GPU Architecture: Overview

P J Narayanan
Centre for Visual Information Technology
IIIT, Hyderabad

IIIT Hyderabad

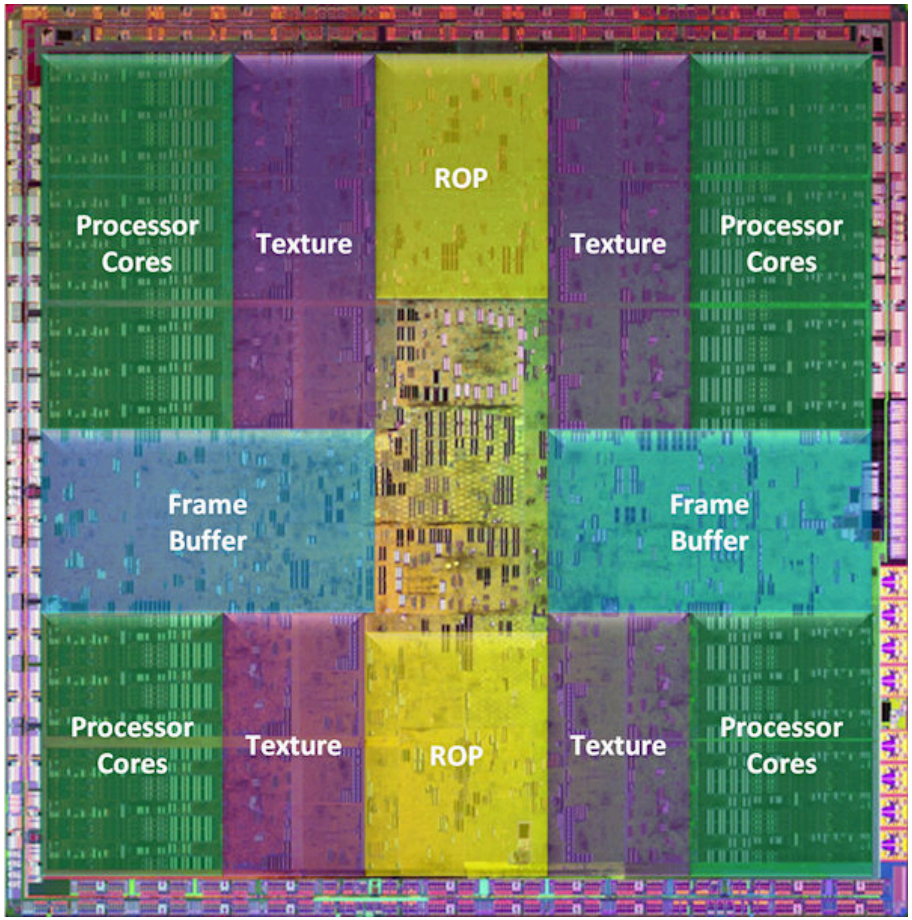
PPoPP Tutorial on
GPU Architecture, Programming and Performance Models

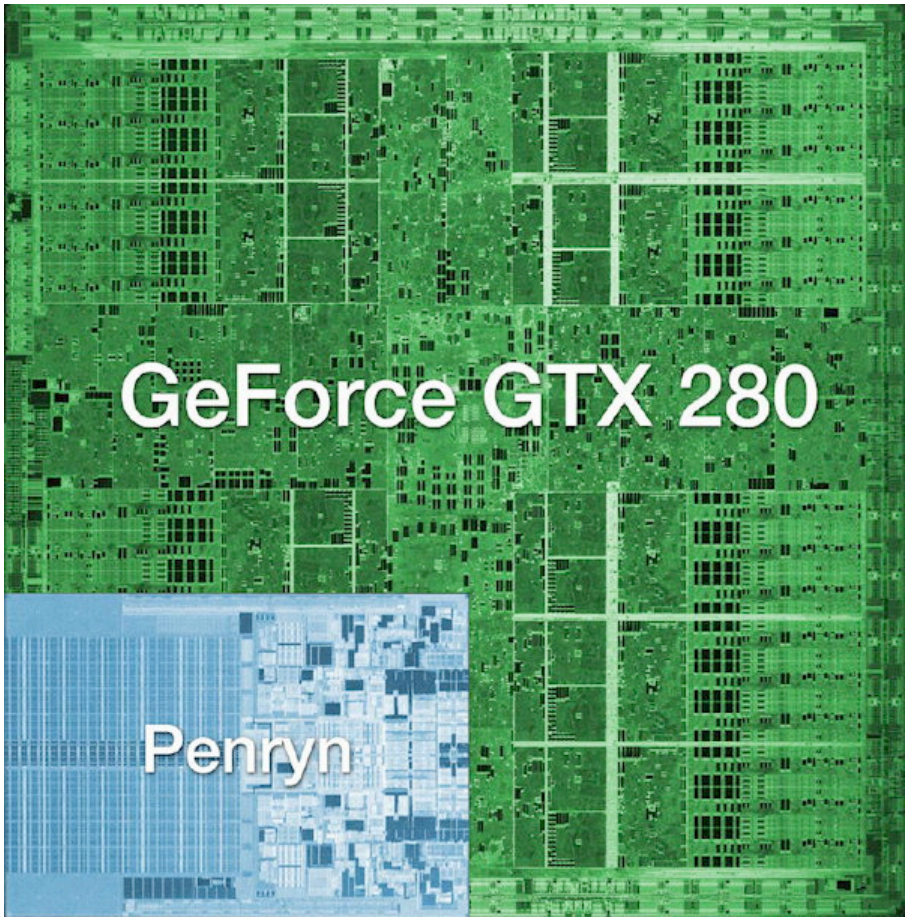
GPU: Evolution

- Graphics : a few hundred triangles/vertices map to a few hundred thousand pixels
- Process pixels in parallel. Do the same thing on a large number of different items.
- Data parallel model: parallelism provided by the data
  - Thousands to millions of data elements
  - Same program/instruction on all of them
- Hardware: 8-16 cores to process vertices and 64-128 to process pixels by 2005
  - Less versatile than CPU cores
  - SIMD mode of computations. Less hardware for instruction issue
  - No caching, branch prediction, out-of-order execution, etc.
  - Can pack more cores in same silicon die area

GPU & CPU

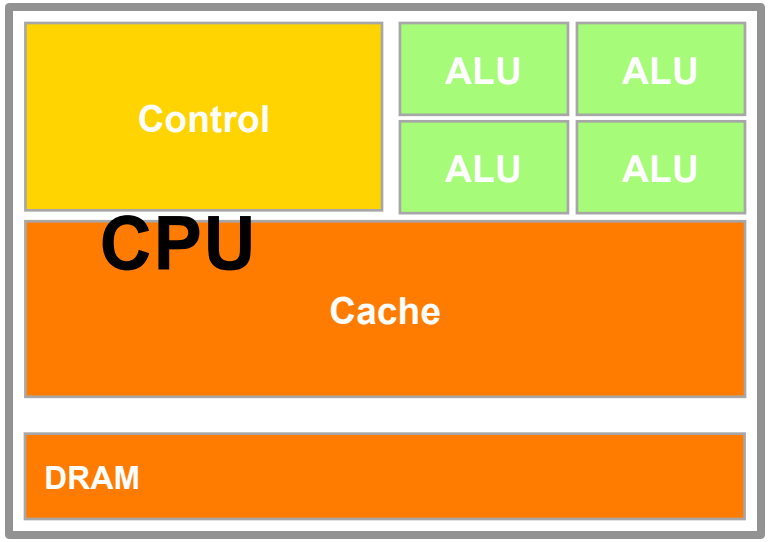
Nvidia GTX280

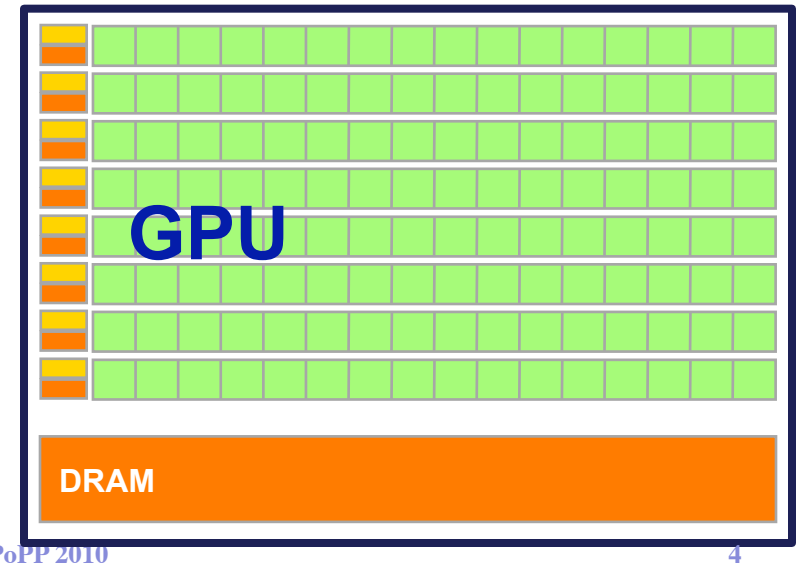




CPU vs GPU

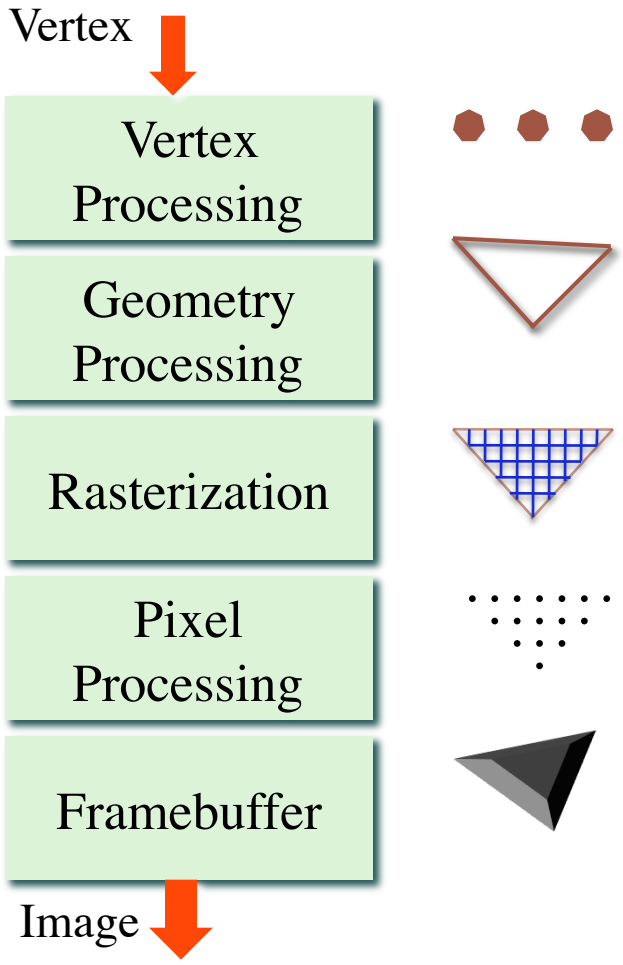
- CPU Architecture features:
  - Few, complex cores
  - Perform irregular operations well
    - Run an OS, control multiple IO, pointer manipulation, etc.
- GPU Architecture features:
  - Hundreds of simple cores, operating on a common memory (like the PRAM model)
  - High compute power but high memory latency (1:500)
  - No caching, prefetching, etc
  - High *arithmetic intensity* needed for good performance
    - Graphics rendering, image/signal processing, matrix manipulation, FFT, etc.





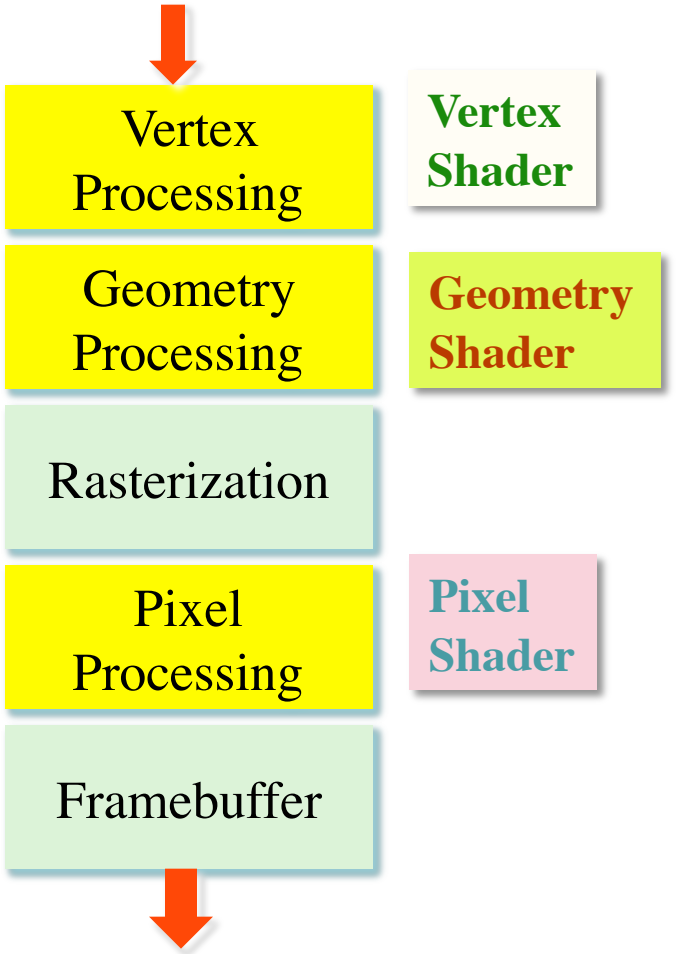
What do GPUs do?

- GPU implements the graphics pipeline consisting of:
  - Vertex transformations
    - Compute camera coords, lighting
  - Geometry processing
    - Primitive-wide properties
  - Rasterizing polygons to pixels
    - Find pixels falling on each polygon
  - Processing the pixels
    - Texture lookup, shading, *Z*-values
  - Writing to the framebuffer
    - Colour, *Z*-value
- Computationally intensive



Programmable GPUs

- Parts of the GPU pipeline were made programmable for innovative shading effects
- ***Vertex***, ***pixel***, & later ***geometry*** stages of processing could run user's *shaders*.
- Pixel shaders perform *Data-parallel* computations on a parallel hardware
  - 64-128 single precision floating point processors
  - Fast texture access
- GPGPU: High performance computing on the GPU using shaders. Efficient for vectors, matrix, FFT, etc.

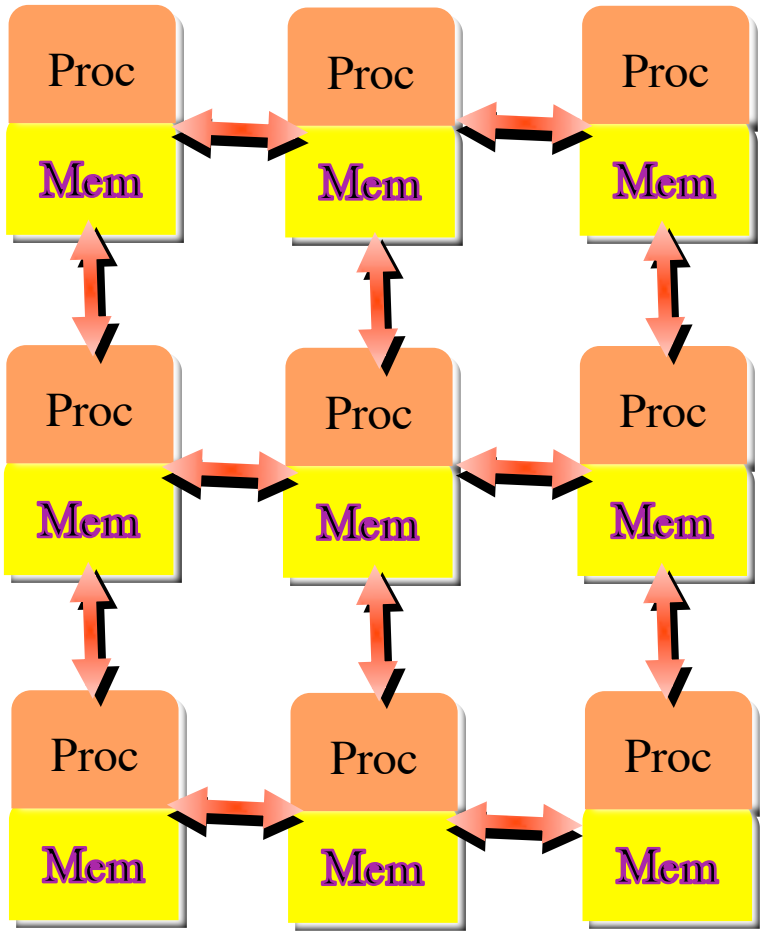


New Generation GPUs

- The DX10/SM4.0 model required a uniform shader model
- Translated into common, unified, hardware cores to perform vertex, geometry, and pixel operations.
- Brought the GPUs closer to a general parallel processor
- A number of cores that can be reconfigured dynamically
  - More cores: 128 → 240 → 320
  - Each transforms data in a common memory for use by others

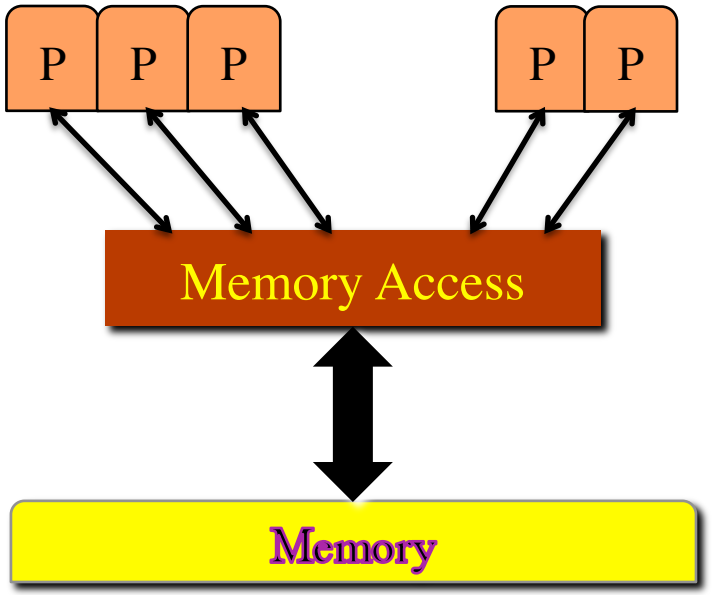
Old Array Processors

- Processor and Memory tightly attached
- A network to interconnect
  - Mesh, star, hypercube
- Local data: Memory read/ write
  Remote data: network access
- Data reorganization is expensive to perform
- Data-Parallel model works
- Thinking Machines CM-1, CM-2. MasPar MP-1, etc



Current GPU Architecture

- Processors have no local memory
- Bus-based connection to the common, large, memory
- Uniform access to all memory for a PE
  - Slower than computation by a factor of 500
- Resembles the PRAM model!
- No caches. But, instantaneous locality of reference improves performance
  - Simultaneous memory accesses combined to a single transaction
- Memory access pattern determines performance seriously
- Compute power: Upto 3 TFLOPs on a $400 add on card



What is the GPU Good at?

- The GPU is good at

  data-parallel processing

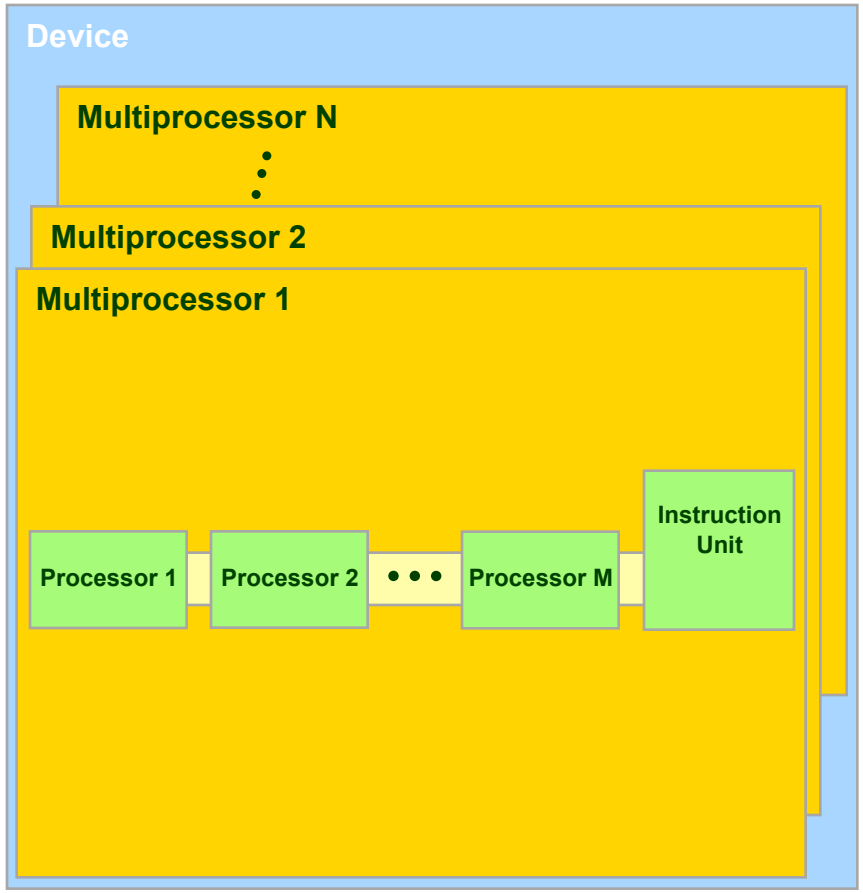
  - The same computation executed on many data elements in parallel – low control flow overhead

  with high SP floating point arithmetic intensity

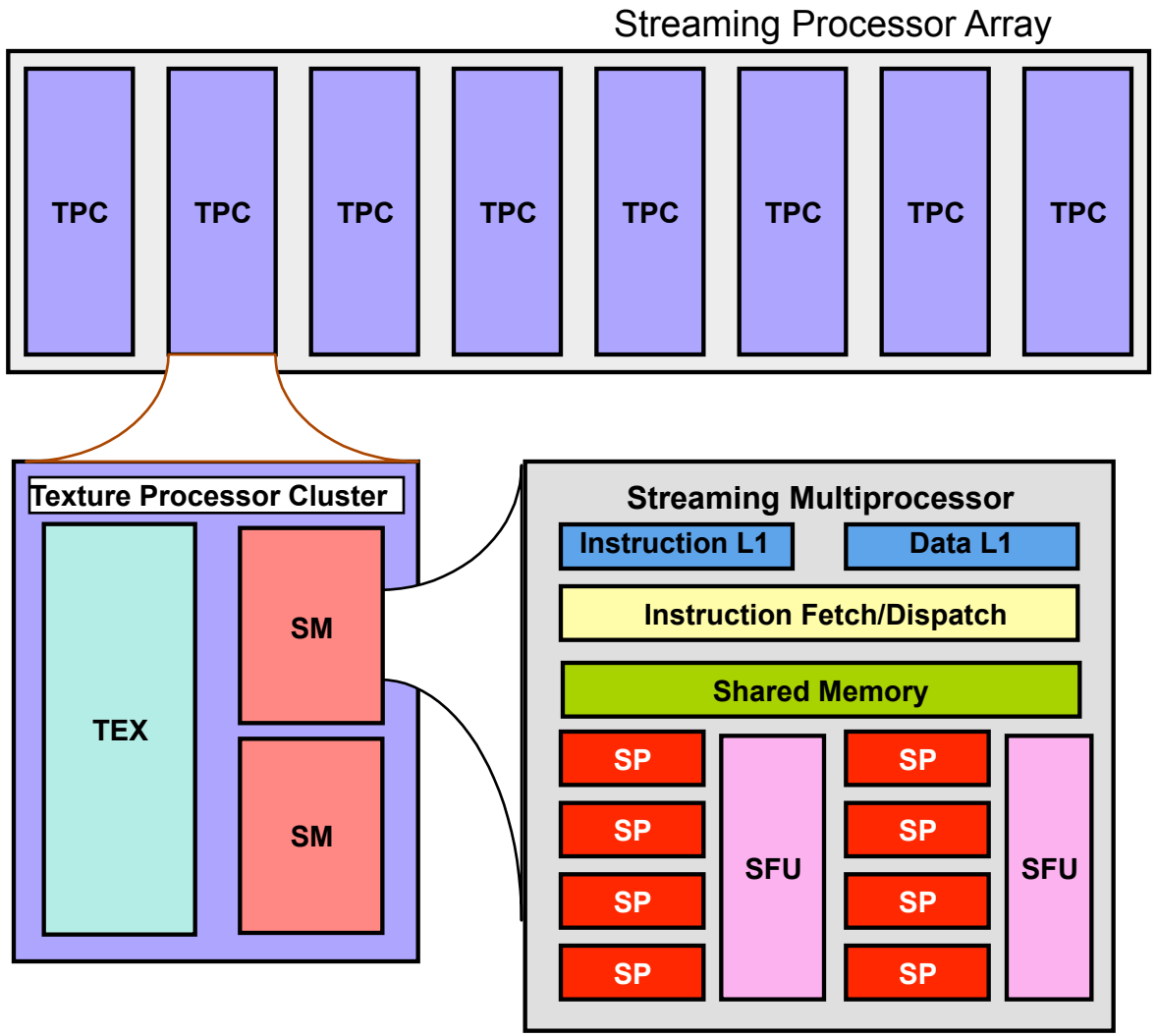
  - Many calculations per memory access
  - Currently also need high floating point to integer ratio
- High floating-point arithmetic intensity and many data elements can hide memory access latency without big data cache

SIMD Multiprocessors

- The device is a set of 16 or 30 multiprocessors
- Each multiprocessor is a set of 32-bit processors with a Single Instruction Multiple Data architecture – shared instruction unit
- At each clock cycle, a multiprocessor executes the same instruction on a group of threads called a warp
- The number of threads in a warp is the warp size

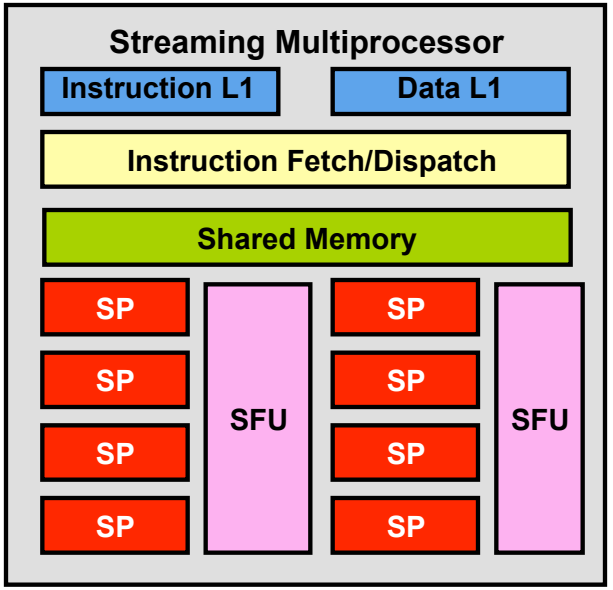


HW Overview


Streaming Processor Array
TPC
TPC
TPC
TPC
TPC
TPC
TPC
TPC
Texture Processor Cluster
TEX
SM
SM
Streaming Multiprocessor
Instruction L1
Data L1
Instruction Fetch/Dispatch
Shared Memory
SP
SP
SP
SP
SFU
SP
SP
SP
SP
SFU

Streaming Multi-Processor

- Streaming Multiprocessor
  - 8 Streaming Processors (SP)
    - 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - 1 to 512 threads active
  - Shared instruction fetch per 32 threads
  - Cover latency of texture/memory loads
- 30+ GFLOPS
- 16K registers
  - Partitioned among active threads
- 16 KB shared memory
  - Partitioned among logical blocks

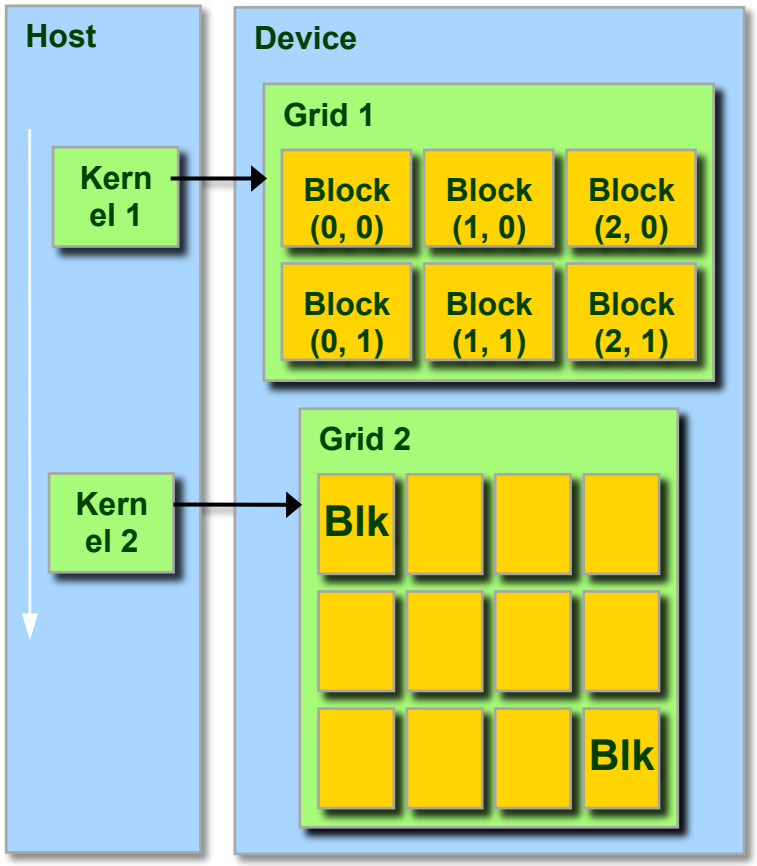


Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

Thread Batching: Grids and Blocks

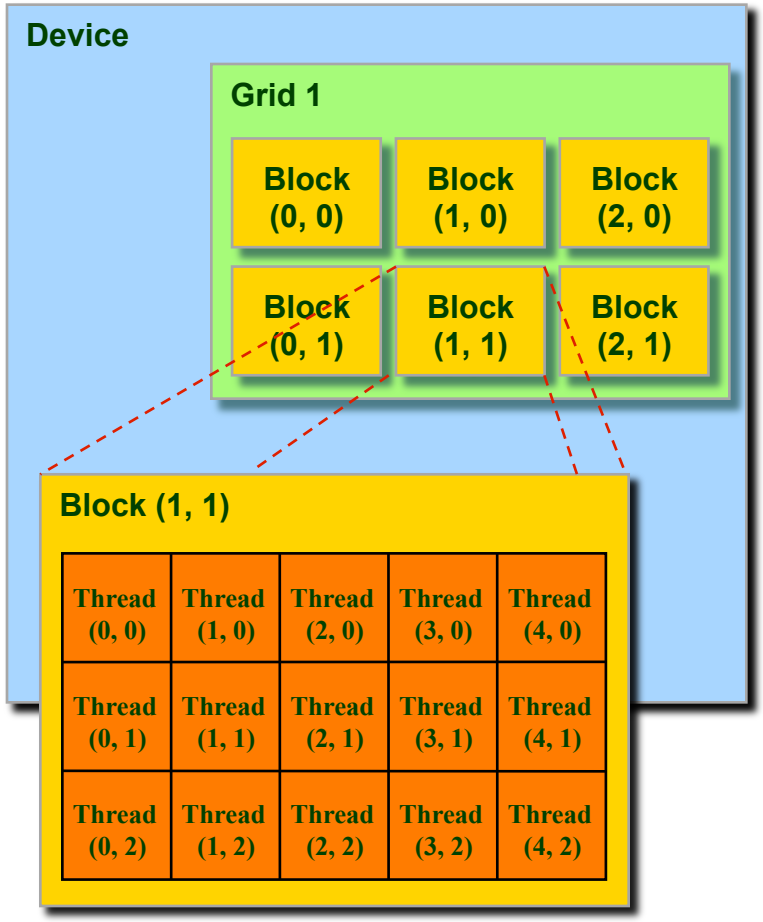
- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate



Courtesy: NDVIA

Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Courtesy: NDVIA

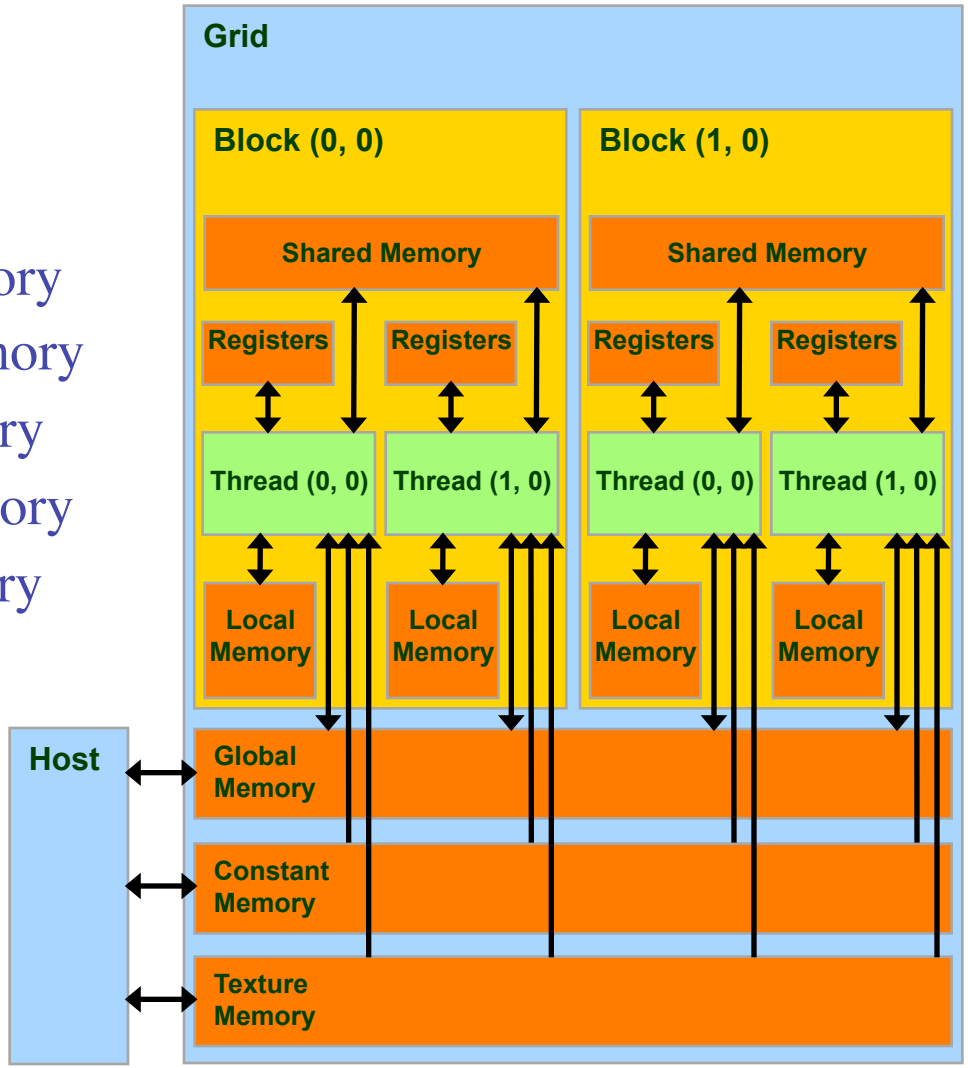
Threads, Warps, Blocks

- 32 threads in a Warp or a scheduling group
  - Only <32 when there are fewer than 32 total threads
- There are (up to) 16 Warps in a Block
- Each Block (and thus, each Warp) executes on a single SM
- G80 has 16 SMs, G280 has 30 SMs
- At least 16 Blocks required to “fill” the device
- More is better
  - If resources (registers, thread space, shared memory) allow, more than 1 Block can occupy each SM

Memory Spaces

- Each thread can:
  - Read/write per-thread registers
  - Read/write per-thread local memory
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

The host can read/write global, constant, and texture memory



Memory Access Times

- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW - single cycle
- Local Memory – DRAM, no cache - *slow*
- Global Memory – DRAM, no cache - *slow* (400-500 cycles)
- Constant Memory – DRAM, cached, 1…10s… 100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s… 100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached

Thread Scheduling/Execution

Each Thread Blocks consists of 32-thread *warp*s currently

Warps are scheduling units in SM. A warp is schedule at one time

Multiple warps time share the SM processors

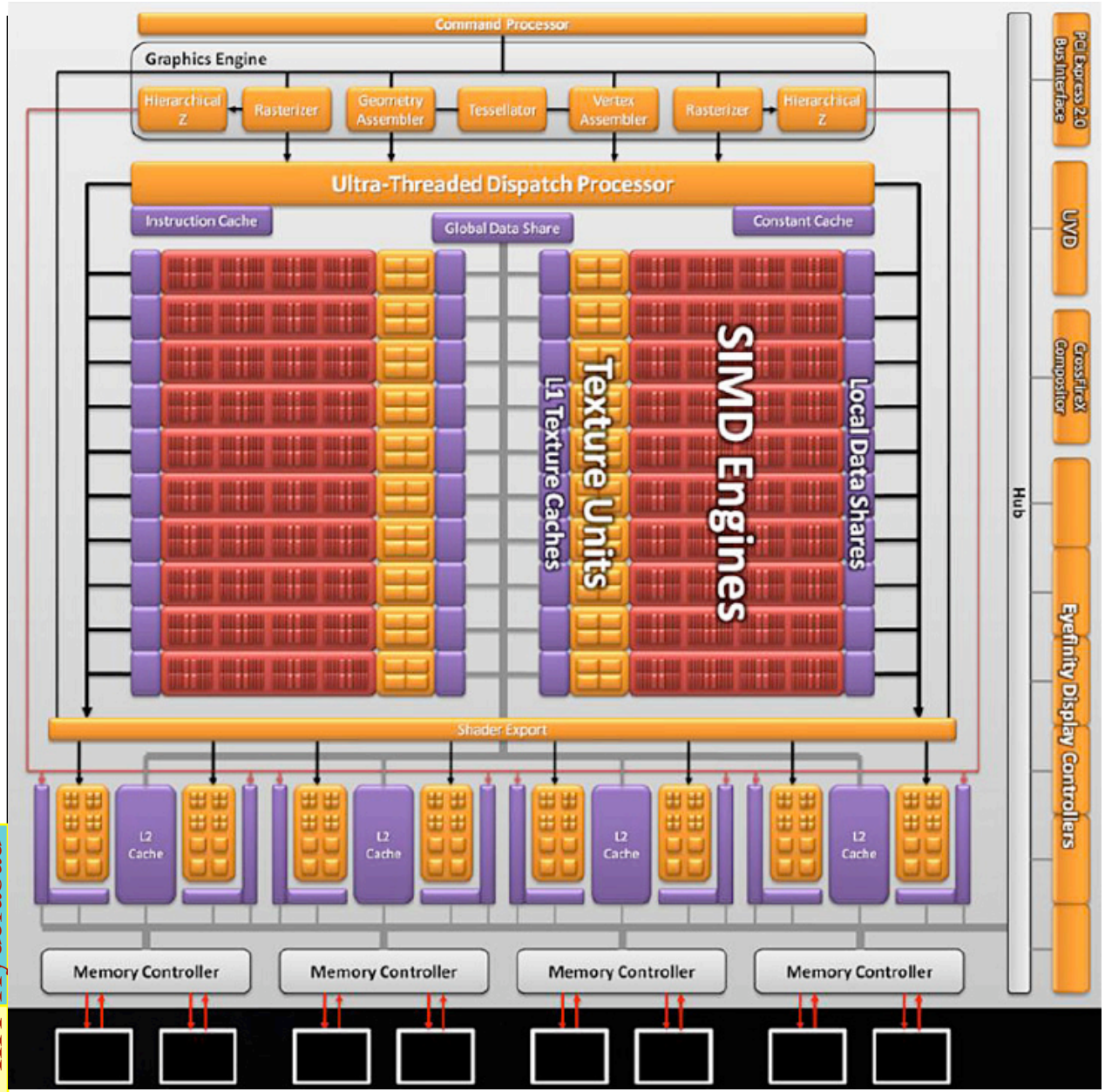
Multiple blocks can also share an SM, if resources permi
Available resources are vertically shared between bloc
that time-share an SM

If more blocks are needed, they use the hardware sequentially.

Processors, Memory

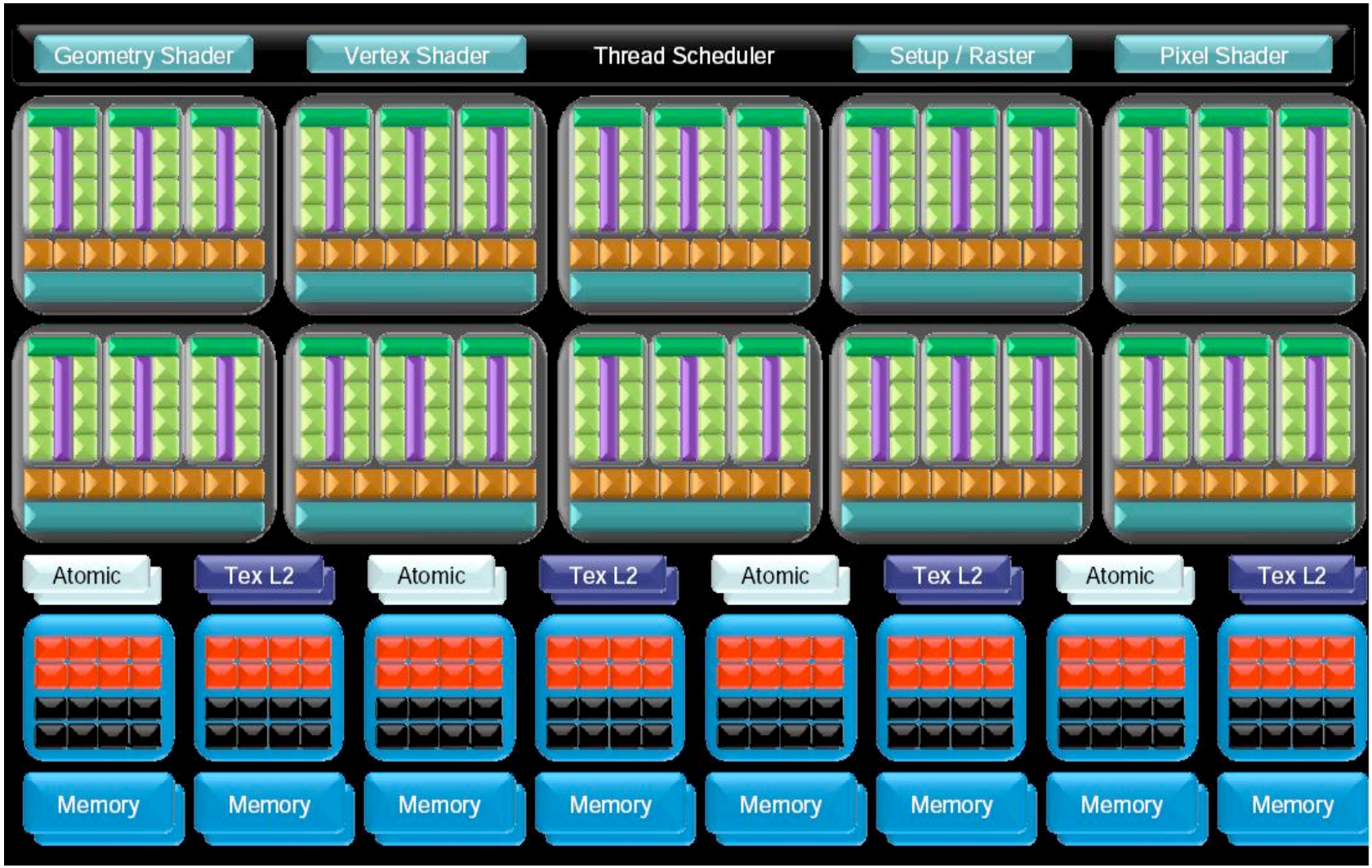
- Nvidia 280GTX: 240 Streaming Processors, grouped into 30 Streaming Multiprocessors
  - One instruction sequencer per SM
  - 16KB of on-chip shared memory per SM
  - 16K 32-bit registers per SM
  - Single clock access of registers, shared memory
- 1 GB of common, off-chip global memory
  - 130 GB/s of theoretical peak memory bandwith
  - High memory access latency: 300-500 cycles
  - 128 byte, 64 byte, or 32 byte memory transactions
- 10 special texture access units to the same global memory. 30 SMs grouped into 10 Texture processor clusters
- 1.3 GHz clock, 933 GFLOPs peak
- Integer and single-precision float operations in one clock cycle. Slower double-precision support

AMD 5870 Architecture



- 20 SIMD engines with 16 stream cores each
  - Each SC with 5 PEs (1600 Pes in total)
  - Each with IEEE754 and integer support
  - Each with local data share memory
    - 32 kb shared low latency memory
    - 32 banks with hardware conflict management
    - 32 integer atomic units 80 Read Address Probes
  - 4 addresses per SIMD engine
  - 4 filter or convert logic per SIMD Global Memory access
- 153 GB/sec GDDR5 memory interface

Nvidia 280GTX: Architecture



Performance Considerations

- Thread divergence
  - SIMD width is 32 threads. They should execute the same very instruction
  - Serialization otherwise
- Memory access coherence
  - A half-warp of 16 threads should read from a local block (128, 64, or 32 bytes) for speed
  - Random memory access very expensive
- Occupancy or degree of parallelism
  - Optimum use of registers and shared memory for maximum exploitation of parallelism
  - Memory latency hidden best with high parallelism
- Atomic operations
  - Global and shared memory support slow atomic operations

Tools and APIs

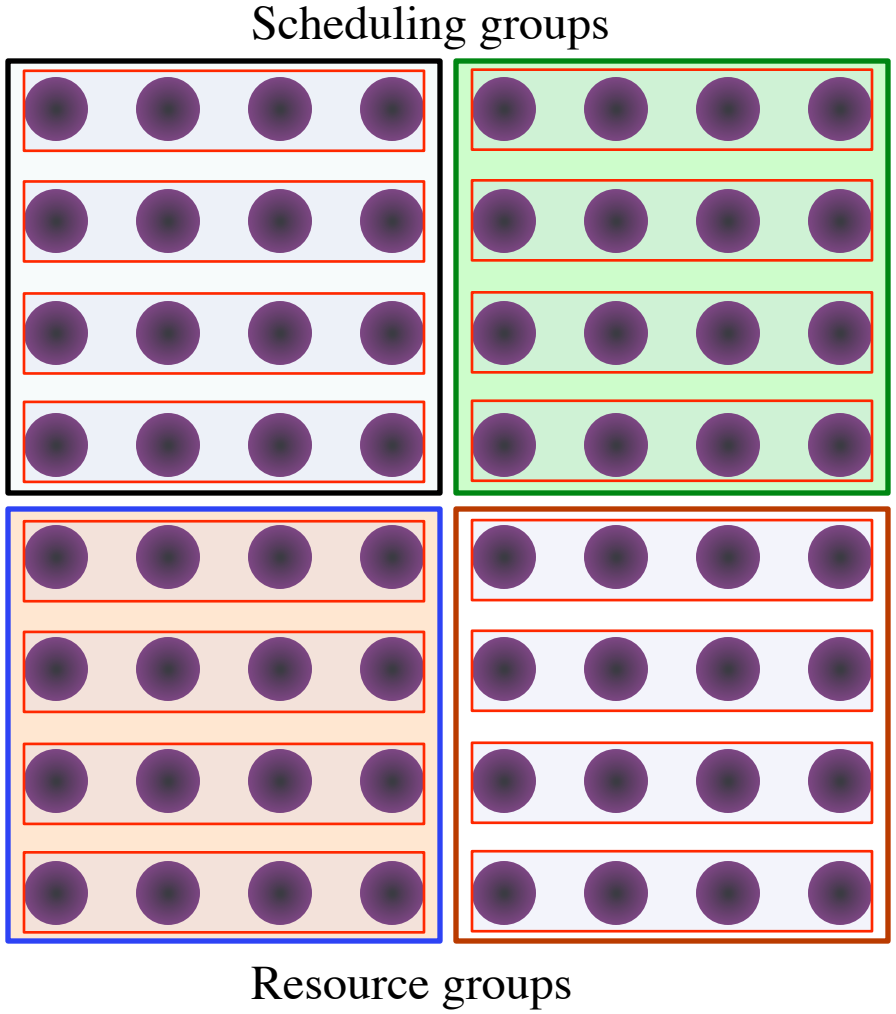
- OpenGL/Direct3D for older, GPGPU exposure
  - Shaders operating on polygons, textures, and framebuffer
- CUDA: an alternate interface from Nvidia
  - Kernel operating on grids using threads
  - Extensions of the C language
- DirectX Compute Shader: Microsoft's version
- OpenCL: A promising open compute standard
  - Apple, Nvidia, AMD, Intel, TI, etc.
  - Support for task parallel, data parallel, pipeline-parallel, etc.
  - Exploit the strengths of all available computing resources

Massively Multithreaded Model

- Hiding memory latency: Overlap computation & memory access
  - Keep multiple threads in flight simultaneously on each core
  - Low-overhead switching. Another thread computes when one is stalled for memory data
  - Alternate resources like registers, context to enable this
- A large number of threads in flight
  - Nvidia GPUs: up to 128 threads on each core on the GTX280
  - 30K time-shared threads on 240 cores
- Common instruction issue units for a number of cores
  - SIMD model at some level to optimize control hardware
  - Inefficient for if-the-else divergence
- Threads organized in multiple tiers

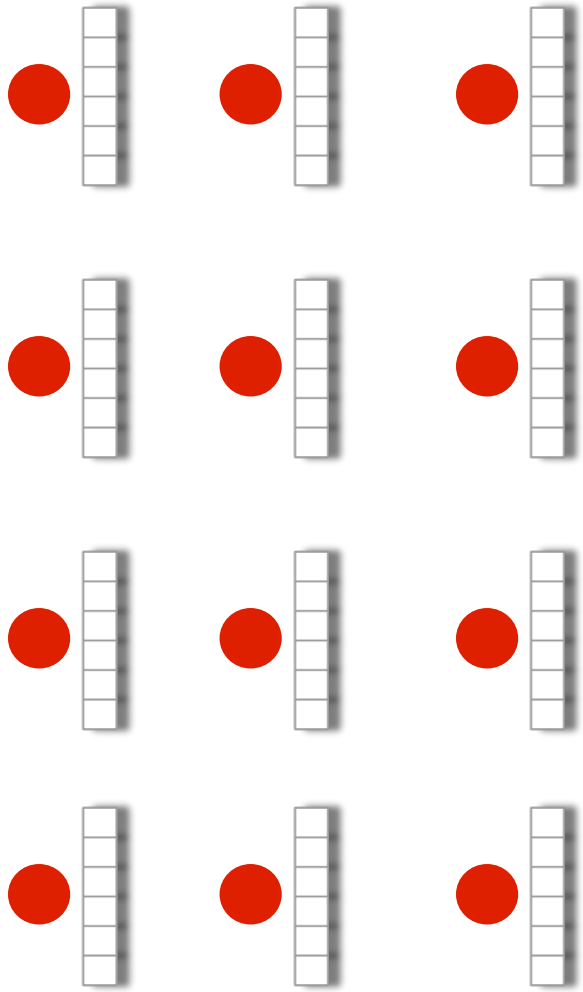
Multi-tier Thread Structure

- Data parallel model: A kernel on each data element
  - A kernel runs on a core
  - CUDA: an invocation of the kernel is called a *thread*
  - OpenCL: the same is called a *work item*
- Group data elements based on simultaneous scheduling
  - Execute truly in parallel, SIMD mode
  - Memory access, instruction divergence, etc., affect performance
  - CUDA: a *warp* of threads
- Group elements for resource usage
  - Share memory and other resources
  - May synchronize within group
  - CUDA: *Blocks* of threads
  - OpenCL: *Work groups*



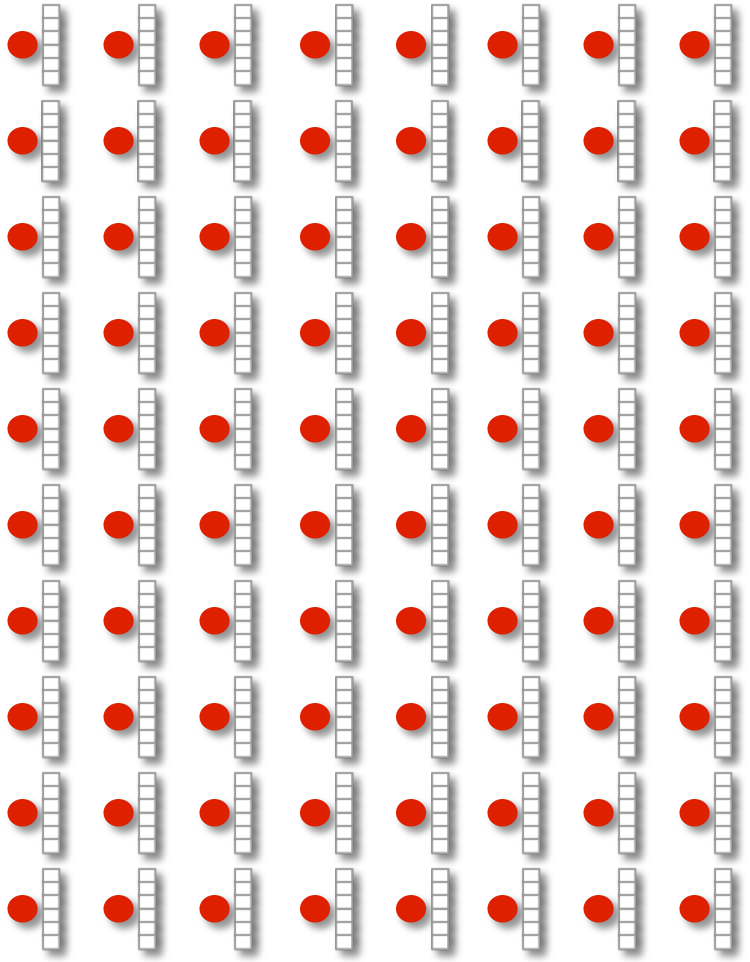
Data-Parallelism

- Data elements provide parallelism
  - Think of many data elements, each being processed simultaneously



Data-Parallelism

- Data elements provide parallelism
  - Think of many data elements, each being processed simultaneously
  - Thousands of threads to process thousands of data elements
- Not necessarily SIMD, most are SIMD or SPMD
  - Each kernel knows its location, identical otherwise
  - Work on different parts using the location

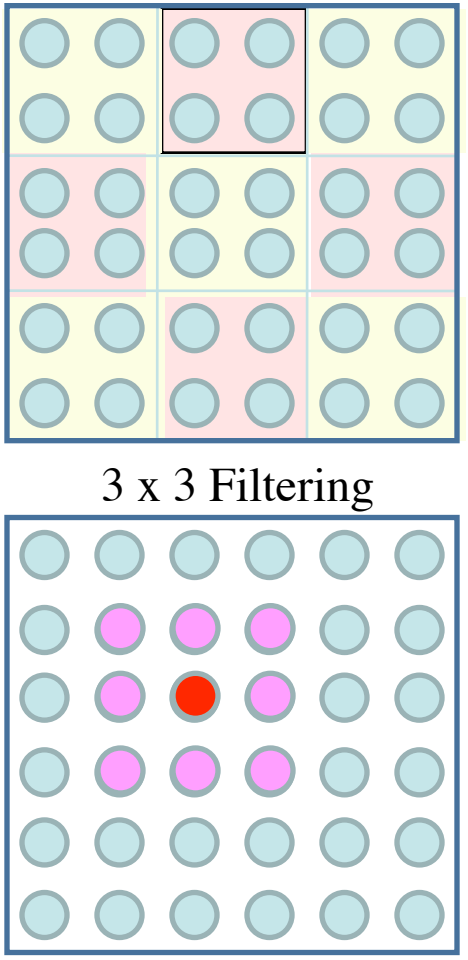


Thinking Data-Parallel

- Launch *N* data ***locations***, each of which gets a ***kernel*** of code
- Data follows a *domain* of computation.
- Each invocation of the kernel is aware of its location *loc* within the domain
  - Can access different data elements using the *loc*
  - May perform different computations also
- Variations of SIMD processing
  - Abstain from a compute step: if (**f(*loc*)**) then … else …
    - Divergence can result in serialization
  - Autonomous addressing for gather: a := b[**f(*loc*)**]
  - Autonomous addressing for scatter: a[**g(*loc*)**] := b
    - GPGPU model supports gather but not scatter
  - Operation autonomy: Beyond SIMD.
    - GPU hardware uses it for graphics, but not exposed to users

Image Processing

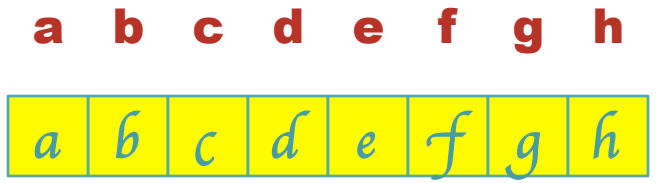
- A kernel for each location of the 2D domain of pixels
  - Embarrassingly parallel for simple operations
- Each work element does its own operations
  - Point operations, filtering, transformations, etc.
- Process own pixels, get neighboring pixels, etc
- Work groups can share data
  - Get own pixels and "apron" pixels that are accessed multiple times

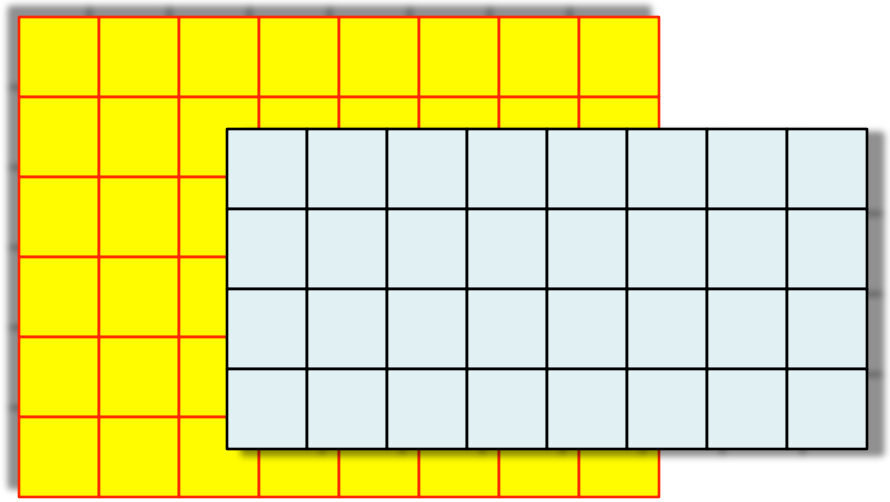


3 x 3 Filtering

Regular Domains

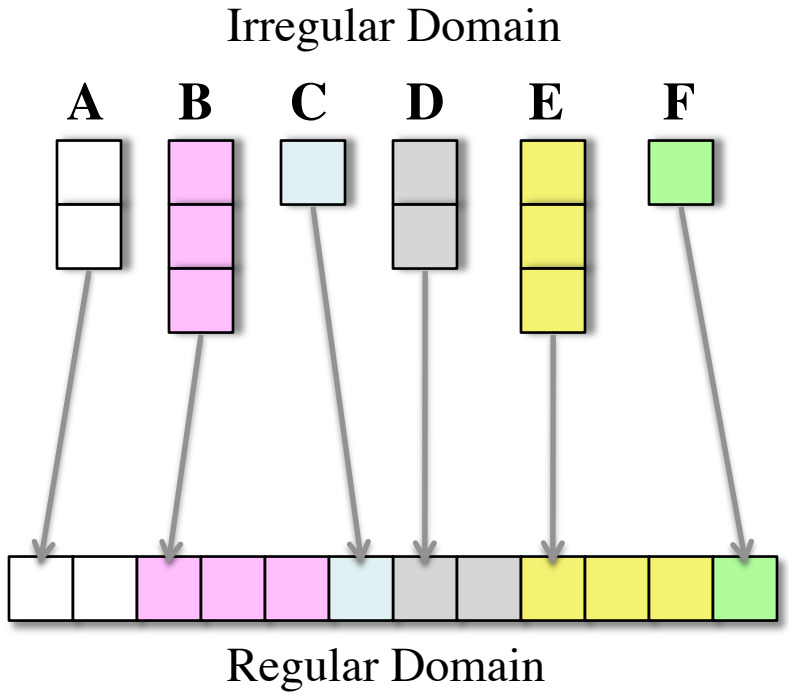
- Regular *1D*, *2D*, and *nD* domains map very well to data-parallelism
- Each work-item operates by itself or with a few neighbors
- Need not be of equal dimensions or length
- A mapping from *loc* to each domain should exist





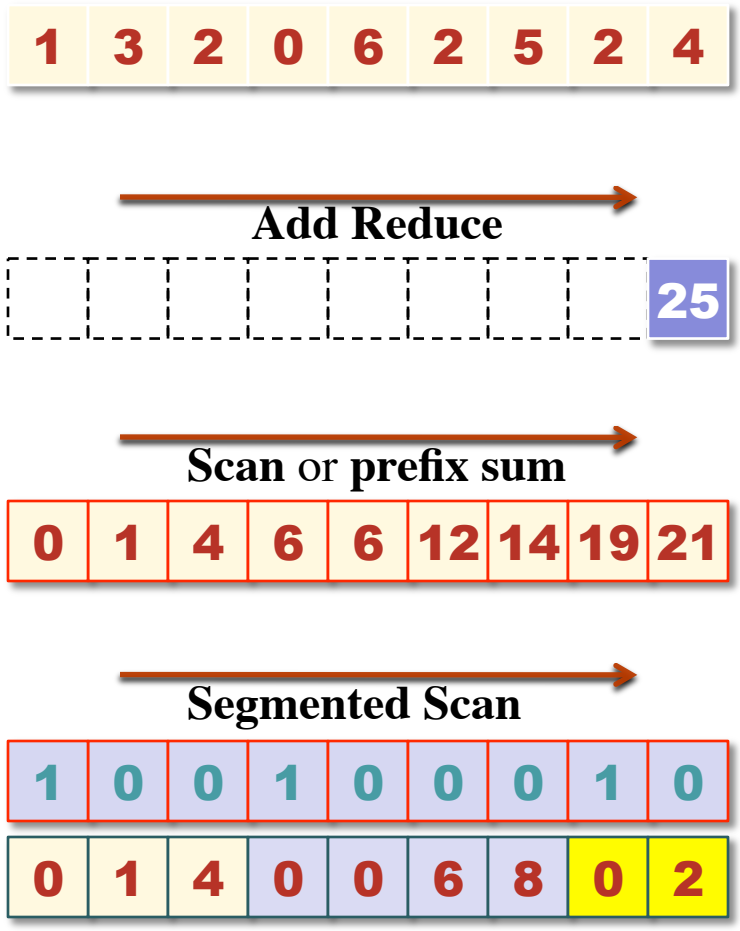
Irregular Domains

- A regular domain generates varying amounts of data
  - Convert to a regular domain
  - Process using the regular domain
  - Mapping to original domain using new location possible
- Needs computations to do this
- Occurs frequently in data structure building,work distribution, etc.

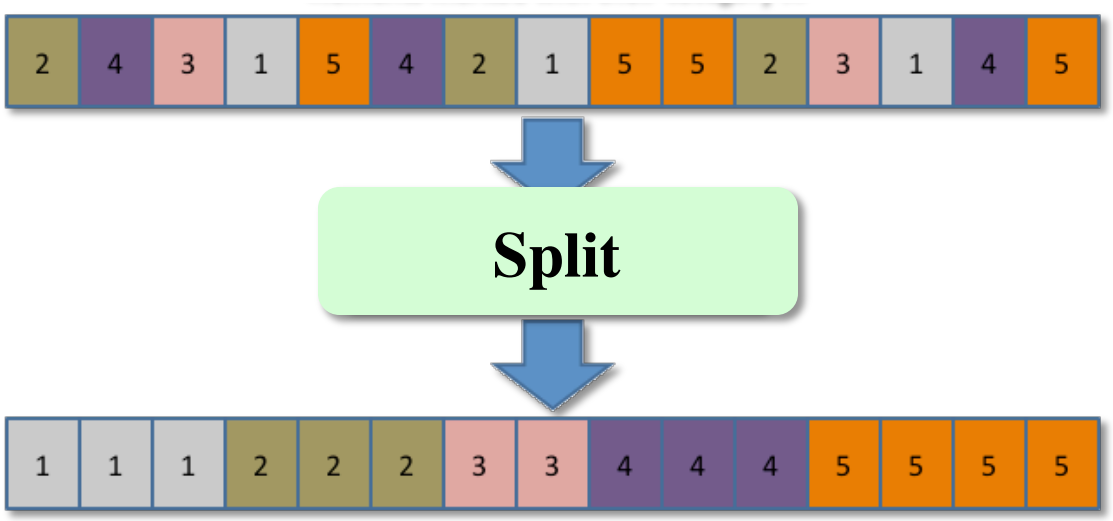


Data-Parallel Primitives

- Deep knowledge of architecture needed to get high performance
  - Use primitives to build other algorithms
  - Efficient implementations on the architecture by experts
- **reduce, scan, segmented scan**: Aggregate or progressive results from distributed data
  - Ordering distributed info
- **split, sort**:
  - Mapping distributed data [Blelloch 1989]



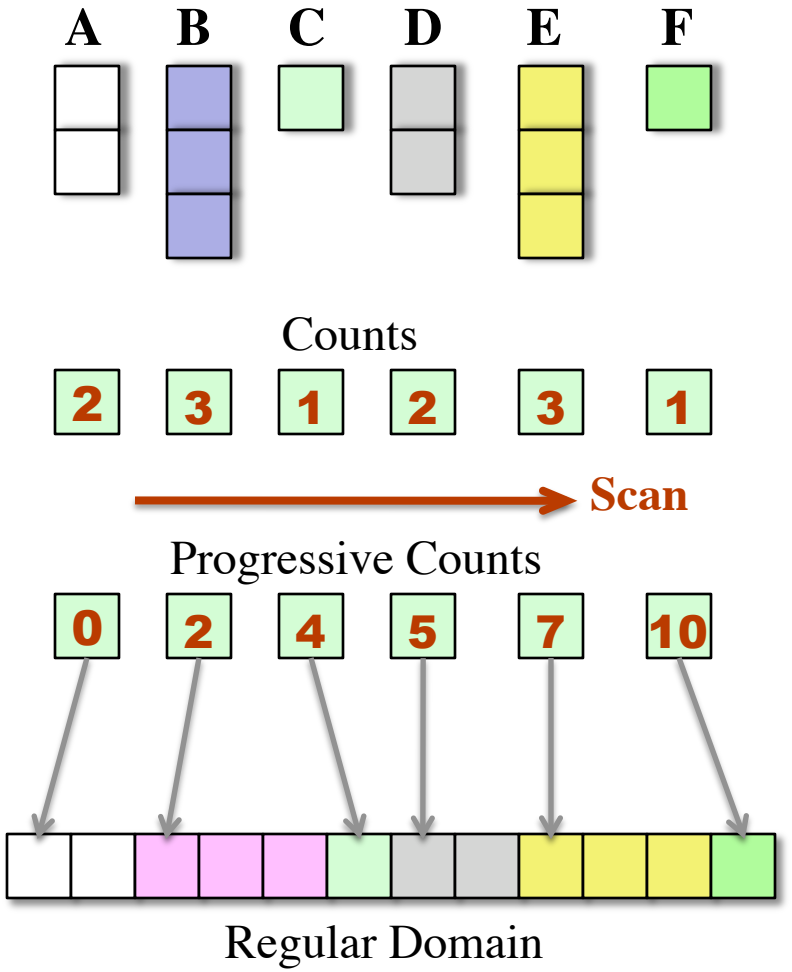
Split Primitive



- Rearrange data according to its category. Categories could be anything.
- Generalization of sort. Categories needn't ordered themselves
- Important in distributing or mapping data

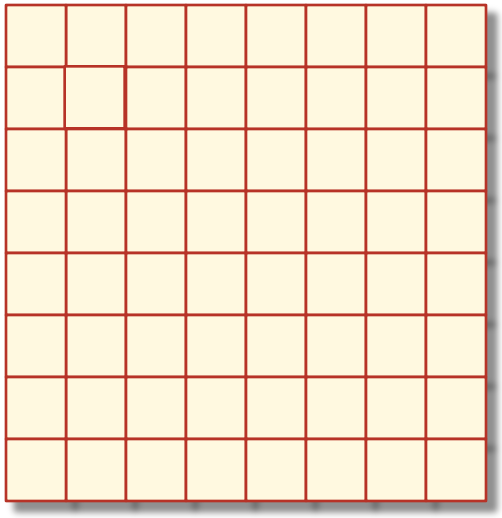
Handling Irregular Domains

- Convert from irregular to a regular domain
- Each old domain element counts its elements in new domain
- Scan the counts to get the progressive counts or the starting points
- Copy data elements to own location

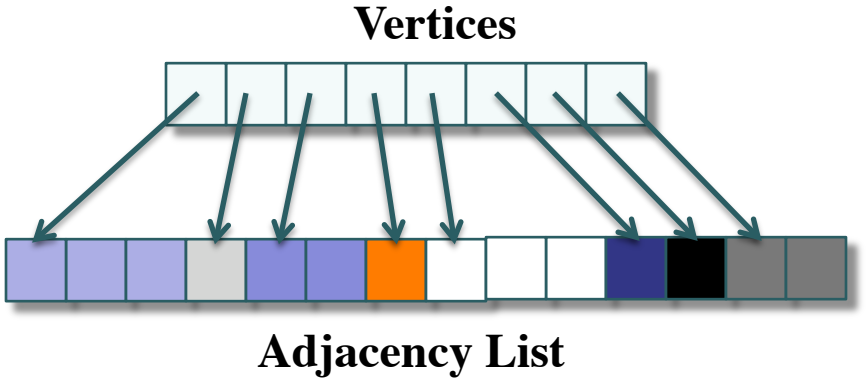


Graph Algorithms

- Not the prototypical data-parallel application; an irregular application.
- Source of data-parallelism: Data structure (adjacency matrix or adjacency list)
- A 2D-domain of V^2 elements or a 1D-domain of **E** elements
- A thread processes each edge in parallel. Combine the results



Adjacency Matrix



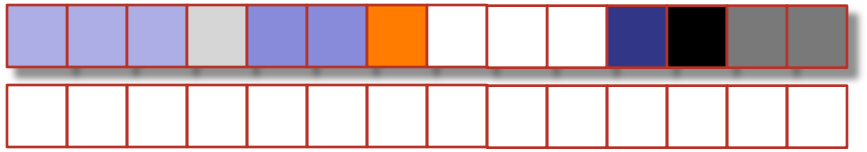
Find min edge for each vertex

Example: Find the minimum outgoing edge of each vertex

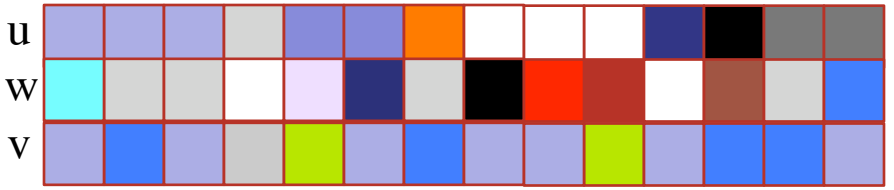
Soln 1: Each node-kernel loops over its neighbors, keeping track of the minimum weight and the edge

```
for each node in parallel
  for all neighbours v
    if w[v] < min
        min = w[v]
        mv = v
```

Soln 2: Segmented min-scan of the weights array + a kernel to identify min vertex

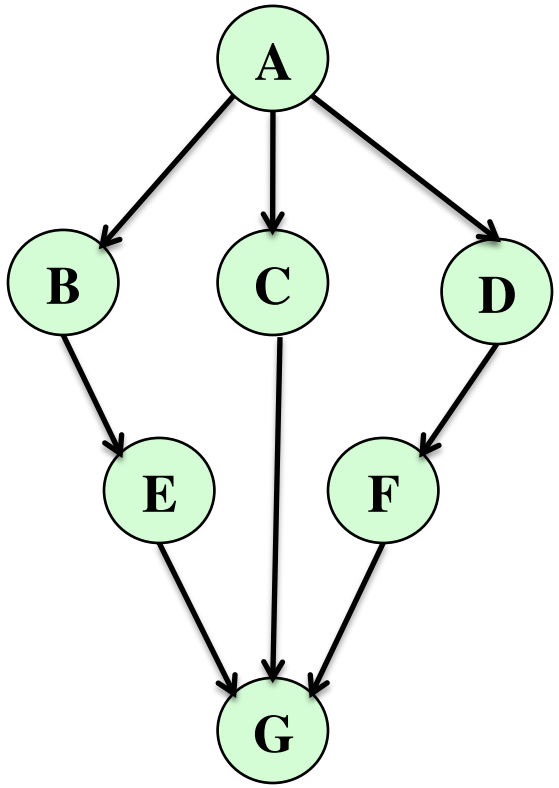


Soln 3: Sort the tuple (u, w, v) using the key (w, v) for all edges (u, v) of the graph of weight w. Take the first entry for each u.



Task Parallel Computing

- The problem is divided into a number of tasks; Data may also be partitioned or shared
- Some can be done in parallel, others depend on previous results
- Combine the results finally
- CPU cores and GPU can be doing task-parallel computing
- OpenCL supports this model of computation as well as the pipelined model
- More on OpenCL later today



Summary

- GPU can be an essential computing platform with a massively multithreaded programming model
- Data-parallel model fits the GPUs best.
- High performance requires deep knowledge of the architecture. High-level primitives can alleviate this greatly.
- Think of CPU **and** GPU together achieving your computing goals. Not one instead of the other
- OpenCL is an exciting new development that can make this possible and portable!

For More Information

- GPGPU: gpgpu.org
- SIGGRAPH Courses:
  - SIGGRAPH 2008: Available at UC, Davis. http://s08.idav.ucdavis.edu/
  - SIGGRAPH Asia 2008: Available at UC, Davis http://sa08.idav.ucdavis.edu/
  - Upcoming course at SIGGRAPH 2009
- CudaZone for Nvidia
- And more …

Thank you!

Image credits to owners such as Intel, Nvidia, AMD/ATI, etc.

Thank you!

Image credits to owners such as Intel, Nvidia, AMD/ATI, etc.