The GPGPU Phenomenon : Understanding its Scope, Applicability, and its Limitations

Kishore Kothapalli
kkishore@iiit.ac.in
International Institute of Information Technology
Hyderabad, India 500 032

The Need for Parallel Computing

- Parallel Programming is slowly emerging as a main-stream topic given its requirement.
- Great emphasis in curriculum and research.
- Lots of interesting and important applications to current problems
  - Search – imagine searching in videos, apart from text
  - Weather modelling, life sciences – real time weather prediction is quite intensive computationally
  - Digital multimedia and special effects – A 90 minute movie can take up to 10^{19} floating point operations.
- Issues can be fundamentally different!
- Hence, a focussed effort to understand and apply is needed.
- Several problems of practical interest, ranging from

Synopsis

- GPUs as the main-stream computing platform.
  - Can deliver up to 1 Teraflop at low price.
  - 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 several areas
  - But, applications need to re-interpreted in massively-multithreaded form to work on GPUs.

Synopsis

- Sample success stories
  - Sorting [Vineet and Narayanan : HPG 2009]
    - can now sort 16 M elements in under half a second.
    - a variant of radix sort
  - Several on image processing, eg. FFT
  - Graph connected components
    - Can process a 10 M vertex, and 60 M edges graph in half a second.
    - a variant of a popular parallel (PRAM) algorithm
  - Many such success abound
    - See SC 2010, GTC 2010, and major international conferences

Synopsis

- Researchers and practitioners alike want to understand this phenomenon.
- Important questions
  - The scope and applicability of GPGPU
  - Algorithmic implications
  - The limits of GPU computation
  - Some future trends

About this Tutorial

- Help beginners understand the GPGPU phenomenon.
- Illustrate the GPU architectural model and its features.
- Explore the GPGPU programming model
- Study a few examples including algorithmic and engineering aspects.
- Touches upon future directions.

About this Tutorial

- Basic knowledge of computer architecture, programming, and algorithms is assumed.
- Knowledge of parallel algorithms and parallel programming helpful, but not assumed.
- Efforts will be made to provide a quick review of concepts required.

Schedule

0:00 – 0:10 : Introductory remarks

0:10 – 0:30 : Basics of computer architecture

0:30 – 0:55 : GPGPU architectural features

0:55 – 1:20 : GPGPU programming

1:20 – 1:50 : Regular algorithms on the GPU

S H O R T B R E A K

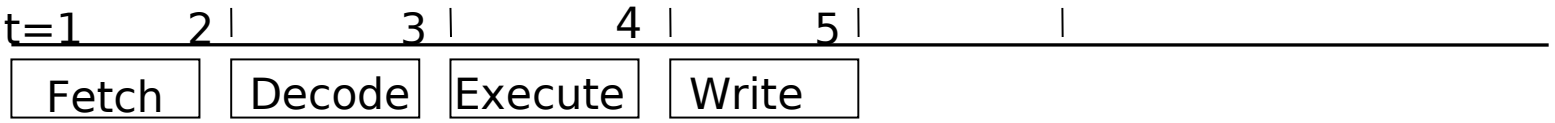
2:10 – 2:40 : Irregular algorithms on the GPU

2:40 – 3:00 : Limitations of GPUs

3:00 – 3:20 : Future trends and directions

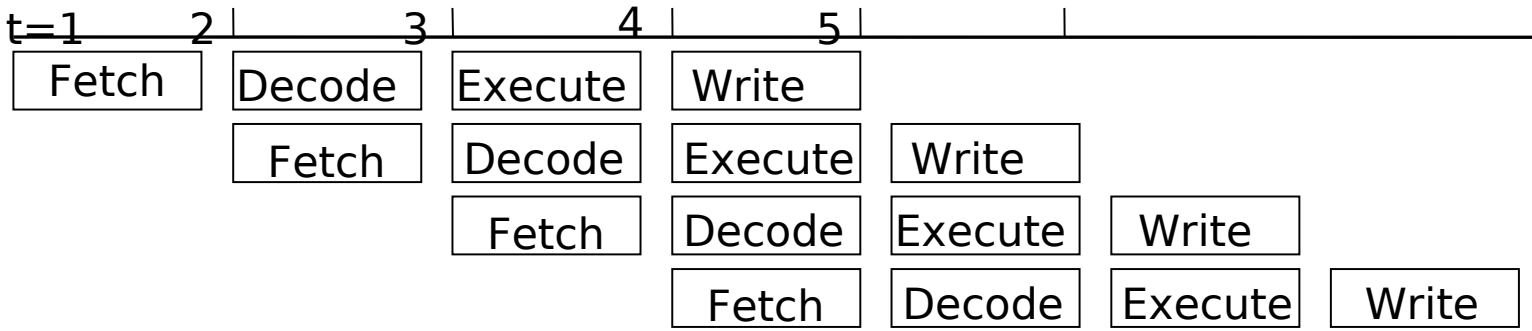
3:20 – 3:30 : Open discussion

Basics of Computer Architecture



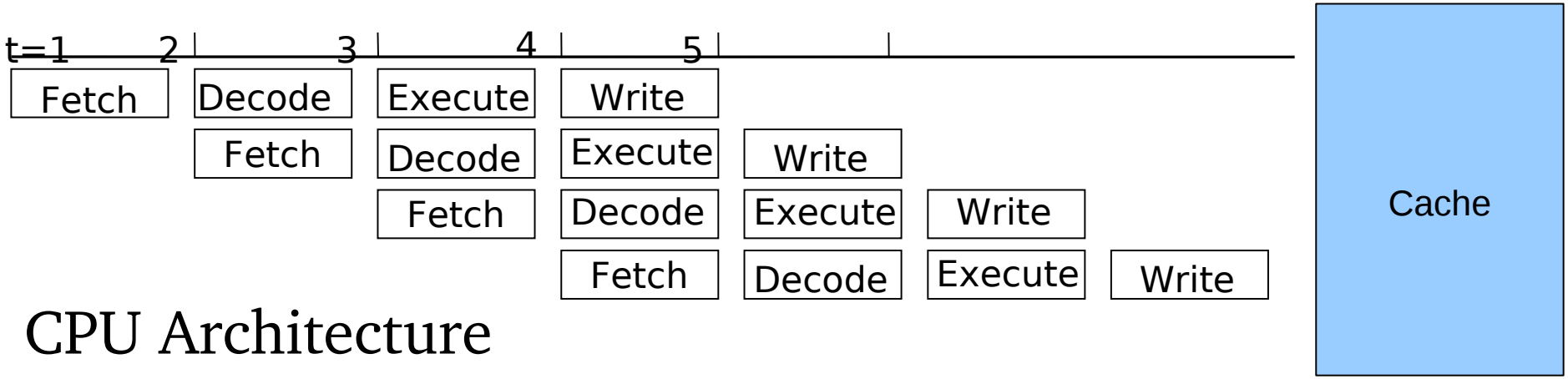
- CPU Architecture
  - 4 stages of instruction execution
    - Too many cycles per instruction (CPI)

Basics of Computer Architecture



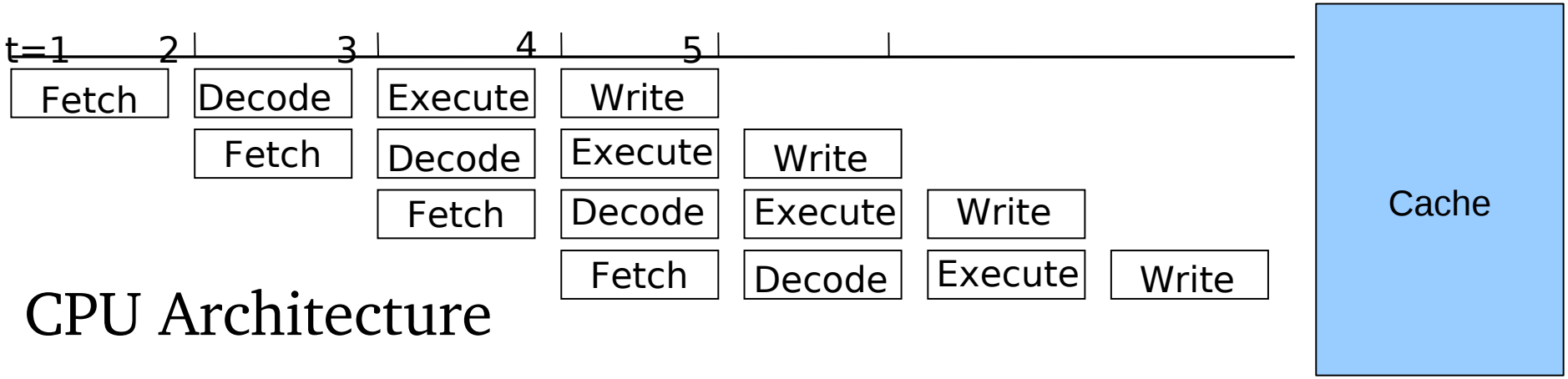
- CPU Architecture
  - 4 stages of instruction execution
    - Too many cycles per instruction (CPI)
  - To reduce the CPI, introduce pipelined execution

Basics of Computer Architecture



- 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

Basics of Computer Architecture



- 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 (1 PPU + 8 SPU)
- NVidia GeForce 8800 includes 128 scalar processors, Tesla, and Fermi (~500 cores)
- Sun T1, T2, and T3 (16 cores, 128 threads)
- Tilera Tile64 (64 cores, 100 cores in a mesh network)
- Picochip combines 430 simple RISC cores (multicore DSP)
- Cisco 188
- TRIPS (Tera-op, Reliable, Intelligently adaptive Processing System)

Why GPUs?

- Given the wide choice as evident from the last slide, why are GPUs so popular?
- Several reasons
  - Low cost and power usage
  - Comparable peak performance
  - Todays PCs already have a GPU card, used primarily for graphical functions.
  - While GPUs may not be suitable for all operations, should use them for appropriate tasks.

Evolution of GPUs

- Hardware: 8 – 16 cores to process vertices and 64 – 128 to process pixels by 2005.
  - Vertices and pixels are important stages in graphics processing
- Less versatile than CPU cores
- SIMD mode of computations.
- Less hardware for instruction issue
- Can pack more cores in same silicon die area
- No caching, branch prediction, out-of-order execution, etc.

More about GPUs

- GPGPU – General Purpose Computing on GPUs
- In their early years, can be programmed using OpenGL.
  - Difficult however.
- Present generation GPUs come with a programmable interface.
  - For instance, Nvidia supports a C-like interface called CUDA.

CPU Vs. GPU

- Fundamentally, there are differences in design philosophies.
- Few powerful cores vs. lots of small cores.
- No system managed cache in GPUs,

GPGPU Tools and APIs

- CUDA – A programmable interface to Nvidia GPUs
- OpenCL – Open Compute Language : An abstraction for several programming environments
- OpenGL – Programming interface for early generation Nvidia GPUs
- Brook – A programmable interface to AMD GPU

Schedule

0:00 – 0:10 : Introductory remarks

0:10 – 0:30 : Basics of computer architecture

0:30 – 0:55 : GPGPU architectural features

0:55 – 1:20 : GPGPU programming

1:20 – 1:50 : Regular algorithms on the GPU

S H O R T B R E A K

2:10 – 2:40 : Irregular algorithms on the GPU

2:40 – 3:00 : Limitations of GPUs

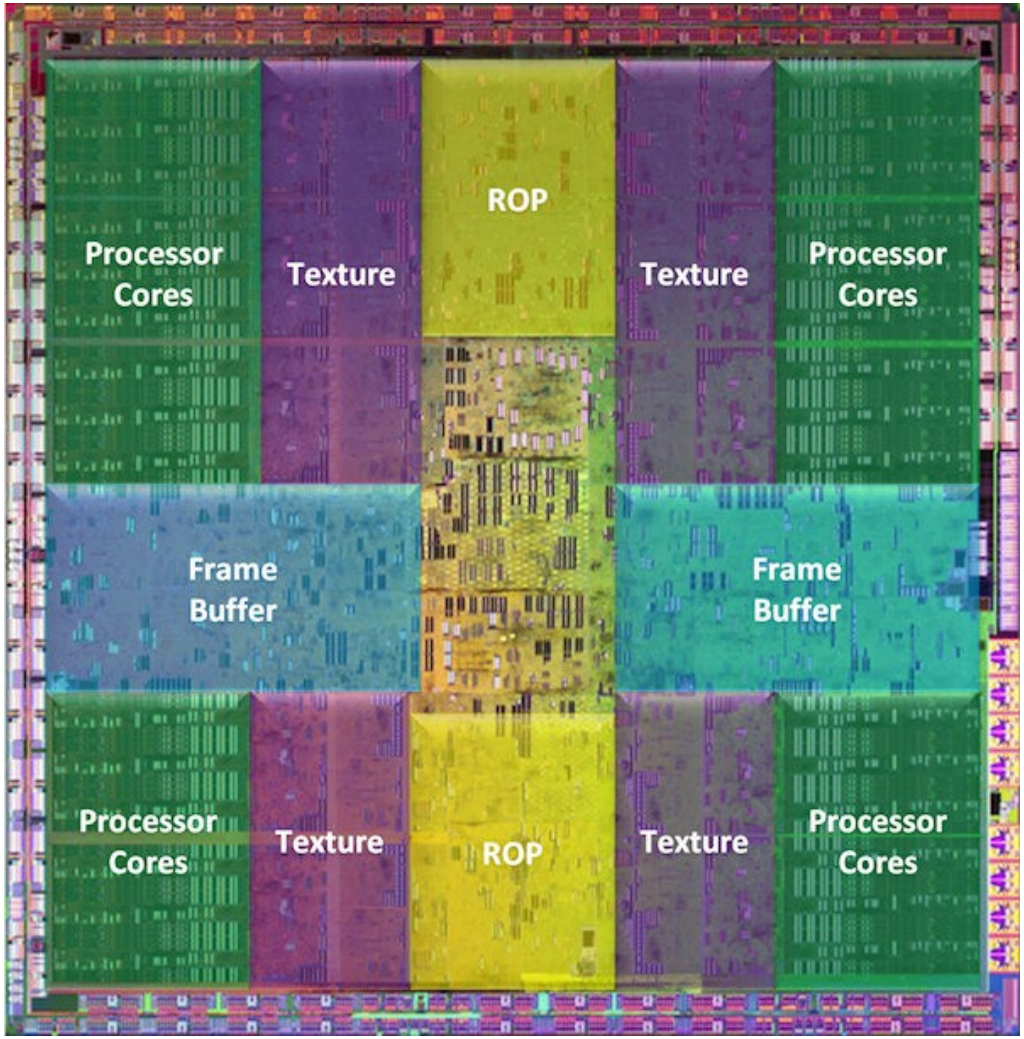
3:00 – 3:20 : Future trends and directions

3:20 – 3:30 : Open discussion

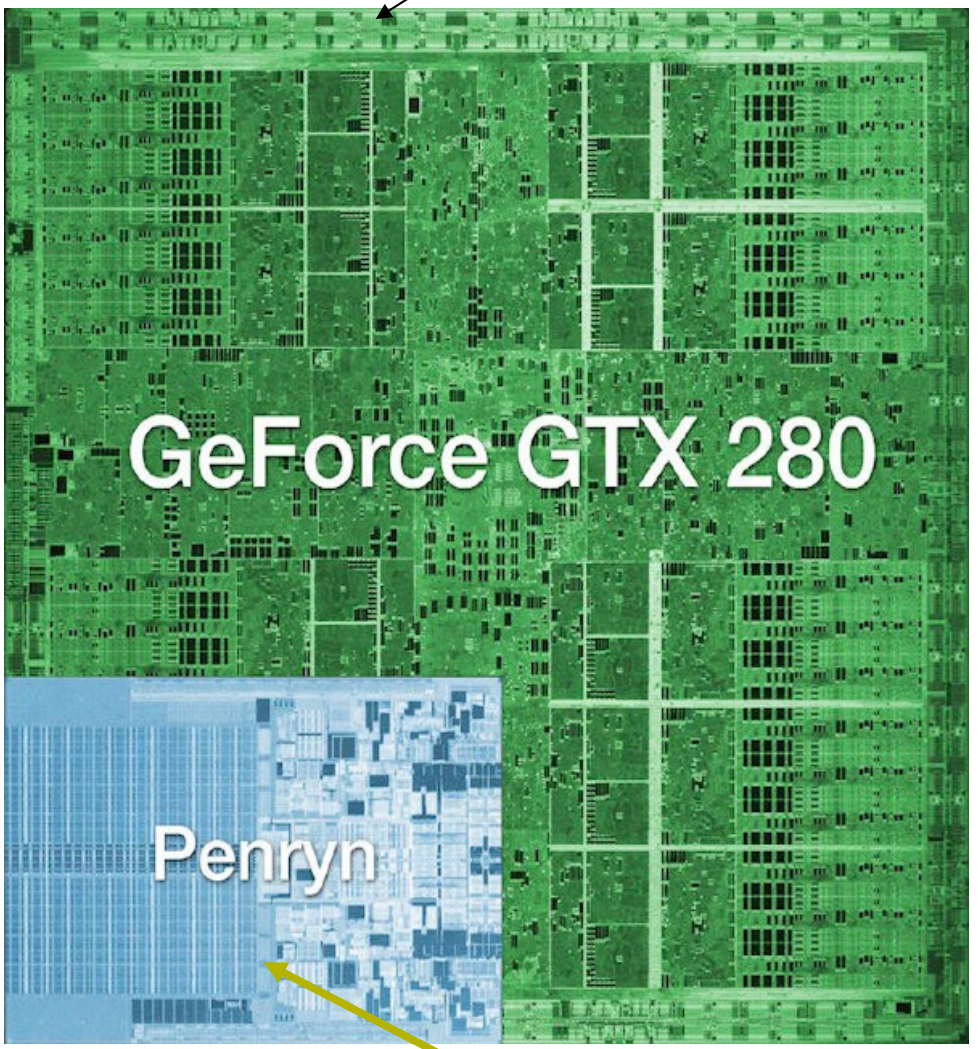
GPU Architectural Features

- We'll focus on one of the latest offerings from Nvidia, the GTX 280, or the Tesla C1070.
- A general description that is applicable to other offerings from Nvidia.

The GTX 280



576 mm² area



110 mm² area

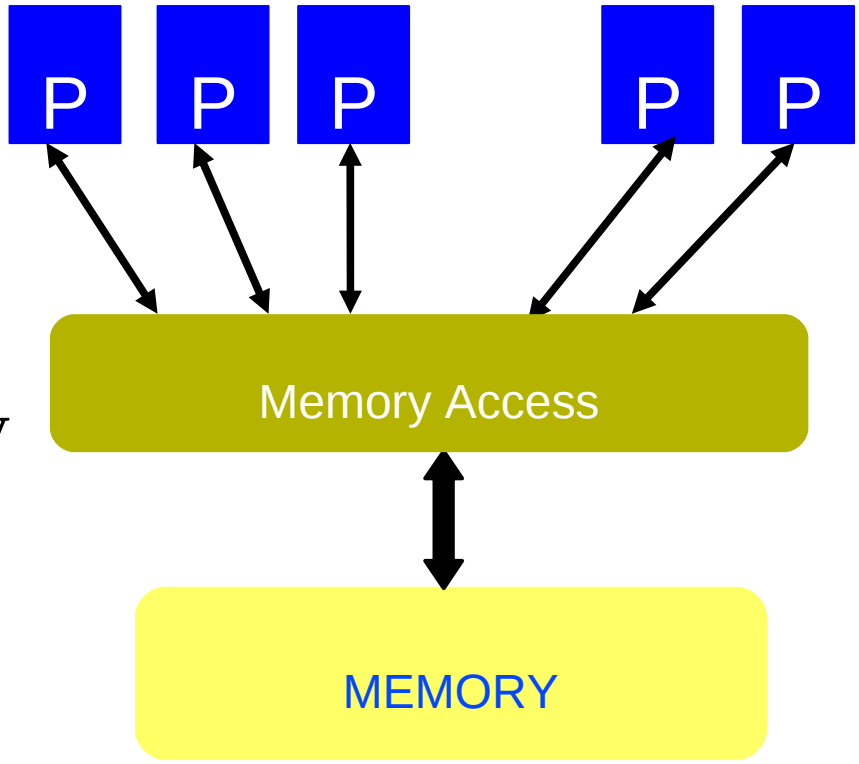
Intel dual core CPU

The GTX 280

- Large chip area of close to 600 sqmm.
- May induce several manufacturing defects, but none noticed so far.
- Thermal design power of 236 W
  - Idle power consumption at 25 W
- Power consumption is not too high.
  - For instance, Tianhe-I would have needed three times more power, at 12 MW, if built using only CPUs.
- Several other power modes also available at varying power consumption levels.

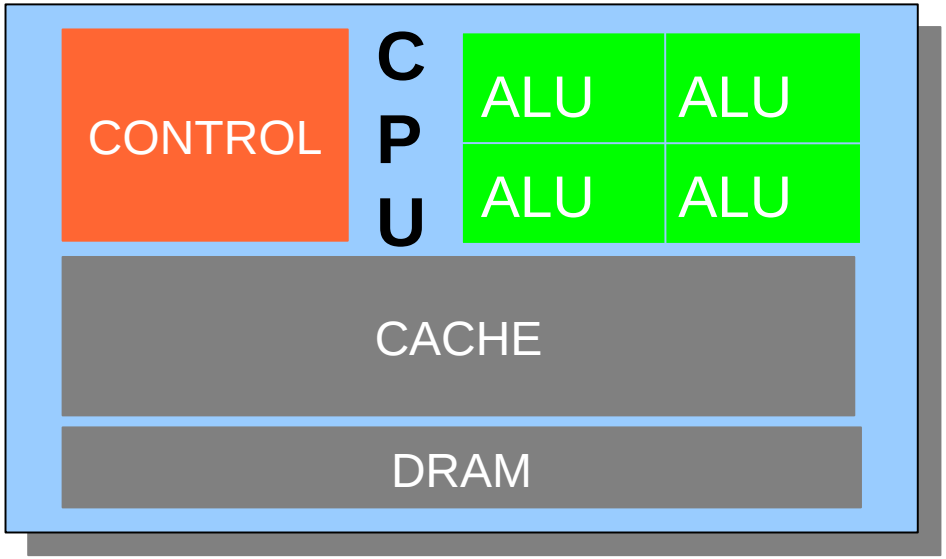
Basics

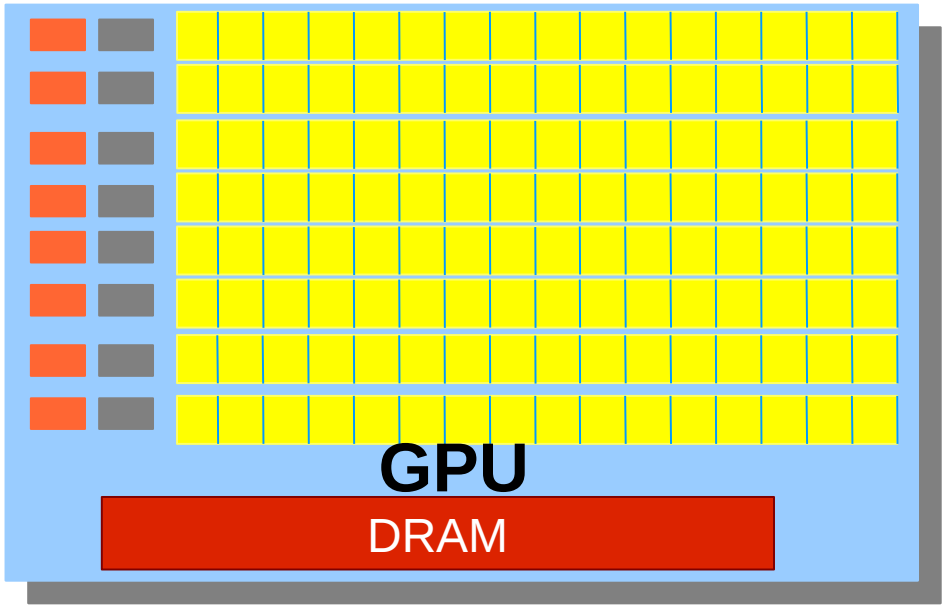
- Processors have no local memory
- Bus-based connection to a common, large, memory
- Uniform access to all memory for a PE
  - 500 times slower than computation
- Resembles the PRAM model!
- No caches. But, instantaneous locality of reference improves performance
  - Simultaneous memory accesses combined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



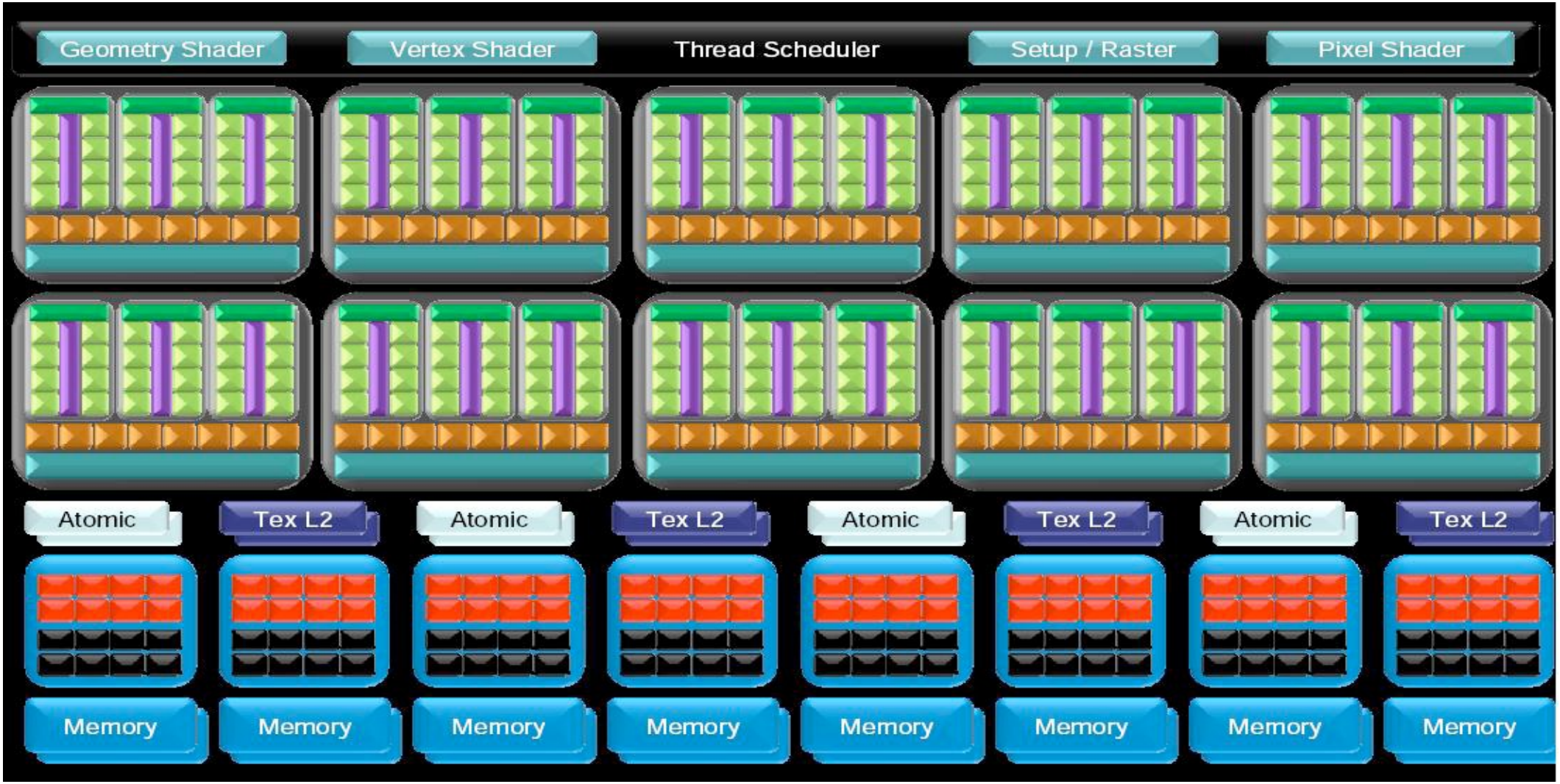
GPU Architecture in Contrast to CPU

- 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 such as Graphics rendering, image/signal processing, matrix manipulation, FFT, etc.





NVidia GTX 280 Architecture

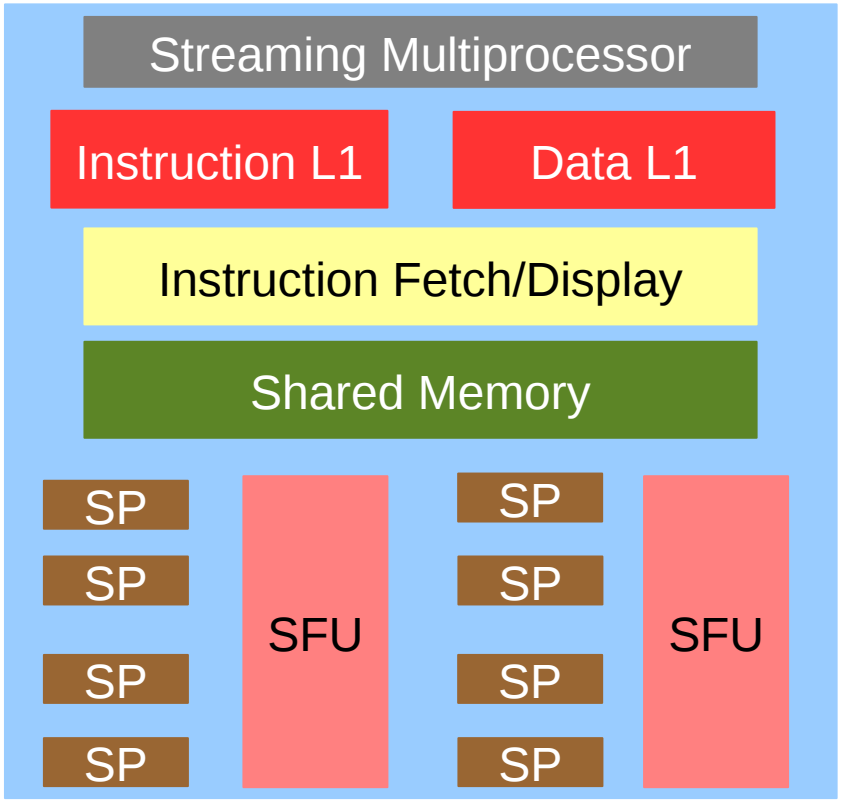


High Level Model

- Streaming multiprocessors (SMs) each of which is a collection of 32-bit processors.
- Each SM runs in a (Single Instruction Multiple Data (SIMD) mode.
- Devices have multiple SMs, current generation have 30 of these SMs.

Streaming Multi-Processors

- 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



More on GPU Memory Hierarchy

- GPU has several types of memory.
  - Global memory
  - Registers
  - Shared memory
  - Texture
  - Constant
- Understanding these memory types and their properties is important for application performance.
- Before understanding the memory model, need some concepts on threads, etc.

A Small Detour – Threads, Warps and Blocks

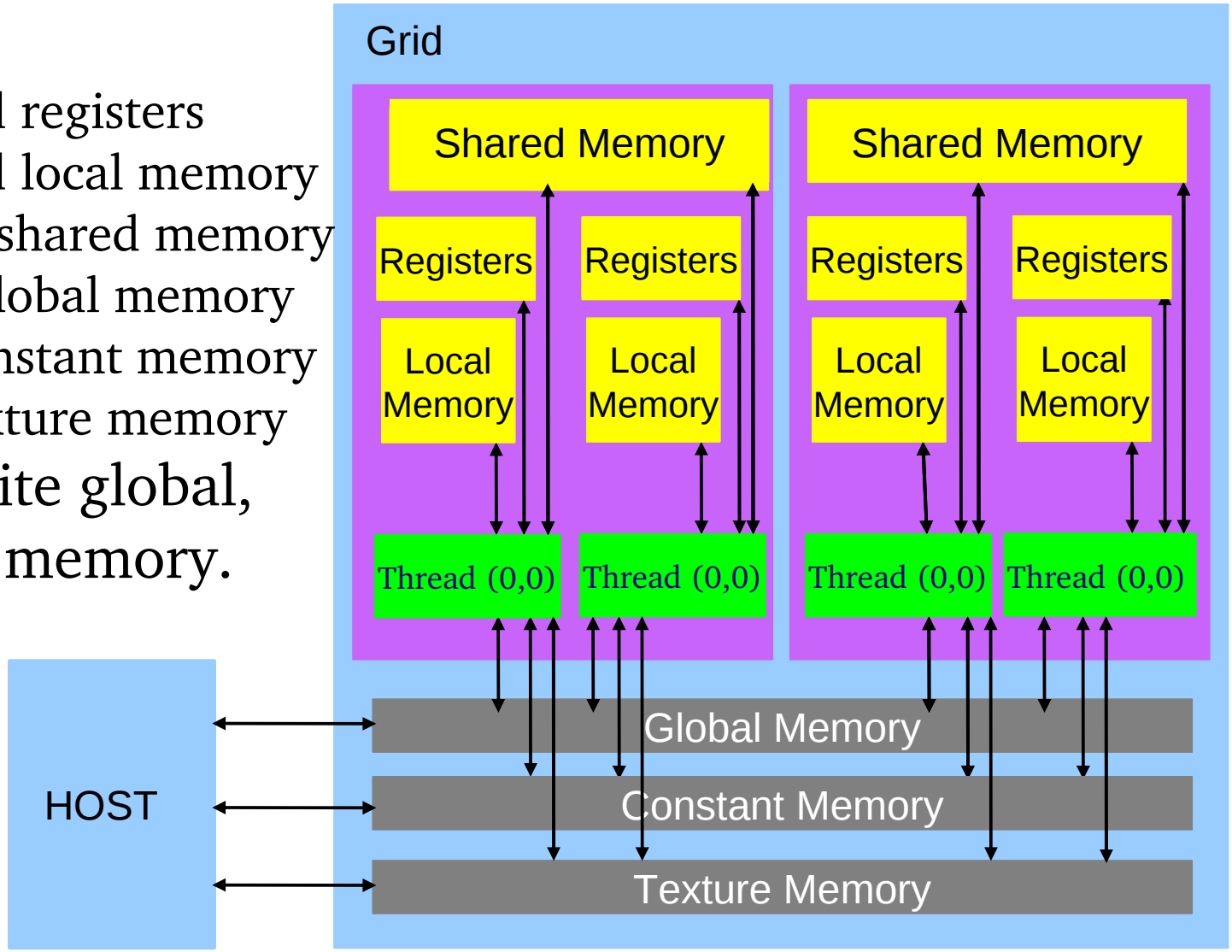
- A thread is the basic unit of computation.
- 32 threads in a Warp or a scheduling group
  - Only <32 when there are fewer than 32 total threads
- There are (up to) 16 Warps in a Block
- Each Block (and thus, each Warp) executes on a single SM
- GTX 280 has 30 SMs
- At least 30 Blocks required to “fill” the device
  - More is better
  - If resources (registers, thread space, shared memory) allow, more than 1 Block can occupy each SM

What if Threads Need to Communicate?

- Inter-thread communication is essential to parallel processing.
  - Rarely is there computation which does not require communication between threads.
- This inter-thread communication could be by messages, or by sharing memory cells.
- The former is quite difficult to implement compared to the latter.
  - GPUs have the latter feature.

What can Threads Do to Communicate?

- 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.



Registers

- Nvidia GTX 280
  - 16 K registers each 32 bit wide
  - Registers shared across threads in a warp
  - High speed access, just like registers in a CPU.

Global Memory

- Nvidia GTX 280
  - 1 GB of common, off-chip global memory
  - 130 GB/s of theoretical peak memory bandwidth
- 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

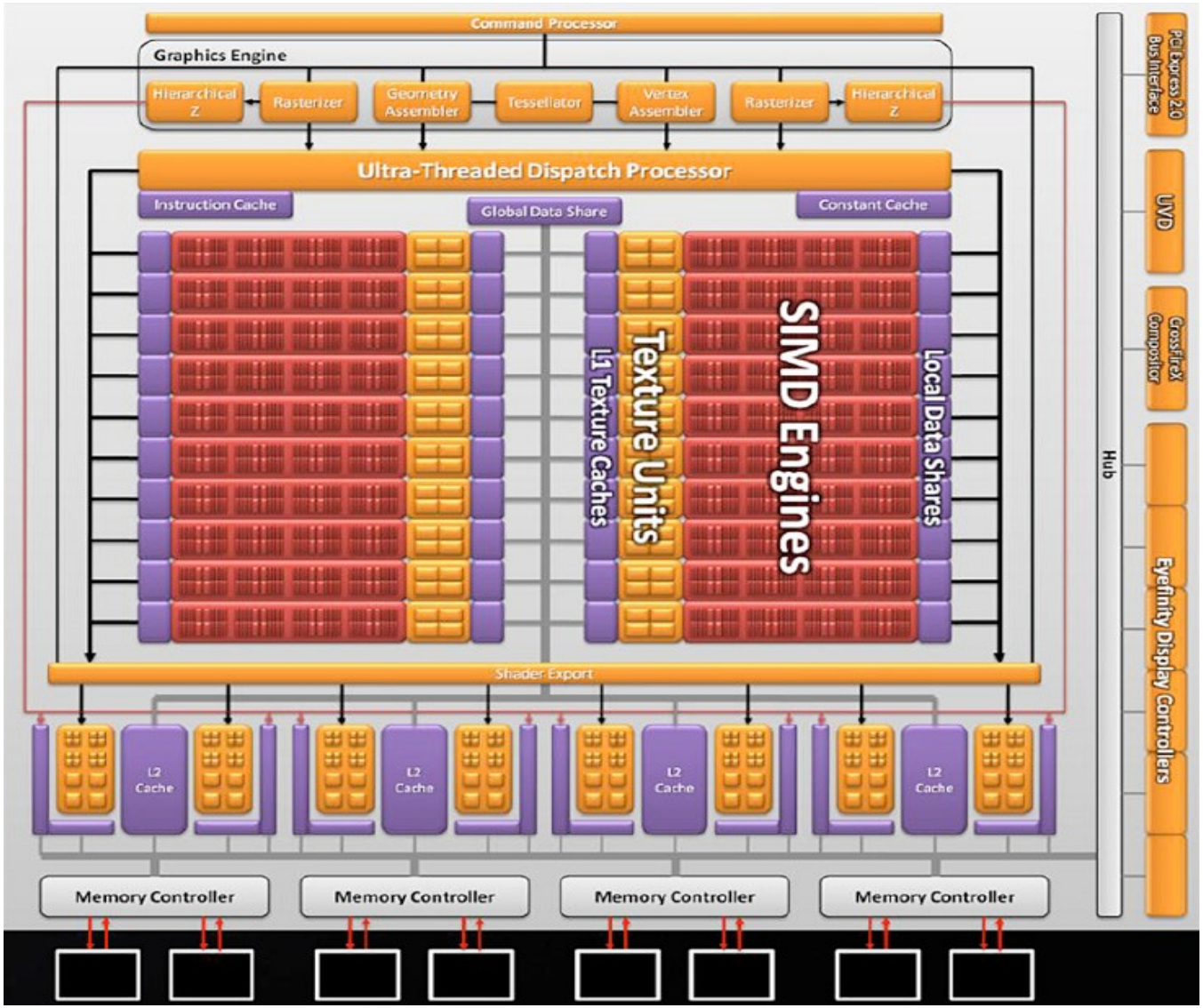
Shared Memory

- NVidia GTX 280
  - 16 KB of shared memory per block of threads
  - 16 KB organized as banks of 1 KB each.
  - Access to banks should be exclusive. Otherwise, requests are queue up.
    - Alike the Queue-Read-Queue-Write PRAM model.
  - Low access cost
  - Typically, to be used for variables used by more than one thread in a warp.

Texture Memory and Constant Cache

- Constant Cache
  - 8 KB per SM
  - All threads in an SM can use it
  - But only readable.
- Texture memory
  - About 6-8 KB per SM.
  - All threads in an SM can access this memory
  - But it is only readable.
- Typical uses
  - Store the vector in matrix-vector computations.

AMD 5870 Architecture



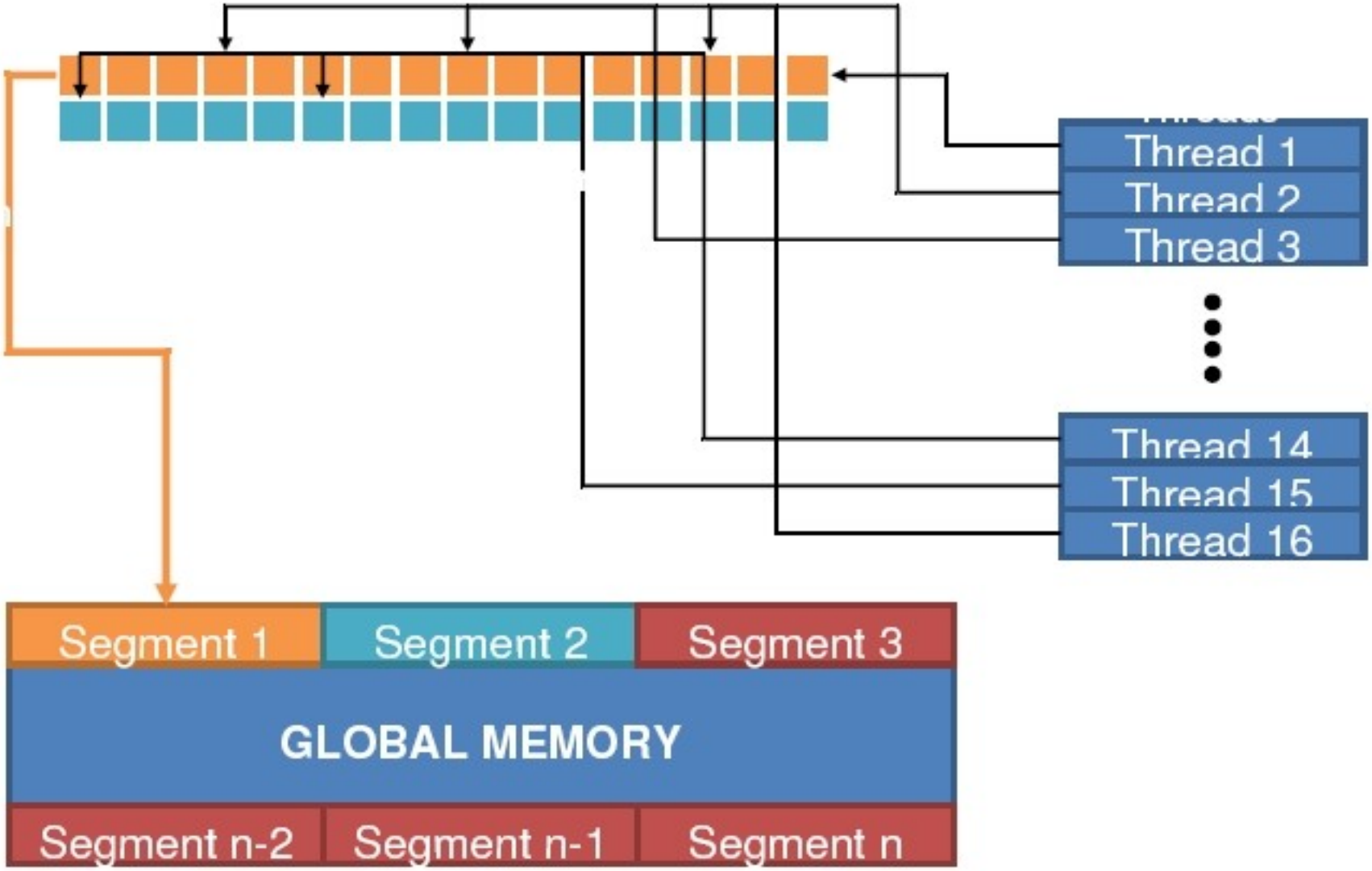
- 20 SIMD engines with 16 Stream cores
- 334 sqmm area
- Each SC with 5 Pes (1600 Pes in total)
- Each with IEEE754 and integer support
- Each with local data share
  - 32 kb shared low latency memory
  - 32 banks with hardware conflict Management
  - 32 integer atomic units 80 Read Address Probe
- 4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

Performance Considerations

- Thread divergence
  - SIMD width is 32 threads. They all should execute the same very instruction
  - Serialization otherwise

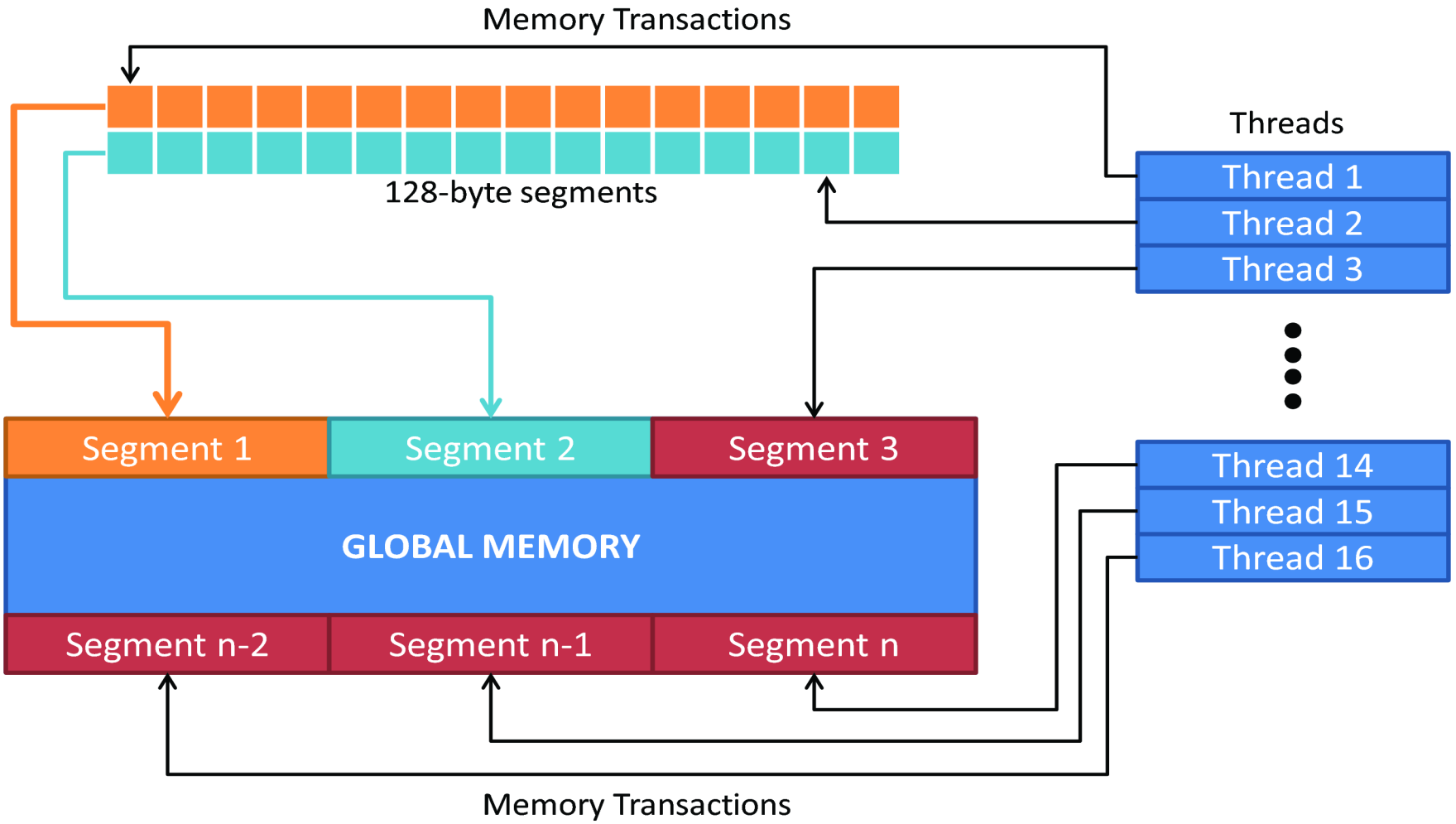
- Memory access coherence
  - A half-warp of 16 threads should read from a local block (128, 64, or 32 bytes) for speed
  - Random memory access very expensive

Performance Considerations



- Coherence of memory accesses
  - Coalesced access is beneficial to threads

Performance Considerations



- Coherence of memory accesses
  - Uncoalesced access is very costly.

Performance Considerations

- 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
  - As these are most likely to be serialized, should be used sparingly.
  - Examples : histogram calculation

Massively Multi-threaded 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-then-else divergence
- Threads organized in multiple tiers

Schedule

0:00 – 0:10 : Introductory remarks

0:10 – 0:30 : Basics of computer architecture

0:30 – 0:55 : GPGPU architectural features

0:55 – 1:20 : GPGPU programming

1:20 – 1:50 : Regular algorithms on the GPU

S H O R T B R E A K

2:10 – 2:40 : Irregular algorithms on the GPU

2:40 – 3:00 : Limitations of GPUs

3:00 – 3:20 : Future trends and directions

3:20 – 3:30 : Open discussion

CUDA Basics

- GPU is typically attached as a device to a CPU.
- The CPU is called the host.
- Code to be run on the GPU is written as kernels
- A group of threads bunched together is called a block (a WorkGroup in OpenCL)
- A group of blocks is a grid.

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

The SIMD Model

- SIMD – Single Instruction Multiple Data
  - Also called as data parallelism
  - Part of Flynn's taxonomy
  - A popular model of parallel execution.
- Data elements provide parallelism
  - Think of many data elements each being processed simultaneously
  - Thousands of threads to process thousands of data elements

CUDA Terms

- An extension to the ANSI C programming Language
  - Easy learning curve

- Language Extensions in form of
  - Function type qualifiers providing a variety of functions
  - Variable type qualifiers providing a types of variables
  - Execution Configuration providing parameters to kernel
  - Built-In variables support for 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 the host
- `__host__`
  - Executed on the host
  - Callable only from host
- For functions executed on the device
  - No support for recursion
  - static variable declarations inside the function not allowed.
  - C-style variable number of arguments not supported

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

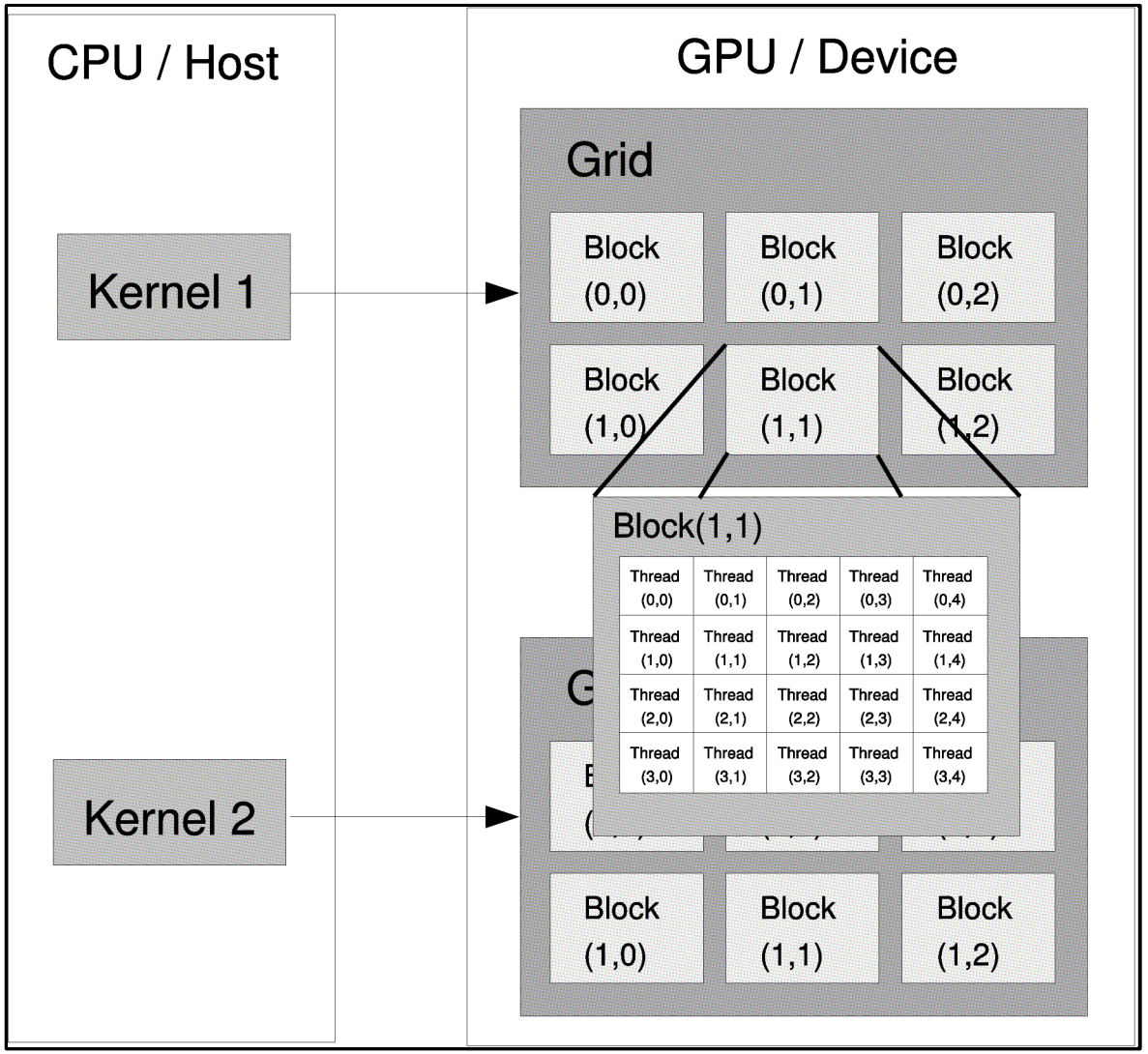
- `gridDim`
  - Variable holding the dimensions of a grid
- `blockIdx`
  - Variable holding the block index within the grid
- `blockDim`
  - Variable holding the dimensions of a block
- `threadIdx`
  - Variable holding the thread index within the block
- Can not assign values to them nor can you get the address of the above variables
  - But useful to locate the data item that a thread has to work on

More on CUDA Kernels, and Grids

- One kernel is executed at a time on the device.
- Many threads can execute a kernel.
- All threads run the same code.
- Each thread has an ID that it uses to compute memory addresses and make control decisions.

GPU Programming Model

- Kernels are run in a one after another mode.
- Kernel has a collection of threads arranged in grids.
- Grid has up to three dimensions.
  - Grid is again a collection of blocks.
  - Block consists of threads, with id for each thread.



Scheduling

- Blocks are assigned to SMs.
  - One block completely processed before starting another block.
  - Multiple blocks can be assigned but only if resources permit.
- Within a block, warps are executed concurrently.
- Can swap warps with zero overhead.
  - But no guarantees on which warp is scheduled when.
  - Scheduling policy is not public knowledge.
- To fill the GPU completely
  - At least 1 block per SM
  - Each block has 16 warps, each of which has 32 threads.
  - GTX 280 has 30 SMs, implying that at least 30x16x32 = 15360 threads to be in flight simultaneously.

A Small Example

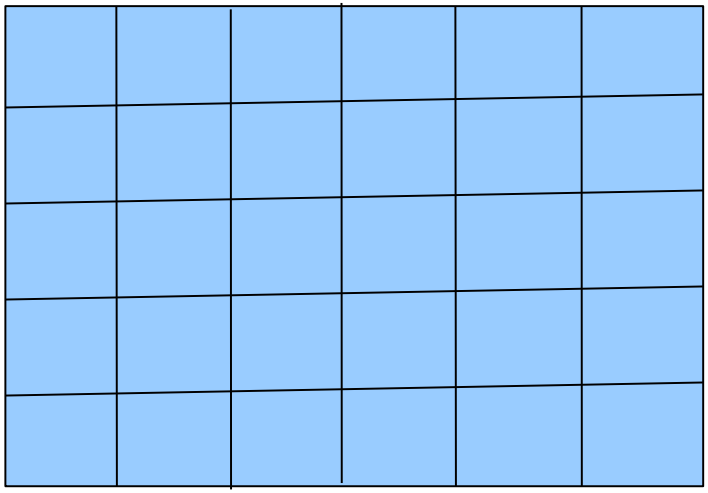
```
void inc_cpu (int*a, intN)
{
      int idx;
      for (idx = 0; idx<N; idx++)
            a[idx] = a[idx] + 1;
}

voidmain() {
…
inc_cpu(a, N);
…
}
```

```
__global__ void inc_gpu(int*a_d, intN)
{
      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);
…
}
```

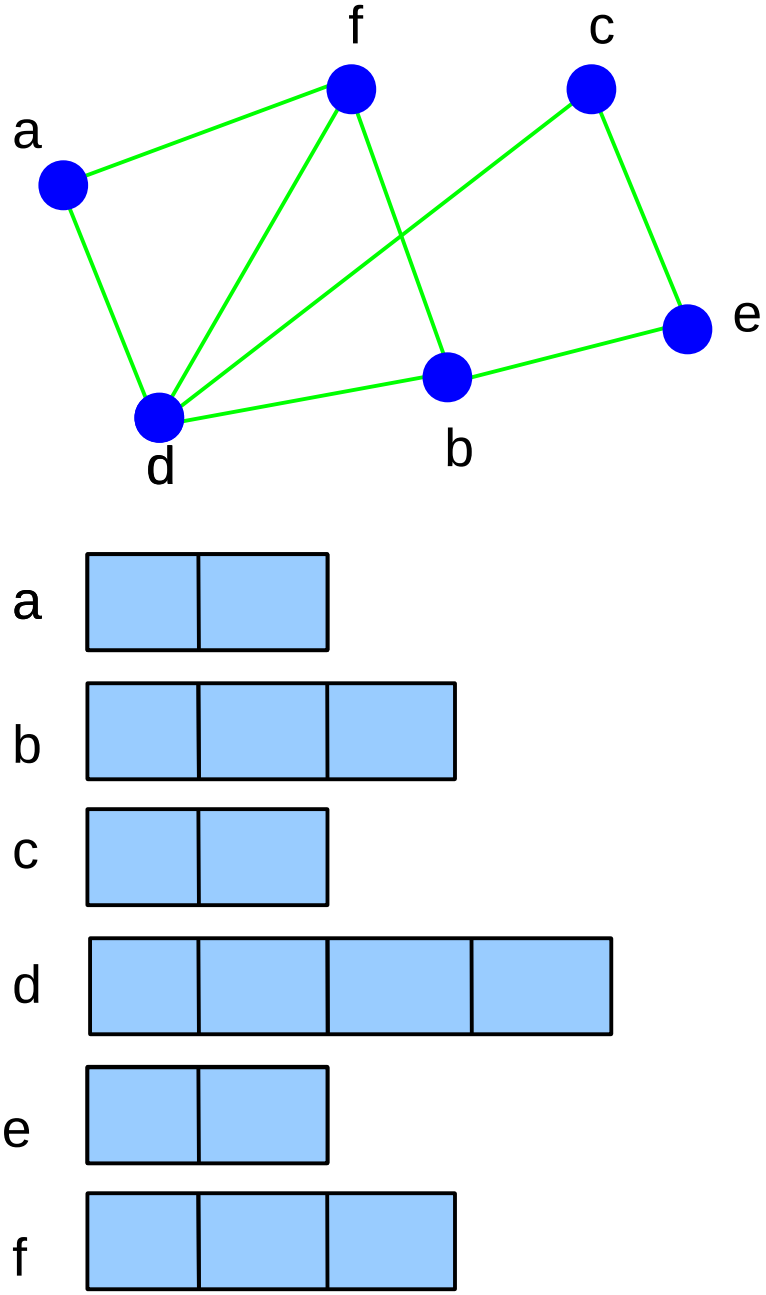
Regular Computations

- 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



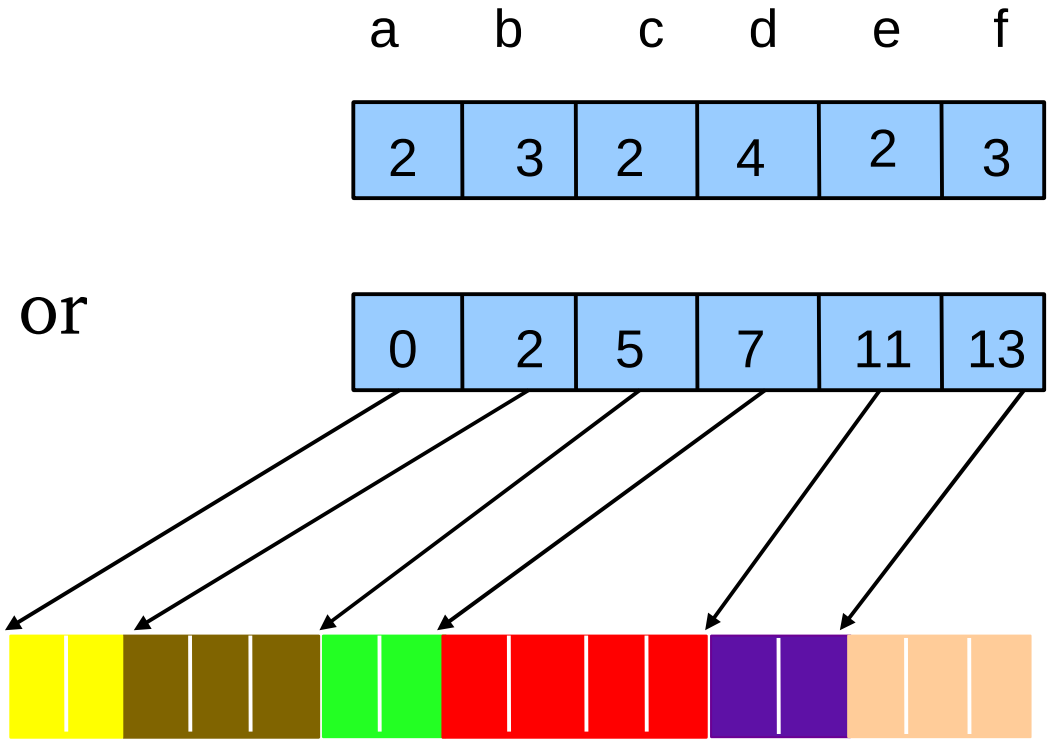
Irregular Domains

- An irregular 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, graph algorithms, sparse matrices, etc.



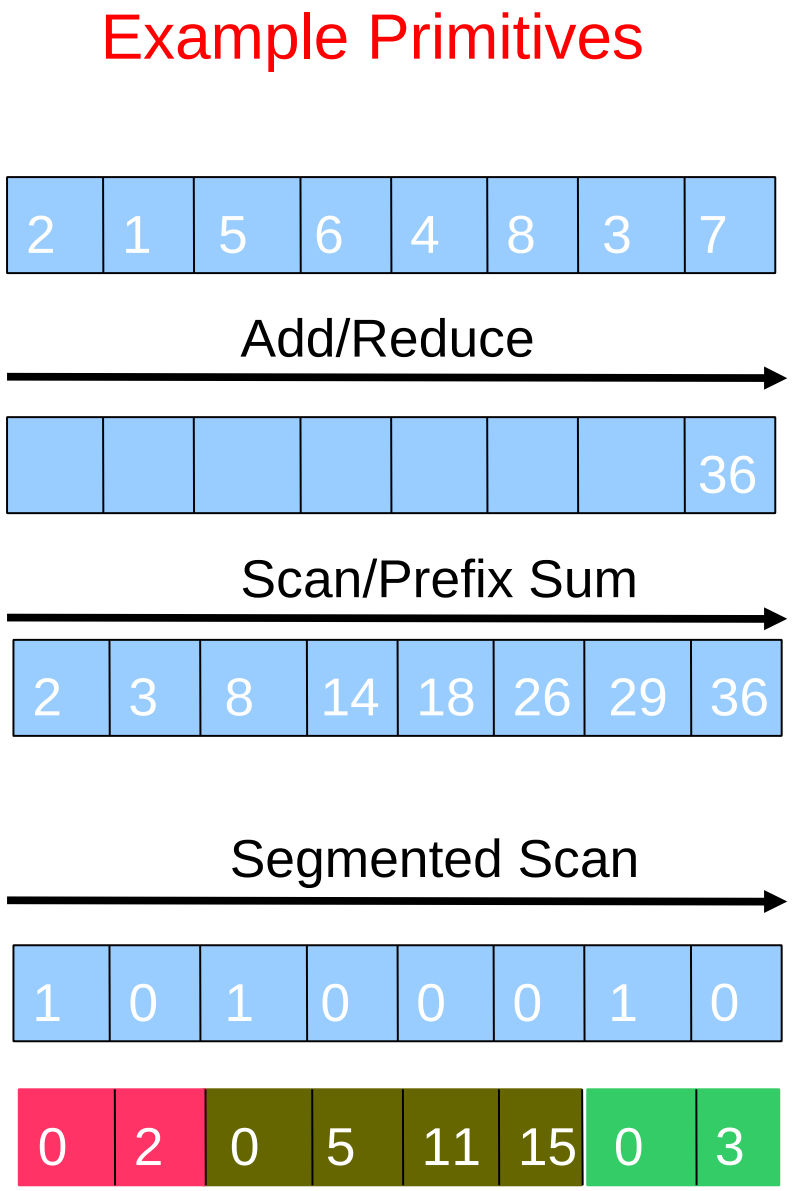
Handling Irregular Domains

- 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.



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]



Schedule

0:00 – 0:10 : Introductory remarks

0:10 – 0:30 : Basics of computer architecture

0:30 – 0:55 : GPGPU architectural features

0:55 – 1:20 : GPGPU programming

1:20 – 1:50 : Regular algorithms on the GPU

S H O R T B R E A K

2:10 – 2:40 : Irregular algorithms on the GPU

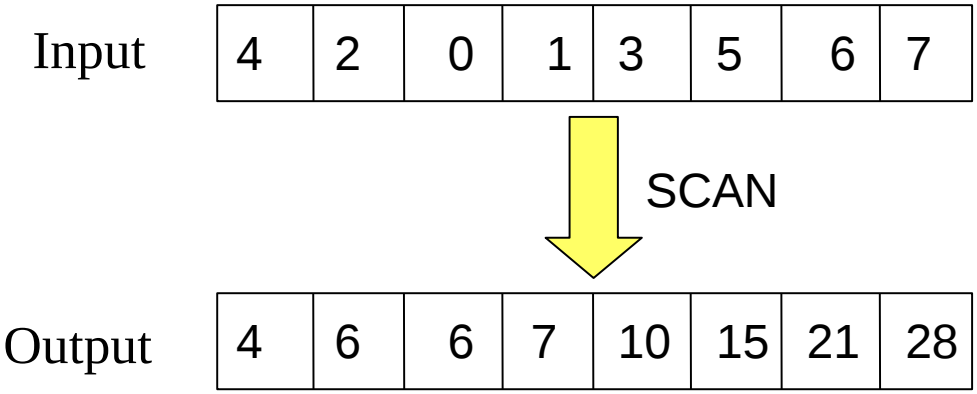
2:40 – 3:00 : Limitations of GPUs

3:00 – 3:20 : Future trends and directions

3:20 – 3:30 : Open discussion

Regular Applications – Scan

- An important primitive in parallel computing.
  - called as prefix operation in PRAM literature.
- Input : An array A of n integers.
- Output : An array S of n integers so that $S[i] = \Sigma_{j=1}^{i} A[j]$.



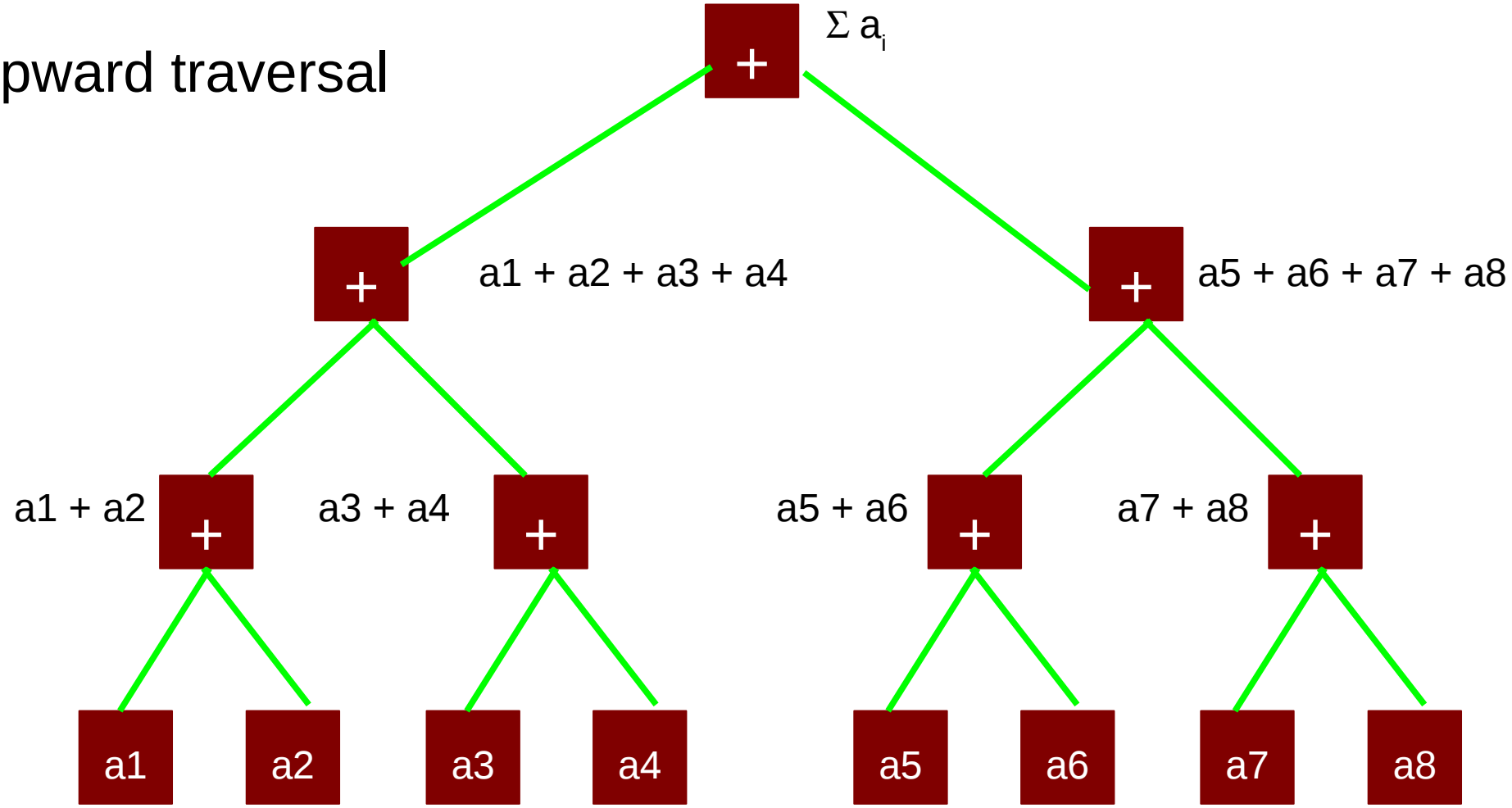
- Very easy to compute in the sequential setting.
- Requires new techniques to solve in the parallel setting.

Solution Algorithmically

- Solution can be designed in the PRAM model as follows.
- Define an array B of size n/2 so that B[i] = A[2i] + A[2i+1].
- Let S_B be the prefix sums of B.
  - So, $S[i] = \Sigma_{j=1}^{i} B[j]$.
- Using S_B, we can compute S as follows.
  - S[1] = A[1]
  - $S[2i] = \Sigma_{j=1}^{2i} A[j] = \Sigma_{j=1}^{i} B[j]$.
  - What about S[2i+1]. Indeed, S[2i+1] = S[2i] + A[2i+1].
  - Once, even indices of S are computed, can compute odd indices by one more addition.
- For more details, see JaJa [Chapter 2]
- Technique has the name called balanced binary tree, with applications to several parallel computations.

Balanced Binary Tree – Prefix Sum

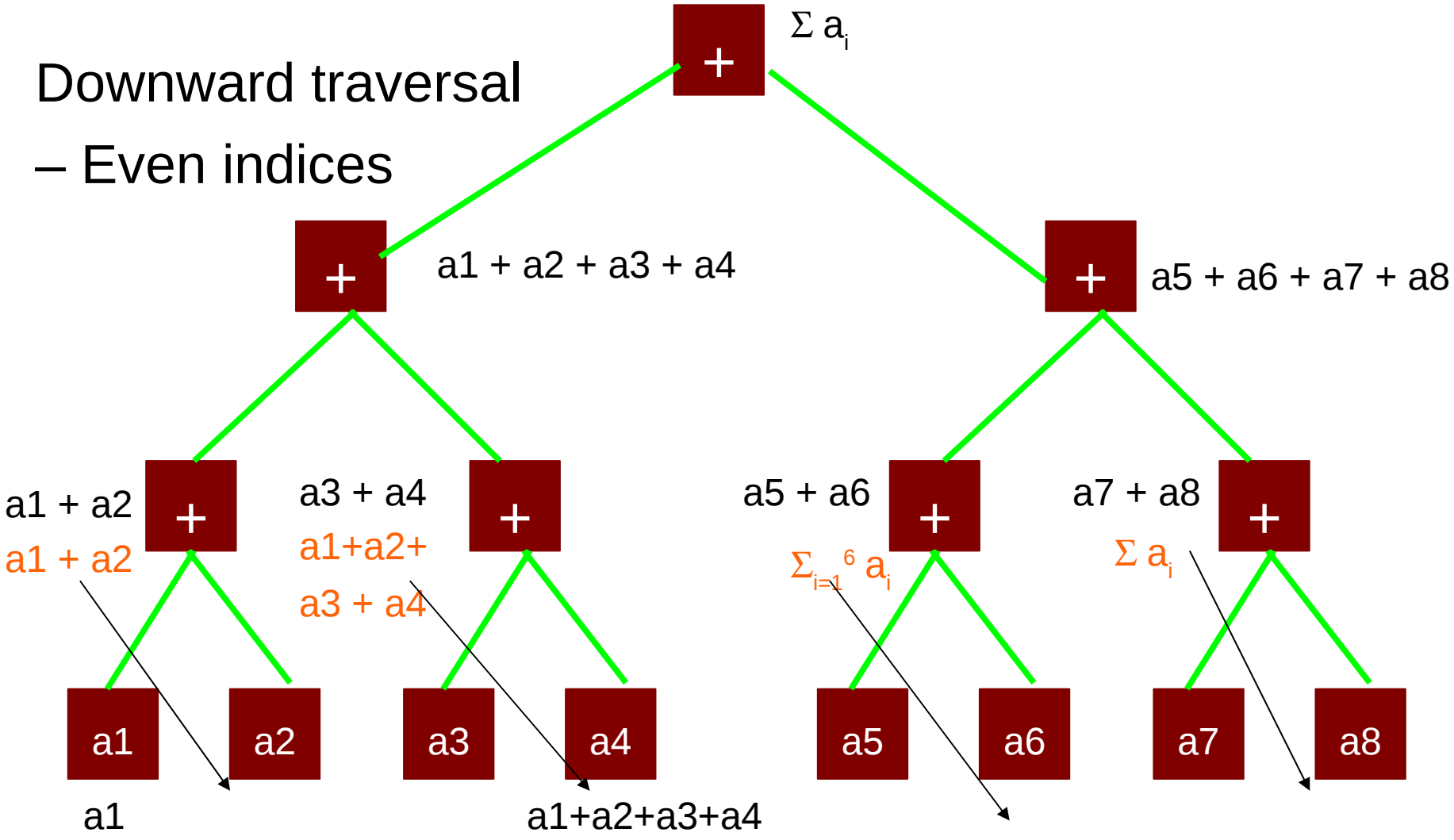
Upward traversal



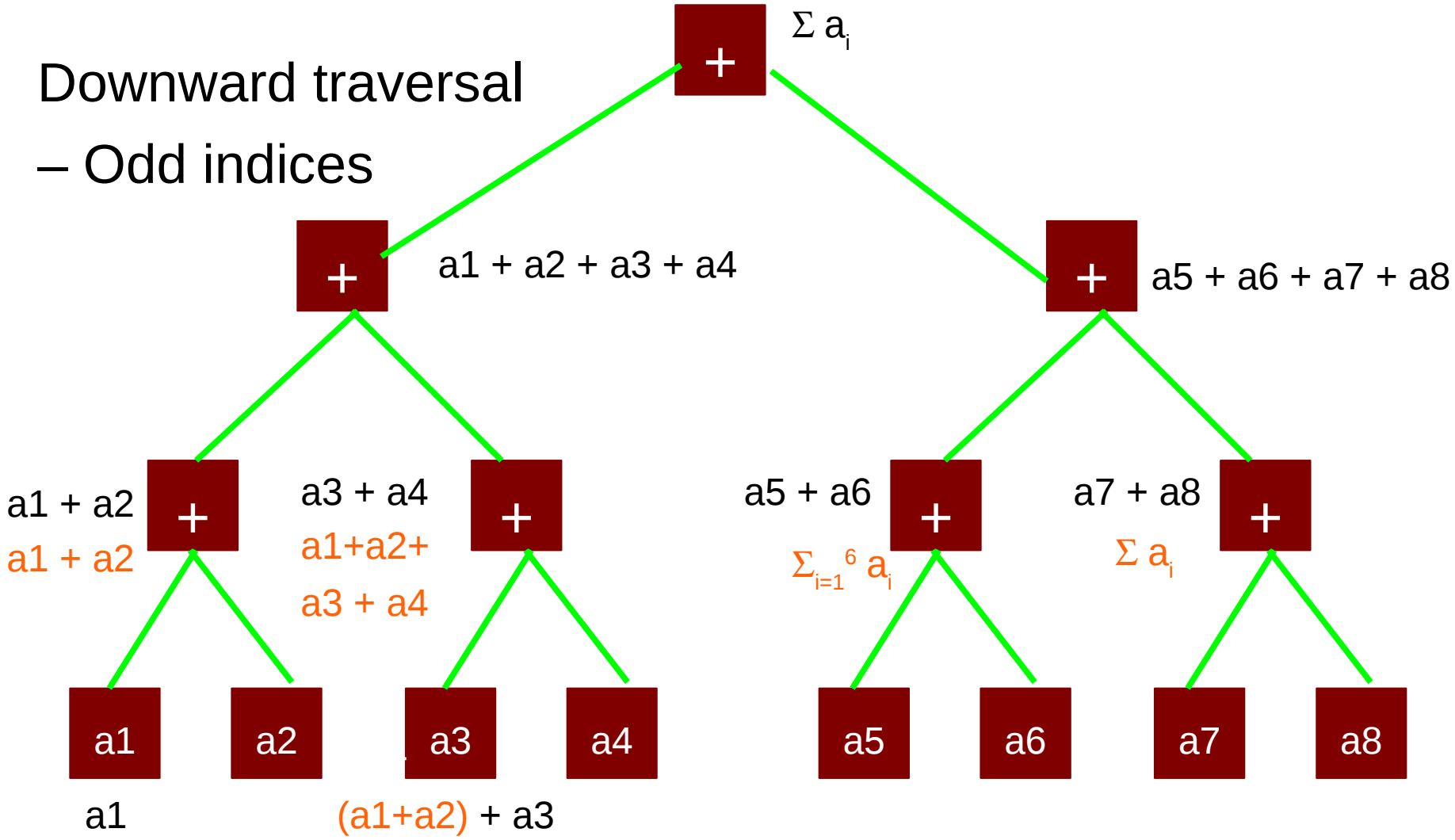
Balanced Binary Tree – Prefix Sum

Downward traversal

– Even indices



Balanced Binary Tree – Prefix Sum



The PRAM Algorithm

- Shown on the right side is a pseudo-code in the PRAM style.
- Time complexity = O(log n)
- Work complexity = O(n)
  - Meets the sequential work complexity, hence called optimal.
- Can also write a non-recursive variant.
  - See JaJa [Chapter 2]

```
//upward traversal
1. for i = 1 to n/2 do in
parallel
 b_i = a_2i-2 o a_2i
2. Recursively compute the
prefix sums of B= (b_1,
b_2, ..., b_n/2) and store them
in C = (c_1, c_2, ..., c_n/2)
//downward traversal
3. for i = 1 to n do in
parallel
   i is even : s_i = c_i
   i= 1 : s_i = x_i
   i is odd : s_i = c_(i-1)/2 o a_i
```

From Algorithm to a GPU Implementation

- Most PRAM algorithms assume that a lot of processors are available and are working at the same time.
  - Practical architectures cannot support that assumption.
  - Requires rethinking on synchronization, if any, that is needed.
  - One possible alternative is to use double-buffering.
  - Helps in cases where the PRAM algorithm works-in-place.

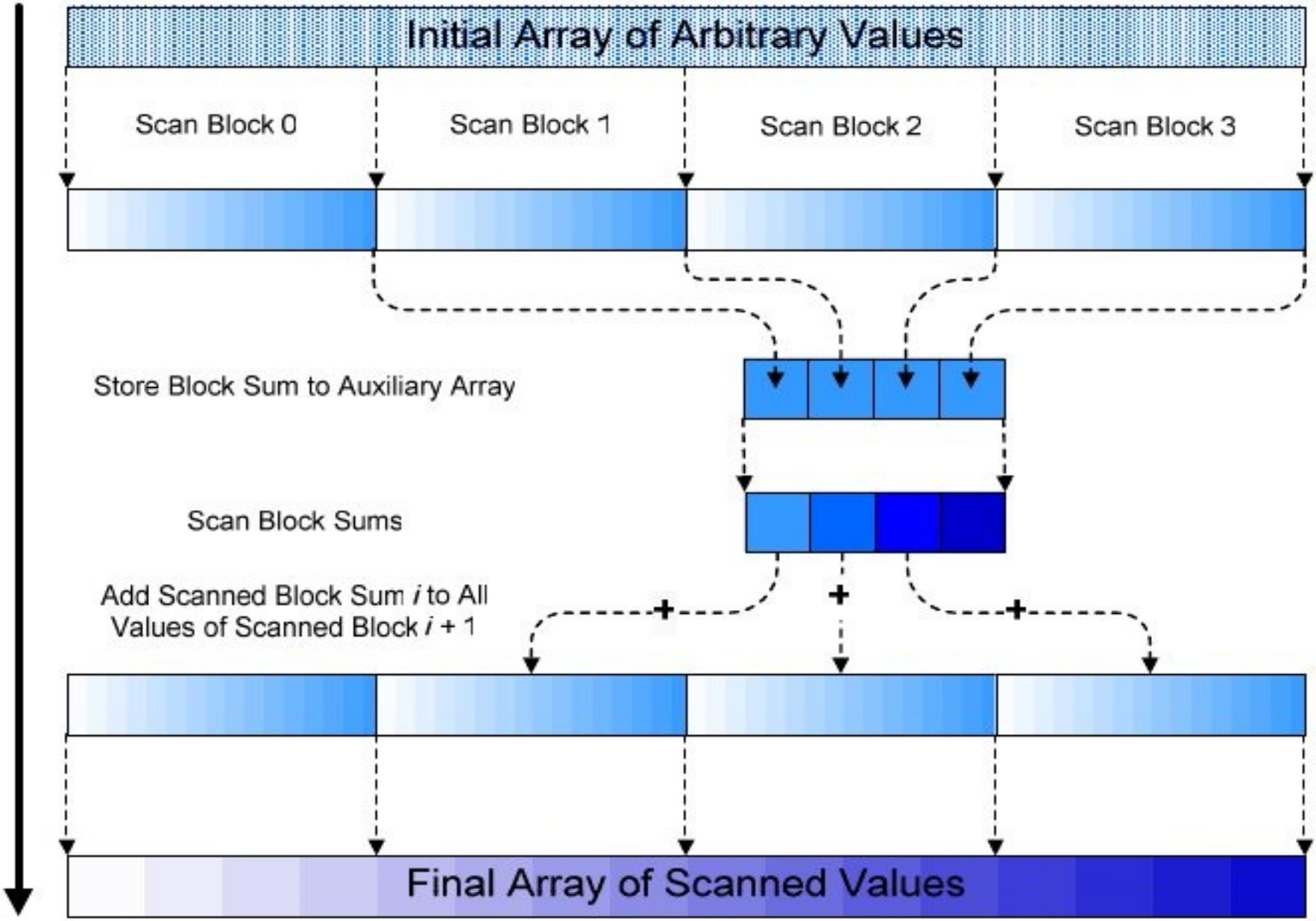
From Algorithm to a GPU Implementation

- Algorithmically, the PRAM solution is quite efficient.
- But, on the GPU, can result in several bank conflicts.
  - Recall that bank conflicts to shared memory can degrade performance.
  - Bank conflicts are given serialized access to the shared memory bank.

From Algorithm to a GPU Implementation

- The PRAM algorithm as stated, requires **all** threads to share information.
- This is not possible on the GPU. Only a block of threads can share data.
- The algorithm can however be recast.
  - Compute solutions per block of threads
  - Use these partial results to form an auxiliary array
  - Solve the same problem on the auxiliary array
  - Extend the solution to the original array.

Handling Large Arrays – in Pictures



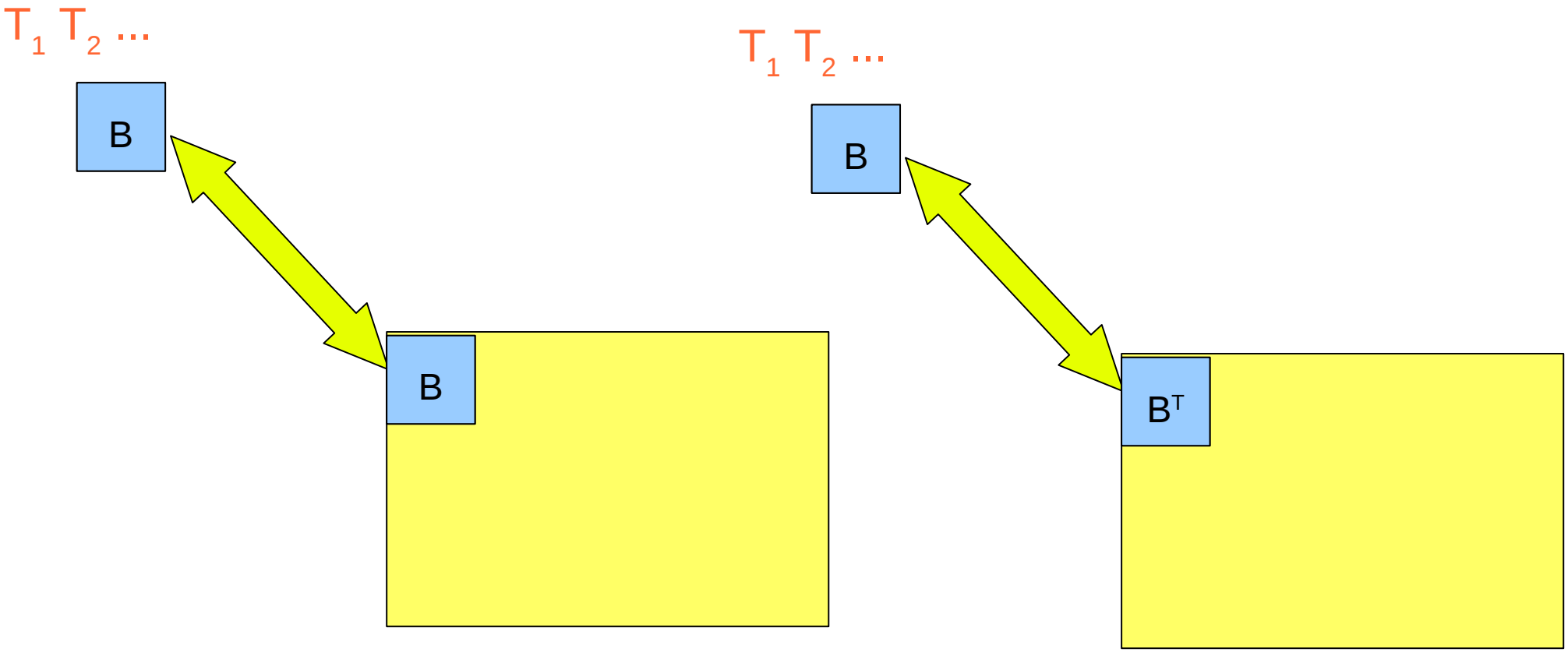
Picture taken from [Harris 2008].

Another Example – Matrix Transpose

$$M = \begin{bmatrix} 4 & -1 & 6 \\ 3 & 0 & 8 \\ 1 & 2 & 7 \end{bmatrix} \qquad M' = \begin{bmatrix} 4 & 3 & 1 \\ -1 & 0 & 2 \\ 6 & 8 & 7 \end{bmatrix}$$

- Input: A square matrix M of size nxn.
- Output: A square matrix M' such that M'[i,j] = M[j,i].
- Very simple operation, yet challenging enough on architectures such as a GPU.

Matrix Transpose



- Basic approach
  - Each thread reads an element M[i,j] and writes to M'[j,i].
  - Reading phase is has a high degree of memory coherence.
  - However, writing phase exhibits total lack of memory coherence.

Matrix Transpose

- Each write therefore is very expensive.
- Naturally, performance is not good.

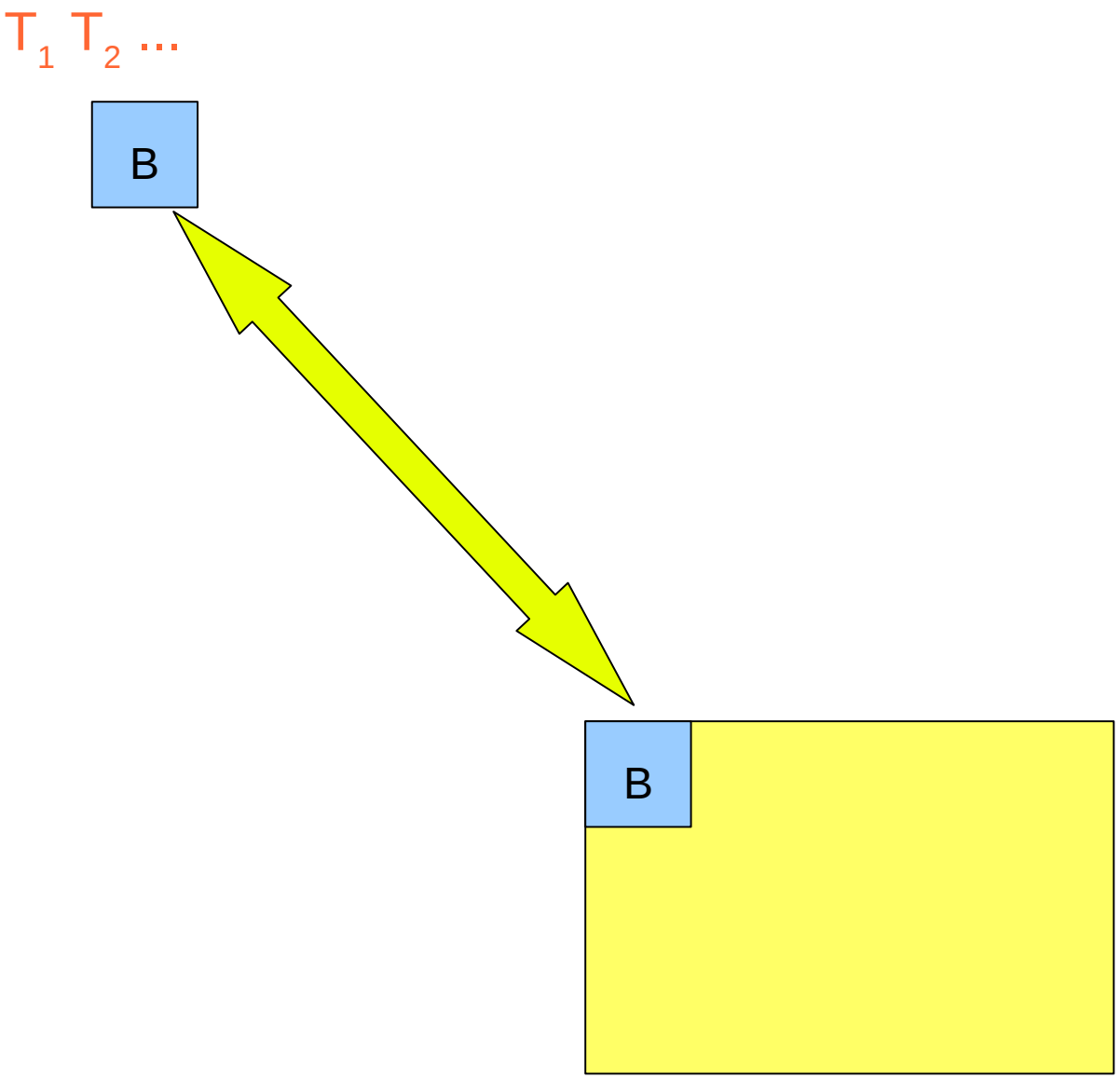
Matrix Transpose

- The simple operation illustrates the difficulty of GPGPU programming.
- We have however not thought of shared memory so far in this application.
- We will now use shared memory to get memory coherence during write operation also.

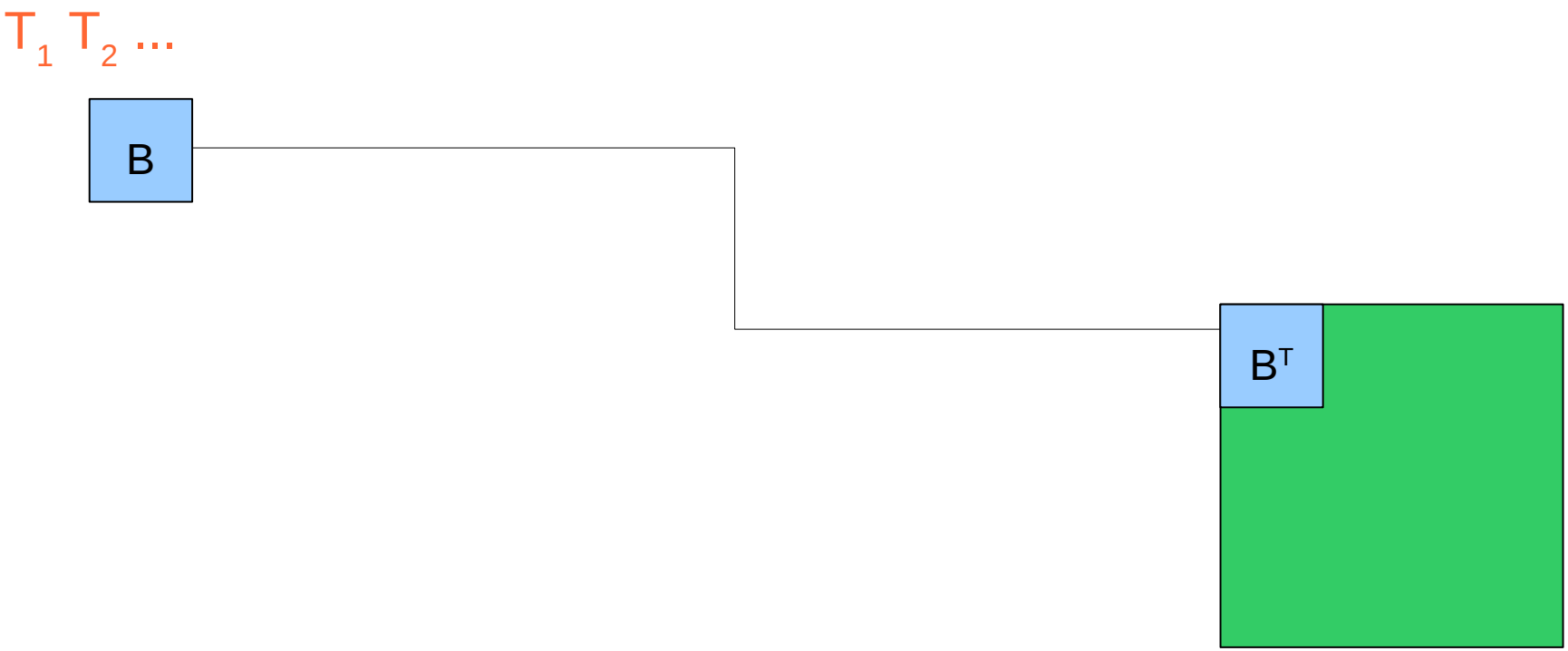
Matrix Transpose

- Recall that a block of threads have access to 16 KB of shared memory.
- Let thread Tij read M[i,j] from the global memory and write it to M'[j,i] in shared memory.
- Now, Tij writes M'[i,j] from shared memory to global memory.
- Now, also writes to global memory exhibit memory coherence.

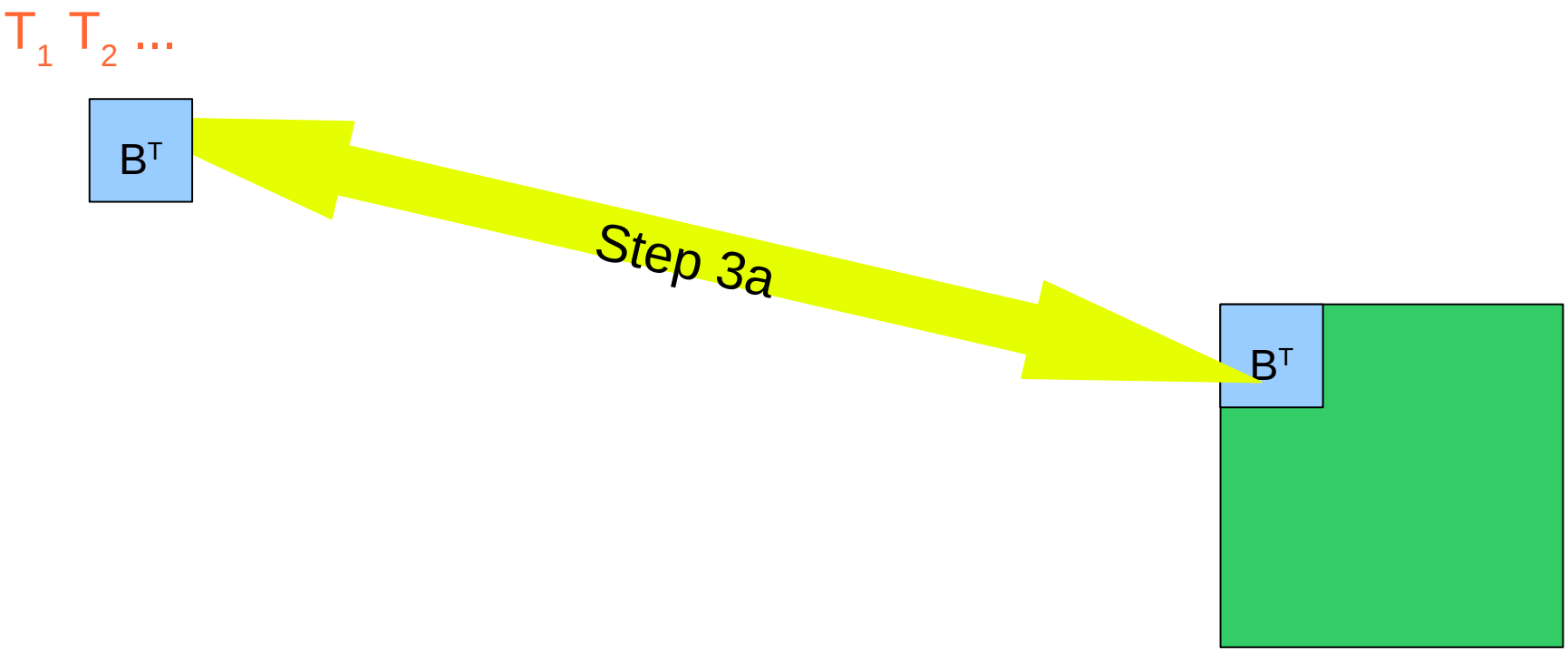
In Pictures Step 1 : Read from Global Memory



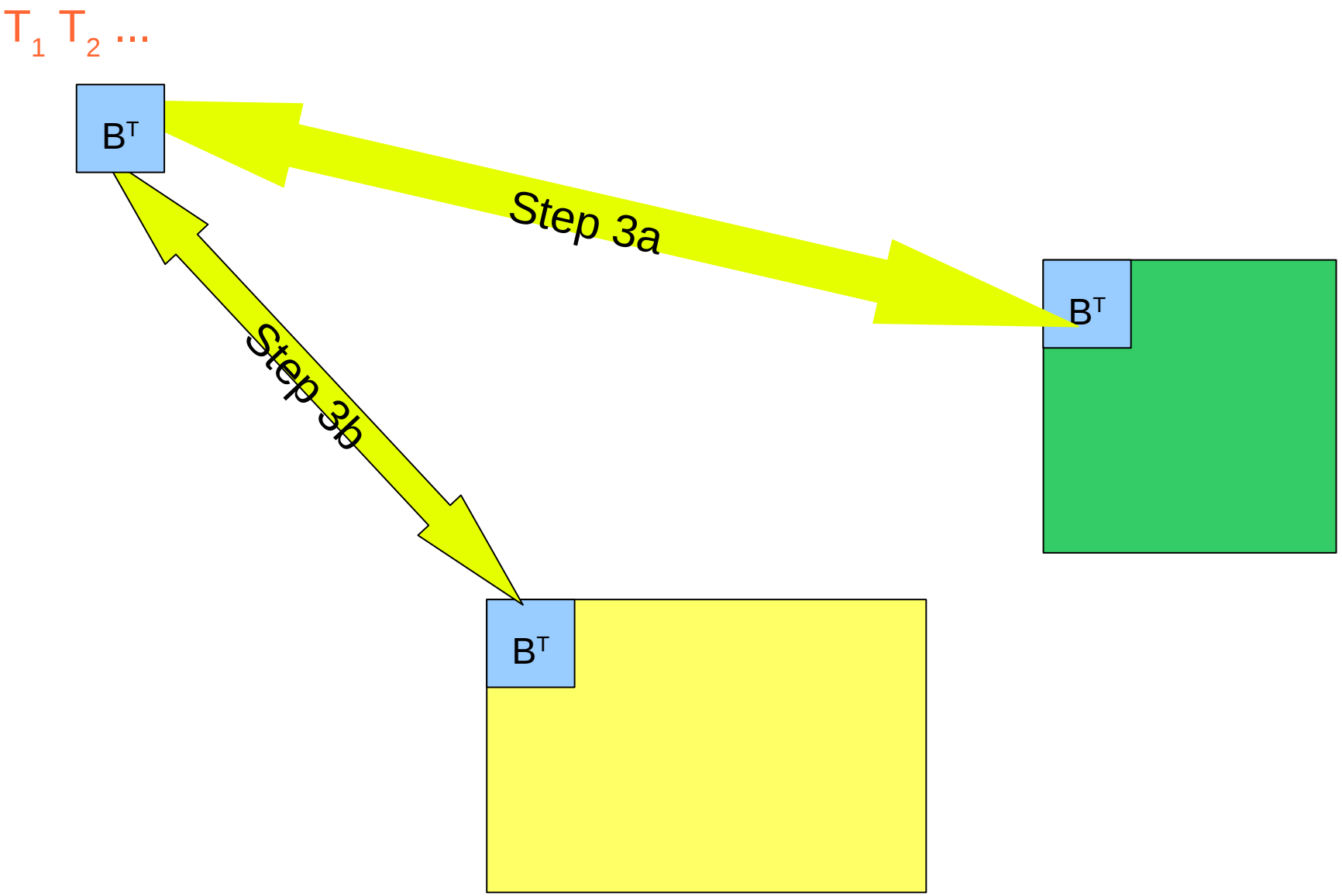
In Pictures : Step 2 : Write to Shared Memory



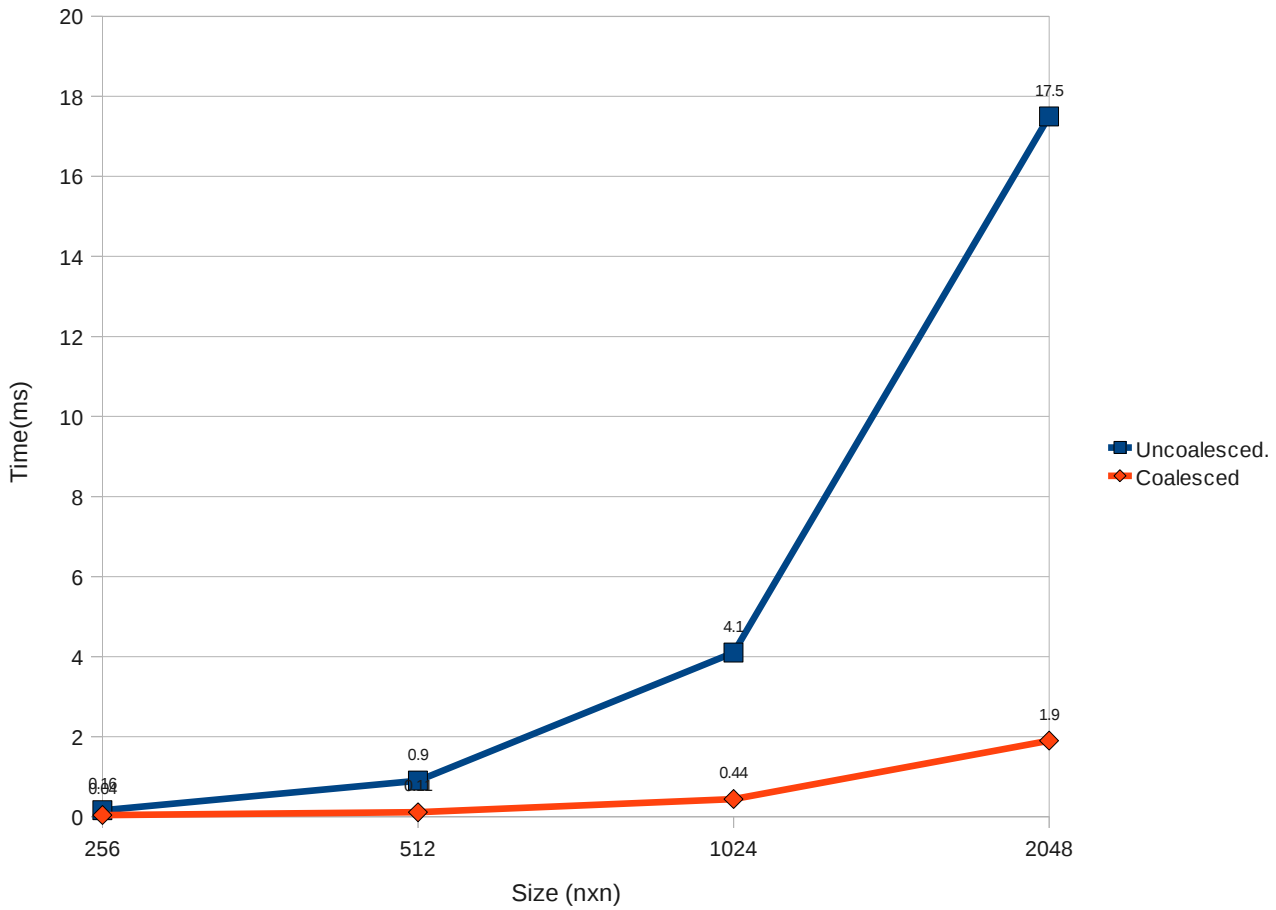
In Pictures Step 3 : From Shared to Global



In Pictures Step 3 : From Shared to Global



Matrix Transpose – Results



Summary of Regular Algorithms

- While PRAM algorithms appear easy and intuitive, realizations on present architectures are not straight-forward.
- Adaptations to suit the architecture and programming model required.
- Typically have to go through iterations of development, profiling, optimization, and tuning.
- It may be also possible that the best PRAM algorithm may not be the best choice for a GPU implementation.

Schedule

0:00 – 0:10 : Introductory remarks

0:10 – 0:30 : Basics of computer architecture

0:30 – 0:55 : GPGPU architectural features

0:55 – 1:20 : GPGPU programming

1:20 – 1:50 : Regular algorithms on the GPU

S H O R T B R E A K

2:10 – 2:40 : Irregular algorithms on the GPU

2:40 – 3:00 : Limitations of GPUs

3:00 – 3:20 : Future trends and directions

3:20 – 3:30 : Open discussion

Irregular Algorithms on GPU

- Problems that require irregular or random memory accesses or sequential compute dependencies are not ideally suited to the GPU.
- The list ranking problem is a typical problem that has these characteristics
- Other problems with similar characteristics include
  - graph problems
  - sparse matrix computations

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

Popular Parallel List Ranking Algorithms

- Wylie's Algorithm (1979)
  - First Algorithm, Pointer Jumping and not work-optimal
- Cole and Vishkin (1989)
  - First Optimal
- Anderson and Miller (1990)
  - Optimal. Deterministic, Independent set based – Difficult to Implement
- Hellman & JáJá Algorithm (1999)
  - Spare Ruling Set Approach for Symmetric Multiprocessor Systems – original algorithm by Reid-Miller (1994)

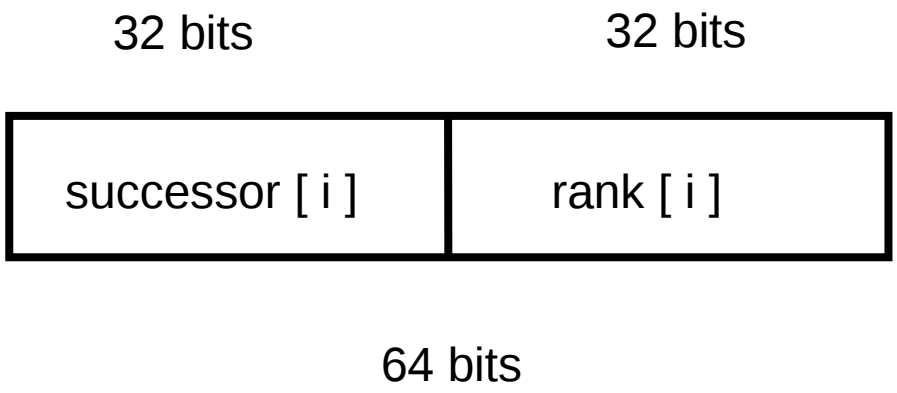
Parameters that dictate performance on a GPU

- Optimal Algorithm and Division of work
- Massive parallelism
- Efficiency of Memory operations
  - Global Memory *Coalescing*
- Good ratio of compute to memory operations
  - Allow hardware to hide latencies as much as possible.
- GPU Treated as pure PRAM for this application

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)

Synchronization and Hazards

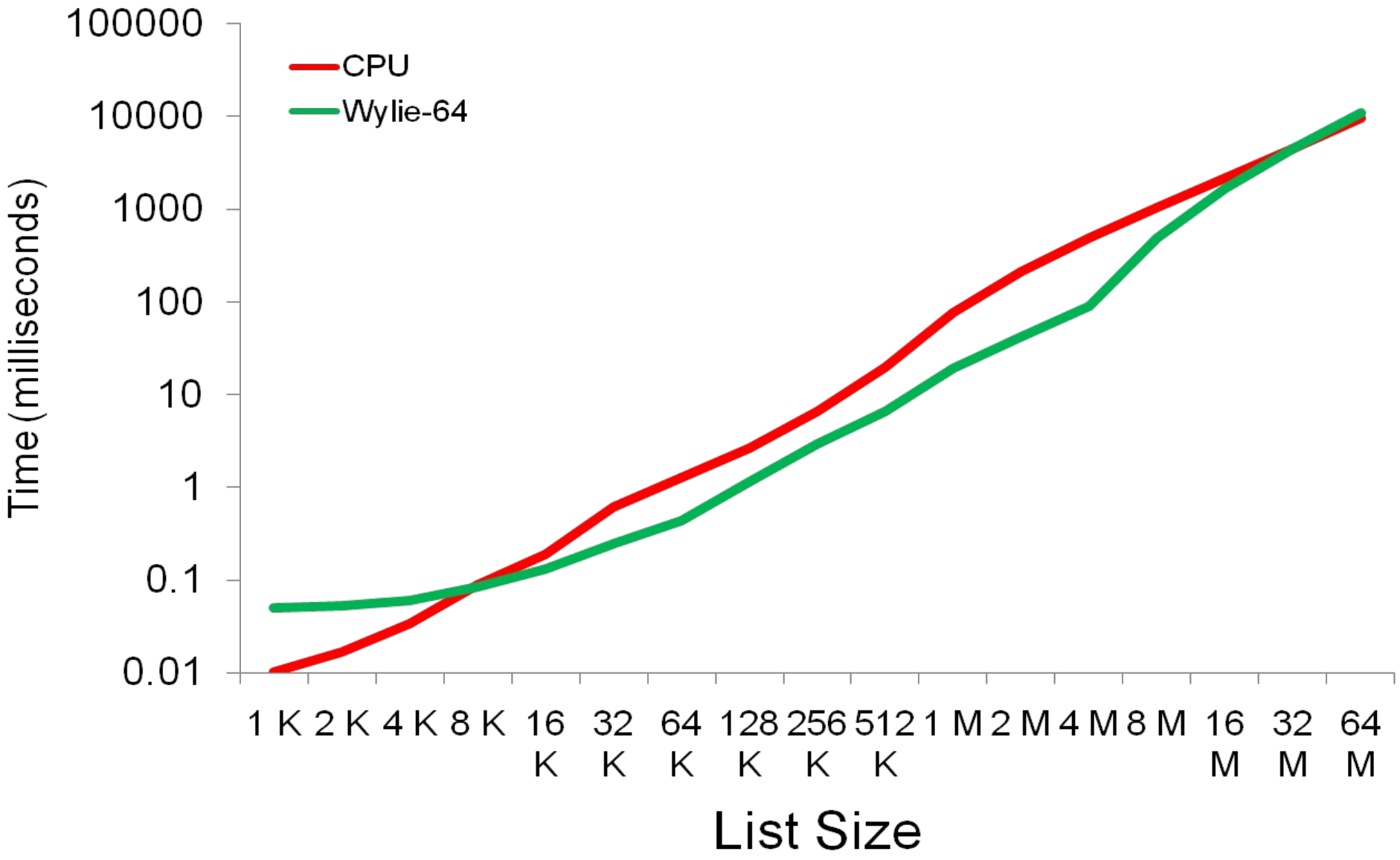


- Algorithm requires that both Rank and Successor be updated simultaneously
- We pack the variables into a 64-bit word and write to guarantee simultaneous update

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

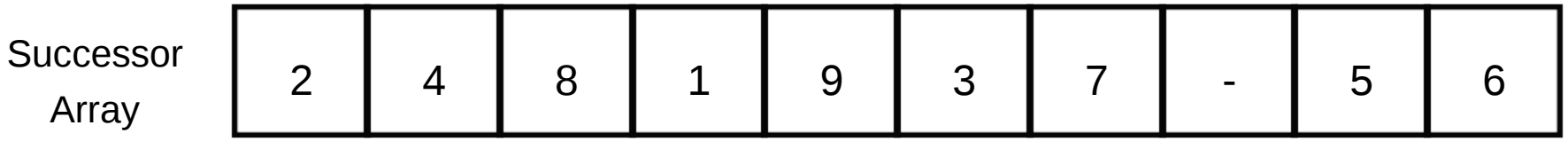
Helman JáJá Algorithm

- 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.

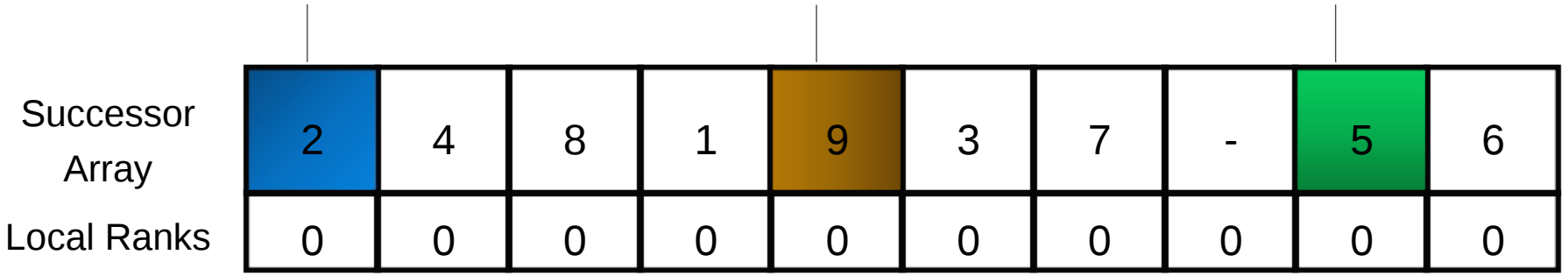
Helman-JáJá (Contd.)

- 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.

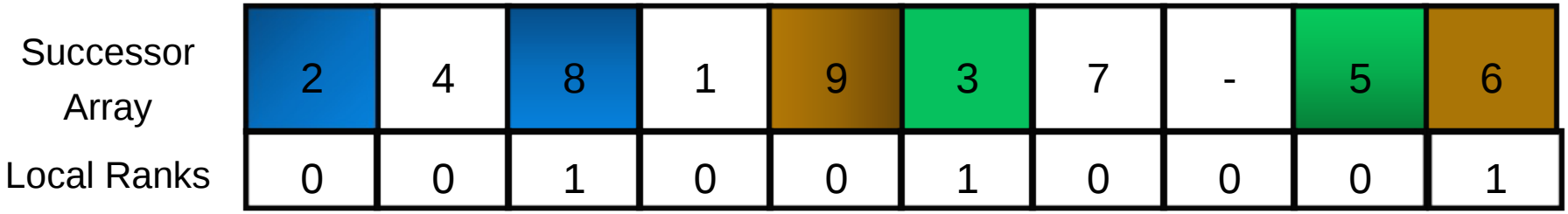
An example of the Helman-JaJa algorithm on a small liked List



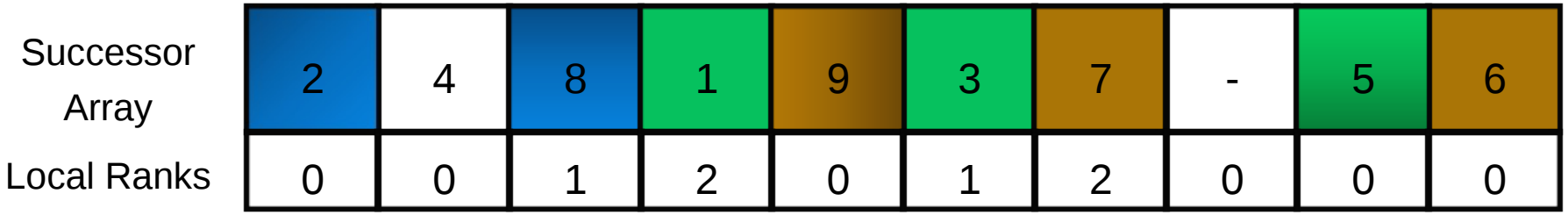
Step 1. Select **Splitters** at equal intervals



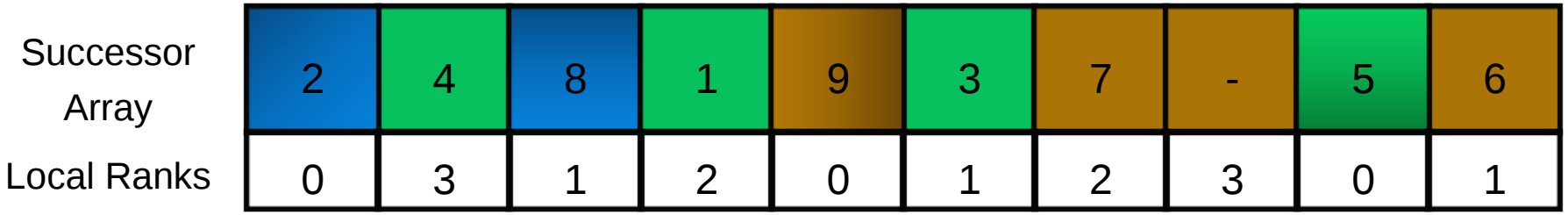
Step 2. **Traverse** the List until the next splitter is met and **increment** local ranks as we progress



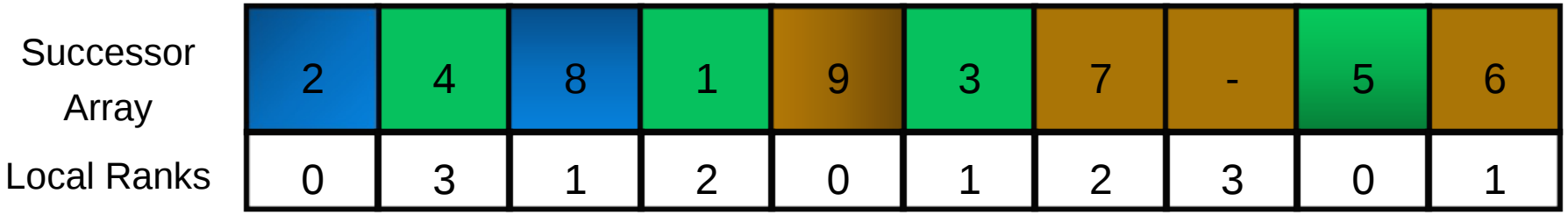
Step 2. **Traverse** the List until the next splitter is met and **increment** local ranks as we progress



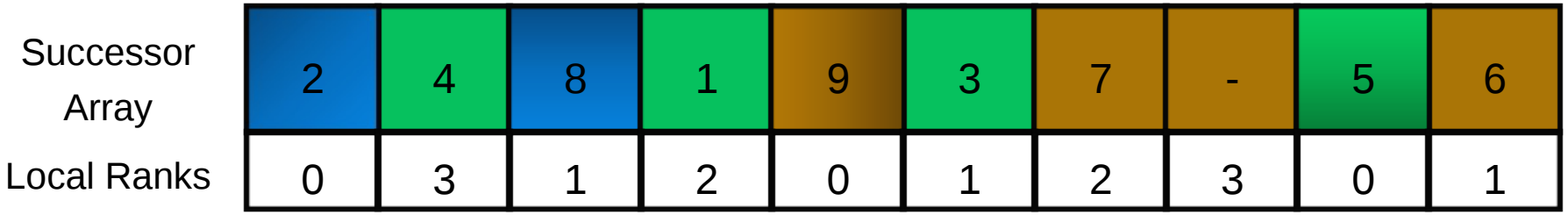
Step 2. **Traverse** the List until the next splitter is met and **increment** local ranks as we progress



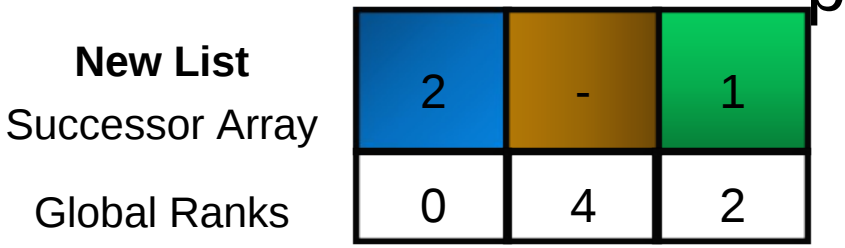
Step 3. **Stop** When all elements have been assigned a local rank

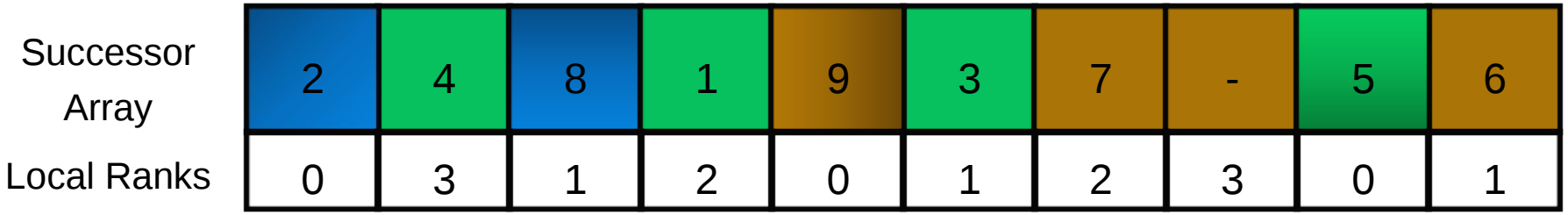


Step 4. **Create** a new list of splitters which contains a **prefix value** that is equal to the local rank of it's predecessor

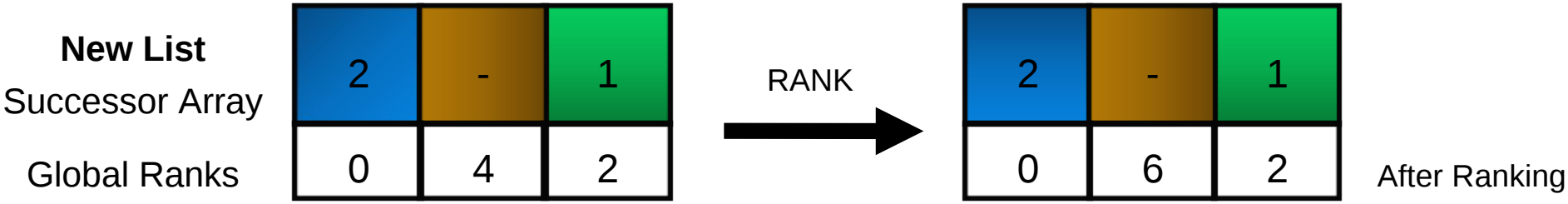


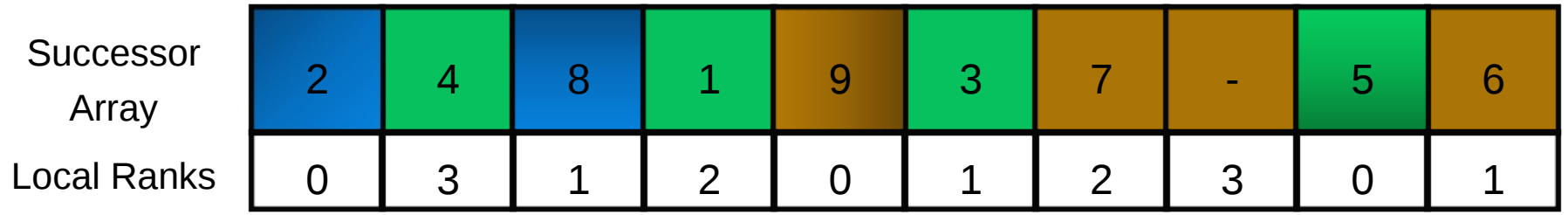
Step 4. **Create** a new list of splitters which contains a **prefix value** that is equal to the local rank of it's predecessor



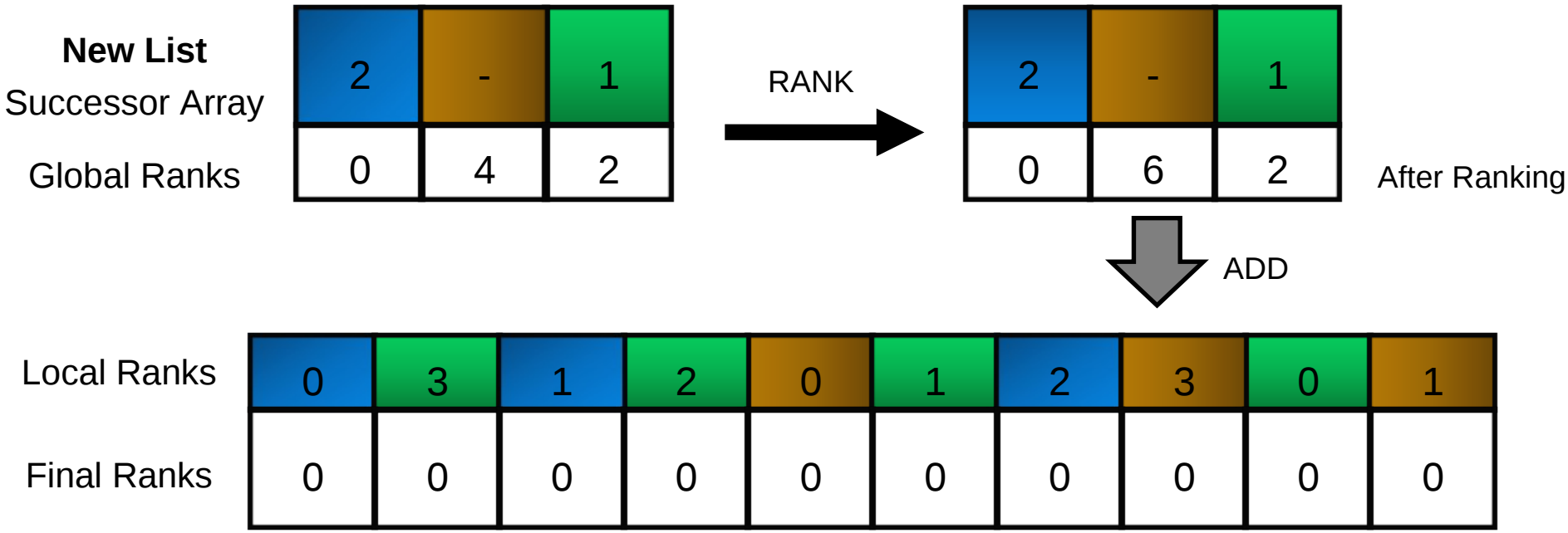


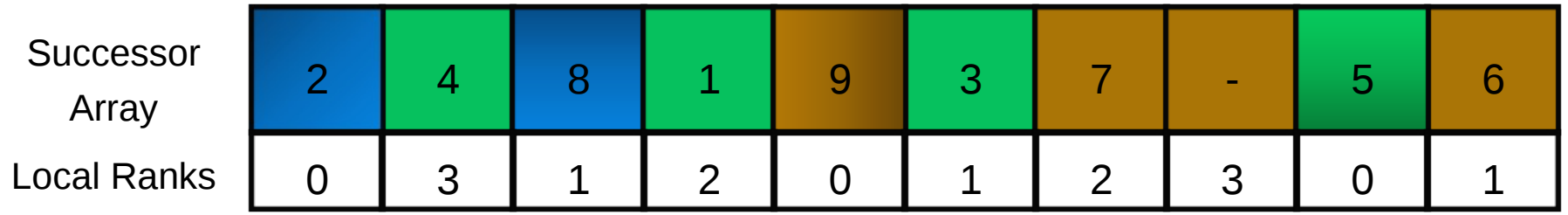
Step 5. Scan the global ranks array **sequentially**



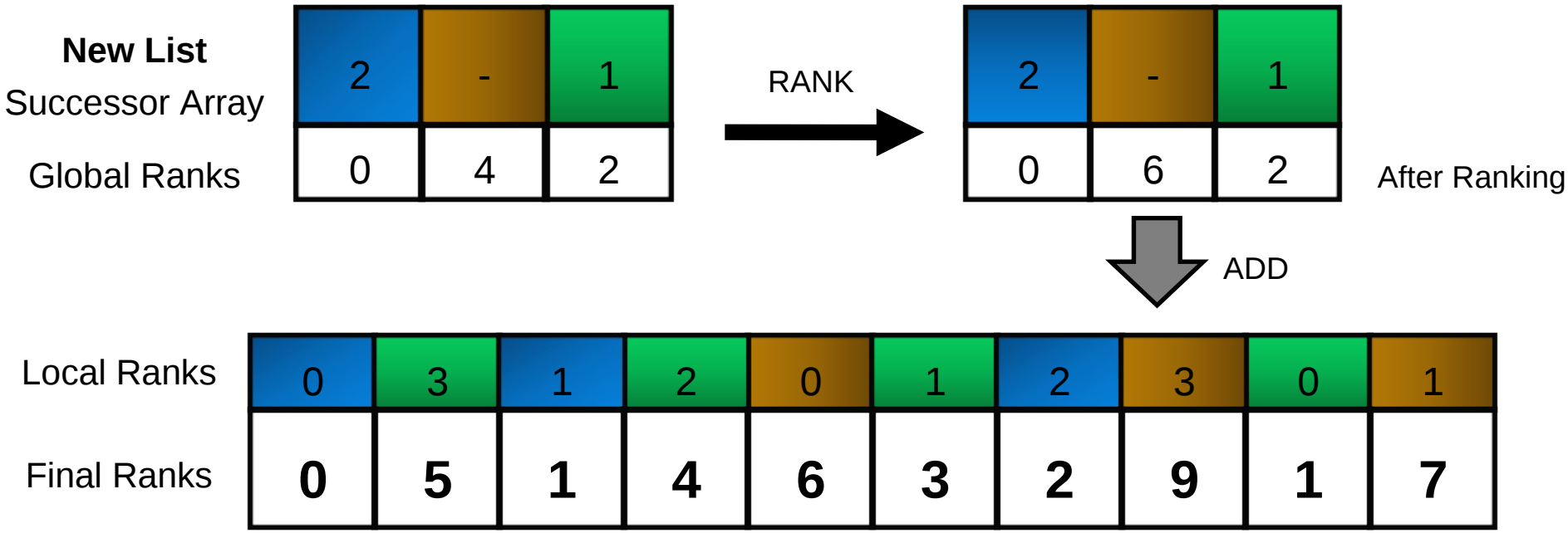


Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.



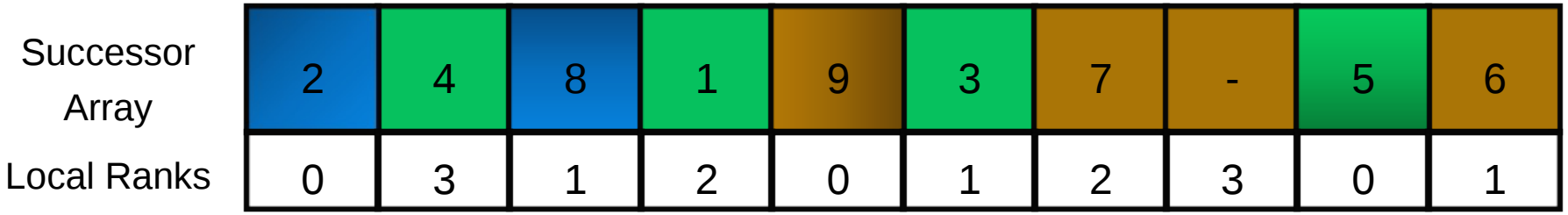


Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

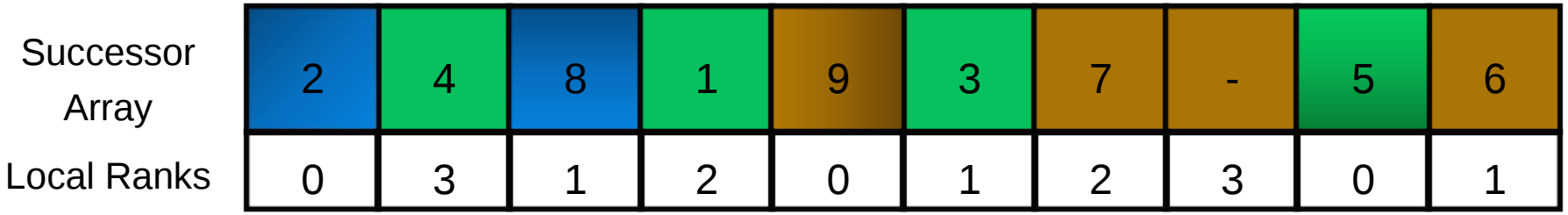


Modifying the algorithm for GPU

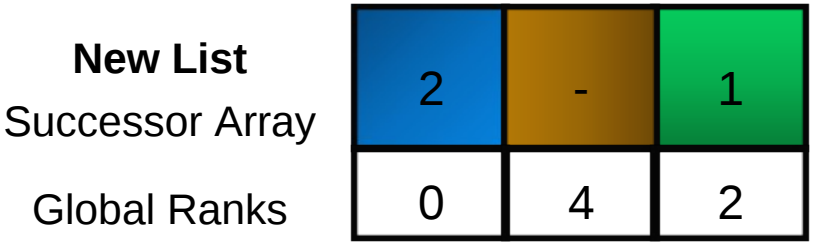
- Step 5 is a sequential ranking step.
- When we choose log n splitters, we reduce the list to n/log n, which is still large amount of sequential work
- By Amdahl's law, this is a bottleneck for parallel speedup. More so in the case of GPU.

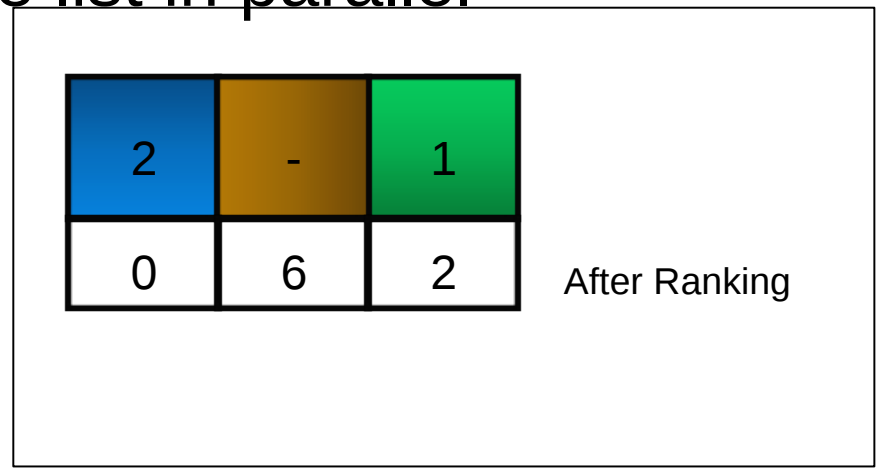


Make step 5 **recursive** to allow the GPU to continue processing the list in parallel



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





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

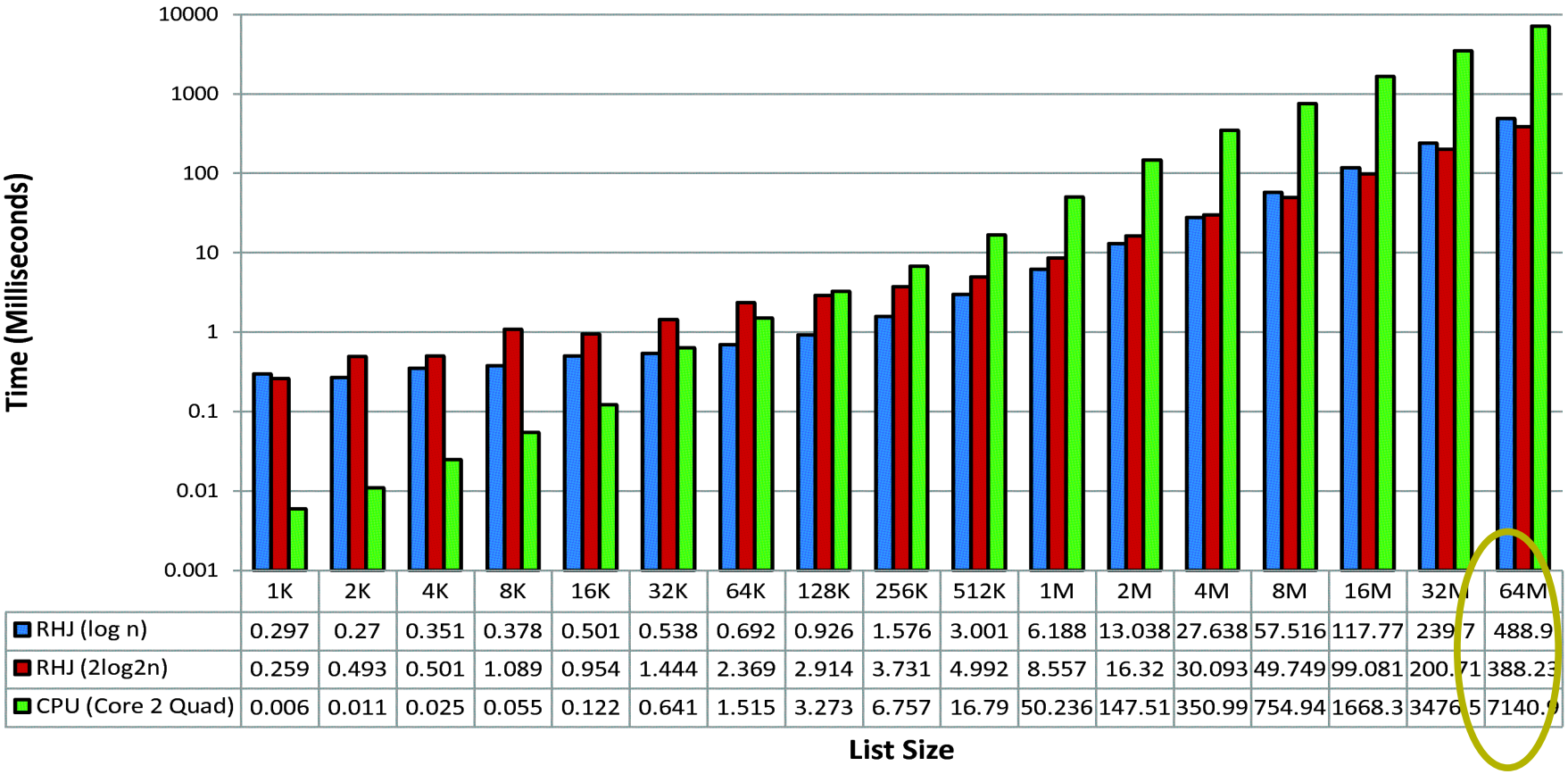
GPU Implementation

- 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
- Stopping criteria:
  - List size is 1
  - Stop at a small size when the CPU can take over.

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



	1K	2K	4K	8K	16K	32K	64K	128K	256K	512K	1M	2M	4M	8M	16M	32M	64M
RHJ (log n)	0.297	0.27	0.351	0.378	0.501	0.538	0.692	0.926	1.576	3.001	6.188	13.038	27.638	57.516	117.77	239.7	488.9
RHJ (2log2n)	0.259	0.493	0.501	1.089	0.954	1.444	2.369	2.914	3.731	4.992	8.557	16.32	30.093	49.749	99.081	200.71	388.23
CPU (Core 2 Quad)	0.006	0.011	0.025	0.055	0.122	0.641	1.515	3.273	6.757	16.79	50.236	147.51	350.99	754.94	1668.3	3476.5	7140.9

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

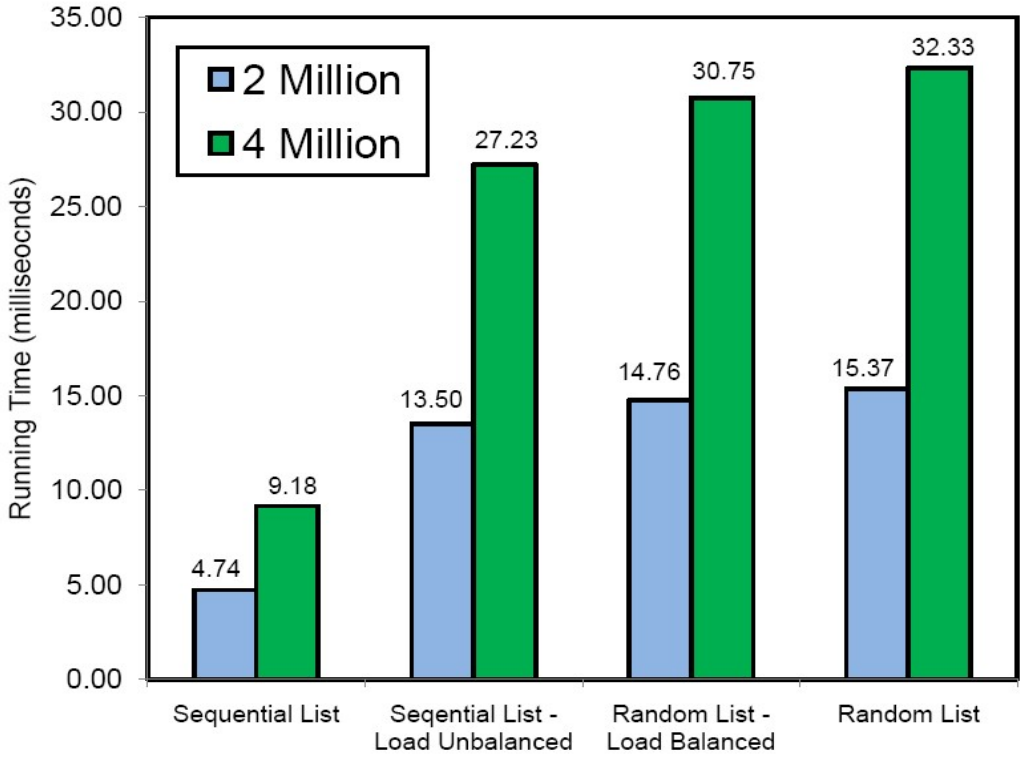
Implementation Bottlenecks I

- Execution profiled in CUDA
- Max. time taken is in the local ranking for first iteration
- Rest of the time spent in recursion is < 1% of the total.

List Size	Time for 1 iteration (µsec)			Total Time (µsec)
	Split	Local Rank	Recombine	
1 M	11	5350	1094	7182
2 M	15	12273	922	13367
4 M	24	24851	1881	26927
8M	46	124658	4256	129203

Implementation Bottlenecks II

- Load Balancing among threads
- Irregular Memory access
- In these tests, we show that the combination of both play an important role in determining runtime.



A Different Approach

- Recently used in Wei and JaJa, 2010.
- Instead of recursive ranking, use a fixed number of sub-lists.
- Rank the list of splitters on the CPU by transferring that list.
- Improves on the performance by about 30%.

Summary of Irregular Algorithms

- With irregular algorithms, GPU can be thought of as a highly non-uniform PRAM.
  - Different costs for memory and computation.
- Performance suffers heavily compared to CPUs
  - CPUs do not have much of a problem handling irregular algorithms.
- PRAM algorithms do not account for such high memory access costs.

Schedule

0:00 – 0:10 : Introductory remarks

0:10 – 0:30 : Basics of computer architecture

0:30 – 0:55 : GPGPU architectural features

0:55 – 1:20 : GPGPU programming

1:20 – 1:50 : Regular algorithms on the GPU

S H O R T B R E A K

2:10 – 2:40 : Irregular algorithms on the GPU

2:40 – 3:00 : Limitations of GPUs

3:00 – 3:20 : Future trends and directions

3:20 – 3:30 : Open discussion

Limitations of GPUs

- Support for varied data types
  - GPU computing most efficient on 32-bit values.
    - Double precision arithmetic comes with a performance penalty.
  - Smaller or larger bit-sized values are not efficiently supported.
  - Smaller do make sense, especially bit or char.
  - Larger are required in some cases, for instance double precision arithmetic.
  - Offer a good way to hide memory latency.

Limitations of GPUs

- How much data parallelism is available in the application?
- This issue studied also theoretically as the class NC of problems.
  - But the NC theory may not suffice.
  - NC theory kicks in at a very high degree of parallelism,
  - In the present case, suffices for even moderate degree
  - But the theory is missing.
- More critical for GPU algorithms
  - Any thread divergence can prove to be costly.

Limitations of GPUs

- Gather/Scatter
  - An important primitive in parallel computing
  - Scan is an example of gather.
  - Scatter is much difficult on GPUs due to lack of memory coherence.

Limitations of GPUs

- Huge Memory Latency
- Especially, for irregular memory accesses, the memory latency is too high.
- However, peak compute flops are continuing to rise.
- This can deepen the impact of memory latency and bandwidth.
- Current GPU memory technology cannot scale to large capacities also
  - Has pin-count and power limitations.
  - Should explore other memory technologies such as 3D-stacking

Limitations of the GPUs

- No hardware cache – why does this hurt?
- Difficult for a user to arrange for good data locality
  - Recall the advantages of having the working set in a cache
- While this may be easy to figure out in a uniprocessor setting, much more difficult in a multi-threaded environment.

Limitations of GPUs

- Atomic operations
  - Required when multiple threads need to update a common value.
  - For instance, histogram
- There is limited support, but thread divergence is very limiting on performance.
- Good hardware support for atomic operations can speed up several computations, such as histogram

Schedule

0:00 – 0:10 : Introductory remarks

0:10 – 0:30 : Basics of computer architecture

0:30 – 0:55 : GPGPU architectural features

0:55 – 1:20 : GPGPU programming

1:20 – 1:50 : Regular algorithms on the GPU

S H O R T B R E A K

2:10 – 2:40 : Irregular algorithms on the GPU

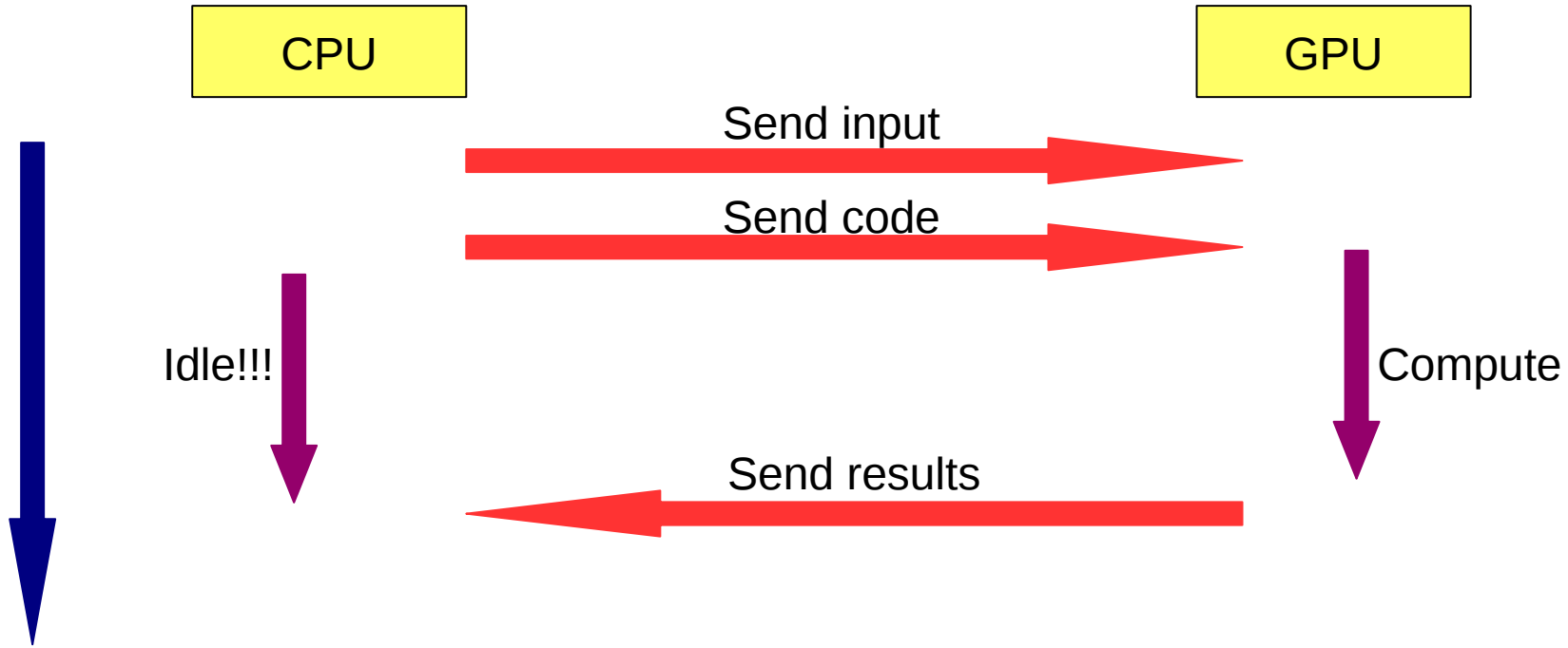
2:40 – 3:00 : Limitations of GPUs

3:00 – 3:20 : Future trends and directions

3:20 – 3:30 : Open discussion

Future Trends and Directions

- Where did the CPUs go?
  - The story so far is that of GPUs making the CPU almost redundant.
- The GPGPU model so far:



- Can the CPU be also used along with the GPU?

Future Trends and Directions

- Hybrid Multicore Computing
  - An emerging line of thought
  - Use multiple, possibly heterogeneous, devices at the same time.

- What about CPU-GPU hybrid computing?
- Brings several questions to the fore
  - How to design hybrid algorithms?
  - Analytical models of hybrid algorithms?
  - How to synchornize computation between CPU and the GPU?

Future Trends and Directions

- Synchronization may be a big hurdle at present.
- The bandwidth between the CPU and the GPU is a Gb/s.
- For some problem sizes of interest, this small bandwidth means that transfer times may be a big chunk of overall time taken.
- Suggests the following approach
  - Design hybrid algorithms so that very little synchronization is needed.
  - Works in some cases, but not always.

Future Trends and Directions

- Previous limitations indicate that future GPUs should have the following architectural features
  - Better memory access
  - Better synchronization support
  - Better support for double precision operations
- We will now see how some of these concerns may be addressed in the near future, plus some other interesting leads.

Future Trends and Directions

- On Chip CPU-GPU combinations
- Integrated model is much more economical in terms of power usage and space.
- Also helps synchronization between the two.
- One possible model is that CPU handles costly operations such as floating point arithmetic,
- AMD Zacate

Future Trends and Directions

- Have a balance between SIMD and MIMD execution capabilities.
  - This also suggests hybrid computing.
- Special purpose hardware for particular primitives
- The key hardware architecture features for future throughput computing machines -
  - high compute and bandwidth,
  - large caches,
  - gather/scatter support,
  - efficient synchronization, and
  - fixed functional units

Future Trends and Directions

- Space on the GPU is always constrained.
- Techniques do not allow for speedy swap-in and swap-out between the host and the device.
- Two issues here:
  - Can an efficient (competitive) swap-in and swap-out mechanism be designed, especially when future usage is not known.
    - See HiPC 2009 Satish et al. for the offline case.
    - The problem has similarities to paging schemes.
  - Another possibility is to use the available space efficiently.
    - Succinct representations may help in some cases.
    - See HiPC 2010 Soman et al. for an example.

Future Trends and Directions

- Modeling of GPGPU
  - Modeling parallel computing is generally accepted to be hard.
  - Too many parameters to handle
  - Limited successes in this direction
    - ISCA 2010, Kim et al.
    - HiPC 2009, Kothapalli et al.
  - Both these models do not have much to say about scheduling.
    - While scheduling is not public knowledge, a model for the same can help programmers.

Future Trends and Directions

- Primitives led parallel computing
- Parallel programming is at present too tough to let everyone practice it.
- Nevertheless, parallel programming is going to stay.
- How to have a huge programmer population adapt to parallel programming?
- Primitives are the way forward
  - Have a uniform set of routines that are highly optimized
  - Write programs using mostly these routines.
- What are the right primitives?
- How to ensure coverage and completeness?
- Several such questions are relevant.

Summary

- We have covered a wide range of topics associated with GPGPU

- Architectural features
  - Performance considerations
- Programming environment
  - Trivial and non-trivial Examples
- Case studies
  - regular, and
  - irregular applications
- Limitations of GPUs and expected future trends
  - Hardware as well as algorithmic trends

References

- The Nvidia Programmer Reference Manual
- Fast and Scalable list ranking on the GPU, Rehman, Kothapalli, and Narayanan, ACM ICS 2009.
- Scan Primitives for GPU Computing, Sengupta, Harris, Zhang, and Owens, in Graphics Hardware, 2007.
- Debunking the 100x GPU Myth, An Evaluation of Throughput Computing on CPU and GPU, Lee et al., in ISCA 2010.
- Computer Architecture : A Quantitative Approach, Henessey and Patterson.
- A Performance Prediction Model for the GPGPU, Kothapalli et al. in HiPC 2009.
- An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness., Hong and Kim, In ISCA 2009.
- Satish Nadathur, HiPC 2009
- Efficient Discrete Range Searching Primitives with Applications, Soman, Kothapalli, and Narayanan, in HiPC 2010. (for succinct representation on GPUs)
- Optimization of Linked List Prefix Computations on Multithreaded GPUs Using CUDA, Wei and JaJa, in IPDPS 2010.
- Towards dense linear algebra for hybrid GPU accelerated manycore systems, in Parallel Computing, Vol 12., December 2009

Questions and Discussion