The GPGPU Architectural, Programming, and Performance Models

Kishore Kothapalli and P. J. Narayanan

International Institute of Information Technology
Hyderabad, India.

Suryakant Patidar

Nvidia Corporation, Bangalore, India.

Synopsis

- Kishore Kothapalli, IIIT Hyderabad
  - High-performance Computing, Graph Theory
- P. J. Narayanan, IIIT Hyderabad
  - High-performance Computing, Computer Vision
- Suryakant Patidar, Nvidia, Bangalore
  - High performance Computing

Synopsis

- GPUs as the main-stream computing platform.
  - Can deliver up to 1 Teraflop at low $$.
  - Have matured from OpenGL extensions to vendor specific C-like extensions such as CUDA.
- GPGPU
  - Use GPUs for also general purpose computing
  - Lots of success stories in computer vision, sorting, and several other domains
  - But, applications need to re-interpreted in massively multi-threaded form to work on GPUs.

Synopsis

- What is the architectural model of a GPU?
- What is the programming abstraction?
- Example Programs
- Regular vs. Irregular Applications
- Performance Modelling.

Assumptions

- Basic knowledge of computer architecture.
- Basic knowledge of sequential algorithms.
- Basic knowledge of programming.

Schedule

0:00 -- 0:10 : Introduction to the Tutorial, Theme, Speakers.

0:10 -- 0:30 : Basic Concepts -- CPU Architectures, GPUs -- evolution, comparison to earlier models of parallel computing

0:30 -- 0:55 : GPU Architectures in Detail -- NVidia architecture, Intel Larrabee architectural features

0:55 -- 1:30 : GPU Programming models with short examples, CUDA

BREAK

1:50 -- 2:15 : Case studies of regular applications on the GPU

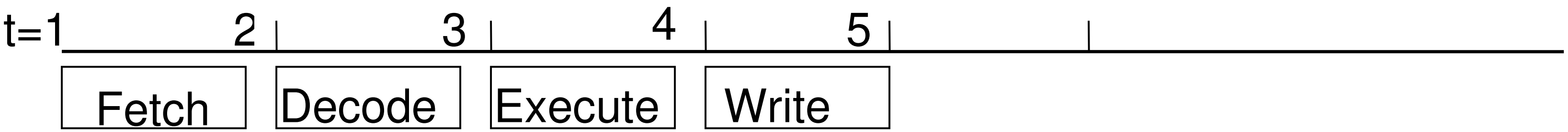
2:15 -- 2:45 : Case studies of irregular applications on the GPU

2:45 -- 3:20 : GPU Analytical Models

Design Space Optimization, Performance Prediction

3:20 -- 3:30 : Concluding remarks, discussion

Basic Architecture Concepts

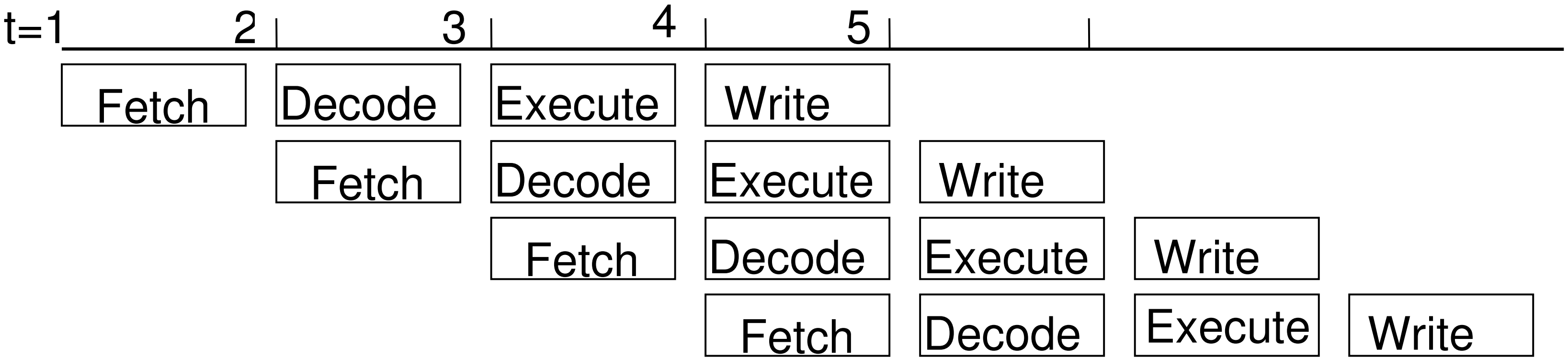


t=1 2 3 4 5

Fetch Decode Execute Write

- CPU Architecture
  - 4 stages of instruction execution
    - Too many cycles per instruction (CPI)

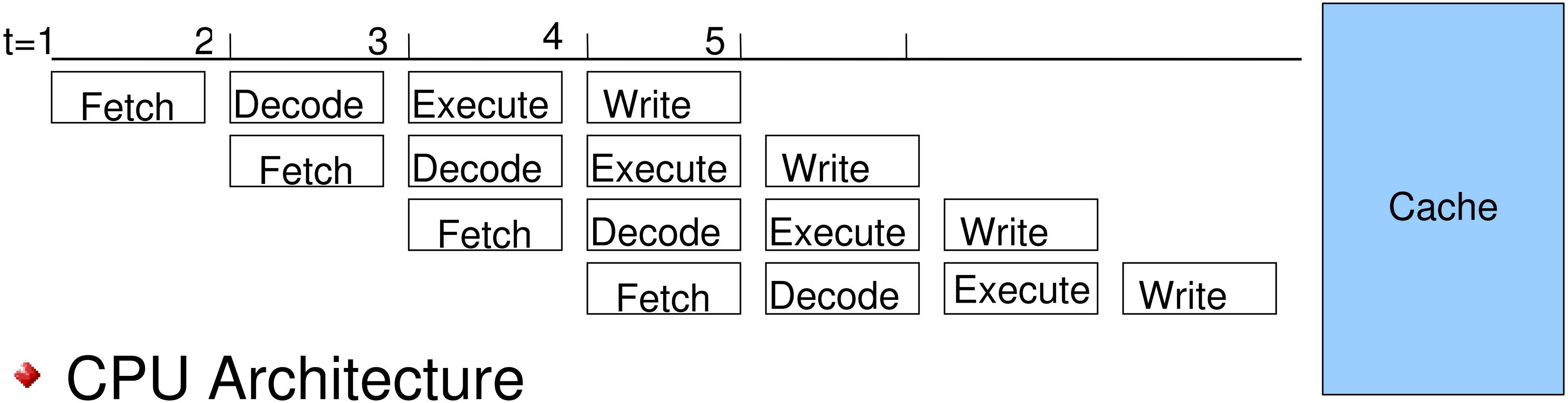
Basic Architecture Concepts



t=1	2	3	4	5		
Fetch	Decode	Execute	Write			
	Fetch	Decode	Execute	Write		
		Fetch	Decode	Execute	Write	
			Fetch	Decode	Execute	Write

- CPU Architecture
  - 4 stages of instruction execution
    - Too many cycles per instruction (CPI)
  - To reduce the CPI, introduce pipelined execution

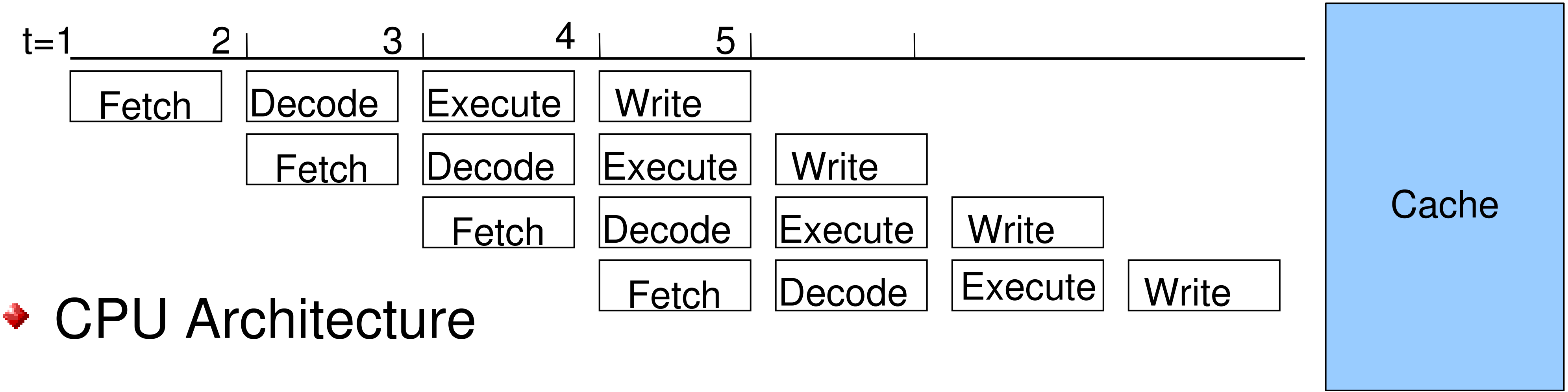
Basic Architecture Concepts



t=1	2	3	4	5		
Fetch	Decode	Execute	Write			
	Fetch	Decode	Execute	Write		
		Fetch	Decode	Execute	Write	
			Fetch	Decode	Execute	Write

- CPU Architecture
  - 4 stages of instruction execution
    - Too many cycles per instruction (CPI)
  - To reduce the CPI, introduce pipelined execution
    - Needs buffers to store results across stages.
    - A cache to handle slow memory access times

Basic Architecture Concepts



t=1	2	3	4	5		
Fetch	Decode	Execute	Write			
	Fetch	Decode	Execute	Write		
		Fetch	Decode	Execute	Write	
			Fetch	Decode	Execute	Write

- CPU Architecture
  - 4 stages of instruction execution
    - Too many cycles per instruction (CPI)
  - To reduce the CPI, introduce pipelined execution
    - Needs buffers to store results across stages.
    - A cache to handle slow memory access times
    - Multilevel caches, out-of-order execution, branch prediction, ...

Basic Architecture Concepts

- CPU architecture getting too complex.
- Not translating to equivalent performance benefits
  - Need a rethink on traditional CPU architectures.

Basic Architecture Concepts

- Couple with this the new wisdom in computer architectures.
  - Memory Wall – memory latencies far higher
  - ILP Wall – Reducing benefits from instruction level parallelism
  - Power Wall – Increase in power consumption with increase in clock rates.
- Multi-core is the way forward
  - Ex: GPUs, Cell, Intel Quad core, ...
  - Predicted that 100+ core computers would be a reality soon.

Multicore and Manycore Processors

- IBM Cell
- NVidia GeForce 8800 includes 128 scalar processors and Tesla
- Sun T1 and T2
- Tilera Tile64
- Picochip combines 430 simple RISC cores
- Cisco 188
- TRIPS

The Case for the GPUs

- GPUs are now common. They also have high computing power per dollar, compared to the CPU
- Today’s computer system has a CPU and a GPU, with the GPU being used primarily for graphics.
- GPUs are good at some tasks and not so good at others. They are especially good at processing large data such as images.
- Let us use the right processor for the right task.
- Goal: Increase the overall throughput of the computer system on the given task. Use CPU and GPU synergistically.

Evolution of GPUs

- Graphics: a few hundred triangles/vertices map to afewhundred 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

GPUs as a Case Study

- GPGPU – General Purpose Programming on GPUs
- OpenGL extensions
  - Very difficult to program
- Recently manufactures started supporting C-like programming abstraction to program GPUs
  - CUDA from NVidia
- Other benefits of GPGPU
  - Affordable cost, easy availability, computational power

GPUs as a Case Study

- GPUs suited for routines with high arithmetic intensity.
- One feature is high memory latency, depending on the nature of access.
  - Should overlap memory with arithmetic.

CPU Vs GPU

- Few powerful cores Vs. lots of small cores
- GPUs: For good performance, applications need high arithmetic intensity
- GPUs : No system managed cache.

GPGPU as a Case Study

- Regular algorithms
  - Map well to data parallel model of GPUs
  - Each work item operates by itself or with a few neighbors
  - Example settings : image processing.
  - Threads can share data, e.g., apron pixels in an image processing kernel.

GPU as a Case Study

- Irregular algorithms
  - Applications with data accesses that are not regular in nature.
  - Occurs in settings such as graph algorithms, data structures building, etc.
  - Difficult to get high efficiency due to high memory latency of accesses.

GPGPU Tools and APIs

- OpenGL
- CUDA
- OpenCL
- Brook

GPU Architecture: Overview

P J Narayanan
Centre for Visual Information Technology
IIIT, Hyderabad

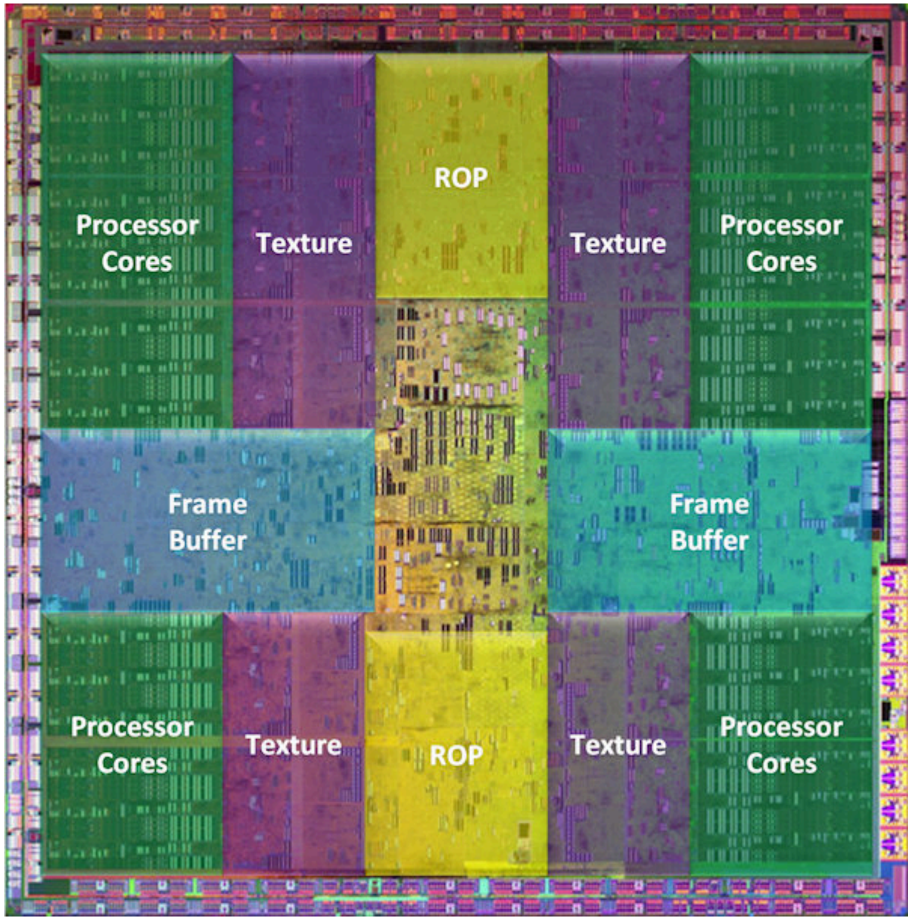
PPoPP Tutorial on
GPU Architecture, Programming and Performance Models

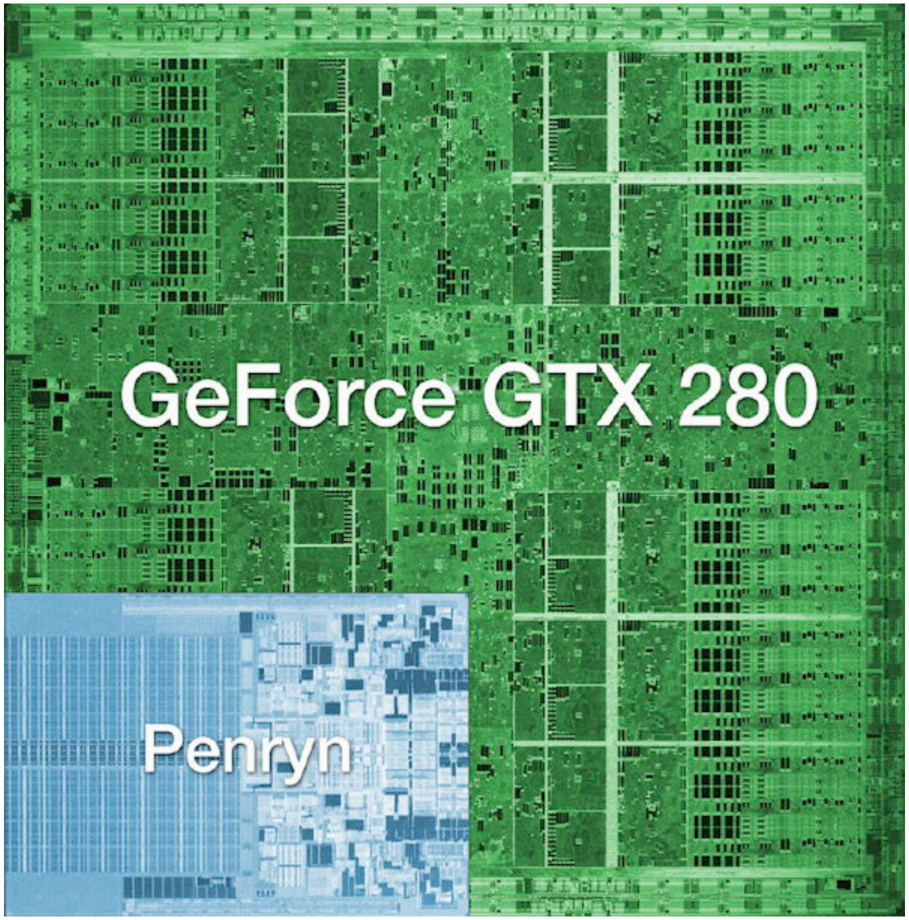
GPU: Evolution

- 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

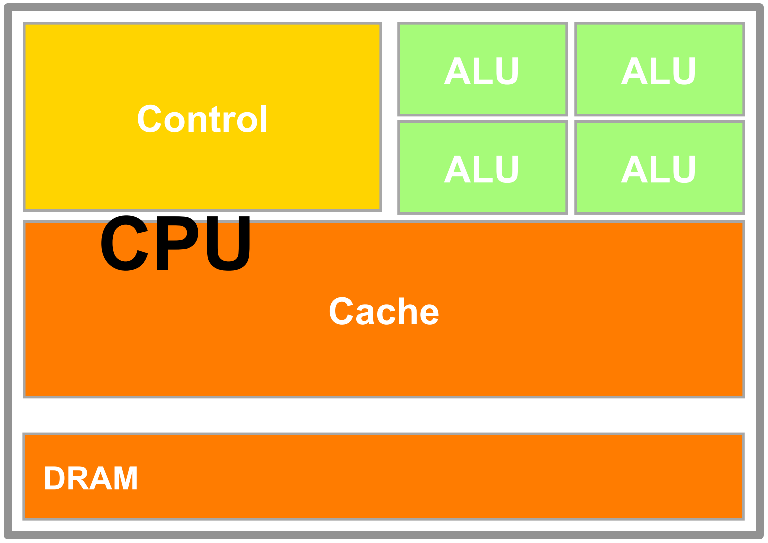
Nvidia GTX280

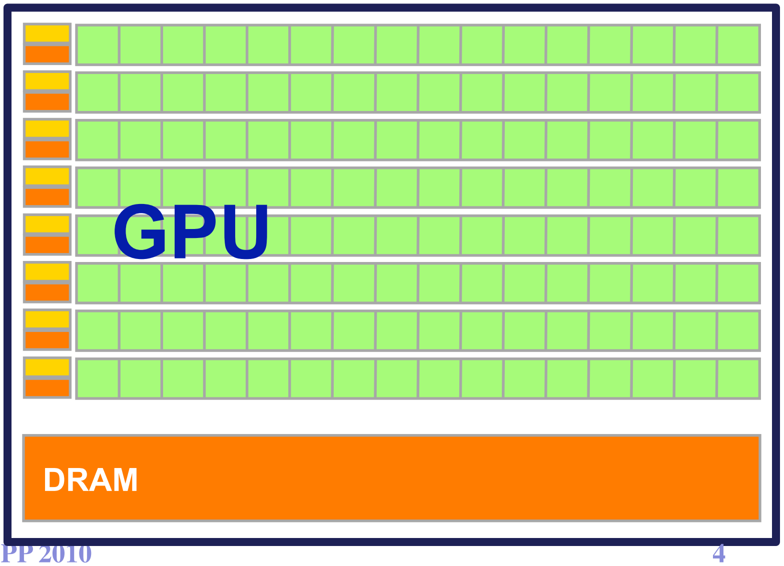




CPU vs GPU

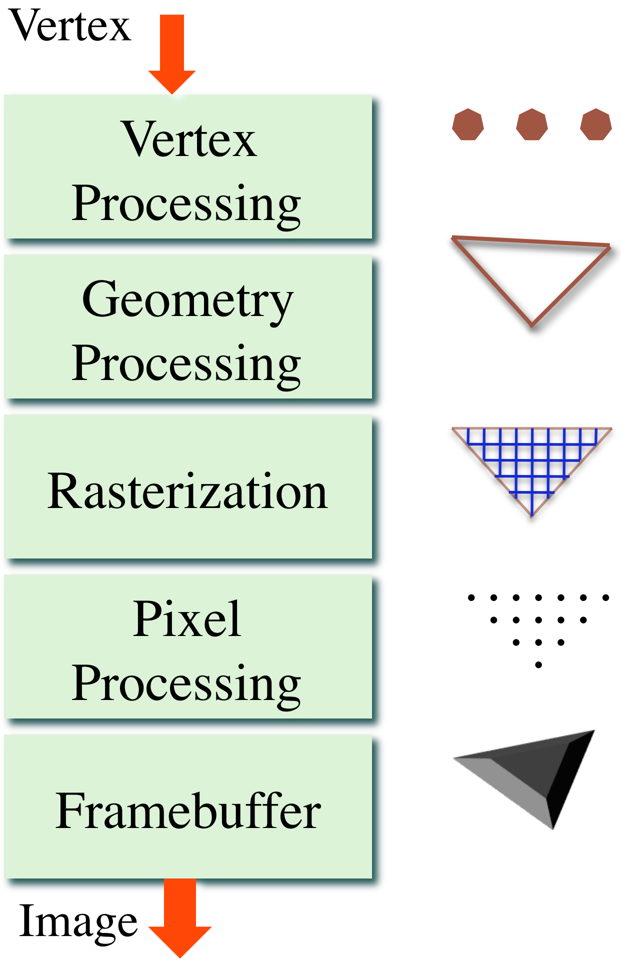
- CPU Architecture features:
  - Few, complex cores
  - Perform irregular operations well
    - Run an OS, control multiple IO, pointer manipulation, etc.
- GPU Architecture features:
  - Hundreds of simple cores, operating on a common memory (like the PRAM model)
  - High compute power but high memory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





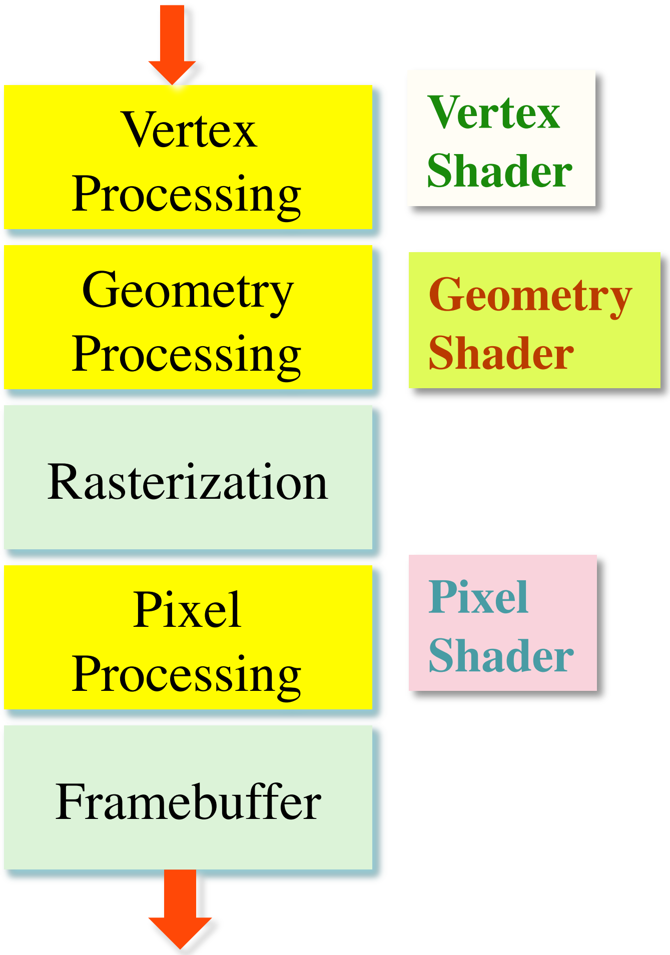
What do GPUs do?

- 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.

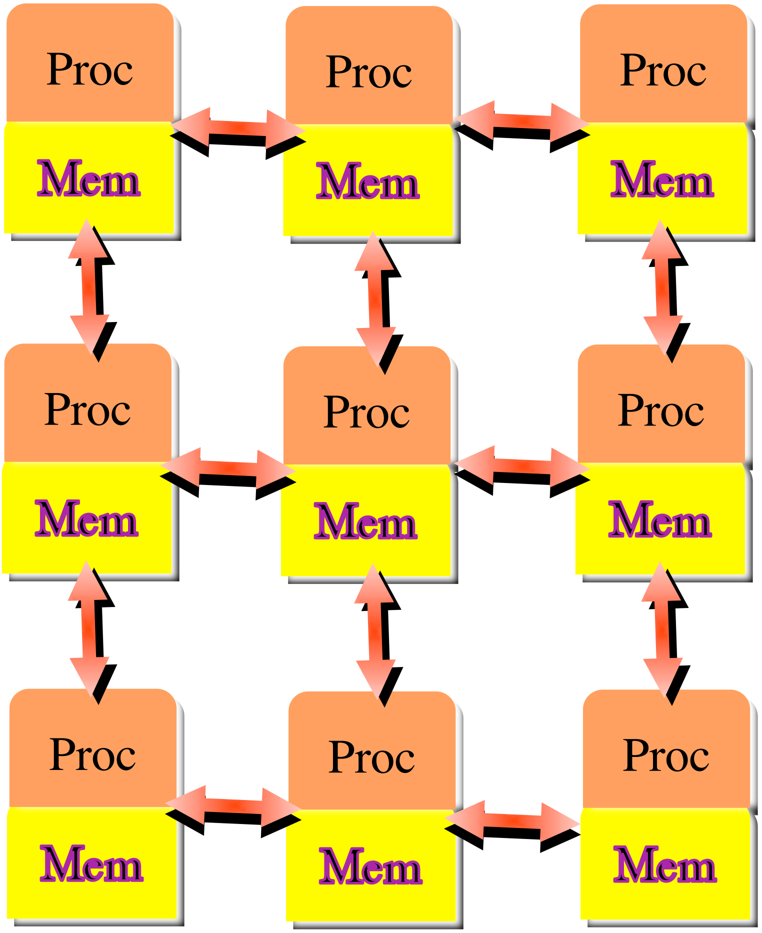


New Generation GPUs

- 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



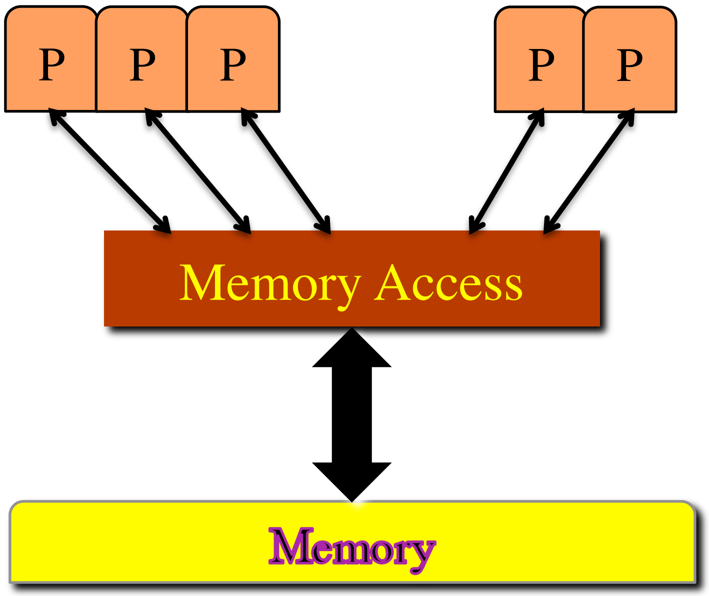
Proc Mem | Proc Mem | Proc Mem

Proc Mem | Proc Mem | Proc Mem

Proc Mem | Proc Mem | Proc Mem

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

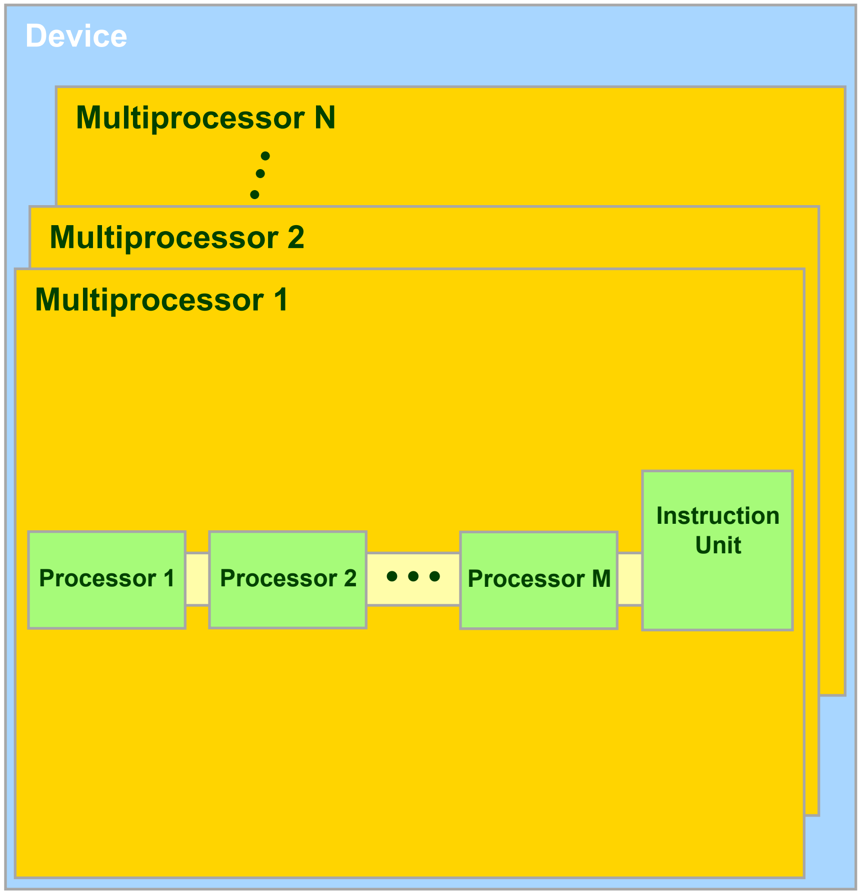
  with high SP floating point arithmetic intensity

  - Many calculations per memory access
  - Currently also need high floating point to integer ratio

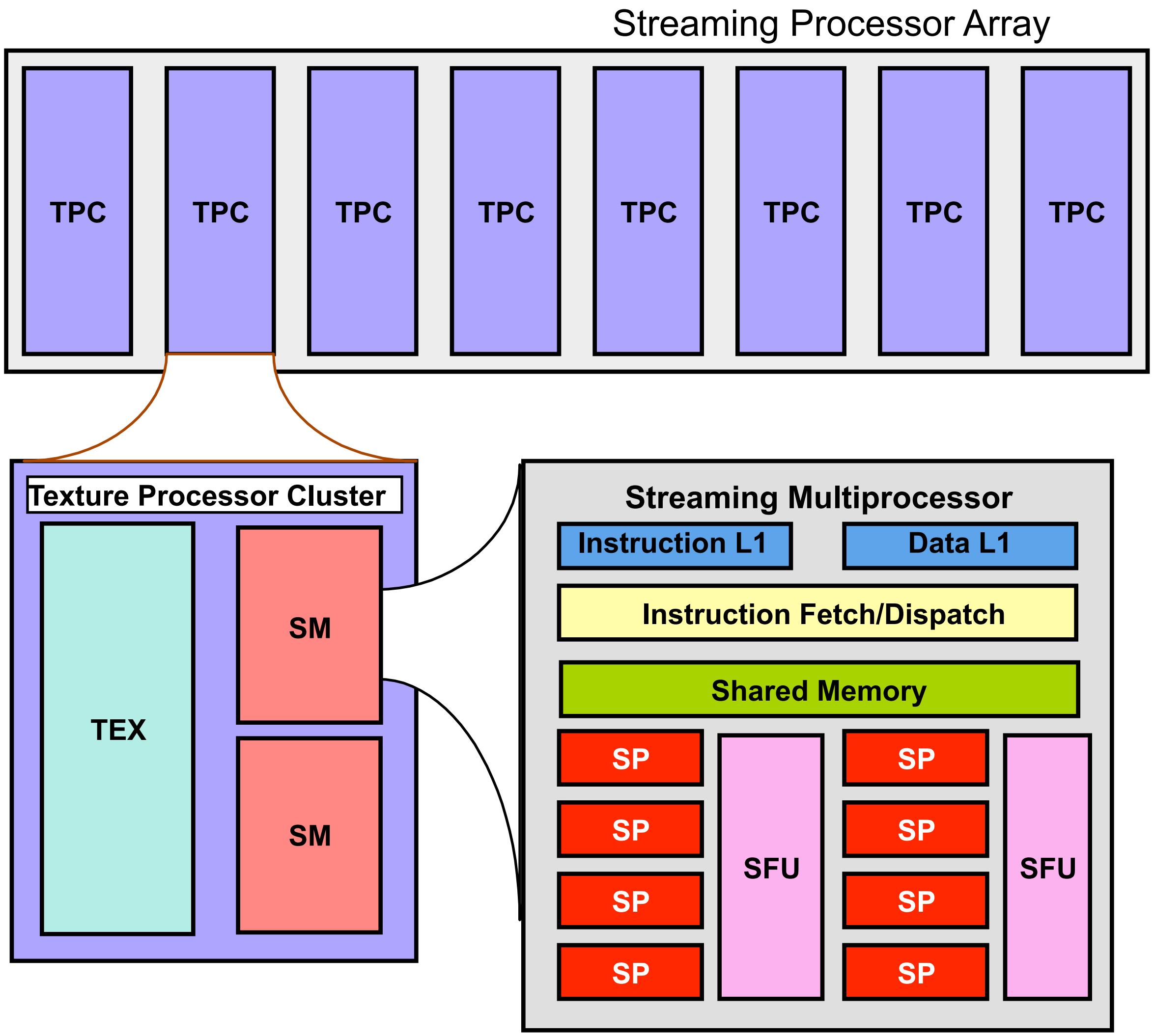
- High floating-point arithmetic intensity and many data elements can hide memory access latency without big data cache

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

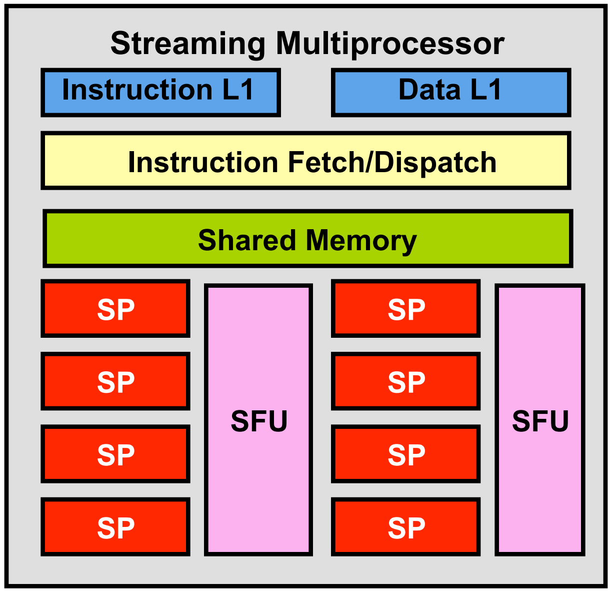


HW Overview


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

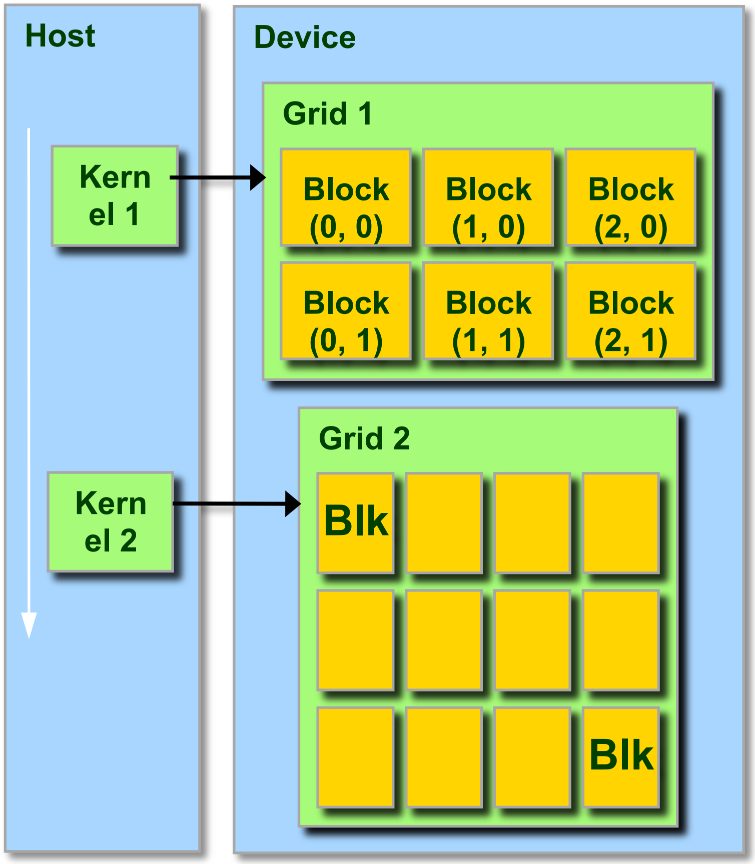


Multithreaded Coprocessor

- 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

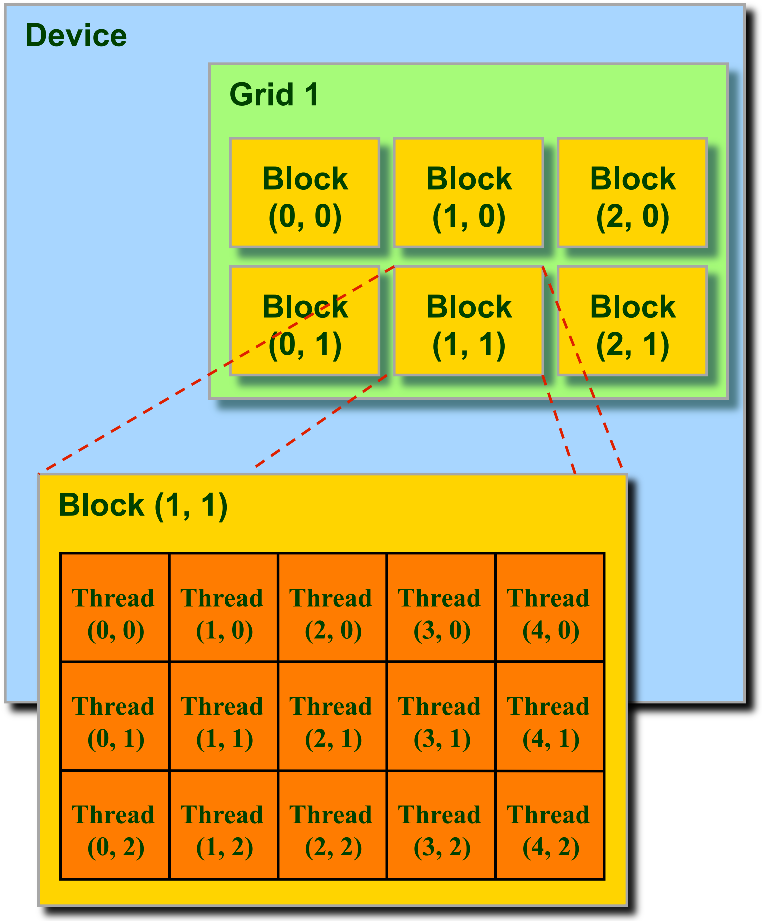
- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared 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
  - …



Device

Grid 1

Block (0, 0) Block (1, 0) Block (2, 0)

Block (0, 1) Block (1, 1) Block (2, 1)

Block (1, 1)

Thread (0, 0) Thread (1, 0) Thread (2, 0) Thread (3, 0) Thread (4, 0)

Thread (0, 1) Thread (1, 1) Thread (2, 1) Thread (3, 1) Thread (4, 1)

Thread (0, 2) Thread (1, 2) Thread (2, 2) Thread (3, 2) Thread (4, 2)

Courtesy: NDVIA

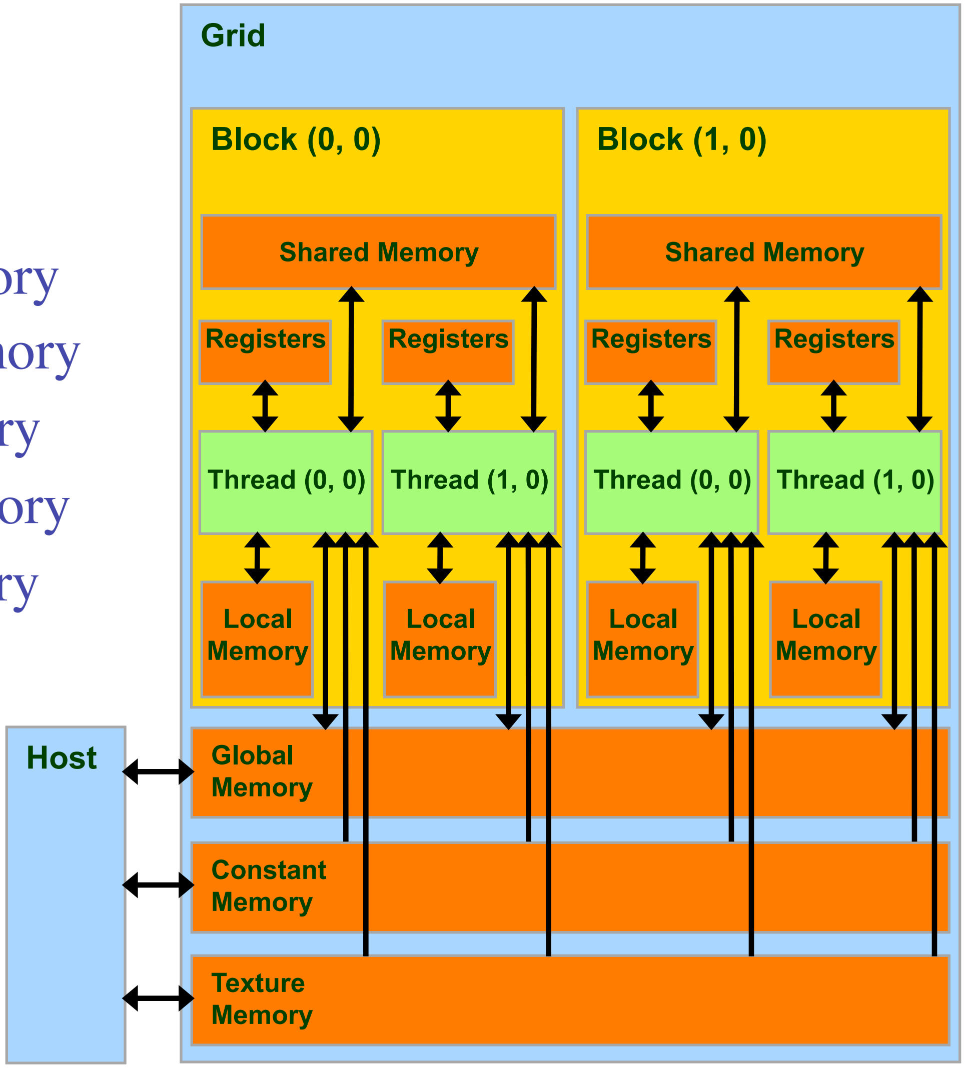
Threads, Warps, Blocks

- 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 *warps* 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

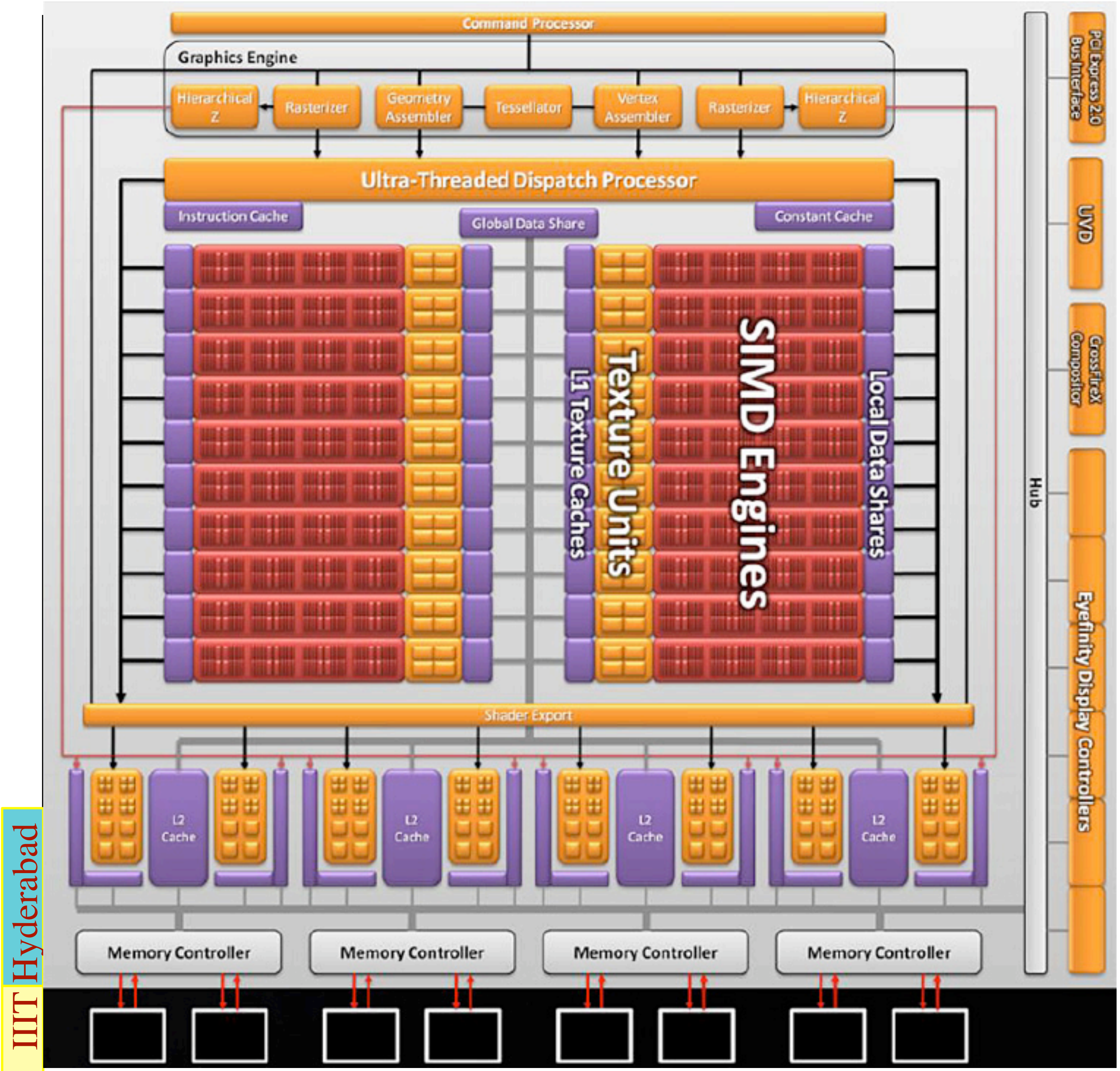
If more blocks are needed, they use the hardware sequentially.

IIIT Hyderabad

Processors, Memory

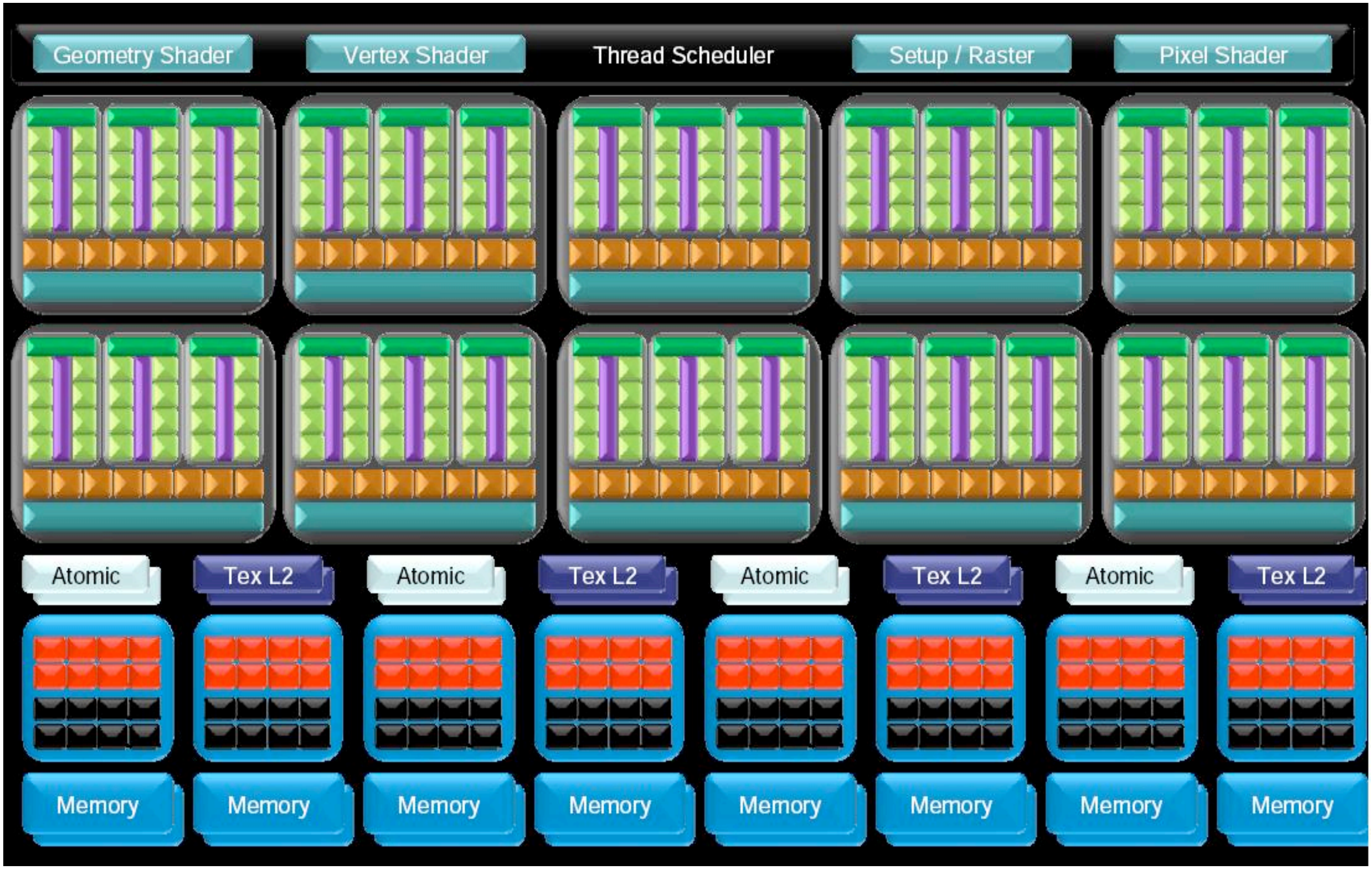
- Nvidia 280GTX: 240 Streaming Processors, grouped into 30 Streaming Multiprocessors
  - One instruction sequencer per SM
  - 16KB of on-chip shared memory per SM
  - 16K 32-bit registers per SM
  - Single clock access of registers, shared memory
- 1 GB of common, off-chip global memory
  - 130 GB/s 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

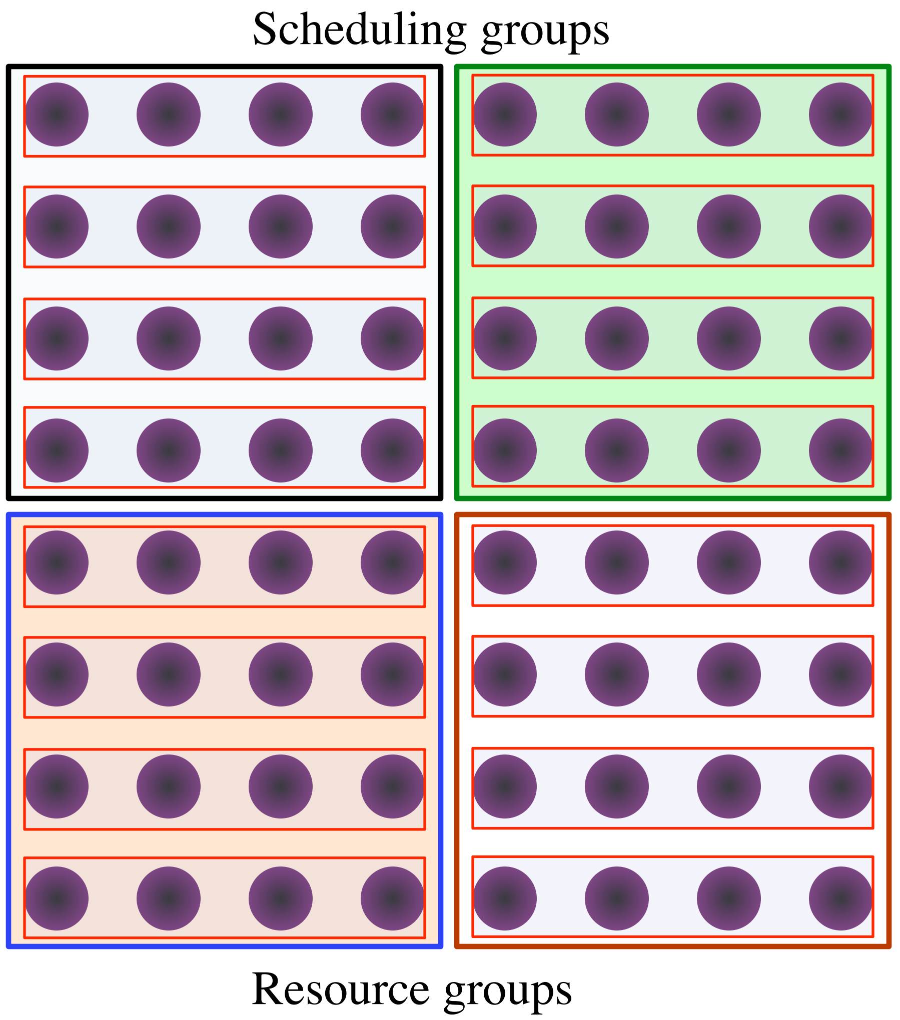
- OpenGL/Direct3D for older, GPGPU exposure
  - Shaders operating on polygons, textures, and framebuffer
- CUDA: an alternate interface from Nvidia
  - Kernel operating on grids using threads
  - Extensions of the C language
- DirectX Compute Shader: Microsoft's version
- OpenCL: A promising open compute 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

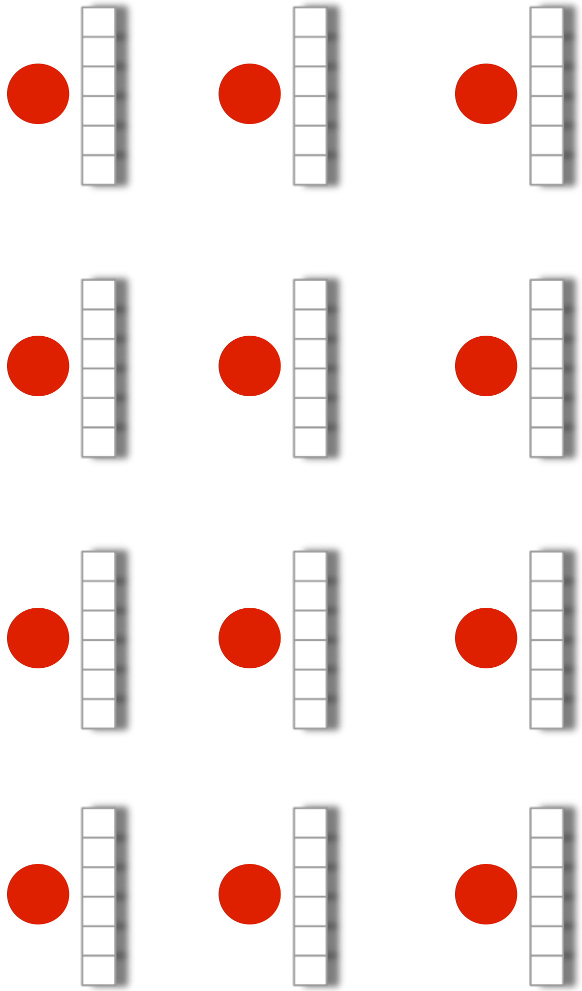
Multi-tier Thread Structure

- 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*



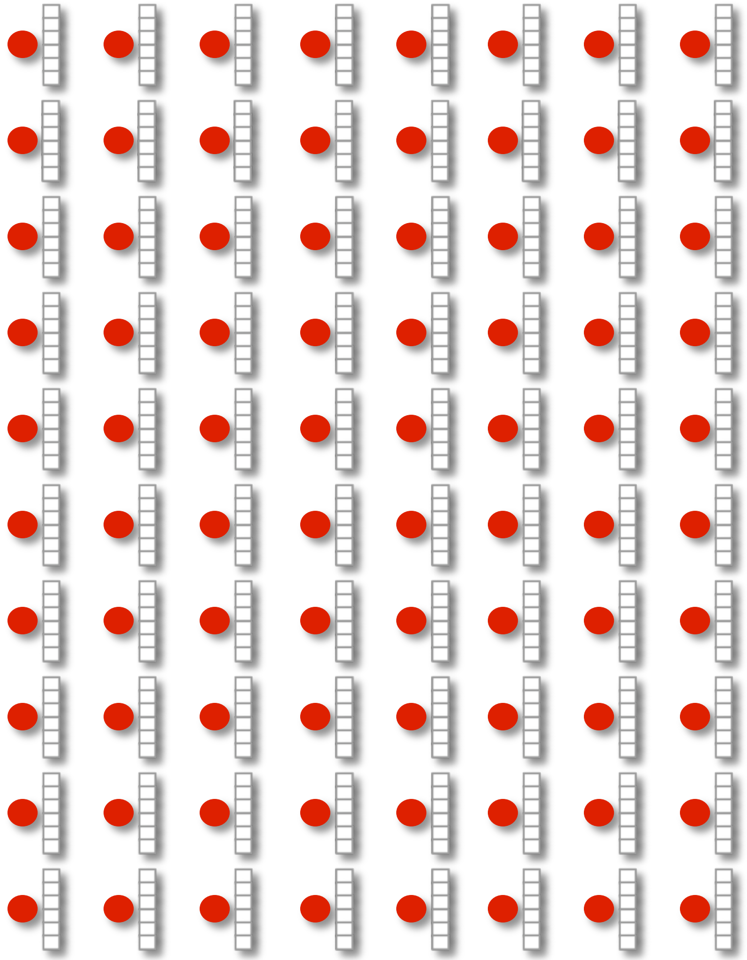
Data-Parallelism

- 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

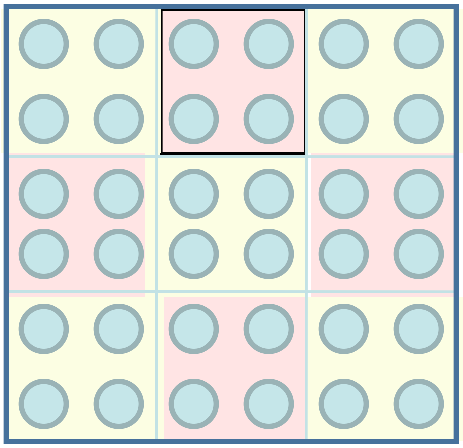


Thinking Data-Parallel

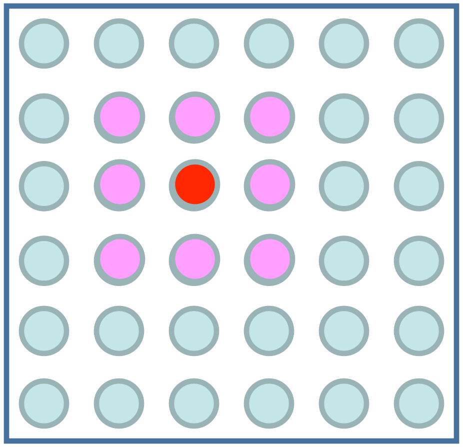
- Launch *N* data ***locations***, each of which gets a ***kernel*** of code
- Data follows a *domain* of computation.
- Each invocation of the kernel is aware of its location *loc* within the domain
  - Can access different data elements using the *loc*
  - May perform different computations also
- 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

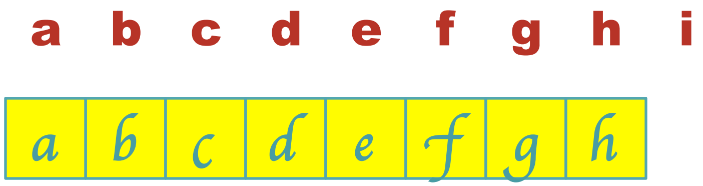


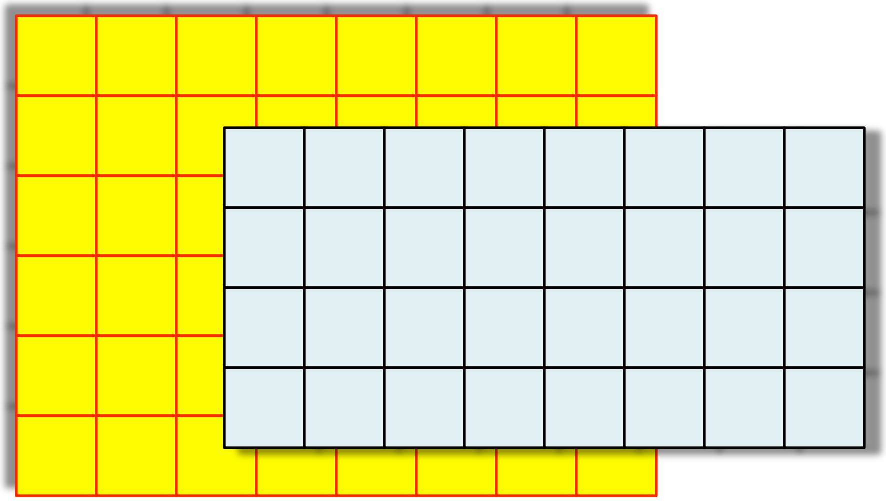
3 x 3 Filtering



Regular Domains

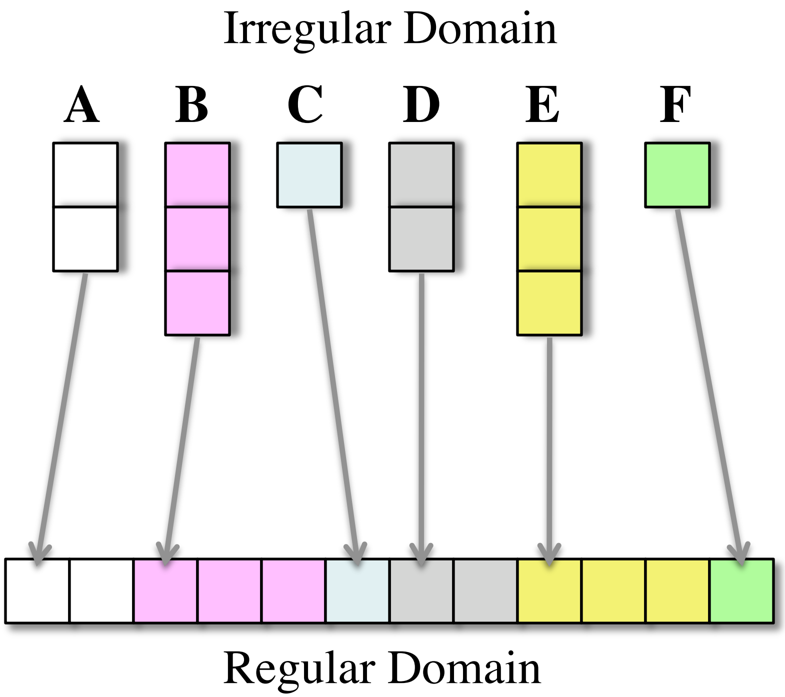
- Regular *1D*, *2D*, and *nD* domains map very well to data-parallelism
- Each work-item operates by itself or with a few neighbors
- Need not be of equal dimensions or length
- A mapping from *loc* to each domain should exist





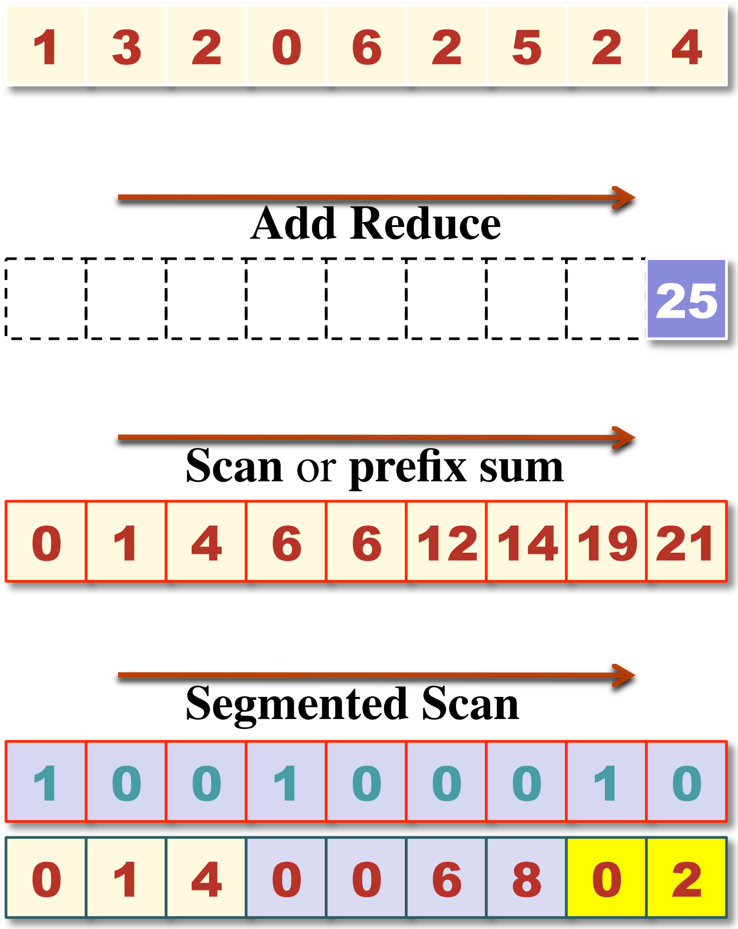
Irregular Domains

- 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]



1	3	2	0	6	2	5	2	4

Add Reduce

								25

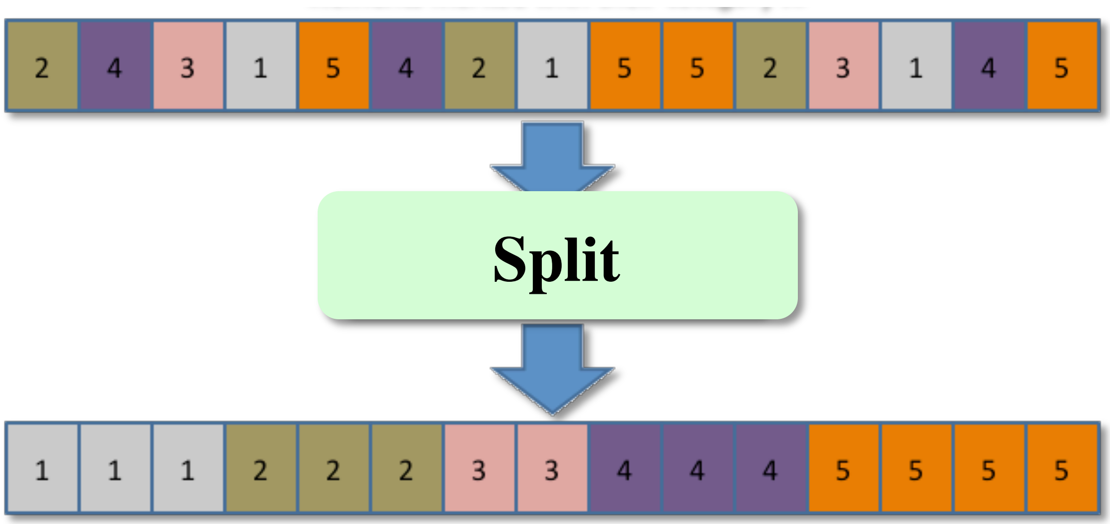
Scan or **prefix sum**

0	1	4	6	6	12	14	19	21

Segmented Scan

1	0	0	1	0	0	0	1	0
0	1	4	0	0	6	8	0	2

Split Primitive



2 4 3 1 5 4 2 1 5 5 2 3 1 4 5

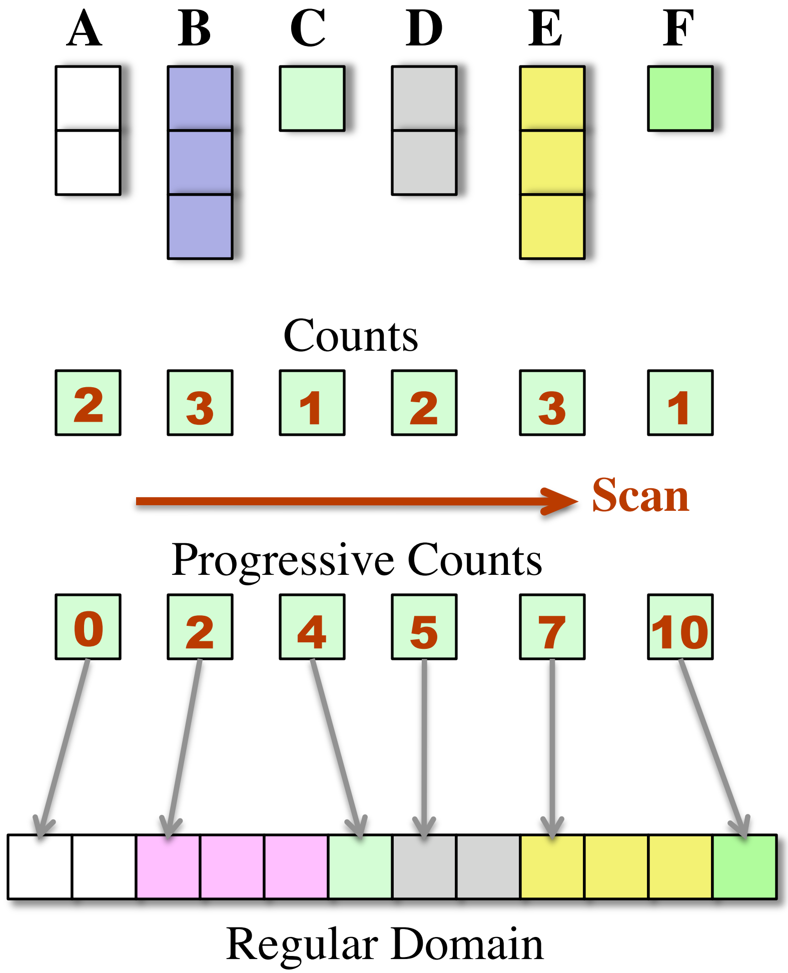
Split

1 1 1 2 2 2 3 3 4 4 4 5 5 5 5

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

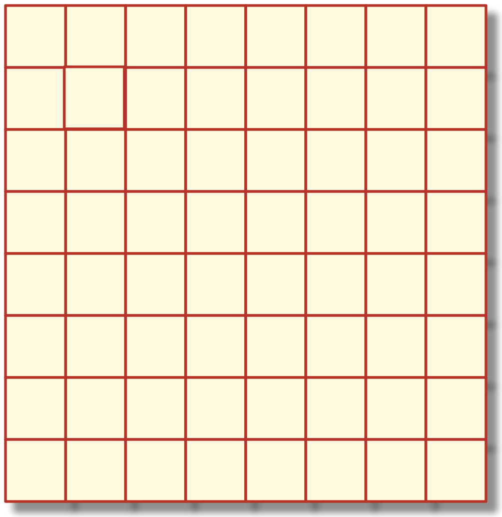
Handling Irregular Domains

- 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

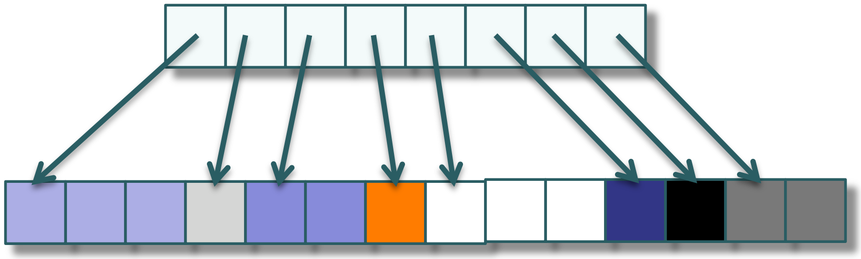


Graph Algorithms

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



Adjacency Matrix



Vertices

Adjacency List

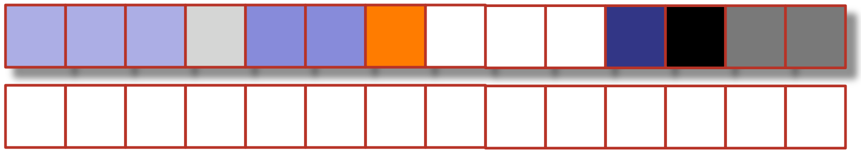
Find min edge for each vertex

Example: Find the minimum outgoing edge of each vertex

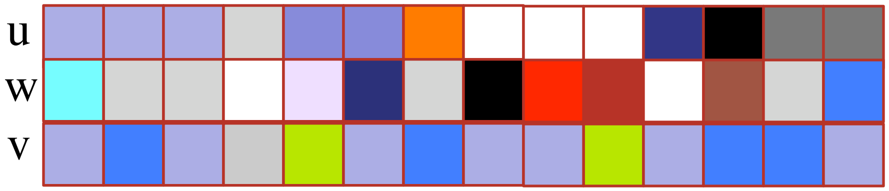
Soln 1: Each node-kernel loops over its neighbors, keeping track of the minimum weight and the edge

```
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

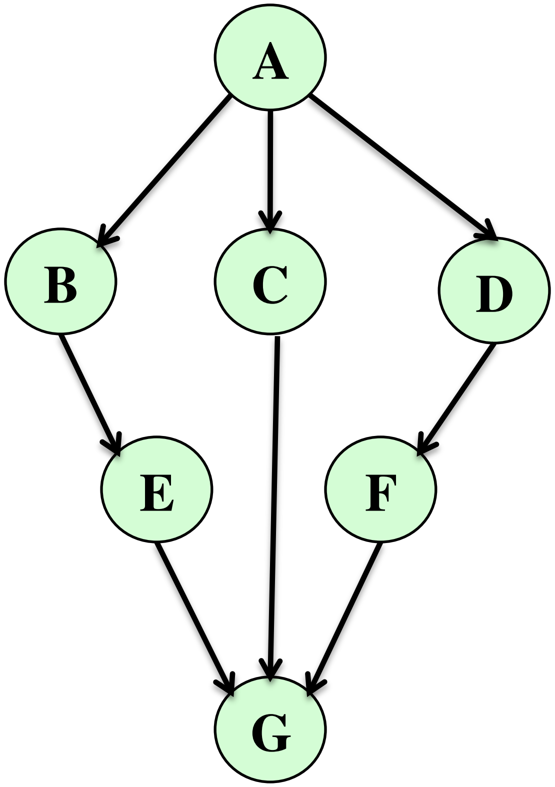


Soln 3: Sort the tuple (*u*, *w*, *v*) using the key (*w*, *v*) for all edges (*u*, *v*) of the graph of weight *w*. Take the first entry for each *u*.



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.

Schedule

0:00 -- 0:10 : Introduction to the Tutorial, Theme, Speakers.

0:10 -- 0:30 : Basic Concepts -- CPU Architectures, GPUs -- evolution, comparison to earlier models of parallel computing

0:30 -- 0:55 : GPU Architectures in Detail -- NVidia architecture, Intel Larrabee architectural features

0:55 -- 1:30 : GPU Programming models with short examples, CUDA

BREAK

1:50 -- 2:15 : Case studies of regular applications on the GPU

2:15 -- 2:45 : Case studies of irregular applications on the GPU

2:45 -- 3:20 : GPU Analytical Models

Design Space Optimization, Performance Prediction

3:20 -- 3:30 : Concluding remarks, discussion

<<< GPU Programming >>>

Suryakant Patidar

spatidar@nvidia.com

Don’t just process, Compute !

- Graphics Processing on GPU
  - OpenGL, DirectX …
  - Games, Visualizations …
- Classic GPGPU : General Processing on the GPU
  - OpenGL, DirectX etc. ? ?
  - General Problems <faked as> Graphics Rendering Problems
  - Easy – Data Parallel, image processing
  - Hard – Irregular Algorithms, MST for a Sparse Graph
- Compute on GPU
  - NVIDIA CUDA – C for GPUs
  - OpenCL – Open Compute Library

Take Away

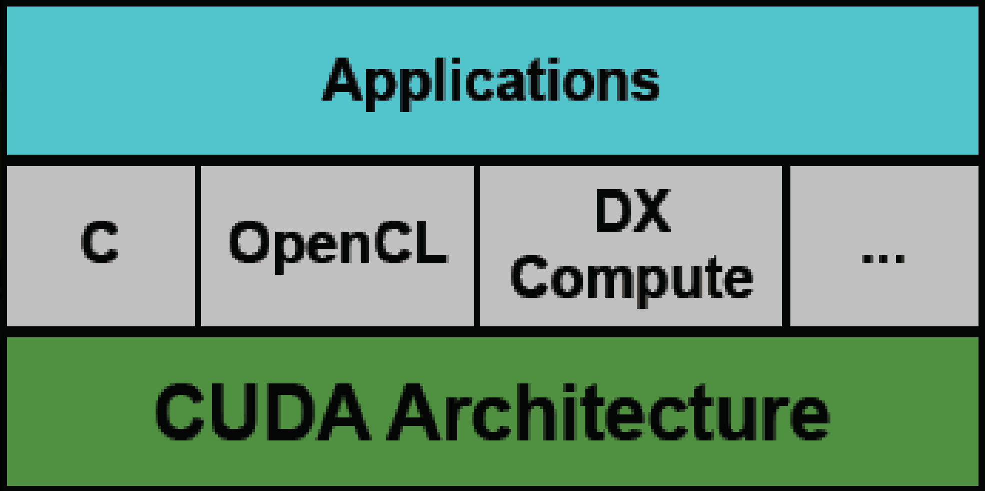
- Compute Architecture
  - C, OpenCL
- Hardware/Software Model
  - Processor & Memory Organization
  - Execution Model
- API
  - Language and Limitations

Compute Architecture

- GPU as Highly Multi-Threaded Co-Processor
  - 1000s of threads, not 2, not 4, not 8
  - 100s of tiny processors

- Supports standard languages and APIs
  - C (CUDA)
  - OpenCL
  - DX Compute

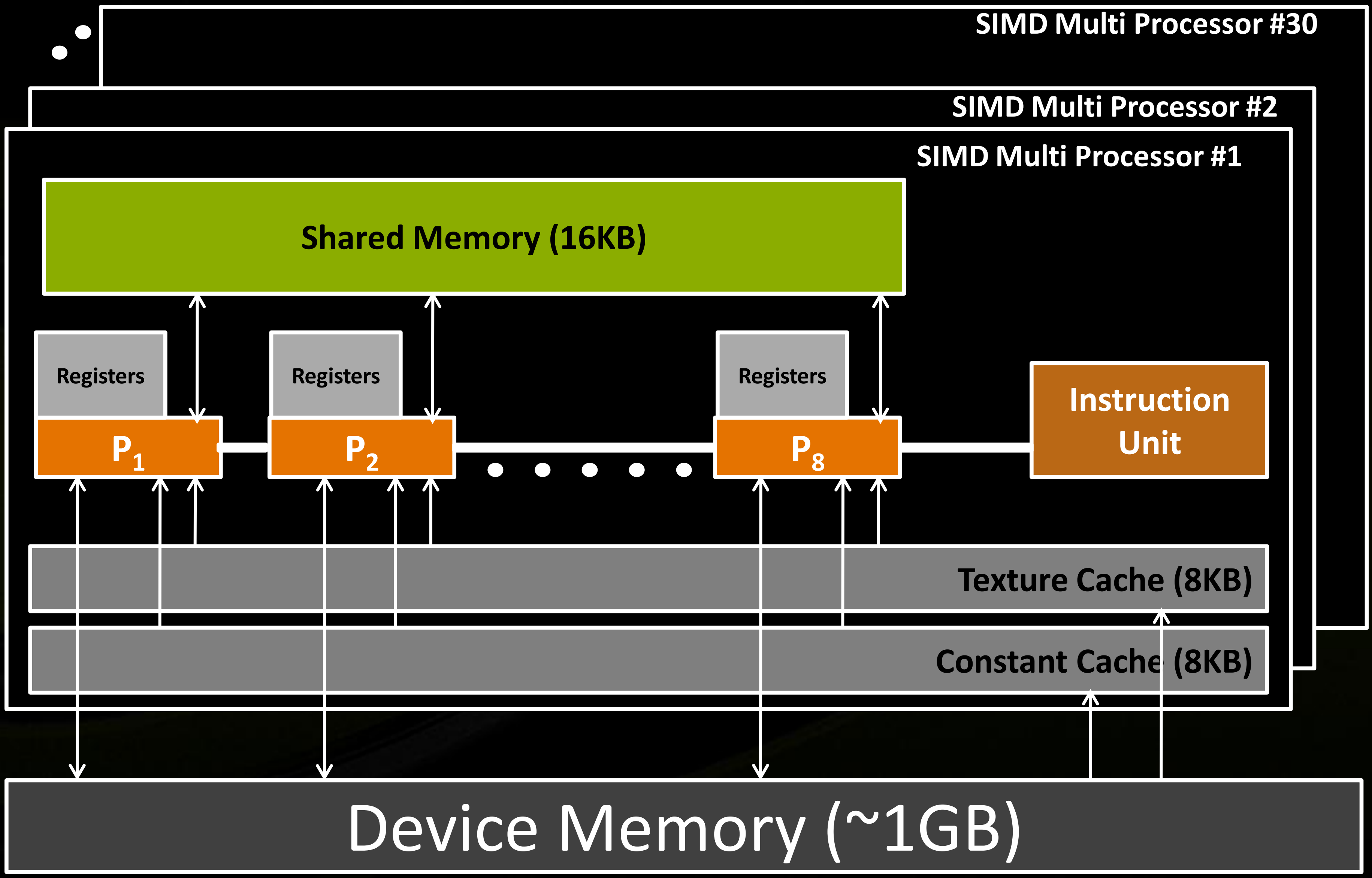
- Supported on common operating systems
  - Linux
  - MacOS
  - Windows



CUDA/OpenCL Programming

- Heterogeneous programming model
  - CPU and GPU are separate devices with separate memory spaces
- CPU code is standard C/C++
  - OpenCL : Look alike as OpenGL(C based)
  - CUDA : C/C++
- GPU code
  - OpenCL/CUDA : Subset of C with extensions

CUDA H/W Architecture



Software – Terminology

- Host – CPU
- Device – GPU

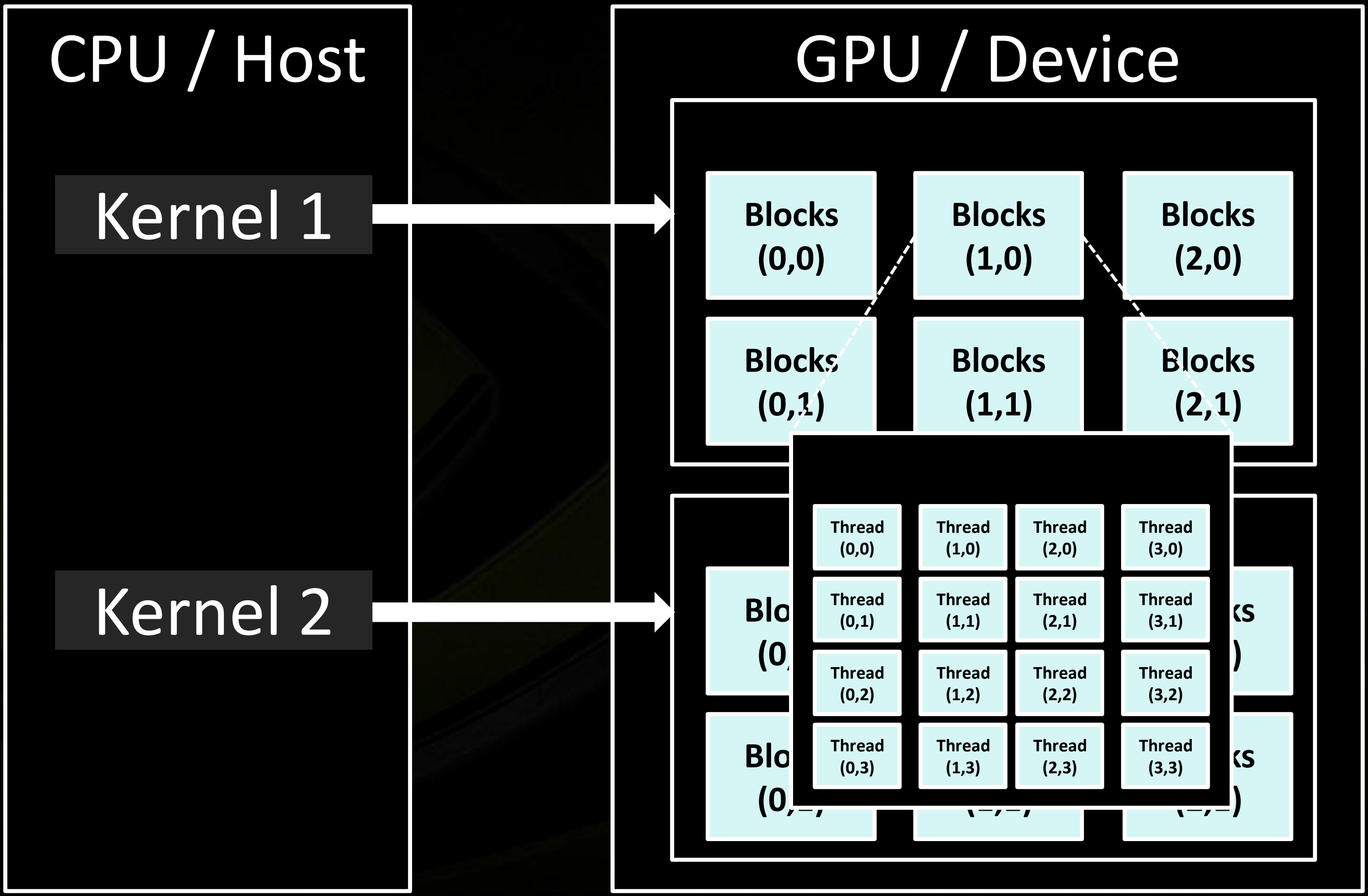
- Kernel – code which we wish to run on the GPU
- Thread/WorkItem – An Instance of a Kernel
- Block/WorkGroup – Group of Threads, bunched together
- Grid – Group of Blocks, one Grid – one Kernel

- OpenCL is very much inspired by CUDA, and given the GPU hardware is common to both, the APIs and approach are similar too

Kernels and Threads

- Parallel portions of an application are executed on the device as kernels
  - One kernel is executed at a time
  - Many threads execute each kernel
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Fast switching
  - CUDA uses 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

CUDA S/W Architecture



CPU / Host

Kernel 1

Kernel 2

GPU / Device

Blocks (0,0)

Blocks (1,0)

Blocks (2,0)

Blocks (0,1)

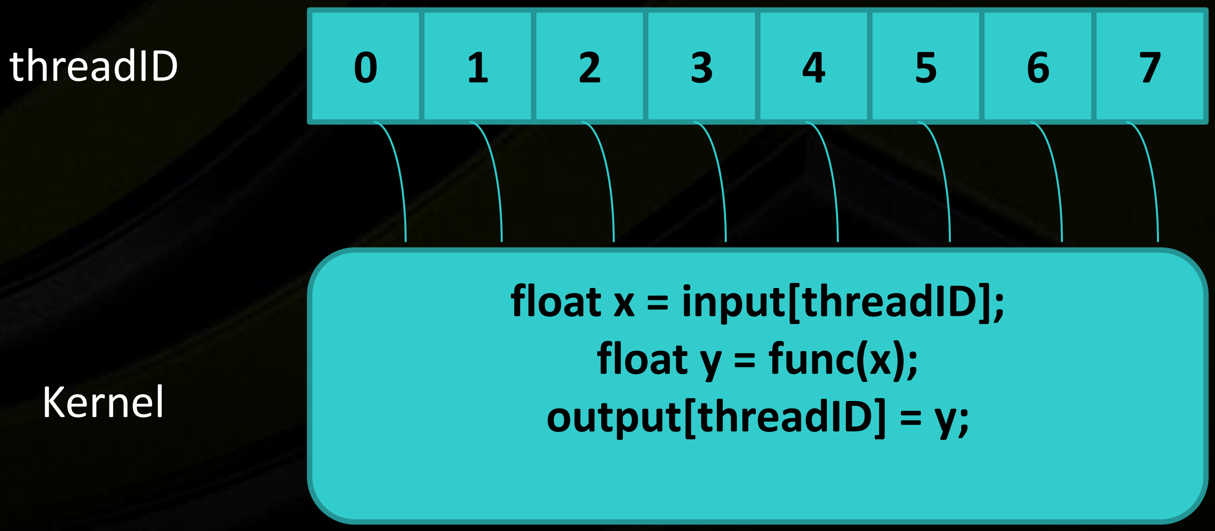
Blocks (1,1)

Blocks (2,1)

Thread (0,0)	Thread (1,0)	Thread (2,0)	Thread (3,0)
Thread (0,1)	Thread (1,1)	Thread (2,1)	Thread (3,1)
Thread (0,2)	Thread (1,2)	Thread (2,2)	Thread (3,2)
Thread (0,3)	Thread (1,3)	Thread (2,3)	Thread (3,3)

Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



```
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

Thread Cooperation

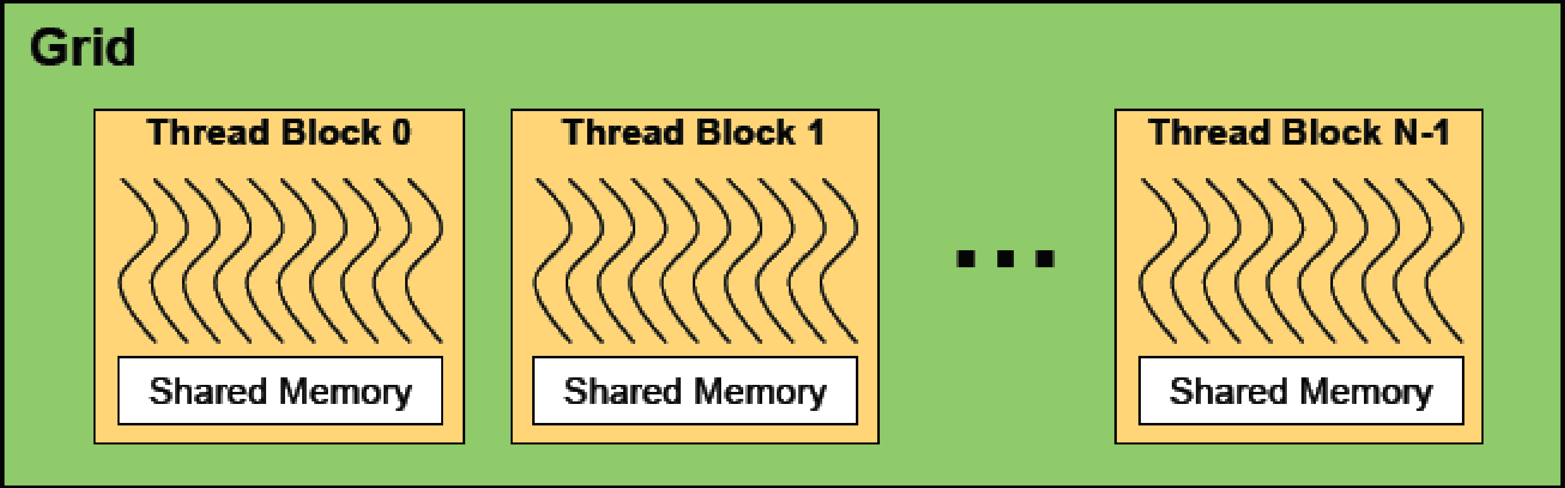
- The Missing Piece: threads may need to cooperate

- Thread cooperation is valuable
  - Share results to avoid redundant computation
  - Share memory accesses
    - Bandwidth reduction

- Cooperation between a monolithic array of threads is not scalable
  - Cooperation within smaller batches of threads is scalable

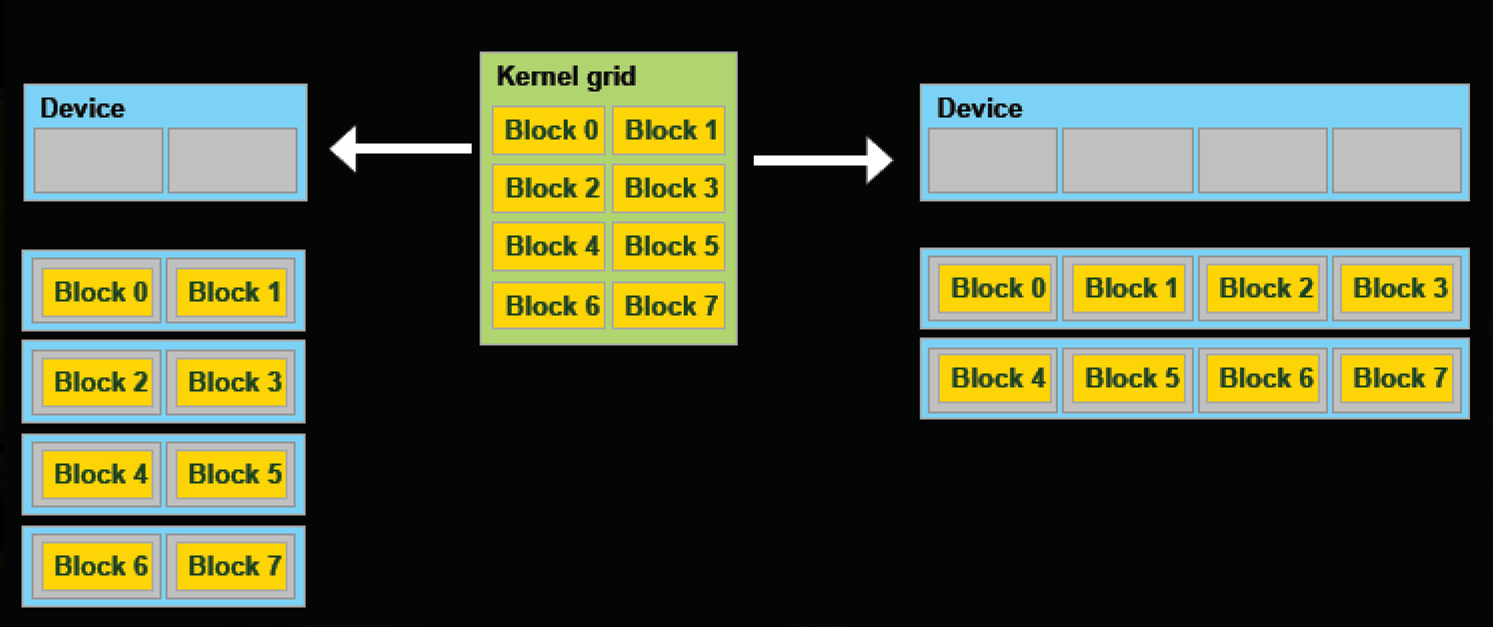
Thread Batching

- Kernel launches a grid of thread blocks
  - Threads within a block cooperate via shared memory
  - Threads within a block can synchronize
  - Threads in different blocks cannot cooperate
- Allows programs to *transparently scale to different* GPUs



Transparent Scalability

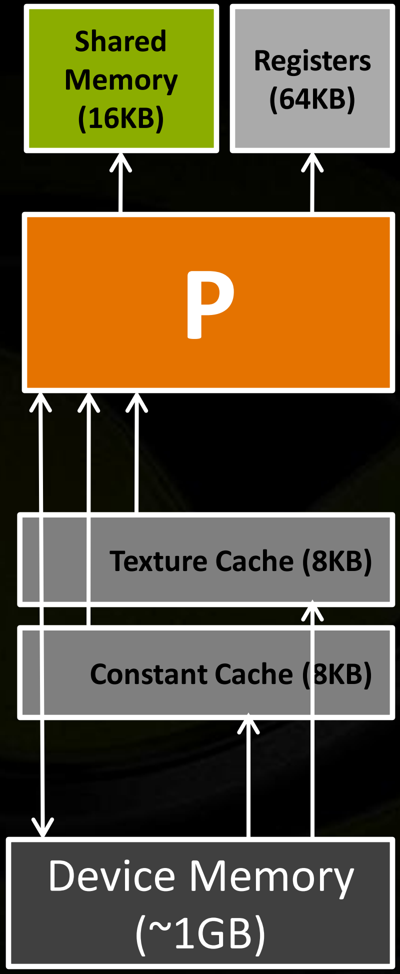
- Hardware is free to schedule thread blocks on any processor
  - A kernel scales across parallel multiprocessors



Kernel Memory Access

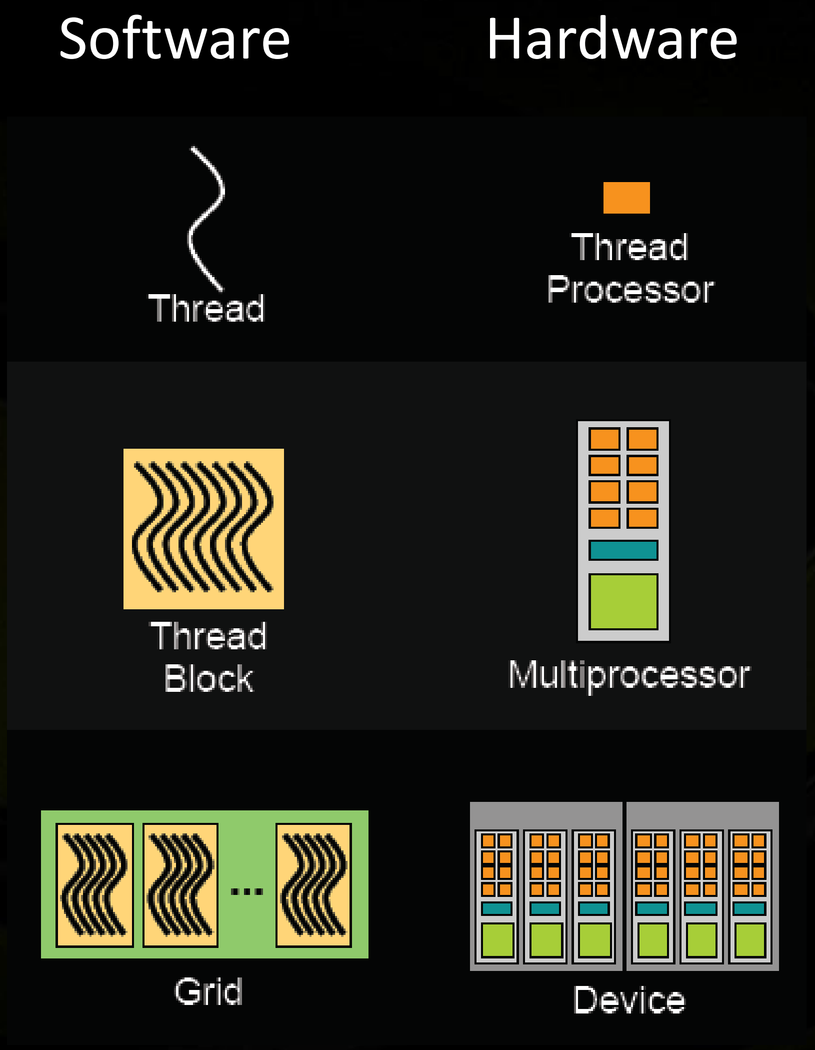
- Per-thread : Registers (fast)
- Per-block : Shared Memory (fast, on-chip)
- Per-device : Global Memory (Uncached, Off-chip, large persistent across kernel launches)

CUDA Memory Model (H/W)



- P has access to Registers (private)
- P has access to Shared-Memory (common to 8 Ps, share and have fun)
- Caches (texture and constant) are not user-managed, true caches
- Device memory is for all Ps of all SMs ! Truly Chaotic !

Execution Model



- Threads are executed by thread processors
- Thread blocks are executed on multiprocessors
  - Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)
- A kernel is launched as a grid of thread blocks
  - Only one kernel can execute on a device at one time

CUDA API

- An extension to the ANSI C programming Language
  - No interference of a Graphics API (OpenGL/DirectX)
  - Good learning curve

- Language Extensions in form of
  - Function type qualifiers (variety of functions)
  - Variable type qualifiers (types of variables)
  - Execution Configuration (parameters to kernel)
  - Built-In variables (block and thread Ids)

Function type qualifiers

- __device__ (internal functions needed by main device function)
  - Executed on the *device*
  - Callable from *device*
- __global__ (main Kernel function)
  - Executed on the *device*
  - Callable *only* from *host*
- __host__
  - Executed on the *host*
  - Callable only from *host*

- For functions executed on the device
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

Variable type qualifiers

- __device__
  - Use with one of the options mentioned below
- __constant__
  - Resides in constant memory space
  - Has the lifetime of an application
  - Accessible from all threads and host
- __shared__
  - Resides in Shared Memory Space of thread block
  - Only accessible from threads within the block
  - Life time of a block

Built-In Variables

- gridDim
  - 3 Dimensional variable holding the dimensions of a *grid*
- blockIdx
  - An int3 type of variable holding the block index within the *grid*
- blockDim
  - 3 Dimensional variable holding the dimensions of a *block*
- threadIdx
  - An int3 type of variable holding the *thread* index within the *block*
- Can not assign values to them nor can you get the address of the above variables

Sample Code Snippets

- DeclSpecs
  - __global__ void convolve (float *image)
  - __shared__ float region[M]
- Keywords
  - region[threadIdx] = image[i]
  - __syncthreads()
- Memory management and Kernel Launch
  - void *myImage = cudaMalloc(bytes)
  - Convolve<<<100,100>>> (myImage);

Increment Array Example

```
void inc_cpu (int *a, int N) {

    int idx;

    for (idx = 0; idx<N; idx++)
        a[idx] = a[idx] + 1;
}

void main() {
    …
    inc_cpu(a, N);
    …
}
```

```
__global__ void inc_cpu (int *a_d, int N) {

    int idx = blockIdx.x * blockDim.x
                  + threadIdx.x;
    if ( idx < N )
        a_d[idx] = a_d[idx] + 1;
}
void main(){
    …
    dim3 dimBlock (num_threads);
    dim3 dimGrid(ceil(N/(float)num_threads));
    inc_gpu<<<dimGrid, dimBlock>>>(a_d, N);
    …
}
```

Q/A

Schedule

0:00 -- 0:10 : Introduction to the Tutorial, Theme, Speakers.

0:10 -- 0:30 : Basic Concepts -- CPU Architectures, GPUs -- evolution, comparison to earlier models of parallel computing

0:30 -- 0:55 : GPU Architectures in Detail -- NVidia architecture, Intel Larrabee architectural features

0:55 -- 1:30 : GPU Programming models with short examples, CUDA

BREAK

1:50 -- 2:15 : Case studies of regular applications on the GPU

2:15 -- 2:45 : Case studies of irregular applications on the GPU

2:45 -- 3:20 : GPU Analytical Models

Design Space Optimization, Performance Prediction

3:20 -- 3:30 : Concluding remarks, discussion

<<< Data Primitives >>>

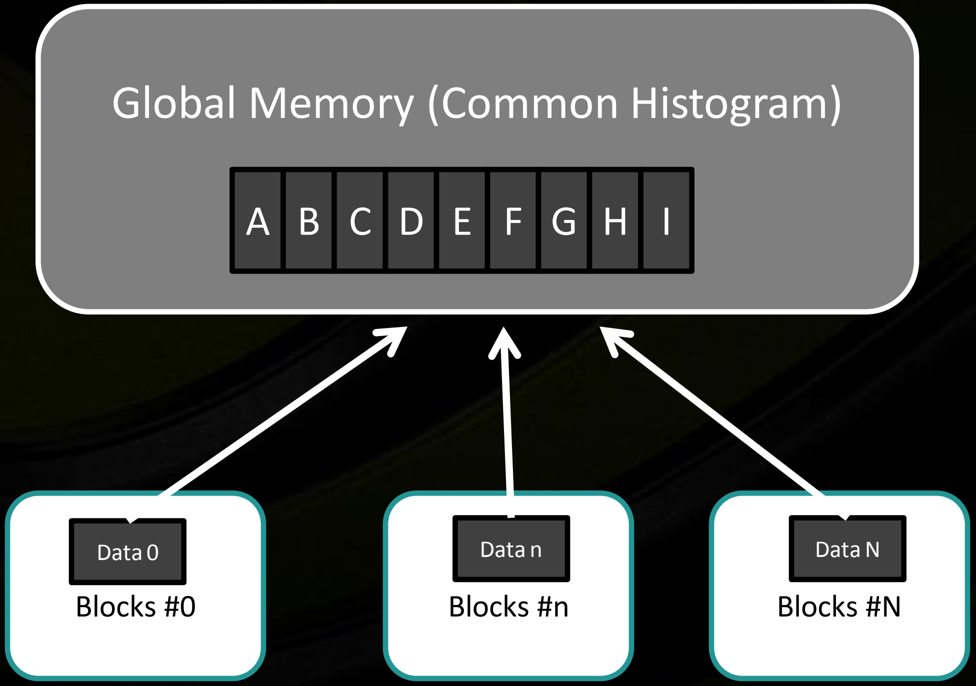
Suryakant Patidar
spatidar@nvidia.com

Histogram Computation

- Counting data based on a given property
  - Counting frequency of words in a book
- Large number of input data
  - Easy to parallelize
- Requires to access common memory areas
  - Memory operations on common area
  - Language provided Atomic Operations
- Widely used

Global Memory Histogram

- Using Atomic operations on Global Memory
- '#Bins' sized array used in global memory to hold the histogram data



```
for each thread in parallel
      for each element 'x' assigned
       to the thread, sequentially
{
       bin = category[x];
       atomicInc(&globalHist[bin]);
}
```

Global Memory Histogram (2)

- Low Memory requirement – one copy of histogram

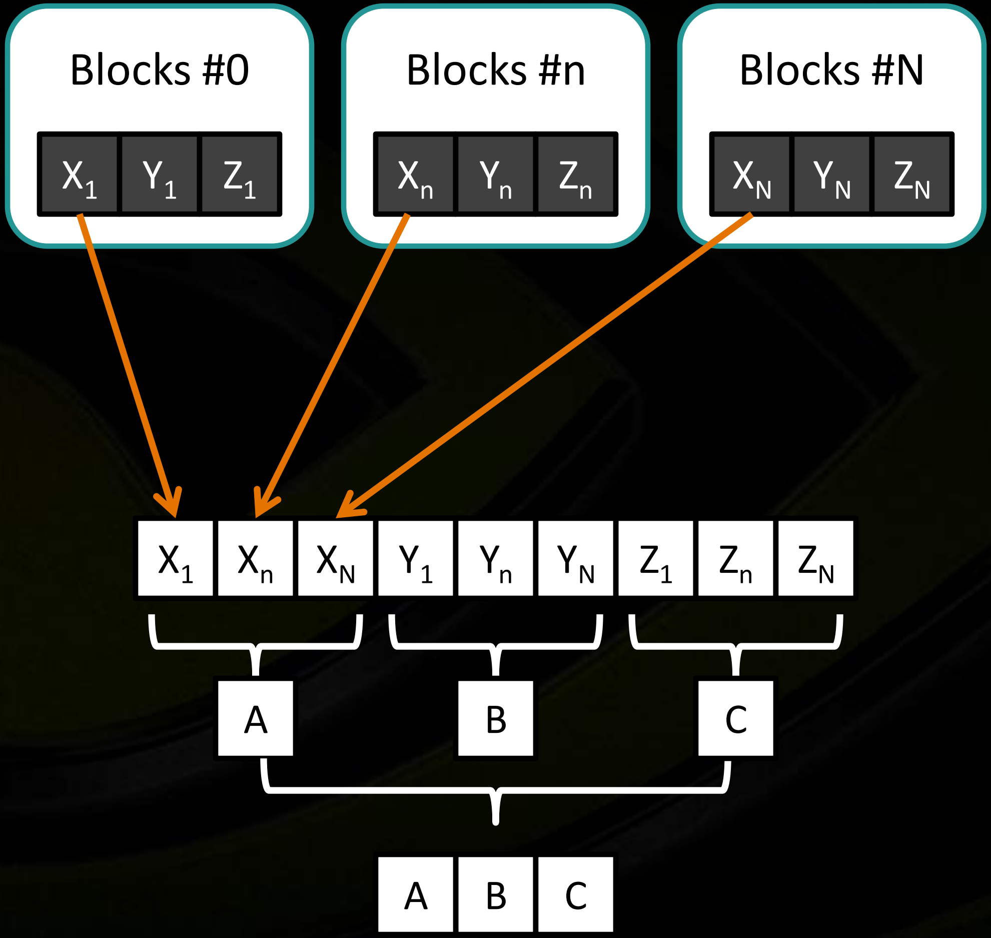
- Number of Clashes ἀ Number of Active Threads
  - Highly data dependent, low number of bins tend to perform really bad

- Global Memory is high-latency, ~500cc I/O

Shared Memory Histograms

- A copy of the histogram for each Block
- Each Block counting its own data
- Once all done, we add the sub-histograms to get the final histogram as needed

Shared Memory Histograms (2)



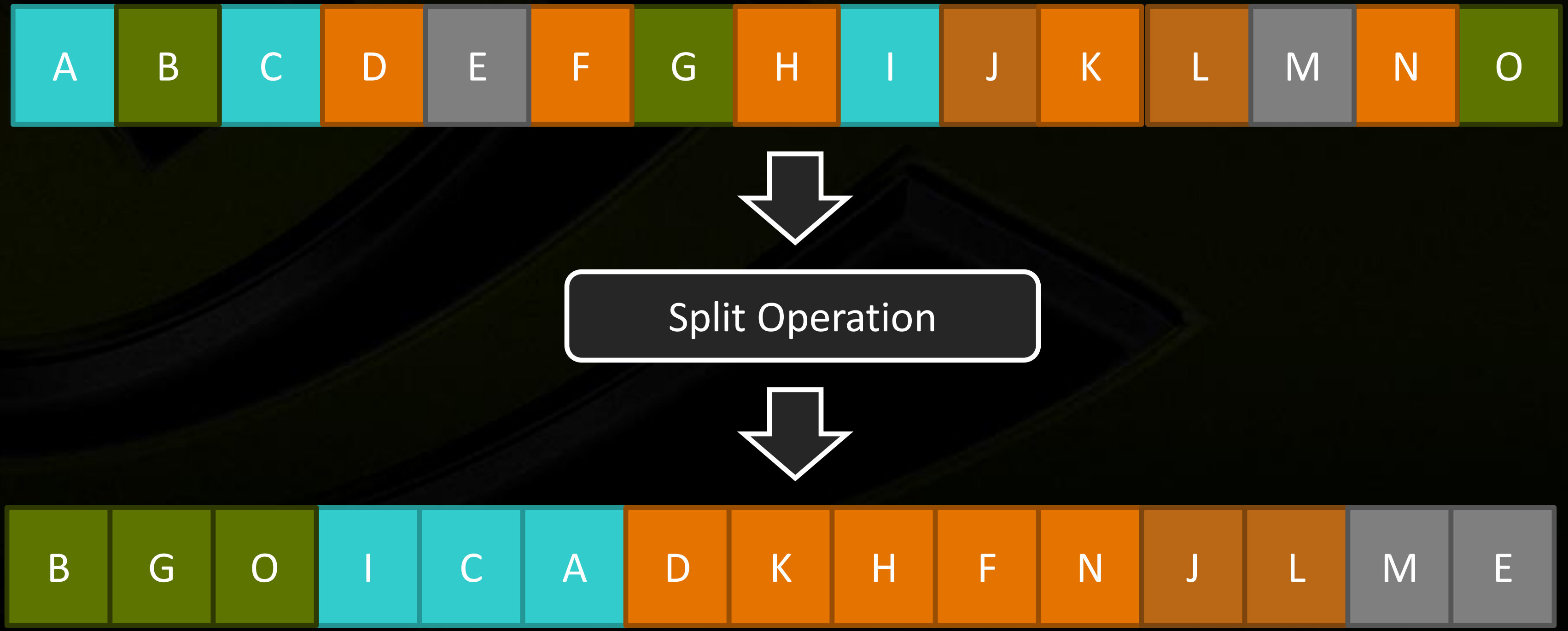
- Local Shared Memory used to store the sub-histograms
- Fast to update and less number of conflicts (N times less conflicts)
- Experimentally found to be faster

Split Operation

Split can be defined as performing ::

append(x,List[category(x)])

for each x, List holds elements of same category together



A B C D E F G H I J K L M N O

Split Operation

B G O I C A D K H F N J L M E

Split Sequential Algorithm

I. Count the number of elements falling into each bin

```
for each element x of list L do
    histogram[category(x)]++    [Requires Atomic]
```

II. Find starting index for each bin (Prefix Sum : Scan Primitive)

```
for each category ‘m’ do
    startIndex[m] = startIndex[m – 1]+histogram[m-1]
```

III. Assign each element to the output

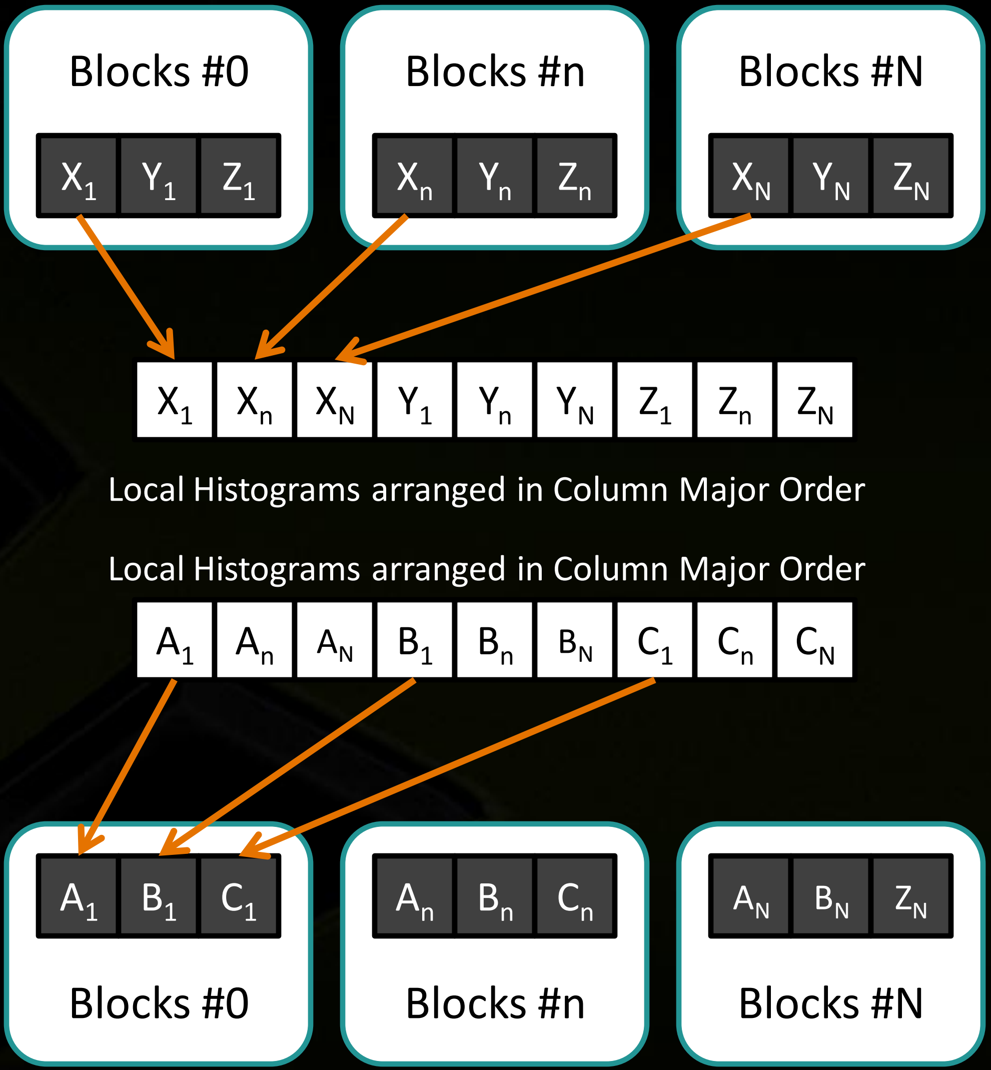
```
for each element x of list L do [Initialize localIndex[x]=0]
    itemIndex = localIndex[category(x)]++ [Requires Atomic]
    globalIndex = startIndex[category(x)]
    outArray[globalIndex+itemIndex] = x
```

Split using Shared Atomic

- Shared Atomic Operations used to build Block-level histograms

- Parallel Prefix Sum used to compute starting index

- Split is performed by each block for same set of elements used in Step 1



Blocks #0 | X_1 Y_1 Z_1

Blocks #n | X_n Y_n Z_n

Blocks #N | X_N Y_N Z_N

X_1 X_n X_N Y_1 Y_n Y_N Z_1 Z_n Z_N

Local Histograms arranged in Column Major Order

Local Histograms arranged in Column Major Order

A_1 A_n A_N B_1 B_n B_N C_1 C_n C_N

A_1 B_1 C_1 | Blocks #0

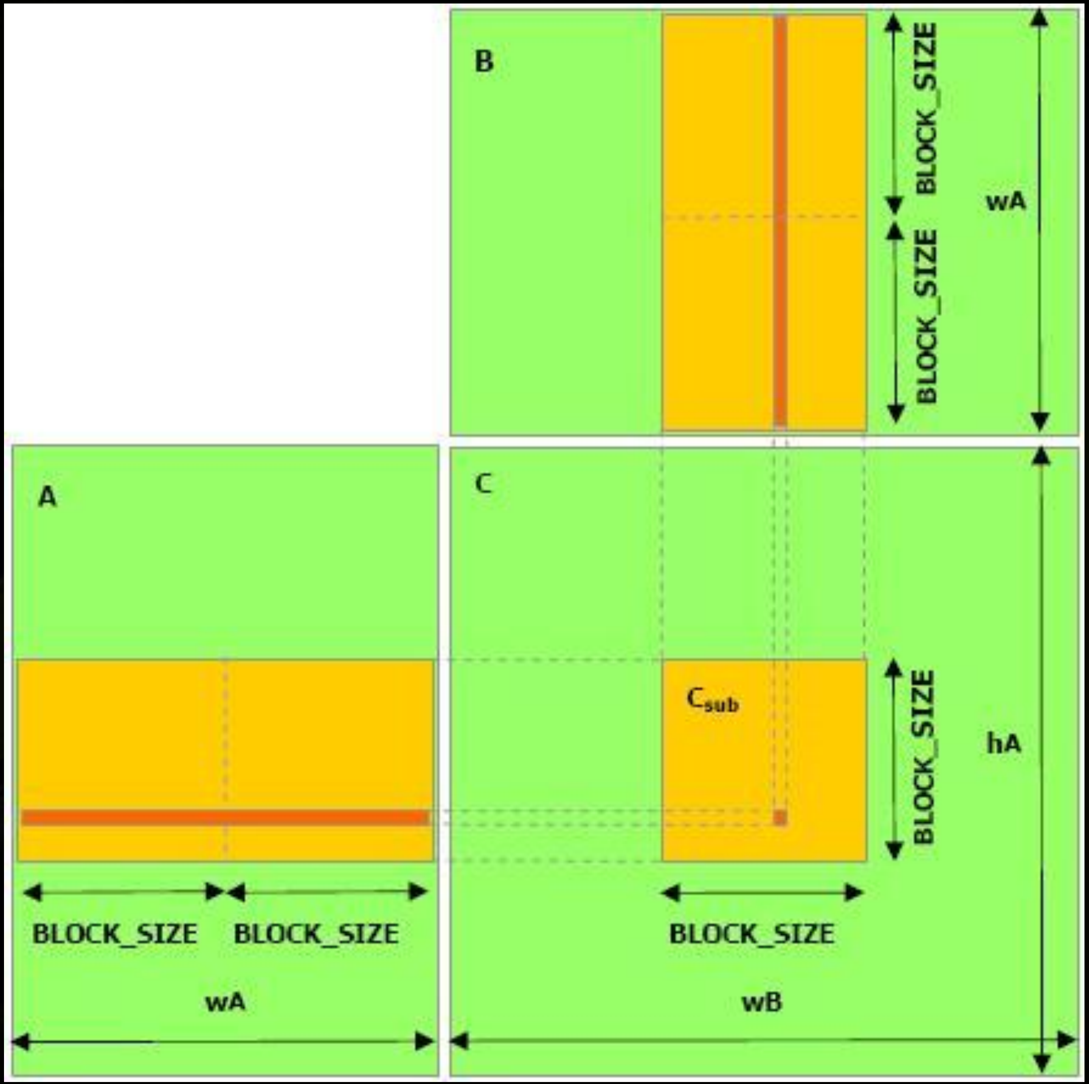
A_n B_n C_n | Blocks #n

A_N B_N Z_N | Blocks #N

Matrix Multiplication

- Each thread block is responsible for computing one square sub-matrix Csub of C

- Each thread within the block is responsible for computing one element of Csub



Host Implementation

```
// Matrix multiplication on the (CPU)
    void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
    {
        for (int i = 0; i < A.height; ++i)
        for (int j = 0; j < B.width; ++j) {
                float sum = 0;
                for (int k = 0; k < A.width; ++k) {
                        float a = A.elements[i * A.width + k];
                        float b = B.elements[k * B.width + j];
                        sum += a * b;
                }
                C.elements[i * B.width + j] = sum;
          }
    }
```

Device Implementation

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(Matrix A, Matrix B, Matrix C)
{
        // 2D Thread ID
        int tx = threadIdx.x;
        int ty = threadIdx.y;
        // cvalue is used to store the element of the matrix that is computed by the thread
        float Cvalue = 0;
        for (int k = 0; k < A.width; ++k)
        {
                float Aelement = A.elements[ty * A.pitch + k];
                float Belement = B.elements[k * B.pitch + tx];
                Cvalue += Aelement * Belement;
        }
        // Write the matrix to device memory;  each thread writes one element
        C.elements[ty * C.pitch + tx] = Cvalue;
}
```

Q/A

Irregular Algorithms on the GPU

P. J. Narayanan
Centre for Visual Information Technology
International Institute of Information Technology
Hyderabad

PPoPP Tutorial on GPUs. Jan 10, 2010

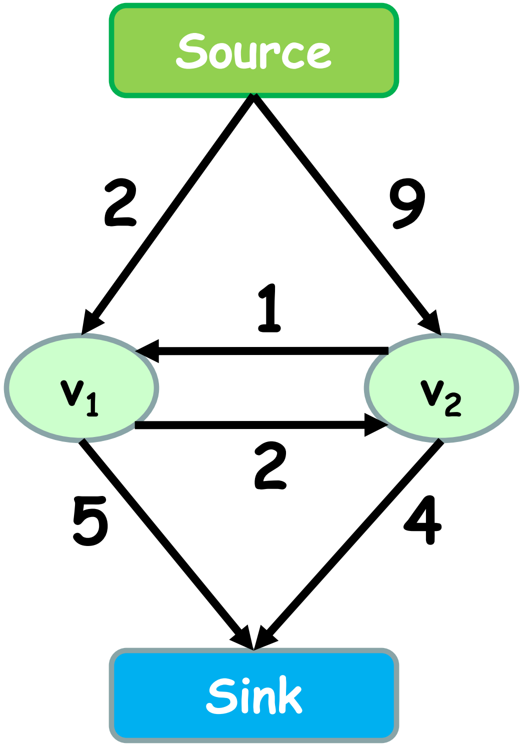
Graph Cuts for Computer Vision on the GPU

Work done with Vibhav Vineet (CVGPU08 Workshop)

Graph Cuts in Computer Vision

- Several optimization problems have been mapped to *maxflow* on a graph built from the pixels with a special *s* node and *t* node.
  - Segmentation: Assign binary labels to pixels
    - Pixels connected to *s* after cut is foreground and the rest are background.
  - Stereo matching: Assign integer labels to pixels
    - Disparity is the standard label.
    - Framework works for many problems
- Many sequential algorithms exist. Goldberg-Tarjan (push-relabel) and Edmonds-Karp (augmenting path based) are popular.
  - Former is more parallelizable

The st-Mincut Problem



Graph (V, E, C)

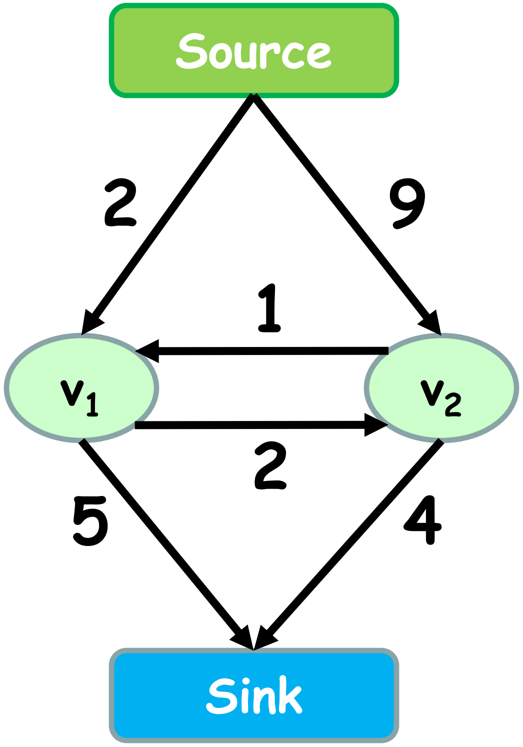
Vertices V = $\{v_1, v_2 \ldots v_n\}$

Edges E = $\{(v_1, v_2) \ldots.\}$

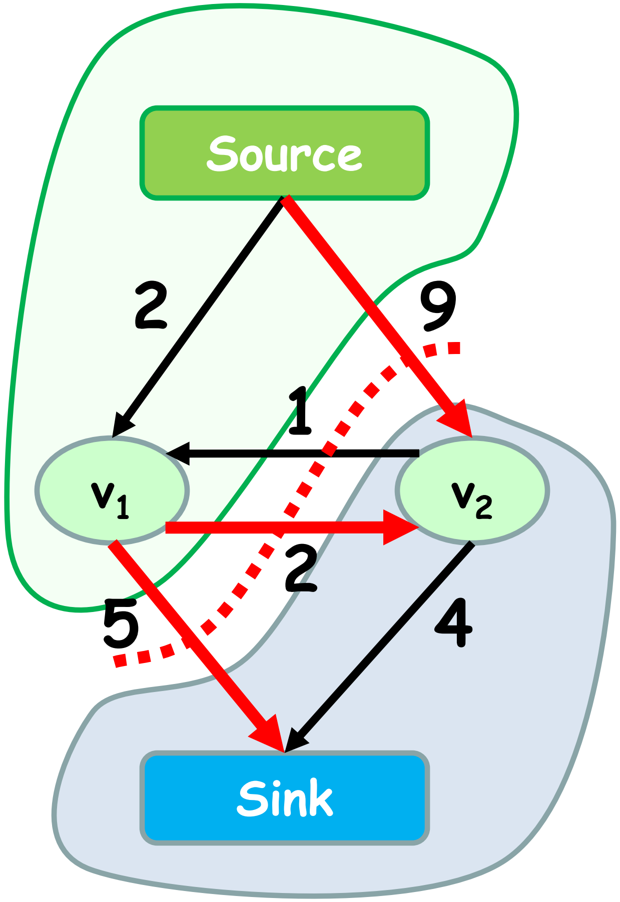
Costs C = $\{c_{(1, 2)} \ldots.\}$

The st-Mincut Problem

What is an st-cut?



The st-Mincut Problem



5 + 2 + 9 = 16

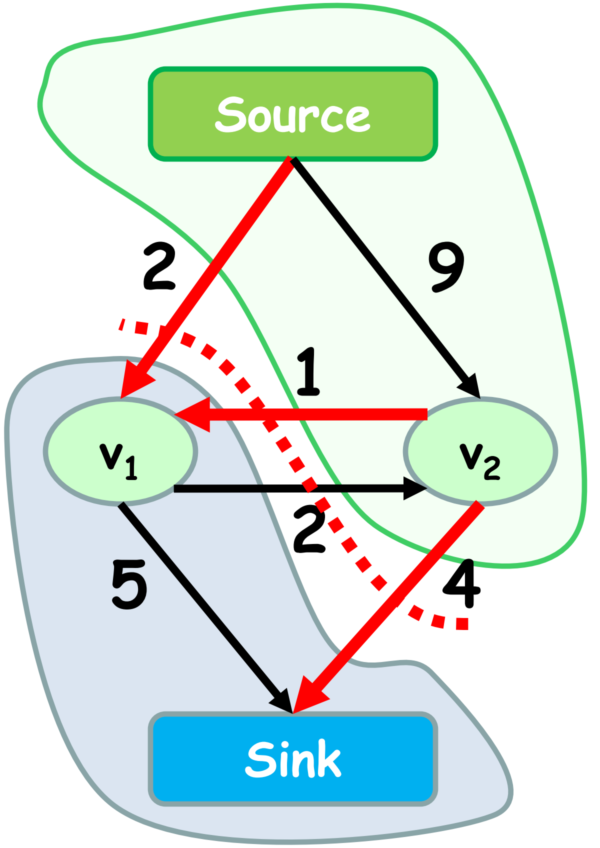
What is an st-cut?

An st-cut (**S**,**T**) divides the nodes between source and sink.

What is the cost of a st-cut?

Sum of cost of all edges going from S to T

The st-Mincut Problem



2 + 1 + 4 = 7

What is an st-cut?

An st-cut (**S**,**T**) divides the nodes between source and sink.

What is the cost of a st-cut?

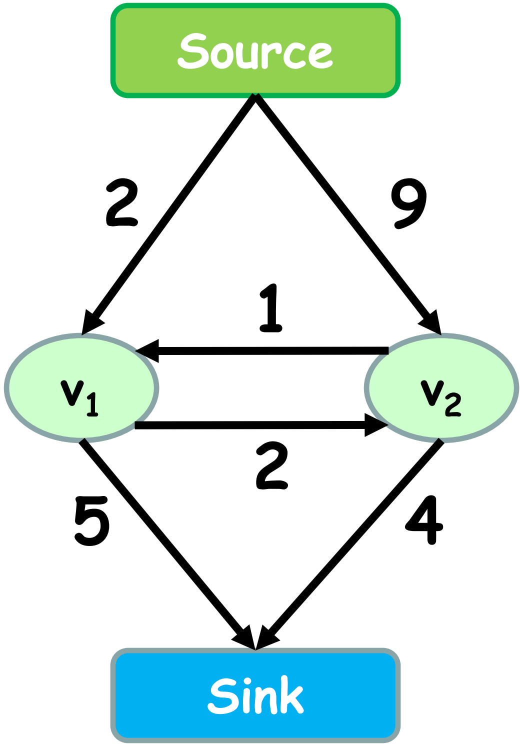
Sum of cost of all edges going from S to T

What is the st-mincut?

st-cut with the minimum cost

Maxflow Algorithms

Flow = 0

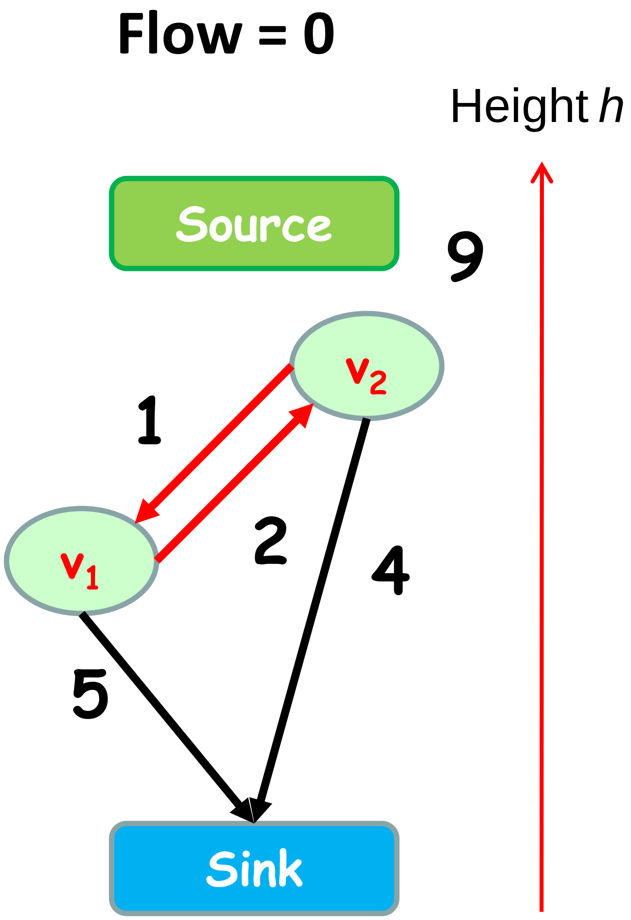


Goldberg's generic Push-Relabel Algorithm

1. Intialize-Preflow(G,s)
2. Perform an applicable push or relabel operation
3. Repeat untill there exists no applicable push or relabel operation

Algorithms assume non-negative capacity

Maxflow Algorithms

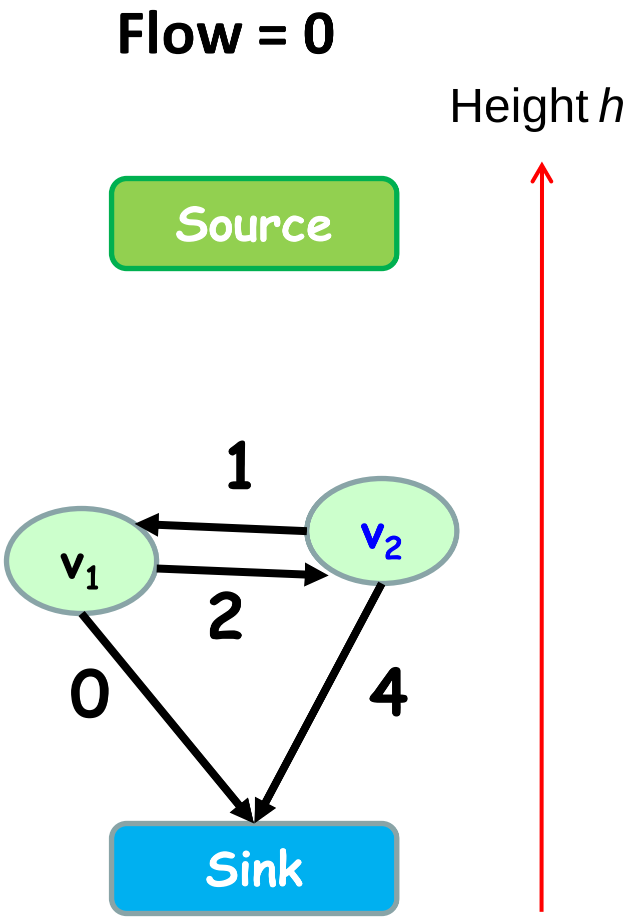


Push Operation

1. V_2 is overflowing
2. Height $h(V_2) == h(V_1) + 1$
3. Push as much unit of flows from V_2 to V_1

Algorithms assume non-negative capacity

Maxflow Algorithms

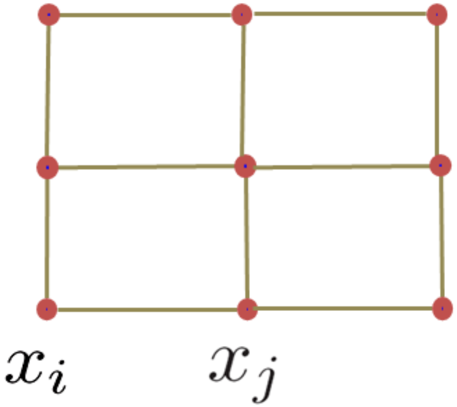


Relabel Operation

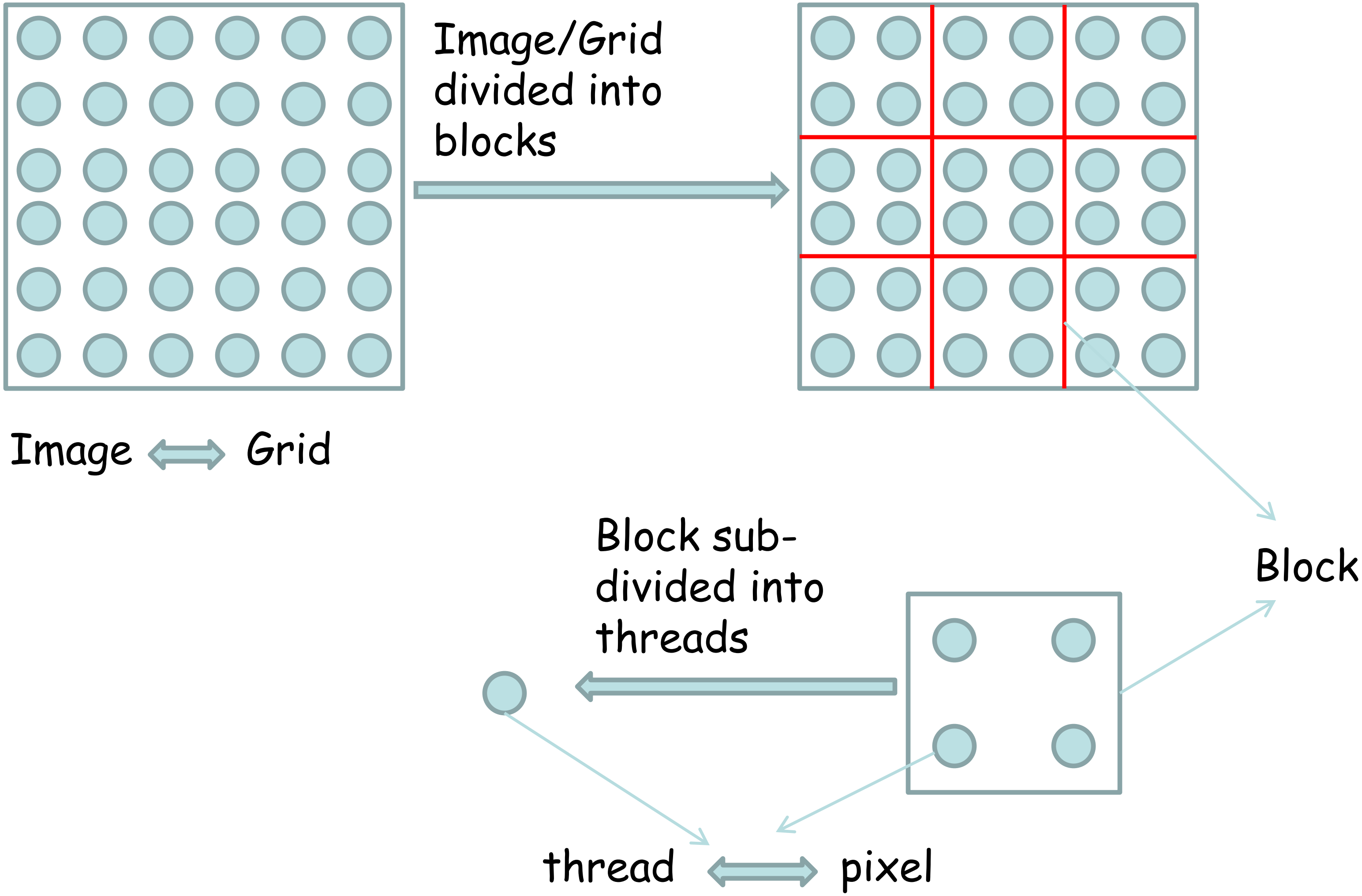
1. V_2 is overflowing and is in residual graph
2. Height $h(V_2) \le h(V_1)$
3. Increase the height of V_2

GraphCuts on Images

- Specialized algorithms for vision problems
  - – Grid graphs
  - – Low connectivity; typically limited to 4, 8 or 27



Mapping Image On CUDA

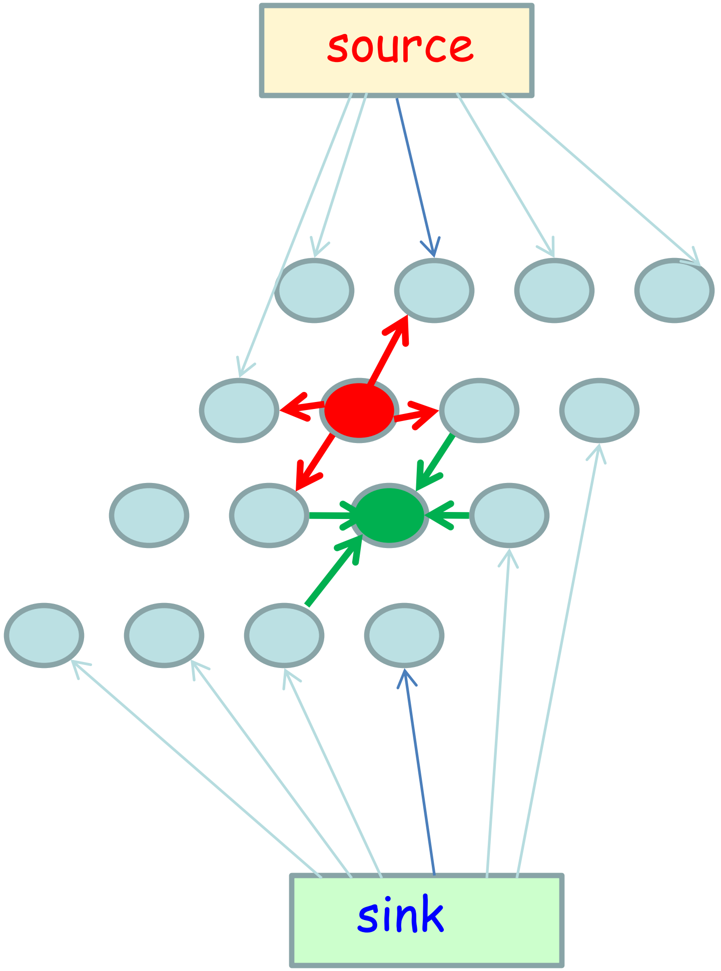

Image/Grid divided into blocks
Image ⟺ Grid
Block
Block sub-divided into threads
thread ⟺ pixel

Push Relabel Algorithm on CUDA

1. Push is an local operation with each node sending flows to its neighbors
2. Relabel is also a local operation
3. Problems faced:
   1. RAW problems: (Read after write)
   2. Synchronization is limited to the threads of a block.

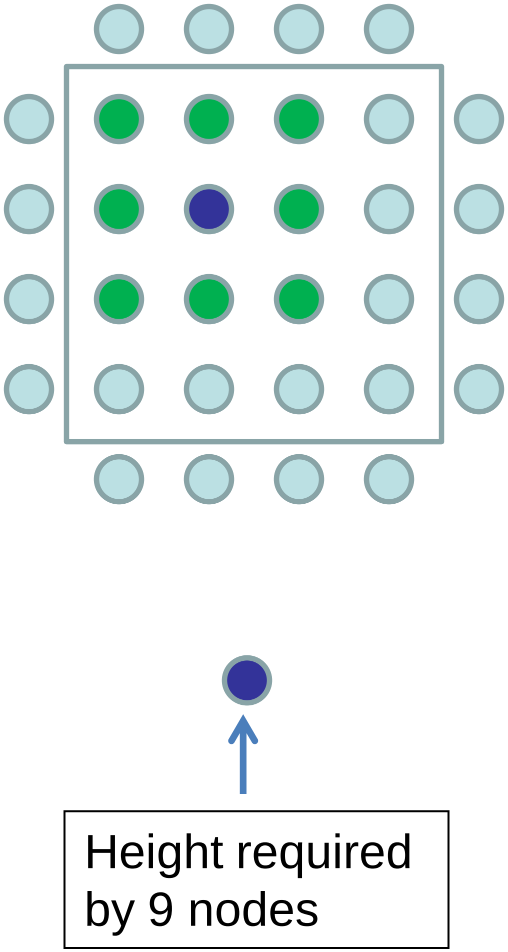
Push Relabel Algorithm on CUDA

1. Push operation is divided into two phases: Push Phase and Pull Phase
2. Relabel is also local operation
3. Naïve Solution: Three Kernels
   1. Push Kernel
   2. Pull Kernel
   3. Relabel Kernel



Push Kernel (node u)

1. Load h(u) from the global memory to shared memory of the block.
2. Synchronize threads to ensure completion of load
3. Push flow to the eligible nodes without violating the preflow conditions.
4. Update the residual capacities of edges(u,v) in the residual graphs.
5. Store the flow pushed to each edge in a special global memory array F.



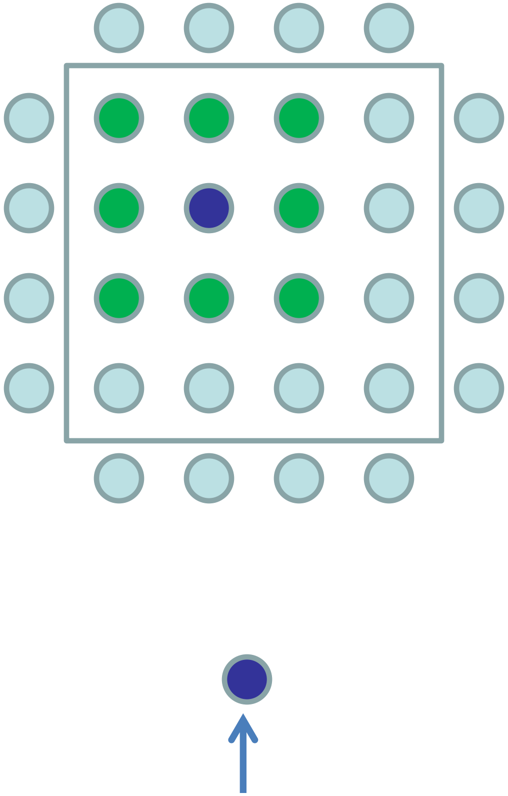
Height required by 9 nodes

Pull Kernel (node u)

1. Read the flows pushed to u from the F array of its neighbors.
2. Compute the final excess flow by aggregating all incoming flows. Store it as the e(u) value in global memory.

Relabel Kernel (node u)

1. Load h(u) from the global memory to the shared memory.
2. Synchronize to ensure the completion of load of heights.
3. Compute the minimum heights of neighbors of node u.
4. Write the new height to global memory location h(u).



Height required by 9 nodes

Push and Relabel Kernels (Shared Memory)

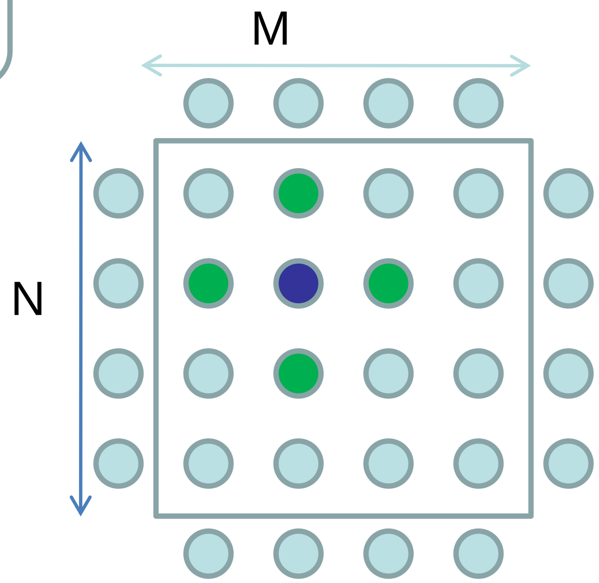
1. Load h(u) from the global memory to shared memory of the block.

Shared Memory Used:

- Each Block has MxN threads.

Internal Nodes:

Each Internal Node (🔵) requires heights of 4 other nodes (🟢) from the same block.



Push and Relabel Kernels (Shared Memory)

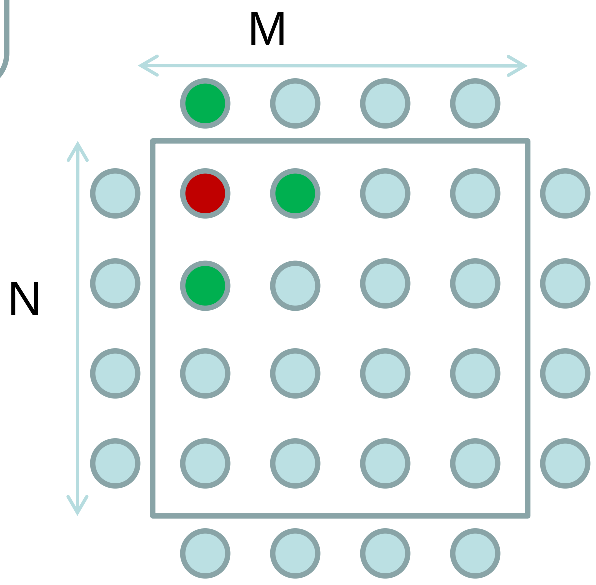
1. Load h(u) from the global memory to shared memory of the block.

Shared Memory Used:

- Each Block has MxN threads.

Border Nodes:

Each Border Node(🔴) requires heights of other nodes(🟢) from the different blocks.

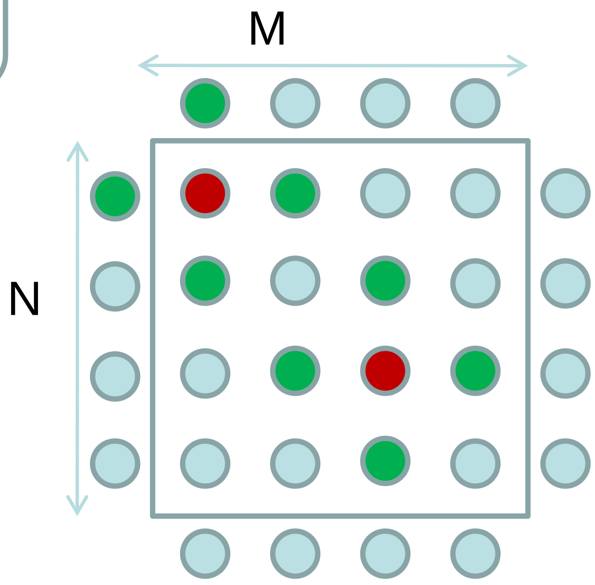


Push and Relabel Kernels (Shared Memory)

1. Load h(u) from the global memory to shared memory of the block.

Shared Memory Used:

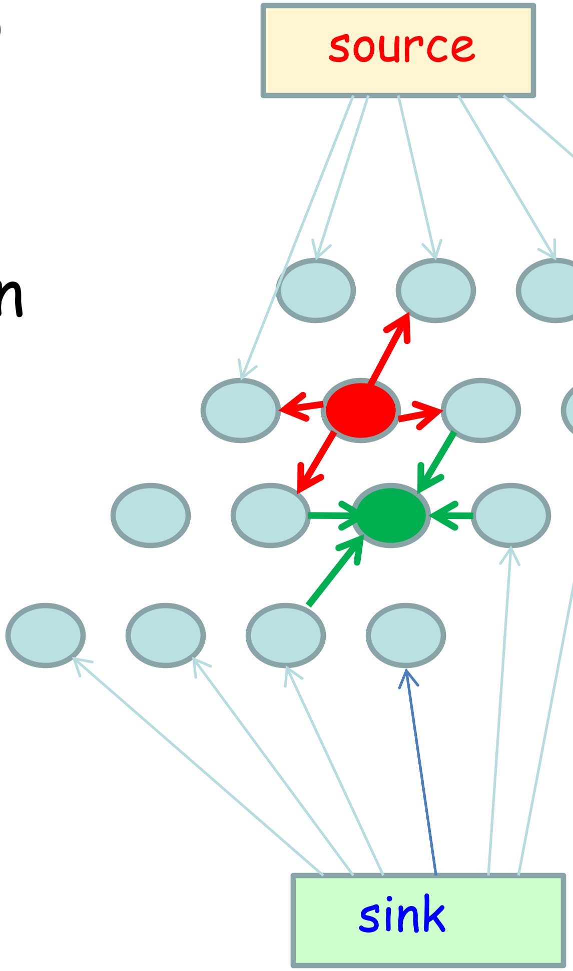
- Each Block has MxN threads.
- Total Shared Memory Used:
  - (M+2)x(N+2)x(sizeof(element))



CUDA Block

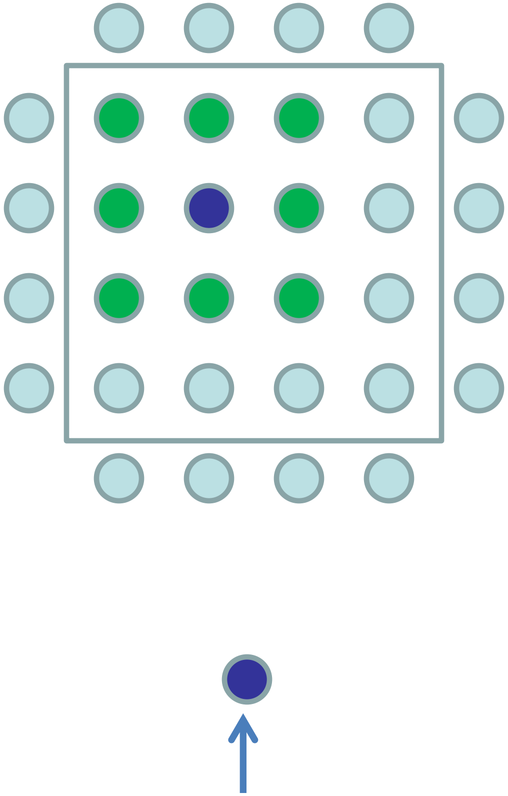
Push Relabel Algorithm

1. Push operation is divided into two phases: Push Phase and Pull Phase
2. Relabel is also local operation
3. Different Solution : Two kernels
   1. Push Kernel
   2. Pull + Relabel Kernel



Pull + Relabel Kernel (node u)

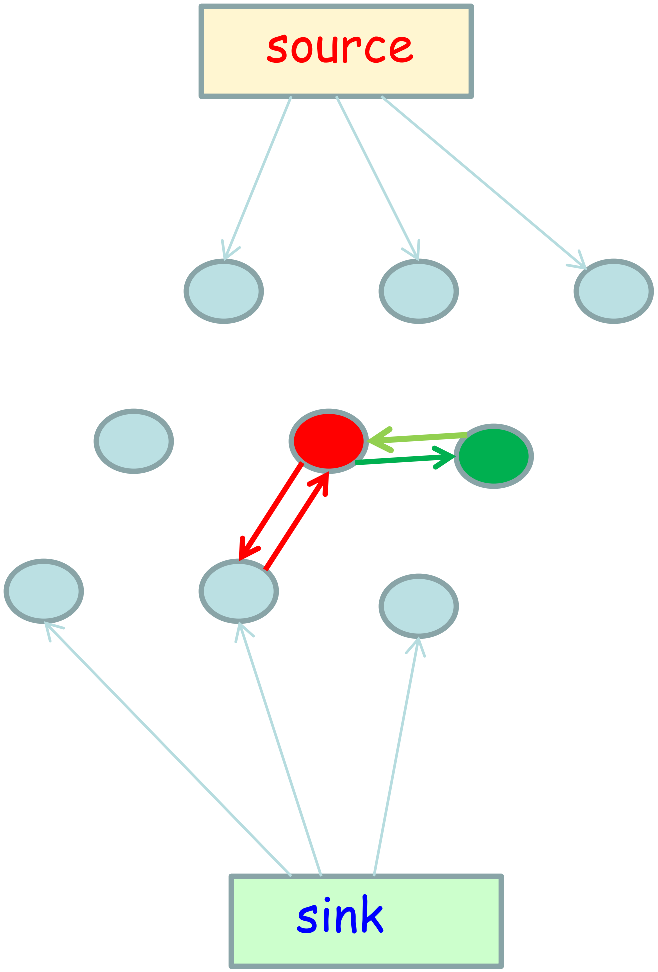
1. Load h(u) from the global memory to the shared memory.
2. Synchronize threads to ensure the completion of load.
3. Update the excess flow e(u) and residual capacities of edges (u,v) in the residual graph with the flows from the global memory array F.
4. Synchronize to ensure completion of updation of edge-weights and excess flow.
5. Compute the minimum heights of neighbors of node u.
6. Write the new height to global memory location h(u).



Height required by 9 nodes

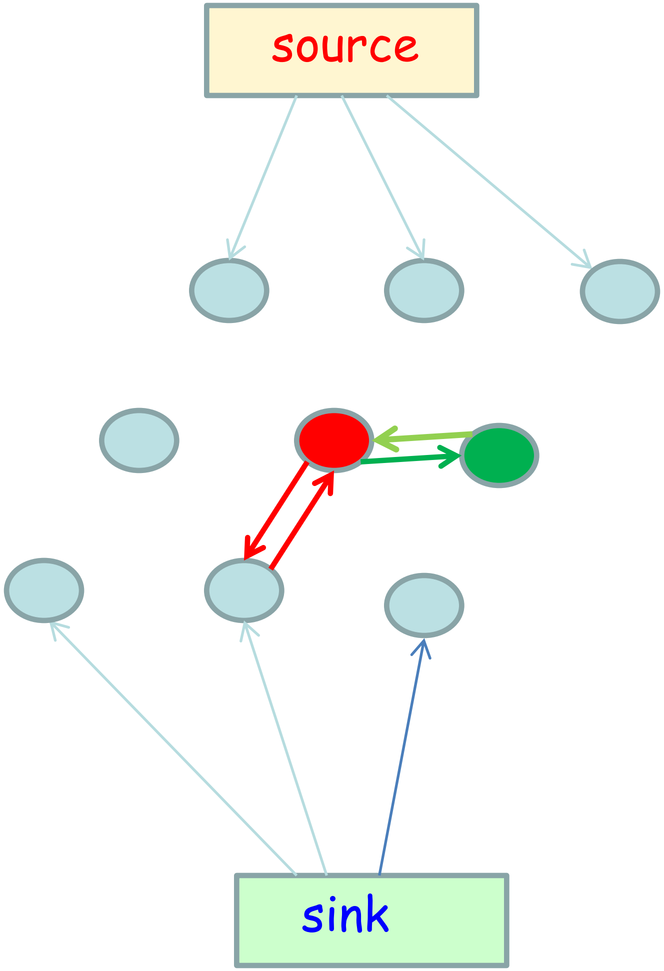
On Hardware with Atomic Capabilities

1. Push and Pull operations can performed without any RAW problem.
2. Relabel is also local operation
3. Third Solution on Hardware with Atomic Capabilities: Two kernels
   1. Push + Pull Kernel
   2. Relabel Kernel



Push + Pull Kernel (node u)

1. Load h(u) from the global memory to the shared memory.
2. Synchronize threads to ensure the completion of load.
3. Push flows to eligible neighbors **atomically** without violating the preflow condition.
4. Update the edge-weights of (u,v) and (v,u) **atomically** in the residual graph.
5. Update the excess flow of e(u) and e(v) atomically in the residual graph.



Results

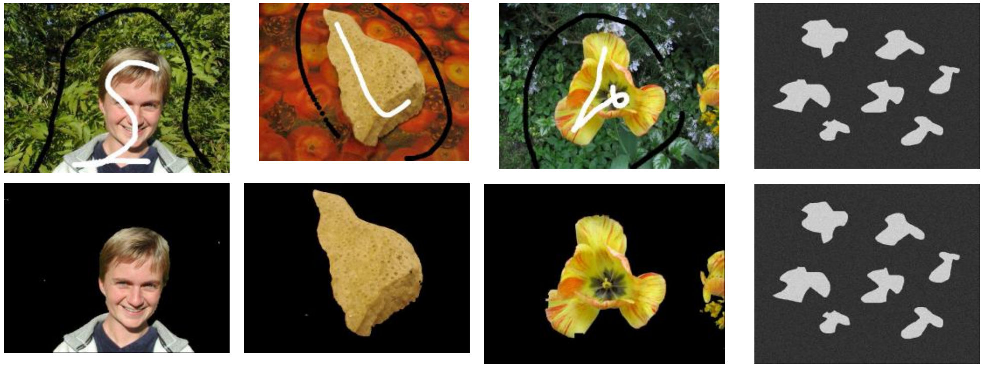


Image	Size	Time (CPU) (millisecond)	Time (Non- Atomic)	Time (Atomic)	Time (Stochastic)
Sponge	640x480	142	28	16	11
Flower	608x456	188	33	26	16
Person	608x456	140	31	27	20
Synthetic	1Kx1K	655	19	10	7

Vibhav Vineet and P J Narayanan. “CudaCuts”. IEEE CVPR Workshop on Computer Vision on the GPUs. Alaska, June 2008.

Fast and Scalable List Ranking on the GPU

M. Suhail Rehman, Kishore Kothapalli, P. J. Narayanan

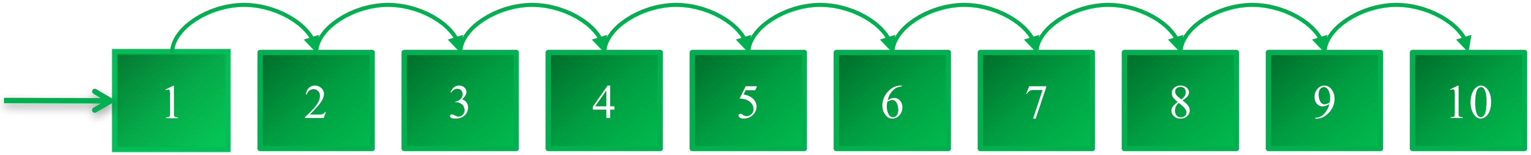
Center for Security, Theory, and Algorithmic Research
Center for Visual Information Technology
International Institute of Information Technology, Hyderabad

IIIT Hyderabad

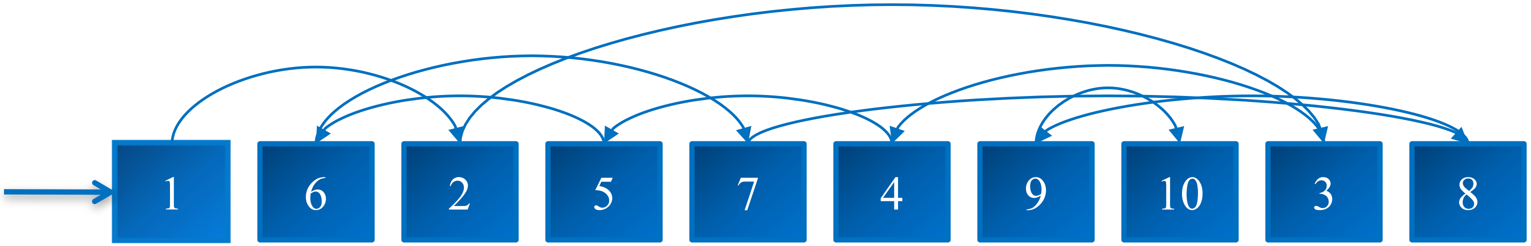
The List Ranking Problem

- Given a list of N elements, rank each element based on the distance of that element with the end of the list.
- A sequential algorithm is trivial and runs on O(n)
- Many parallel algorithms exist for various models.

Types of Linked Lists



Ordered List



Unordered List

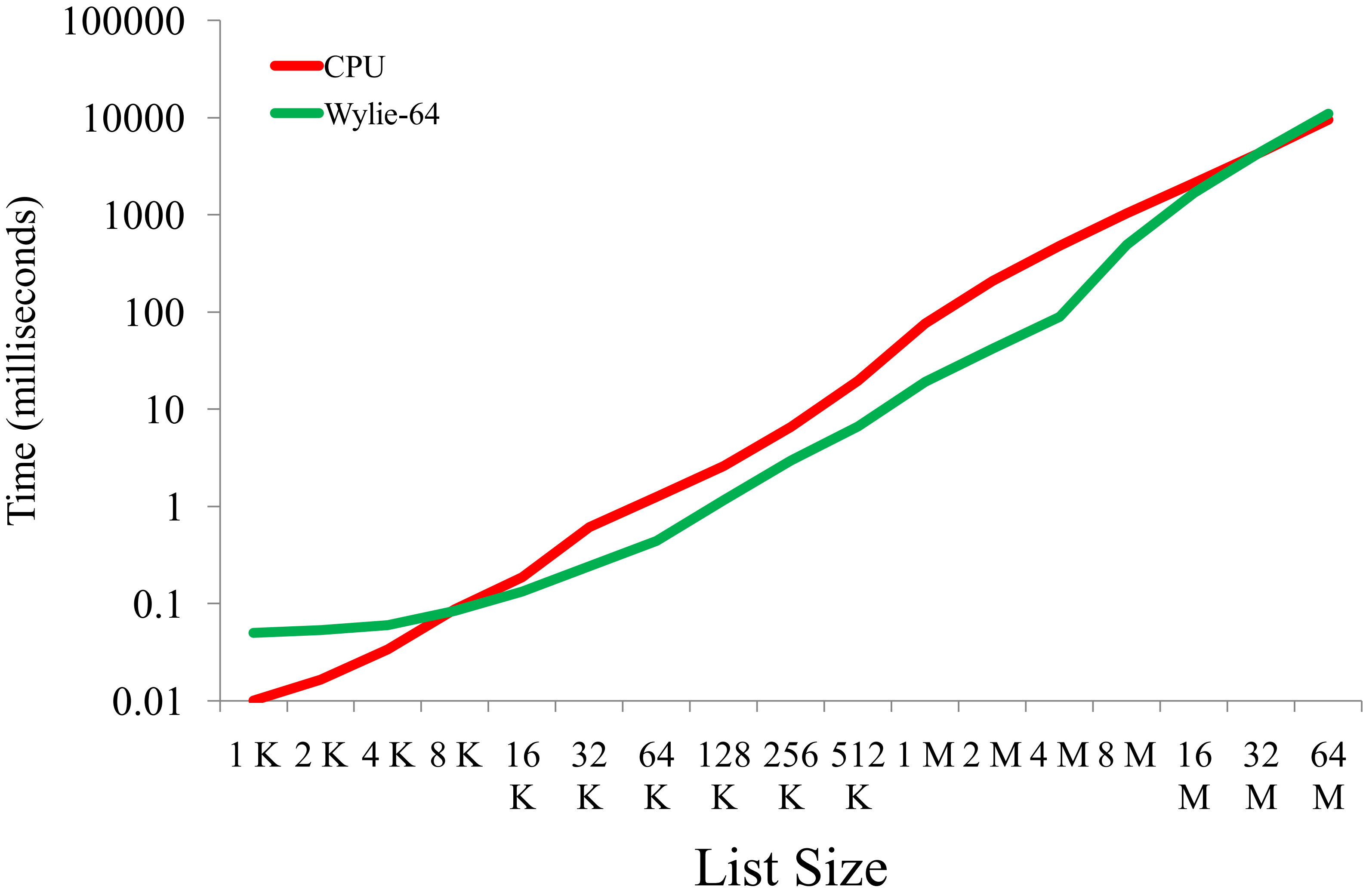
Baseline Implementation

- Wyllie’s Algorithm uses Pointer Jumping
- Initialize Ranks to 1
- For each element in Array, set it’s rank to rank + rank of Successor
- Reset the Successor value to the successor of it’s successor (effectively jumping over and contracting the list)

GPU-Specific Optimizations

- Load the data elements when needed
- Bitwise operations to pack and unpack data
- Block-level thread synchronization to force threads to write in a coalesced manner
- Current best implementation of Pointer Jumping on the GPU

Results


100000
10000
1000
100
10
1
0.1
0.01
Time (milliseconds)
CPU
Wylie-64
1 K
2 K
4 K
8 K
16 K
32 K
64 K
128 K
256 K
512 K
1 M
2 M
4 M
8 M
16 M
32 M
64 M
List Size

Helman JáJá Algorithm

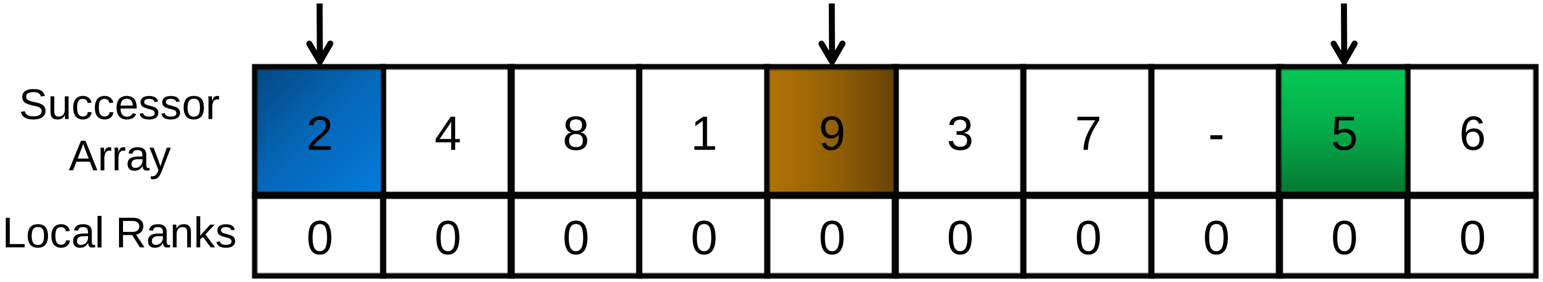
- Wyllie's algorithm is work suboptimal at ***O (n log n)***
- Helman JáJá is based on sparse ruling set approach from Reid-Miller
- Originally devised for Symmetric multiprocessor systems with low processor count.
- Algorithm of choice for all recent work in this field
- Worst Case runtime is O(log n + n/p) and O(n) work.

Helman-JáJá (Contd.)

- Helman JáJá algorithm originally devised for SMP with low processor count
- Splits a list into smaller sublists, computes local rank of each sublist and stores it into a smaller, new list.
- Perform prefix sum on the new list
- Recombine the global prefix sum of the new list with the local ranks of the original list.

Successor Array	2	4	8	1	9	3	7	-	5	6

Step 1. Select **Splitters** at equal intervals



Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	0	0	0	0	0	0	0	0	0

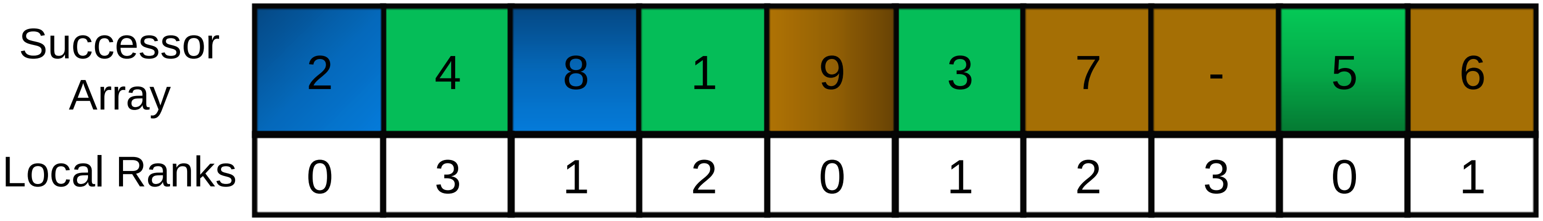
Step 2. **Traverse** the List until the next splitter is met and **increment** local ranks as we progress

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	0	1	0	1	1	0	0	0	1

Step 2. **Traverse** the List until the next splitter is met and **increment** local ranks as we progress

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	0	1	2	0	1	2	0	0	1

Step 2. **Traverse** the List until the next splitter is met and **increment** local ranks as we progress



Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 3. **Stop** When all elements have been assigned a local rank

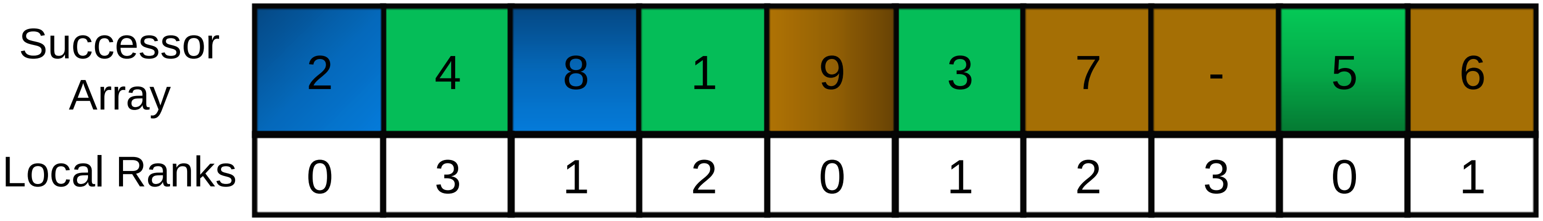
Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 4. **Create** a new list of splitters which contains a **prefix value** that is equal to the local rank of it’s predecessor

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

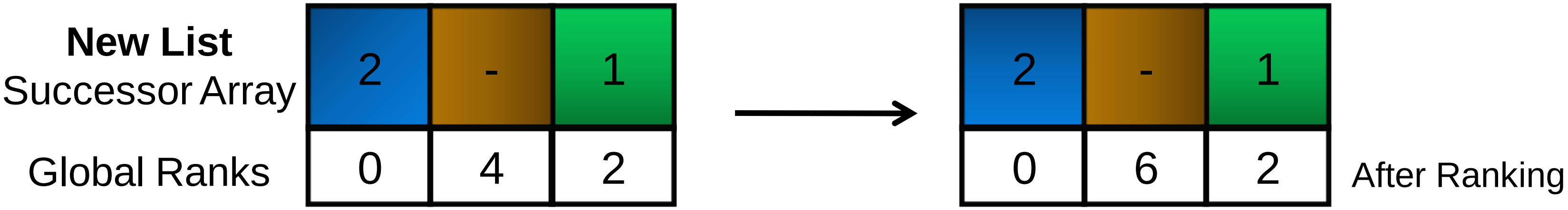
Step 4. **Create** a new list of splitters which contains a **prefix value** that is equal to the local rank of it's predecessor

New List Successor Array	2	-	1
Global Ranks	0	4	2



Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 5. Scan the global ranks array **sequentially**



New List

Successor Array	2	-	1
Global Ranks	0	4	2

→

	2	-	1
After Ranking	0	6	2

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

New List Successor Array	2	-	1
Global Ranks	0	4	2

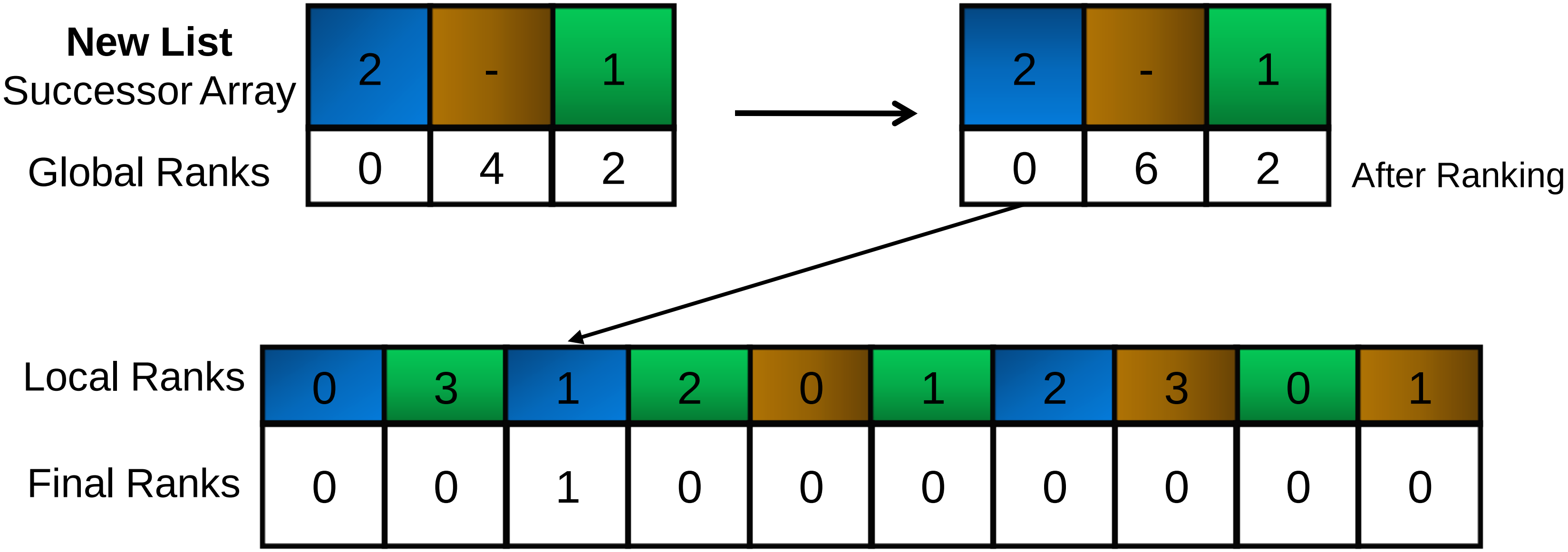
→

	2	-	1
After Ranking	0	6	2

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	0	0	0	0	0	0	0	0	0

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.



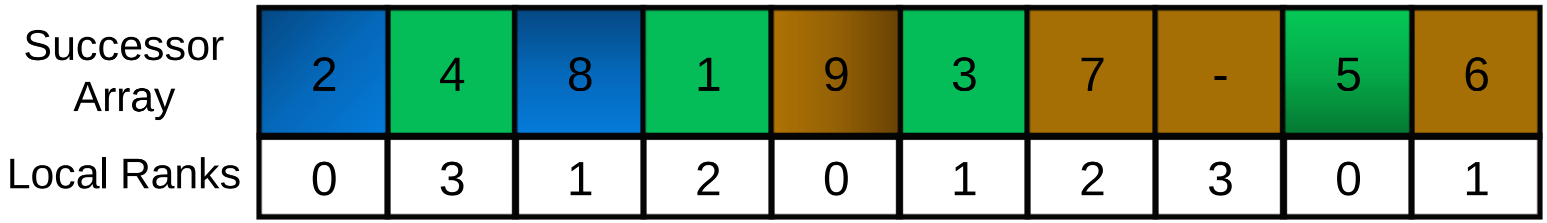
New List

Successor Array	2	-	1
Global Ranks	0	4	2

2	-	1
0	6	2

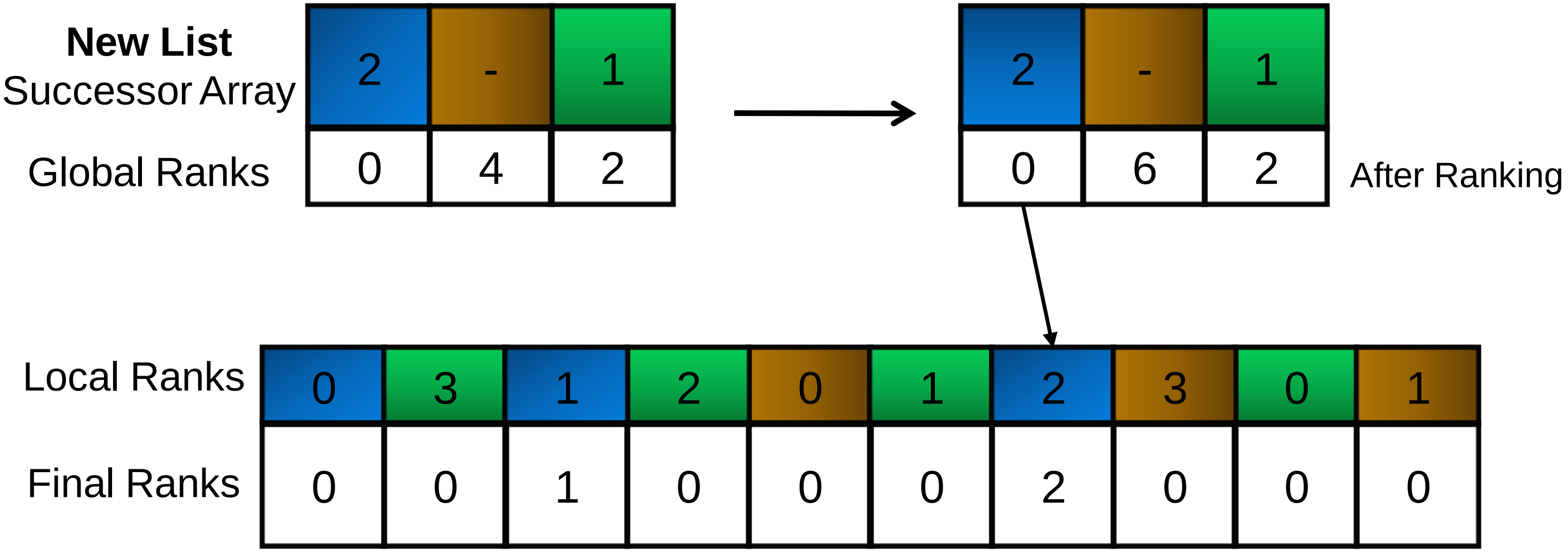
After Ranking

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	0	1	0	0	0	0	0	0	0



Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

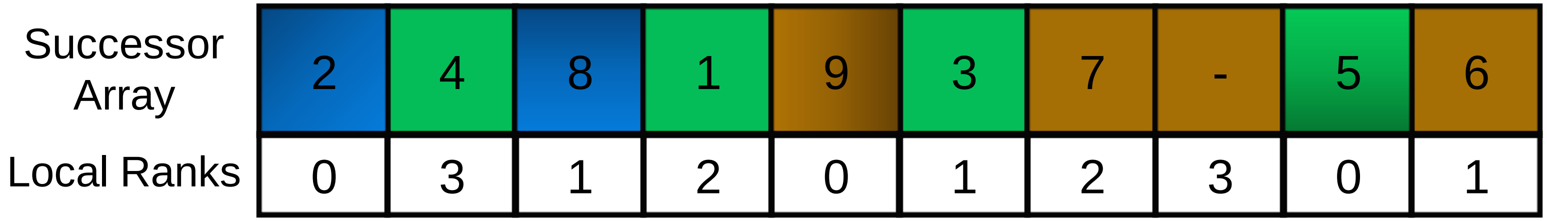
Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.



New List Successor Array	2	-	1
Global Ranks	0	4	2

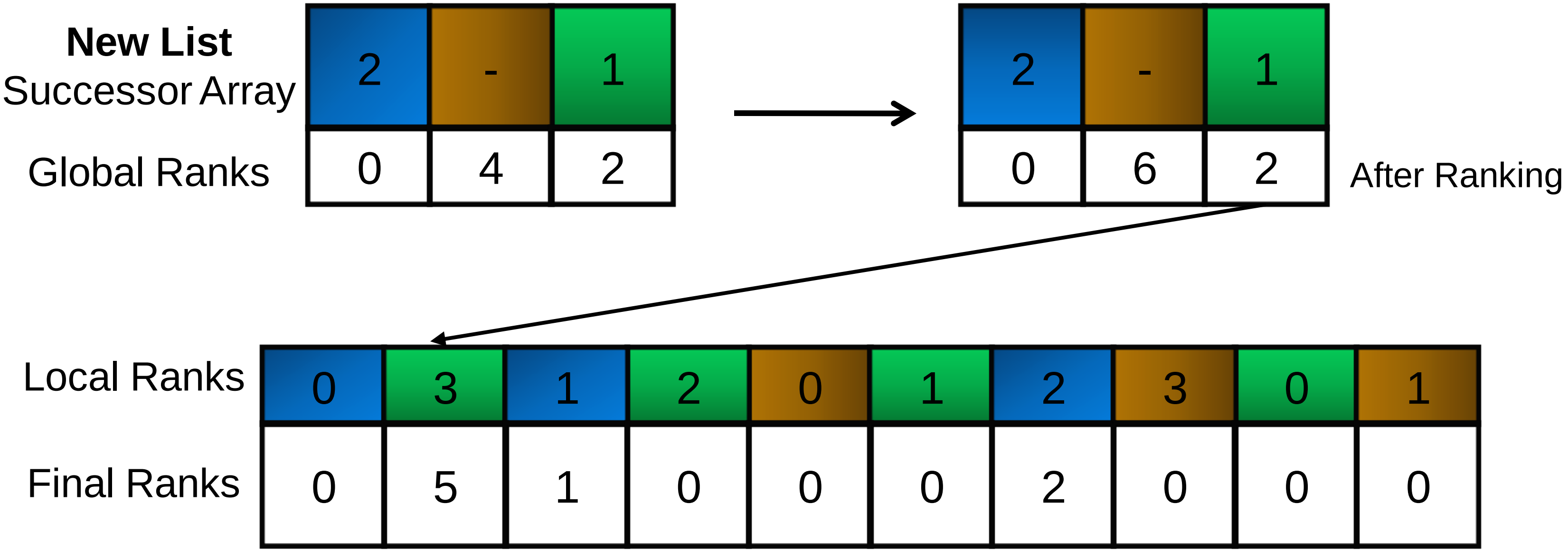
2	-	1	
0	6	2	After Ranking

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	0	1	0	0	0	2	0	0	0



Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.



New List

Successor Array	2	-	1
Global Ranks	0	4	2

→

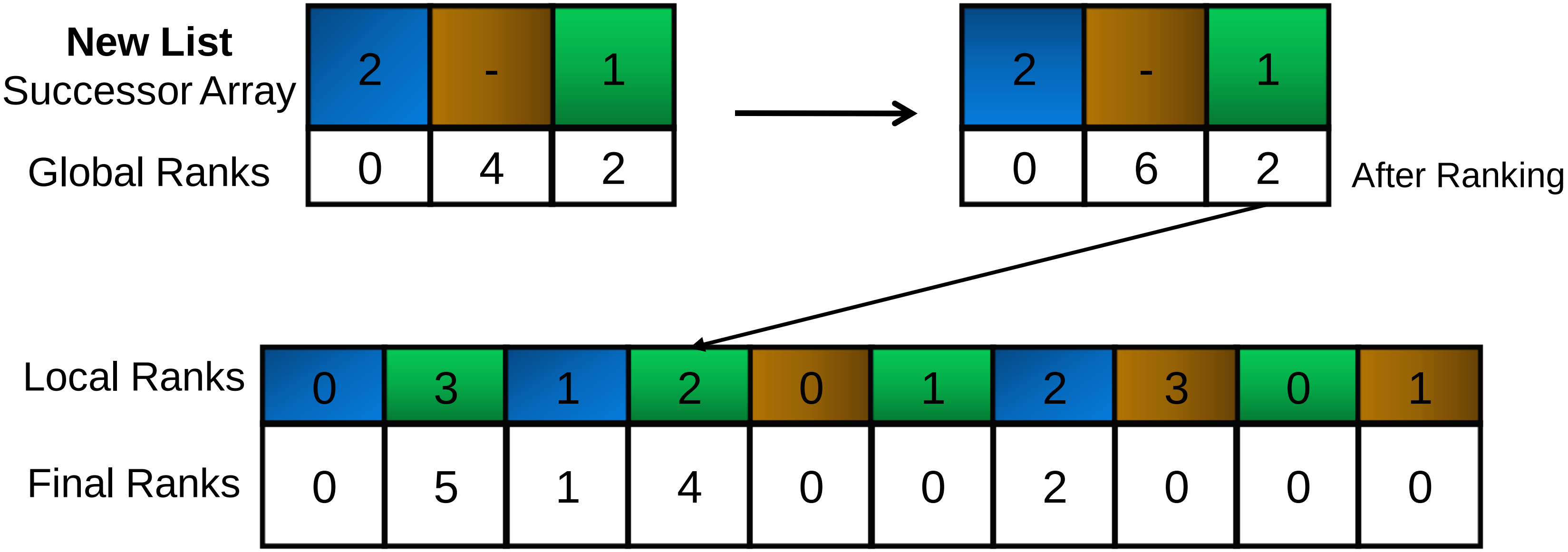
2	-	1
0	6	2

After Ranking

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	5	1	0	0	0	2	0	0	0

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.



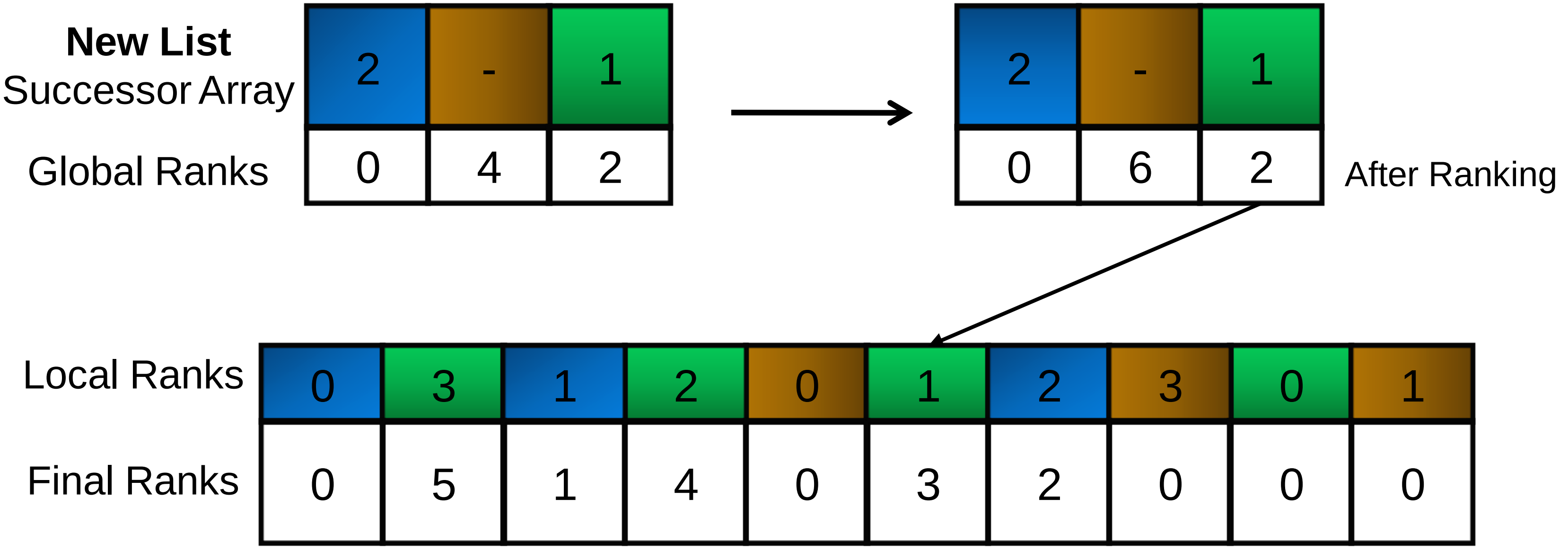
New List			
Successor Array	2	-	1
Global Ranks	0	4	2

	2	-	1
After Ranking	0	6	2

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	5	1	4	0	0	2	0	0	0

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

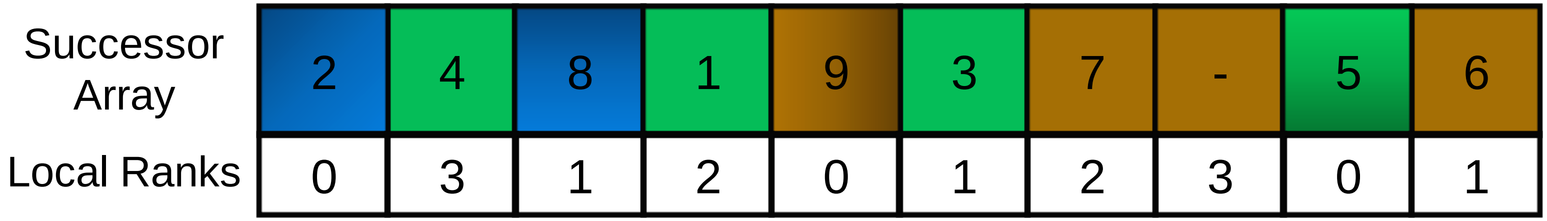


New List

Successor Array	2	-	1
Global Ranks	0	4	2

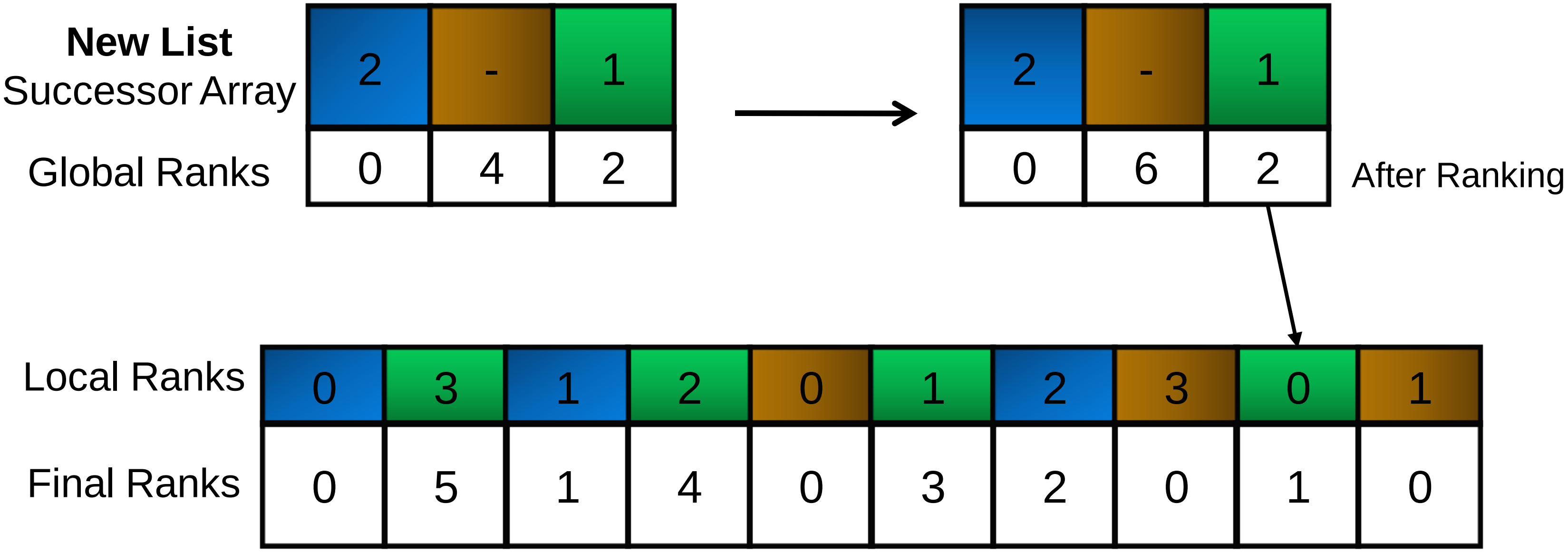
	2	-	1
After Ranking	0	6	2

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	5	1	4	0	3	2	0	0	0



Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.



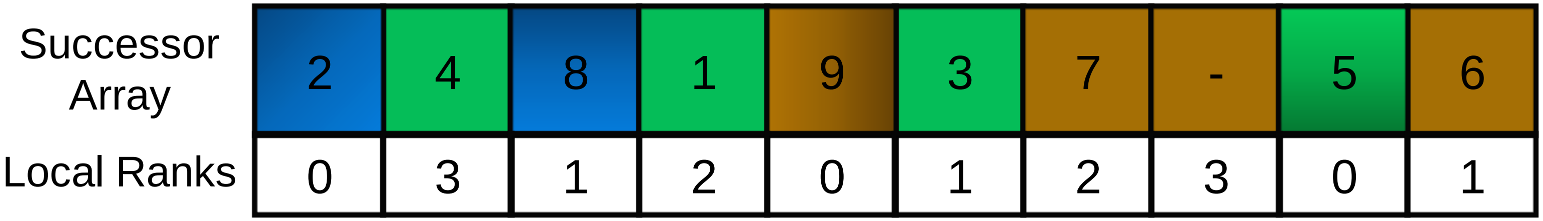
New List

Successor Array	2	-	1
Global Ranks	0	4	2

2	-	1
0	6	2

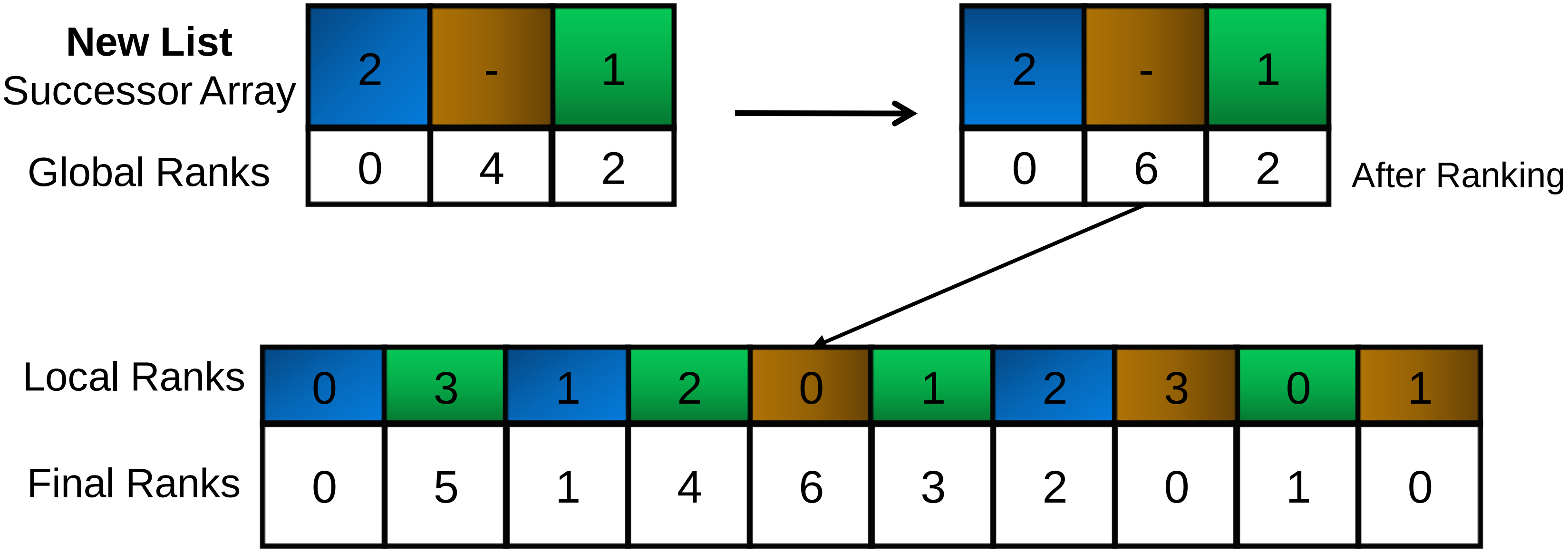
After Ranking

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	5	1	4	0	3	2	0	1	0



Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.



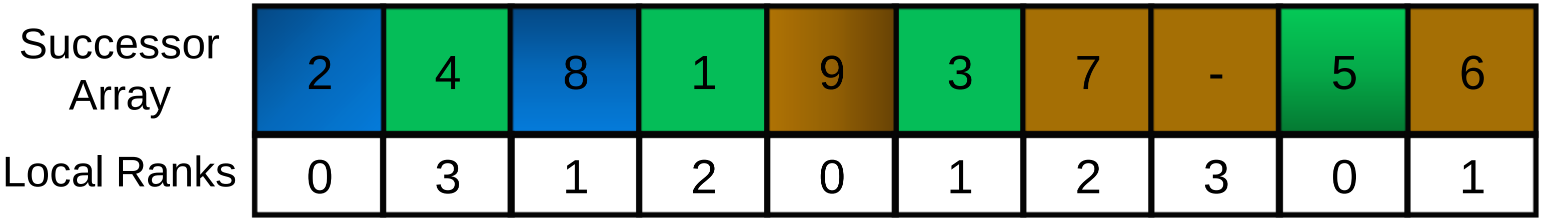
New List

Successor Array	2	-	1
Global Ranks	0	4	2

After Ranking

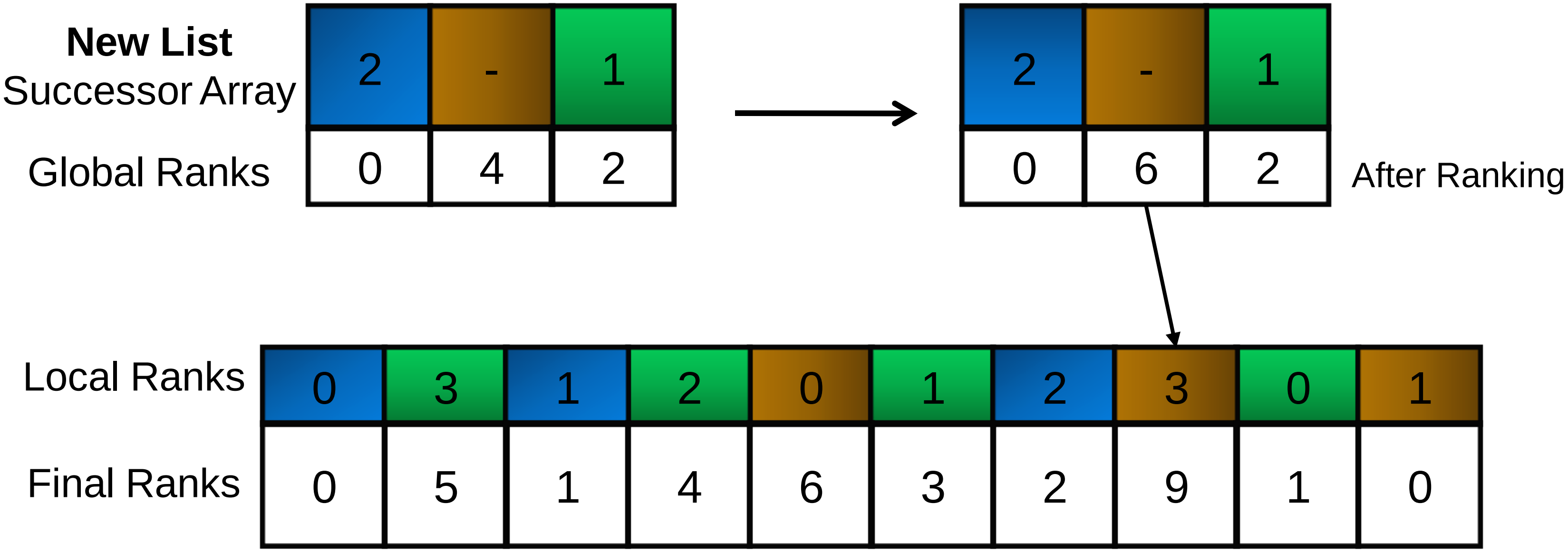
2	-	1
0	6	2

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	5	1	4	6	3	2	0	1	0



Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.



New List Successor Array	2	-	1
Global Ranks	0	4	2

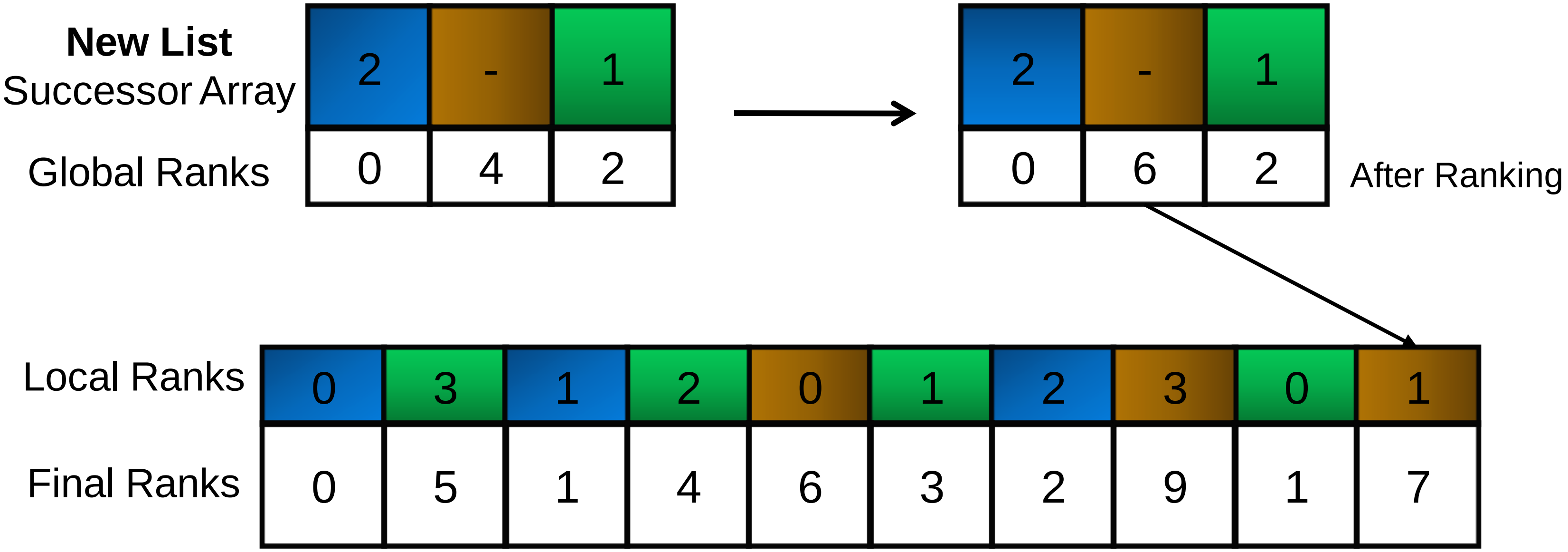
2	-	1
0	6	2

After Ranking

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	5	1	4	6	3	2	9	1	0

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.



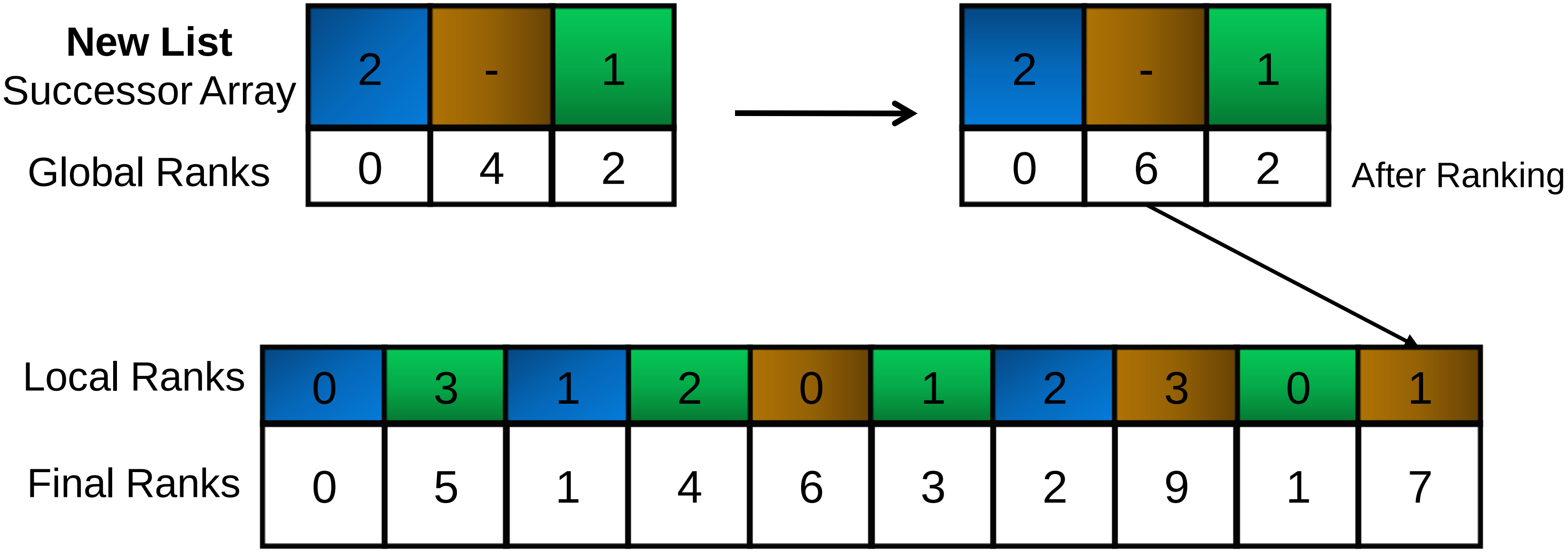
New List Successor Array	2	-	1
Global Ranks	0	4	2

	2	-	1	
	0	6	2	After Ranking

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	5	1	4	6	3	2	9	1	7

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.



New List

Successor Array	2	-	1
Global Ranks	0	4	2

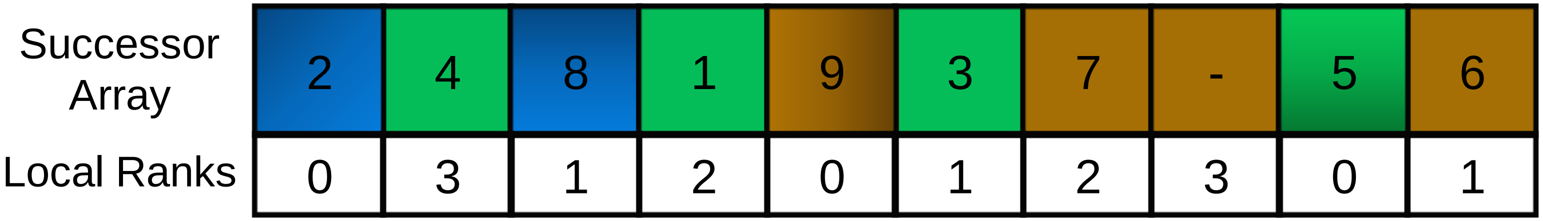
After Ranking

2	-	1
0	6	2

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	5	1	4	6	3	2	9	1	7

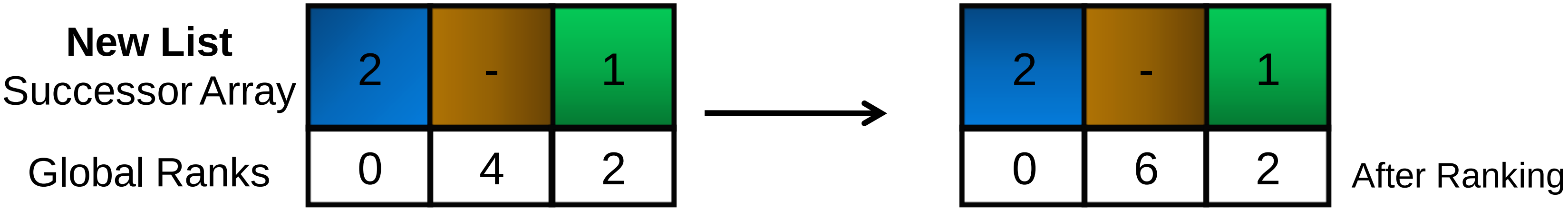
Modifying the algorithm for GPU

- Step 5 is a sequential ranking step.
- When we choose log n splitters, we reduce the list to n/log n, which is still large amount of sequential work
- By Amdahl’s law, this is a bottleneck for parallel speedup. More so in the case of GPU.



Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Make step 5 **recursive** to allow the GPU to continue processing the list in parallel



New List Successor Array	2	-	1
Global Ranks	0	4	2

2	-	1
0	6	2

After Ranking

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Make step 5 **recursive** to allow the GPU to continue processing the list in parallel

New List Successor Array	2	-	1
Global Ranks	0	4	2

→

	2	-	1
After Ranking	0	6	2

↓

Process this list again using the algorithm and reduce it further.

GPU Implementation

- Each phase is coded as separate GPU *kernel*
  - Since each step requires global synchronization.
- Splitter Selection
  - Each thread chooses a splitter
- Local Ranking
  - Each thread traverses its corresponding sublist and get the global ranks
- Recursive Step
- Recombination Step
  - Each thread adds the global and local ranks for each element

When do we stop?

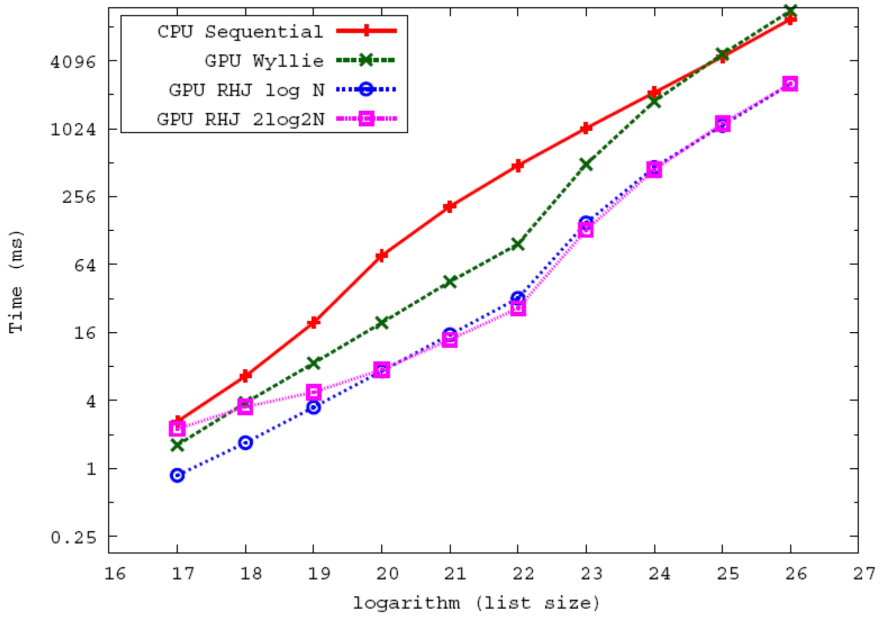
- Convergence can be met until list size is 1
- We also have the option to send a small list to CPU or Wyllie’s algorithm so that it can be processed faster than on this algorithm.
- May save about 1% time

Choosing the right amount of splitters

- Notice that choosing splitters in a random list yields uneven sublists
- We can attempt to load balance the algorithm by varying the no. of splitters we choose.
- n/log n works for small lists, n/2 $\log^2$ n works well for lists > 1 M.

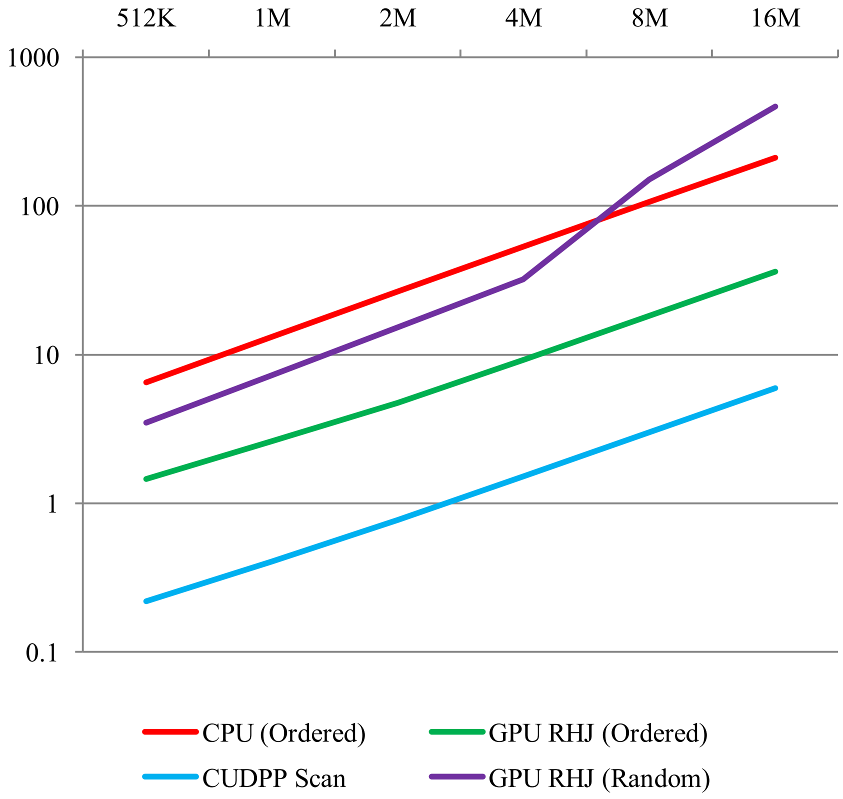
Results

- Significant Speedup over sequential algorithm on CPU ~ 10x
- Wylie's algorithm works best for small lists < 512 K
- GPU RHJ works well for large lists
- 2 log 2N works well for lists > 1M



Ordered Lists

- Perform significantly faster than random lists.
- Data locality is automatically taken advantage of by the global memory access hardware
- Compared with GPU ordered scan.



Other Irregular Applications

- Graph Algorithms:
  - Shortest path
  - Breadth-First Search
  - Spanning Tree, etc.
  - Etc
- Many others

General Graph Algorithms

1. General Graph Algorithms:
   - Breadth First Search
   - ST- Connectivity
   - Single Source Shortest Paths
   - All Pairs Shortest Path
   - Minimum Spanning Tree
   - Max Flow
2. Randomness in the graph posses great difficulty in utilizing the hardware resources.
3. Connectivity is unknown.
4. Graph Representation is not trivial.

Singular Value Decomposition

Work with Sheetal Lahabar

Appeared in IEEE IPDPS.
Rome. June 2009.

Problem Statement

- SVD on GPU

SVD of matrix $A_{(mxn)}$ for $m>n$

$$\mathbf{A} = \mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}$$

U and *V* are orthogonal and **Σ** is a diagonal matrix

Motivation

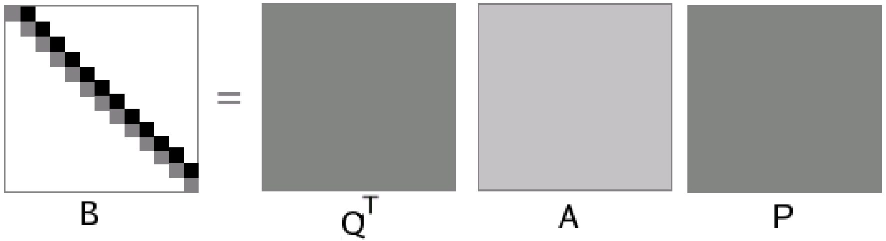
- SVD has many applications
- High computational complexity
- GPUs have high computing power
  - Teraflop performance
- Exploit the GPU for high performance

Methods

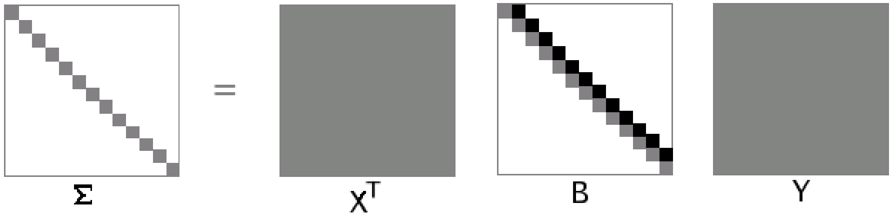
- SVD algorithms
  - Golub Reinsch (Bidiagonalization and Diagonalization)
  - Hestenes algorithm(Jacobi)
- Golub Reinsch method
  - Simple and compact
  - Maps well to the GPU
  - Popular in numerical libraries

Golub Reinsch algorithm

- Bidiagonalization:
  - Series of householder transformations



- Diagonalization:
  - Implicitly Shifted QR iterations



SVD

- Overall algorithm
  - $B = Q^{T}AP$

    Bidiagonalization of A to B

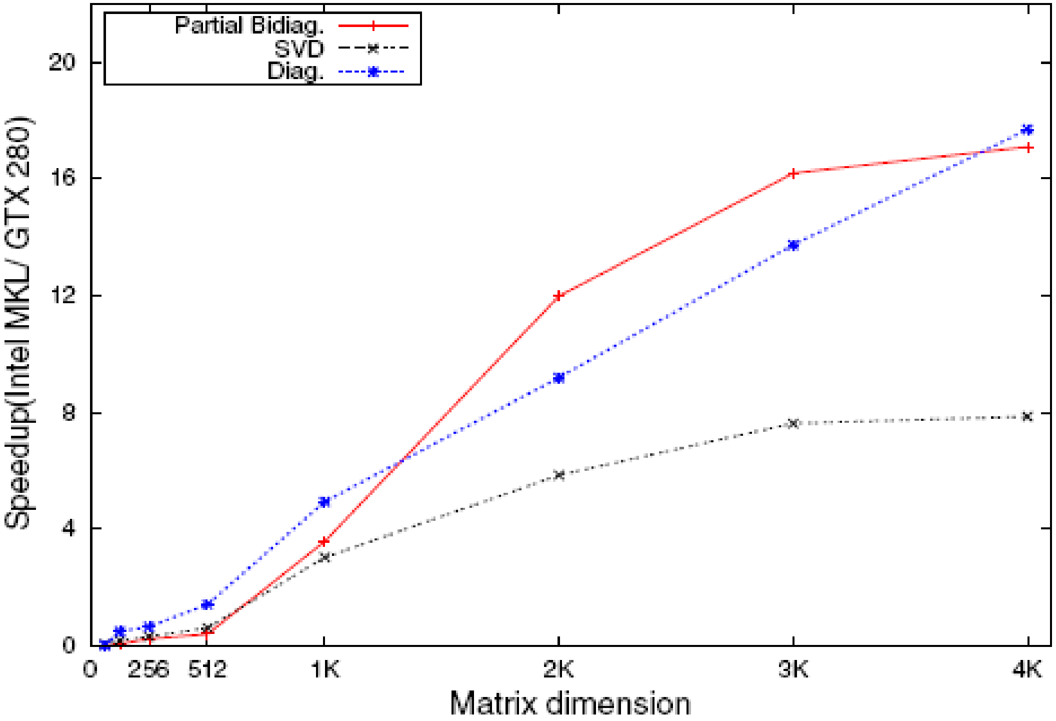
  □ $\Sigma = X^{T}BY$

    Diagonalization of B to Σ
  - $U = QX,\ V^{T} = (PY)^{T}$

    Compute orthogonal matrices U and V^{T}
- Complexity: $O(mn^{2})$ for $m>n$

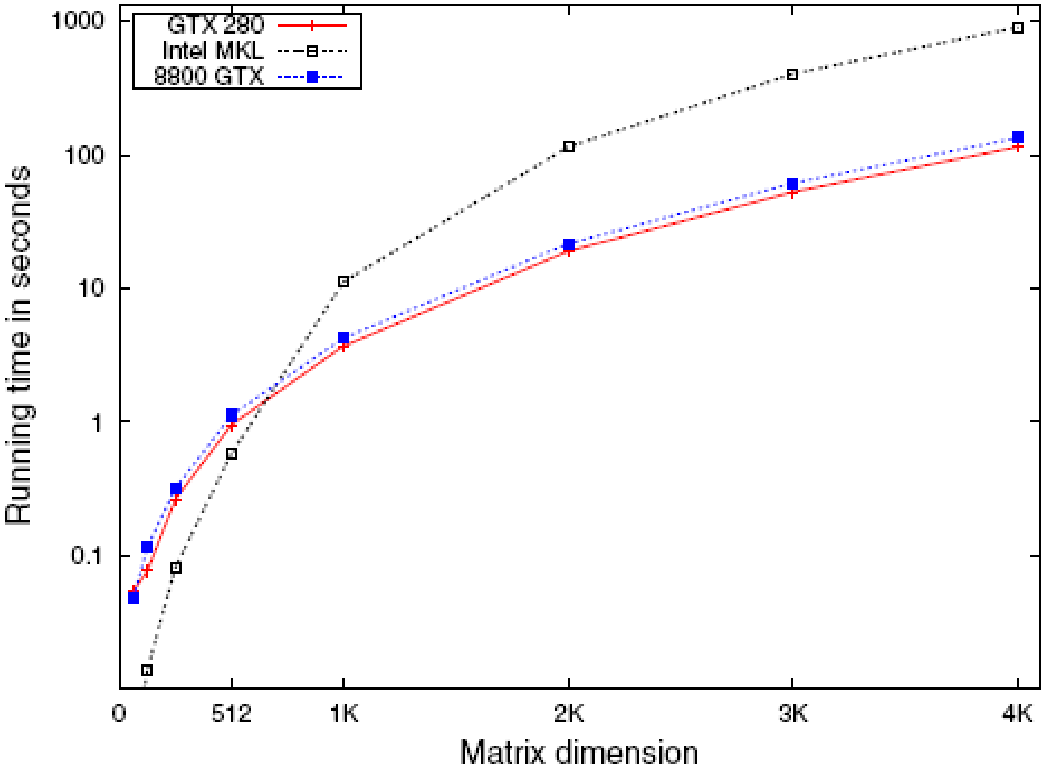
Results

- Intel 2.66 GHz Dual Core CPU used
- Speedup on NVIDIA GTX 280:
  - 3-8 over MKL LAPACK
  - 3-60 over MATLAB



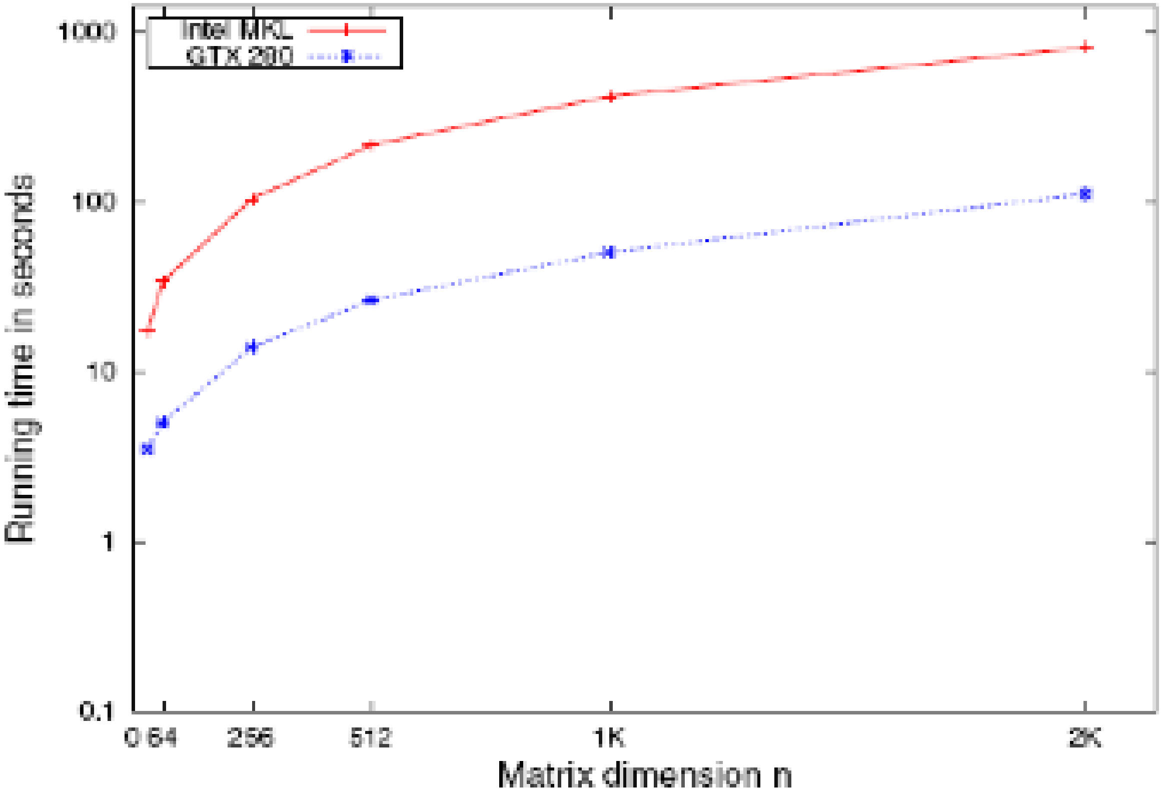
Contd…

- CPU outperforms for smaller matrices
- Speedup increases with matrix size



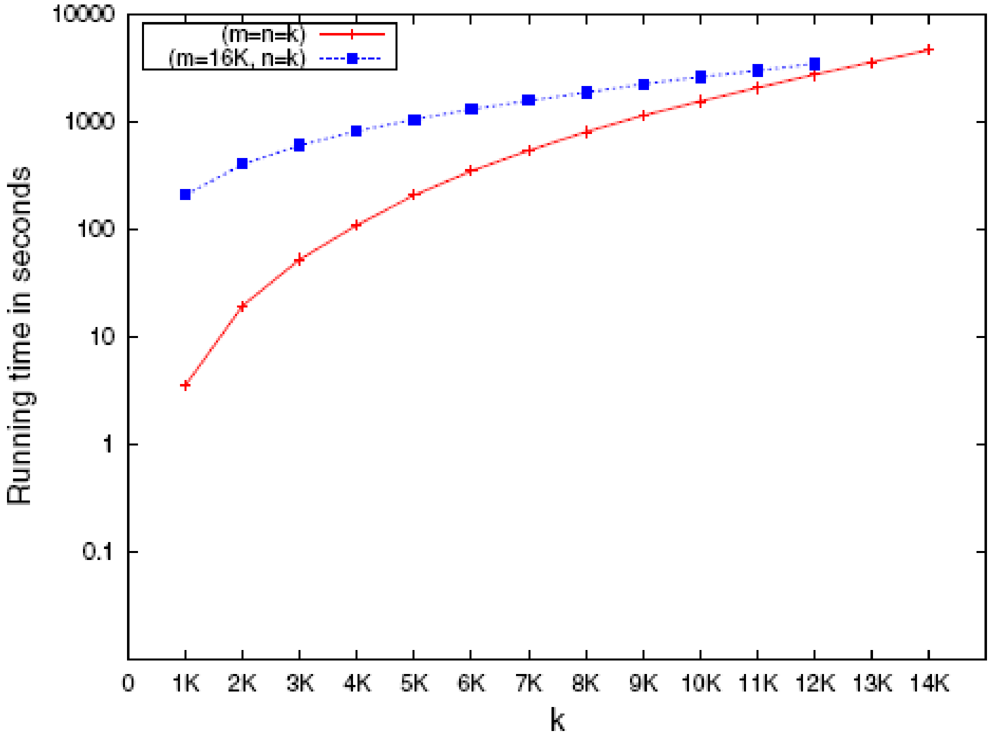
Contd…

- SVD timing for rectangular matrices (m=8K)
  - Speedup increases with varying dimension



Contd…

- SVD of upto 14K x 14K on Tesla S1070 takes 76 mins on GPU
- 10K x 10K SVD takes 4.5 hours on CPU, 25.6 minutes on GPU



Contd…

- Yamamoto achieved a speedup of 4 on CSX600 for very large matrices
- Bobda report the time for 10^6 x 10^6 matrix which takes 17 hours
- Bondhugula report only the partial bidiagonalization time

Timing for Partial Bidiagonalization

- Speedup:1.5-16.5 over Intel MKL
- CPU outperforms for small matrices
- Timing comparable to Bondhugula (11 secs on GTX 280 compared to 19 secs on 7900)

Time in secs

SIZE	Bidiag. GTX 280	Partial Bidiag. GTX 280	Partial Bidiag. Intel MKL
512 x 512	0.57	0.37	0.14
1K x 1K	2.40	1.06	3.81
2K x 2K	14.40	4.60	47.9
4K x 4K	92.70	21.8	361.8

Timing for Diagonalization

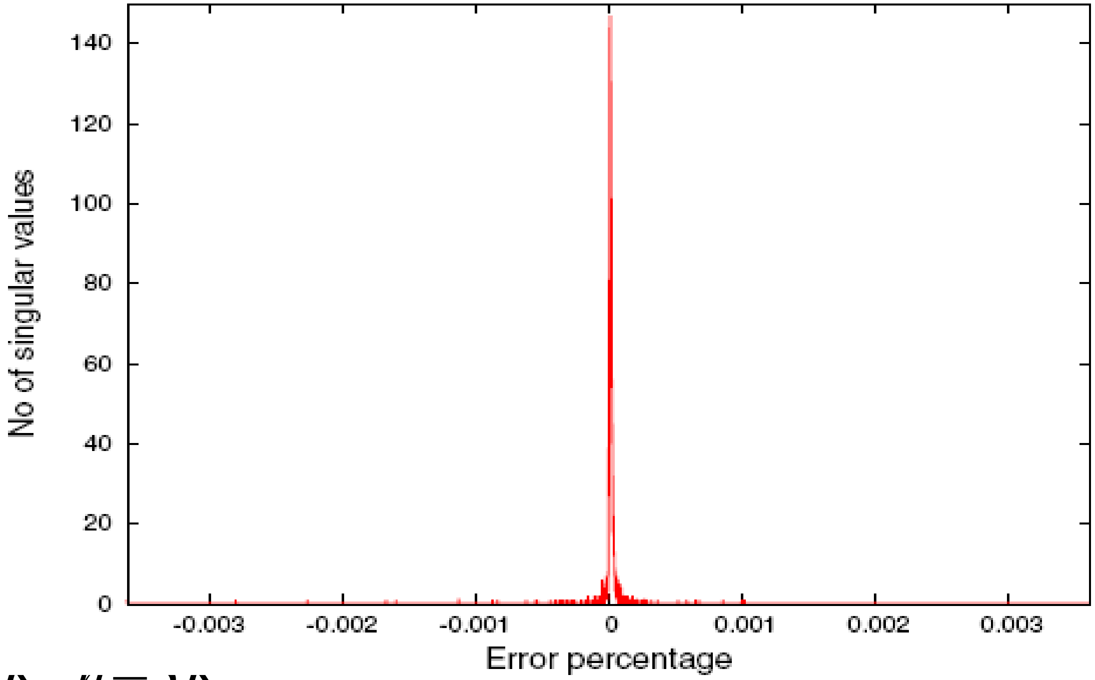
- Speedup:1.5-18 over Intel MKL
- Maximum Occupancy: 83%
- Data coalescing achieved
- Performance increases with matrix size
- Performs well even for small matrices

SIZE	Diag. GTX 280	Diag. Intel MKL
512 x 512	0.38	0.54
2K x 2K	5.14	49.1
4K x 4K	20	354
8K x 2K	8.2	100

Time in secs

Limitations

- Limited double precision support
- High performance penalty
- Discrepancy due to reduced precision



m=3K, n=3K

Contd…

- Max singular value discrepancy = 0.013%

  Average discrepancy < 0.00005%
- Average discrepancy < 0.001% for U and V^T
- Limited by device memory

Regular Algorithms on CUDA

Mapping an Image on CUDA

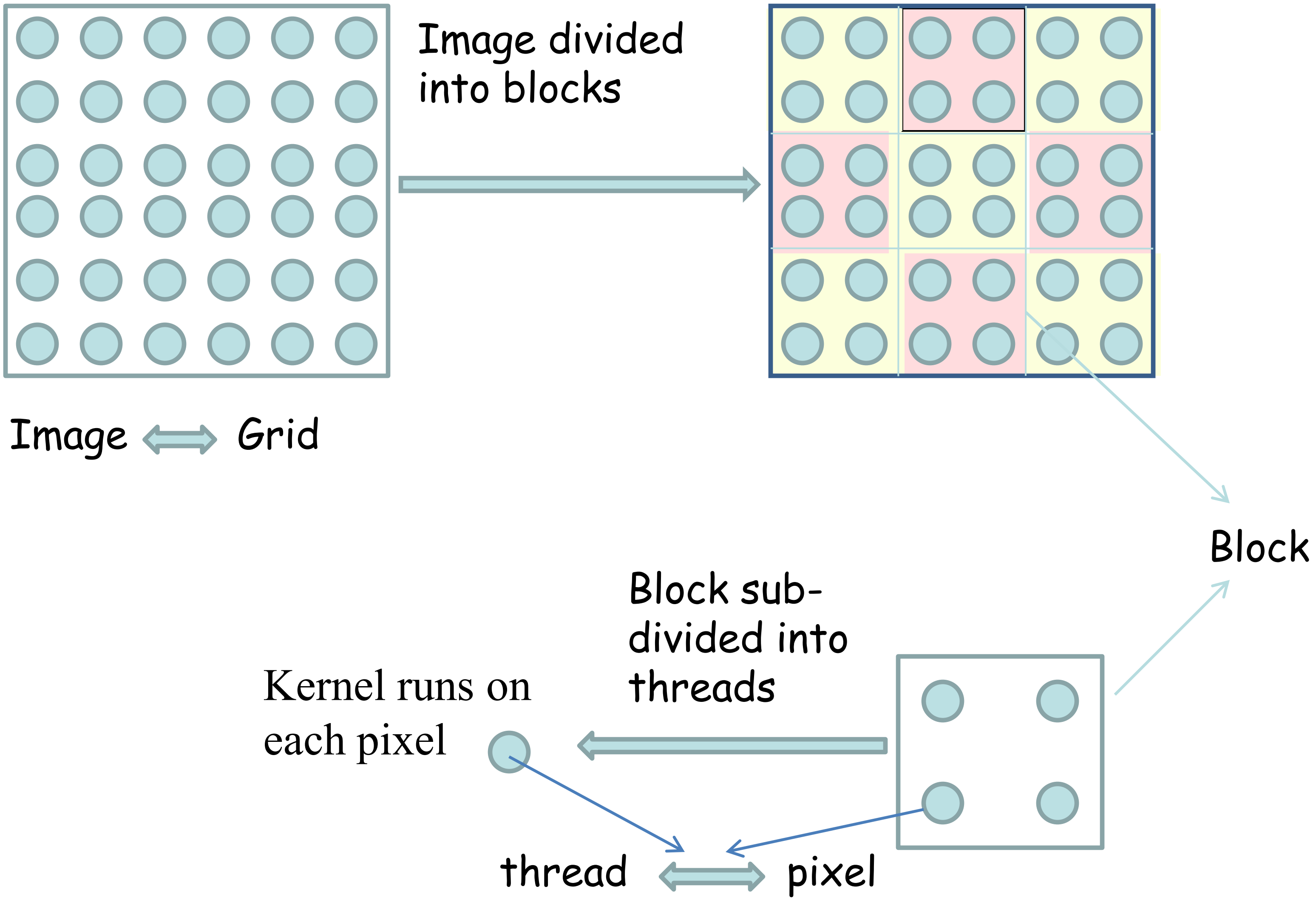


Image divided into blocks

Image ⟺ Grid

Block

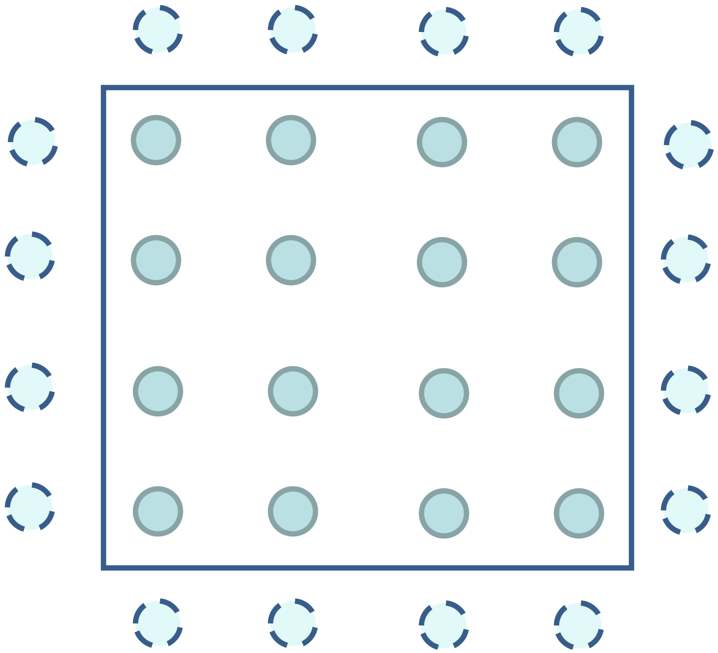
Block sub-divided into threads

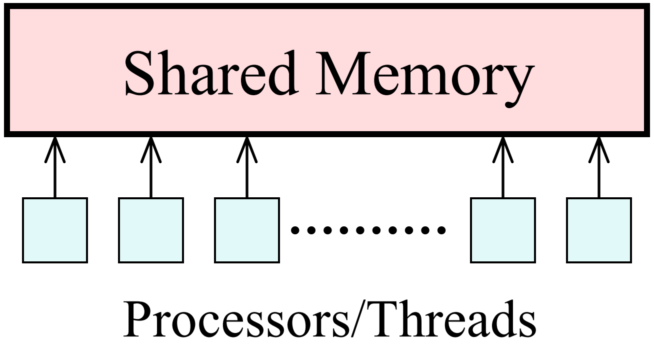
Kernel runs on each pixel

thread ⟺ pixel

Image Processing, Filtering

- Thread accesses its pixel data using thread to pixel mapping
  - Read is efficient: Coalesced
  - Process each pixel independently and write results
- 2D Filtering: Keep block values + neighbouring rows and cols in shared memory
  - Coalesced access to bring to SM
  - Synchronize threads of block to ensure loading
  - A thread computes its pixel's output value from shared memory
  - Write results coalesced





Mean filtering

```
float *shMem = (float *) &sharedMem[0]  // Pointer
// Computer image coordinates
x = blockIdx.x*blockDim.x + threadIdx.x
y = blockIdx.y*blockDim.y + threadIdx.y
// Compute a local coordinate within block
localIndex = threadIdx.x+threadIdx.y*blockDim.x
// Copy own portion to shared memory
shMem[localIndex] = globalImage[y*width + x]
__syncthreads() // Wait till all copying is done
// Compute the required output and copy back
g_odata[y*width + x] = meanGreyValue()
```

Mean Computation

```
float meanValue = 0.0
// Compute the average of the 9 pixels
for (int i=0; i<3; i++)
 for (int j=0; j<3; j++)
      indx = (threadIdx.x – i) + (threadIdx.y – j)*blockDim.x
      meanValue += shMem[indx]
meanValue /= 9.0
```

Note:

- Borders are not handled properly.
- Needs if-then-else to process borders specially
- Divergence: Different threads doing different actions
- Always suffers in performance on SIMD architectures
- Intra-warp divergence only for CUDA

Image Rotation

- Rotate by angle θ.
- $x' = x \cos \theta - y \sin \theta$
  $y' = x \sin \theta + y \cos \theta$
- Fractional coordinates!
- Think reverse and interpolate
- $x = x' \cos \theta + y' \sin \theta$
  $y = x' \sin \theta - y' \cos \theta$
- Can use texture memory to get interpolation

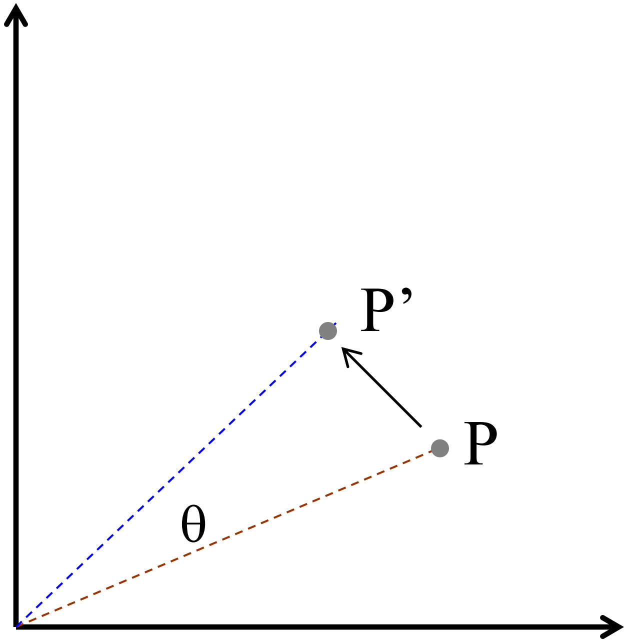


Image Rotation

```
// image/texture coordinates
    x = blockIdx.x*blockDim.x + threadIdx.x
    y = blockIdx.y*blockDim.y + threadIdx.y;
    u = x / (float) width
    v = y / (float) height;
// transform coordinates
    u -= 0.5f, v -= 0.5f;
    tu = u*cosf(theta) + v*sinf(theta) + 0.5f
    tv = v*cosf(theta) - u*sinf(theta) + 0.5f;
// read from texture and write to global memory
    g_odata[y*width + x] = tex2D(tex, tu, tv)

// Interpolation:          img[i,j] (1-b) (1-c) + img[i,j+1] (1-b) c +
//                         img[i+1,j] (1 - b) c + img[i+1,j+1] b c
```

Data-Parallel Computation

- Kernels operate on data elements
  - Little interaction between data elements
  - Simple model. **Think like data elements**. Know little!
- Also called
  - Stream computing
  - Throughput computing
- Application areas
  - Signal processing, Image processing
  - (Large) matrix operations
  - Scientific computing with large data
    - Molecues, fluid flow, ….

Thank you!

Towards a Model

- The GPGPU benefit
  - Low cost (order of hundreds of $)
  - Widely available
  - High computational power of roughly 1 TFLOP
  - C like programming model called CUDA

GPGPU Success Stories

- Image processing
  - FFT, filters
- Graph Algorithms
  - BFS, Shortest Paths, Graph Cuts
- Linear algebra
- Primitives
  - Scan, Split, list ranking (speed up < 10)
- Physics (n-body simulation)
  - speed up of 200

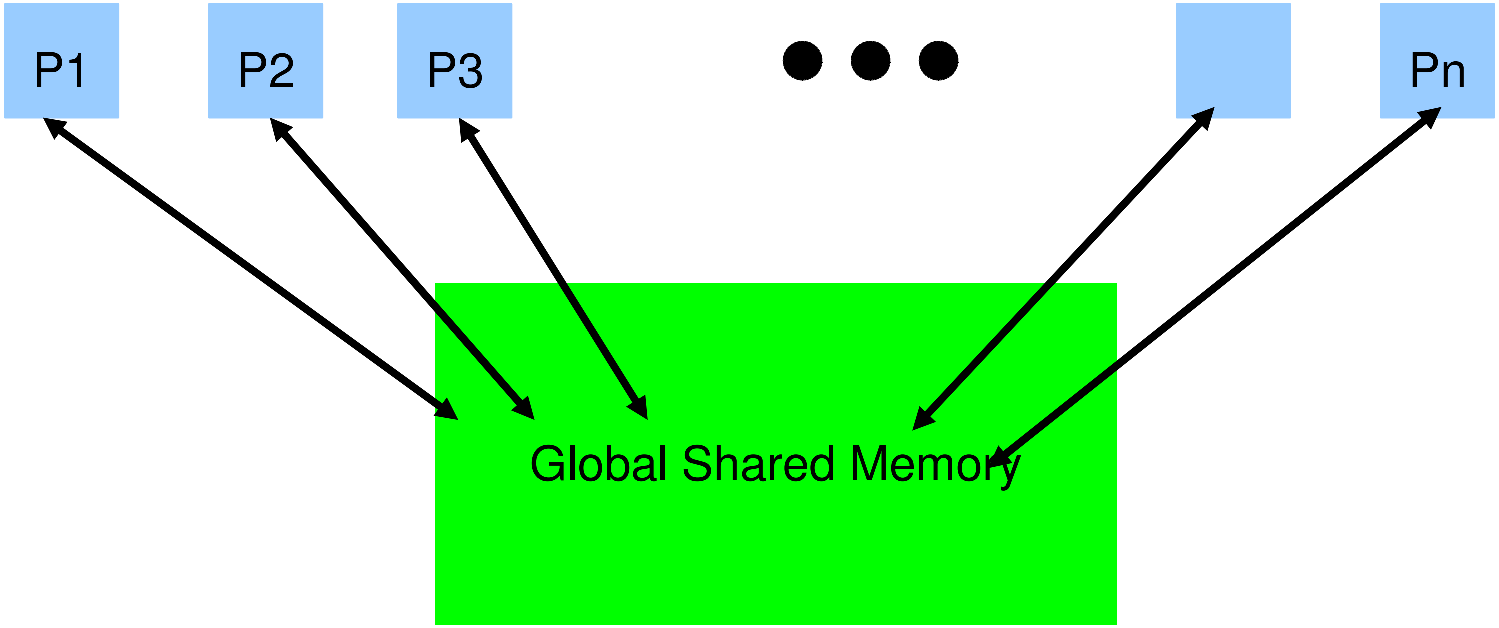
The Missing Link

- Speed up varies across problems and implementations.
  - Can the speed-up be analyzed?
  - Can one predict the speed up?
  - How much of architectural details have to be included?

Benefits of an Analytical Model

- Limits of GPU parallelizability
  - augment the PRAM model
- Program profiling
  - An informative profile identifying performance bottlenecks
- A software simulator for GPU
  - Help future architectural proposals for the GPU

Why PRAM does not suffice



- PRAM is a purely algorithmic model.
  - Very good at identifying the extent of parallelism
- Ignores architectural costs such as memory hierarchy, memory latencies.
- Ignores programming costs such as synchronization.

Few Available Models

- Plenty of models in the parallel algorithms community
  - PRAM
  - BSP, QRQW, LogP, ...,
- Design space optimization [Ryoo et al. 2008]
  - A structured way to handle parameter optimization
  - Two parameters: utilization and efficiency
  - Extended to multi-GPU design space [Schaa et al. 2009]

Few Available Models

- A very similar work [Hong and Kim, 2009]
  - independent work
  - introduces two parameters
    - Memory warp parallelism
    - Compute warp parallelism
- A related work in this conference

Highlights of Our Model

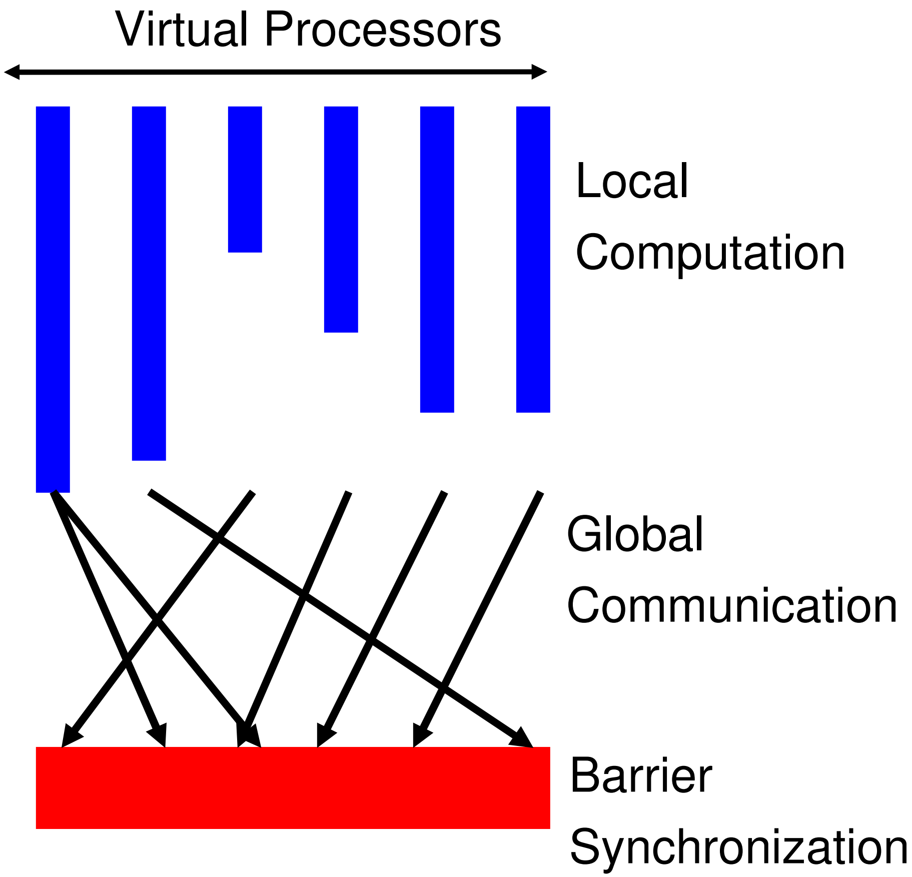
- A performance prediction model for GPGPU
  - Uses extensions to existing models such as BSP, PRAM, and QRQW
- Static runtime prediction
  - Analogous to asymptotic analysis
  - Done at the CUDA code level.
- Experimental validation by targeted experiments
- Three case-studies
  - Matrix multiplication, List ranking, Histogram
  - More case studies in our other works, e.g., AES

Overview of the Model [HiPC 2009]

- Use three existing models
  - BSP [Valiant, 1990]
  - PRAM [Fortune and Wyllie, 1978]
  - QRQW [Gibbons, Mathias, Ramachandran,1996]
- Small extensions to each of the three models
- Separate memory and computation in a kernel

BSP Model

- Proposed as a bridging model for parallel computation.
- Computation organized as supersteps.
- Threads synchronized at the end of each superstep.
  - Maps to kernels in GPGPU, with explicit synchronization.
  - Consider each kernel at a time, and
  - Express the runtime of a program as the sum of the run times of supersteps (kernels).



Measuring Computation in a Kernel

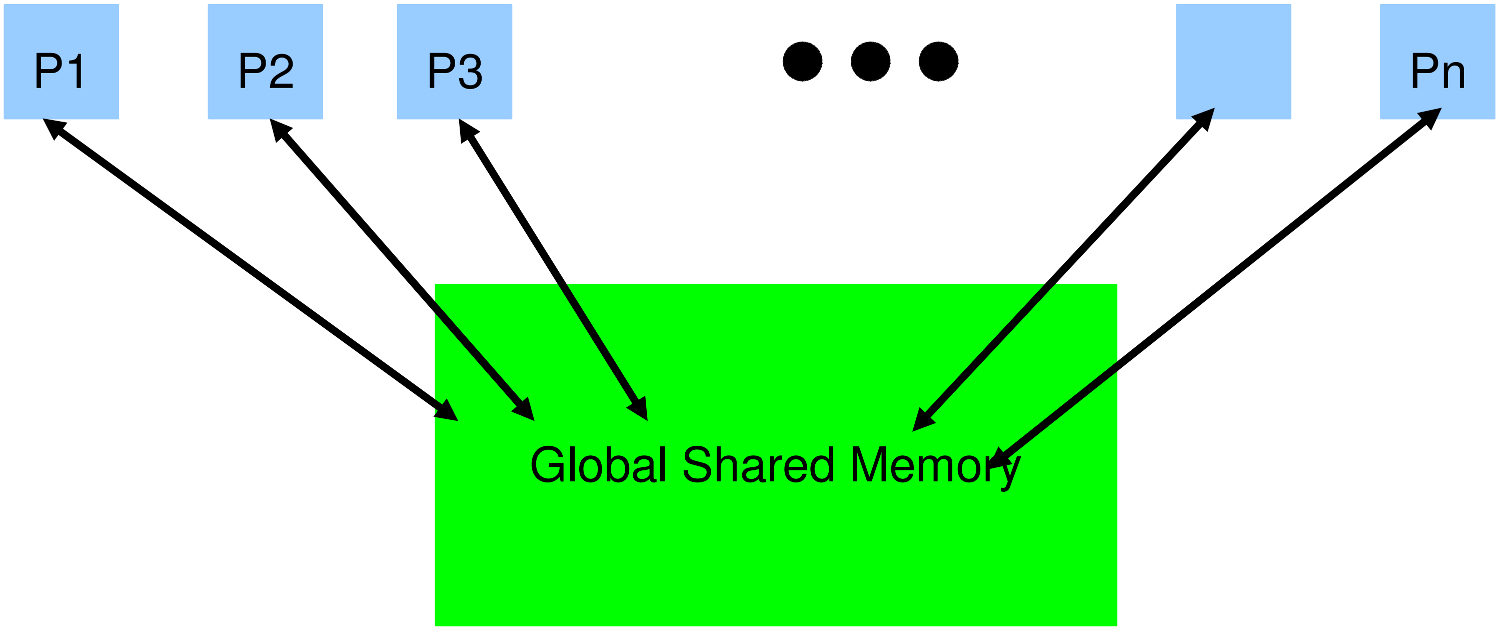
- GPU is not a very versatile architecture
- Number of cycles varies heavily even for simple operations.
  - 4 cycles for a multiply
  - 40 cycles for integer modulo
- Hence, add up the cycles for all the compute operations in a kernel.

```
        .
        .
a = b + c //4 cycles
a = b * c //4 cycles
a = a % 8 //40cycles
        .
        .
        .
```

Measuring Memory Access in a Kernel

- GPU has a deep memory hierarchy
  - Global Memory
  - Shared Memory
  - Constant Memory
  - Texture Memory
- In this model, we'll focus only on the global and the shared memory.

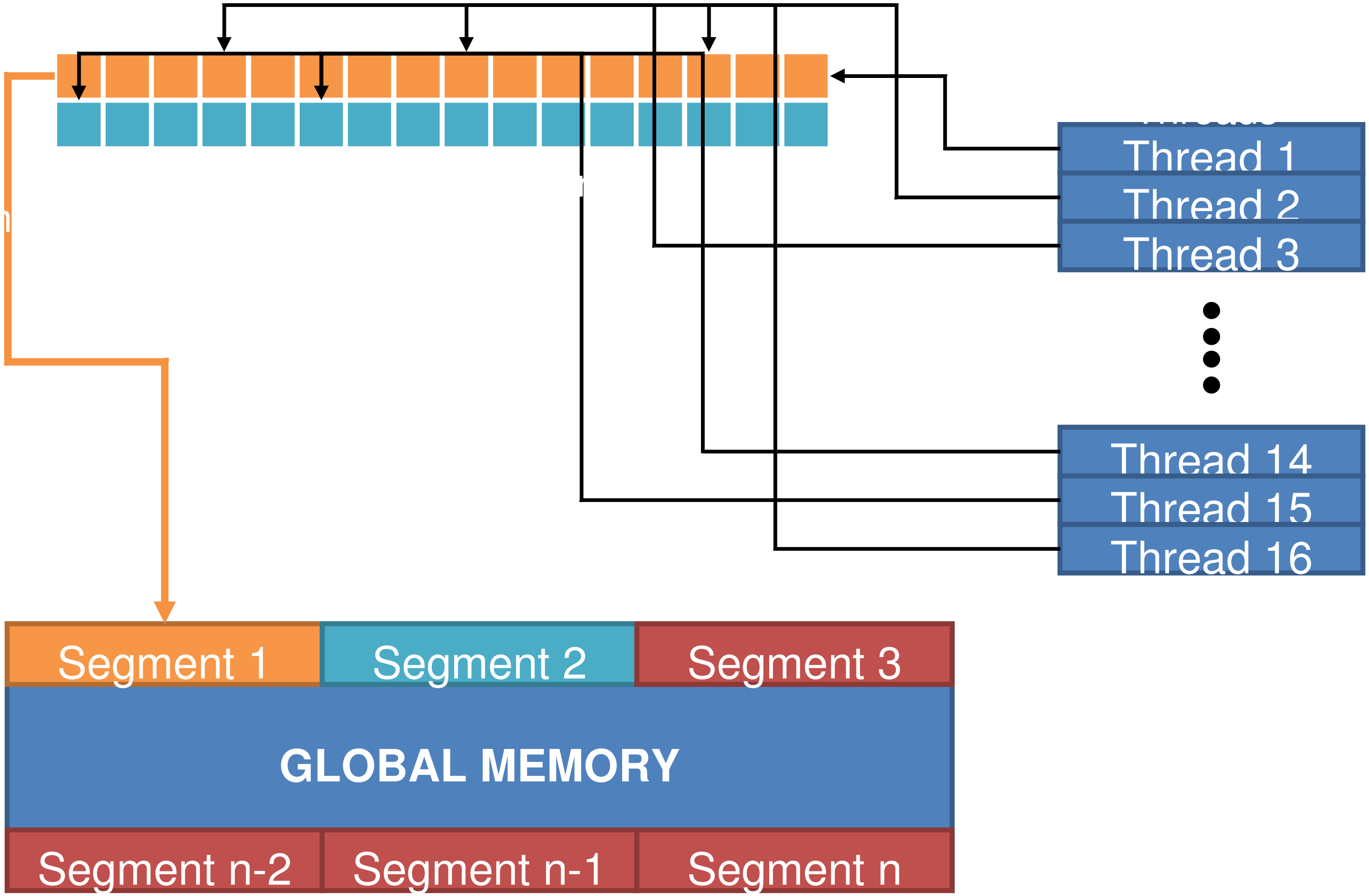
PRAM Model for Global Memory Accesses



- An extension of the RAM model
  - Uniform cost for each memory access and computation.

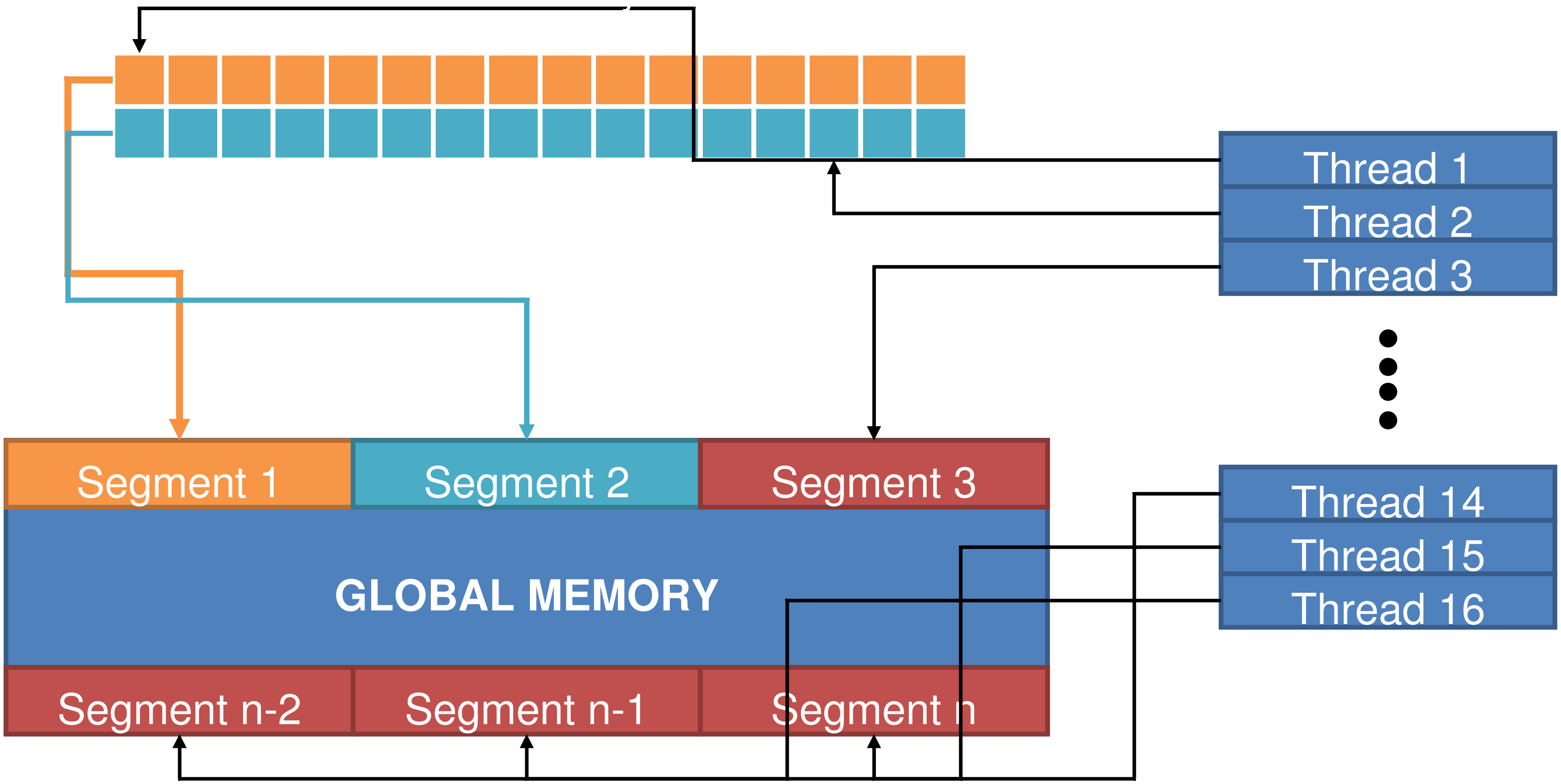
- For the GPU, we need to think of a non-uniform PRAM model.
  - Non-uniformity depends on the nature of global access.

PRAM Model for Global Memory Accesses



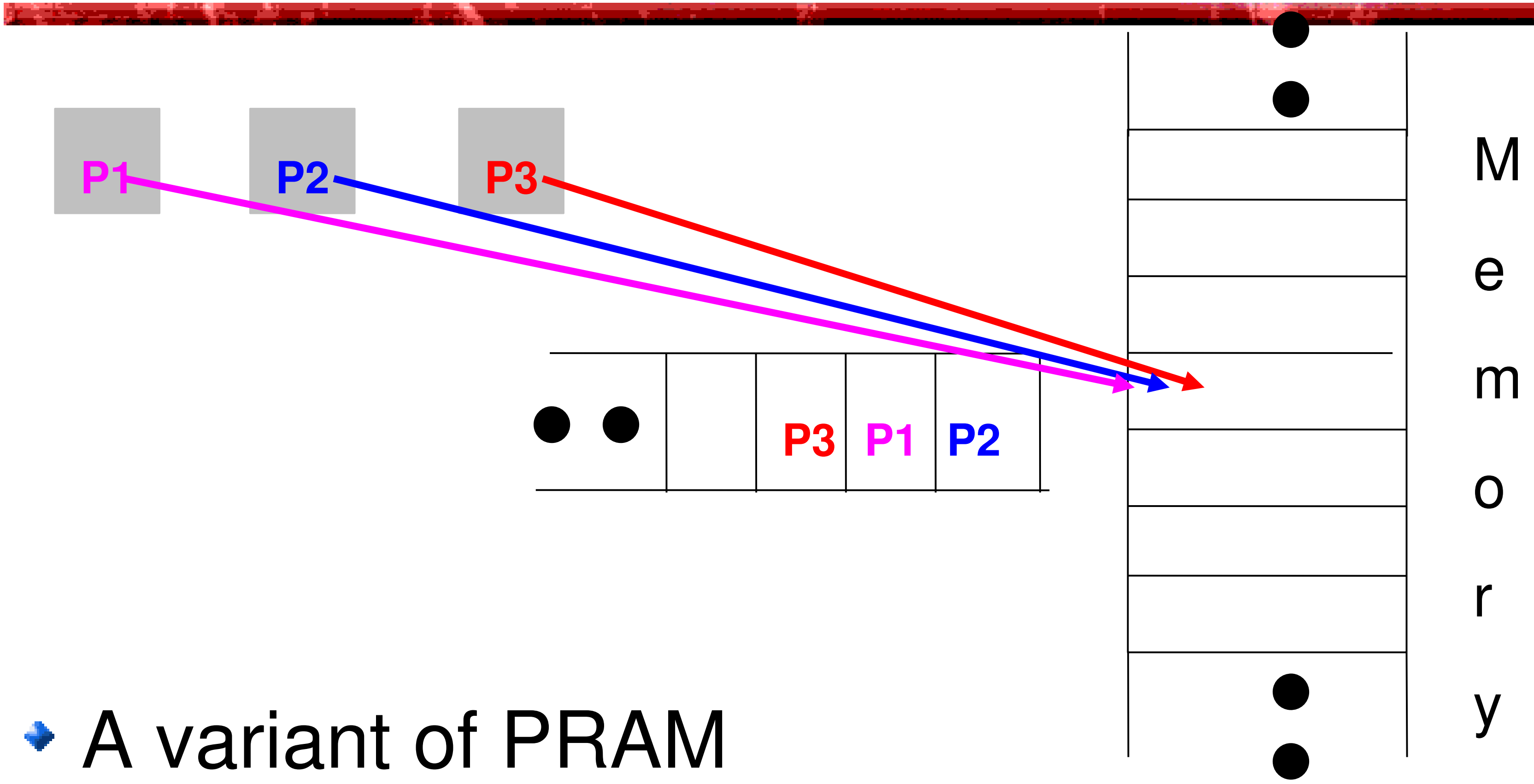
- On the GPU however global memory access cost varies
  - About 20 cycles for coalesced access [Nvidia manual]

PRAM Model for Global Memory Accesses



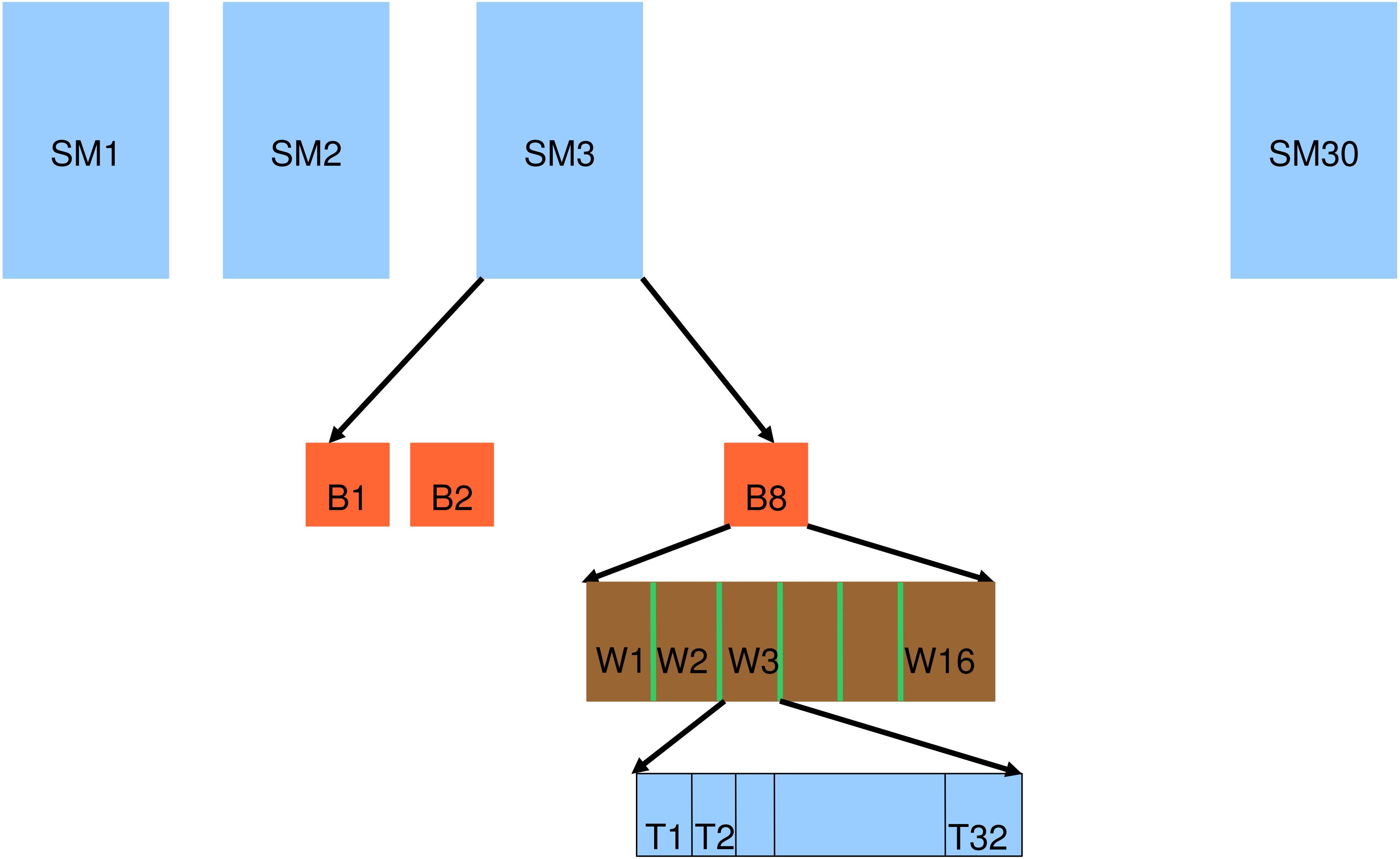
- On the GPU however global memory access cost varies
  - 400 – 600 cycles for non-coalesced access [Nvidia manual]

QRQW Model For Shared Memory Accesses



- A variant of PRAM
- Cost function based on number of access conflicts
  - Identity function – CRCW PRAM
  - Linear – GPU, and other existing parallel computers.

The Overall Model



An Equation

- $T(P)$ = Time for executing a program P
- $T(P) = \sum$ Time for executing a kernel K
- $T(K)$ = No. of Cycles in the kernel K x (1/Clock Rate)
- $C(K) = N_B(K) \times N_w(K) \times N_t(K) \times C_T(K) \times (1/D \times N_c)$
- How to obtain $C_T(K)$?

N_B : No. of blocks/SM

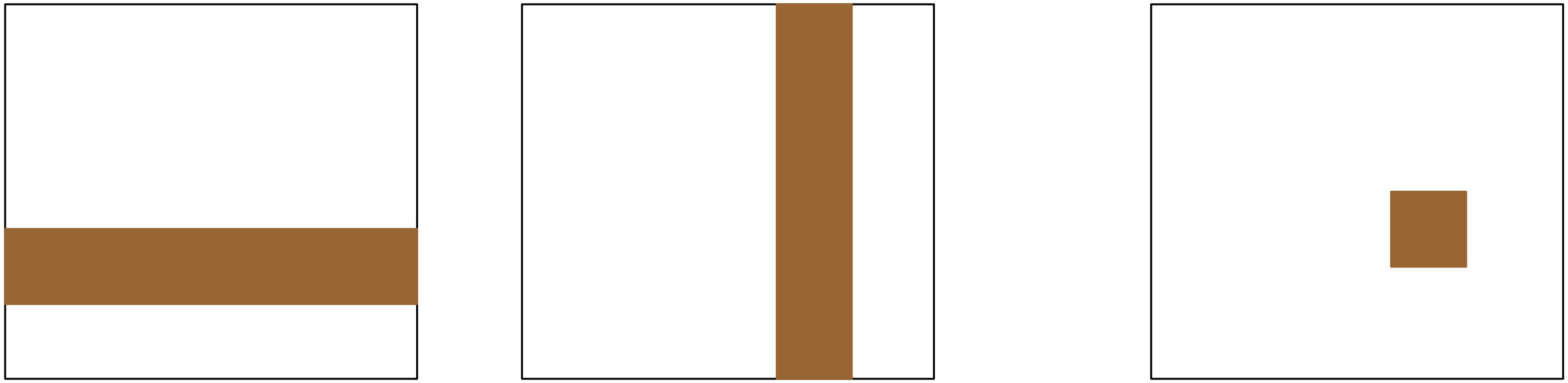
N_w : No. of warps/blocks

N_t : No. of threads/block

How to Obtain $C_T(K)$

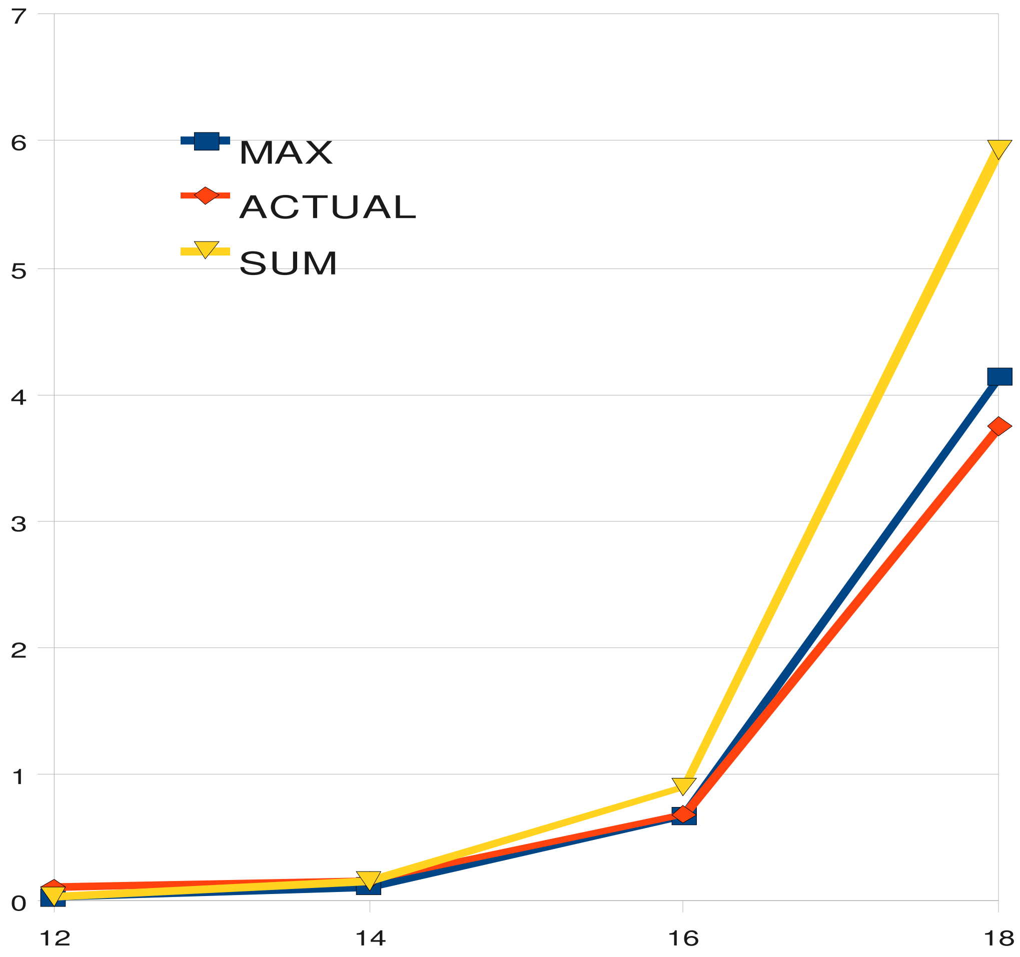
- Separately computing the compute cycles (N_{comp}) and the memory cycles (N_{memory}).
- Which of these dominates the other?
  - Depends on several factors: application, implementation, scheduling, ...
- But can use two possible scenarios
  - Application has good latency hiding: Take MAX
  - Poor latency hiding: Take SUM
- Each scenario gives rise to a model
  - The models place upper and lower bounds on the runtime.
  - Illustrated further in the case studies that follow.

Case Study 1 – Matrix Multiplication



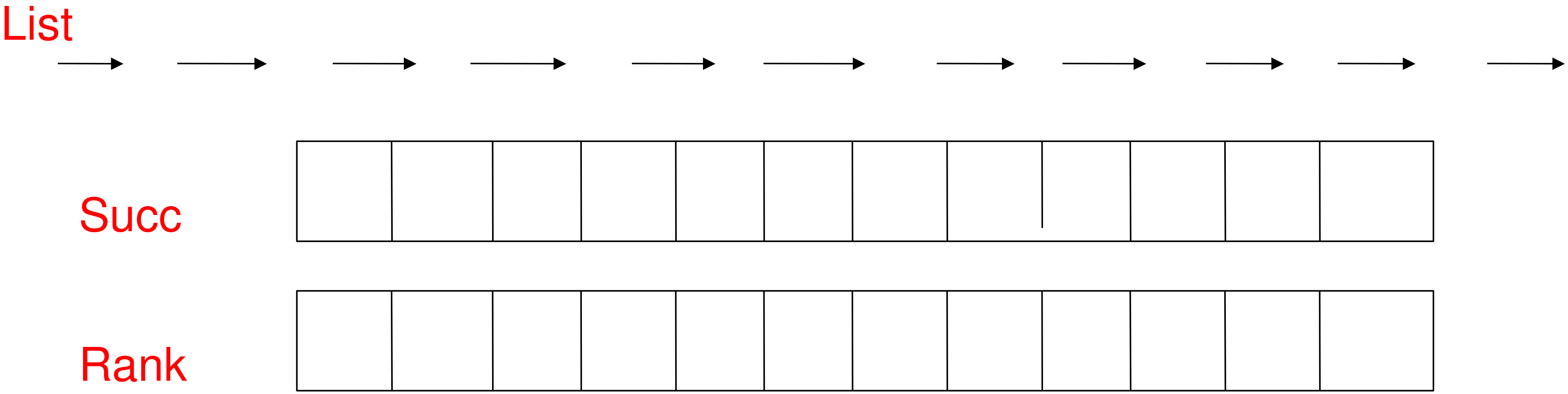
- A popular parallel computing case study
- Typically, compute intensive benefiting from coalesced reads and shared memory accesses.
- Each thread computes a block of the product matrix.
- See Nvidia manual for more details.

Case Study 1 – Matrix Multiplication



- Actual time mostly follows MAX at most places.

Case Study 2 – List Ranking



List

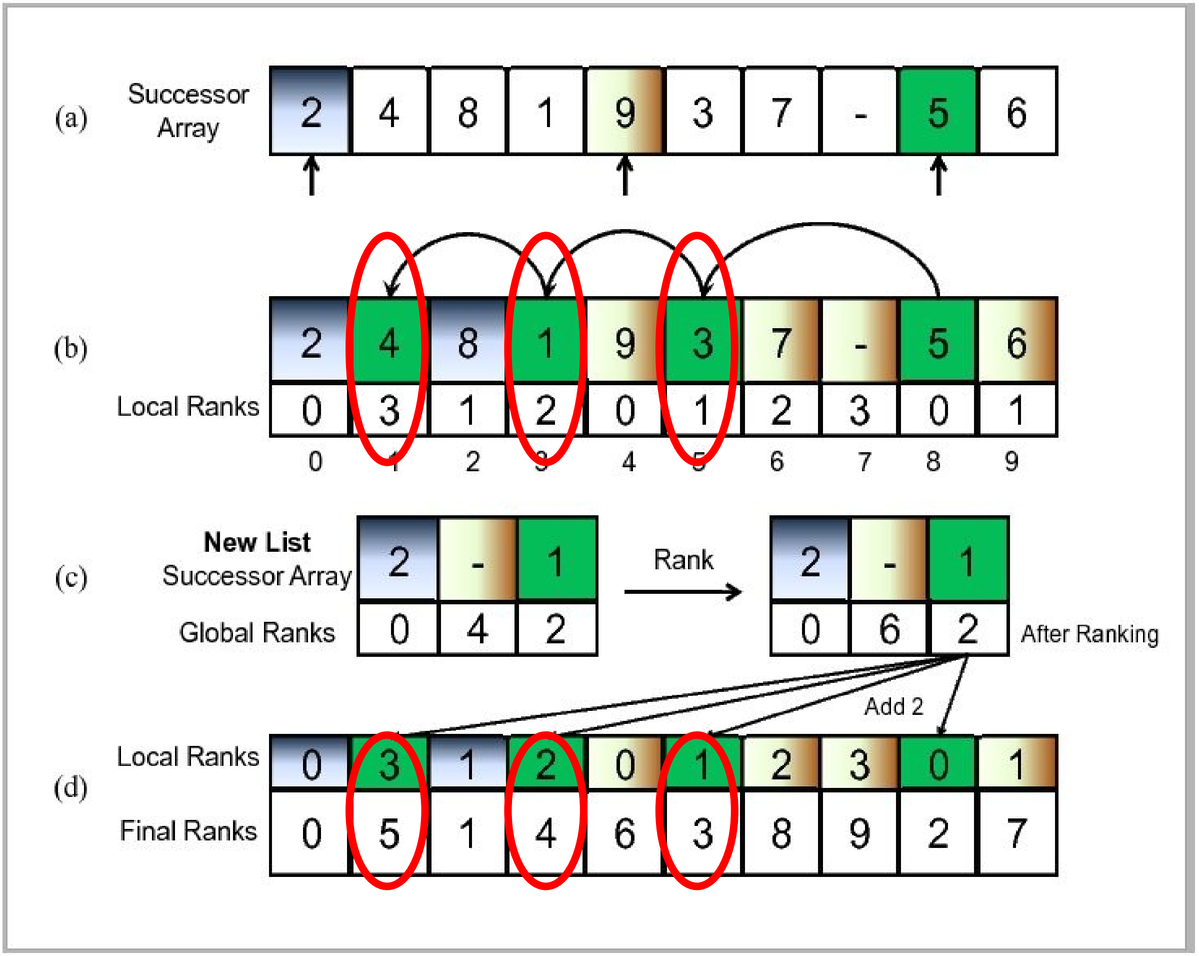
Succ

Rank

- Given a list of elements, as a successor array, find the distance of the elements from one end of the list.
- A fundamental primitive for parallel computing.
- In sequential computation, not a serious problem.

Case Study 2 – List Ranking

- Use a recursive variation to the algorithm of Hellman-JaJa (HJ).
  - HJ Specifically designed for symmetric multiprocessors.
- Basic idea is to sub-lists and compute local ranks.
- Finally rank a smaller list and compute global ranks.
- [Rehman et al. 2009] for more details.



(a)

Successor Array	2	4	8	1	9	3	7	-	5	6

(b)

	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1
	0	1	2	3	4	5	6	7	8	9

(c)

New List Successor Array	2	-	1
Global Ranks	0	4	2

Rank →

	2	-	1
After Ranking	0	6	2

Add 2

(d)

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	5	1	4	6	3	8	9	2	7

Pseudocode for RHJ

The local ranking kernel

```
while(p>=0){
   temp=p;
   p=SUC[p]; // read uncoa
   SUC[temp]=-(index); // write uncoa
   VAL[temp]=++prefix; // write uncoa
   count++;
}
```

Analyzing the Local Ranking Kernel

- Local ranking kernel: Each thread has three read/writes and one compute.
  - All three read/writes are uncoalesced.
  - Per element ranked locally
    - Ncomp = 4
    - Nmemory = 1500
  - How many elements does a thread rank?
    - Not deterministically fixed.
    - Need to analyze probabilistically.

Analyzing the Local Ranking Kernel

- Some observations:
  - Let input be chosen uniformly at random
  - Estimate the probability that a thread ranks k elements locally.
  - Similarly, what is the number of elements ranked by a thread with high probability.
- Analysis suggest that the answer is close to 4 log N elements w.h.p. when using N/log n splitters.

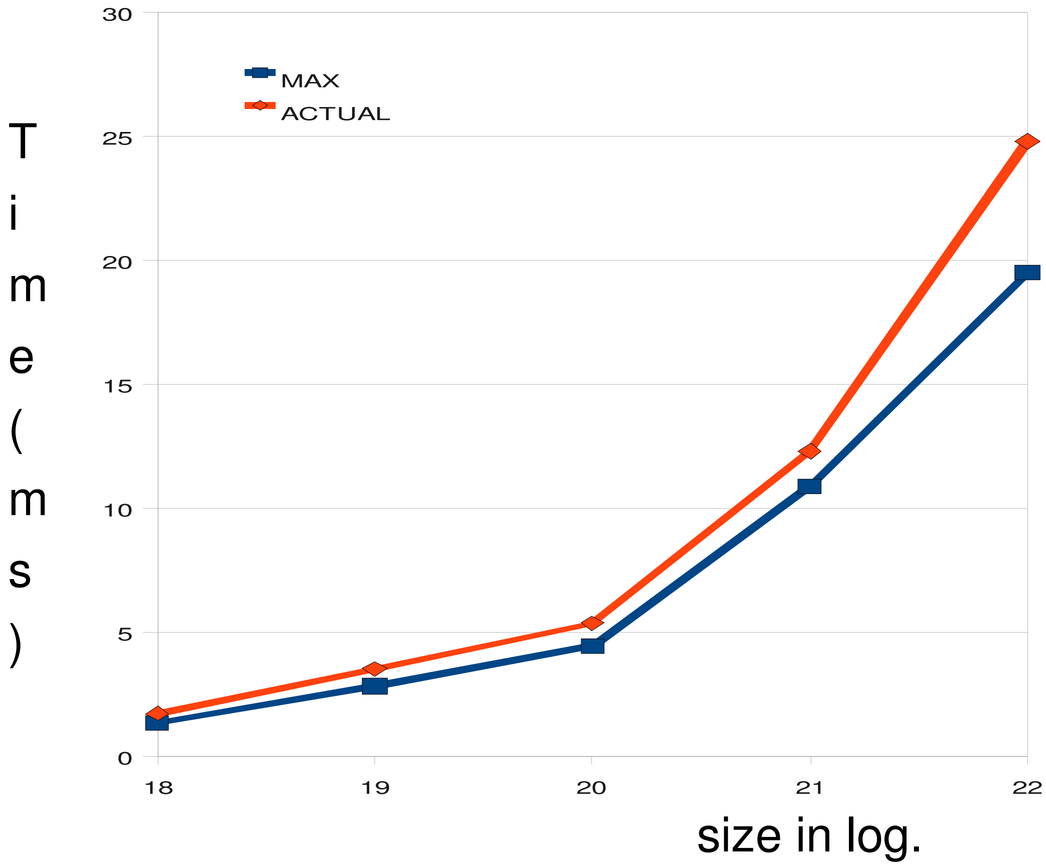
Analyzing the Local Ranking Kernel

- Therefore, time taken can be estimated as:

(N/512x30xlog N) x 16 x (32/8x4) x 4log N x 1500 cycles.

- At N = 2^{22}, this is about 21 ms.

Case Study 2 – List Ranking



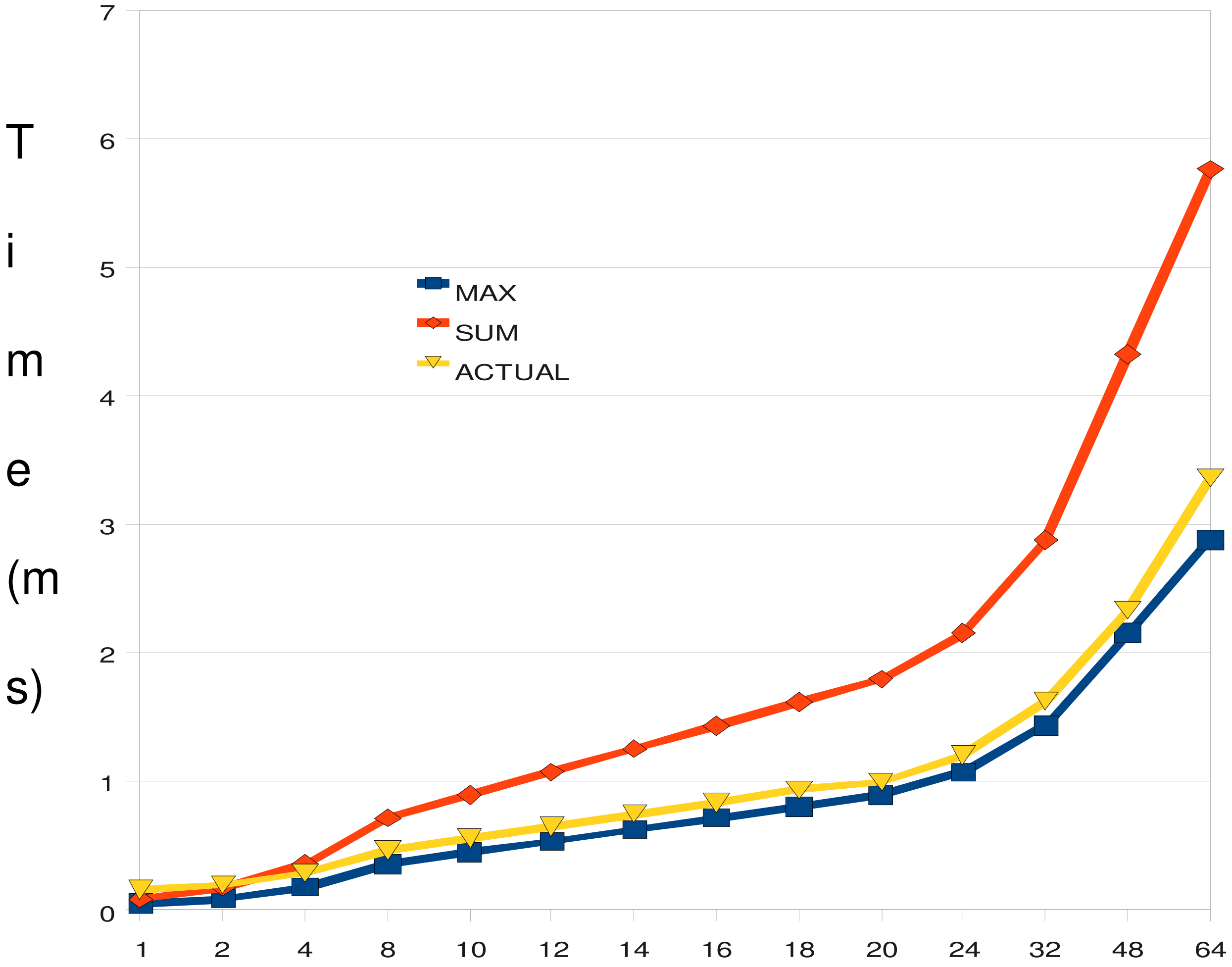
- Each thread ranks a sublist.
- Very little computation.

Case Study 3 – Histogram

- A popular primitive in statistics

- Count all like items.
- Implementation scheme [Patidar and Narayanan, 2009]
  - N data points
  - Each thread works with a set of data points and generates a local histogram.
  - Each thread then updates the global histogram

Case Study 3 – Histogram



- Each thread reads one element from the global memory and updates a counter in shared memory.

The Proposed Model

- An attempt to bridge the gap between theory and practice of GPGPU.
- The model is easy to apply and is reasonably good.
  - More case studies in other reports, e.g., AES
- But has a few drawbacks
  - Atomic operations
  - Divergence of threads, especially dynamic divergence
  - Intra block synchronization
- In future, wish to develop a software simulator for the GPU.