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

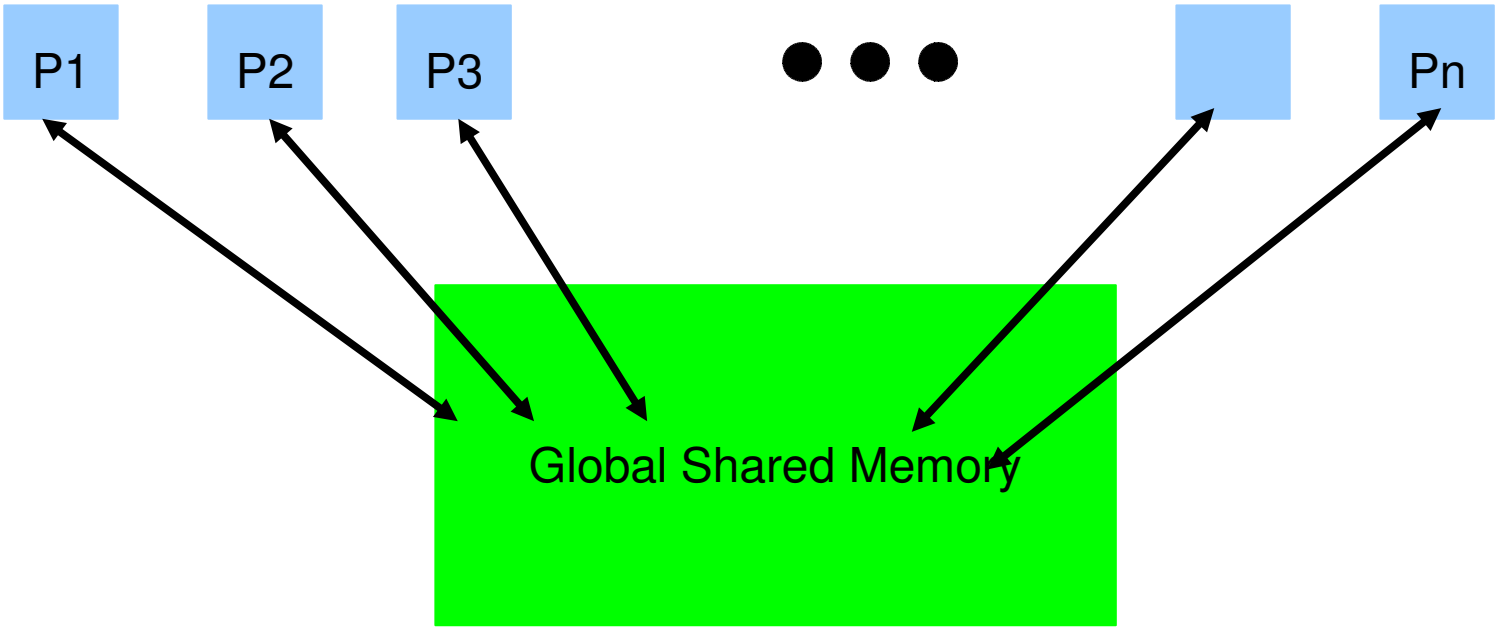
The Missing Link

- 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

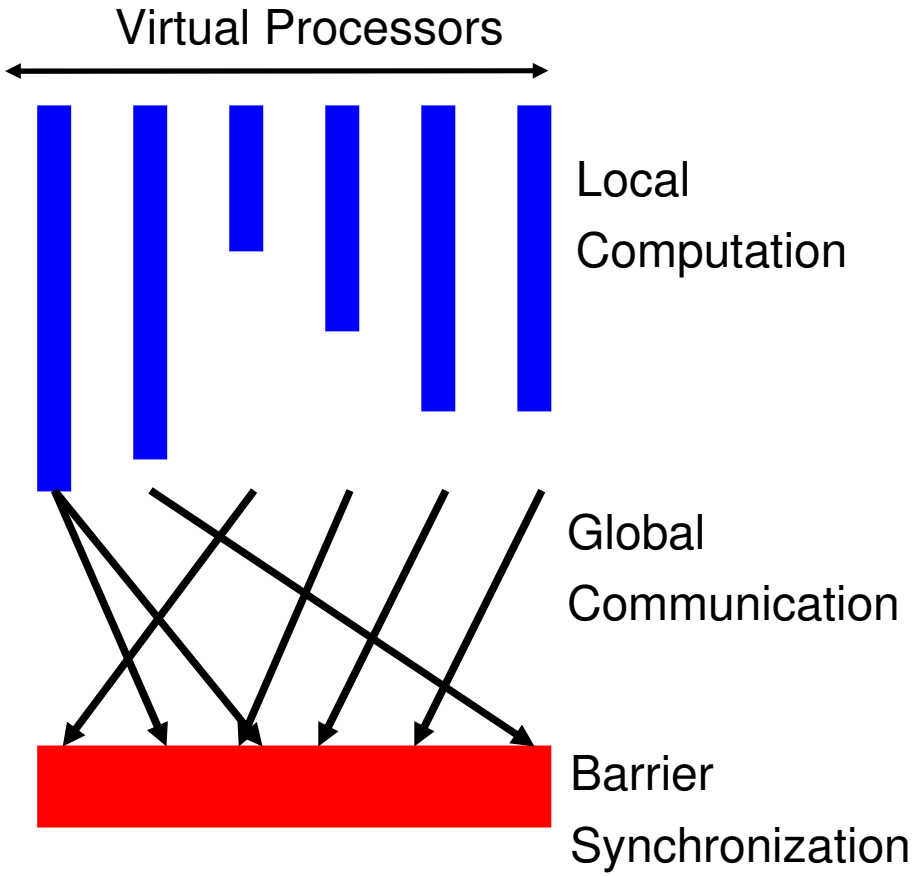
- A performance prediction model for GPGPU
  - Uses extensions to existing models such as BSP, PRAM, and QRQW
- Static runtime prediction
  - Analogous to asymptotic analysis
  - Done at the CUDA code level.
- Experimental validation by targeted experiments
- Three case-studies
  - Matrix multiplication, 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

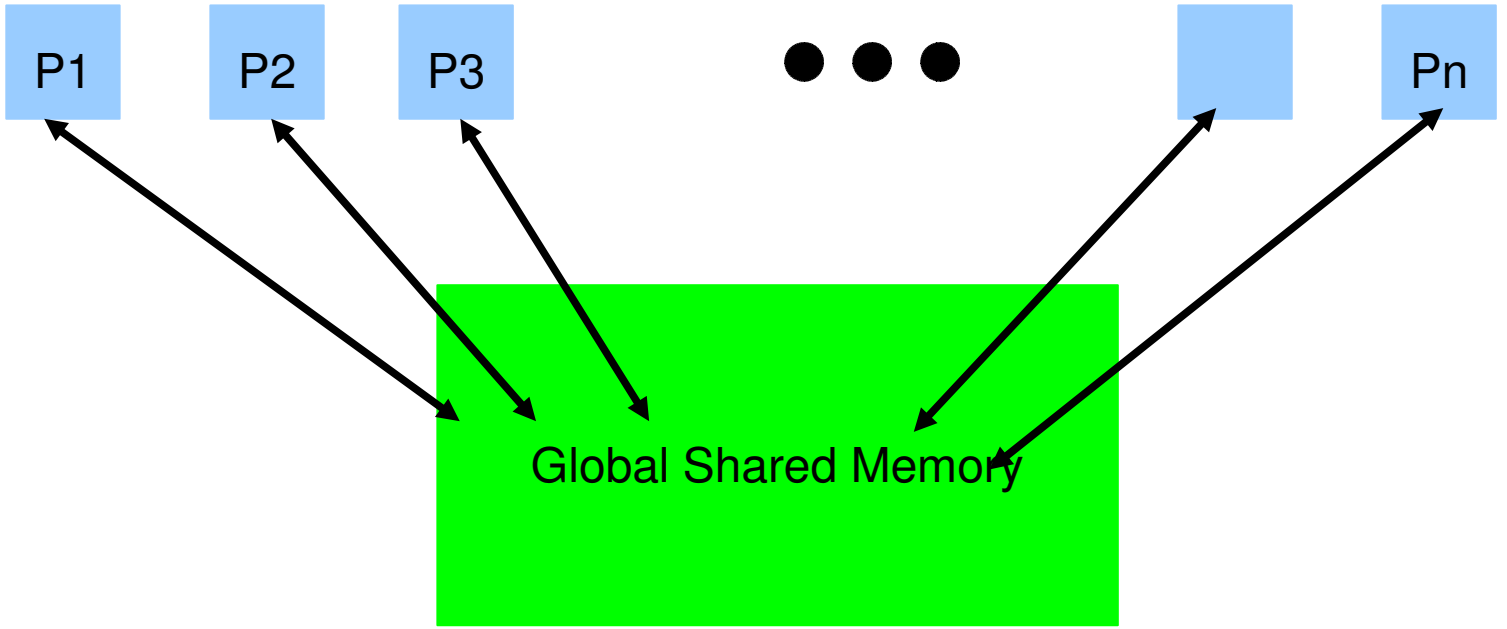
- GPU is not a very versatile architecture
- Number of cycles varies heavily even for simple operations.
  - 4 cycles for a multiply
  - 40 cycles for integer modulo
- Hence, add up the cycles for all the compute operations in a kernel.

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

Measuring Memory Access in a Kernel

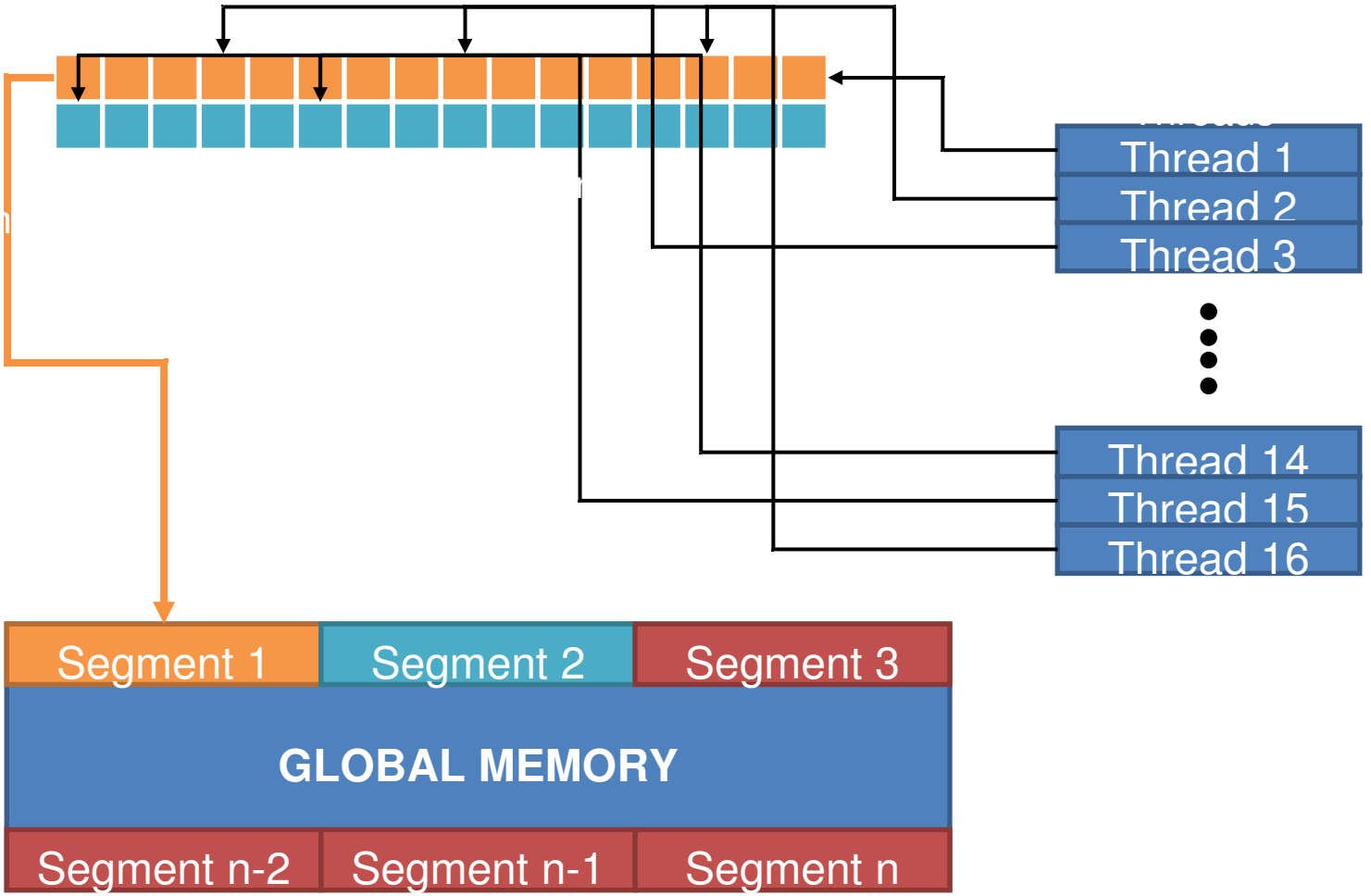
- GPU has a deep memory hierarchy
  - Global Memory
  - Shared Memory
  - Constant Memory
  - Texture Memory
- In this model, we'll focus only on the global and the shared memory.

PRAM Model for Global Memory Accesses



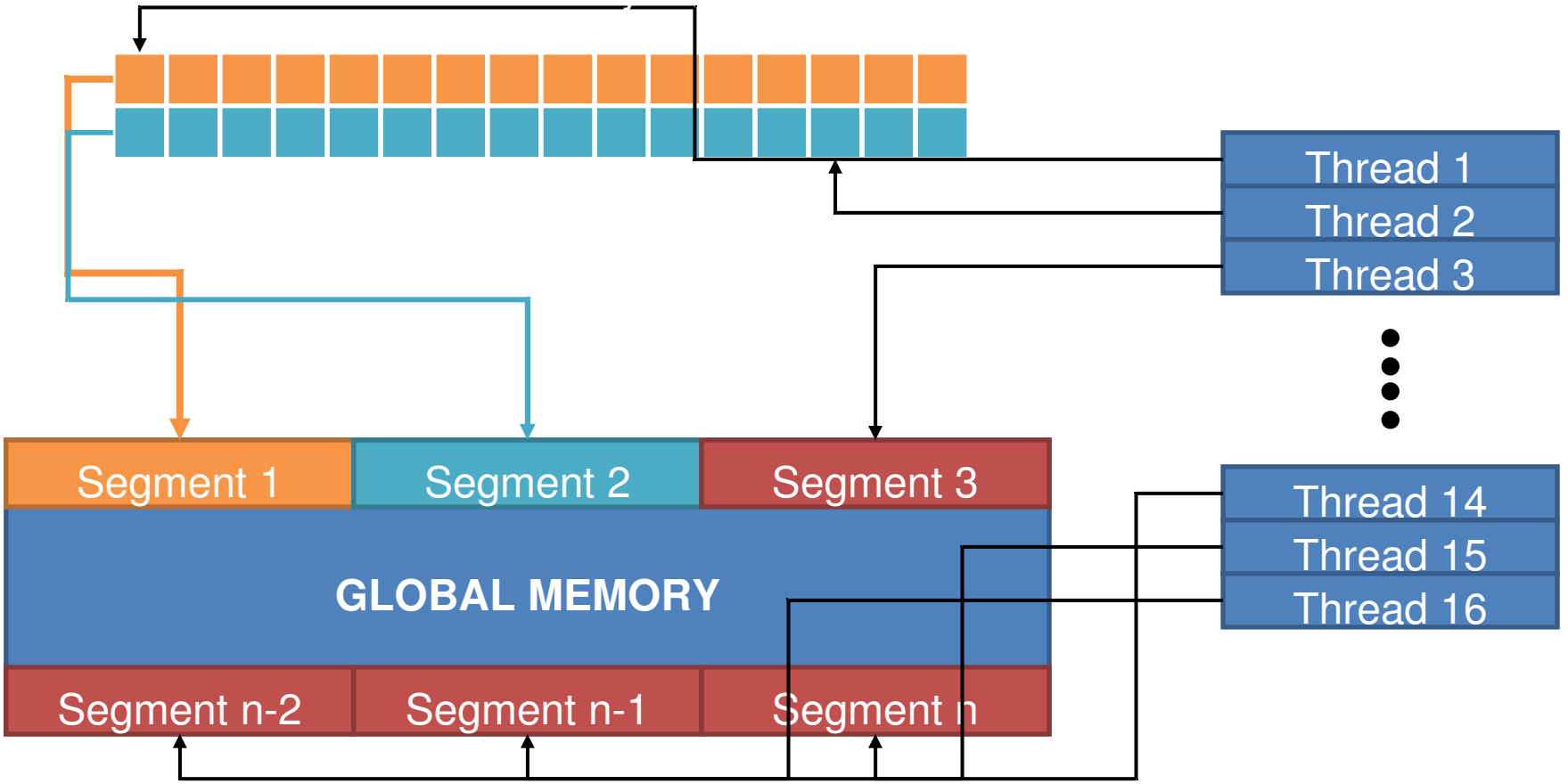
- An extension of the RAM model
  - Uniform cost for each memory access and computation.
- For the GPU, we need to think of a non-uniform PRAM model.
  - Non-uniformity depends on the nature of global access.

PRAM Model for Global Memory Accesses



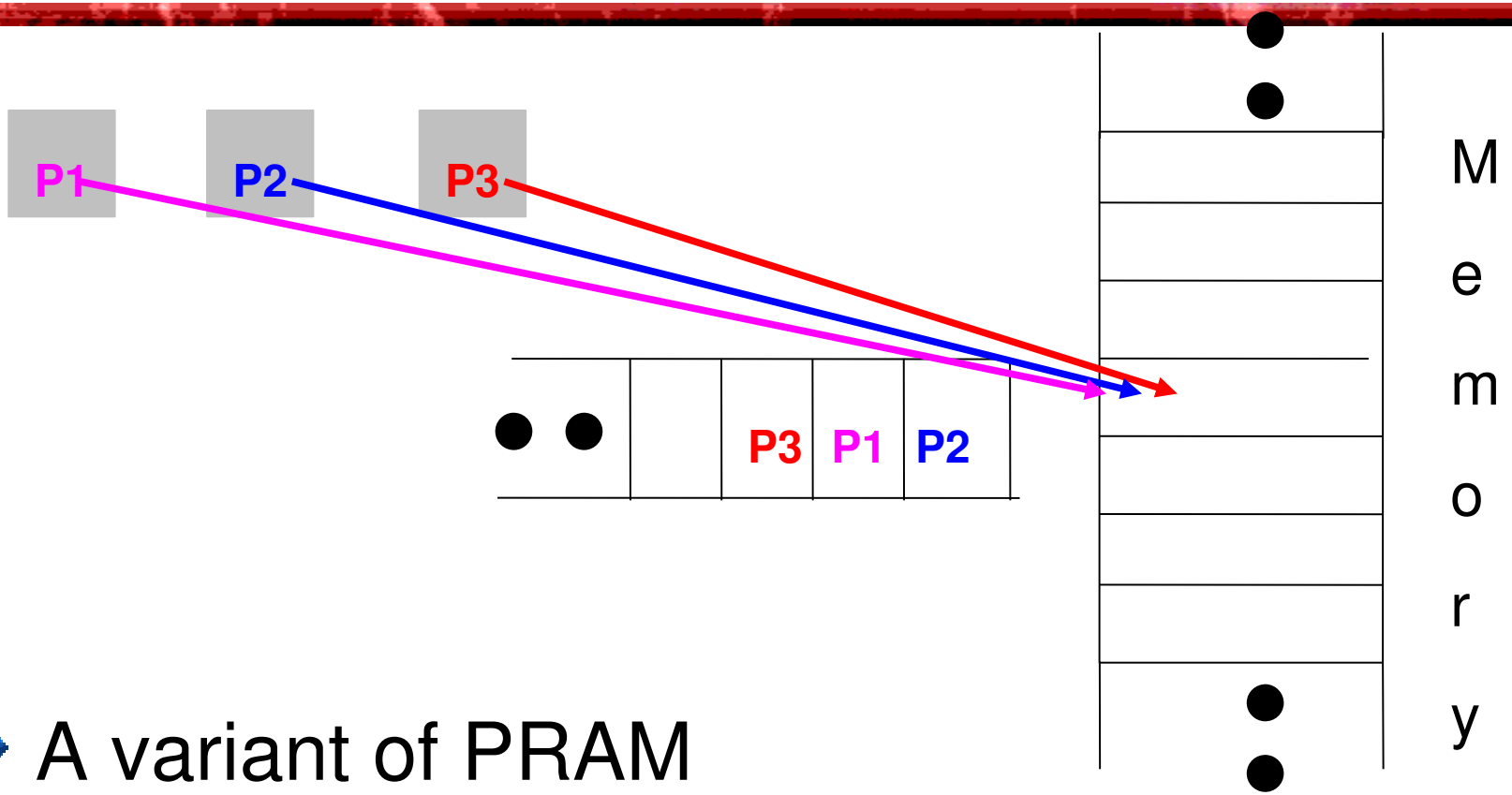
- On the GPU however global memory access cost varies
  - About 20 cycles for coalesced access [Nvidia manual]

PRAM Model for Global Memory Accesses



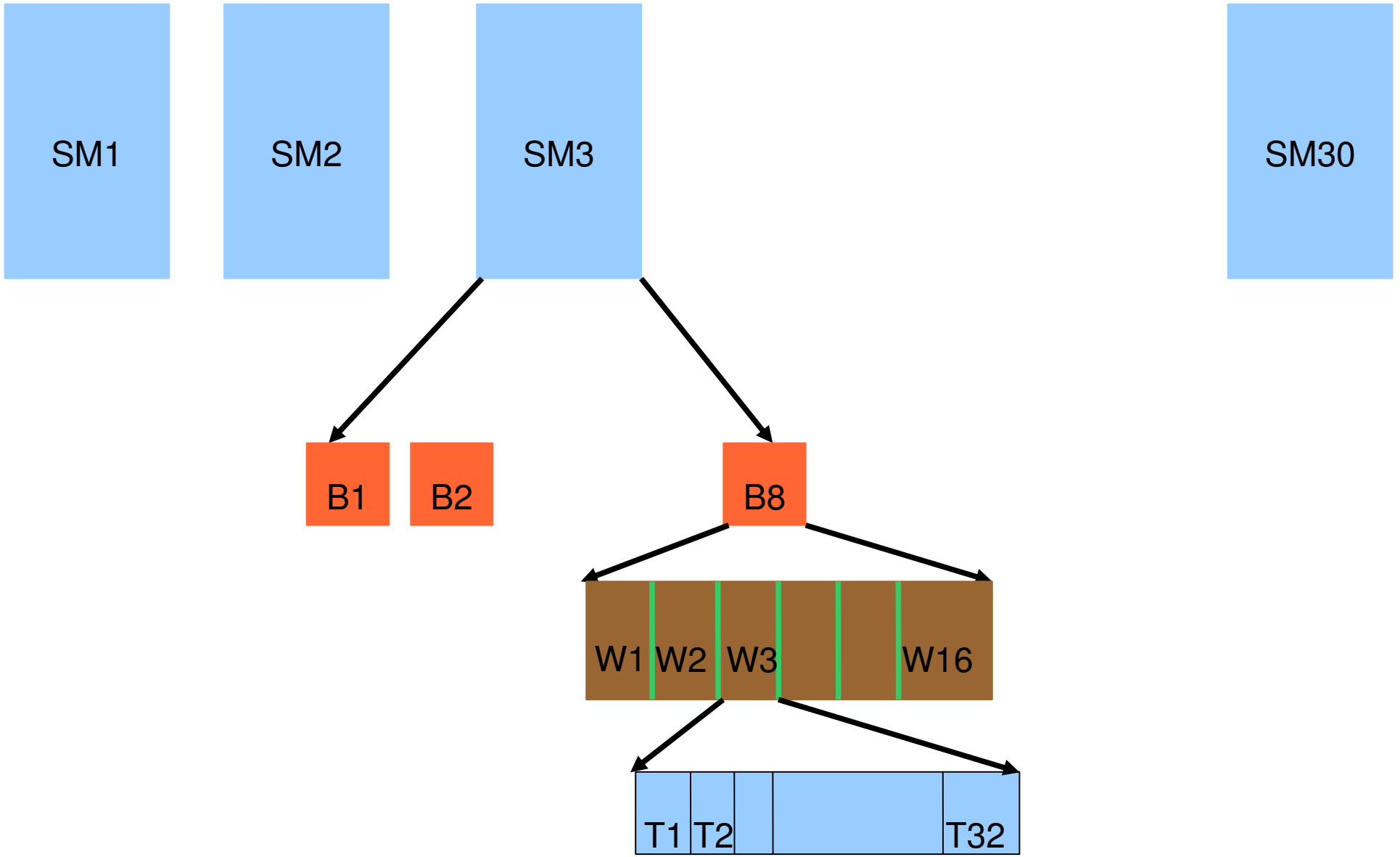
- On the GPU however global memory access cost varies
  - 400 – 600 cycles for non-coalesced access [Nvidia manual]

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

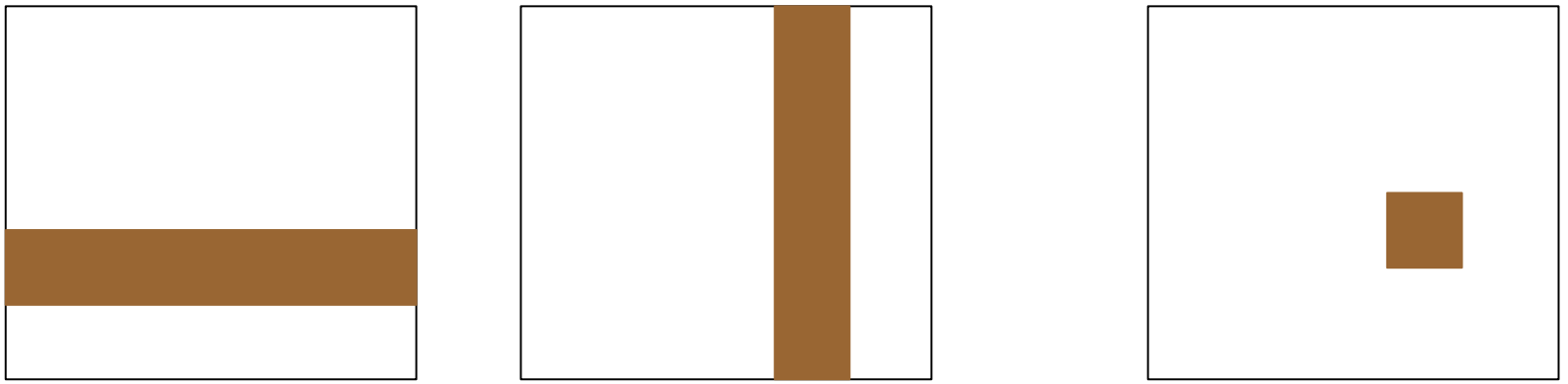
N_w : No. of warps/blocks

N_t : No. of threads/block

How to Obtain $C_T(K)$

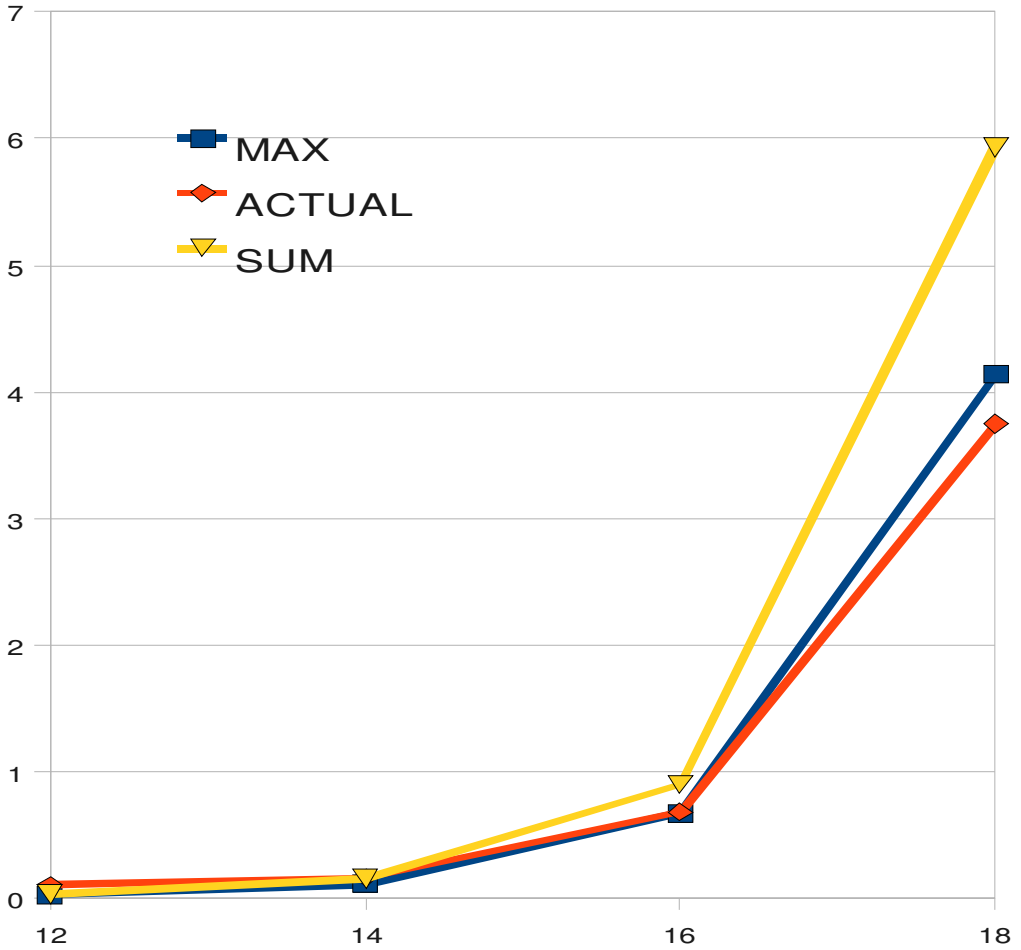
- Separately computing the compute cycles (N_{comp}) and the memory cycles (N_{memory}).
- Which of these dominates the other?
  - Depends on several factors: application, implementation, scheduling, ...
- But can use two possible scenarios
  - Application has good latency hiding: Take MAX
  - Poor latency 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



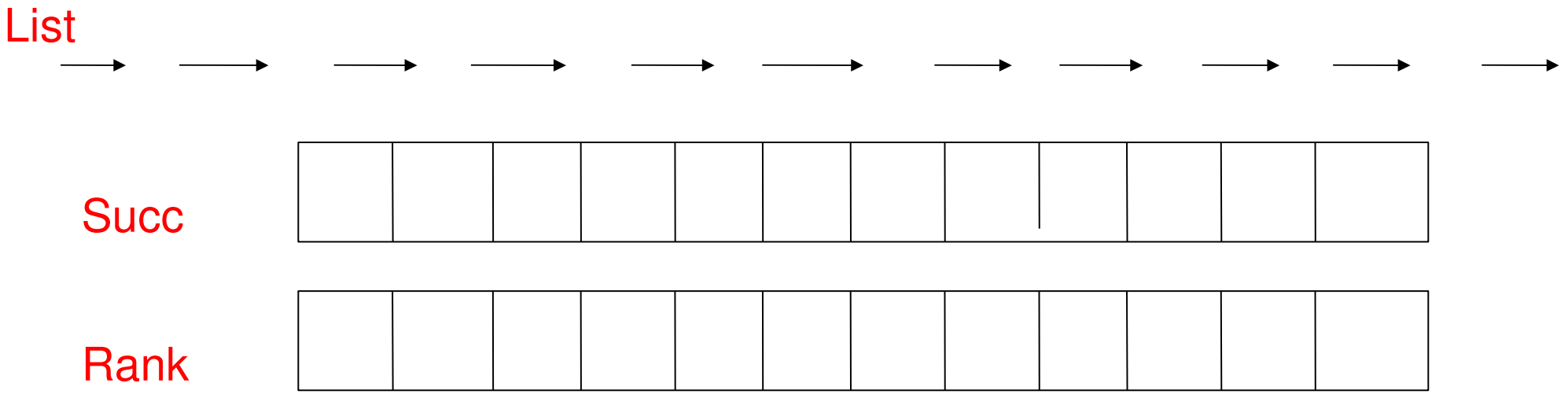
- A popular parallel computing case study
- Typically, compute intensive benefiting from coalesced reads and shared memory accesses.
- Each thread computes a block of the product matrix.
- See Nvidia manual for more details.

Case Study 1 – Matrix Multiplication



- Actual time mostly follows MAX at most places.

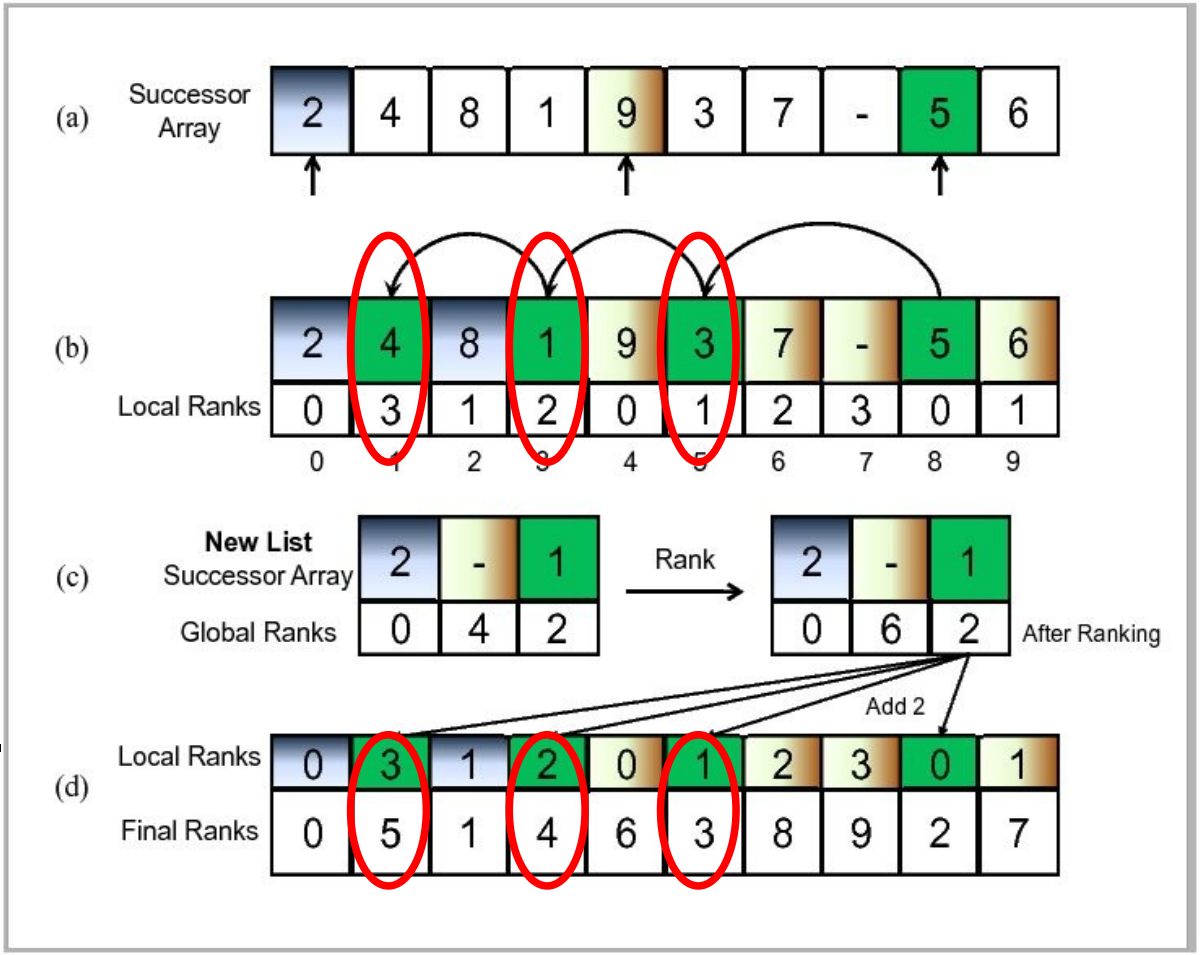
Case Study 2 – List Ranking



- 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.



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.

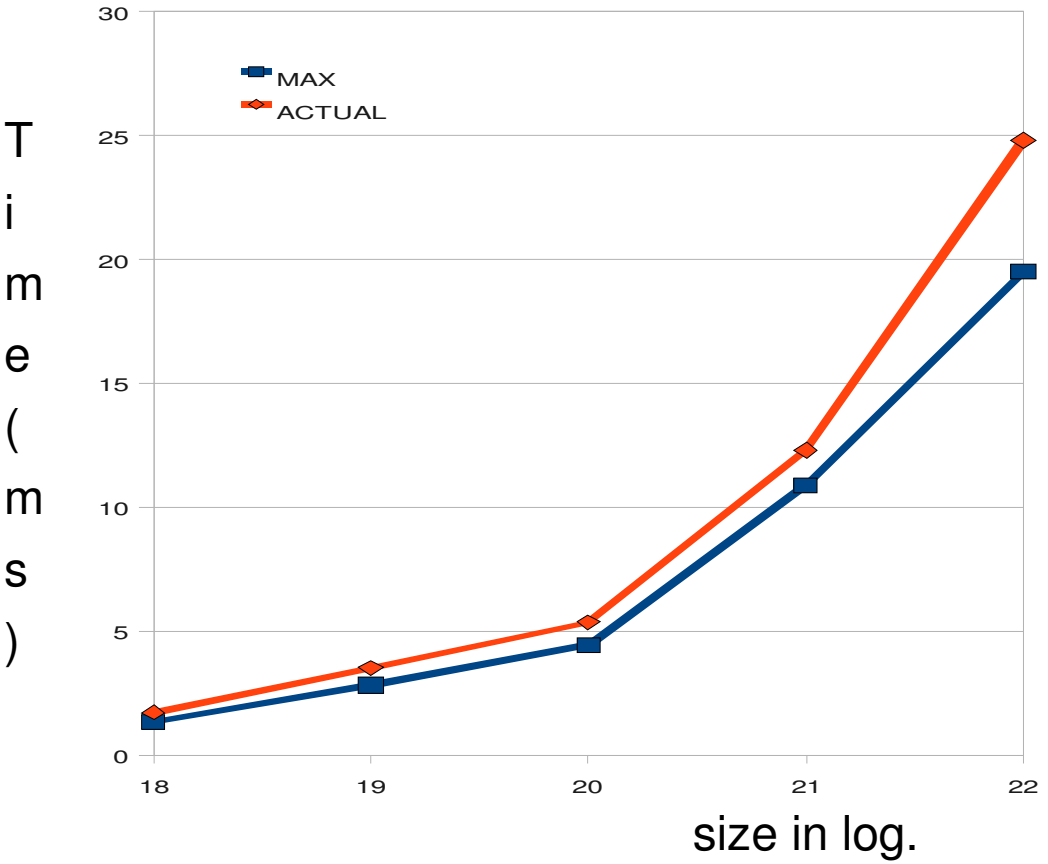
Analyzing the Local Ranking Kernel

- 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



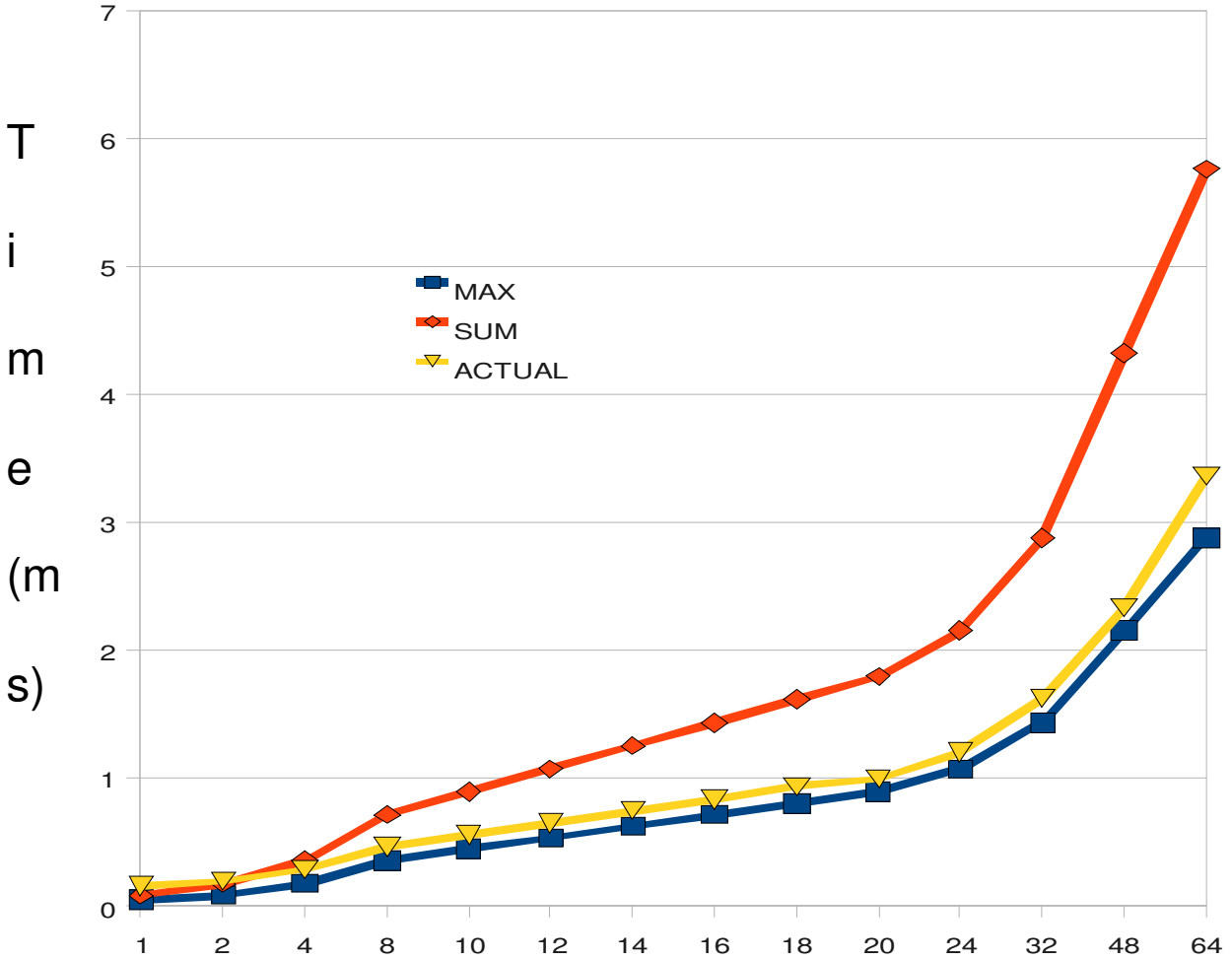
- Each thread ranks a sublist.
- Very little computation.

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.