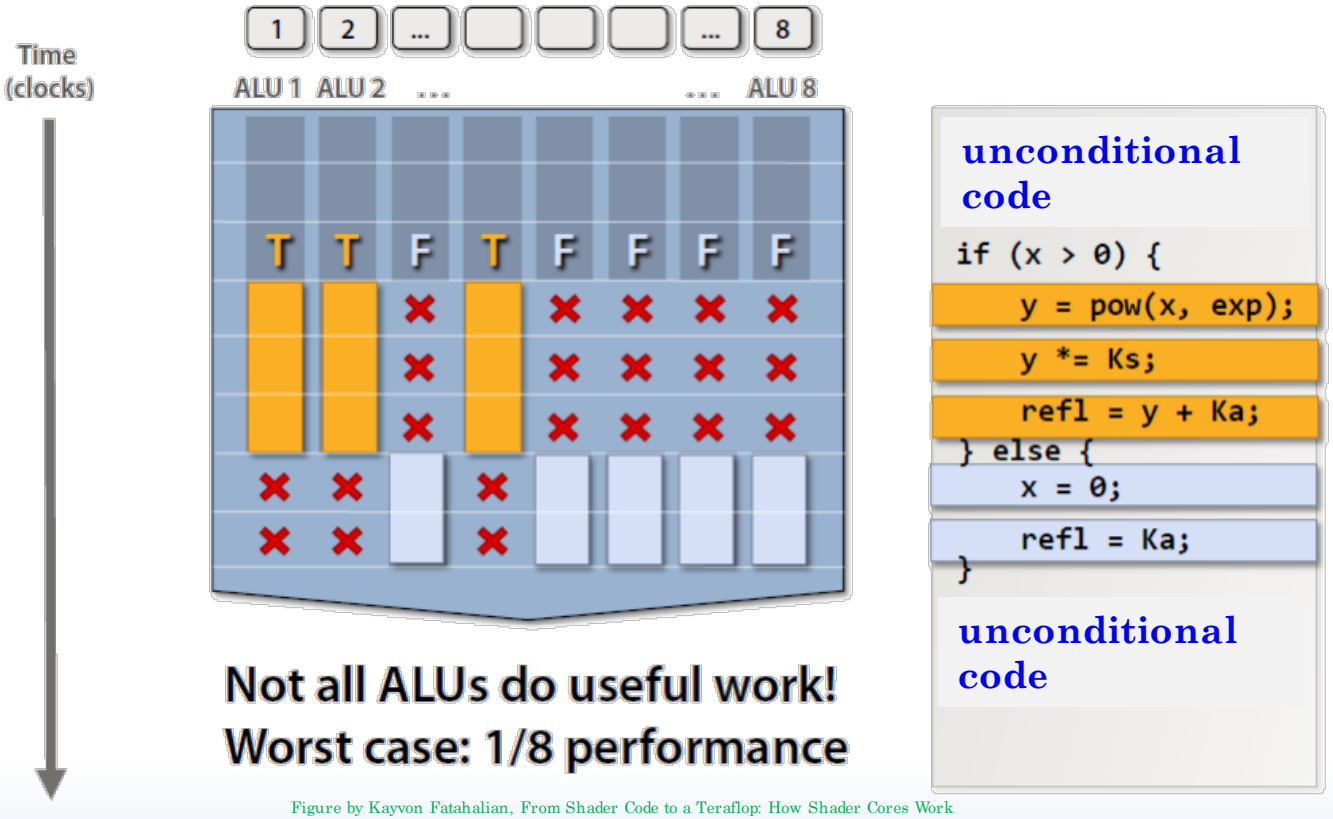


Figure by Kayvon Fatahalian, From Shader Code to a Teraflop: How Shader Cores Work

For good GPU performance

- Have enough parallelism
  - Too few work-items hurts memory latency hiding
- Choose appropriate work-group size
  - Want to keep all execution units fully utilized
- Use fast local memory
  - Has low latency and high bandwidth similar to an L1 cache
- Coalesce memory accesses when possible
  - Maximize memory bandwidth
- Minimize branch divergence

Programming models tied to GPU architecture
Performance favored over programmability
– Often little **performance portability**

Introducing GPGPU

- The GPU on commodity video cards has evolved into a processor that is
  - powerful
  - flexible
  - inexpensive
  - precise

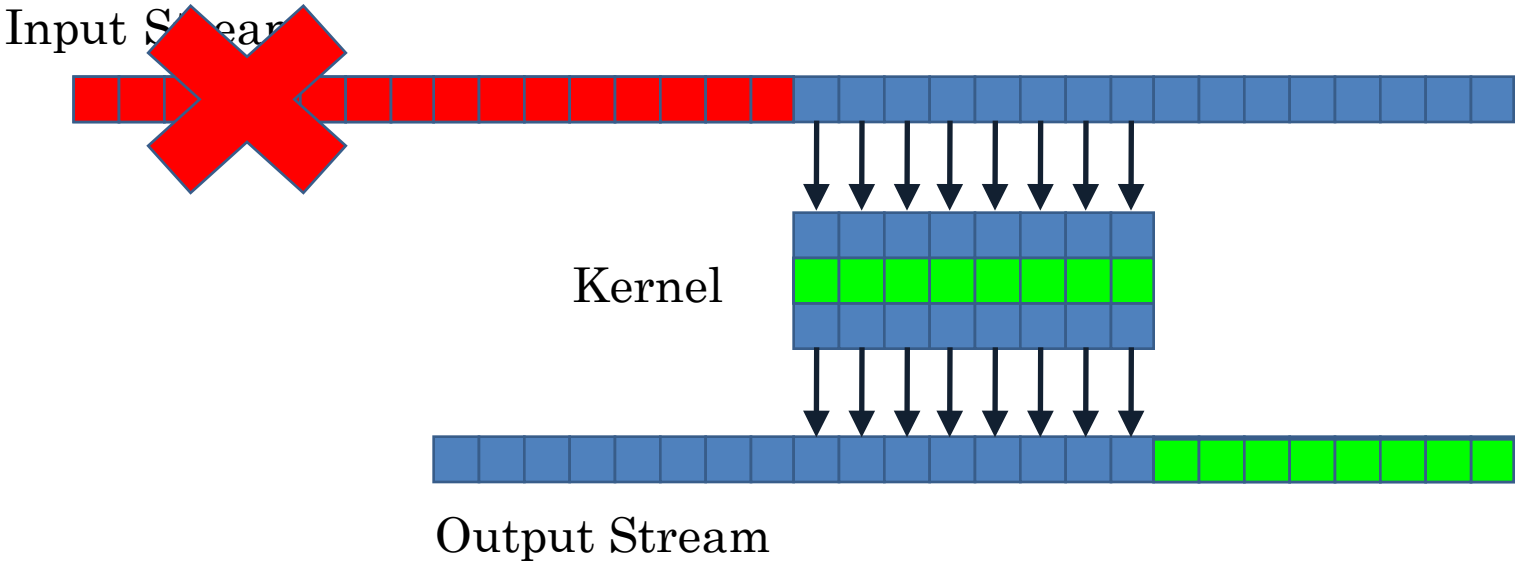
Attractive platform for general-purpose computation

GPGPU

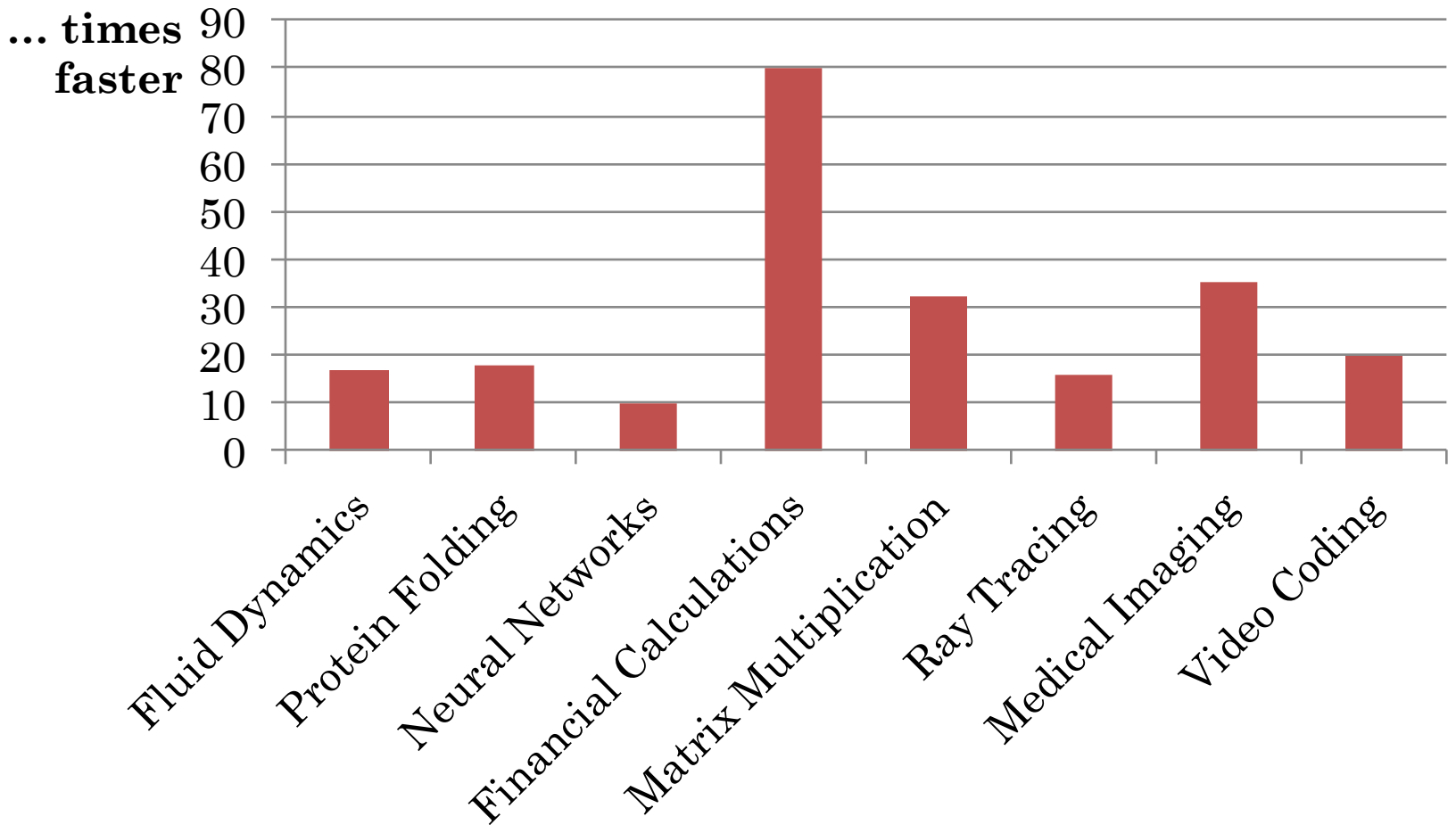
- General-Purpose GPU
  - use the GPU for general-purpose algorithms
- No magical GPU compiler
  - explicit mappings required using advanced APIs
  - Intel Xeon Phi
- Programming close to the hardware
  - trend for higher abstraction, i.e. NVIDIA CUDA
- Techniques are suited for future many-core architectures
  - future CPU/GPU projects, AMD Fusion, Larrabee, …
- Dependency issues
  - hundreds of independent tasks required for efficient use

Stream Processing Model Revisited

- GPU is practically a stream processor
- Applications consist of streams and kernels
- Read back is not possible



GPGPU in Practice



GPGPU APIs

- Classic way
  - (mis)use graphics pipeline
    - render a special 'scene'
  - Direct3D, OpenGL
  - pixel, geometry, and vertex shaders

- New APIs specifically for GPGPU computations
  - NVIDIA CUDA, ATI CTM, DirectX11 Compute Shader, OpenCL

GPGPU programming: SIMT model

- CPU ("host") program often written in C or C++
  - The CPU specifies number of work-items & work-groups, launches GPU work, waits for events & GPU results
- GPU code is written as a sequential kernel in (usually) a C or C++ dialect
  - All work-items execute the same kernel
  - HW executes kernel at each point in a problem domain

E.g., process 1024x1024 image with 1,048,576 work-items

Traditional loops

```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

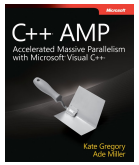
Data-Parallel OpenCL

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```

Credit: Khronos Group, OpenCL Overview

GPGPU programming: frameworks

- OpenCL
- CUDA
- C++ AMP
- Renderscript



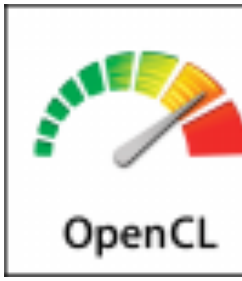
Lower-level performance frameworks

Higher-level productivity frameworks

These differ in
- the capabilities they provide
- how much control they give programmers
- performance portability

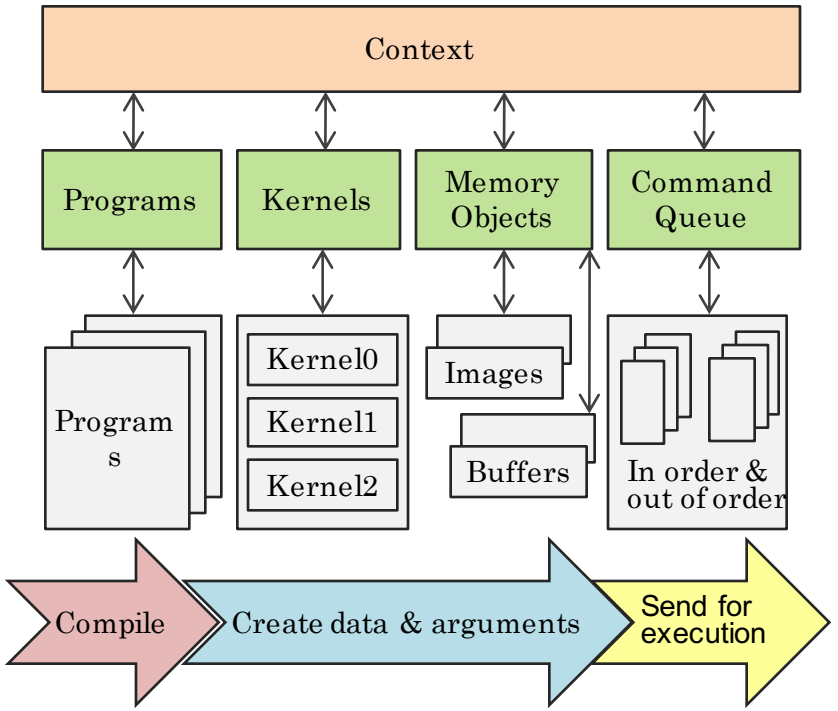
OpenCL

- Cross-platform, cross-vendor standard for parallel & heterogeneous computing
- Host (CPU) API
  - Query, select and initialize compute devices (GPU, CPU, DSP, accelerators)
  - May execute compute kernels across multiple devices
- Kernels
  - Basic unit of executable offloaded code
  - Built-in kernels for fixed-functions like camera pipe, video encode/decode, etc.
- Kernel Language Specification
  - Subset of ISO C99 with language extensions
  - Well-defined numerical accuracy: IEEE 754 rounding with specified max error



OpenCL basics: executing programs

- Query for OpenCL devices
- Create context for selected devices
- Select kernels
- Create memory objects
- Copy memory objects to devices
- Enqueue kernels for execution
- Copy kernel results back to host



http://www.slideshare.net/Khronos_Group/open-cl-overviewsiggraphasianov13

OpenCL memory & work-items

- OpenCL 1.2: explicit memory management
  - Application must move data from host → global → and back
- Work-items/work-groups
- C99 kernel language restrictions
  - No recursion since often no HW call stack
  - No function pointers

Work-group example

Work-items = # pixels
Work-groups = # tiles
Work-group size = (tile width * tile height)

http://www.slideshare.net/Khronos_Group/open-cl-overviewsiggraphasianov13

OpenCL 2.0 changes

- Goals: ease of use & performance improvements
- Shared Virtual Memory (SVM)
  - OpenCL 2.0: SVM required
  - Three kinds of sharing:
    - Coarse-grain buffer sharing: pointer sharing in buffers
    - Fine-grain buffer sharing
    - Fine-grain system sharing: all memory shared with coherency
  - Fine-grain system sharing
    - Can directly use any pointer allocated on the host (malloc/free), no need for buffers
    - Both host & devices can update data using optional C11 atomics & fences
- Dynamic Parallelism
  - Allows a device to enqueue kernels onto itself – no round trip to host required
  - Provides a more flexible execution model
    - A very common example: kernel A enqueues kernel B, B decides to enqueue A again, …

OpenCL 2.0 changes

- C11 atomics
  - Coordinate access to data accessed by multiple agents
  - Atomic loads/stores, compare & exchange, fences …
- OpenCL memory model
  - With SVM and coherency, even more potential for data races
  - Based on the C11 memory model
  - Specifies which memory operations are guaranteed to happen in which order &
    which memory values each read operation will return
    - Supports OpenCL global/local memory, barriers, scopes, host API operations, …

Other GPGPU frameworks

- CUDA
  - Similar to OpenCL
  - Kernel language is C++ subset, no cross-device atomics
  - SVM similar to coarse-grain buffer SVM
    - special allocation APIs, special pointers, non-coherent
- C++ AMP (Accelerated Massive Parallelism)
  - STL-like library for multidimensional array data
    - Runtime handles CPU<->GPU data copying
  - parallel_for_each
    - Executes a C++ lambda at each point in an extent, tiles
  - restrict specifies where to run the kernel: CPU or GPU
- Renderscript
  - Emphasis on mobile devices & performance portability
    - Programmer can't control where kernels run, VM-decided
  - Kernel code is C99-based
    - 1D and 2D arrays, types include size, runtime type checking
  - Script groups fuse kernels for efficient invocation

Performance
- **More control**
- **Often better performance**

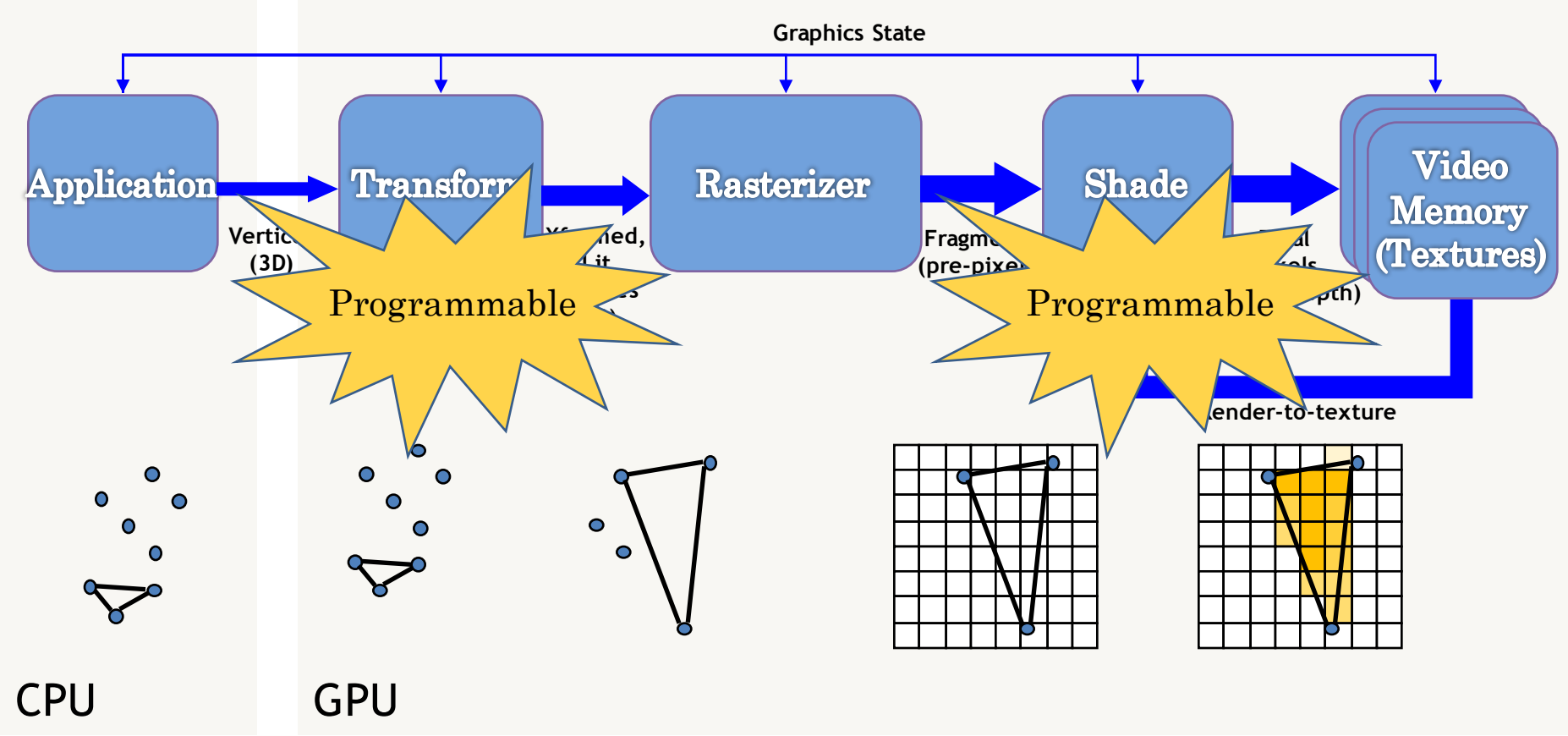
Productivity
- **Ease of use**
- **Runtime checking**
- **More performance portability**

Overview

- Introduction
  - GPU
  - GPGPU
- Programming Concepts and Mappings
  - Direct3D and OpenGL
  - NVIDIA CUDA
- Case Study: Decoding H.264/AVC
  - motion compensation
  - results
- Conclusions
- Q&A

3-D Pipeline

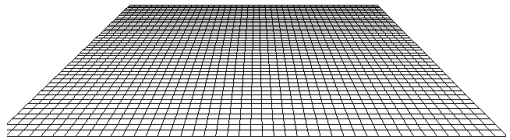
- Deep pipeline



3-D Pipeline - Transform

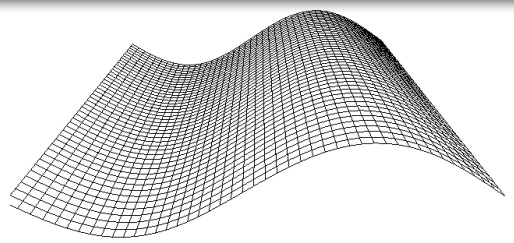
- Vertex Shader
  - processing geometry data
  - input is a vertex
    - position
    - texture coordinates
    - vertex color,...
  - output is a vertex

```
struct Vertex
{
  float3 position : POSITION;
  float4 color    : COLOR0;
};
```



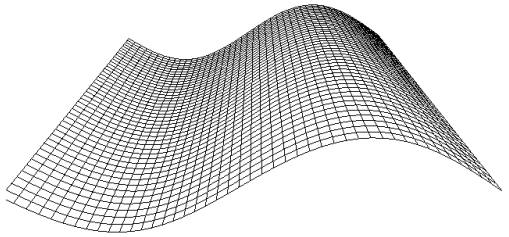
```
Vertex wave(Vertex vin)
{
   Vertex vout;
   vout.x = vin.x;
   vout.y = vin.y;
   vout.z = (sin(vin.x) +
             sin(IN.wave.x)) * 2.5f;

   vout.color = float4(1.0f, 1.0f,
                       1.0f, 1.0f);
   return vout;
}
```

Vertex Shader



3-D Pipeline - Shading

- Pixel (or fragment) Shader
  - input is interpolated vertex data
    - position
    - texture coordinates
    - normals, …
    - pixel color
  - result is stored in the frame buffer or in a texture
    - 'Render to Texture'



```
struct PSIn
{
    float2 tex; : TEXCOORD0
};
```

```
struct PSOut
{
    float4 color; : COLOR0
};
```

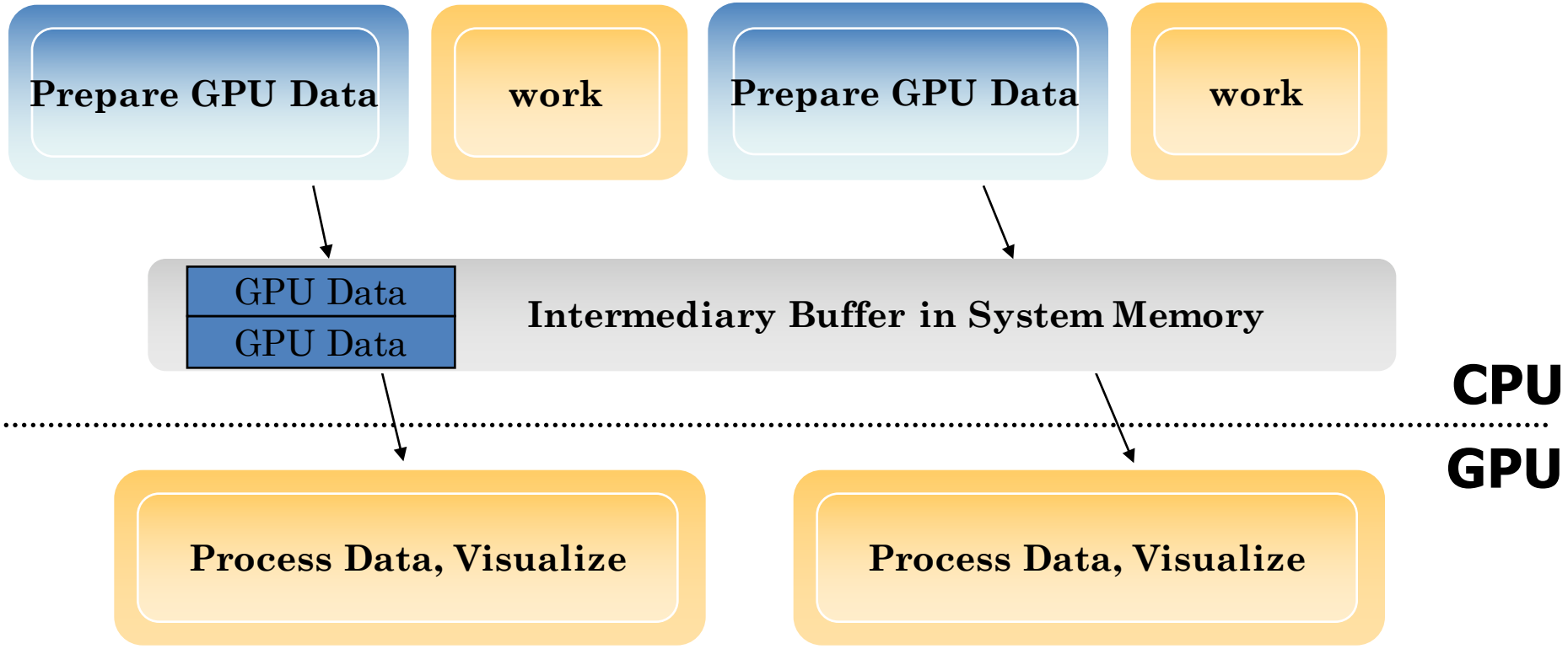
```
PSOut shade(PSIn pin)
{

    PSOut pout;
    pout.color = tex(pin.tex, sampler)

    return pout;

}
```

Pixel Shader

GPU-CPU Analogies

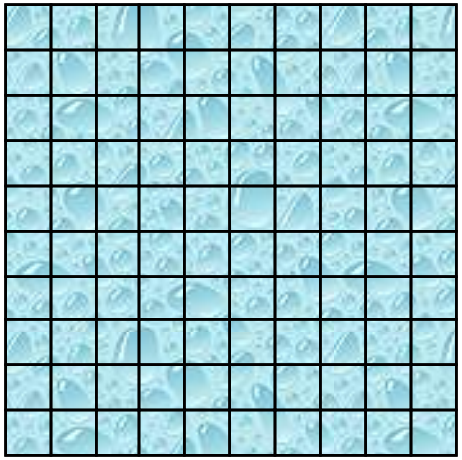
- Explicit mapping on 3-D concepts is necessary
- Rewrite an algorithm and find parallelism
- Use the GPU in parallel to the CPU
  - upload data to the GPU
    - very fast PCI-Express bus, up to 2GB/sec transfer speed
  - process the data
    - meanwhile the CPU is available
  - download result to the CPU
    - recent GPU models have high download speed

GPU-CPU Pipelined Design



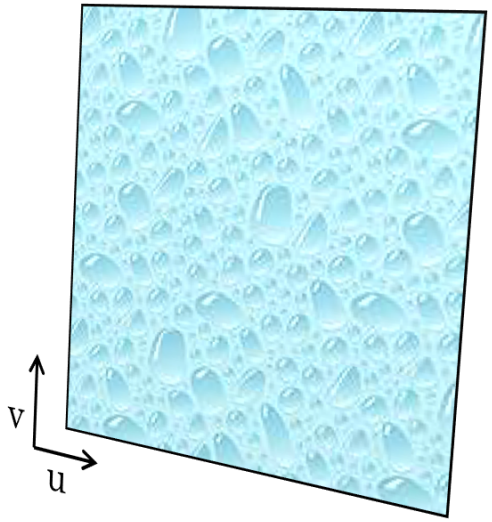
GPU-CPU Analogies (2)

CPU



Array

GPU



Texture

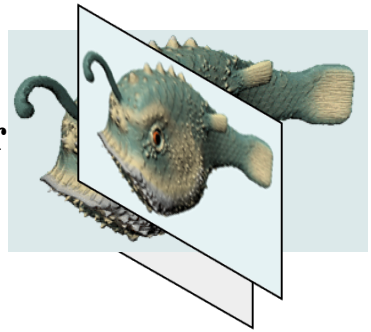
GPU-CPU Analogies (3)

CPU

```
        …
  fish[] = createfish()
        …
for all pixels
 bwfish[i][j]= bw(fish[i][j]);
        …
```

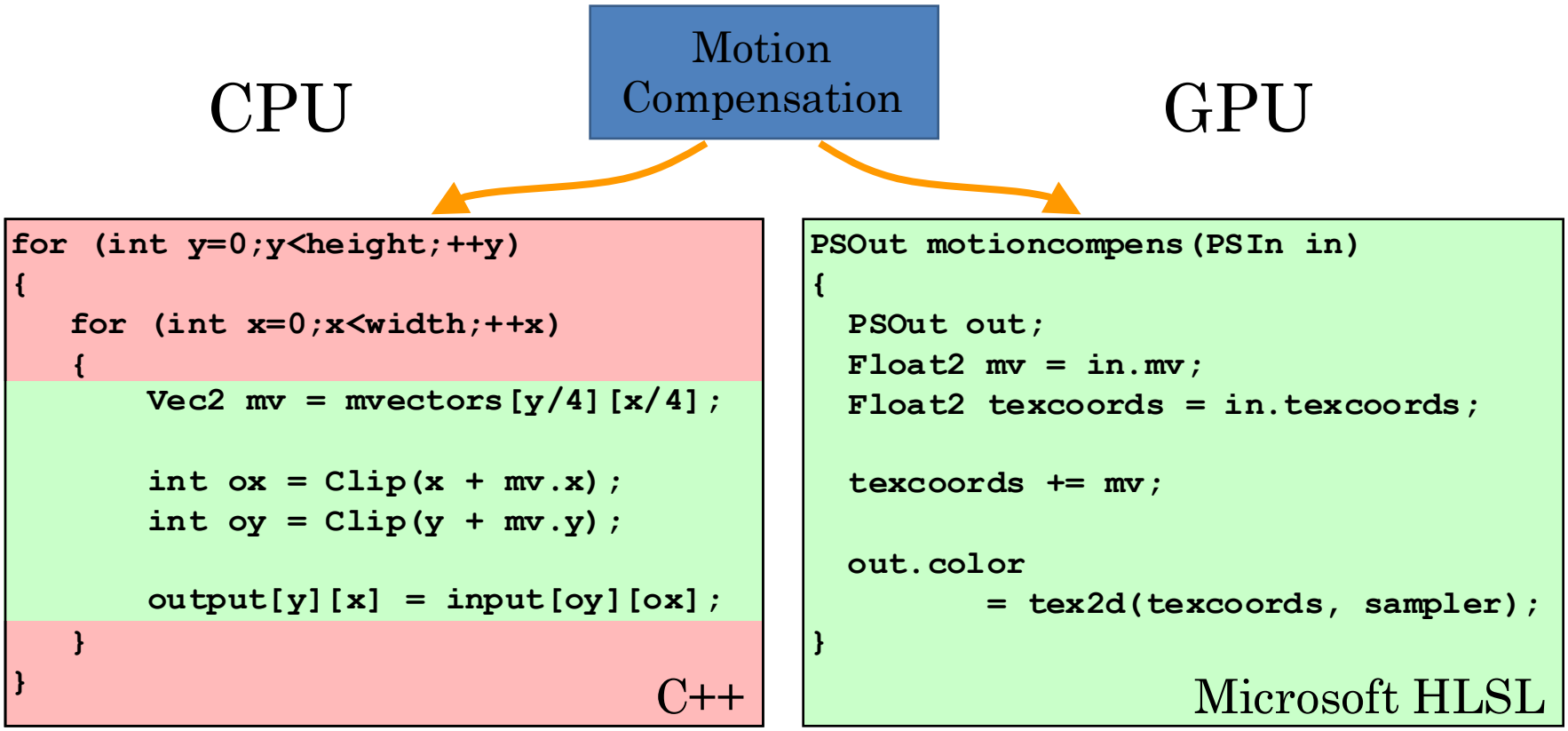
GPU

Render



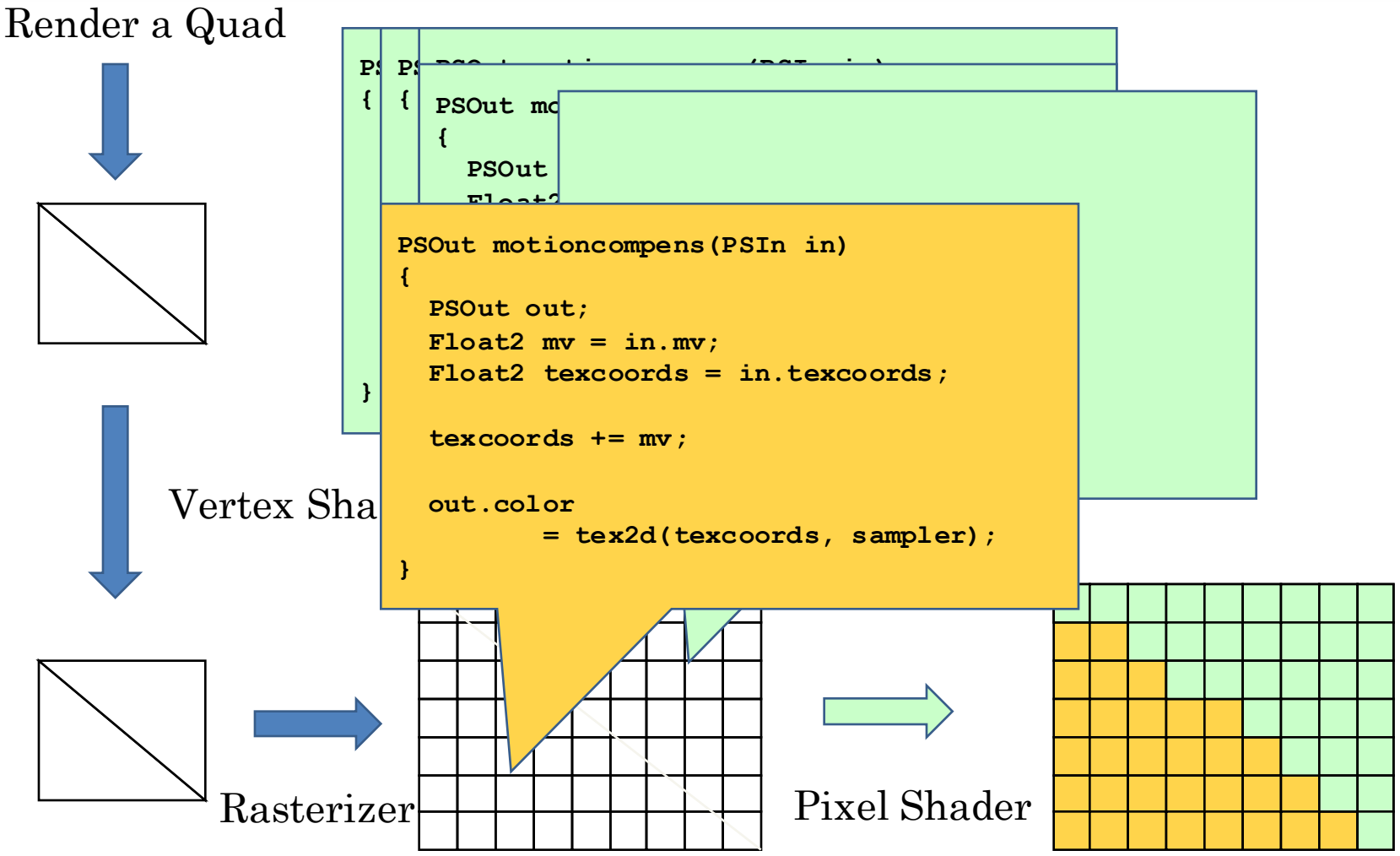
Array Write = Render to Texture

GPU-CPU Analogies (4)



Loop body / kernel / algorithm step = Fragment Program

GPU Loop for Each Pixel



Overview

- Introduction
  - GPU
  - GPGPU
- Programming Concepts and Mappings
  - Direct3D and OpenGL
  - NVIDIA CUDA
- Case Study: Decoding H.264/AVC
  - motion compensation
  - results
- Conclusions
- Q&A

GPGPU-specific APIs

- NVIDIA CUDA
  - ***C**ompute **U**nified **D**evice **A**rchitecture*
  - C-code with annotations compiled to executable code
- DirectX 11 Compute Shader
  - shader execution without rendering
  - technology preview available in latest DirectX SDK
- OpenCL
  - **Open C**omputing **L**anguage
  - C++-code with annotations
- ATI CTM
  - **C**lose to **T**he **M**etal
  - GPU assembler
  - depricated

NVIDIA CUDA

- NVIDIA CUDA
  - **C**ompute **U**nified **D**evice **A**rchitecture
  - C-code with annotations compiled to executable code
- Supported on NVIDIA G80 and higher
- No more (mis)use of 3-D API
- C-code with annotations for
  - memory location
  - host or device functions
  - thread synchronization
- Compilation with CUDA-compiler
  - split host and device code
  - linkable object code

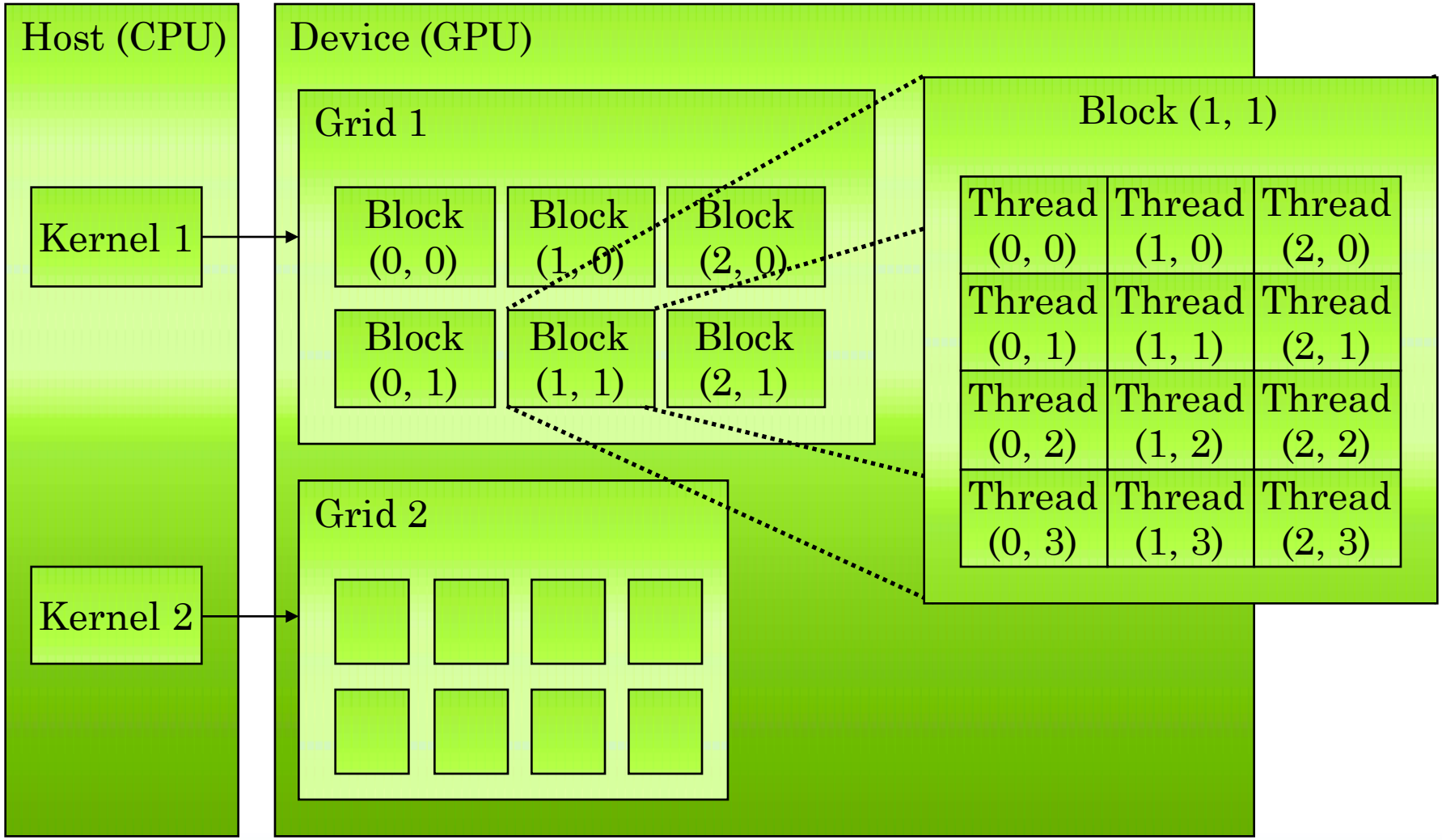
NVIDIA CUDA - Example

```
void runGPUTest()
{
    CUT_DEVICE_INIT();

    ...
    float* d_data = NULL;

    // allocate gpu memory
    cudaMalloc( (void**) &d_data, size);

    dim3 dimBlock(8, 8, 1);
    dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);

    // run kernel on gpu
    transformKernel<<< dimGrid, dimBlock, 0 >>>( d_data );

    // download
    cudaMemcpy( h_data, d_data, size, cudaMemcpyDeviceToHost);

    ...
}
```

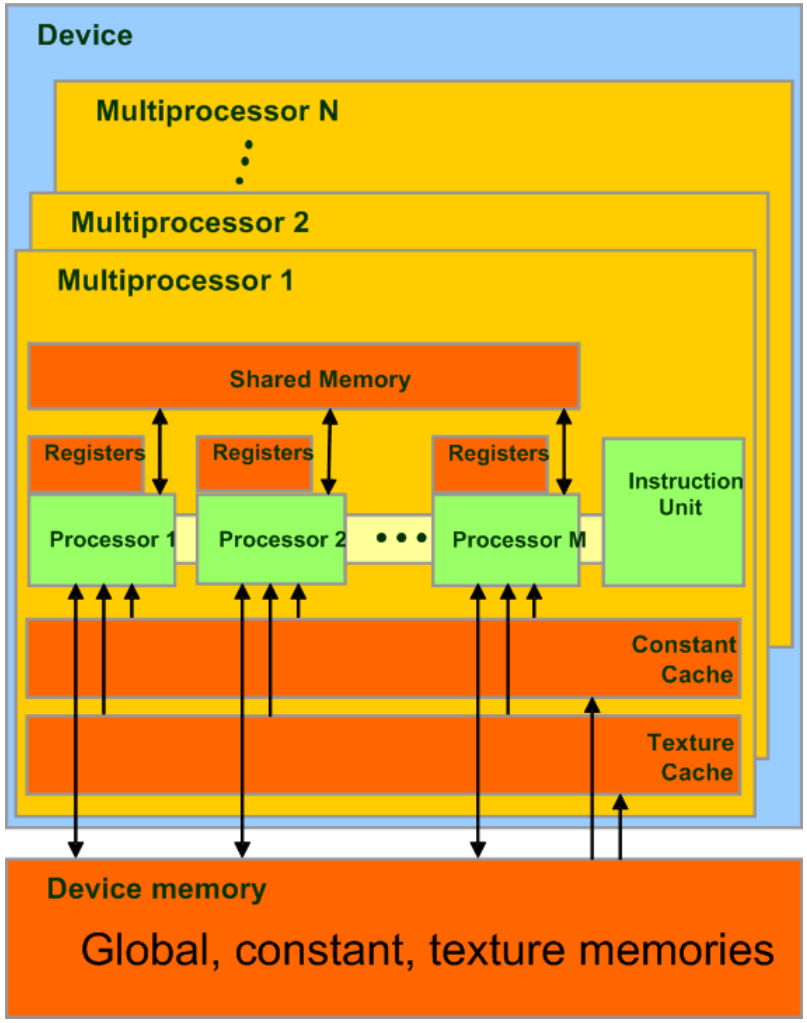
NVIDIA CUDA – Example (2)

```
__global__ void transformKernel( float* g_odata)
{
  // calculate normalized texture coordinates
  unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
  unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;

  int2 mv = tex2D(mvtex, x, y);

  int mx = x + mv.x;
  int my = y + mv.y;

  g_odata[y*width + x] = tex2D(reftex, mx, my);
}
```

Programming Model



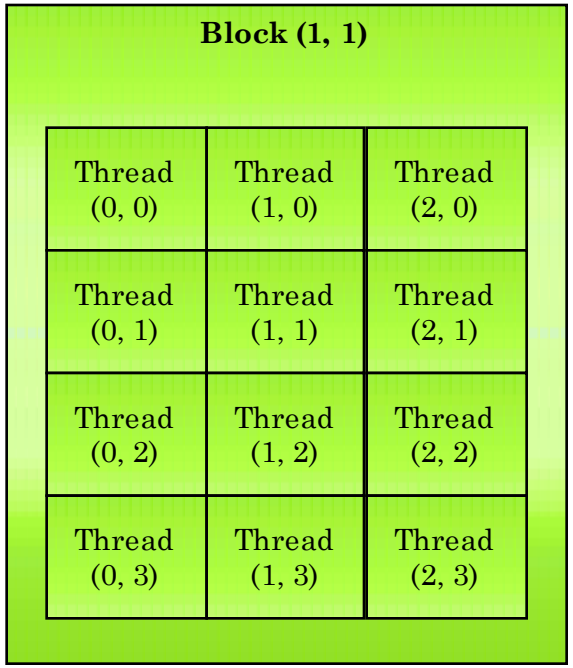
Hardware Model



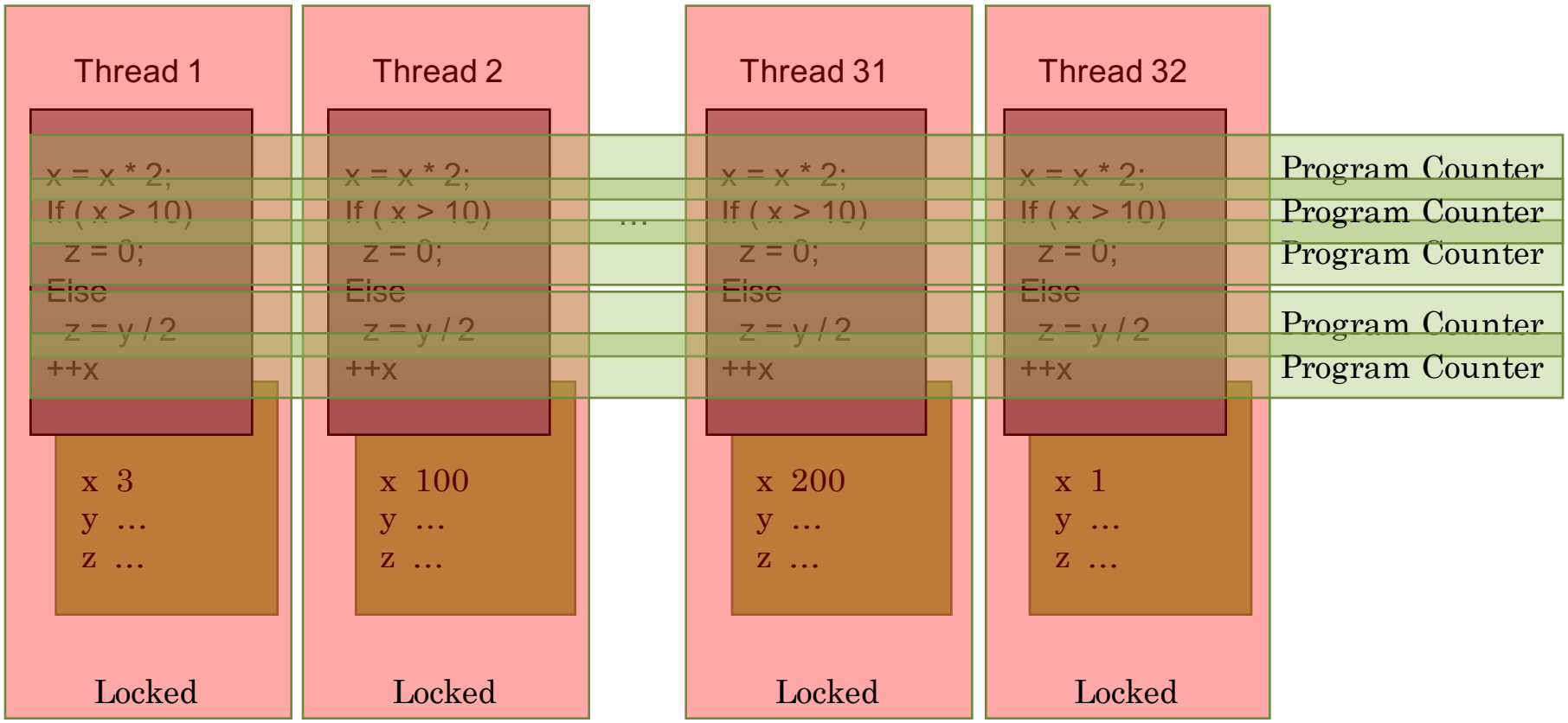
- Multiprocessor – MP (16)
- Streaming Processor (8 per MP)
  - handles one thread
- Memory
  - very fast
  - **uncached**
  - special memory hardware for constants & texture (cached)
- Registers
  - limited amount

CUDA Threads

- Each Streaming Processor handles one thread
  - **240** on GeForce 280GTX!
- Smart hardware can schedule thousands of threads on 240 processors
- Extremely **lightweight**
  - not like CPU threads
- Threads per Multiprocessor handled in **SIMD** manner
  - each thread executes the same instruction at a given clock cycle
  - lock-step execution in MP



Lock-step Execution



- Heavy branching needs to be avoided

Buzzword: Kernel

- In CUDA, a ***kernel*** is code (typically a function) that can be run inside the GPU.
- Typically, the kernel code operates in lock-step on the stream processors inside the GPU.

Buzzword: Thread

- In CUDA, a ***thread*** is an execution of a kernel with a given index.
- Each thread uses its index to access a specific subset of the elements of a target array, such that the collection of all threads cooperatively processes the entire data set.
- So these are very much like threads in the OpenMP or pthreads sense – they even have shared variables and private variables.

Buzzword: Block

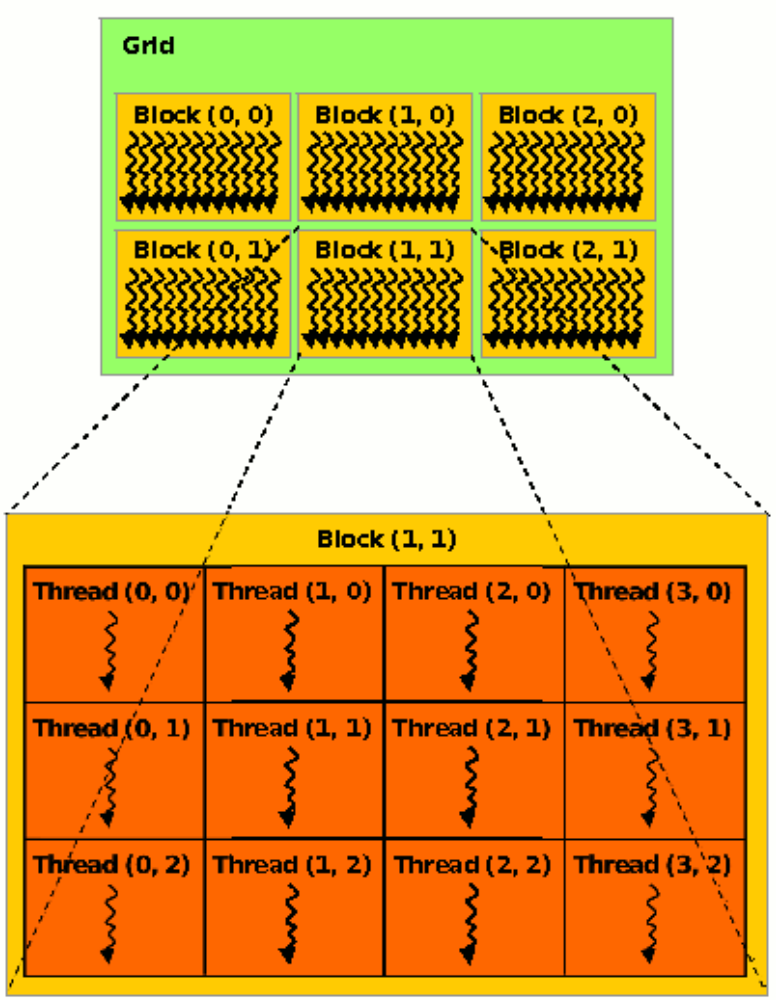
- In CUDA, a ***block*** is a group of threads.
- Just like OpenMP threads, these could execute concurrently or independently, and in no particular order.
- Threads can be coordinated somewhat, using the **_syncthreads()** function as a barrier, making all threads stop at a certain point in the kernel before moving on en mass. (This is like what happens at the end of an OpenMP loop.)

Buzzword: Grid

- In CUDA, a ***grid*** is a group of (thread) blocks, with no synchronization at all among the blocks.

NVIDIA GPU Hierarchy

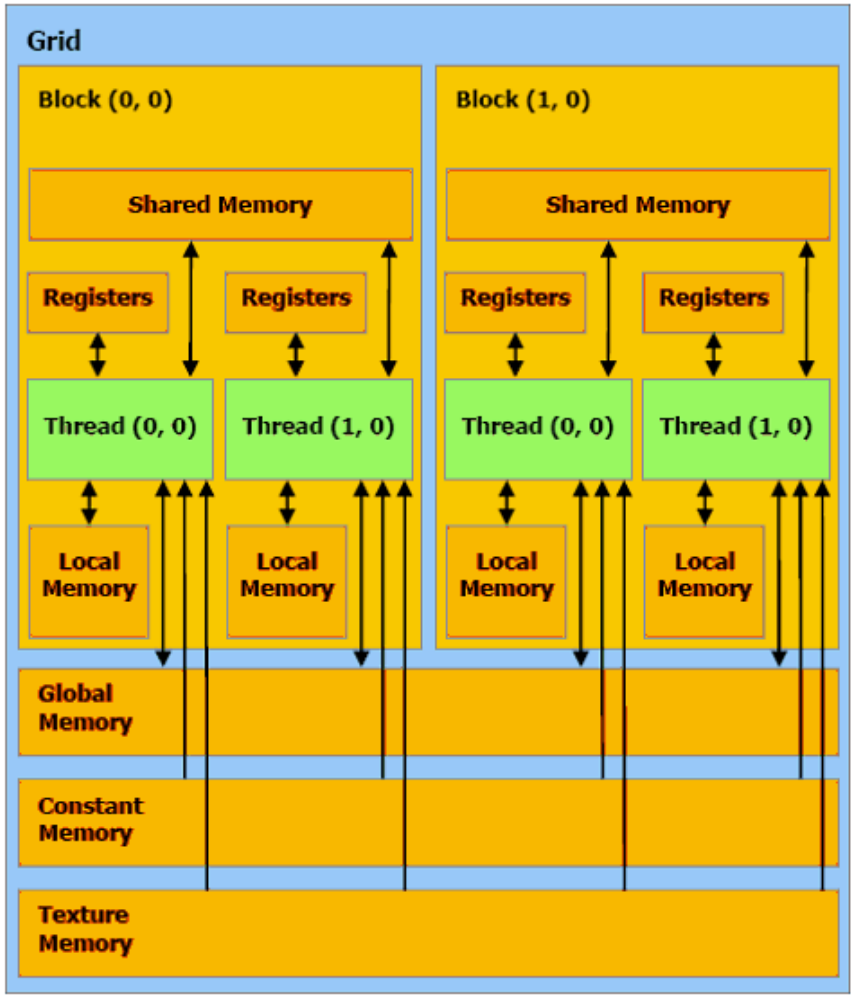
- ***Grids*** map to GPUs
- ***Blocks*** map to the MultiProcessors (MP)
  - Blocks are never split across MPs, but an MP can have multiple blocks
- ***Threads*** map to Stream Processors (SP)
- ***Warps*** are groups of (32) threads that execute simultaneously



CUDA Memory Hierarchy #1

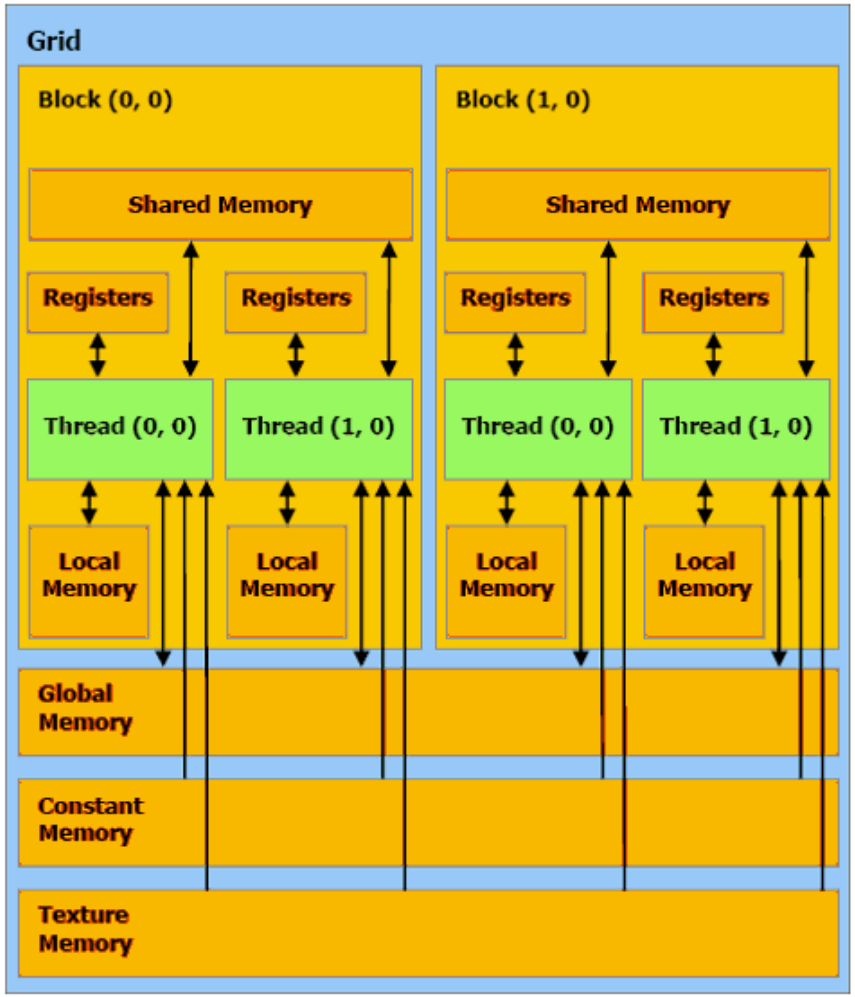
CUDA has a hierarchy of several kinds of memory:

- Host memory (x86 server)
- Device memory (GPU)
  - ***Global***: visible to all threads in all blocks – largest, slowest
  - ***Shared***: visible to all threads in a particular block – medium size, medium speed
  - ***Local***: visible only to a particular thread – smallest, fastest



CUDA Memory Hierarchy #2

- ***Constant***: visible to all threads in all blocks; read only
- ***Texture***: visible to all threads in all blocks; read only



Overview

- Introduction
  - GPU
  - GPGPU
- Programming Concepts and Mappings
  - Direct3D and OpenGL
  - NVIDIA CUDA
- Case Study: Decoding H.264/AVC
  - motion compensation
  - results
- Conclusions
- Q&A

Decoding H.264/AVC

- Many decoding steps are suitable for parallelization
  - quantization
  - transformation
  - motion compensation
  - deblocking
  - color space conversion
- Others introduce dependencies
  - entropy coding
  - intra prediction

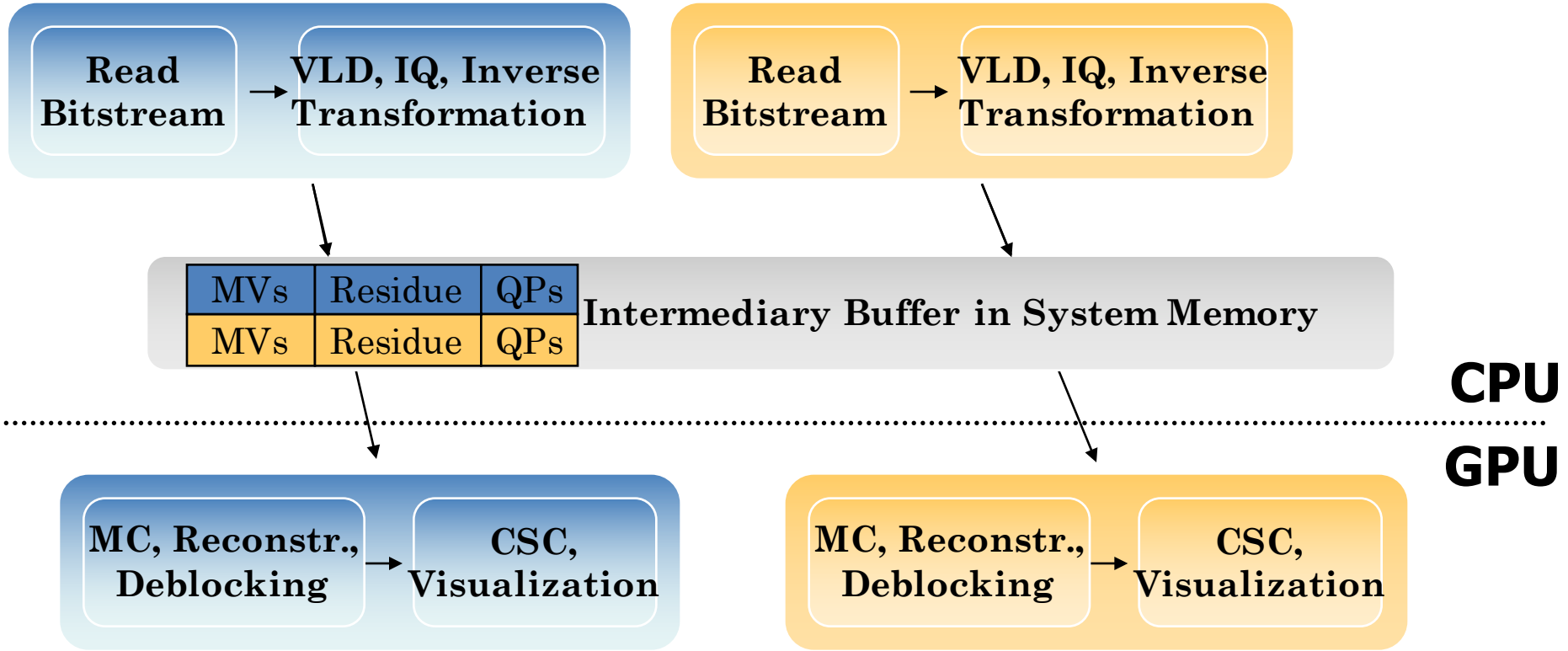
Video Coding Hardware

- Specialized on-board 2-D video processing chips
  - one macroblock at the time
  - black boxes
    - limited support for non-windows systems
  - limited support for various video codecs
    - e.g. H.264/AVC profiles
  - partly programmable
- GPU
  - millions of transistors
  - accessible via 3-D API or General Purpose GPU API

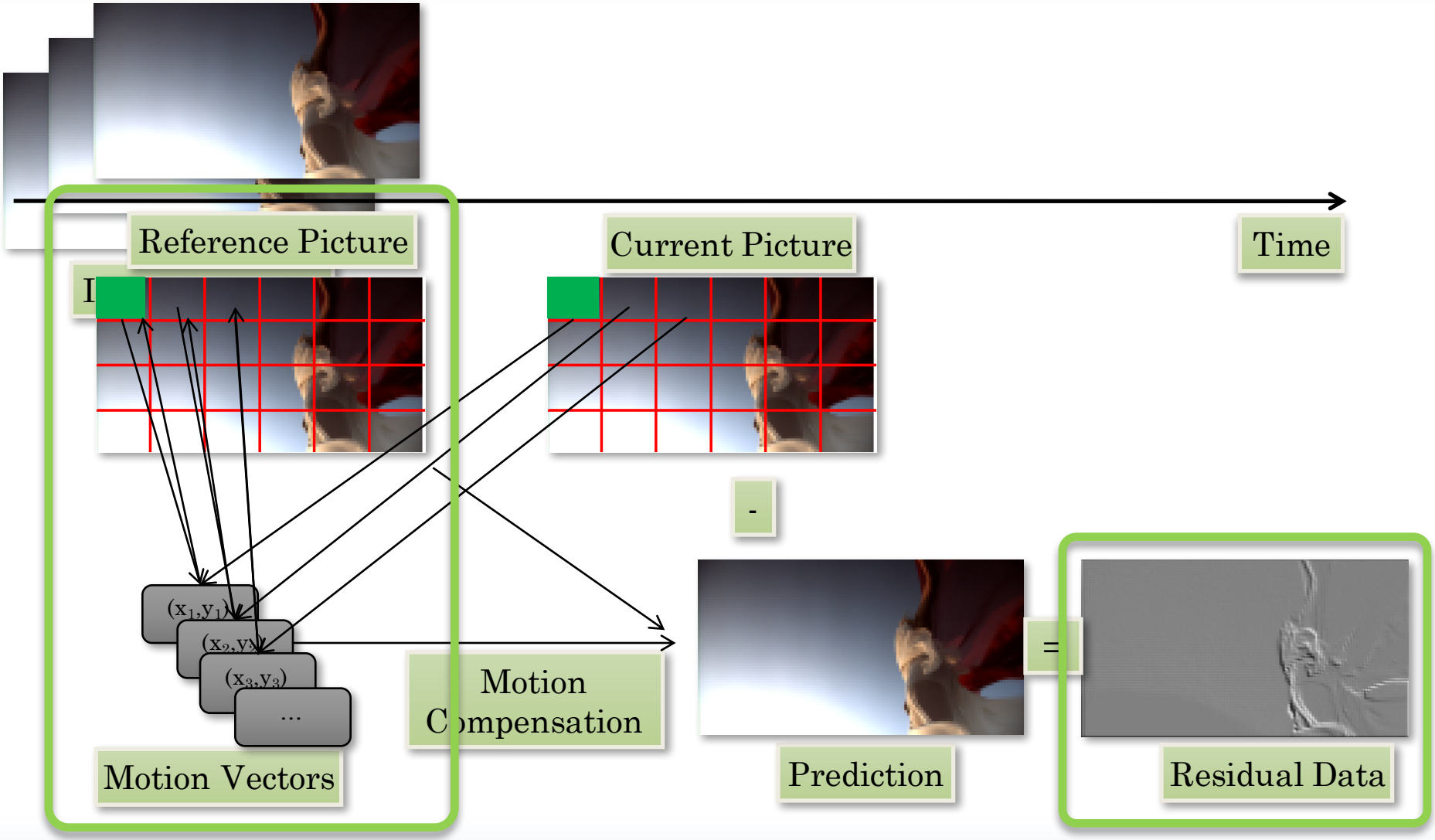
Decoding an H.264/AVC bitstream

- H.264/AVC
  - recent video coding standard
  - successor of MPEG-4 Visual
- Computationally intensive
  - multiple reference frames (up to 16)
  - B-pictures
  - sub-pixel interpolations
- Motion compensation, reconstruction, deblocking, and color space conversion
  - takes up to 80% of total processing time
  - suitable for execution on the GPU

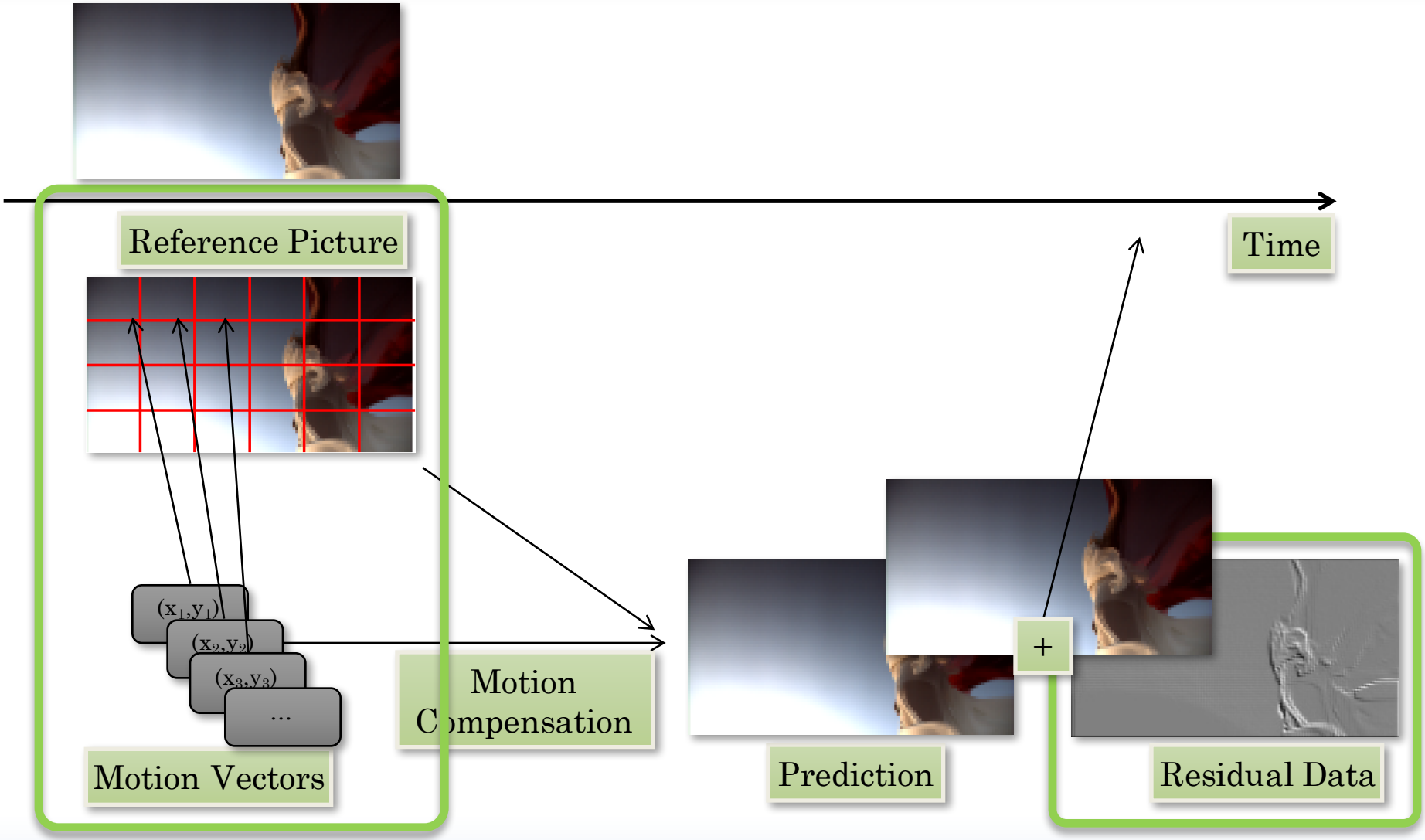
Pipelined Design for Video Decoding



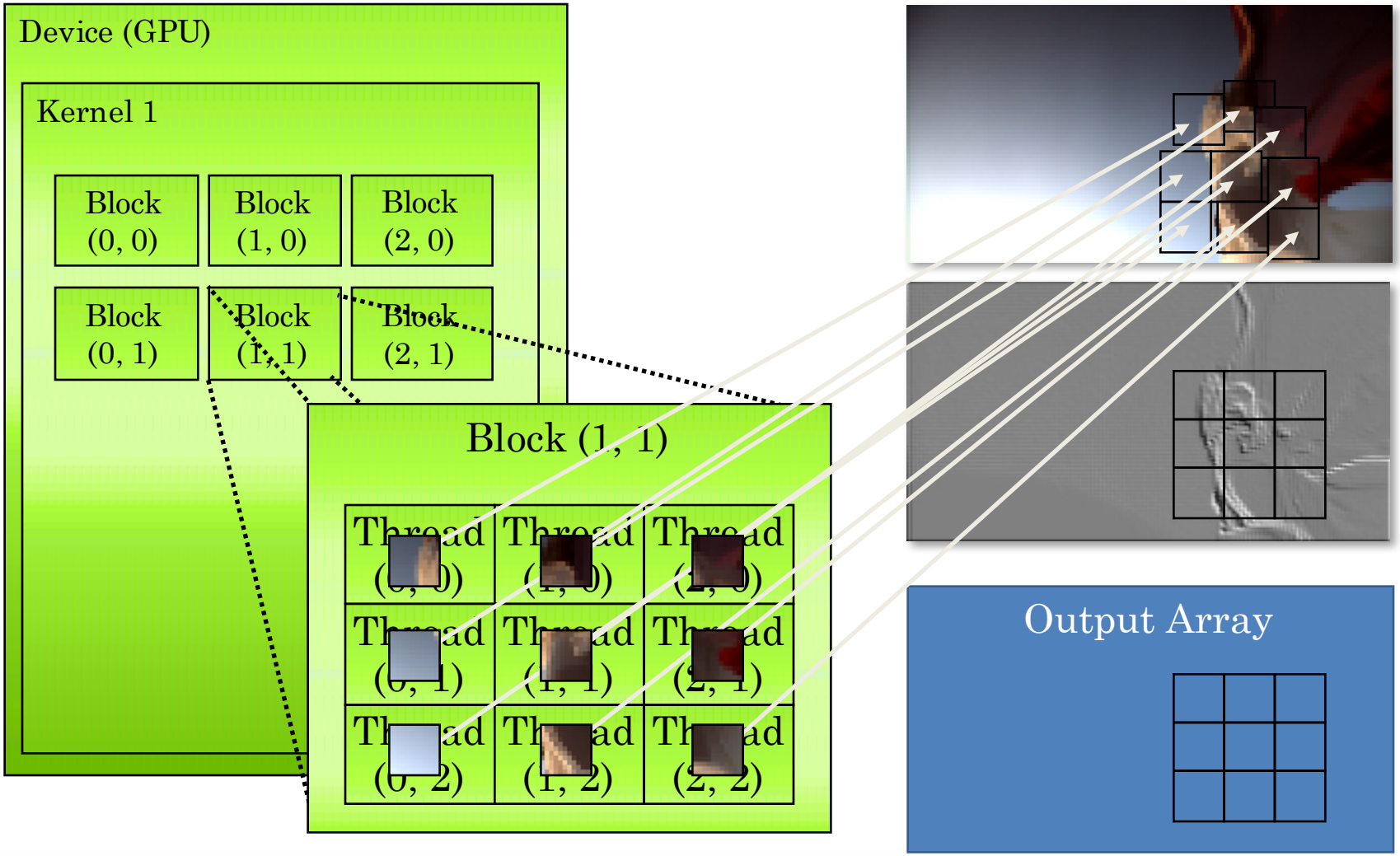
Motion Compensation



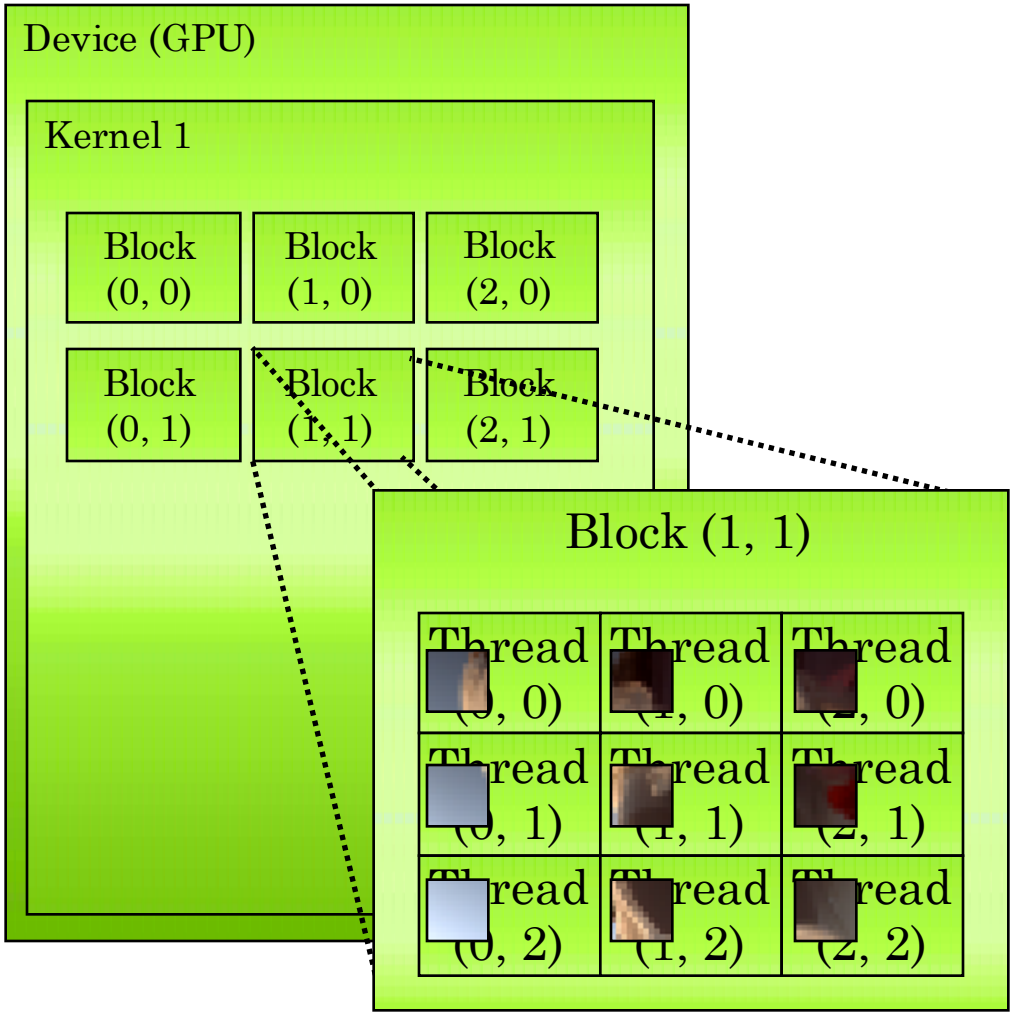
Motion Compensation: Decoder

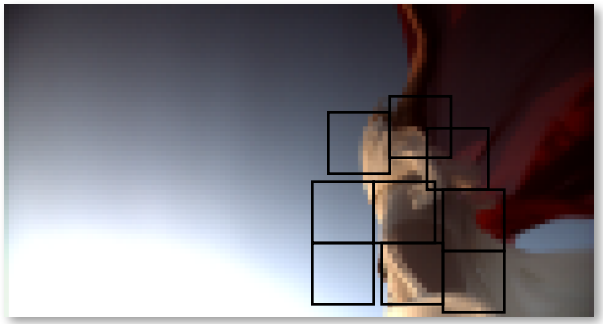


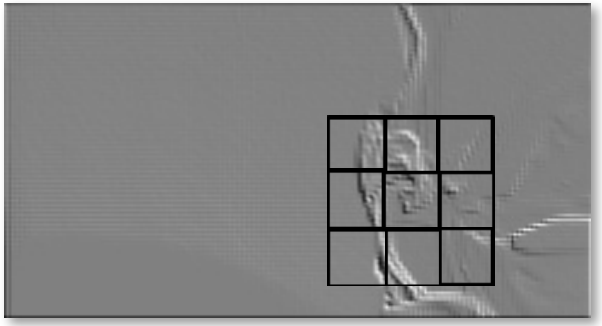
Motion Compensation in CUDA

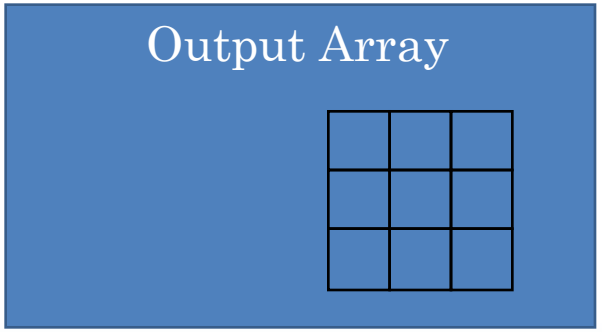


Motion Compensation in CUDA

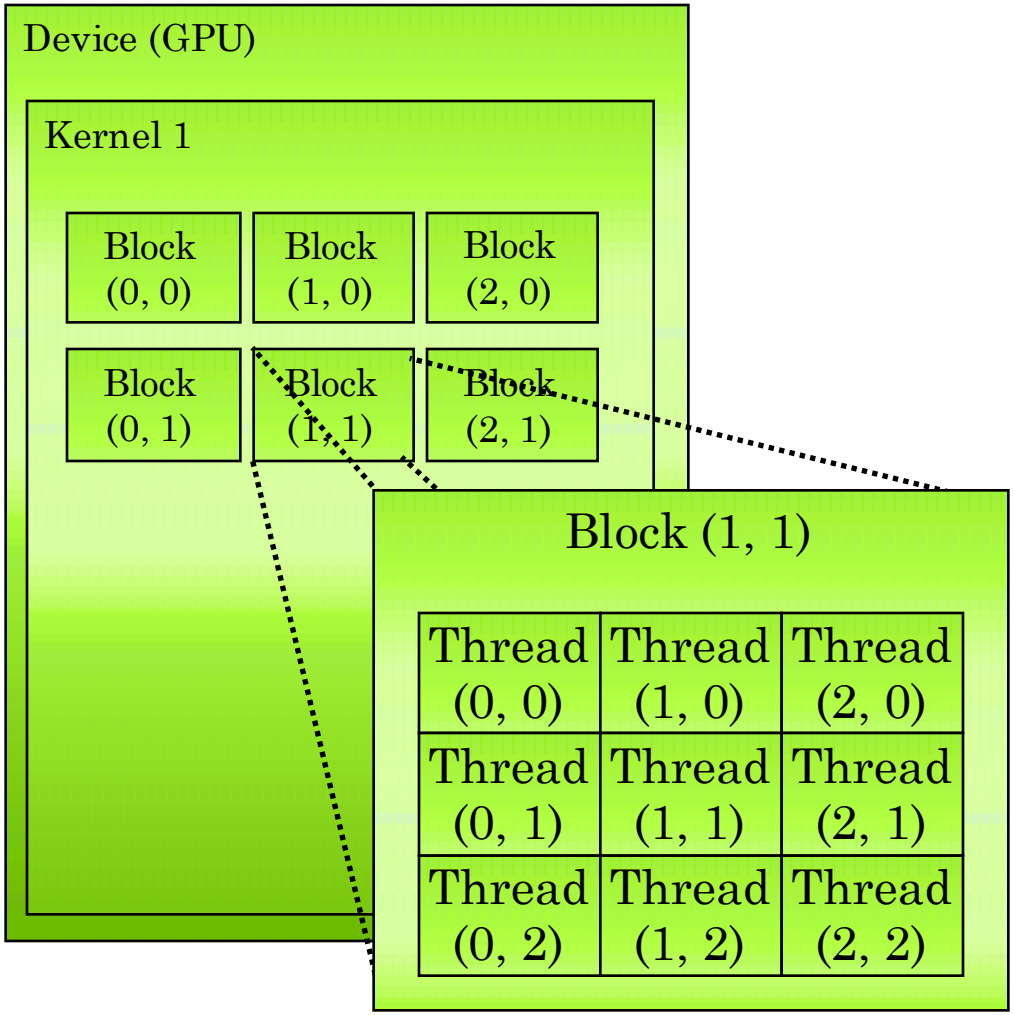


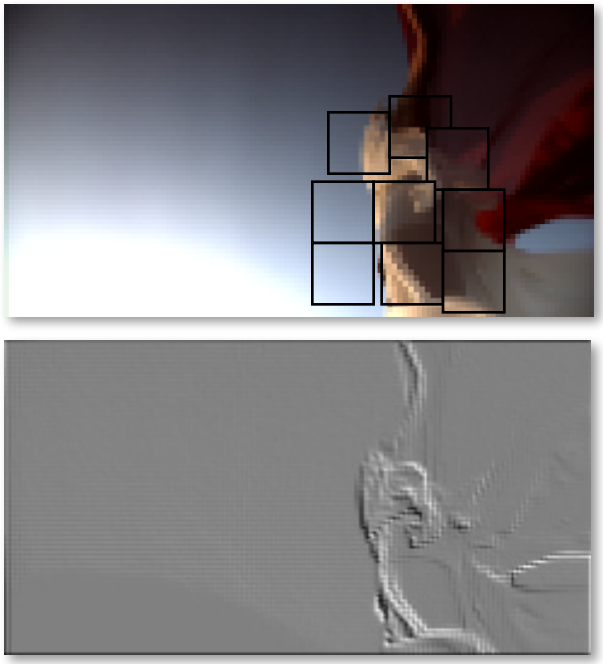






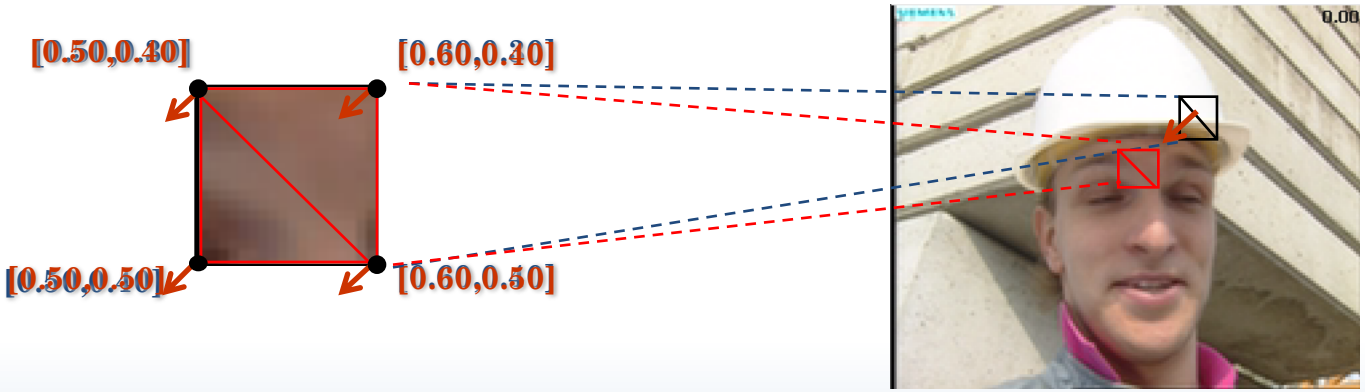
Motion Compensation in CUDA





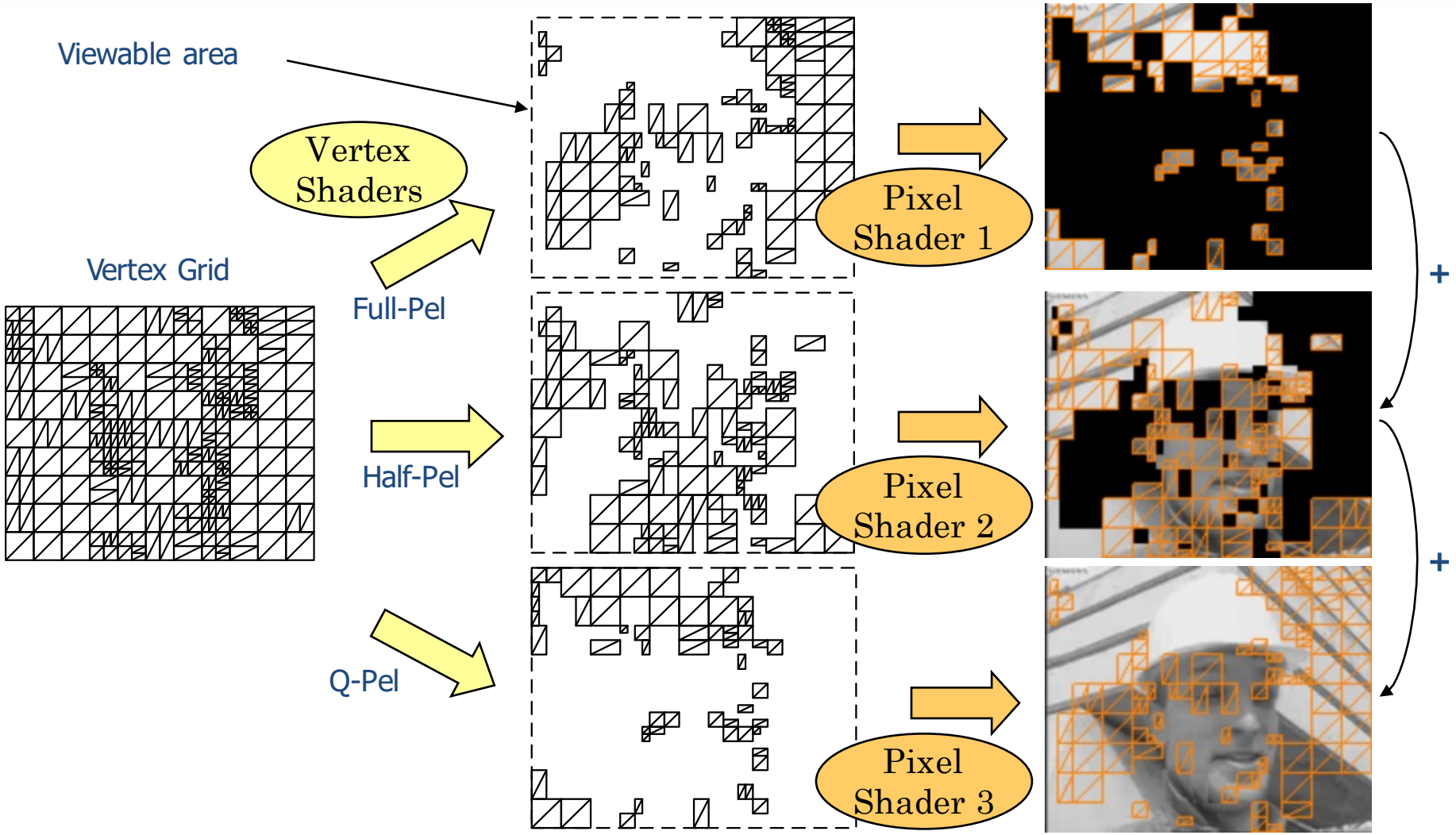
Motion Compensation in Direct3D

- Put video picture in textures
- Use vertices to represent a macroblock
- Let texture coordinate point to the texture
- Full-pel motion compensation
  - manipulate texture coordinates
- Multiple pixel shaders fill macroblocks and interpolate



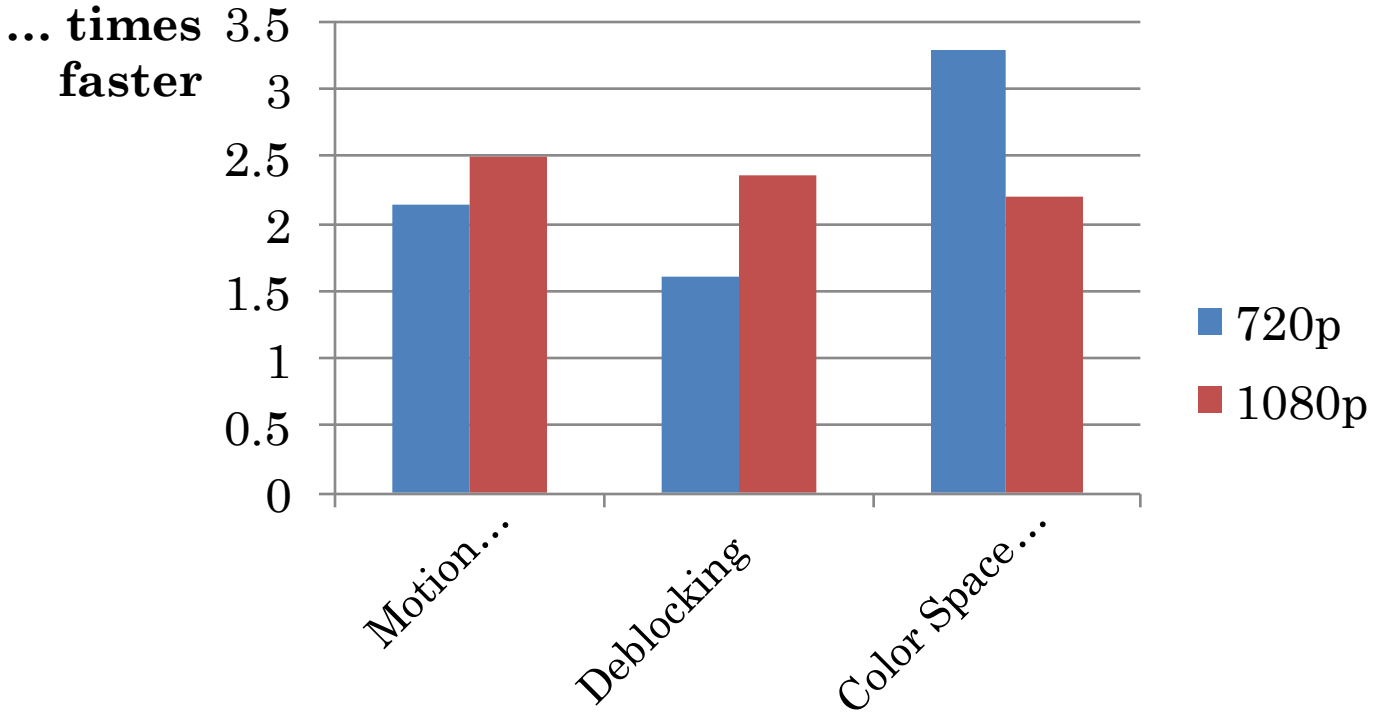
Reference texture for rasterization process

Interpolation Strategies for Sub-pixel MC



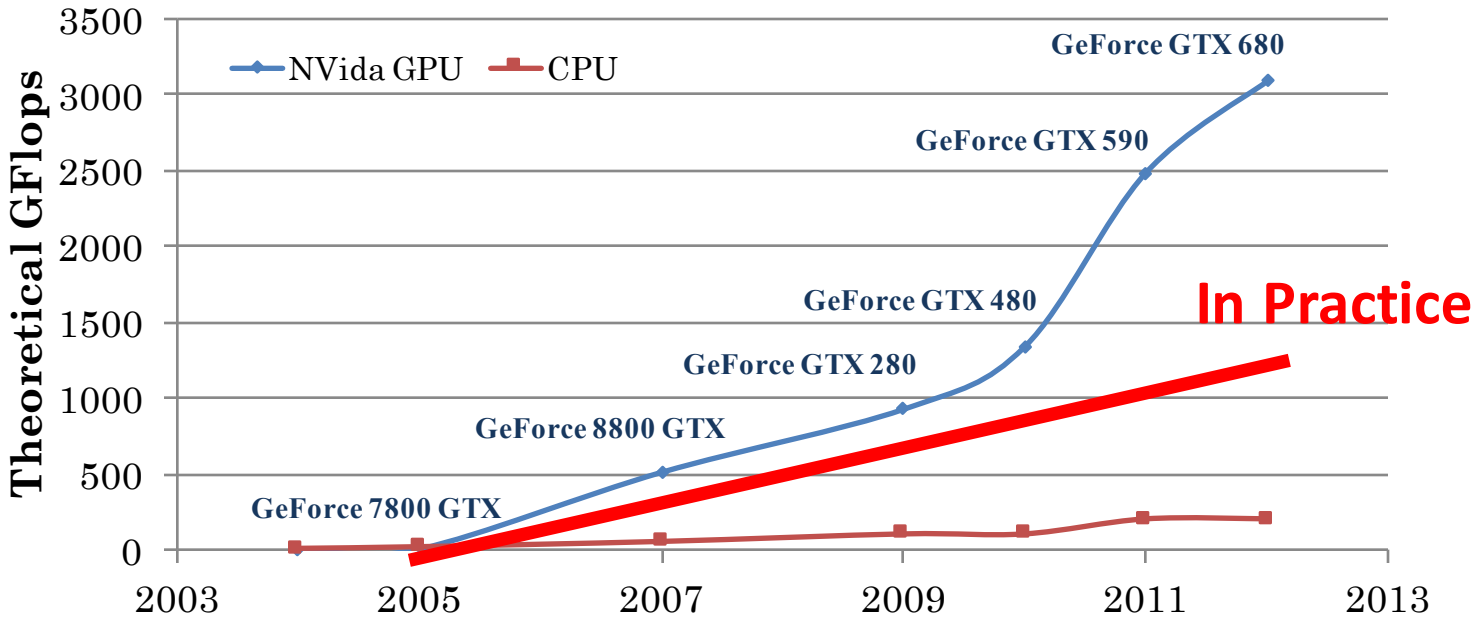
Experimental Results

- GPU algorithm scores faster than CPU algorithm
- CPU is offloaded, free for other tasks



GPU Performance Gap

- High performance at low cost
- Peak performance is difficult to achieve



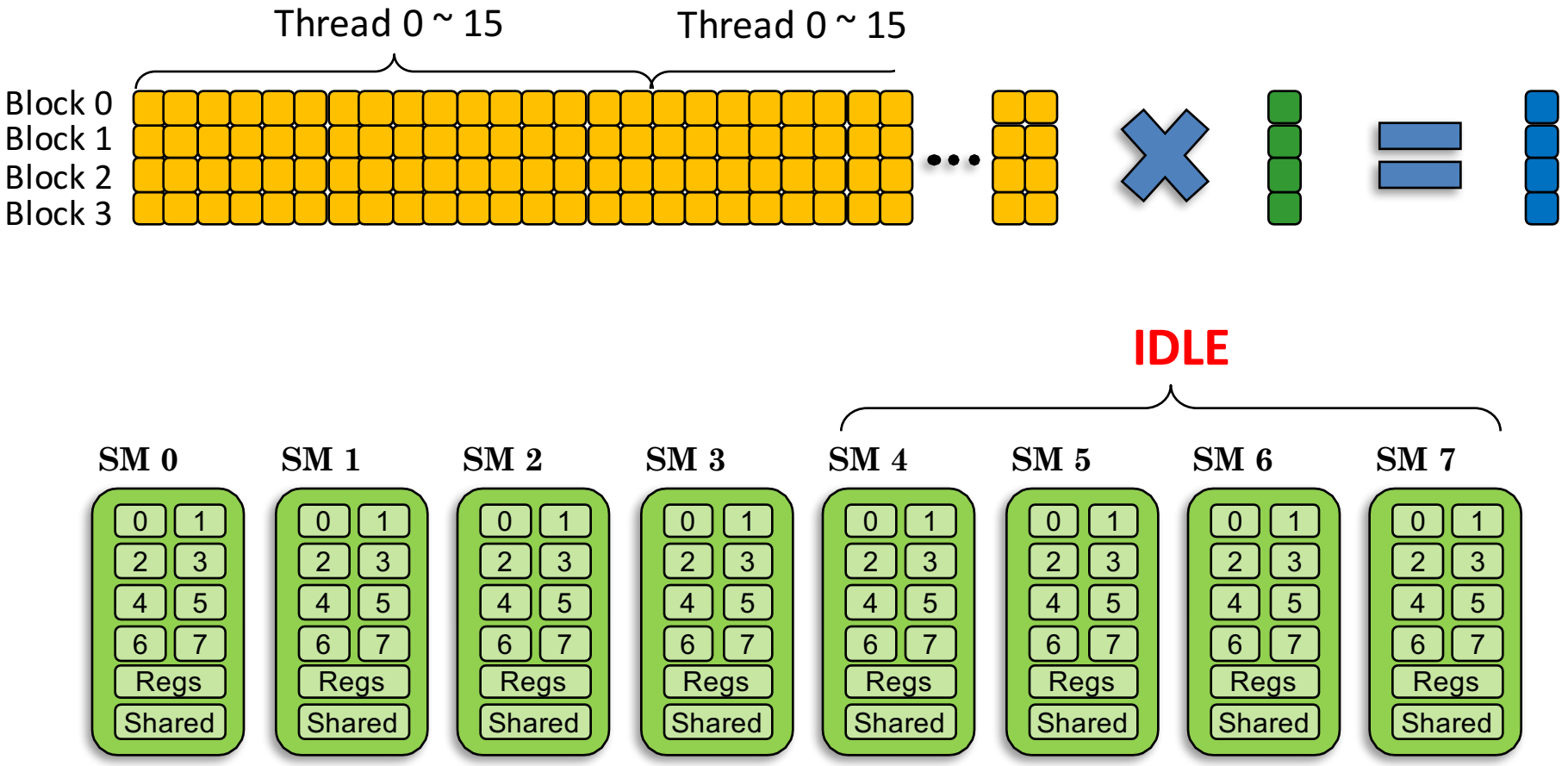
From: "Adaptive Input-aware Compilation for Graphics Engines"

Conclusions

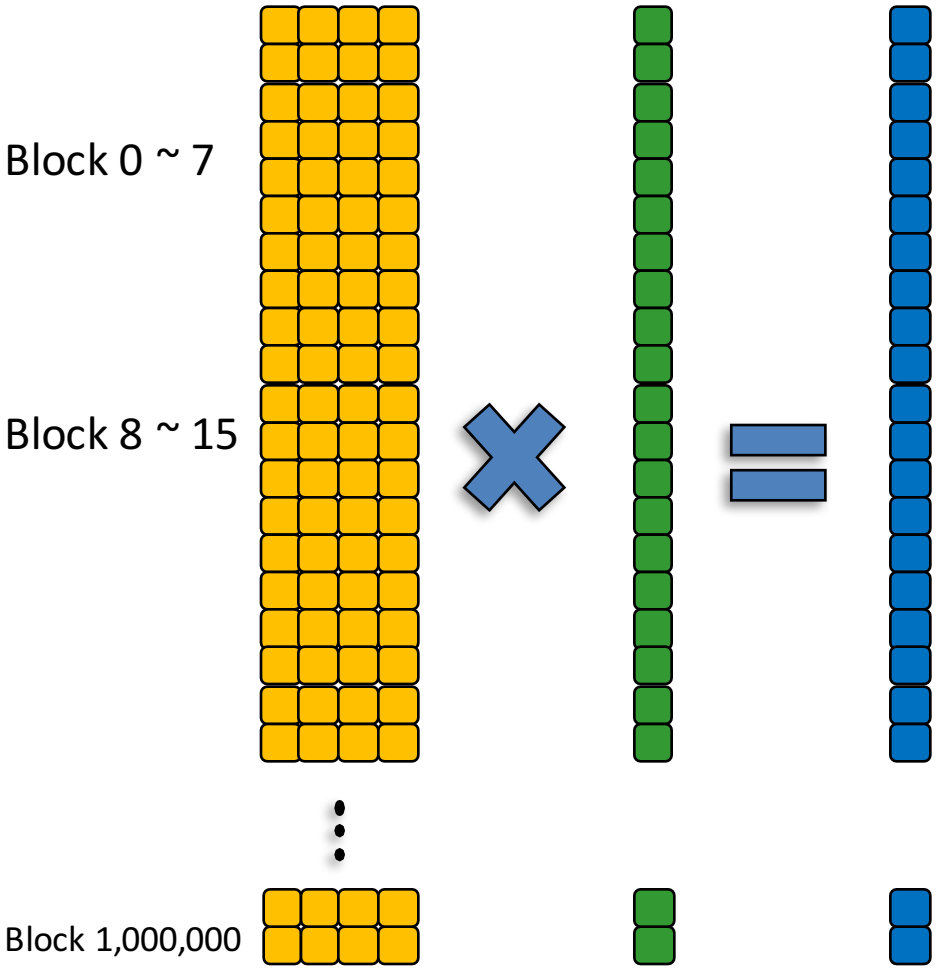
- GPU is an attractive platform for general-purpose computation
  - flexible, powerful, inexpensive
- General-purpose APIs
  - approach the GPU as a super-threaded co-processor
- GPGPU requires lots of parallel jobs
  - e.g. hundreds to thousands
- GPGPU allow faster execution while offloading the CPU
  - e.g. decoding of H.264/AVC bitstreams
- GPGPU techniques are suited for future architectures

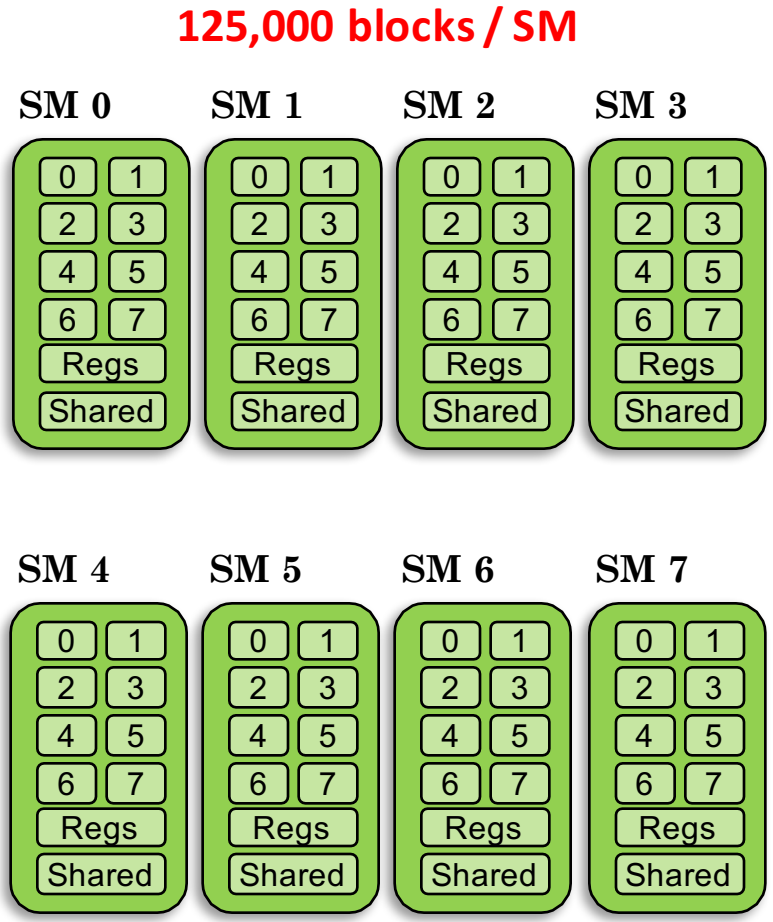
Thanks

Transposed Matrix Vector Multiplication (4 x 1M)

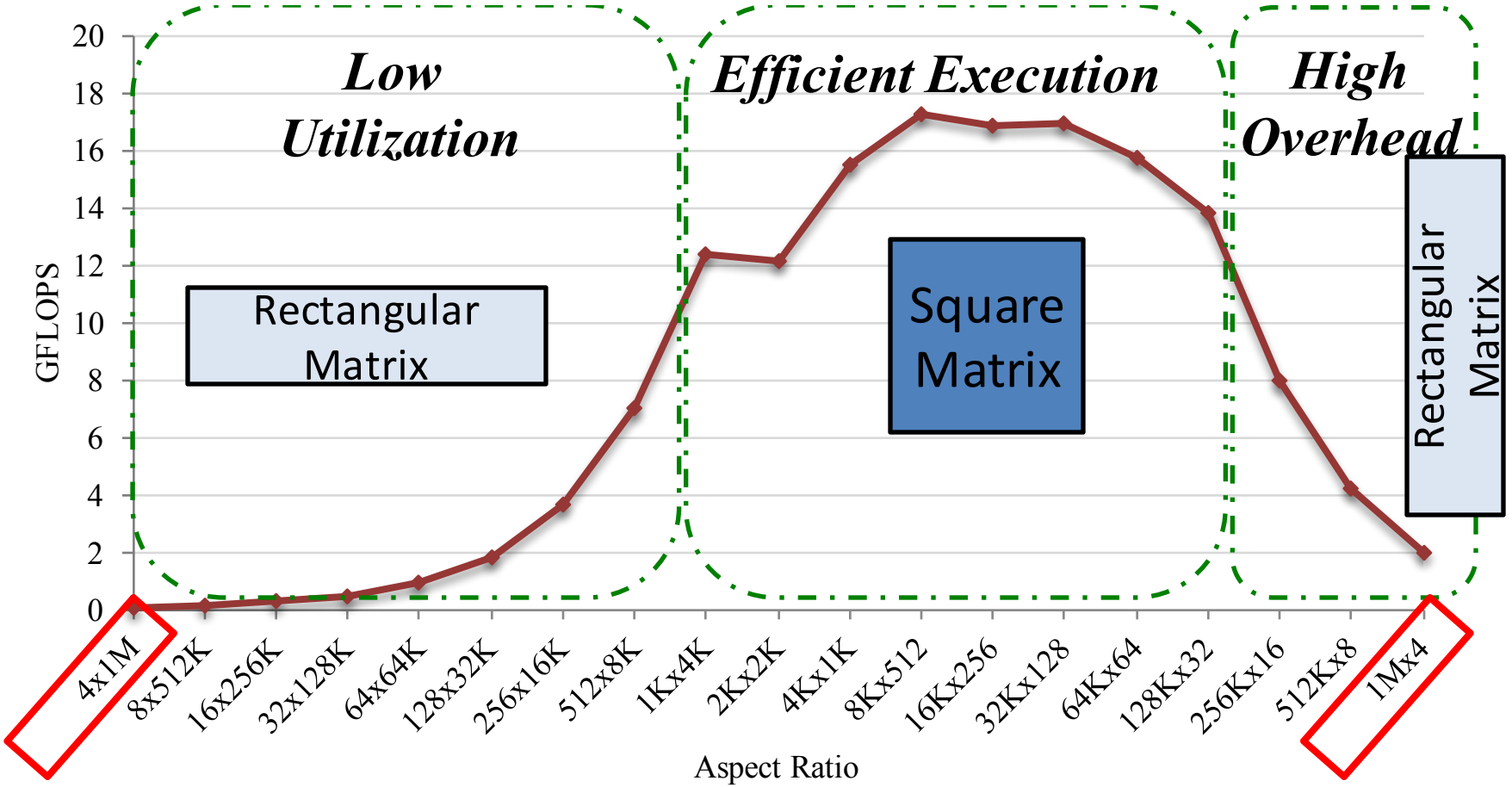


Transposed Matrix Vector Multiplication (1M x 4)

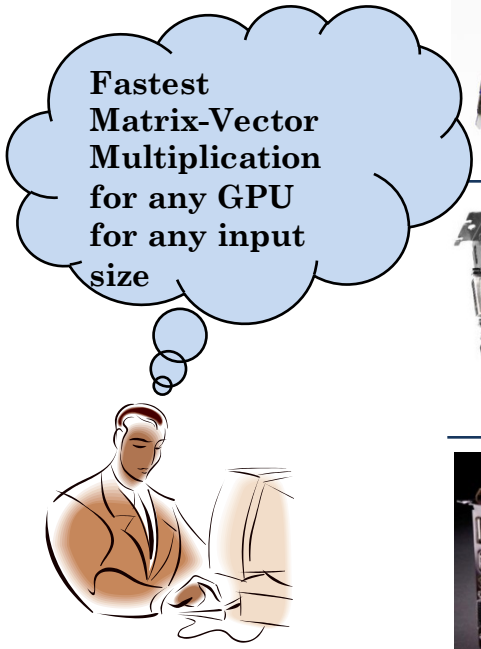




TMV Performance on Various Input



GPU Programming Challenge - Portability



GPU Architectures

2008 Cores : 240

2011 Cores : 512

2012 Cores : 1536

Input Matrix Size	Source Code
4 x 1M	GTX285_MV_4_1M.cu
128 x 32K	GTX285_MV_128_32K.cu
32K x 128	GTX285_MV_32K_128.cu
1M x 4	GTX285_MV_1M_4.cu
4 x 1M	GTX580_MV_4_1M.cu
128 x 32K	GTX580_MV_128_32K.cu
32K x 128	GTX580_MV_32K_128.cu
1M x 4	GTX580_MV_1M_4.cu
4 x 1M	GTX680_MV_4_1M.cu
128 x 32K	GTX680_MV_128_32K.cu
32K x 128	GTX680_MV_32K_128.cu
1M x 4	GTX680_MV_1M_4.cu
⋮	⋮