Towards Efficient, Dynamic Parallelism for GPUs

Duane Merrill
Andrew Grimshaw

UNIVERSITY *of* VIRGINIA

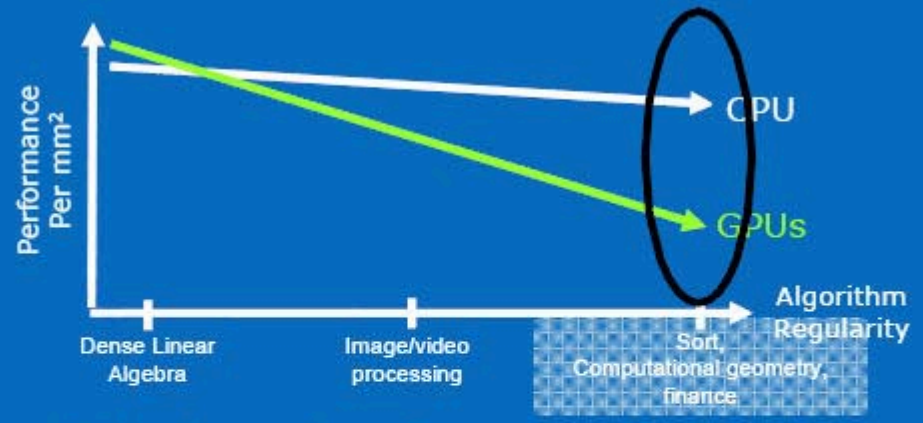
GPU Stream Machine Model

- Many, many concurrent threads of execution
  - All threads run the same program (*kernel*)
    - SIMD + SMT
  - Explicit control over memory storage hierarchy
    - Registers, fast local shared per core, global DRAM
- Report card**:
  - *Excels at*:
    - Flat data-parallelism (i.e., data-independent and statically-known data dependences)
  - ***Needs work***:
    - Dynamic, irregular, and nested parallelism

Lee et al. **Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *SIGARCH* 2010.

Report Card: CPU Territory

Algorithm Examples



- Sort, computational geometry, finance
  - Modest control flow
  - Sparse/Irregular data structures
  - Irregular communication between elements
- CPU Territory
  - General purpose features vital for software efficiency
  - Latency sensitive applications

Integer (32-bit) Sorting Rates

DEVICE	KEY-VALUE RATE (10^6 pairs / sec)		KEYS-ONLY RATE (10^6 keys / sec)	
Name	CUDPP Radix	Our SRTS Radix (speedup)	CUDPP Radix	Our SRTS Radix (speedup)
NVIDIA GTX 480		**775**		**1005**
NVIDIA Tesla C2050		**581**		**742**
NVIDIA GTX 285	134	**490** (3.7x)	199	**615** (2.8x)
NVIDIA GTX 280	117	**449** (3.8x)	184	**534** (2.6x)
NVIDIA Tesla C1060	111	**333** (3.0x)	176	**524** (2.7x)
NVIDIA 9800 GTX+	82	**189** (2.0x)	111	**265** (2.0x)
NVIDIA 8800 GT	63	**129** (2.1x)	83	**171** (2.1x)
NVIDIA Quadro FX5600	55	**110** (2.0x)	66	**147** (2.2x)
Intel Knight's Ferry MIC 32-core**			560	
Intel Core i7 quad-core **			240	
Intel Core-2 quad-core**			138	

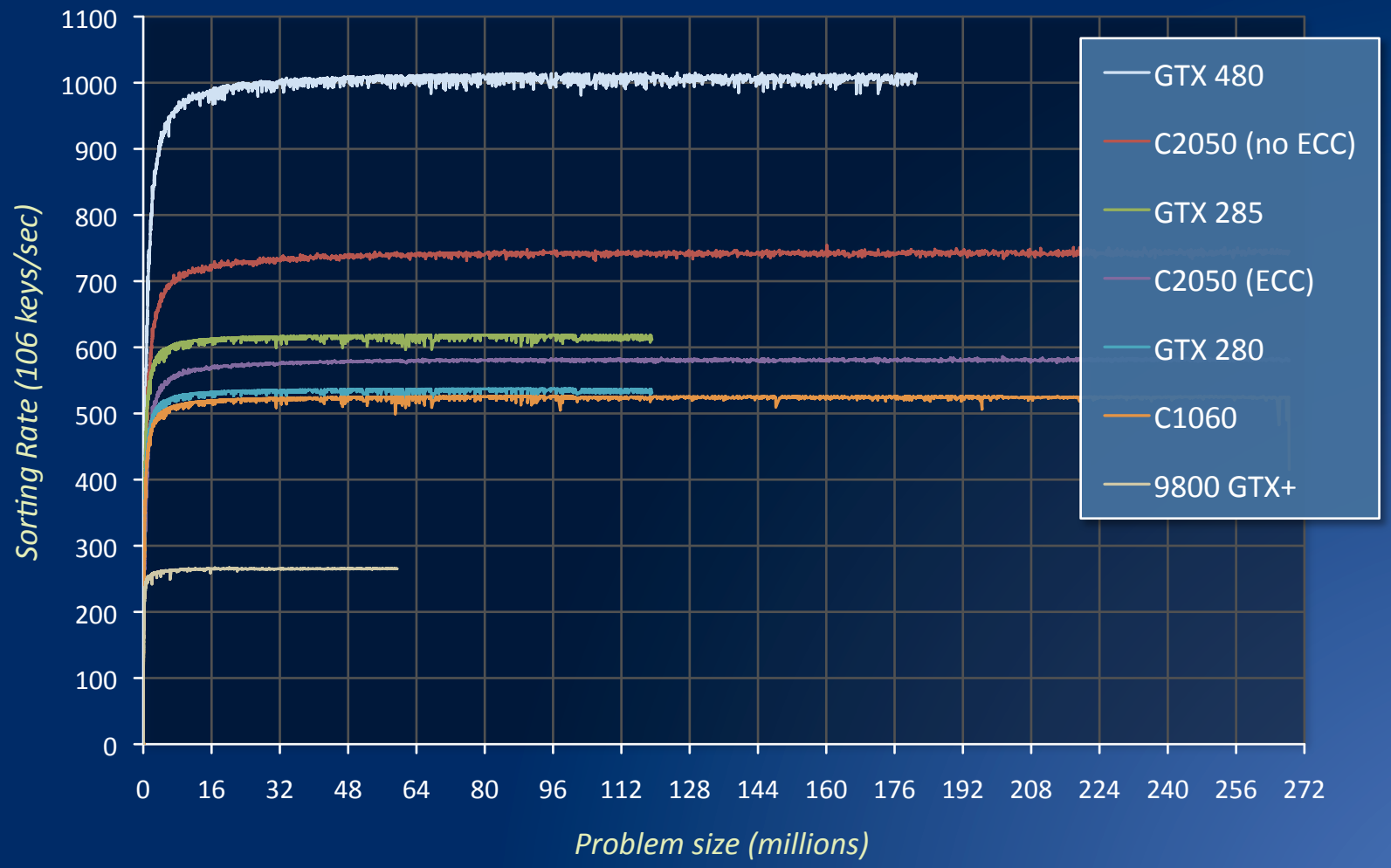
Satish et al., "Fast Sort on CPUs, GPUs and Intel MIC Architectures**," Tech Report 2010.

Integer (32-bit) Sorting Rates

DEVICE	KEY-VALUE RATE (10^6 pairs / sec)		KEYS-ONLY RATE (10^6 keys / sec)	
Name	CUDPP Radix	Our SRTS Radix (speedup)	CUDPP Radix	Our SRTS Radix (speedup)
NVIDIA GTX 480		**775**		**1005**
NVIDIA Tesla C2050		**581**		**742**
NVIDIA GTX 285	134	**490** (3.7x)	199	**615** (2.8x)
NVIDIA GTX 280	117	**449** (3.8x)	184	**534** (2.6x)
NVIDIA Tesla C1060	111	**333** (3.0x)	176	**524** (2.7x)
NVIDIA 9800 GTX+	82	**189** (2.0x)	111	**265** (2.0x)
NVIDIA 8800 GT	63	**129** (2.1x)	83	**171** (2.1x)
NVIDIA Quadro FX5600	55	**110** (2.0x)	66	**147** (2.2x)
Intel Knight's Ferry MIC 32-core**			560	
Intel Core i7 quad-core **			240	
Intel Core-2 quad-core**			138	

Satish et al., "Fast Sort on CPUs, GPUs and Intel MIC Architectures**," Tech Report 2010.

Integer (32-bit) Sorting Rates



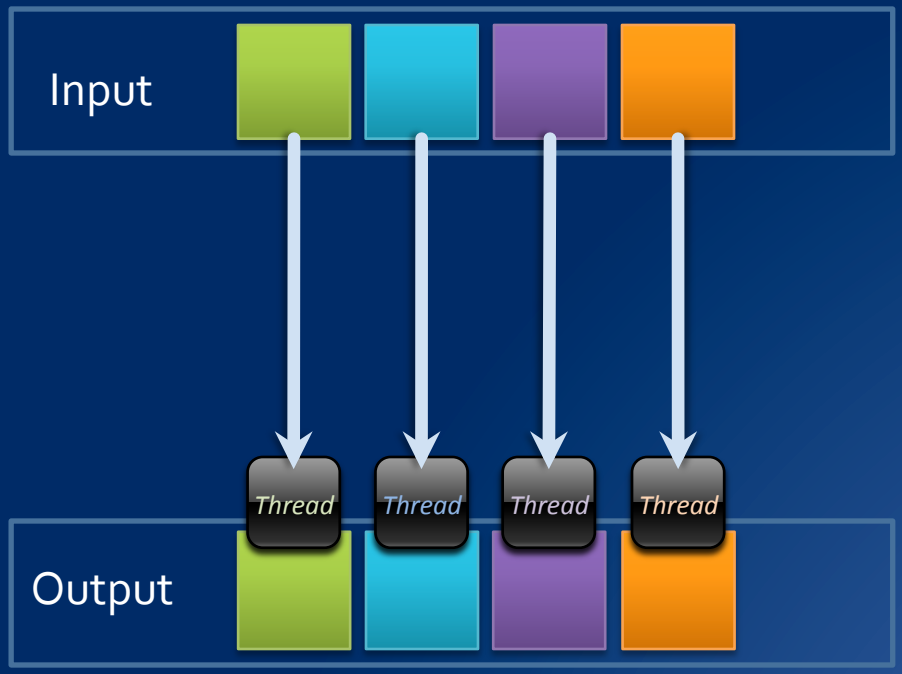
Presentation Overview

- Performance Strategies
  - Design patterns and idioms for program composition
- Challenges for the Programming Model
  - Burdens these techniques place upon the programming model / toolkit

Our Problem Scope:

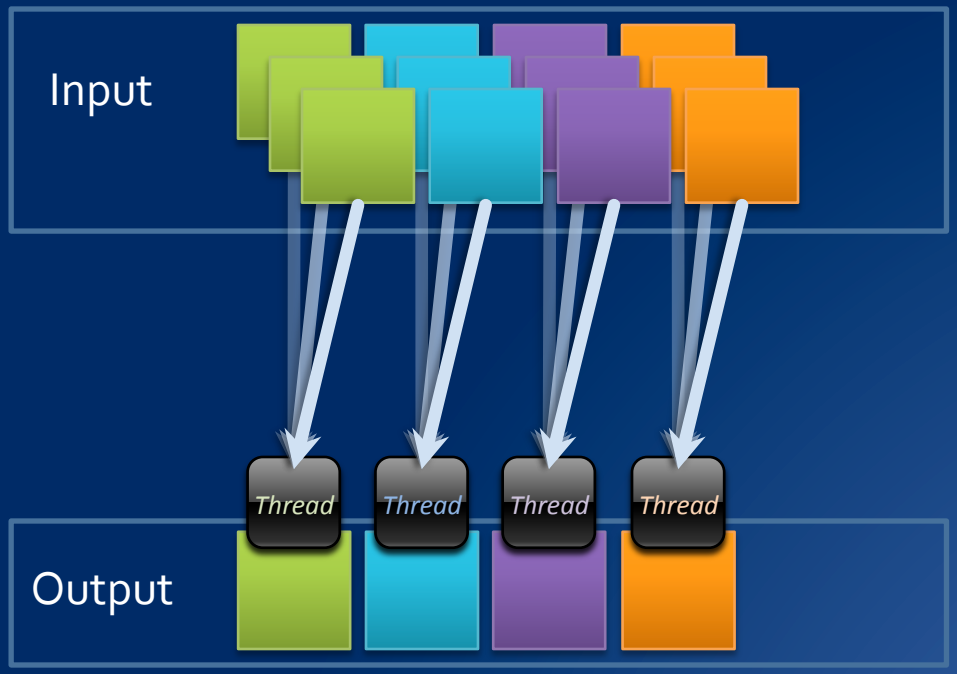
Thread decompositions with variable and dynamic output production

(a) Single input dependence



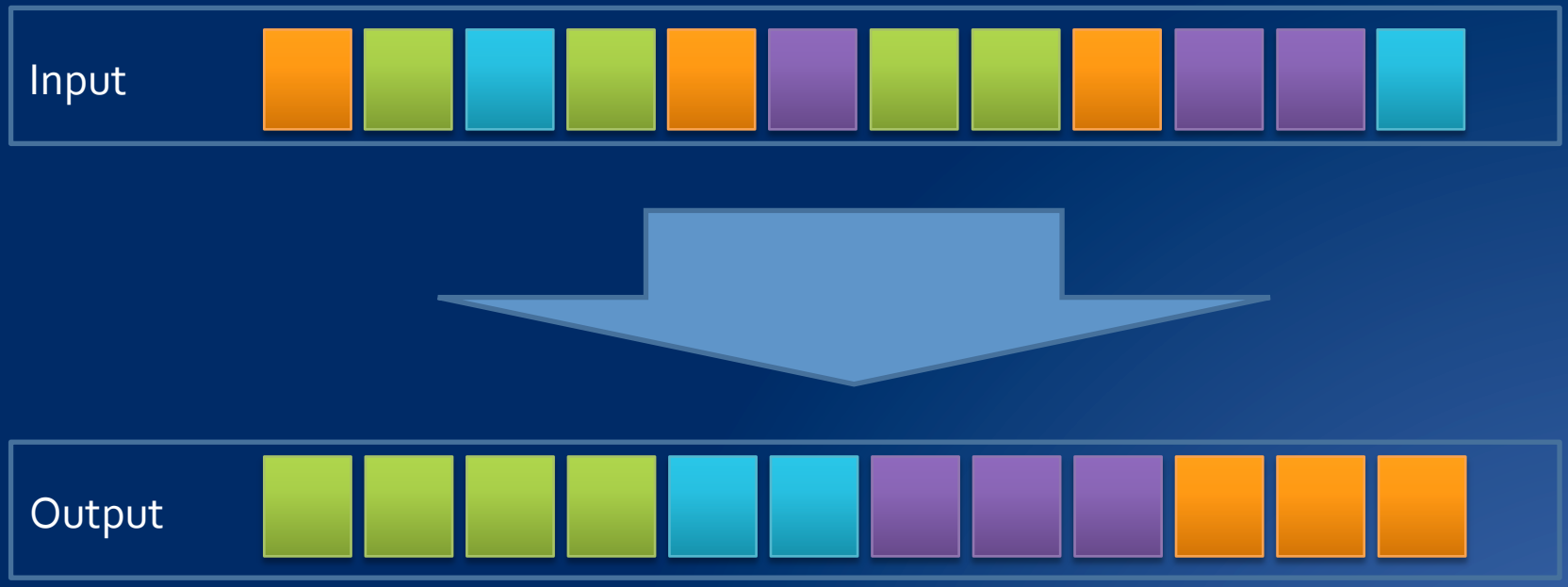
- Each output has a dependence upon a single input element
  - Threads are decomposed by output element
  - Input and output indices are static functions of thread-id
- E.g., scalar operations

(b) Neighborhood input dependences



- Each output has dependences upon a bounded subset of the input
  - Threads are decomposed by output element
  - The output (and at least one input) index is a static function of thread-id
- E.g., matrix / vector multiply

(c) Global input dependences



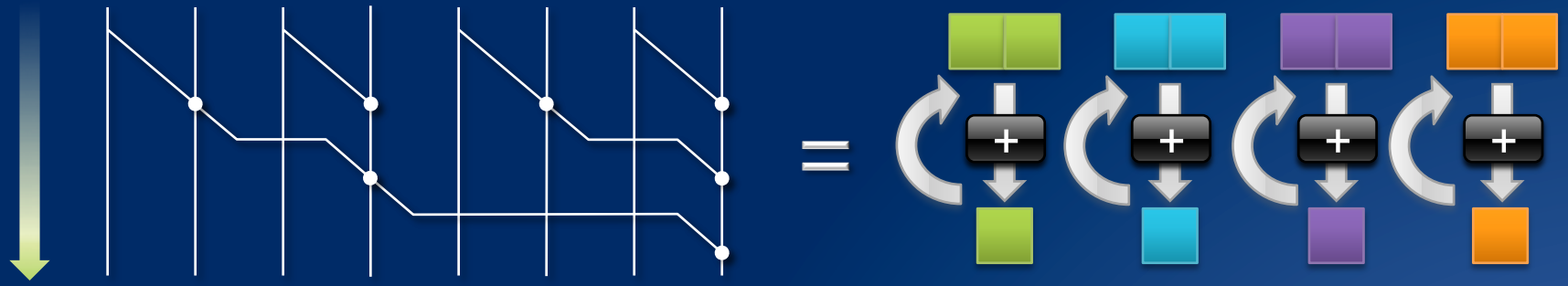
- Each output element has dependences upon any / all input elements
- E.g., sorting, reduction, compaction, duplicate removal, histogram generation, etc.

Composing global transformations

- The GPU machine model is designed for **(a)** ***local*** and **(b)** ***neighborhood*** transformations
  - ***(c) globally-dependent*** transformations must be constructed from multiple passes of Neighborhood transformations

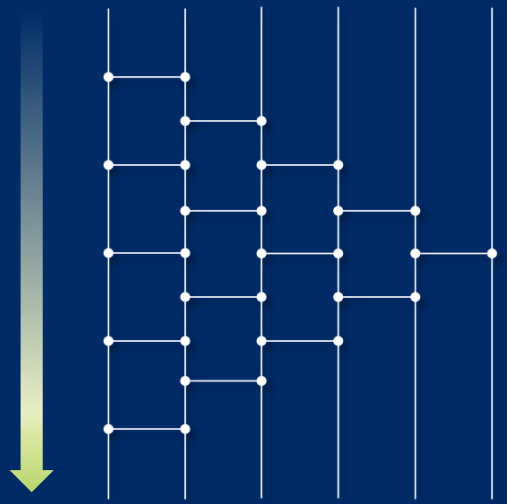
- The "straightforward" thread decomposition:
  - Threads are decomposed by output element
  - Repeatedly iterate over recycled input streams
  - Output stream size is statically known before each pass

Sometimes facilitates work-optimal methods

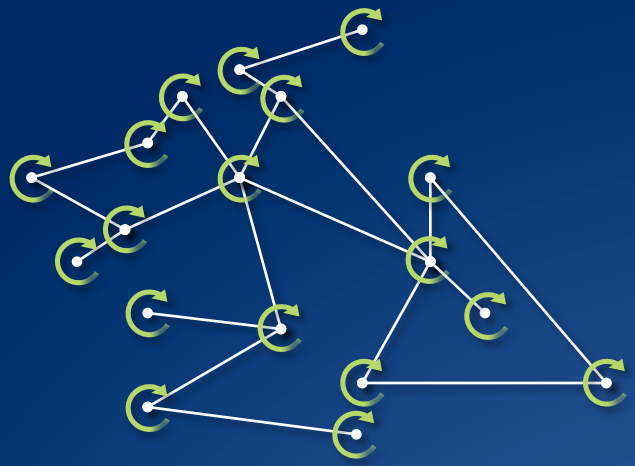


- E.g, reduction
  - O(n) global work from passes of pairwise-neighbor-reduction
  - Static dependences, uniform output
- E.g., Fast Fourier transform

Problem: Sometimes only facilitates work-inefficient methods

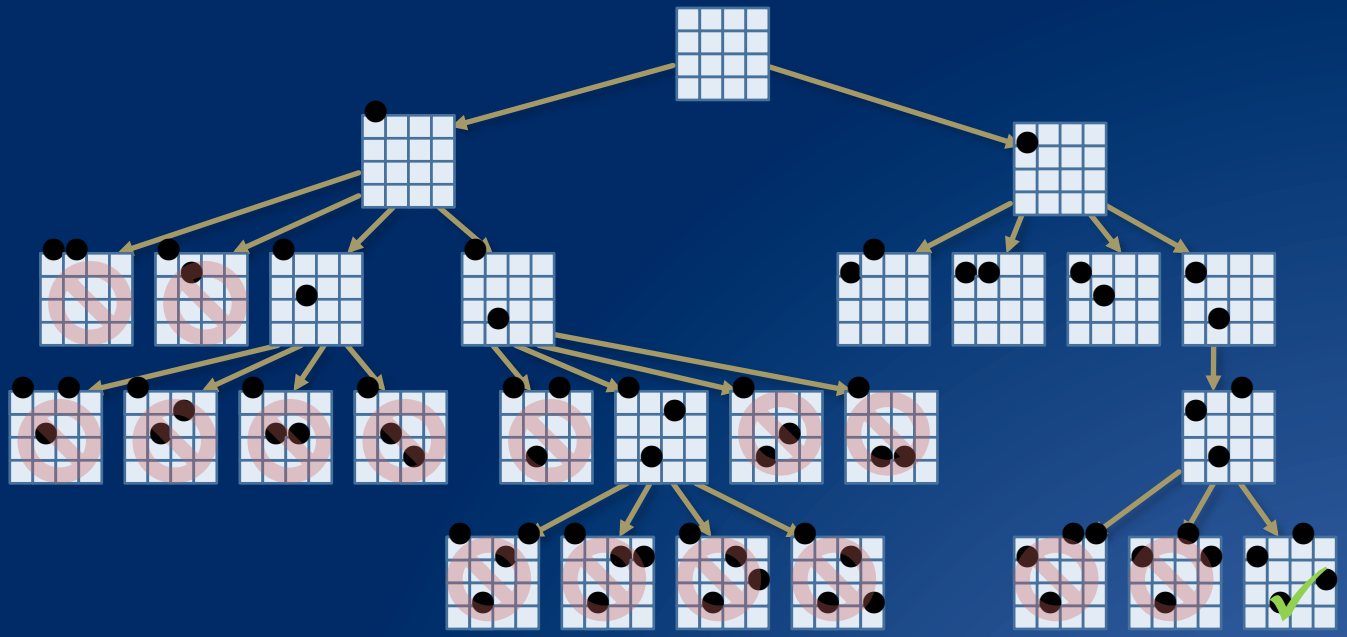


- E.g., sorting networks
  - Repeated, deterministic pairwise compare-smem
    - Bubble sort is $O(n^2)$
    - Bitonic sort is $O(n\log^2 n)$
    - Want $O(n\log n)$ comparison or $O(kn)$ radix sorting
  - Need partitioning: dynamic, cooperative allocation



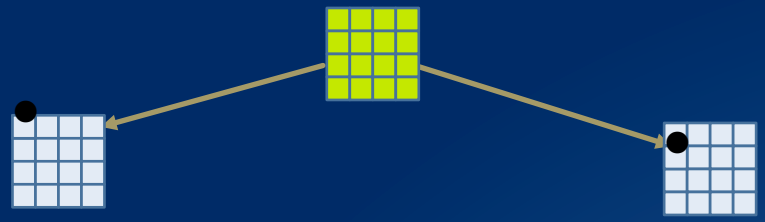
- E.g., graph traversal
  - Repeatedly check each vertex or edge
    - Such breadth-first search is $O(V^2)$
    - Want $O(V + E)$ BFS
  - Need queue: dynamic, cooperative allocation

Problem: Sometimes is completely insufficient



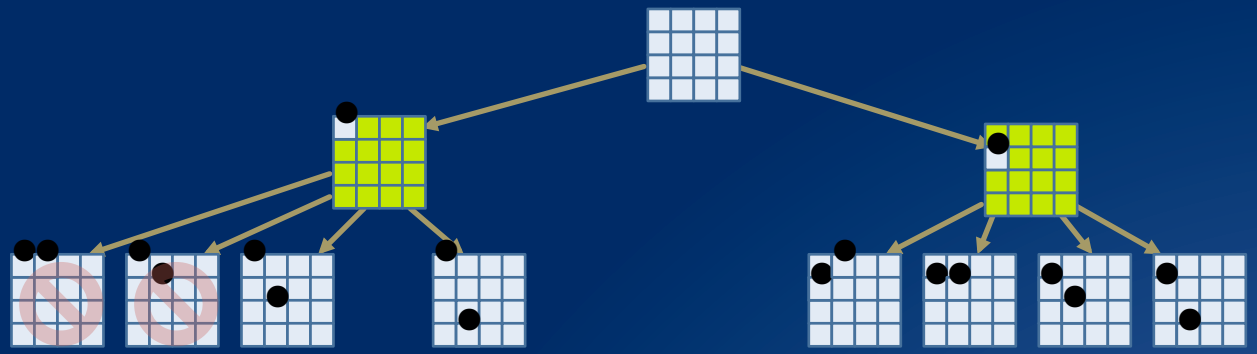
- E.g., parallel search space exploration
  - Variable output per thread
  - Need dynamic, cooperative allocation

Problem: Sometimes is completely insufficient



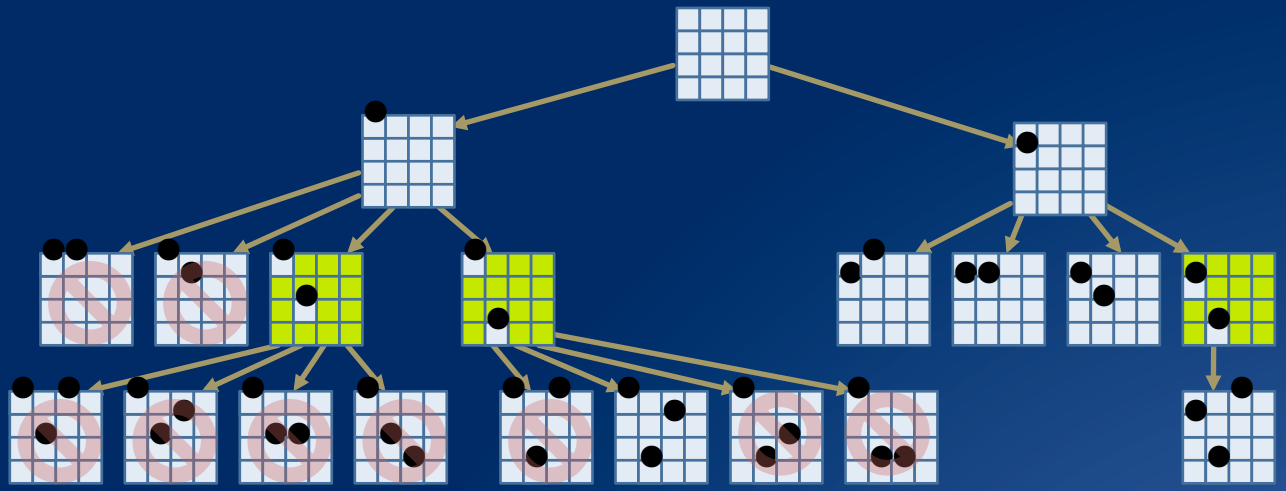
- E.g., parallel search space exploration
  - Variable output per thread
  - Need dynamic, cooperative allocation

Problem: Sometimes is completely insufficient



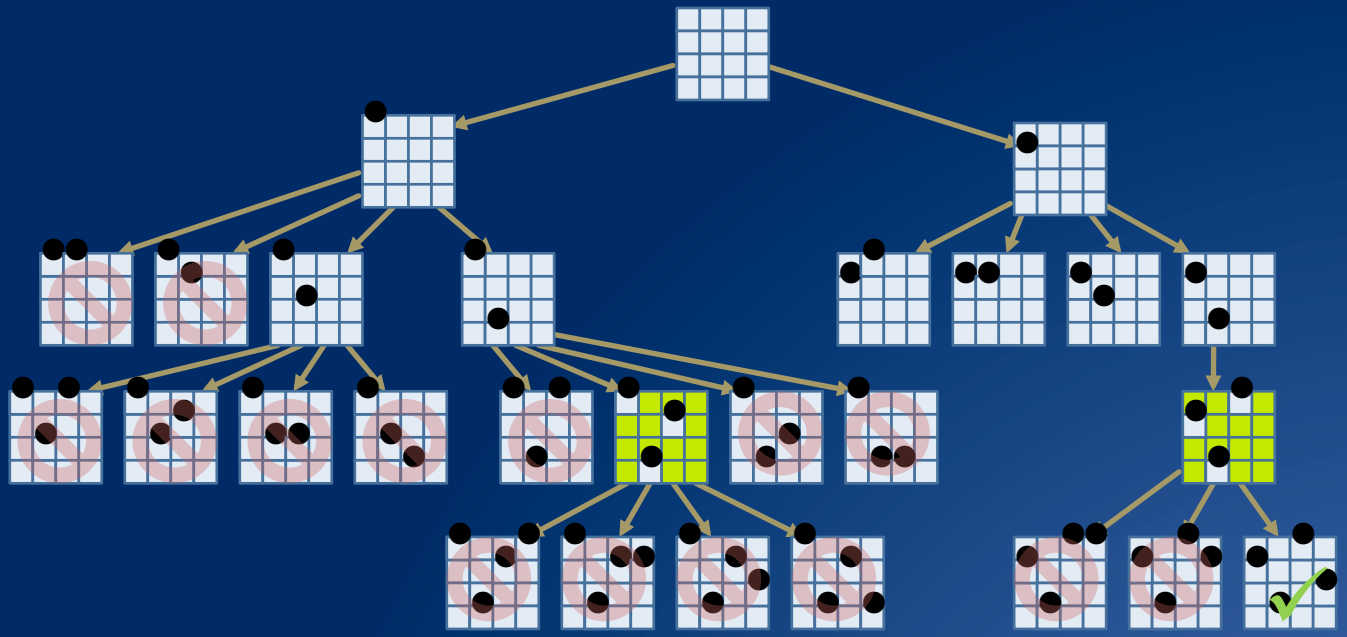
- E.g., parallel search space exploration
  - Variable output per thread
  - Need dynamic, cooperative allocation

Problem: Sometimes is completely insufficient



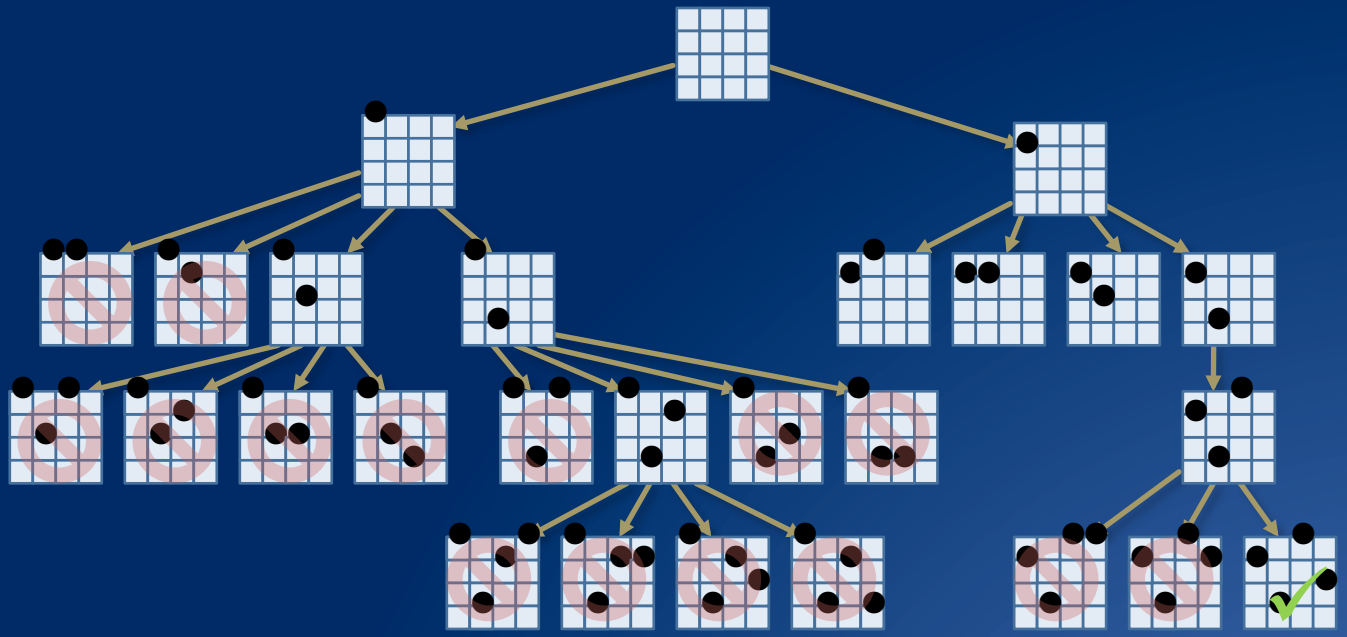
- E.g., parallel search space exploration
  - Variable output per thread
  - Need dynamic, cooperative allocation

Problem: Sometimes is completely insufficient



- E.g., parallel search space exploration
  - Variable output per thread
  - Need dynamic, cooperative allocation

Problem: Sometimes is completely insufficient

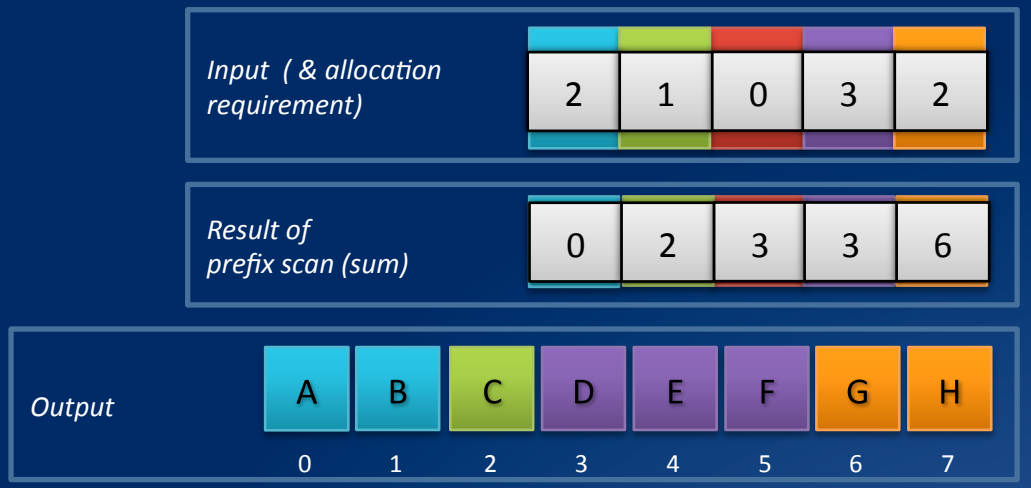


- E.g., parallel search space exploration
  - Variable output per thread
  - Need dynamic, cooperative allocation

Dynamic, irregular, and nested parallelism

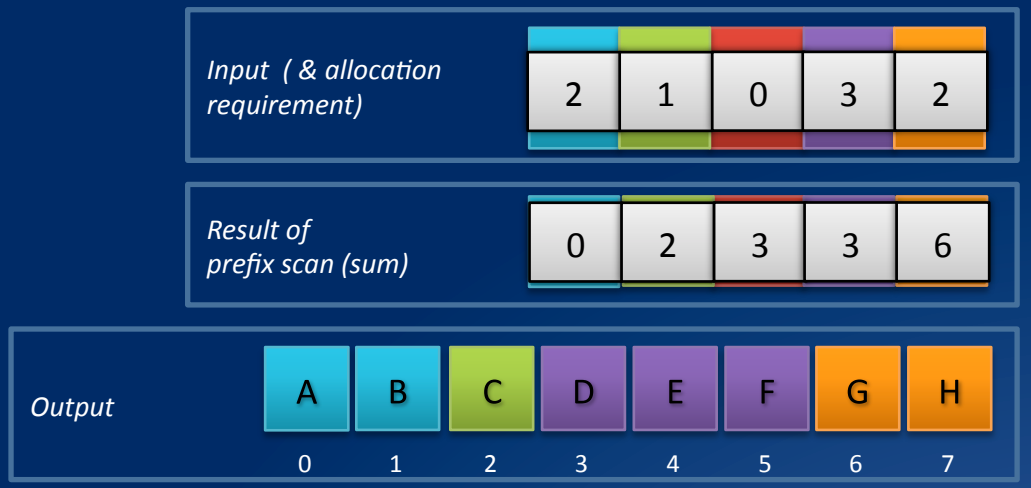
- What we want:
  1. Work-optimal implementations for problems with dynamic dependences...
  2. ...that fit the machine model well
- Use alternative thread decomposition strategy:
  - Input-centric decomposition
    - Input indices are a static function of thread-id, but output indices are completely dynamic
  - A generalized allocation problem
    - *"I may write zero or more output items, and I need to cooperate with everyone to figure out where they go"*
  - Need efficient means of reservation/allocation
    - Parallel prefix scan (and relaxations / generalizations)

Prefix Scan



- Each output index is computed to be the sum of the previous input indices
  - *O*(*n*) work
  - For allocation: use scan results as a scattering vector
  - Origins in adder circuitry, popularized as a parallel primitive by Blelloch et al. in the '90s
- Fits the GPU machine model well
  - Merrill et al. **Parallel Scan for Stream Architectures**. Technical Report CS2009-14, Department of Computer Science, University of Virginia. 2009

Prefix Scan



- Each output index is computed to be the sum of the previous input indices
  - *O*(*n*) work
  - For allocation: use scan results as a scattering vector
  - Origins in adder circuitry, popularized as a parallel primitive by Blelloch et al. in the '90s
- Fits the GPU machine model well
  - Merrill et al. **Parallel Scan for Stream Architectures**. Technical Report CS2009-14, Department of Computer Science, University of Virginia. 2009

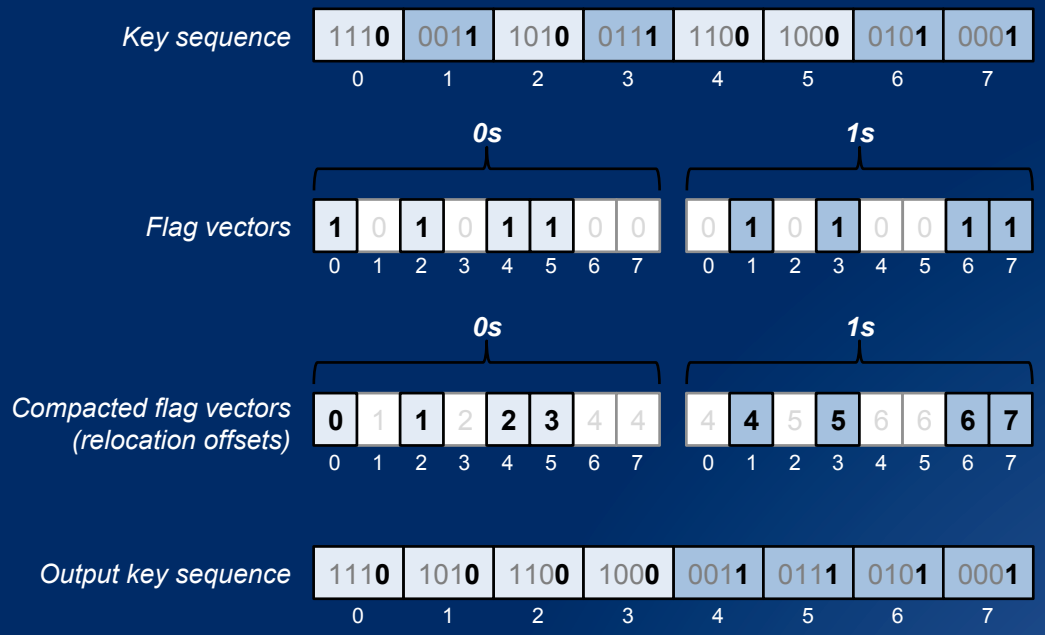
Interactive portion of the talk

(Show of hands please)

- Taken (or taught) an OS course?
- Had a unit on process synchronization?
- Covered multiple producers and consumers?
- Learned how to protect the queue with locks and mutexes?
- Learned how to protect the queue with prefix-sum and barriers?

- Mindset: ~~*cooperation == threads + locks*~~

Prefix Scan for Radix Sorting

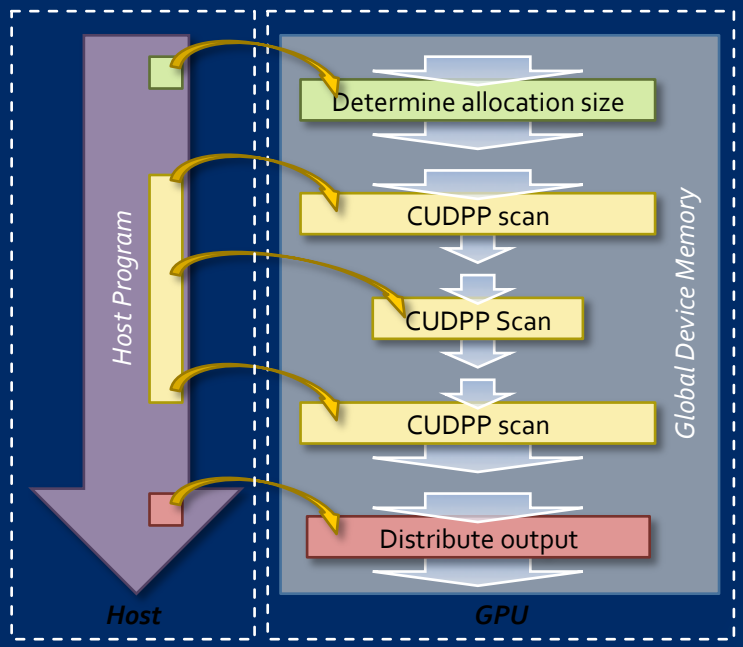


- For radix sorting passes
  - 0/1-flag each key as having a digit of 0,1,2,3, etc.
  - Scan flag vectors for radix r digits
  - Relocate keys into bins for each digit

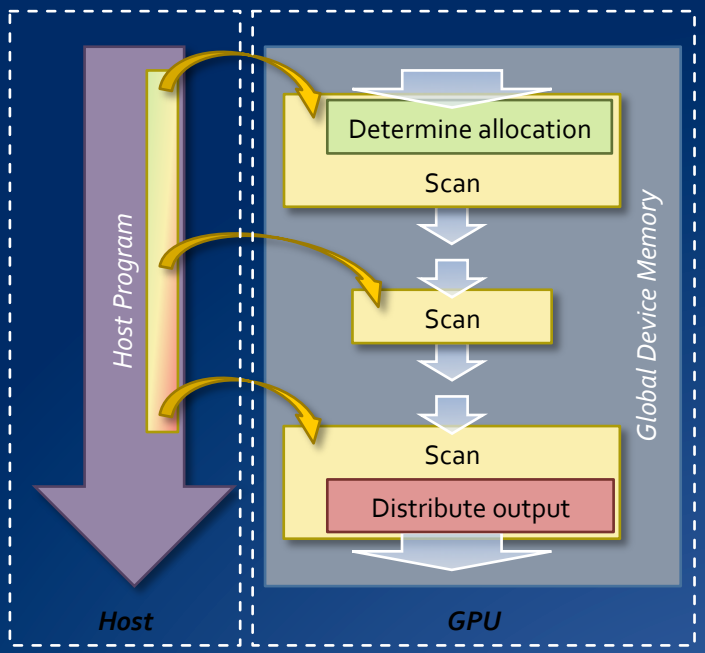
Kernel Fusion

and the efficient prefix-scan “runtime”

Kernel Fusion



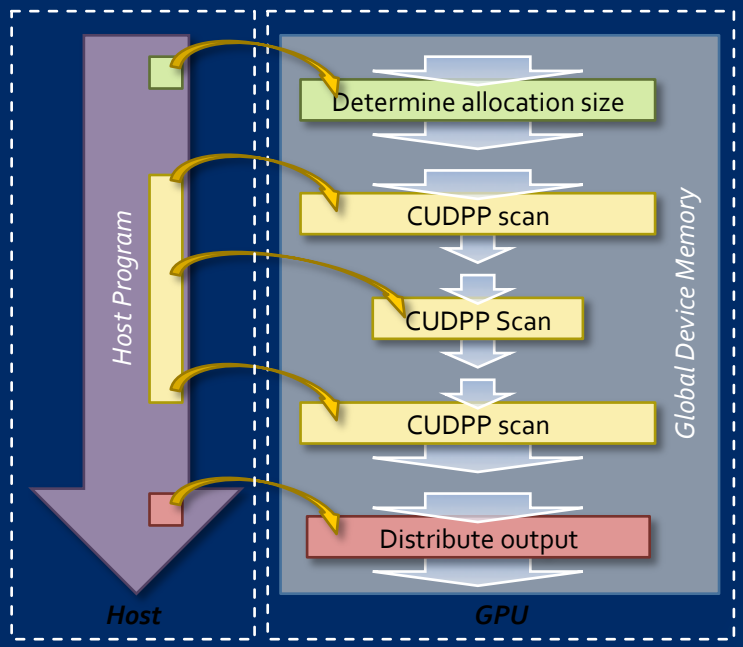
Un-fused



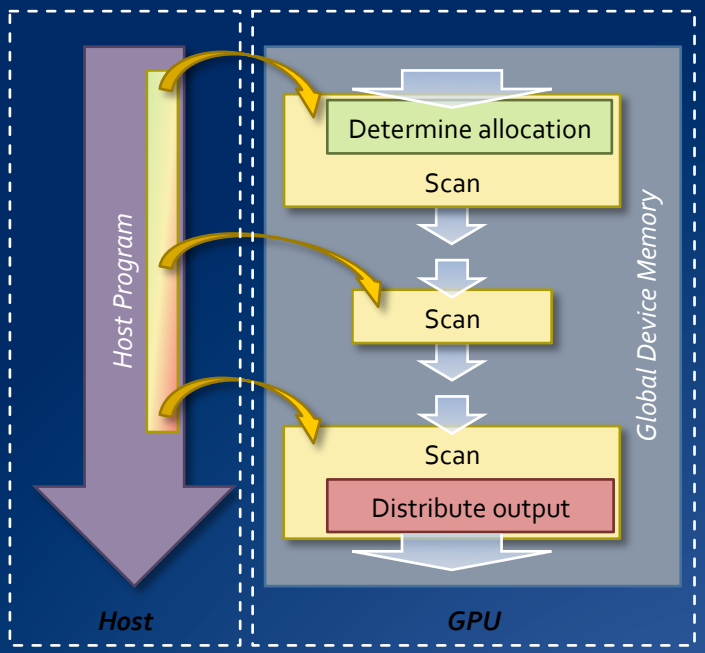
Fused

- Three concepts:
  1. Propagate live data between orthogonal steps in fast registers / smem
  2. Use scan (or variant) as a “runtime” for everything.
  3. Heavy SMT (over-threading) yields usable “bubbles” of free computation

Kernel Fusion



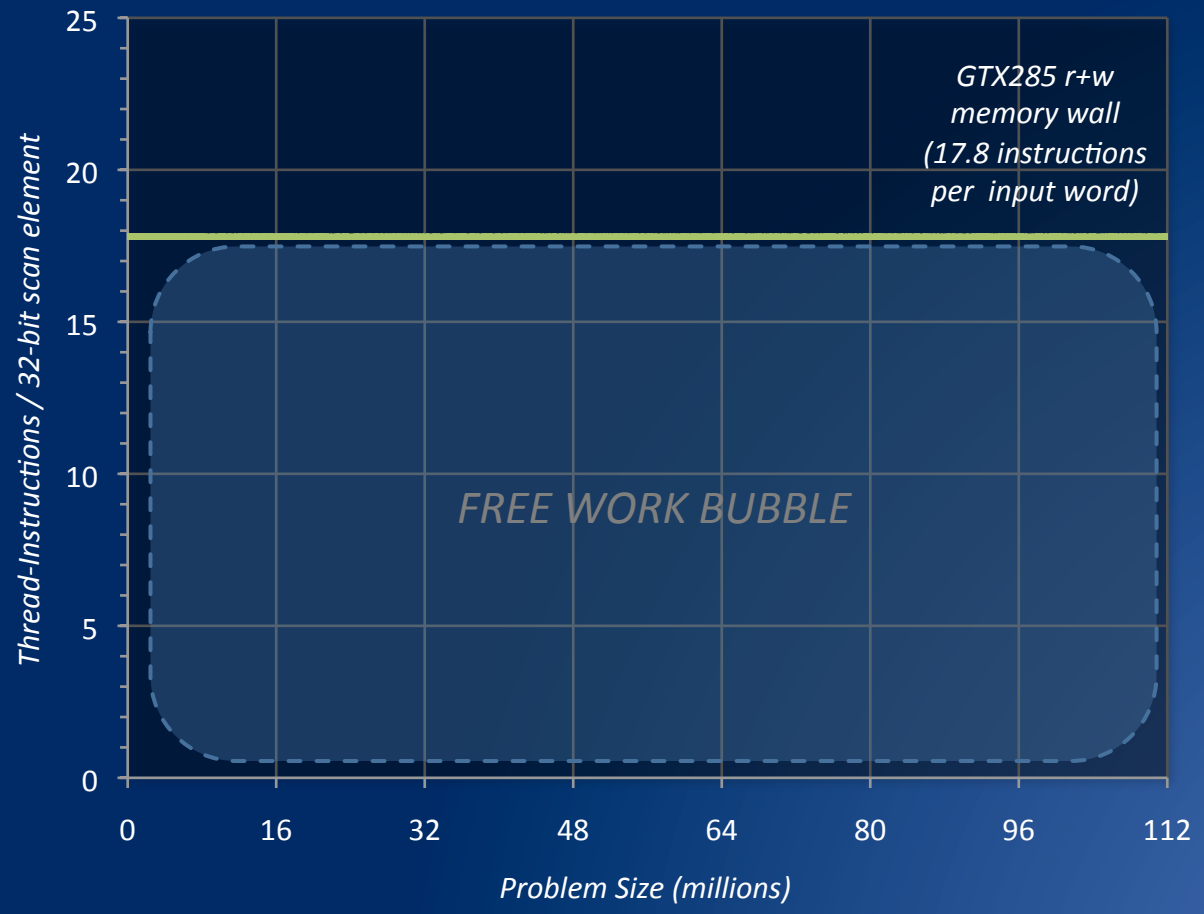
Un-fused



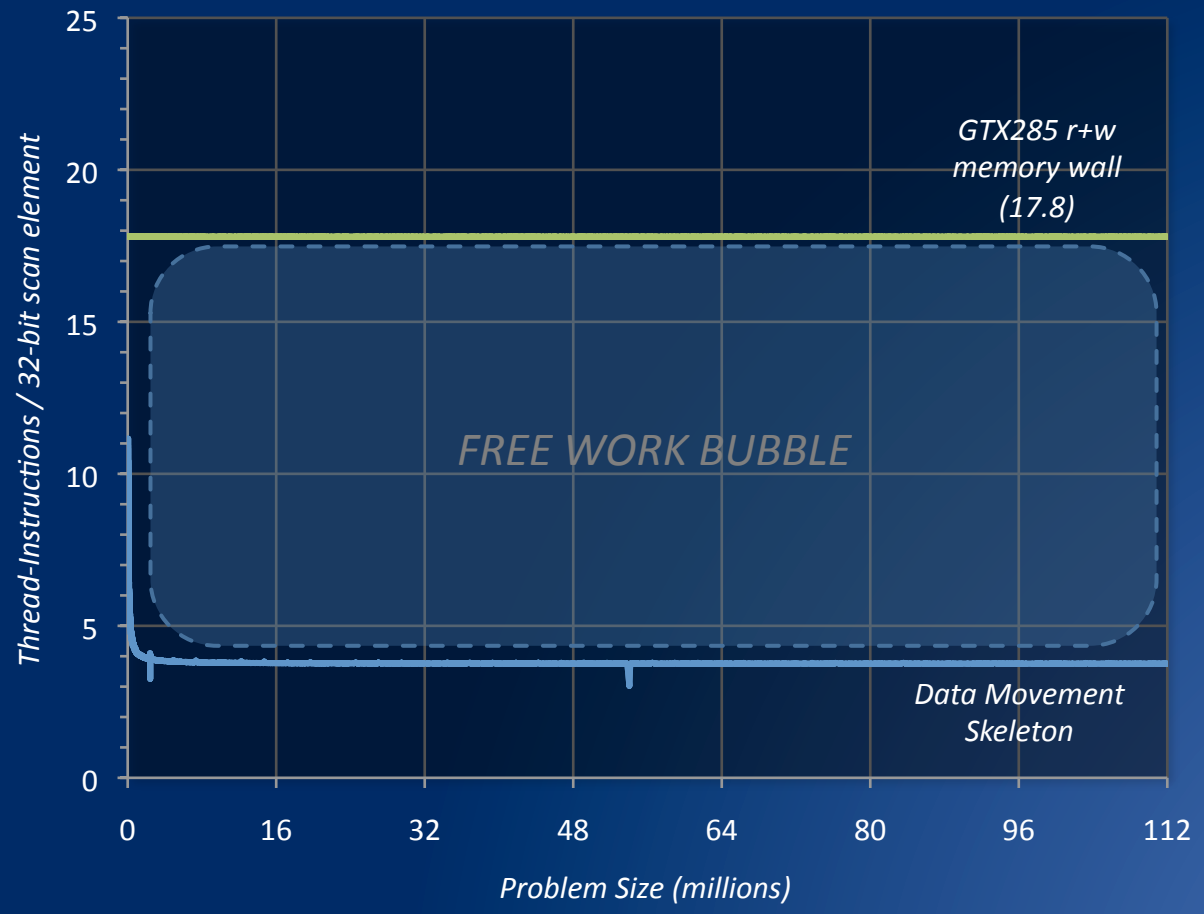
Fused

- Three concepts:
  1. Propagate live data between orthogonal steps in fast registers / smem
  2. Use scan (or variant) as a “runtime” for everything.
  3. Heavy SMT (over-threading) yields usable “bubbles” of free computation

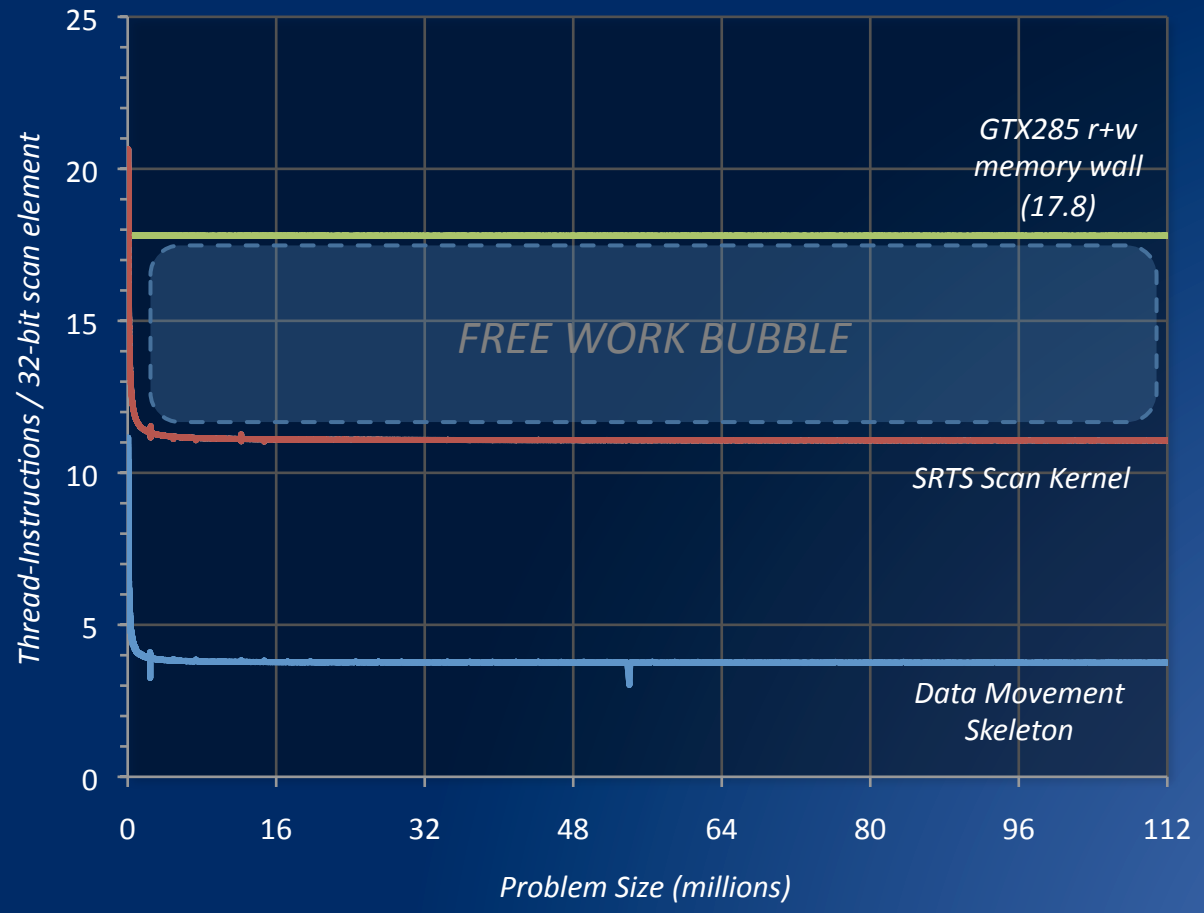
Read + Write Kernel Memory Wall



... after data movement instructions

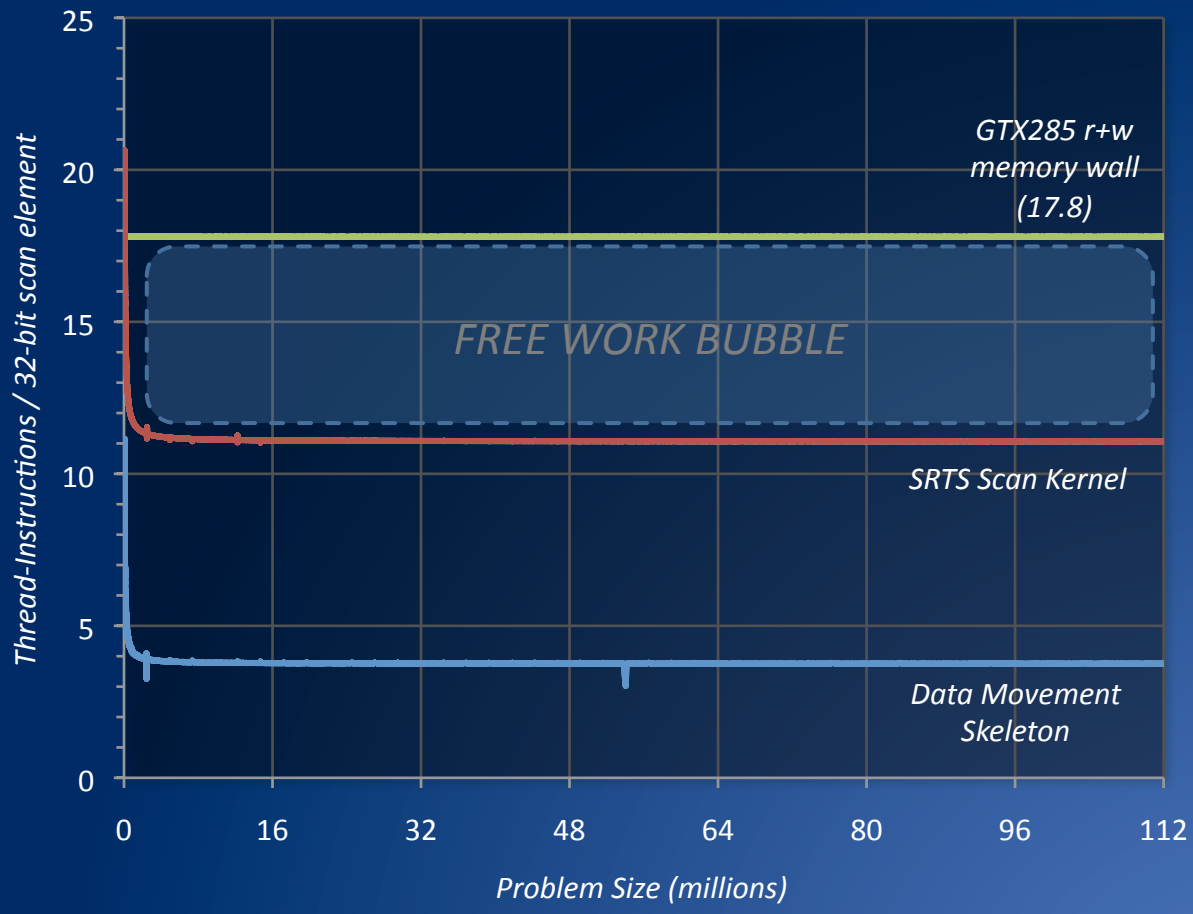


... after prefix-scan runtime



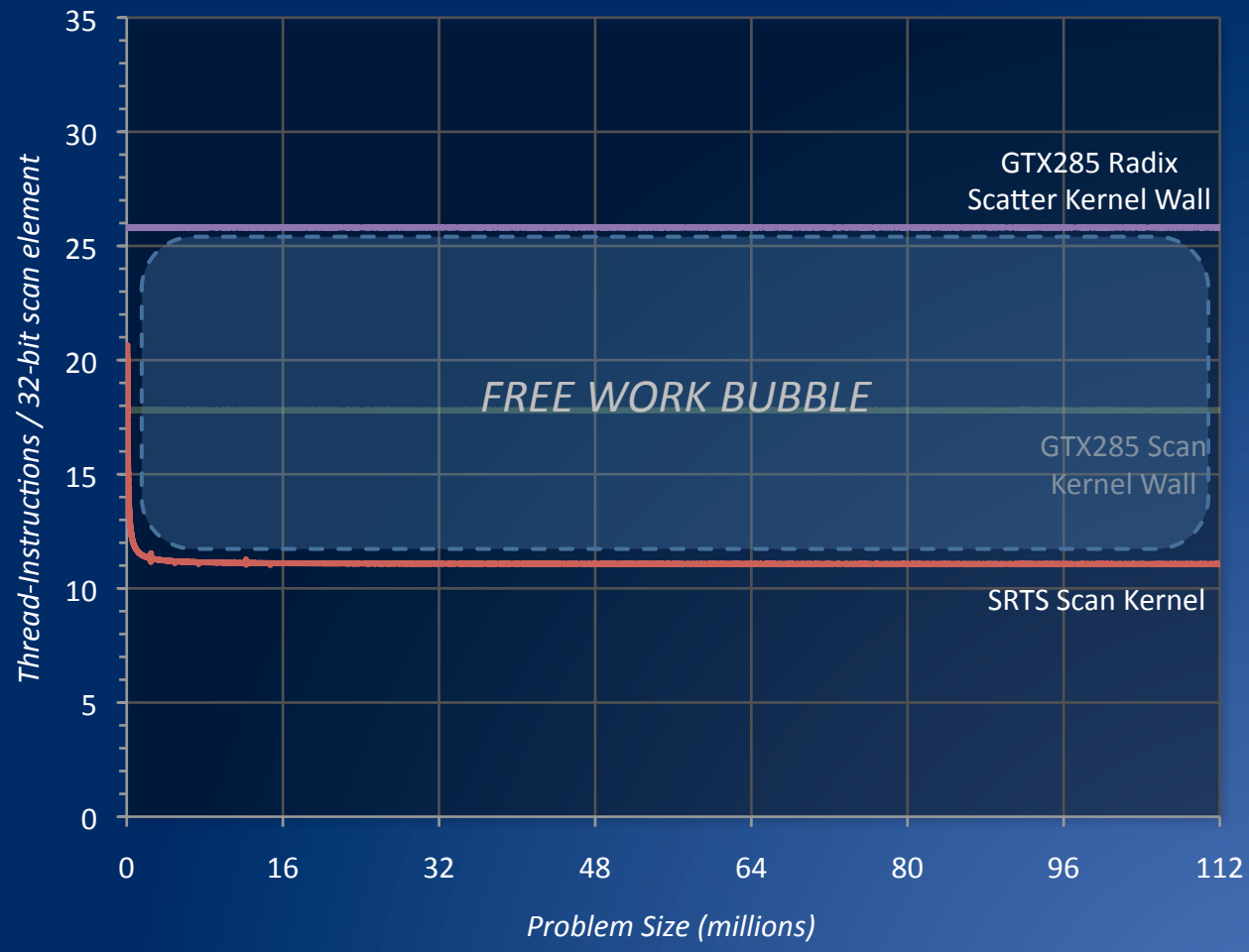
Know your kernel's memory wall

- Being below the wall gives you *flexibility…*
- .. for doing more local work:
  - Better granularity (e.g., increase redundant computation, ghost cells, radix bits, etc.)
  - Orthogonal kernel fusion



...for radix sorting

- Partially-coalesced writes (key scattering) increase write overhead by ~2x
- Bubble helps to accommodate:
  - Decoding key digits
  - Additional local scatter step in shared memory before globally scattering keys
  - Bigger granularity: four total concurrent scan operations (radix 16)



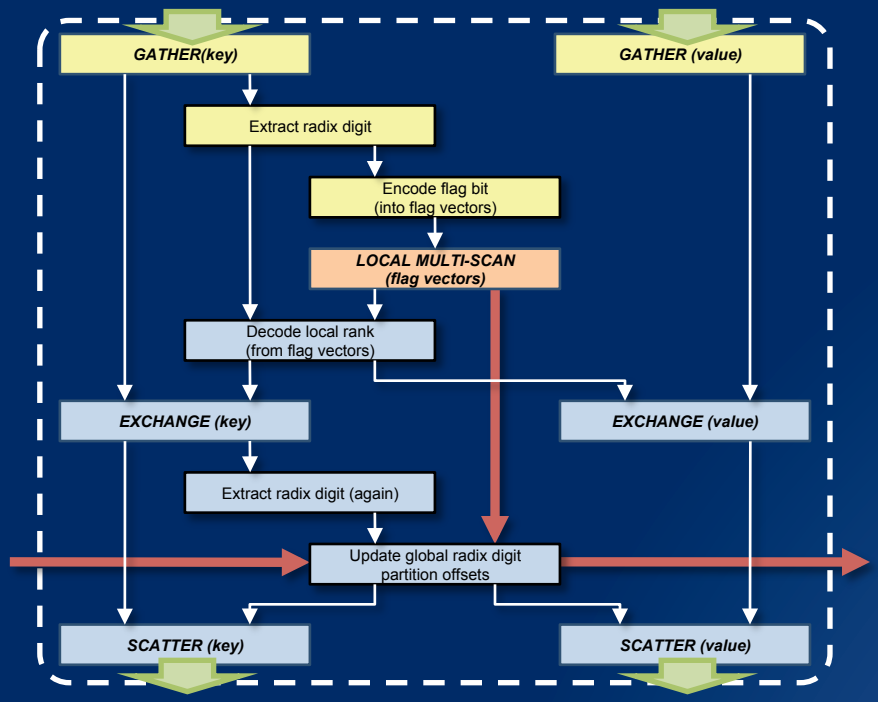
Programming Model Challenges

- Poor functional abstraction
  - A single host-side procedure call launches a kernel that performs orthogonal program steps

```
MyUberKernel<<<grid_size, num_threads>>>(d_device_storage);
```

- Barriers to code reuse
  - No existing public repositories of kernel “subroutines” for scavenging

Programming Model Challenges



Fused radix sorting kernel

- Digit extraction
- Local prefix scan
- Scatter accordingly

- Fusion from higher-order kernel interfaces is limited
  - Callbacks, iterators, visitors, functors, etc.
    - E.g., `ReduceKernel<<<grid_size, num_threads>>>(CountingIterator(100));`
  - Can't express complex subroutine compositions
    - E.g., fused kernel above can't be composed using a callback-based functor/visitor pattern

Algorithm Serialization

Too much expressed parallelism is bad

Decomposing problems as if you had limitless CTAs

It's one of CUDA's biggest accessibility strengths…

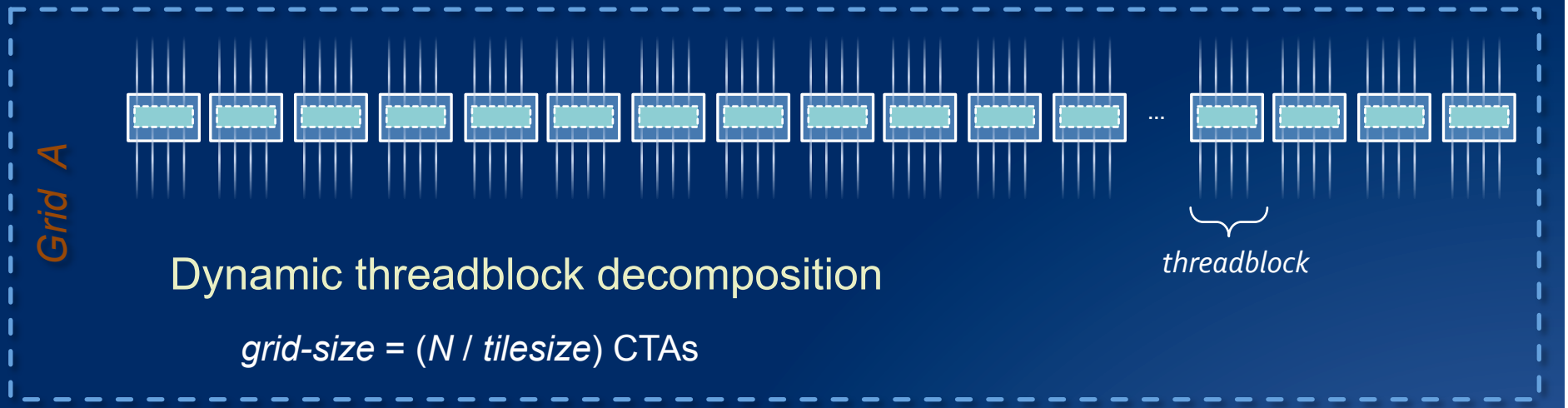
- Virtual processors abstract a diversity of hardware configurations

… and one if its biggest performance weaknesses

- Leads to a host of inefficiencies

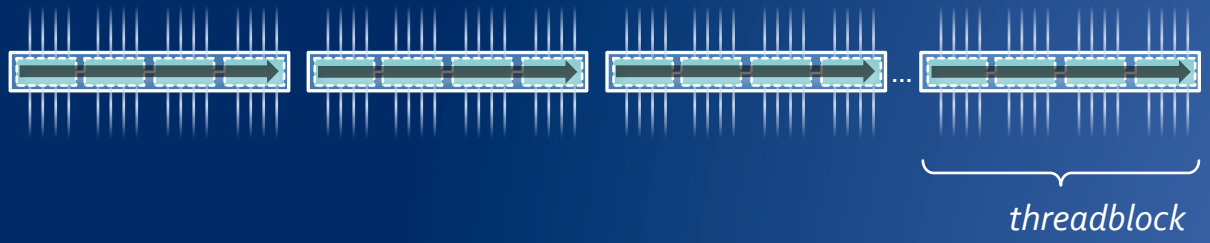
- *Instead: Design kernels for a fixed grid-size*
  - E.g., only several hundred CTAs

Example 1: Threadblock decomposition

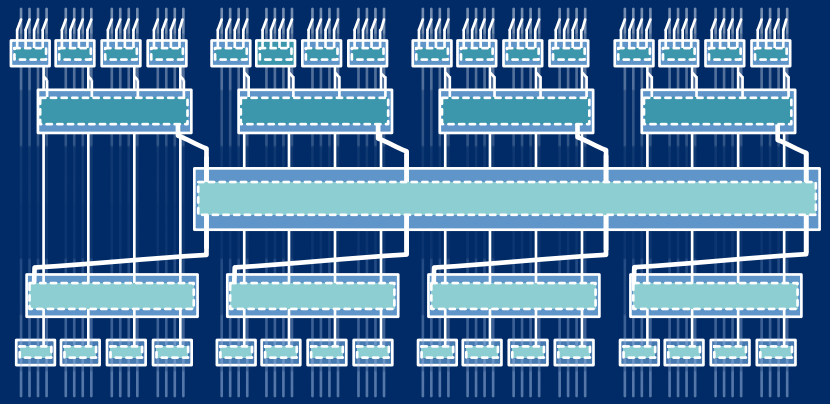


Benefits are many:

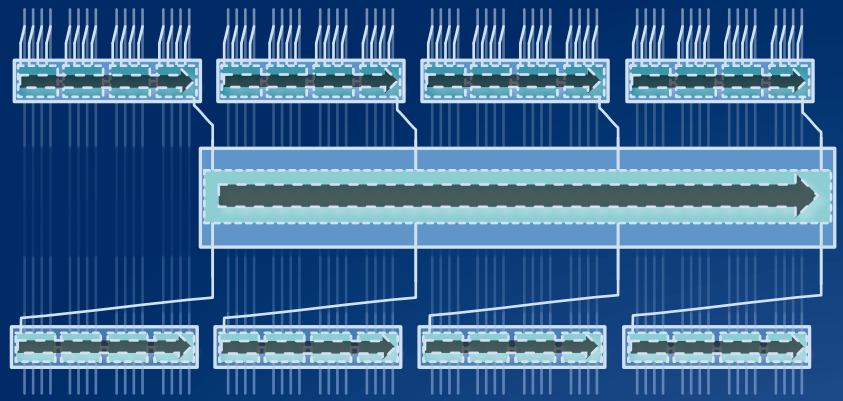
- Common work gets hoisted and reused, e.g.:
  - Thread-dependent predicates
  - Setup and initialization code (notably for smem)
  - Offset calculations (notably for smem)
- No problem size limitations
- Grid size becomes a tuning parameter
- Increased register pressure
  - Common values are hoisted and kept live



Example 2: Recursive threadblock decomposition



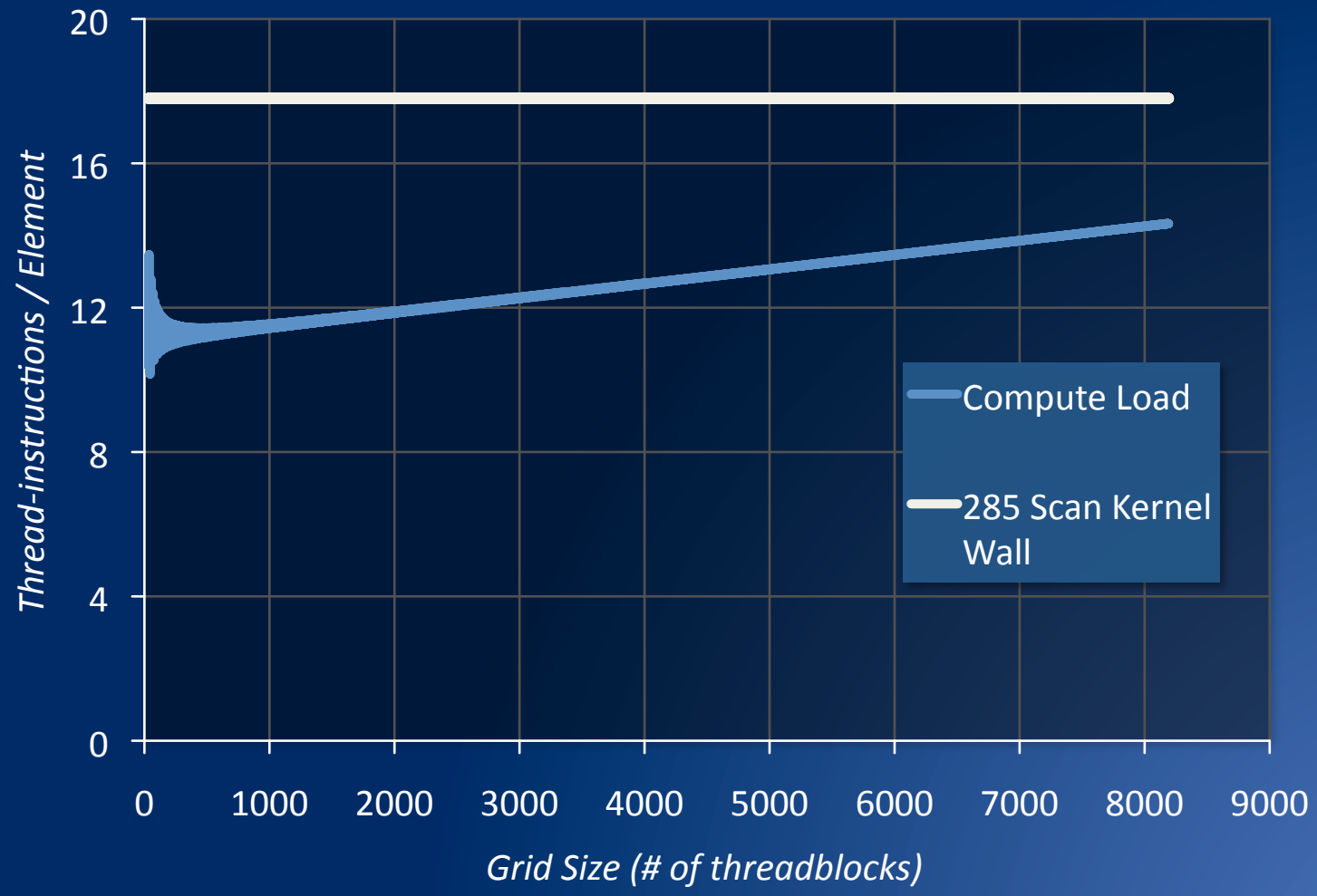
log $_{tilesize}$ (N) -level tree



Two-level tree

- 2-level curries results in registers (or smem) between tiles. Elides:
  - *O*(*N* / *tilesize*) gmem accesses
  - 2-4 instructions per access (offset calcs, load, store)
- 2-level only enacts a small, constant-sized inner tree
  - GPU is least efficient here: get it over with as quick as possible

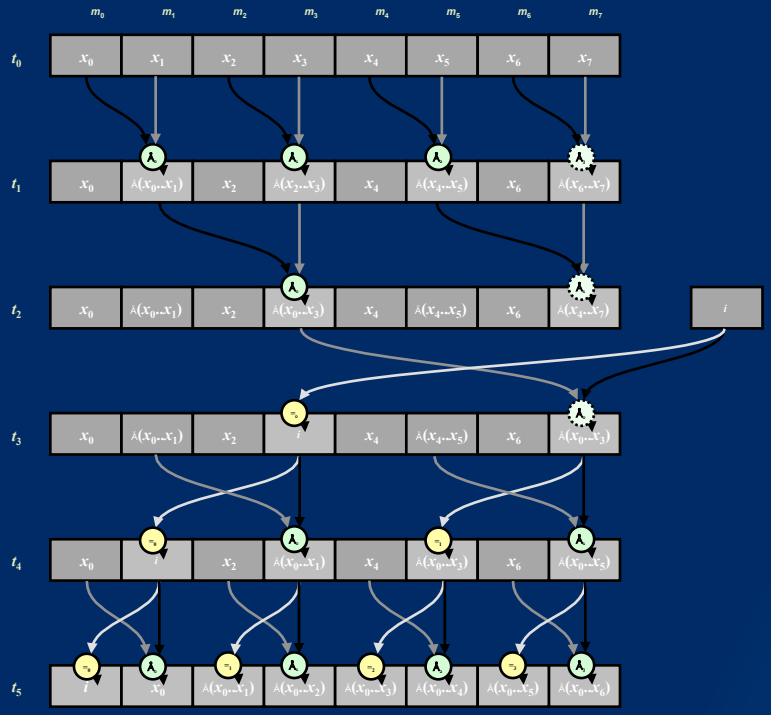
Overheads In Action
(prefix scan)

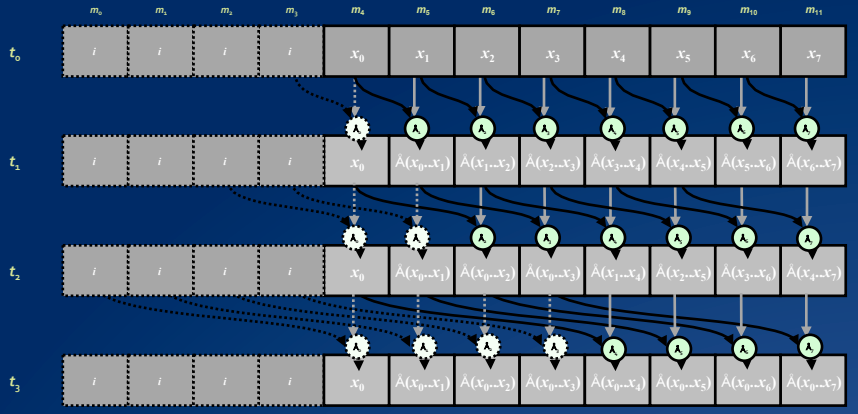


Warp-synchronous Programming

Too much expressed parallelism is bad (part 2)

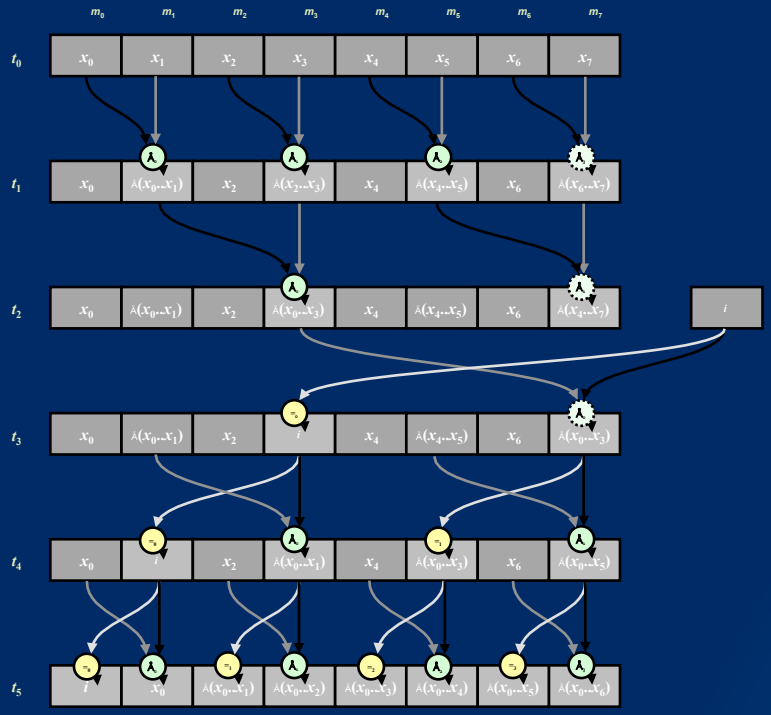
Want hybrid algorithms composed of different phases

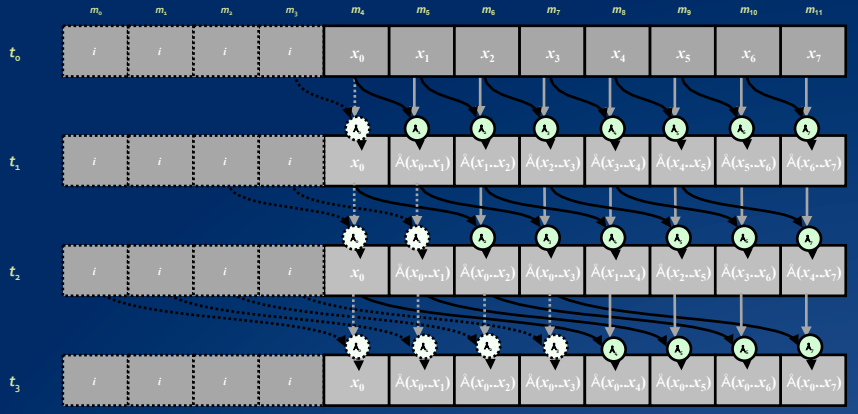




- E.g., local parallel prefix sum:
  - SIMD lanes wasted on $O(n)$-work Brent Kung (left), but less work when $n >$ warp size
  - Kogge-Stone (right) is $O(n \log n)$-work, but faster when $n \leq$ warp size

Want hybrid algorithms composed of different phases

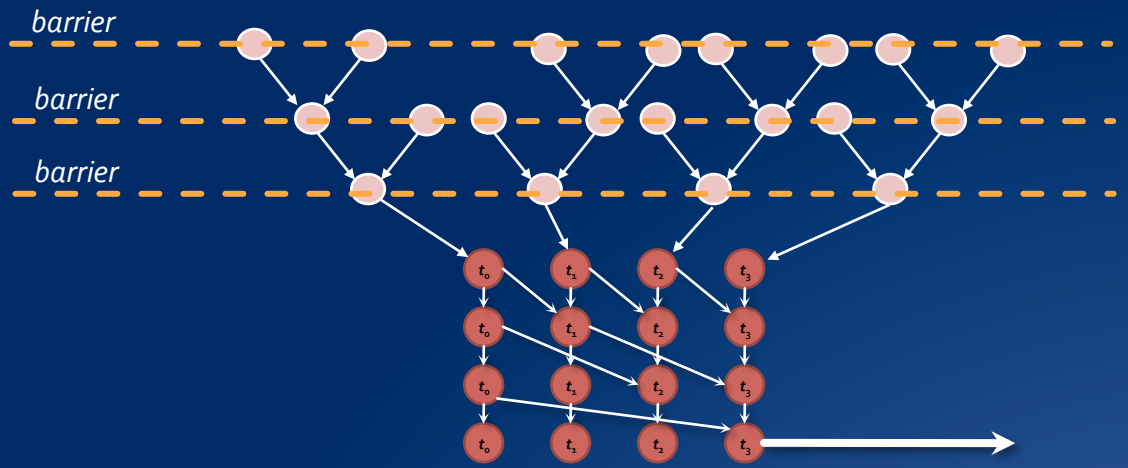




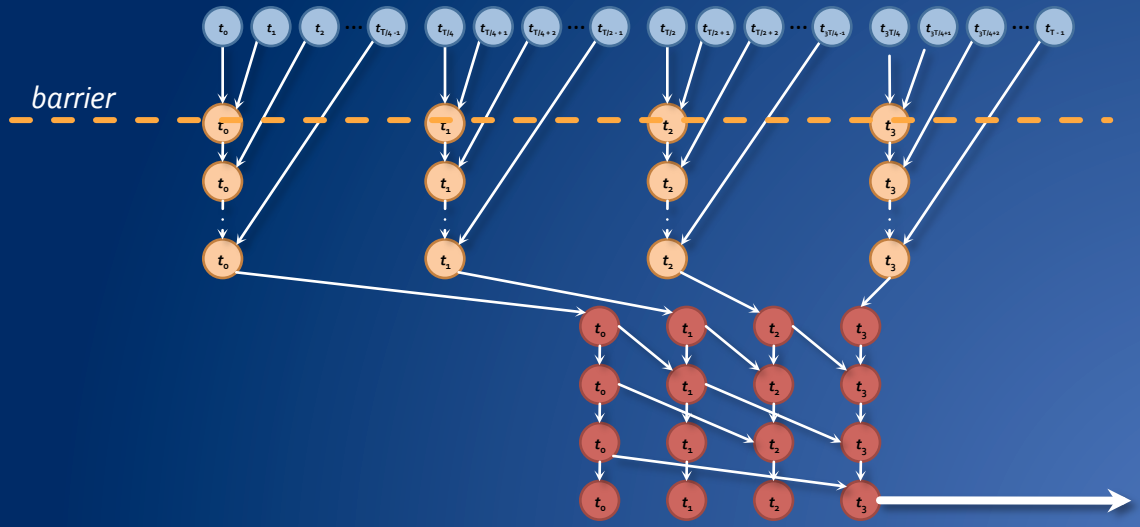
- E.g., local parallel prefix sum:
  - SIMD lanes wasted on $O(n)$-work Brent Kung (left), but less work when $n >$ warp size
  - Kogge-Stone (right) is $O(n \log n)$-work, but faster when $n \leq$ warp size

Warp-synchronous + Algorithm Serialization
(e.g., reduction)

Tree-based:

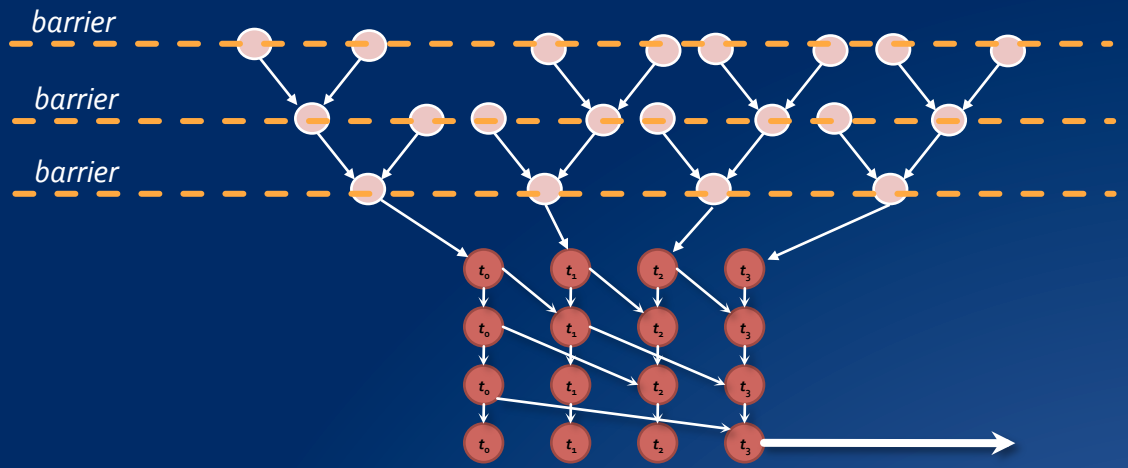


Vs. raking-based:

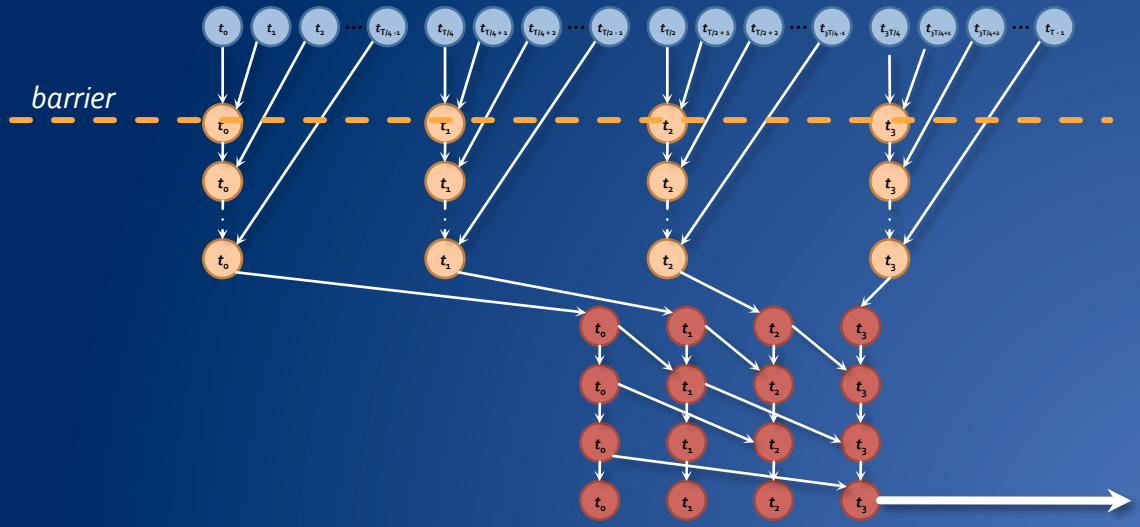


Warp-synchronous + Algorithm Serialization
(e.g., reduction)

Tree-based:



Vs. raking-based:



Diverse Warp Jobs

- Communication between threads is expensive
  - Barriers make O(n) code $O(n \log n)$
- One or two “worker warps”
  - The rest are “DMA engine” threads
  - Use threadblocks to cover pipeline latencies, e.g., Fermi SMs occupied by
    - 2 worker warps per CTA
    - 6-7 CTAs

Meta-programming

Improper granularity == performance cliff

Specialize target code for given devices

- Optimal granularity is different for:
  - Different SMs (varied local storage: registers/smem)
  - Different input types (e.g., sorting `chars` vs. `ulongs`)
- Author a single source implementation
  - # of steps for each algorithm phase is configuration-driven
  - Template expansion + Constant-propagation + Static loop unrolling + Preprocessor Macros
  - Compiler produces a target assembly that is well-tuned for the specifically targeted hardware *and* problem

E.g.: Scattering vector-2 pairs of keys to their binned destinations

```
#define SM20_PAIRS_PER_TILE()(4)        // 4 pairs on GF100
#define SM12_PAIRS_PER_TILE()(2)        // 2 pair on GT200
#define SM10_PAIRS_PER_TILE()(1)        // 1 pairs on G80
#define PAIRS_PER_TILE(version)((version >= 200) ? SM20_PAIRS_PER_TILE() : \
                                (version >= 120) ? SM12_PAIRS_PER_TILE() : \
                        SM10_PAIRS_PER_TILE())

...
template <typename KeyType, uint PAIRS>
    __device__ __forceinline__
void ScatterRankedKeys(
       KeyType *d_out_keys,
       typename VecType<KeyType, 2>::Type pairs[PAIRS],
       uint2 ranks[PAIRS])
{
    #pragma unroll
    for (uint PAIR = 0; PAIR < PAIRS; PAIR++) {
       d_out_keys[rank[PAIR].x] = pairs[PAIR].x;
       d_out_keys[rank[PAIR].y] = pairs[PAIR].y;
    }
}
...
ScatterRankedKeys<float, PAIRS_PER_TILE(__CUDA_ARCH__)> (d_out_keys, pairs, ranks);
```

Programming Model Challenges Pt. II

- Templates have logistical problems
  - Compiled libraries suffer from code bloat
    - CUDPP primitives library is 100s of MBs, yet still doesn't support all built-in numeric types.
    - Specializing for device configurations makes it even worse
  - The alternative is to ship source for #include'ing
    - Have to be willing to share source
  - Need a way to fit meta-programming in at the JIT / bytecode level to help avoid expansion / mismatch-by-omission
- Serializing algorithms is more than just "blocking"
  - Can leverage fundamentally different algorithms for different phases
    - How to teach the compiler do to this?

Summary

- Cooperative allocation crucial for dynamic parallelism
- Performance Strategies
  - Resource-allocation as runtime
  - Kernel fusion
  - Algorithm serialization
  - Warp-synchronous programming
  - Flexible granularity via meta-programming
- Challenges for the Programming Model
  - Poor functional abstraction
  - Little code-reuse
  - How to ship/deploy flexible code (avoid code bloat)

Questions?

{dgm4d, grimshaw} @ virginia.edu