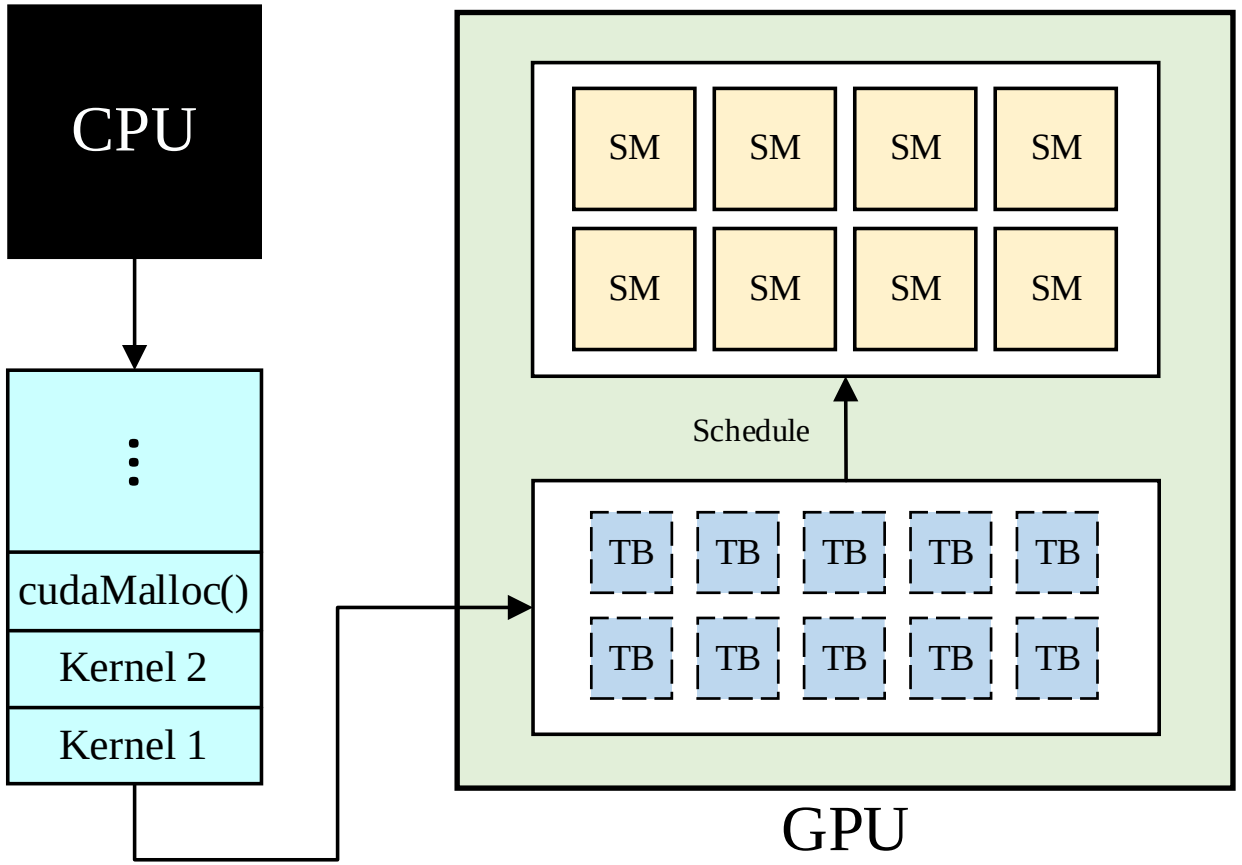
Improving Data-dependent Applications in GPUs

AmirAli Abdolrashidi

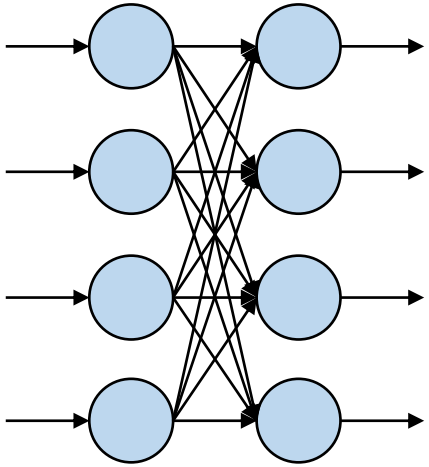
Introduction

- GPUs are ideal for massively parallel computations.
- Single-instruction, multiple-data (SIMD) paradigm
- Thread block (TB) → Task

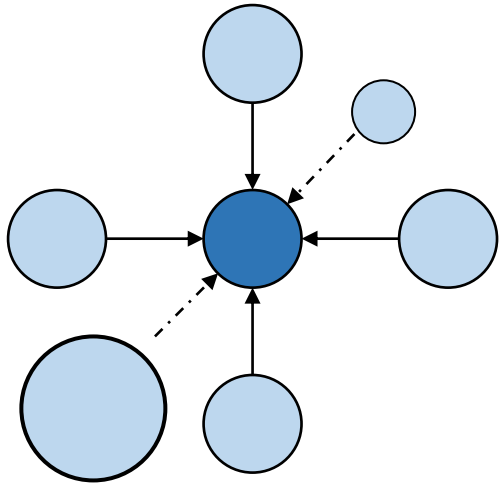


Introduction

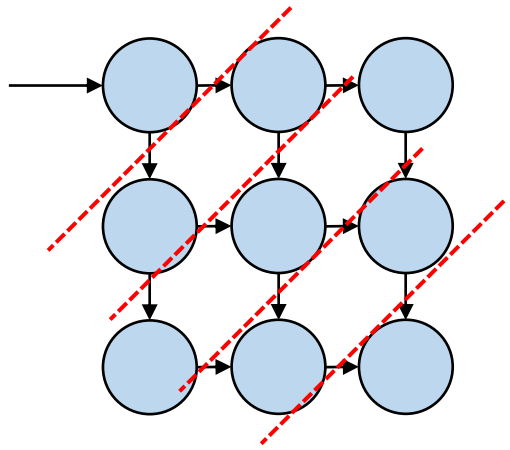
- Workloads are getting more complex
- Data dependency is common among kernels



Deep Learning



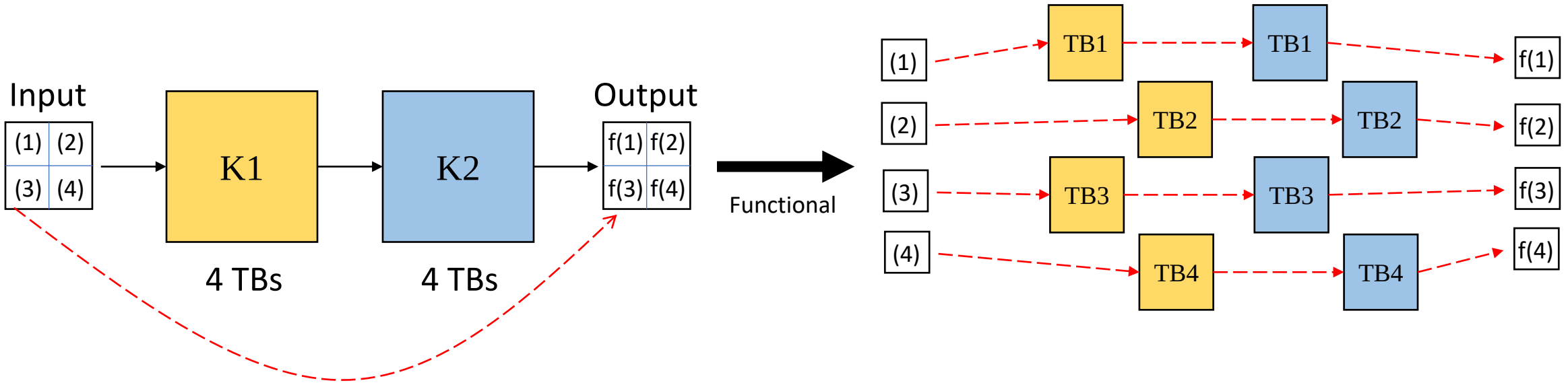
Stencil



Wavefront

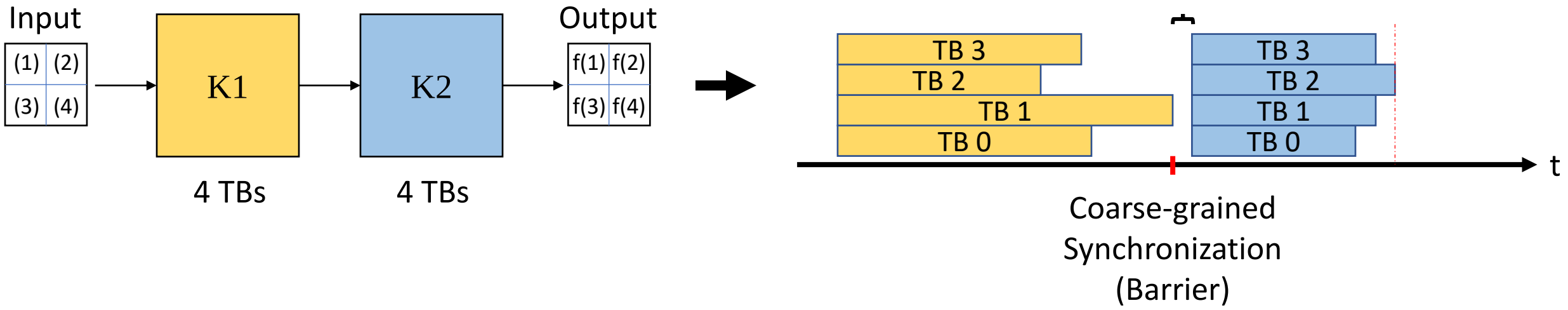
Introduction

- Despite support for massive parallelism, GPUs have limited support for data-dependent parallelism.



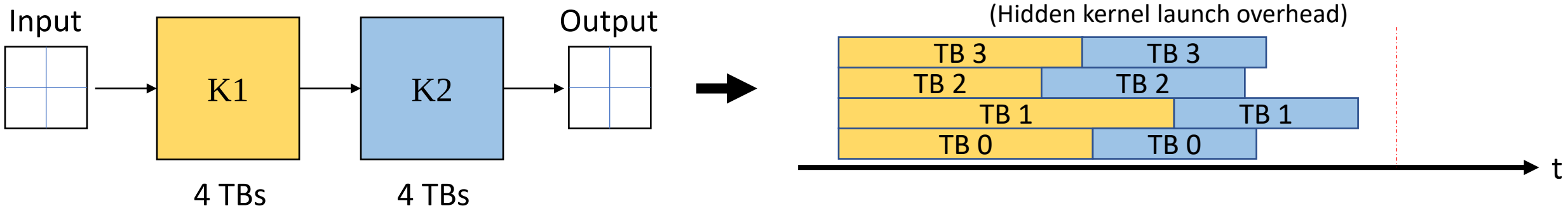
Introduction

- Despite support for massive parallelism, GPUs have limited support for data-dependent parallelism.



Introduction

- With fine-grained data dependency support
  - More utilization
  - Speedup

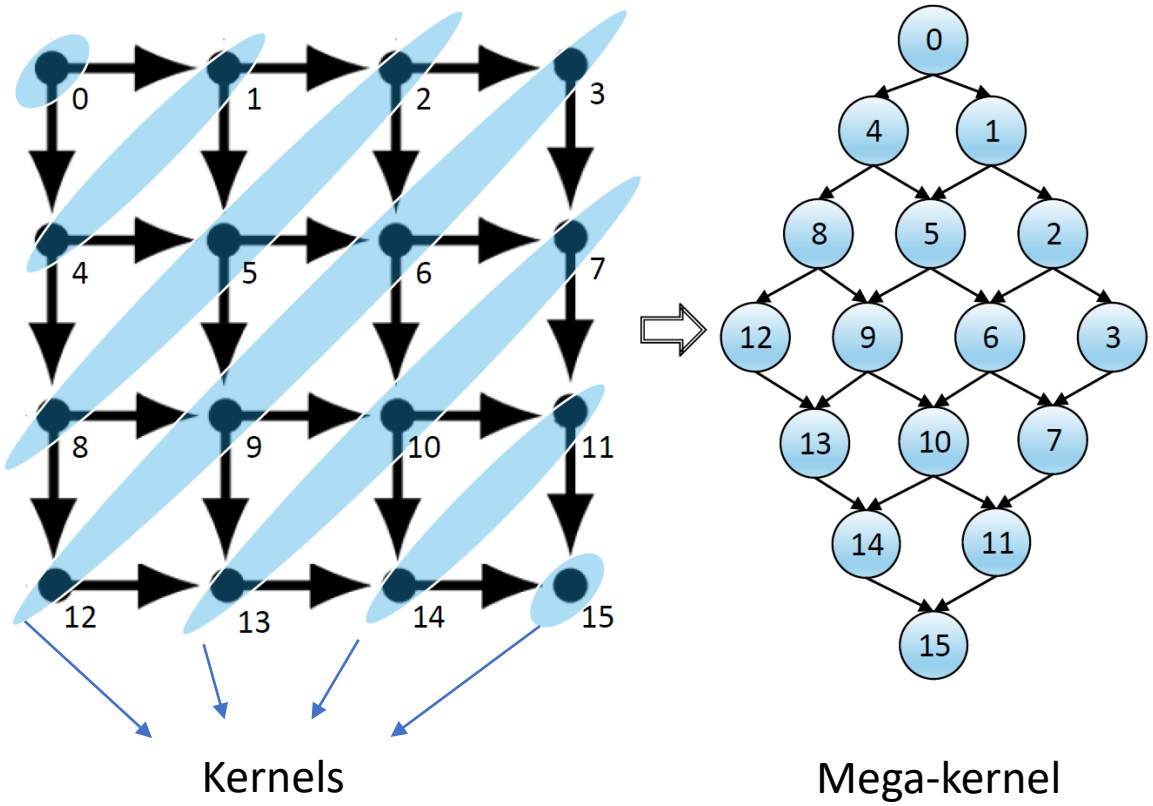


Motivation

- Need for generic finer-grain data dependency management
  - More performance
  - More efficient

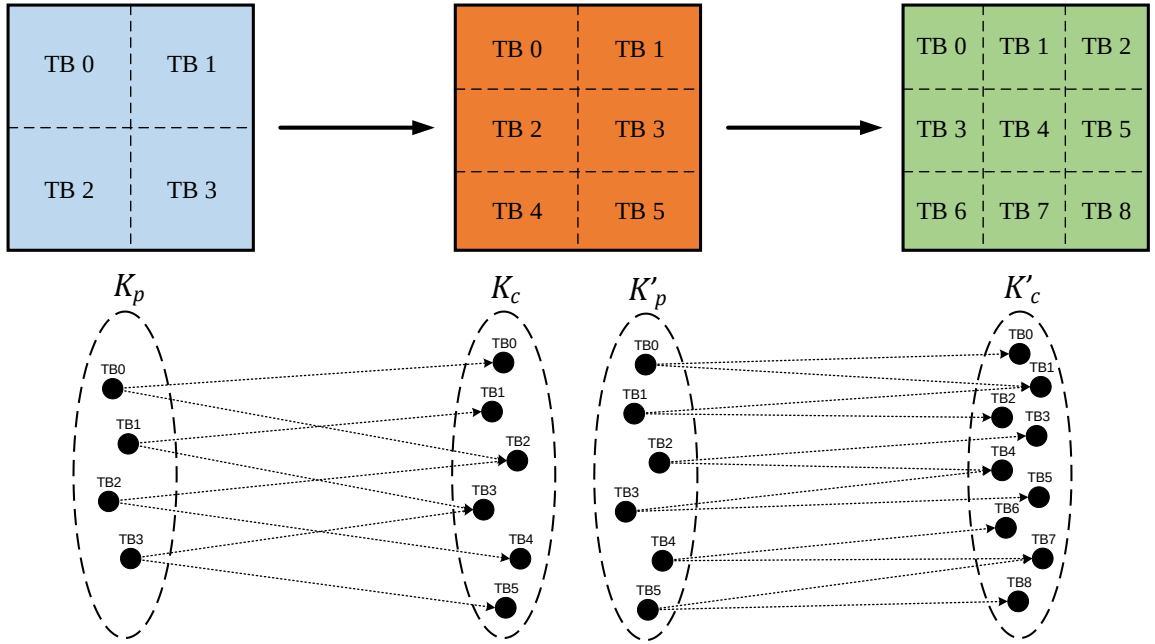
Our Goal

- Towards a fine-grained generalized dependency support
- **Step 1**
  - A single *mega-kernel*
  - Task-based API
    - Dependency graph provided by user
  - Dependency-aware TB scheduler
    - Handles dependencies within kernel



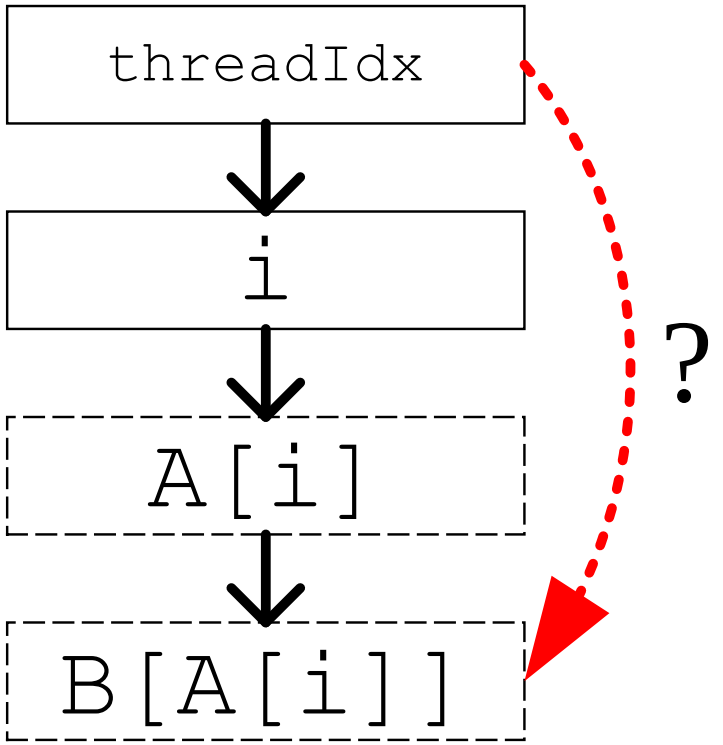
Our Goal

- User must alter code/provide graph
  - User must know the dependencies
- **Step 2**
  - TBs as tasks
  - Minimize user intervention in extracting dependencies
    - Static JIT analysis
  - Kernel pre-launching
    - Hides kernel-launch overhead
  - TB scheduling
    - Inter-kernel dependency resolution



Our Goal

- **Step 3** (Ongoing work)
  - We cannot detect non-static dependencies
    - Indirect memory access
    - Input-dependent branch
- Indirect estimation of data dependency without profiling
- Use pre-trained ML framework in the device
- Handle Misprediction



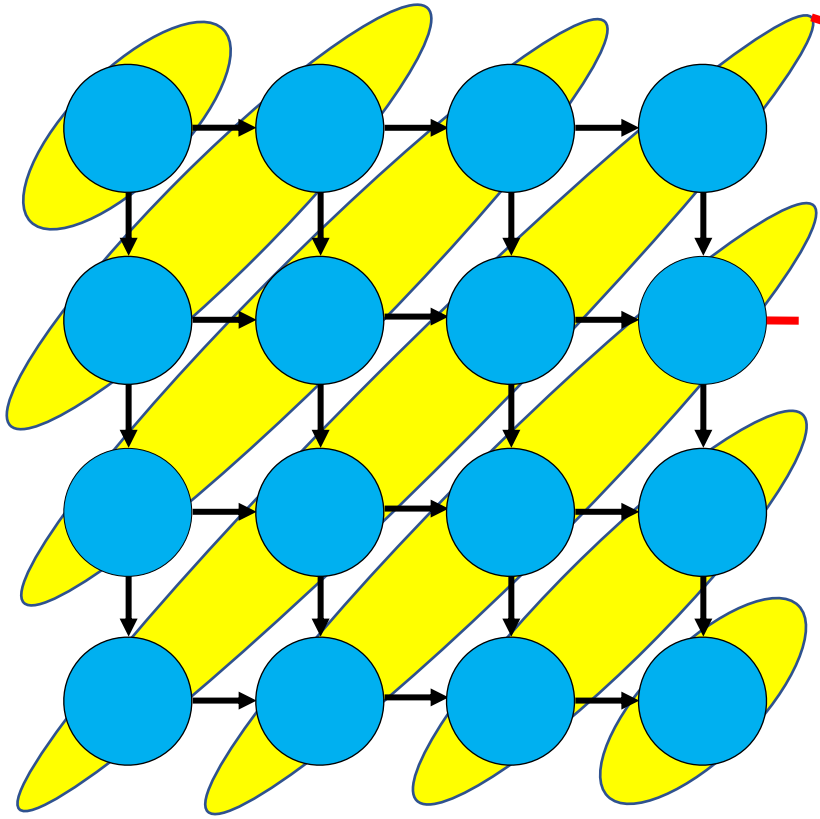
Introduction

- Main goals
  - Mitigate overheads, such as kernel launch, etc.
  - Reduce significant programming required by the user
  - More generalized hardware framework for dependency resolution

Step 1: Wireframe

Supporting Data-dependent Parallelism through Dependency Graph Execution in GPUs

Example: Wavefront Pattern



Barrier

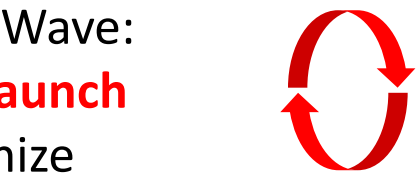
Thread block

…until the application ends

Example

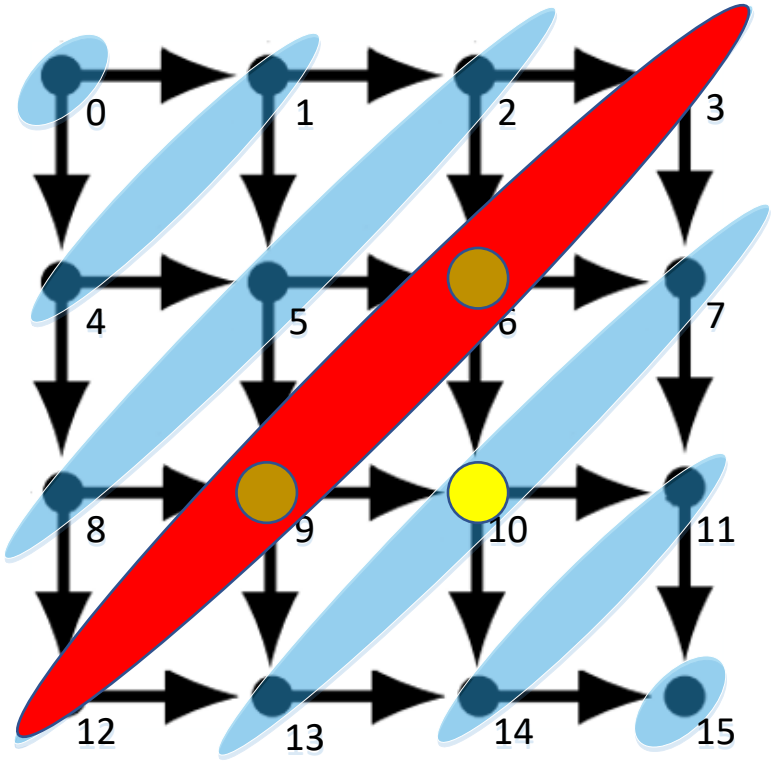
Global Barriers (Original)

for i = 1 to nWave:
-Kernel Launch
-Synchronize



Enormous host-side kernel-launch overhead!

Waiting on non-parent thread blocks



Example

CUDA Dynamic Parallelism (Nested)

RUN:

-Parent Kernel Launch

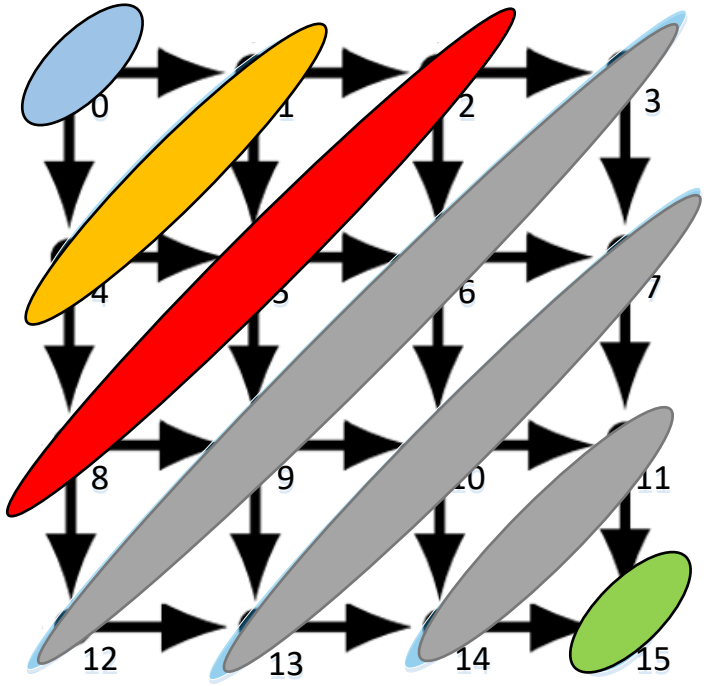
-Synchronize

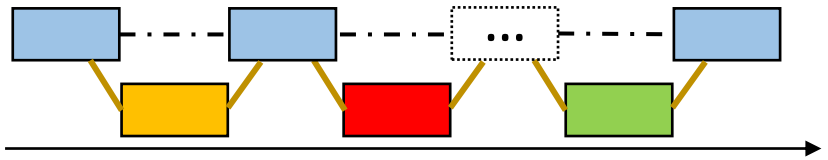
Parent Kernel:

for i = 1 to nWaves:

-Child Kernel Launch

-Synchronize





Kernel Execution Pattern

Example

CUDA Dynamic Parallelism (Nested)

RUN:

-Parent Kernel Launch

-Synchronize

Parent Kernel:

for i = 1 to nWaves:

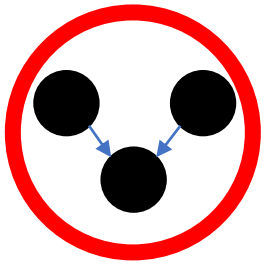
-Child Kernel Launch

-Synchronize

↑ No more host-side kernel launch

↓ Device-side kernel launch still has significant overhead

↓ NO multi-parent dependency support

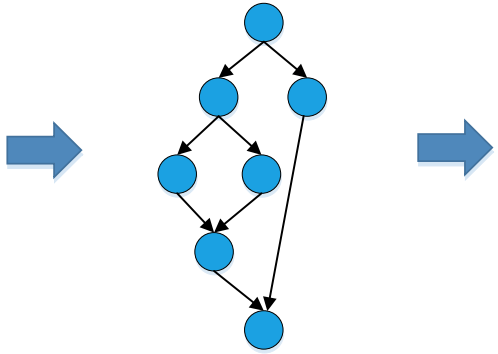


↓ Still NO general dependency support!

Wireframe Overview

Programming Model

```
#define parent1 dim3 (blockIdx.x-1,
blockIdx.y, blockIdx.z);
#define parent2 dim3 (blockIdx.x, blockIdx.y-
1, blockIdx.z);
void* DepLink() {
    if (blockIdx.x > 0)
WF::AddDependency(parent1);
    if (blockIdx.y > 0)
WF::AddDependency(parent2);
}
int main() {
    kernel<<<GridSize, BlockSize,
DepLink>>>(0, args);
}
__WF__ void kernel(args) {
    processWave();
}
```



Convert to CSR

Node Array

Edge Array

Host
(CPU)

Device
(GPU)

DATS Hardware
(Dependency Graph Buffer)

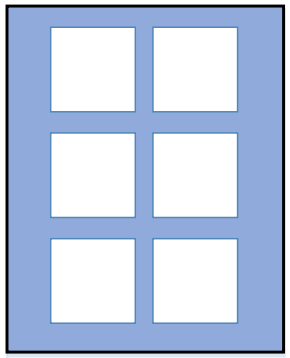
Local Node Array

Local Edge Array

Pending Update Buffer

Node Insertion Buffer

TB Scheduler



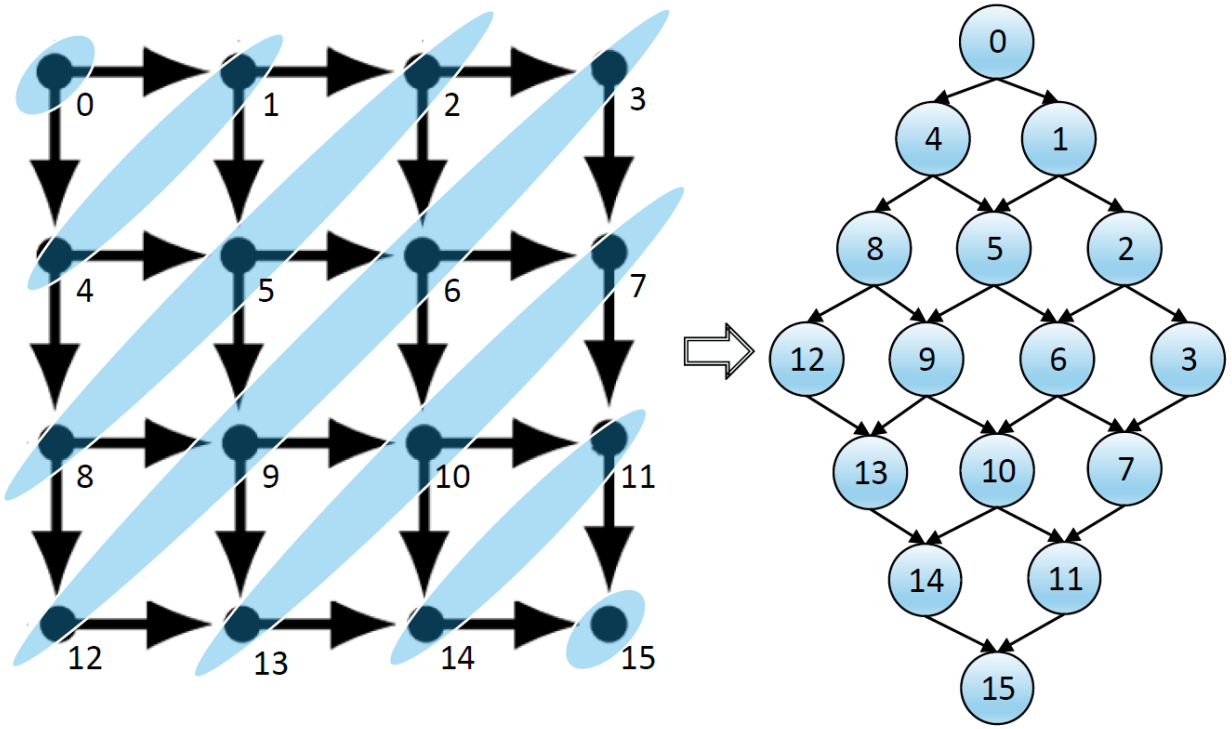
Global Memory

Global Node Array

Global Edge Array

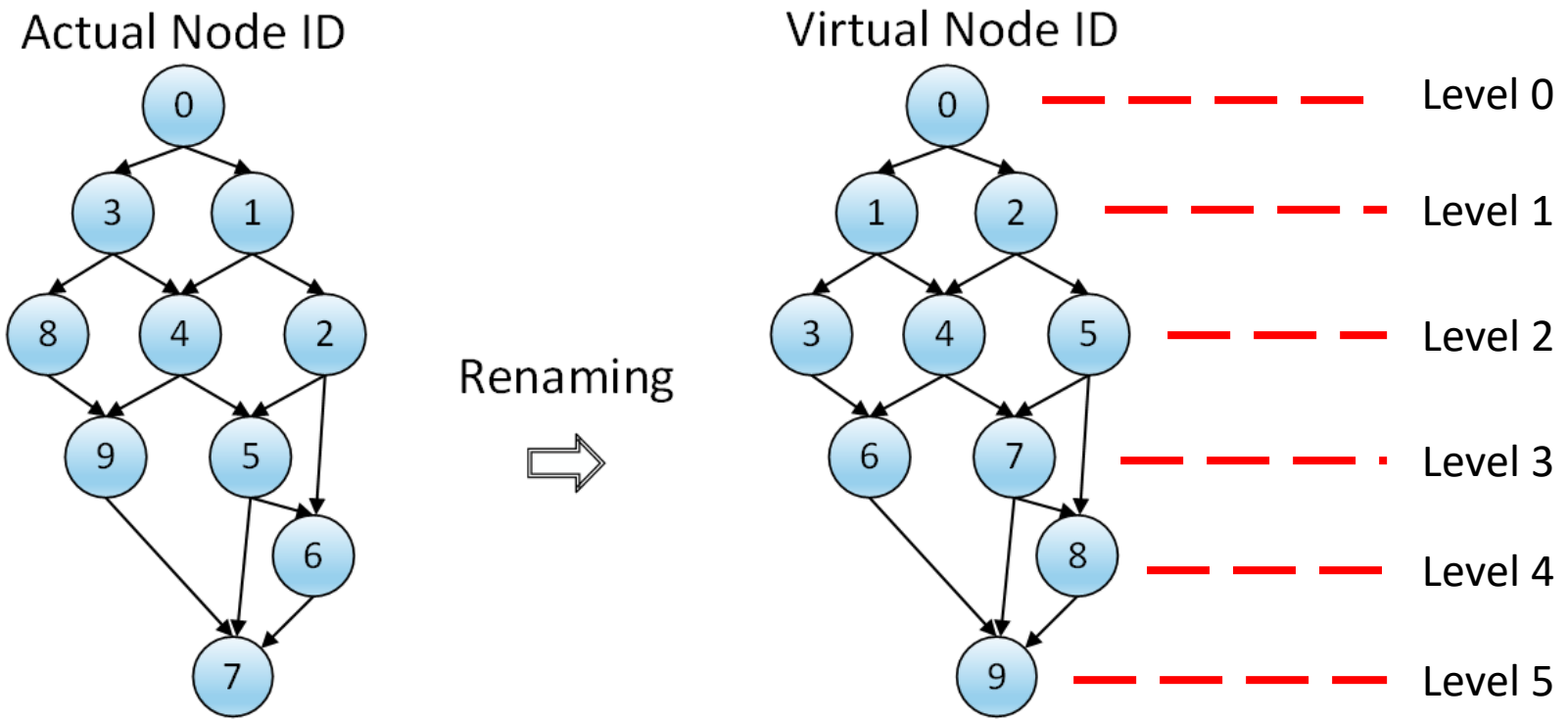
Dependency Graph

- Application → One mega-kernel!
  - User-provided dependencies
- Contains node information such as “Parent count” and “Level (from root)”
- Sent to the GPU’s global memory at kernel call



Node Renaming

- To minimize data level range in the buffers



Dependency-Aware TB Scheduler (DATS)

- Thread block scheduler
  - Issues the relevant thread block at the time for execution based on the dependency graph

- Dependency Graph Buffer (DGB)
  - Cache data from global memory
  - Challenge: Efficient caching and data utilization

Dependency-Aware TB Scheduler (DATS)

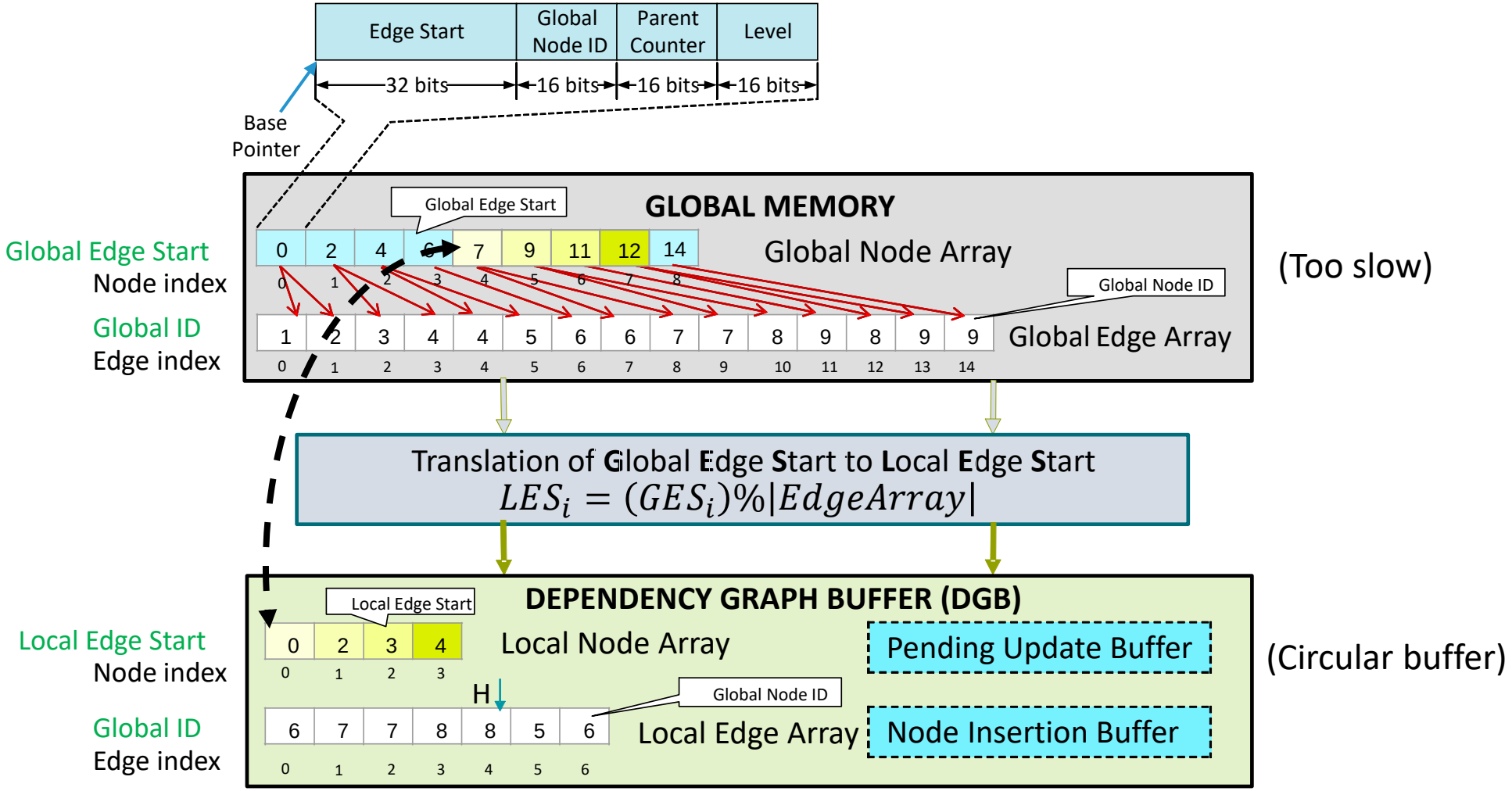
Data stored in compressed sparse (CSR) format

- **To reduce memory usage**

- **Thread blocks → Node Array**
- **Dependencies → Edge Array**

- $\boldsymbol{O(n \log n)}$ space complexity

DATS Overview

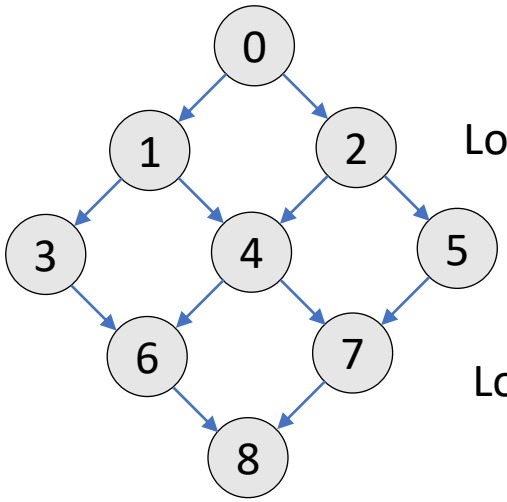


Node State Table

State	R	W	W	W
Parent Count	0	1	1	1
Level	0	1	1	2
Global Node ID	0	1	2	3

States:

- Wait
- Ready
- Processing
- Done



Tail

Head

Local Node Array

0	2	4	6

Local Edge Start

Local Edge Array

1	2	3	4	4	5	6

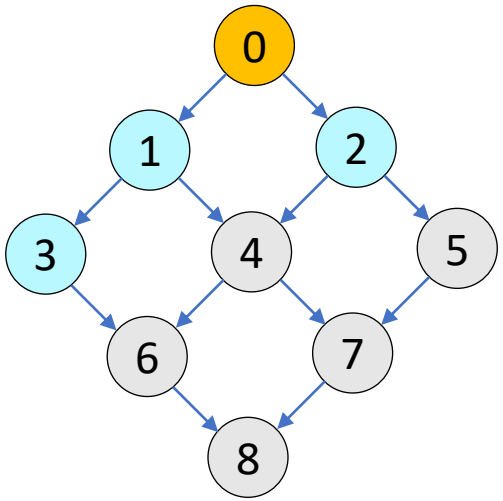
Global Node ID

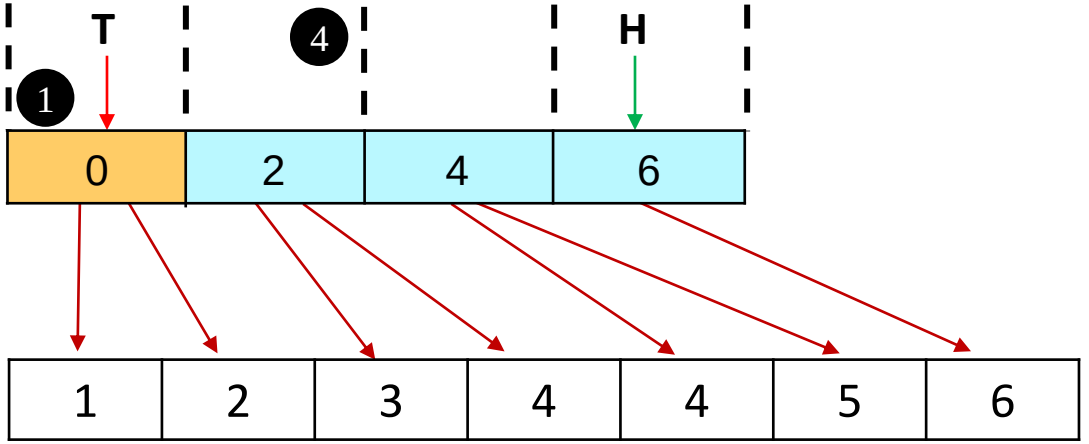
Example: Child Node Execution

State	1 D	3 R	3 R	W
Parent Count	0	2 0	2 0	1
Level	0	1	1	2
Global Node ID	0	1	2	3

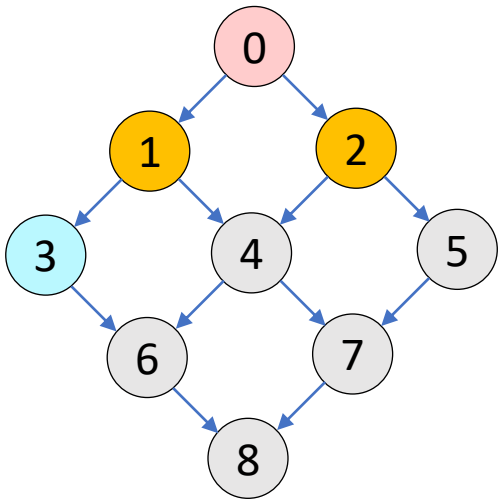
States:

- Wait
- Ready
- Processing
- Done





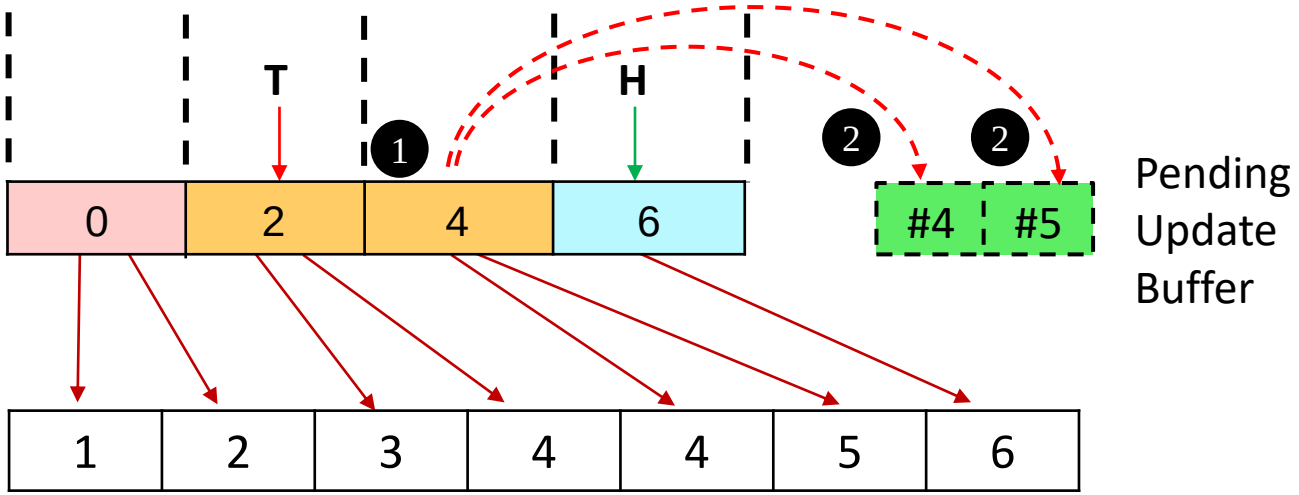
Example: Update Buffer Store



State	D	P	1 D	W
Parent Count	0	0	0	1
Level	0	1	1	2
Global Node ID	0	1	2	3

States:

- Wait
- Ready
- Processing
- Done



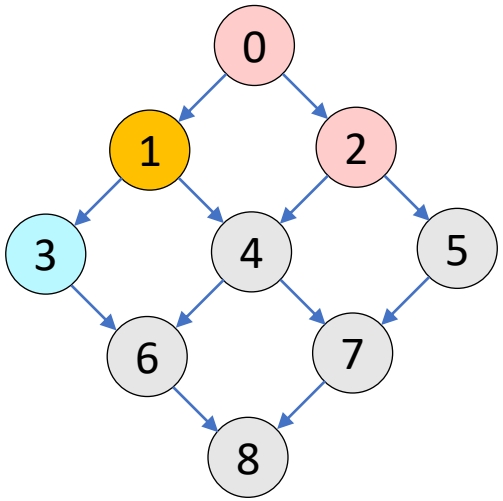
Example: Invalidation…

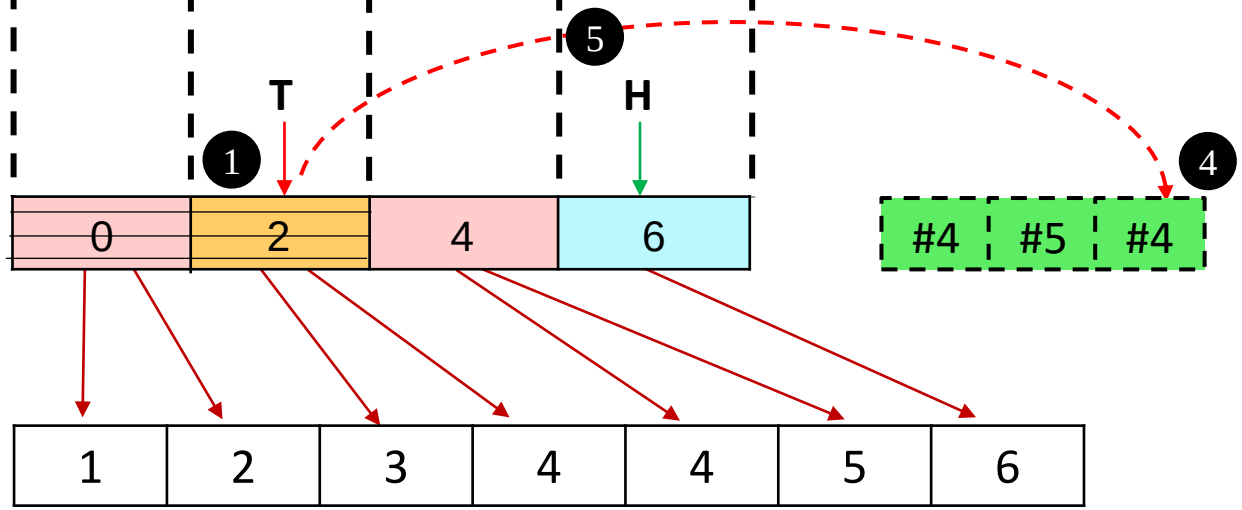
Enough spaces to load to DGB

State	D	D	D	3 R
Parent Count	0	0	0	2 0
Level	0	1	1	2
Global Node ID	0	1	2	3

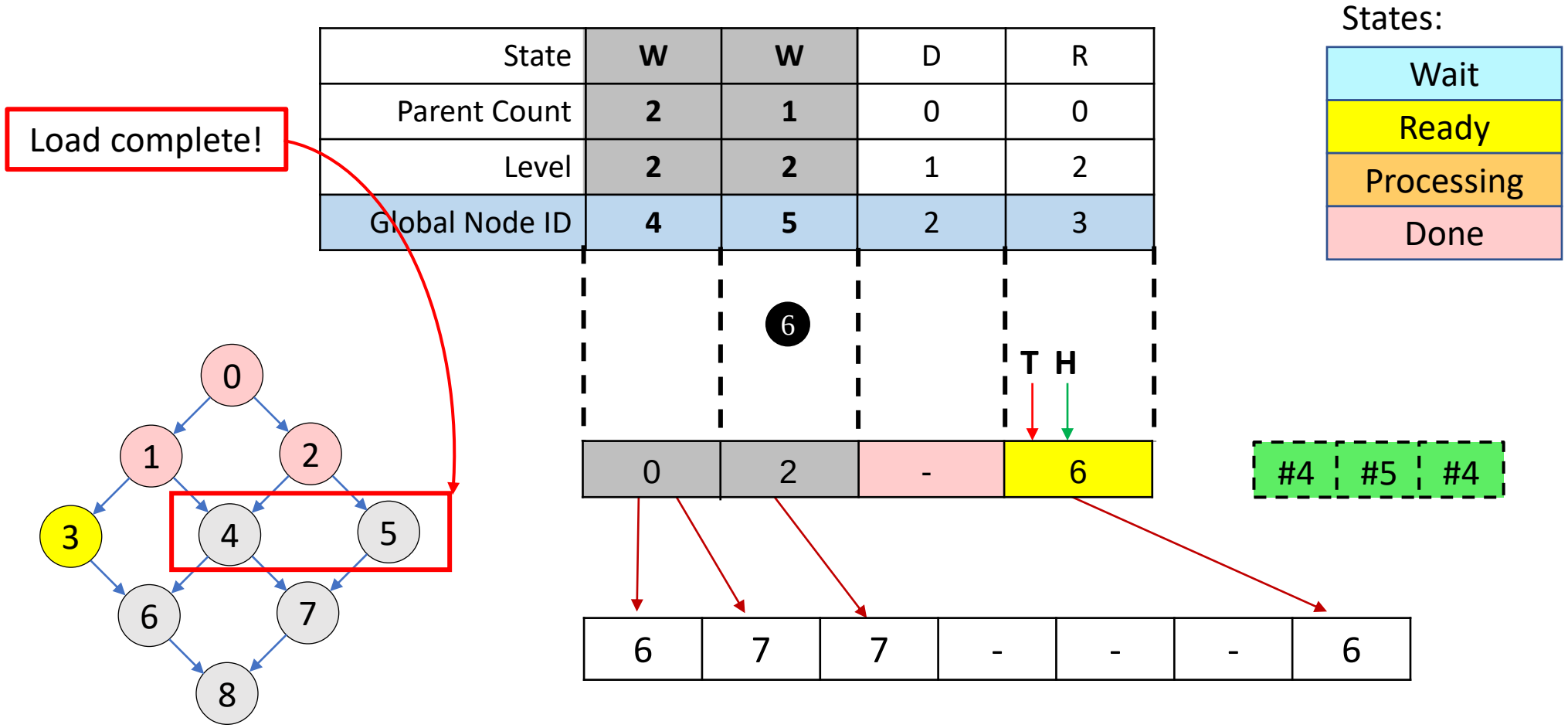
States:

- Wait
- Ready
- Processing
- Done





Example: …Reloading data

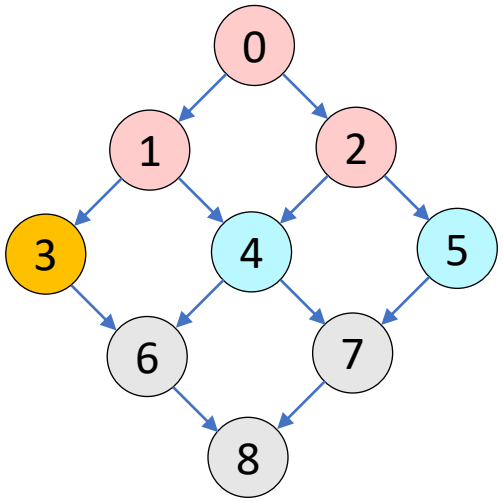


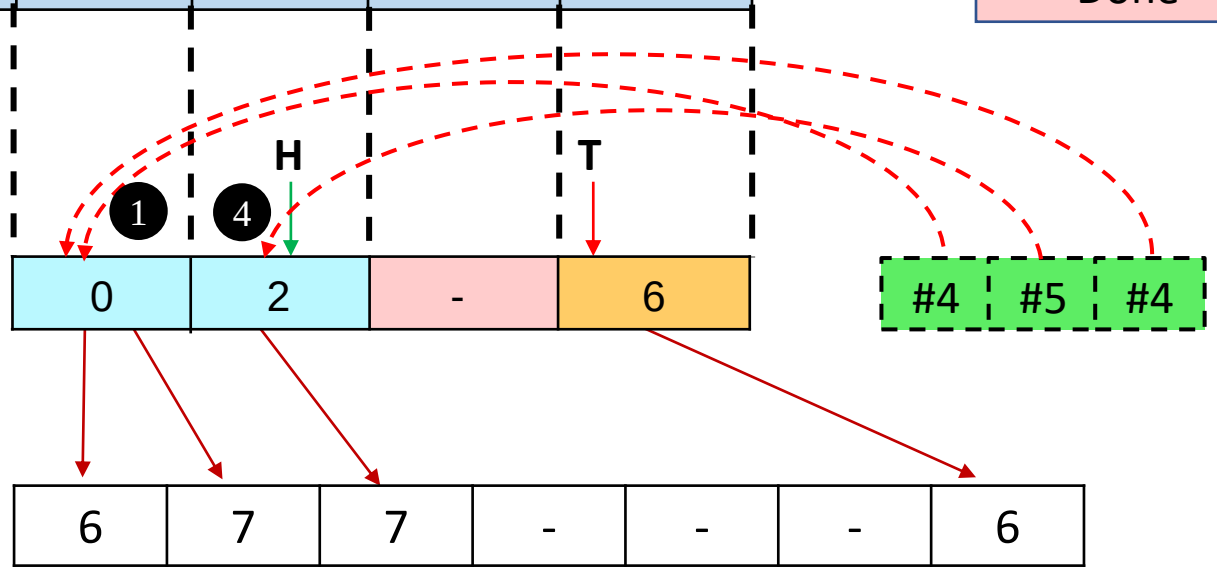
Example: Update Buffer Load

State	R 3	R	D	P
Parent Count	0 2	0	0	0
Level	2	2	1	2
Global Node ID	4	5	2	3

States:

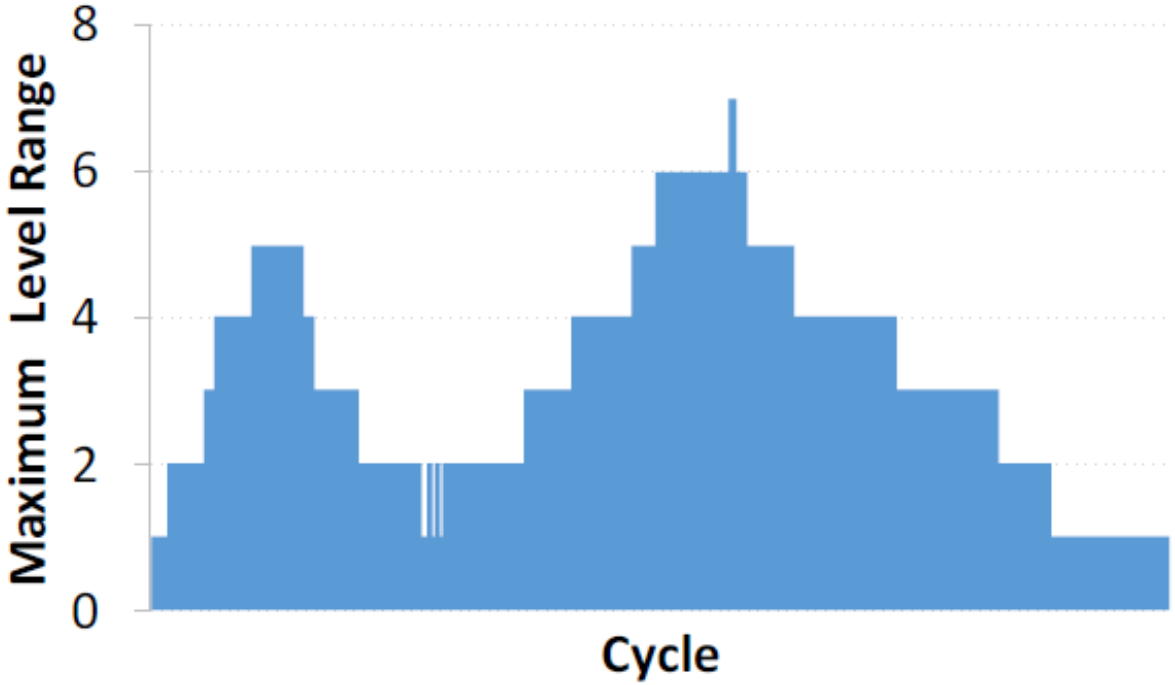
- Wait
- Ready
- Processing
- Done





Level Range

- Unbalanced execution may entail using the baseline TB scheduling policy (LRR).

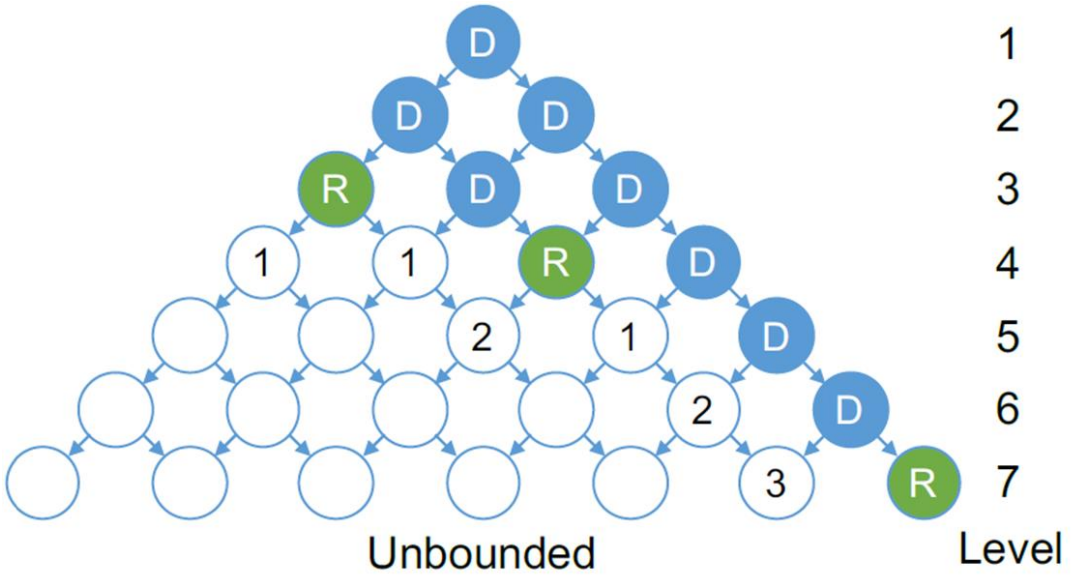


Sample benchmark (HEAT2D) w/ LRR scheduler

Level Range

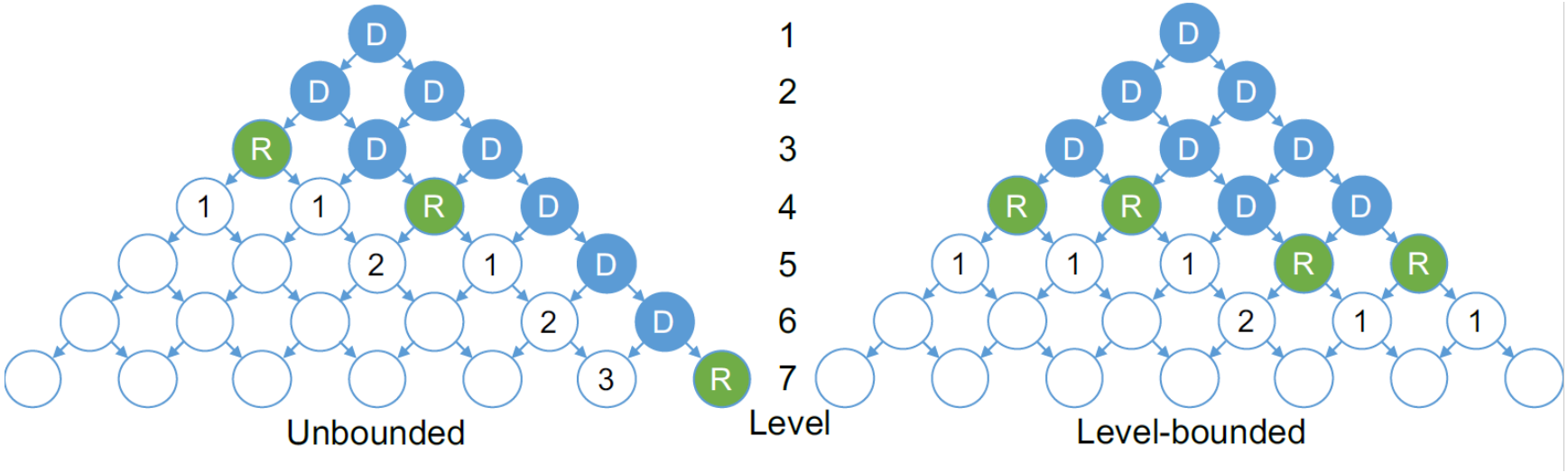
- Unbounded level range means:
  - Larger DGB is required
  - Limiting TB execution

Key challenge:
Efficient scheduling



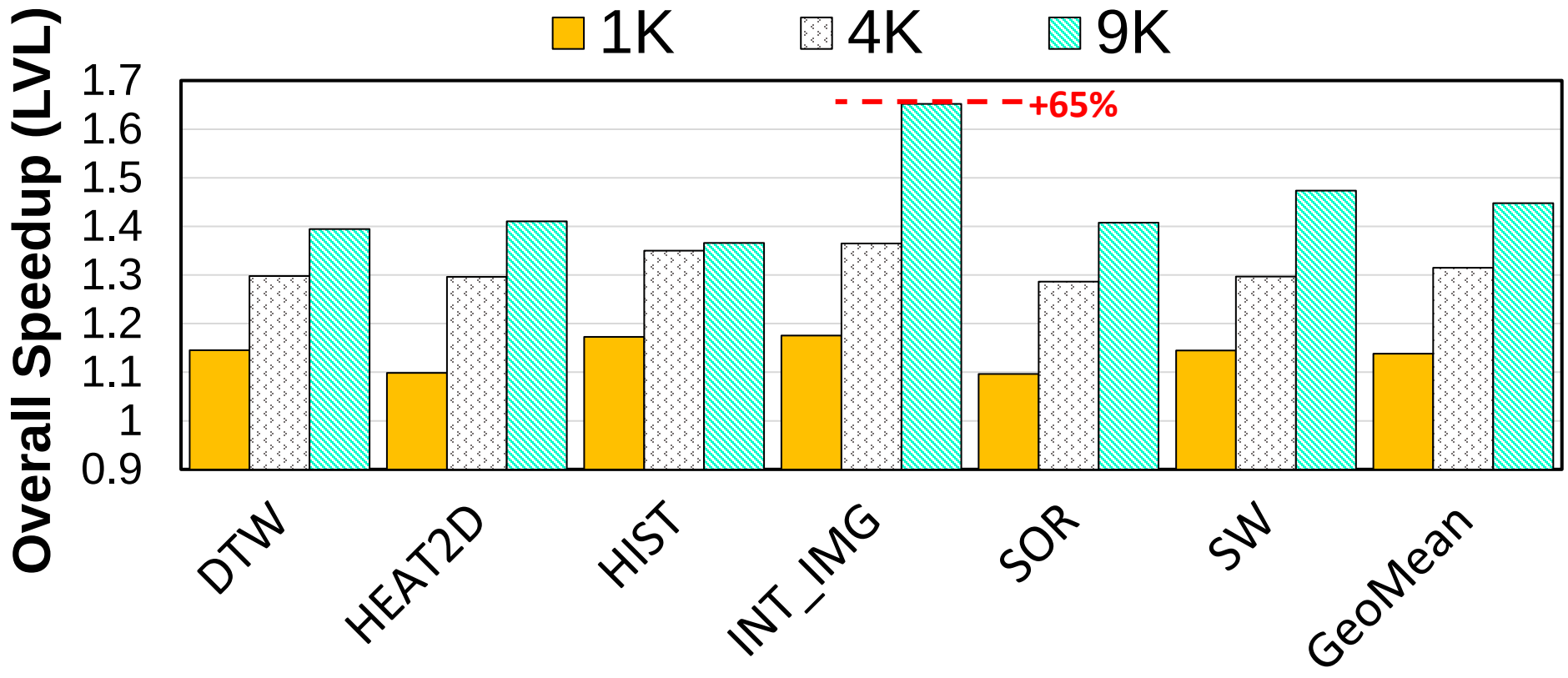
Level-bound Scheduling (LVL)

- Prioritizing lower-level thread blocks in the graph
- Minimizing the buffering operation
- Limiting the level range to avoid serialization



Performance

- Speedup across different number of nodes (same input size)



Works best for many smaller TBs

Step 2: BlockMaestro (BM)

Enabling Programmer-Transparent Task-based Execution in GPU Systems

Overview

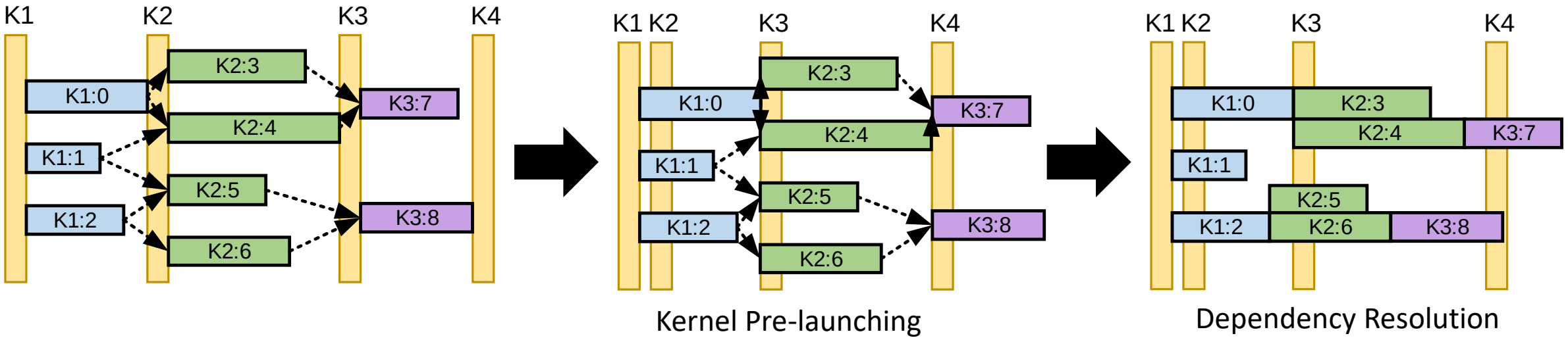
- Wireframe requires user intervention
  - Heavy burden to establish dependencies
    - Difficult for more complex/irregular applications
  - Program rewrite required

- Wireframe needs converting application's algorithm into tasks.

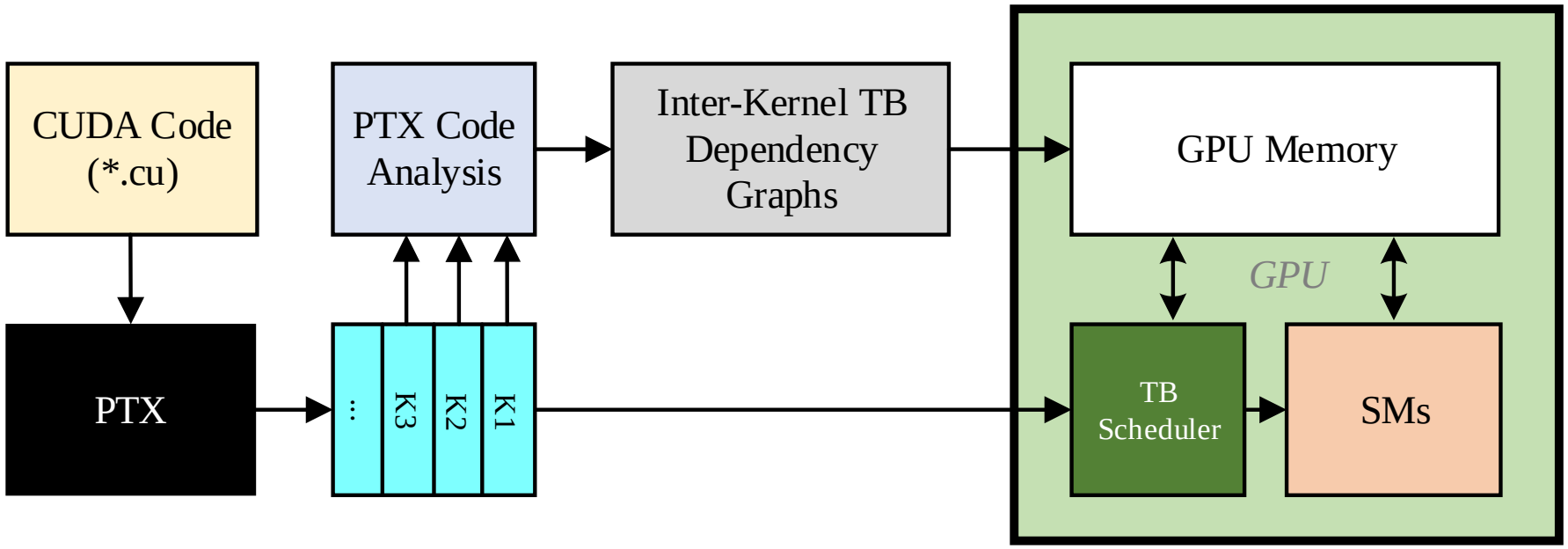
- Wireframe works on only one kernel

Motivation

- Finding dependencies among TBs **across kernels** with minimal burden
  - Different grid/block sizes
  - Minimal user intervention
- Increasing utilization by exploiting fine-grained dependencies



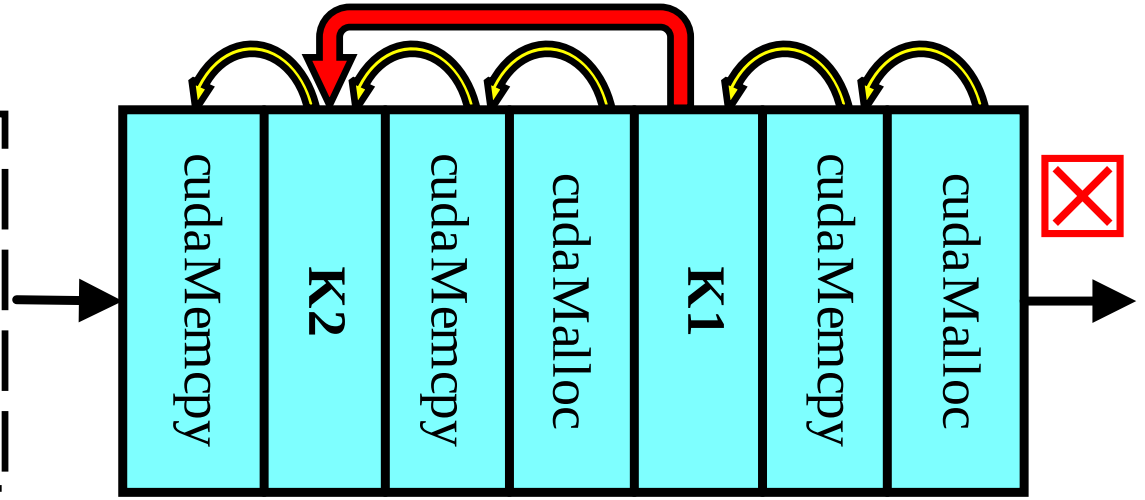
BlockMaestro Overview

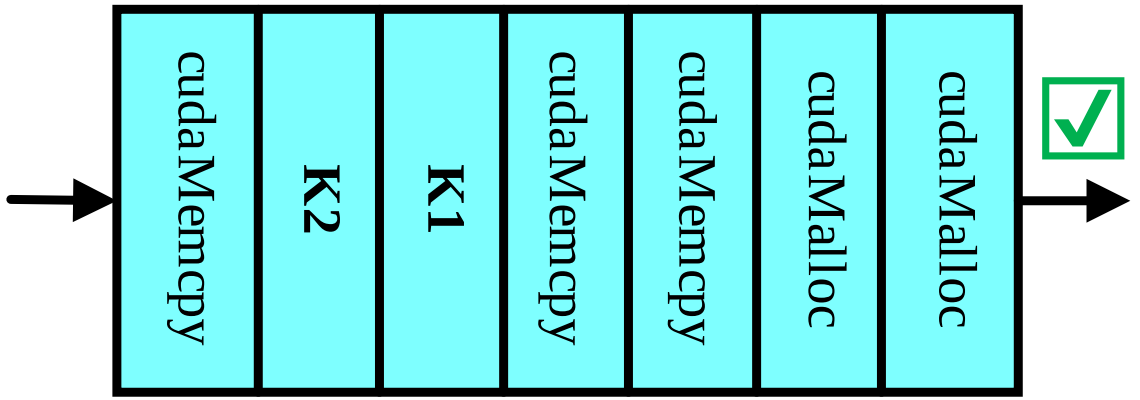


Command Queue

- Kernels can be launched in a non-blocking fashion.
- Host must wait for blocking instructions.
- Adjacent kernels can be launched together

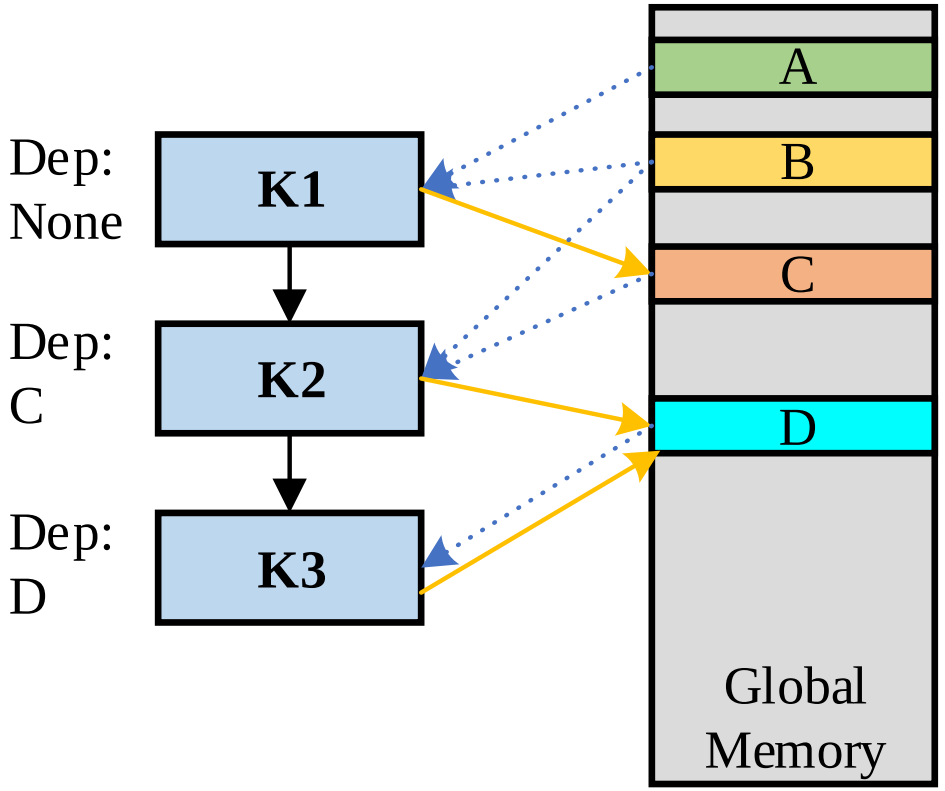
```
cudaMalloc(A)
cudaMemcpy(A, H2D)
K1<<<>>>(A)
cudaMalloc(B)
cudaMemcpy(B, H2D)
K2<<<>>>(A, B)
cudaMemcpy(D2H, B)
```





Inter-kernel Dependencies

- Dependencies can be detected through just-in-time (JIT) analysis
  - e.g., Read-after-write between the kernels
- Available info at kernel launch
  - Thread ID
  - Block ID
  - Base pointers
  - Kernel parameters



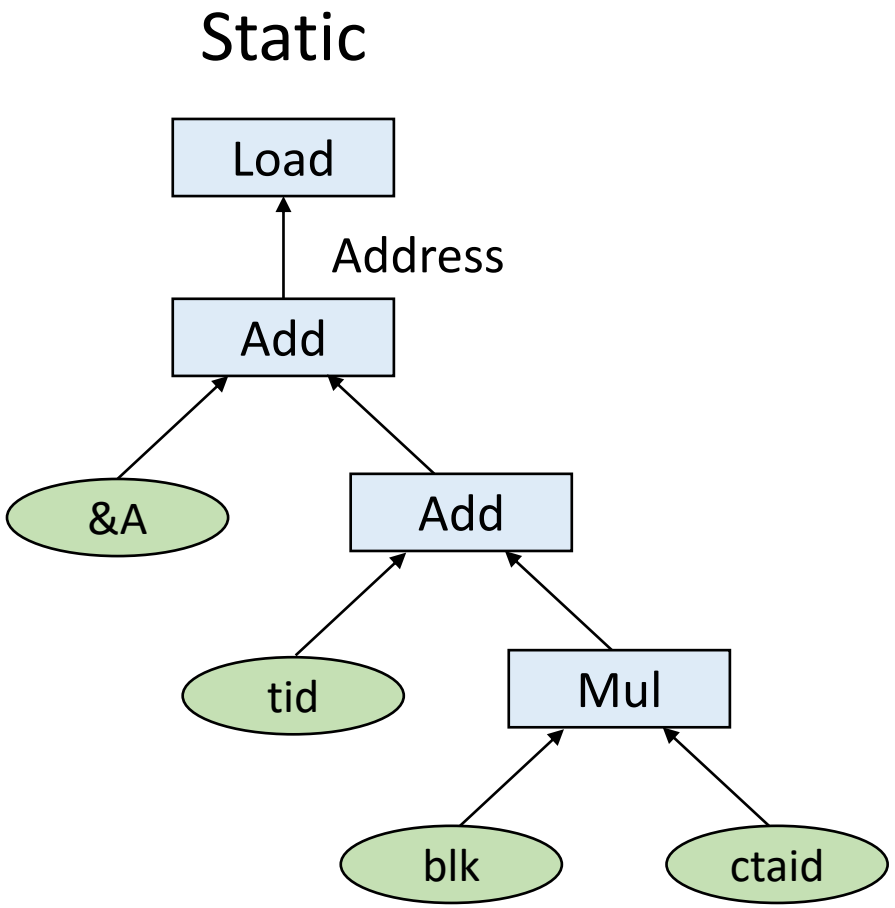
Static vs Non-static

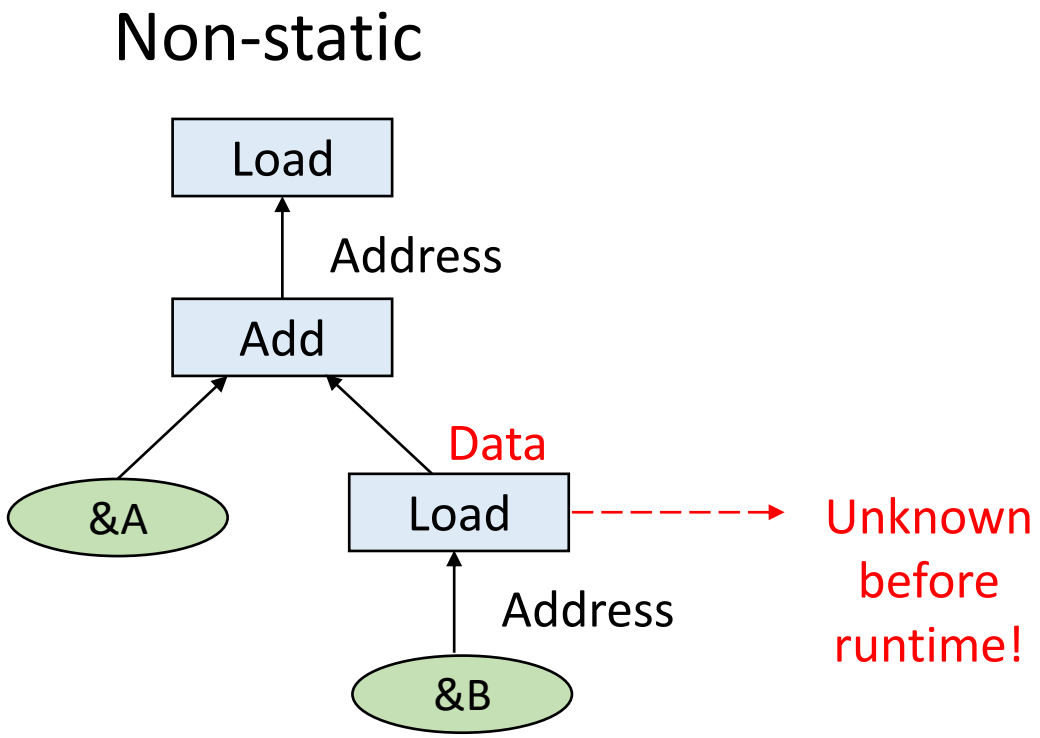
- Only memory locations determined up to kernel-launch time can be used for dependency resolution.

```
mov r1, ctaid.x
mul r1, r1, blk.x
add r1, r1, tid.x
add r1, r1, [A]
ld.global r2, [r1]
…
```

```
mov r1, [B]
ld.global r2, [r1]
add r2, r2, [A]
ld.global r3, [r2]
…
```

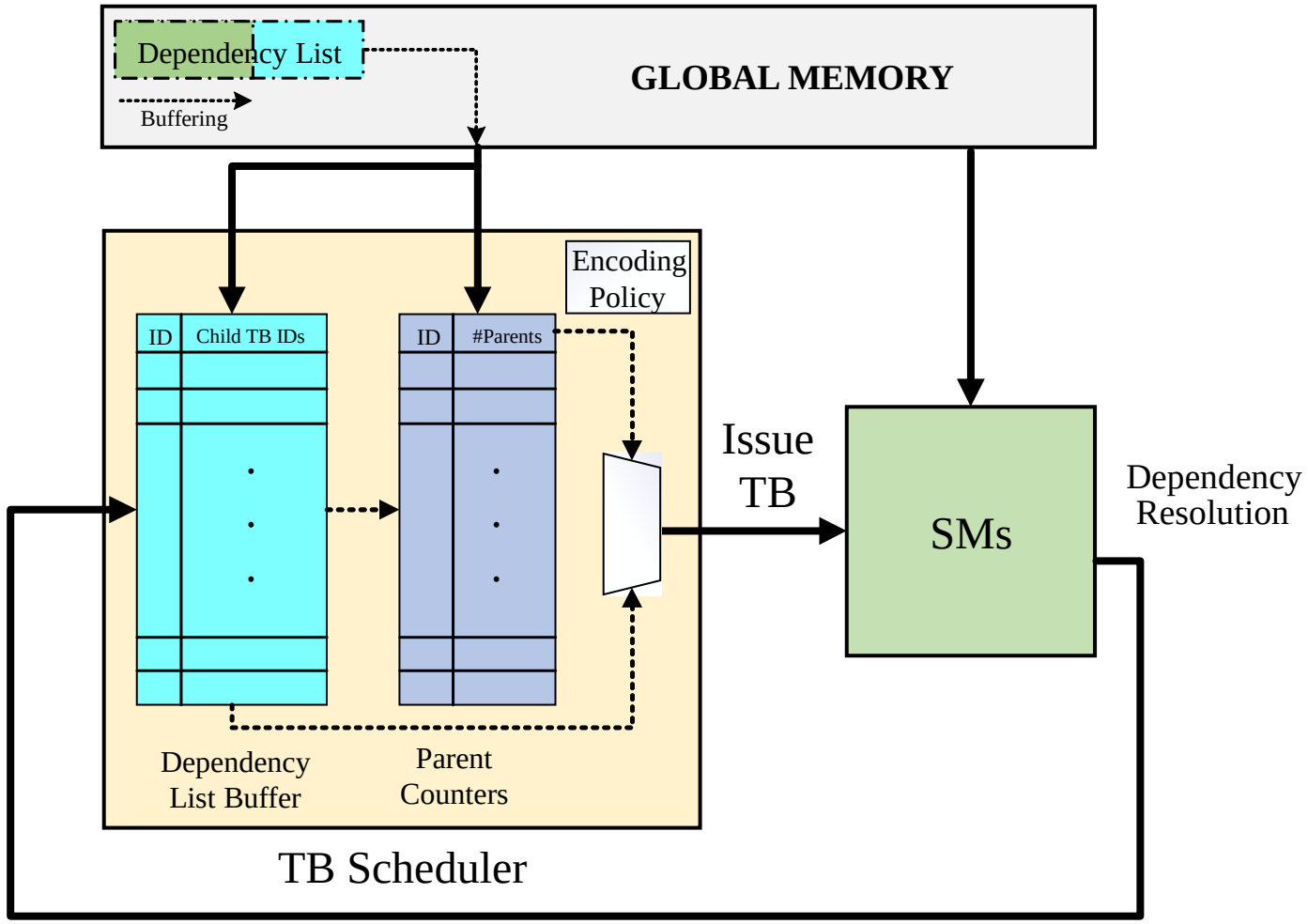
Static vs Non-static





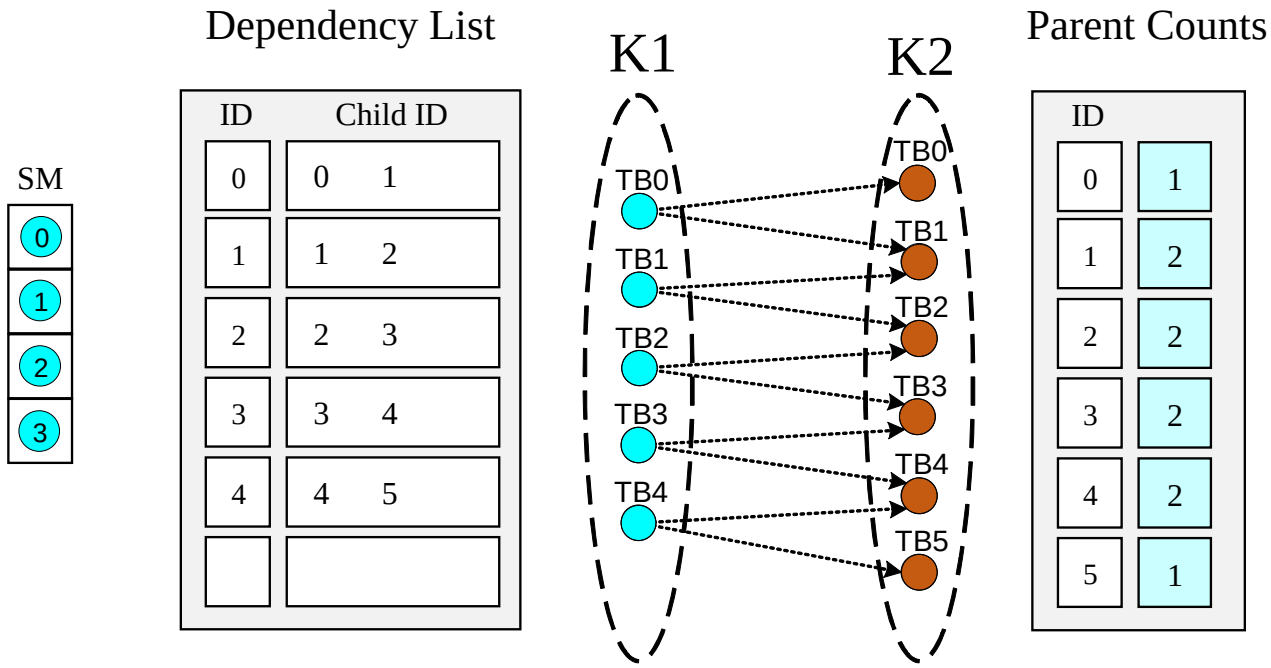
Hardware Support

- Dependency List Buffer
  - Child kernel TB IDs for parent kernel TBs
- Parent counter
  - # of unfinished parents for child kernel TBs



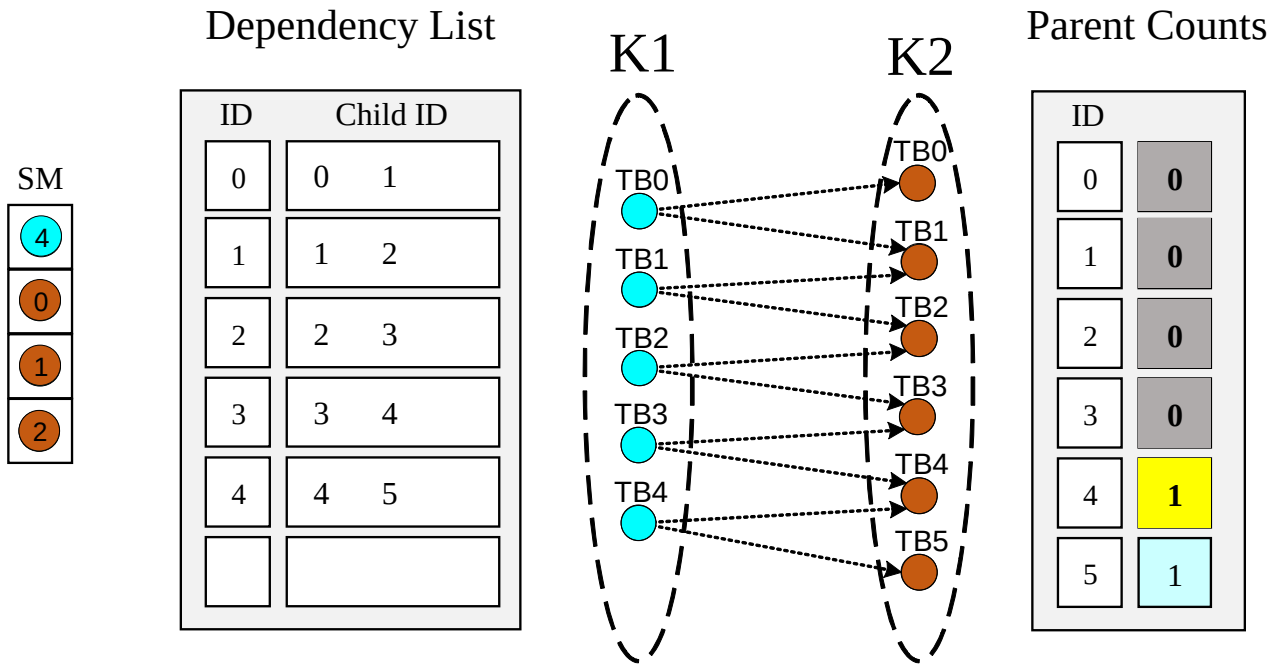
Example:
Enforcing Dependencies

- K1 and K2 are loaded in the GPU
- K1 TBs are scheduled and start executing



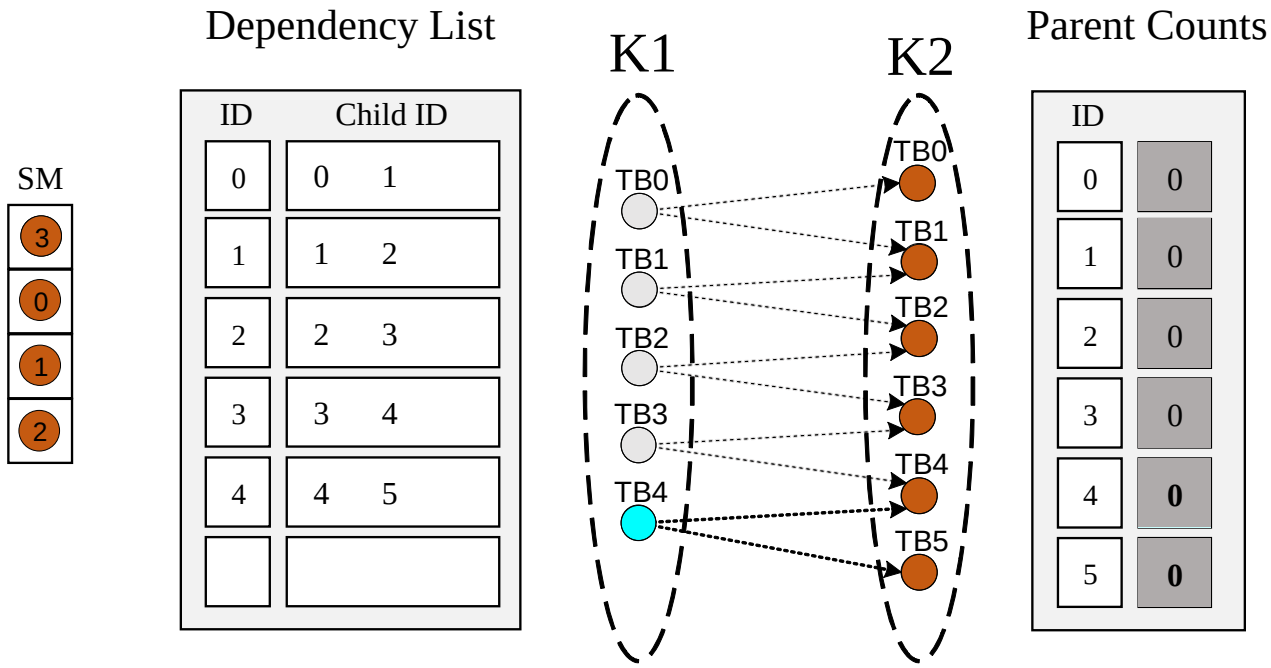
Example:
Enforcing Dependencies

- K1 TBs finish
  - The rest of K1 TBs are loaded
- Parent count of finished TBs decrements
  - Their child TBs from K2 can now begin execution



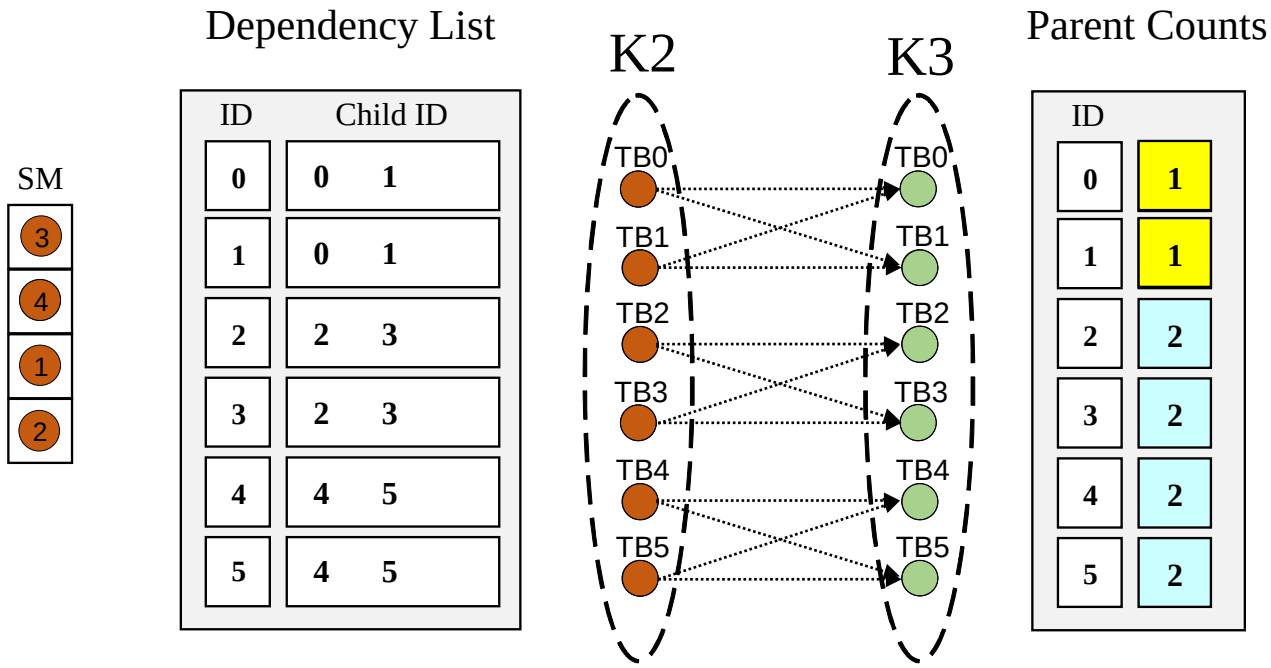
Example:
Enforcing Dependencies

- All TBs from K1 have finished
  - K1 is marked as finished.
  - K2 becomes the producer kernel for the next kernel to be loaded.
- All parent counts are invalidated
  - Other TBs from K2 can now run.

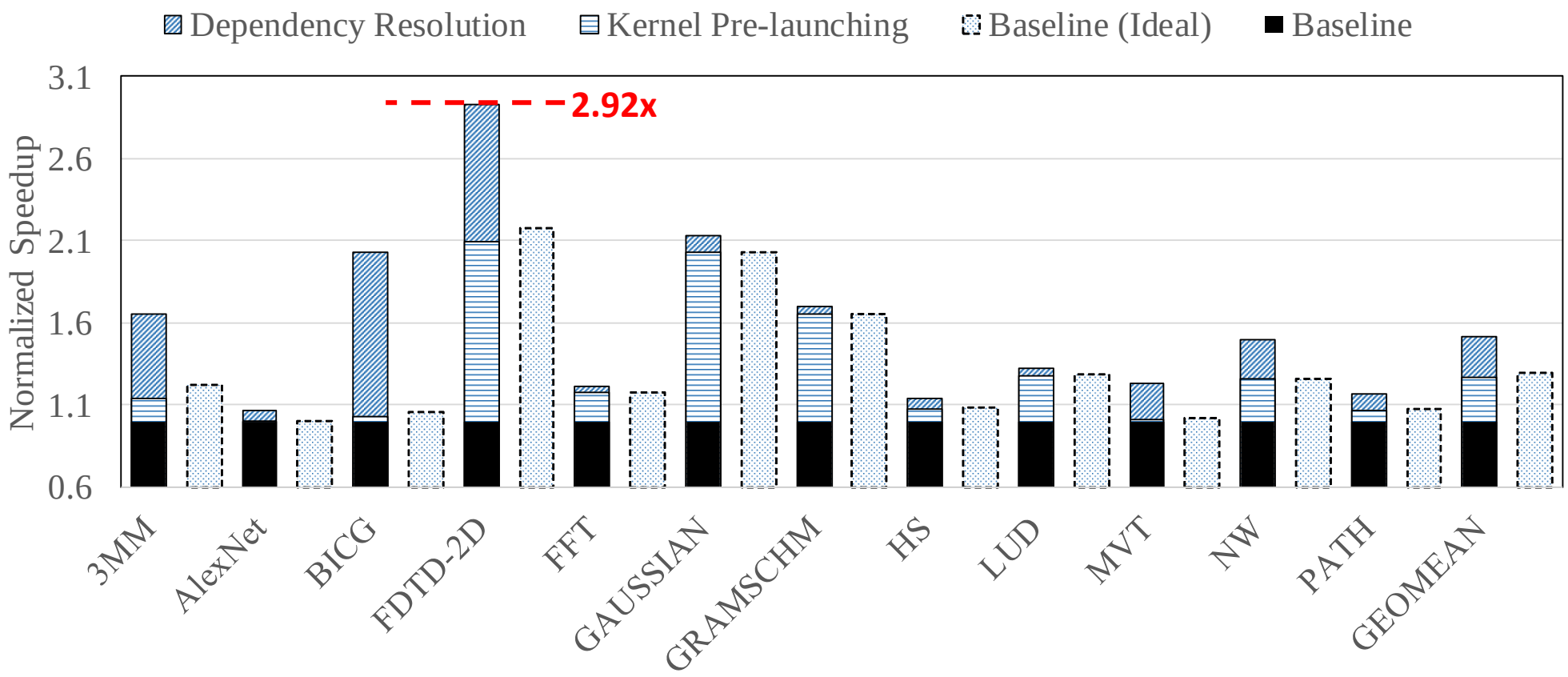


Example:
Enforcing Dependencies

- K3 is loaded as the new consuming kernel
- K3 dependency list is read from memory and loaded on TB scheduler



Performance



Works best for many smaller TBs

Step 3: SEER *(Ongoing work)*

Estimating Runtime Data Dependencies in GPU Applications

Overview

- BM only handles static data dependencies
- Uncertain fine-grained dependencies in non-static cases
  - Indirect memory access
  - Input-dependent data flow
- Possible to speculate non-static dependencies?
  - Ongoing work…

Example: Breadth-First Search (BFS)

```
__global__ void Kernel( … ) {
    int tid = blockIdx.x * MAX_THREADS_PER_BLOCK + threadIdx.x;
    if( tid < no_of_nodes && g_graph_mask[tid])
    {
        g_graph_mask[tid] = false;
        for(int i = g_graph_nodes[tid].starting
            i < (g_graph_nodes[tid].no_of_edges + g_graph_nodes[tid].starting);
            i++)
            {
            int id = g_graph_edges[i];
            if(!g_graph_visited[id])
                {
                g_cost[id] = g_cost[tid] + 1;
                g_updating_graph_mask[id] = true;
                }
            }
    }
}
```

① PC=0x050 (_1.ptx:77) ld.global.s8 %r5, [%rd3+0];

② PC=0x078 (_1.ptx:83) st.global.s8 [%rd3+0], %rh2;

③ PC=0x098 (_1.ptx:88) ld.global.s32 %r7, [%rd6+0];

④ PC=0x0a8 (_1.ptx:90) ld.global.s32 %r9, [%rd6+4];

⑤ PC=0x0f0 (_1.ptx:102) ld.global.s32 %r11, [%rd10+0];

⑥ PC=0x110 (_1.ptx:108) ld.global.s8 %r12, [%rd13+0];

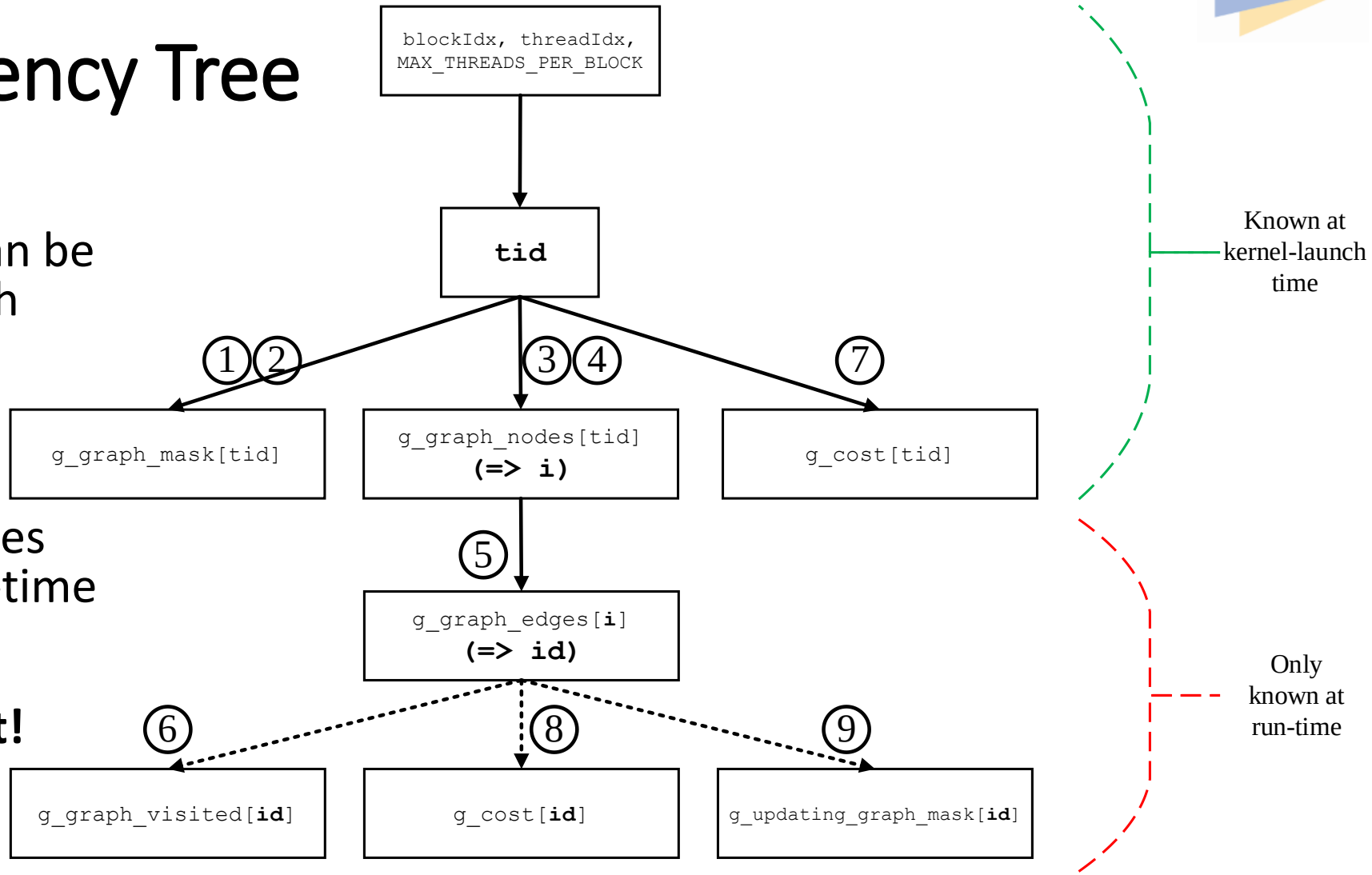
⑦ PC=0x148 (_1.ptx:116) ld.global.s32 %r14, [%rd16+0];

⑧ PC=0x168 (_1.ptx:120) st.global.s32 [%rd18+0], %r15;

⑨ PC=0x188 (_1.ptx:125) st.global.s8 [%rd20+0], %rh3;

Kernel Dependency Tree

- Static dependencies can be known at kernel-launch
  - e.g. `tid`
- Non-static dependencies are only known at run-time
- **Correlation could exist!**
  - **`tid` → `id`?**



Future Challenges

- How to represent the data?
- ML network?
- Input-dependent branches?
- Error in dependency resolution?