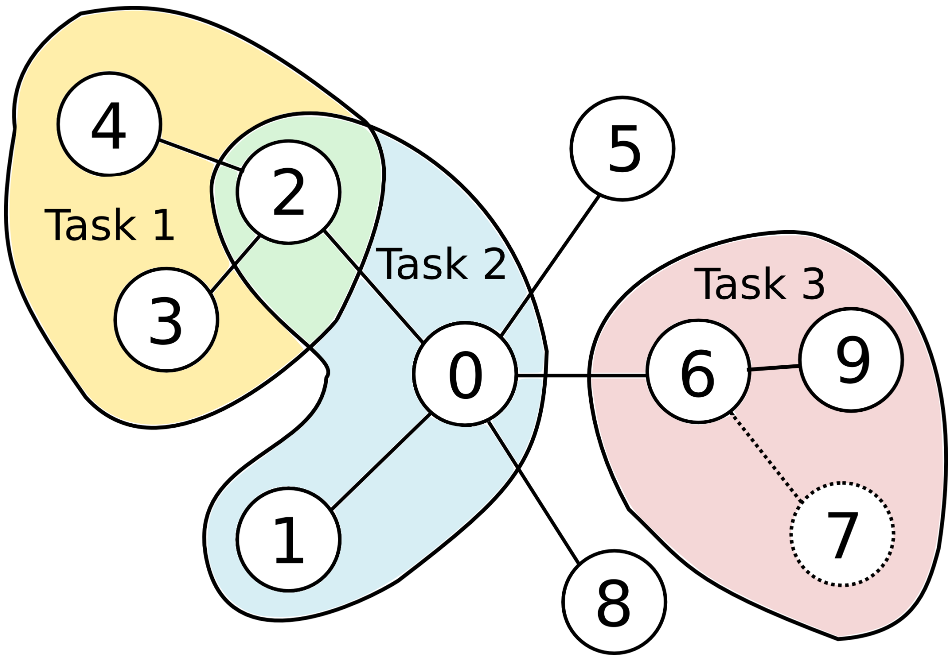
Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists

Ji Kim and Christopher Batten

Cornell University

IEEE/ACM International Symposium on Microarchitecture 2014 (MICRO-47)

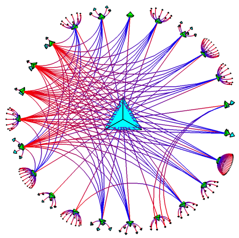
Amorphous Data Parallelism



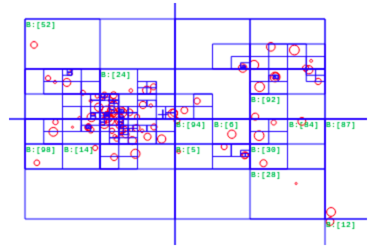
- Explored in-depth by Pingali et al. in PLDI 2011
- Generalization of conventional data parallelism
  - **Conflict**: Tasks can conflict with each other
  - **Dynamic**: New tasks can be generated dynamically
  - **Morph:** Tasks can modify the underlying data structure dynamically
- Difficult to map amorphous data parallelism to GPGPUs

Target Benchmarks (LonestarGPU)

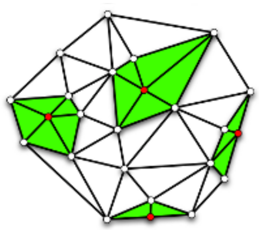
Breadth-First Search



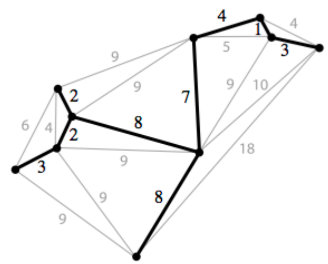
Barnes-Hut N-Body



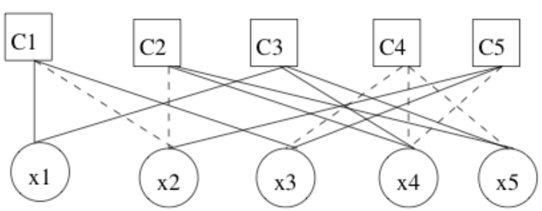
Delaunay Mesh Refinement



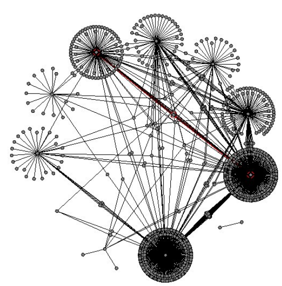
Minimum Spanning Tree



Survey Propagation



Single-Source Shortest-Path



Benchmark	Conflict	Dynamic	Morph
BH		X	
BFS	X	X	
SSSP	X	X	
DMR	X	X	X
MST	X	X	X
SP	X	X	X

Burtscher et al. A Quantitative Study of Irregular Programs on GPUs. IISWC 2012.

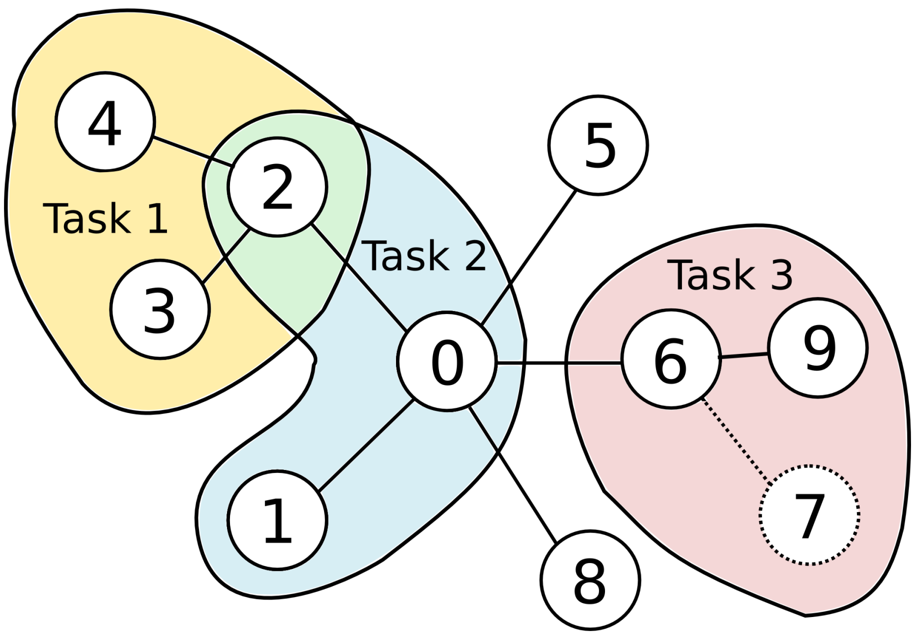
Previous Work on Software Optimizations

- *The Tao of Parallelism in Algorithms*, Pingali et al. (PLDI 2011)
- *A Quantitative Study of Irregular Programs on GPUs*, Burtscher et al. (IISWC 2012)
- *Data-Driven versus Topology-Driven Irregular Computations on GPUs*, Nasre et al. (IPDPS 2013)
- Many others…

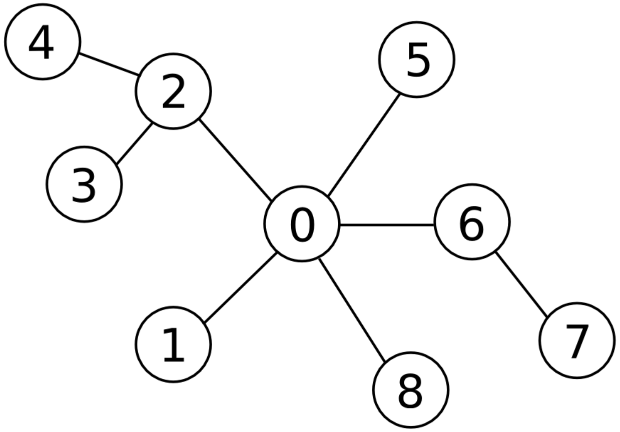
What can architects do to accelerate amorphous data parallel applications on GPGPUs?

Presentation Outline

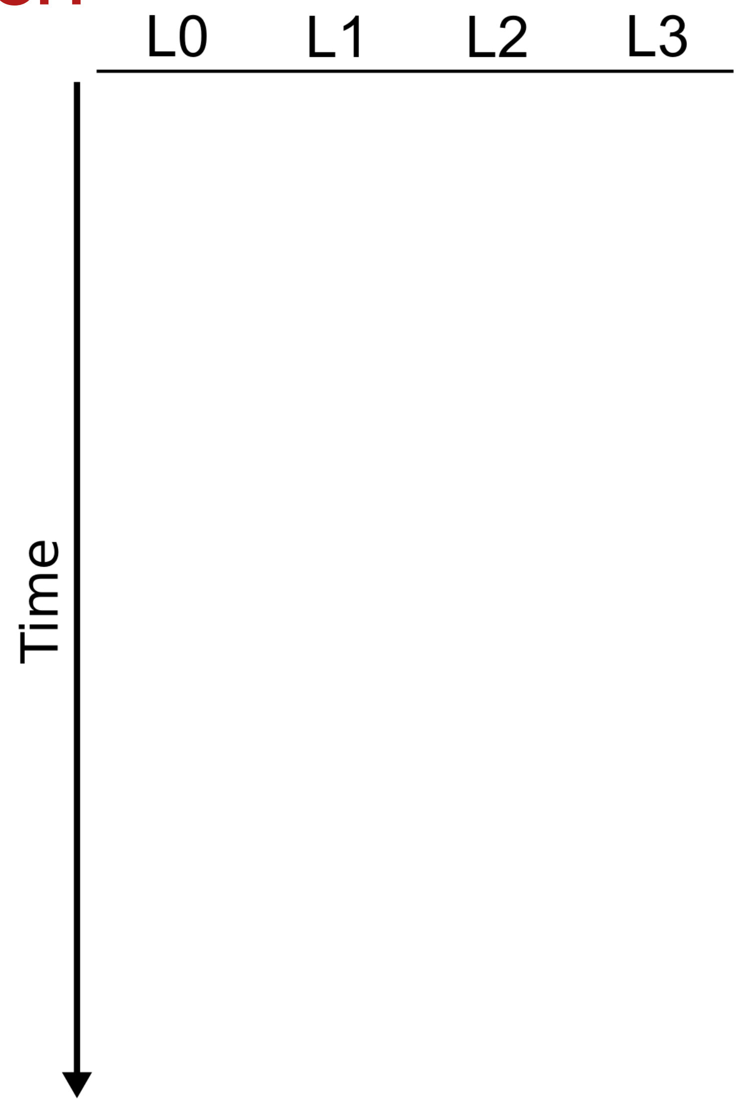
- Motivation
- **Mapping Irregular Algorithms to GPGPUs**
- Developing Optimized Software Baselines
- Fine-Grain Hardware Worklists
- Evaluation



Topology-Driven Approach

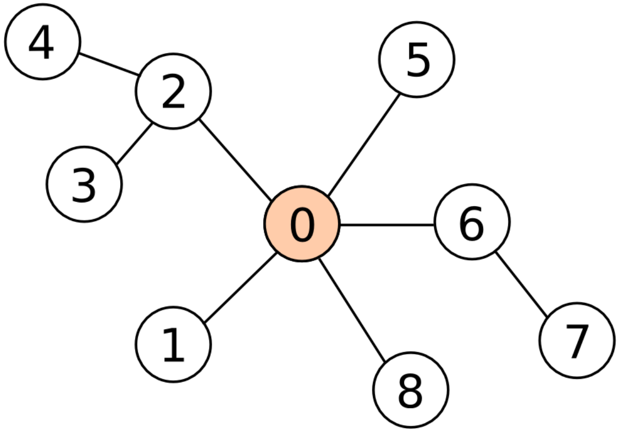



```
def topo_driven:
  idx = get_tid()
  my_node = nodes[idx]
  if check( my_node ):
    compute( my_node )
    *done_ptr = false

def main:
  done = false
  while not done:
    done = true
    topo_driven<<<N>>>( nodes, &done )
```

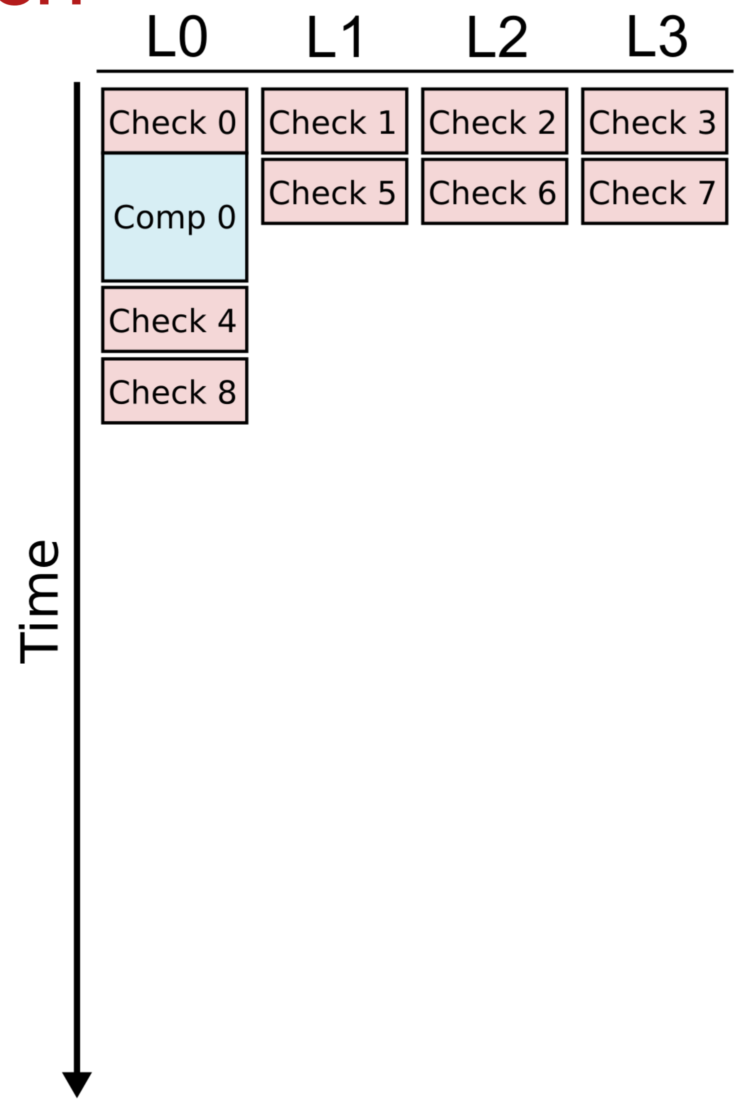


- **Low work efficiency!**

Topology-Driven Approach

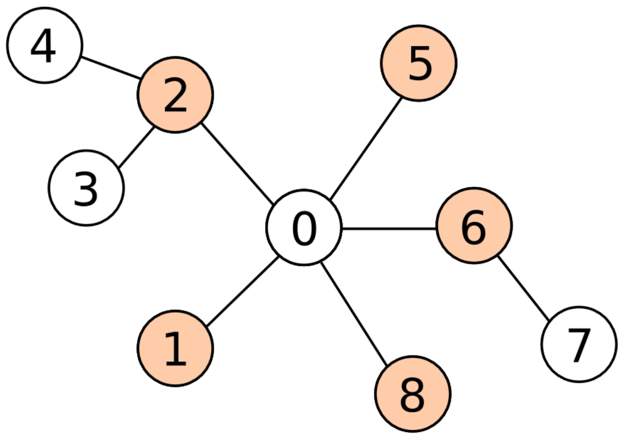



```
def topo_driven:
  idx = get_tid()
  my_node = nodes[idx]
  if check( my_node ):
    compute( my_node )
    *done_ptr = false

def main:
  done = false
  while not done:
    done = true
    topo_driven<<<N>>>( nodes, &done )
```

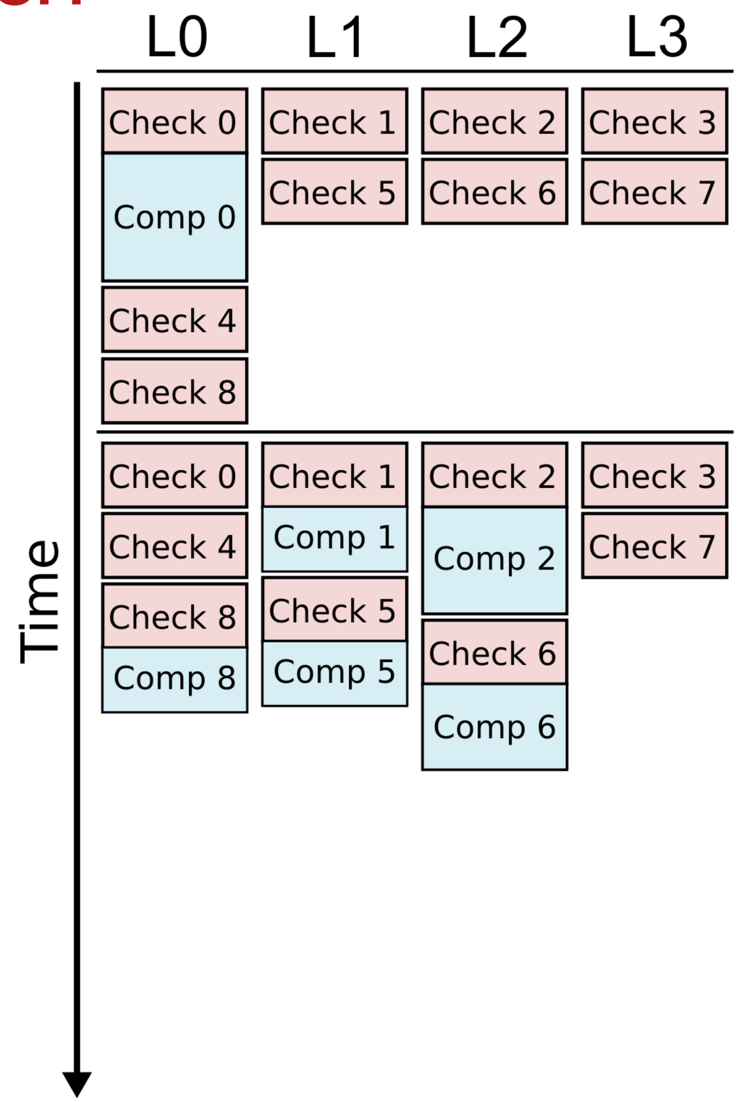


- **Low work efficiency!**

Topology-Driven Approach

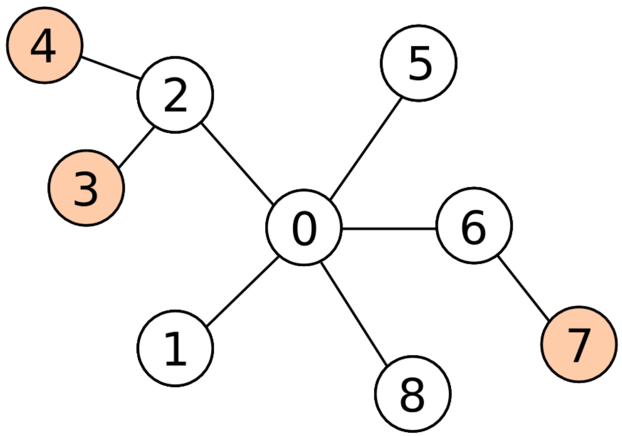



```
def topo_driven:
  idx = get_tid()
  my_node = nodes[idx]
  if check( my_node ):
    compute( my_node )
    *done_ptr = false

def main:
  done = false
  while not done:
    done = true
    topo_driven<<<N>>>( nodes, &done )
```

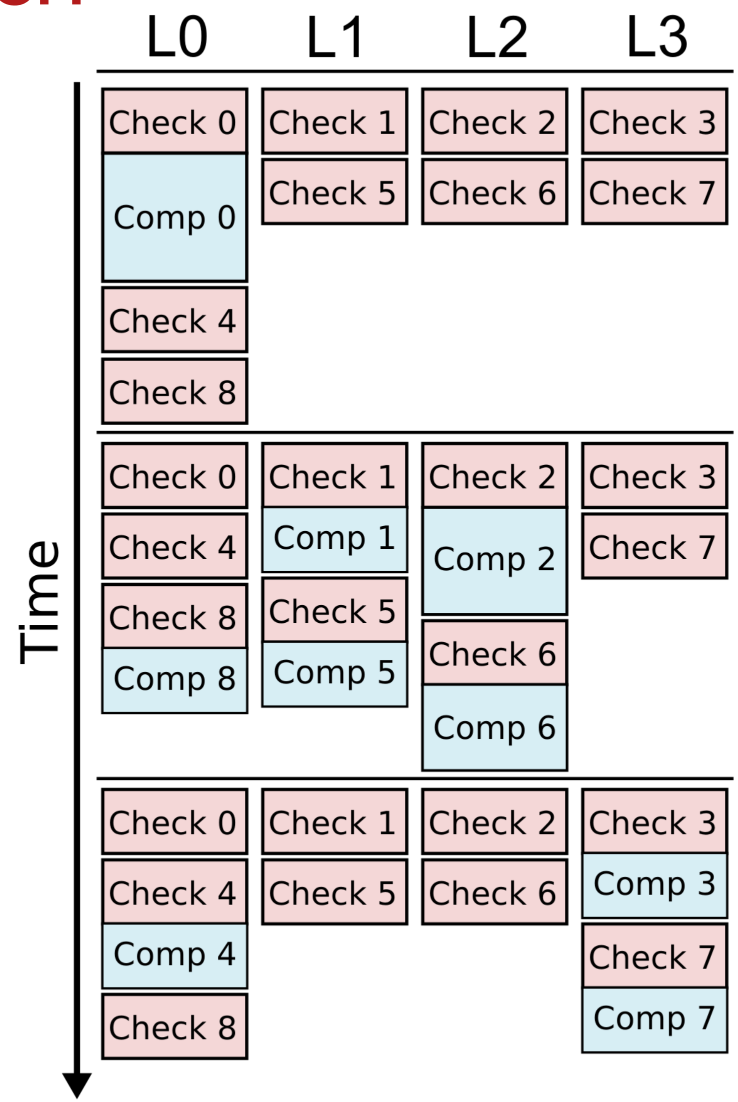


- **Low work efficiency!**

Topology-Driven Approach

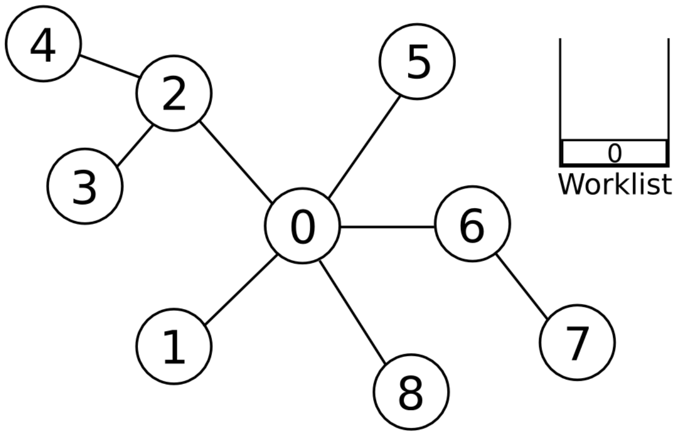


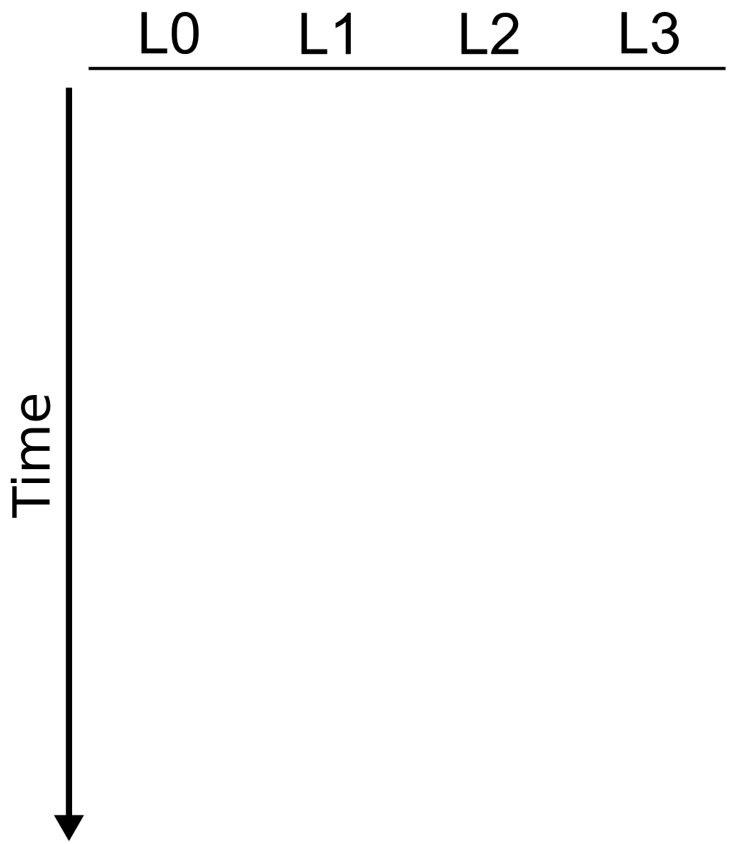

```
def topo_driven:
  idx = get_tid()
  my_node = nodes[idx]
  if check( my_node ):
    compute( my_node )
    *done_ptr = false

def main:
  done = false
  while not done:
    done = true
    topo_driven<<<N>>>( nodes, &done )
```



- **Low work efficiency!**

Data-Driven Approach

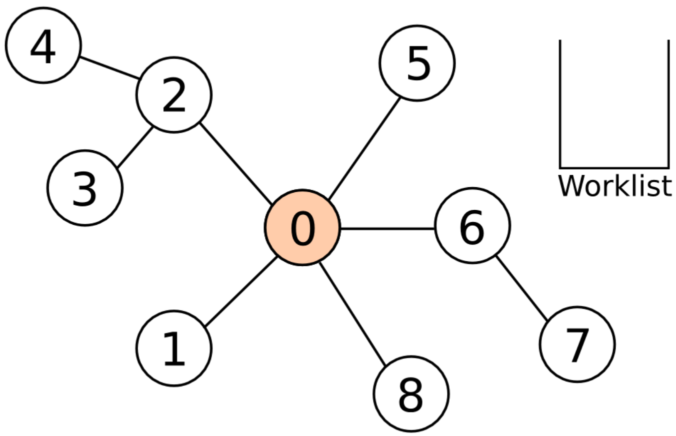




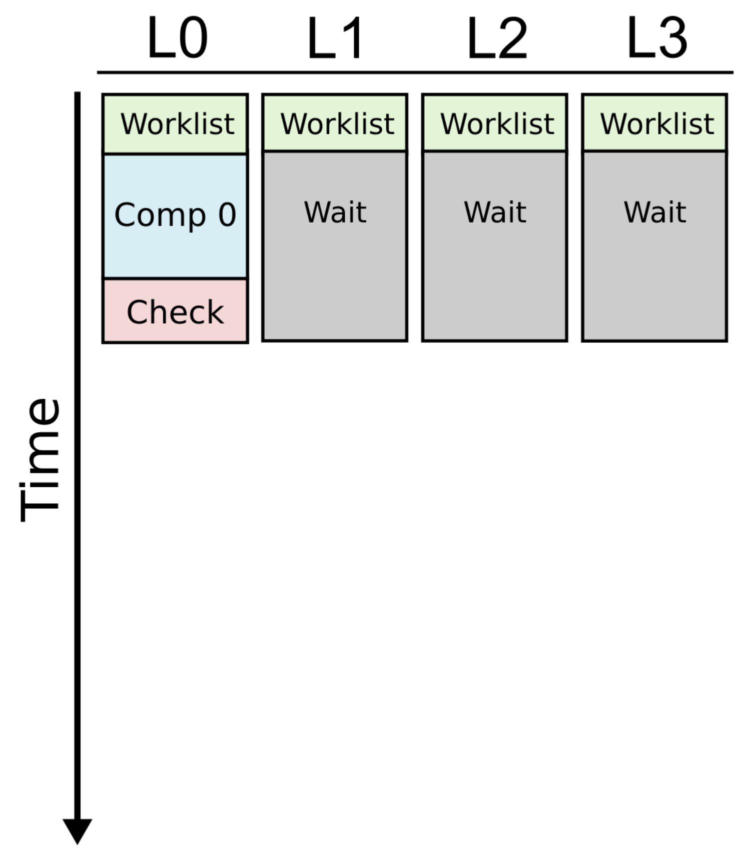

```
def data_driven:
  while idx = wl.pull():
    my_node = nodes[idx]
    compute( my_node )
    for all neighbors of my_node:
      if check( neighbor ):
        wl.push( idx )

def main:
  init_wl<<<N>>>( nodes, wl )
  data_driven<<<M>>>( nodes, wl )
```

- **High Memory Contention!**
- **SW Worklist Overhead!**

Data-Driven Approach

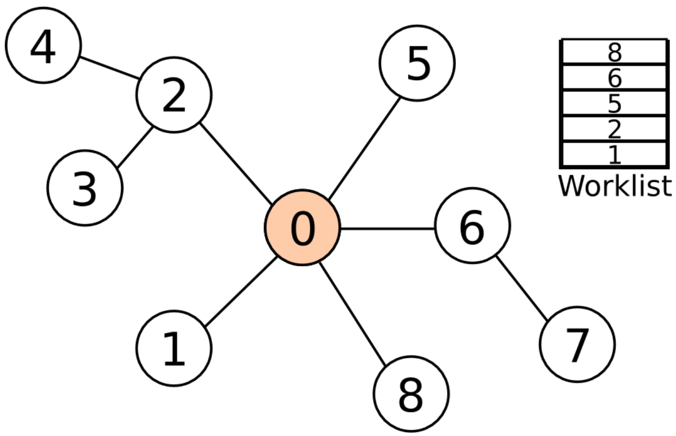


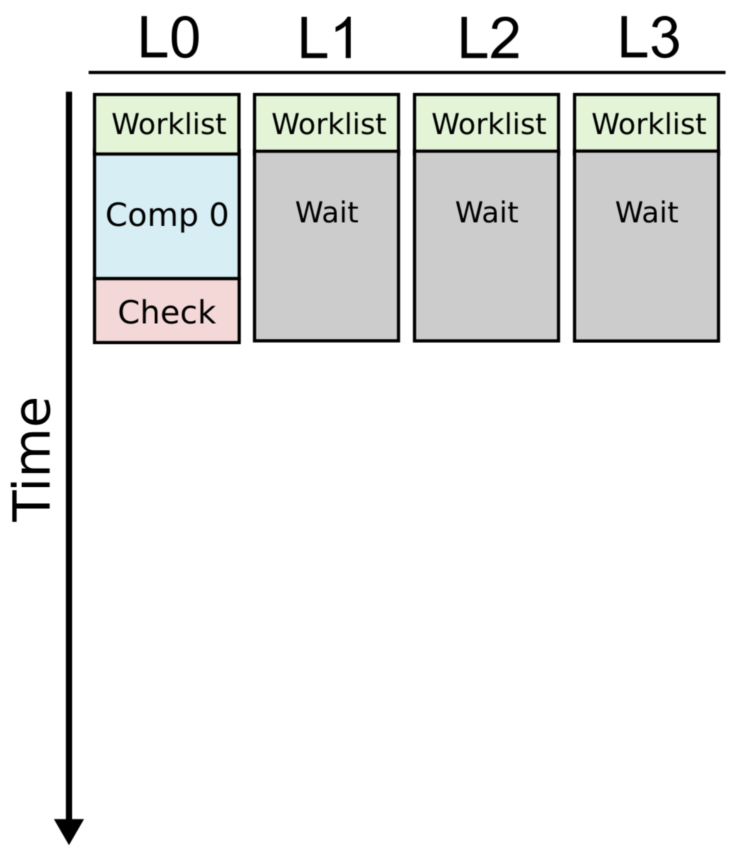

```
def data_driven:
  while idx = wl.pull():
    my_node = nodes[idx]
    compute( my_node )
    for all neighbors of my_node:
      if check( neighbor ):
        wl.push( idx )

def main:
  init_wl<<<N>>>( nodes, wl )
  data_driven<<<M>>>( nodes, wl )
```



- **High Memory Contention!**
- **SW Worklist Overhead!**

Data-Driven Approach

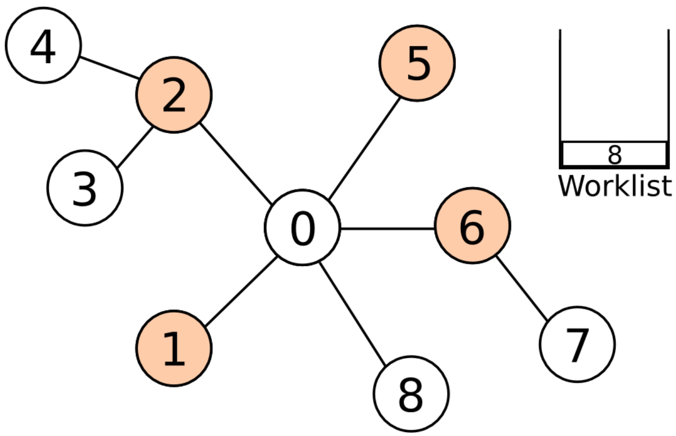


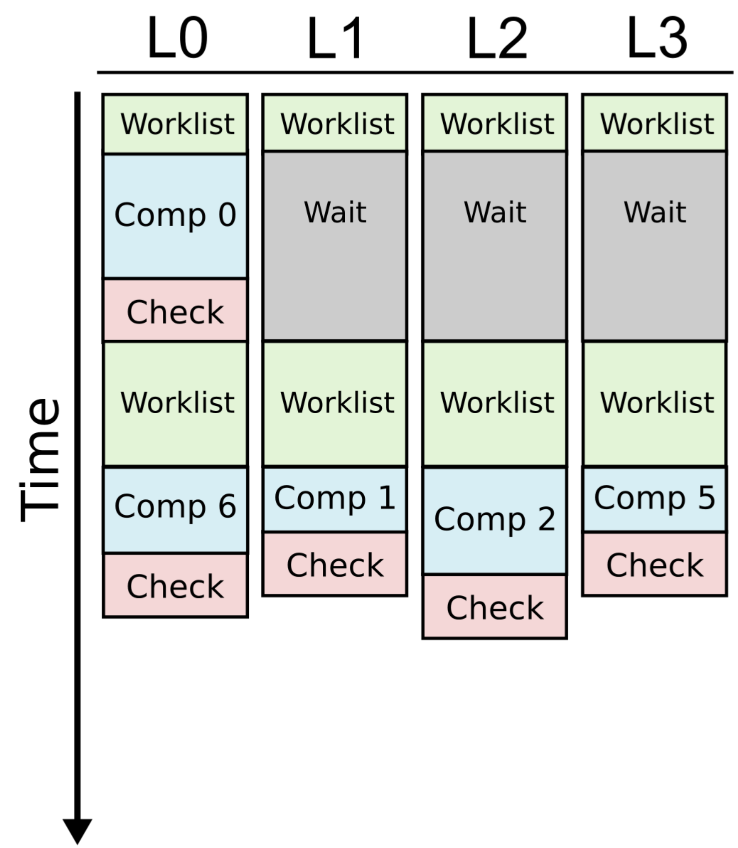



```
def data_driven:
  while idx = wl.pull():
    my_node = nodes[idx]
    compute( my_node )
    for all neighbors of my_node:
      if check( neighbor ):
        wl.push( idx )

def main:
  init_wl<<<N>>>( nodes, wl )
  data_driven<<<M>>>( nodes, wl )
```

- **High Memory Contention!**
- **SW Worklist Overhead!**

Data-Driven Approach

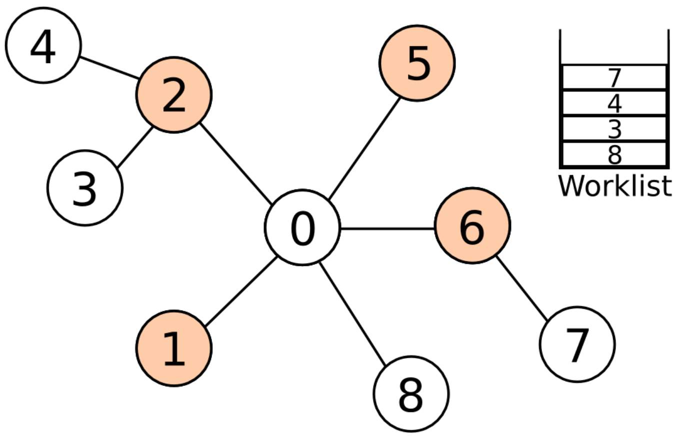


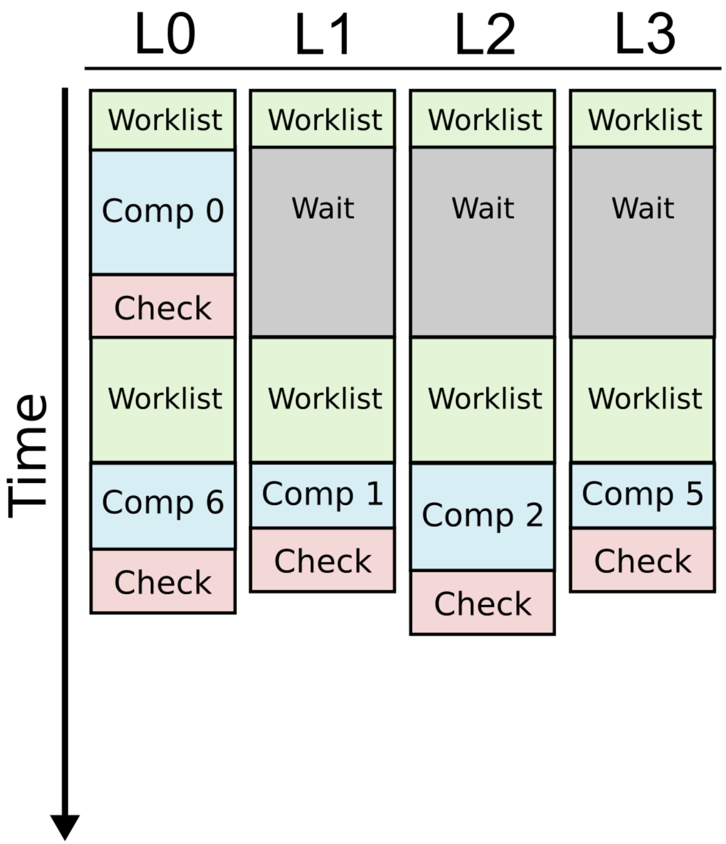



```
def data_driven:
  while idx = wl.pull():
    my_node = nodes[idx]
    compute( my_node )
    for all neighbors of my_node:
      if check( neighbor ):
        wl.push( idx )

def main:
  init_wl<<<N>>>( nodes, wl )
  data_driven<<<M>>>( nodes, wl )
```

- **High Memory Contention!**
- **SW Worklist Overhead!**

Data-Driven Approach

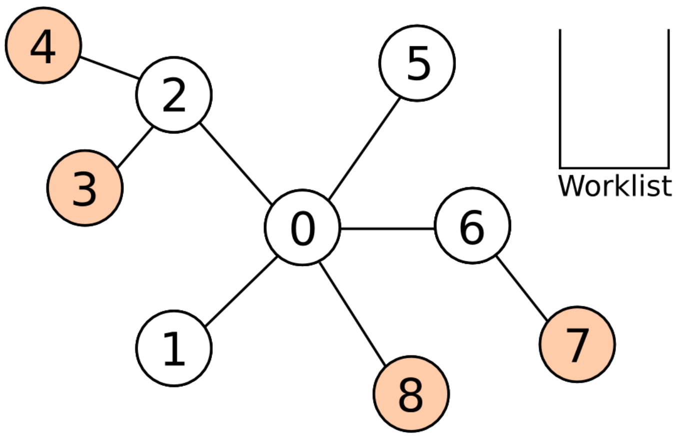




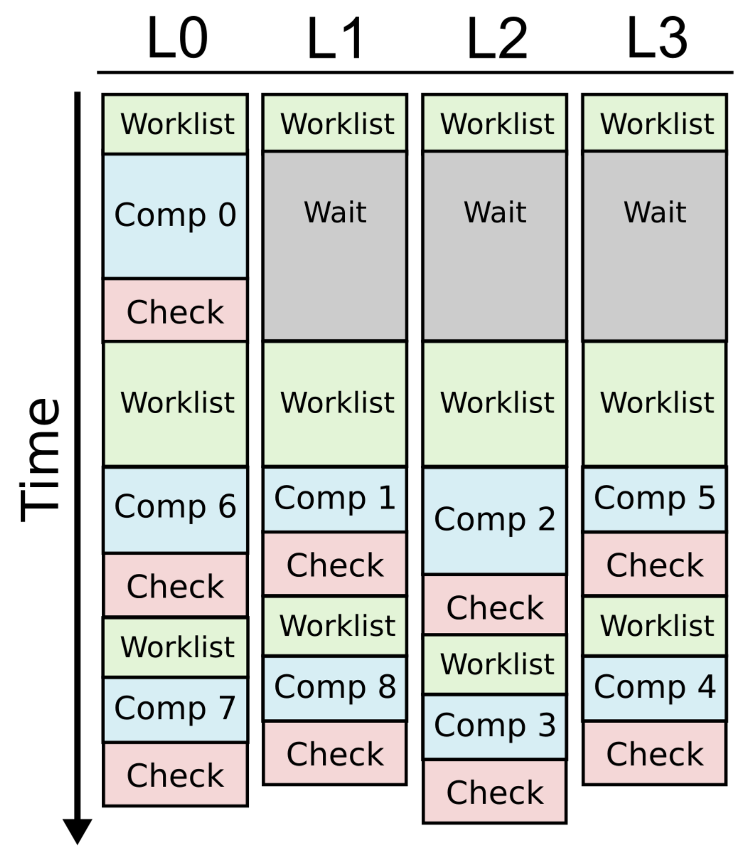

```
def data_driven:
  while idx = wl.pull():
    my_node = nodes[idx]
    compute( my_node )
    for all neighbors of my_node:
      if check( neighbor ):
        wl.push( idx )

def main:
  init_wl<<<N>>>( nodes, wl )
  data_driven<<<M>>>( nodes, wl )
```

- **High Memory Contention!**
- **SW Worklist Overhead!**

Data-Driven Approach



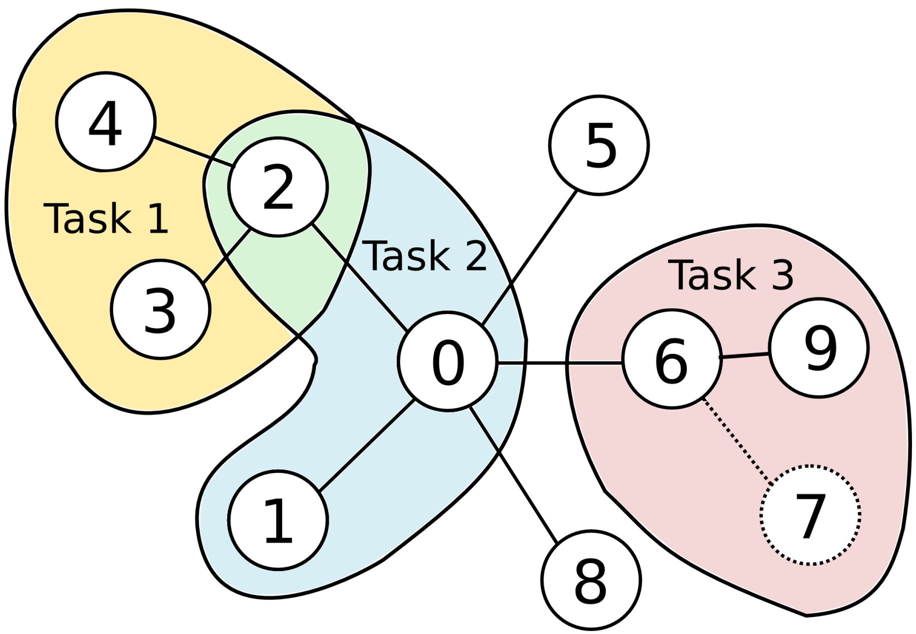

```
def data_driven:
  while idx = wl.pull():
    my_node = nodes[idx]
    compute( my_node )
    for all neighbors of my_node:
      if check( neighbor ):
        wl.push( idx )

def main:
  init_wl<<<N>>>( nodes, wl )
  data_driven<<<M>>>( nodes, wl )
```



- **High Memory Contention!**
- **SW Worklist Overhead!**

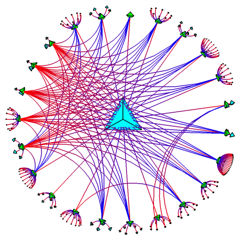
Presentation Outline

- Motivation
- Mapping Irregular Algorithms to GPGPUs
- **Developing Optimized Software Baselines**
- Fine-Grain Hardware Worklists
- Evaluation

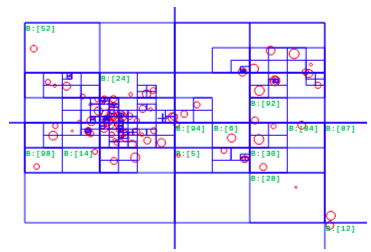


Developing Optimized SW Baselines

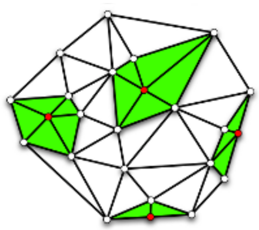
Breadth-First Search



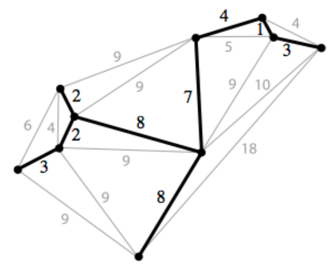
Barnes-Hut N-Body



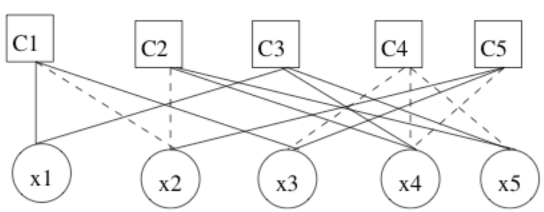
Delaunay Mesh Refinement



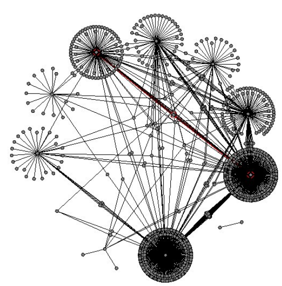
Minimum Spanning Tree



Survey Propagation

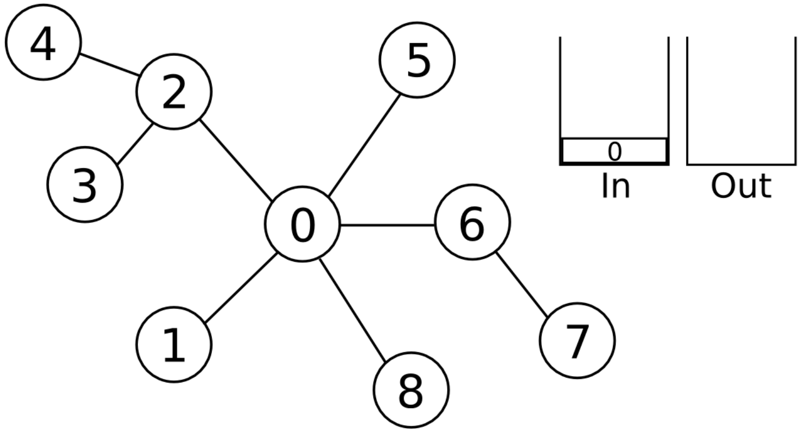


Single-Source Shortest-Path

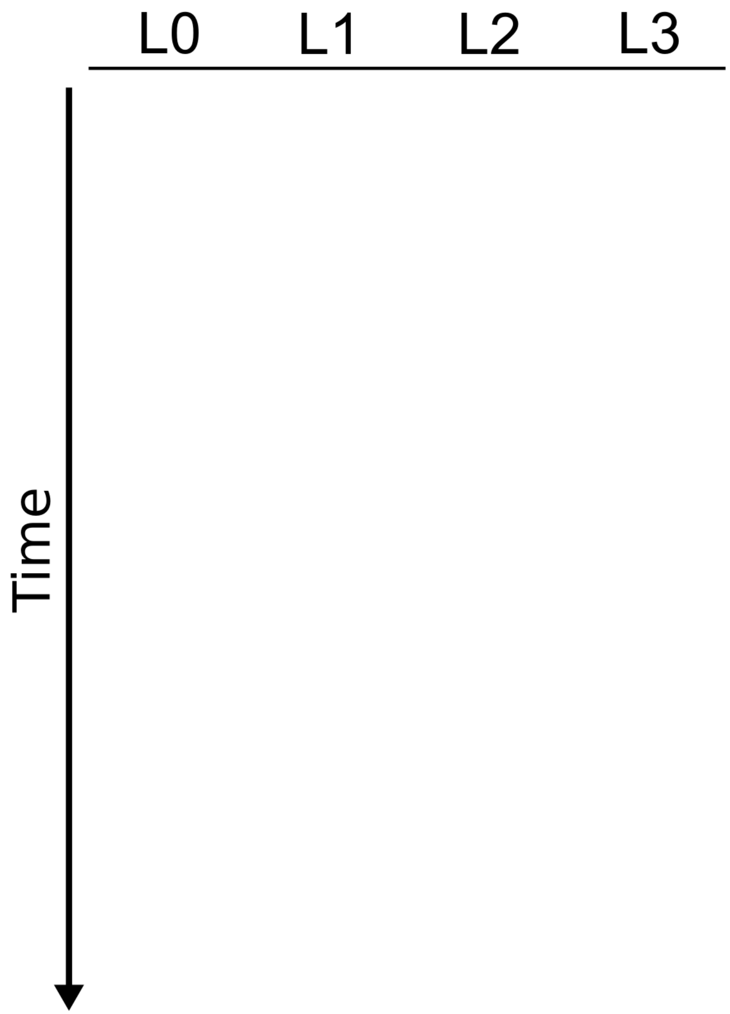


- LonestarGPU 1.02 only has topology-driven
- LonestarGPU 2.0 released but not better in all cases
- Missing some state-of-the-art optimizations
  - **Double-buffering**
  - Work chunking
  - Work donating
  - Variable kernel config

Double-Buffered Data-Driven Approach

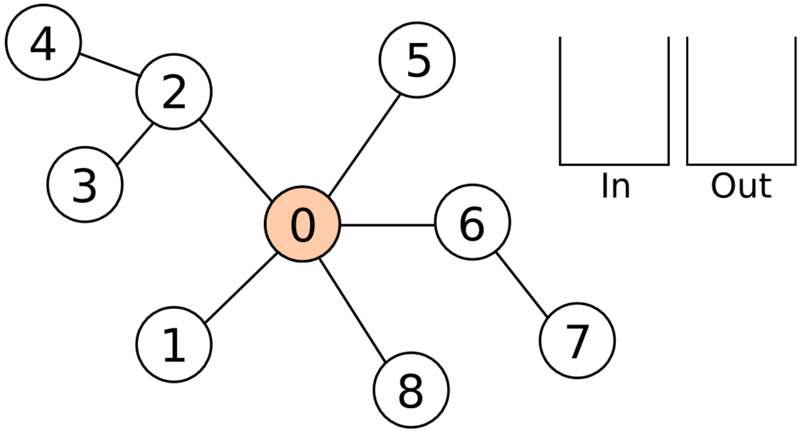



```
def data_driven:
  for wid in range( start, end ):
    idx = inwl.pull( wid )
    my_node = nodes[idx]
    compute( my_node )
    for all neighbors of my_node:
      if check( neighbor ):
        outwl.push( neighbor.idx )

def main:
  init_wl<<<N>>>( nodes, inwl )
  while not inwl.empty():
    data_driven<<<M>>>( nodes, inwl, outwl )
    swap( outwl, inwl )
```

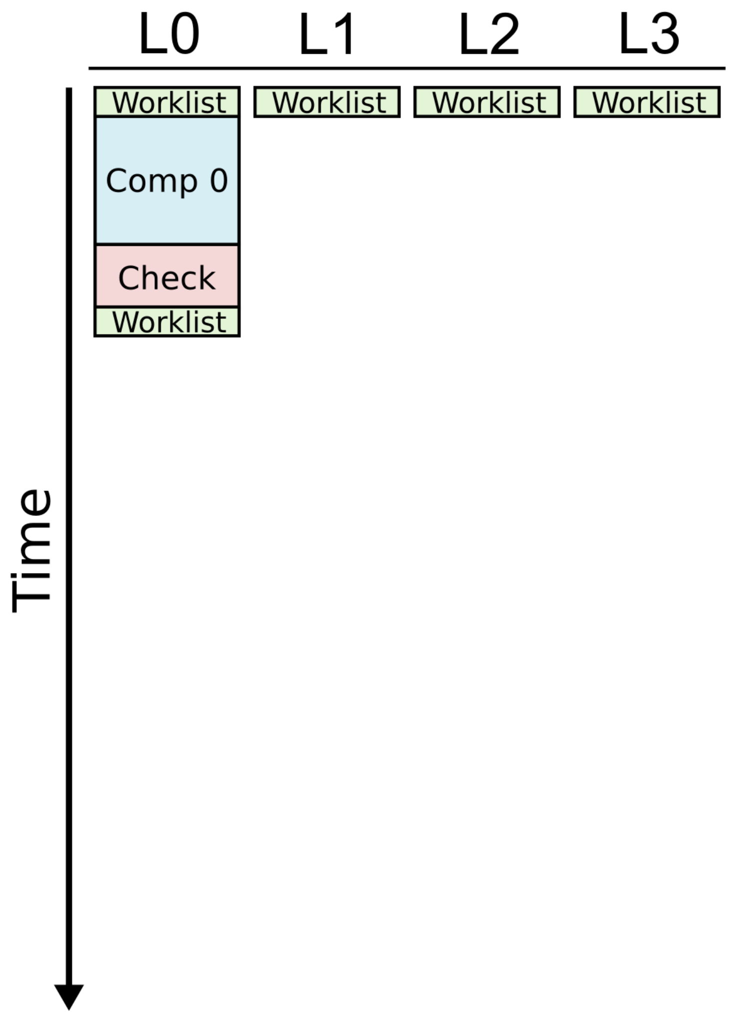


- **Less load balancing!**

Double-Buffered Data-Driven Approach

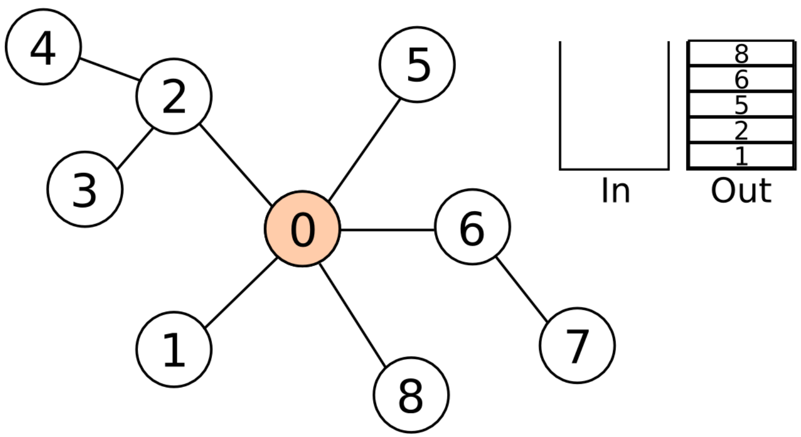



```
def data_driven:
  for wid in range( start, end ):
    idx = inwl.pull( wid )
    my_node = nodes[idx]
    compute( my_node )
    for all neighbors of my_node:
      if check( neighbor ):
        outwl.push( neighbor.idx )

def main:
  init_wl<<<N>>>( nodes, inwl )
  while not inwl.empty():
    data_driven<<<M>>>( nodes, inwl, outwl )
    swap( outwl, inwl )
```

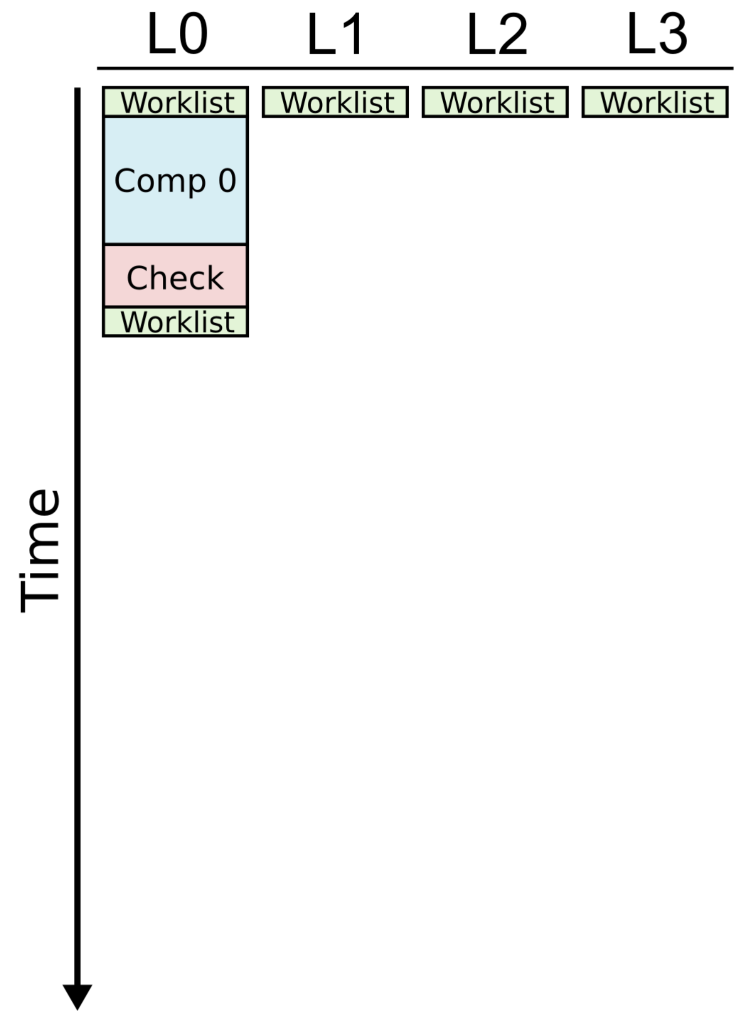


- **Less load balancing!**

Double-Buffered Data-Driven Approach

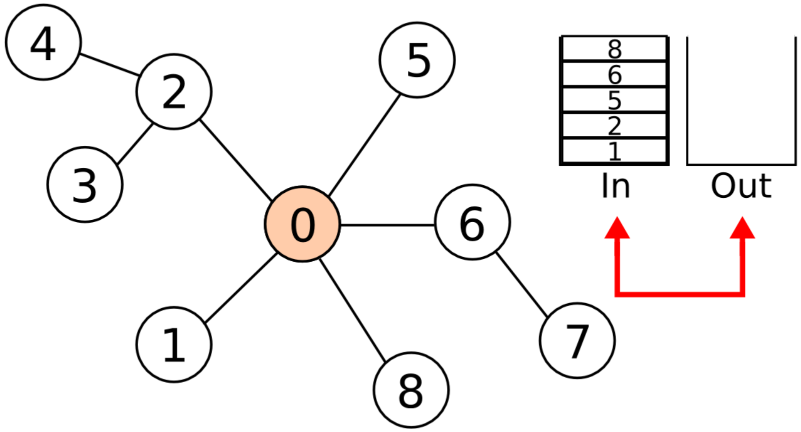



```
def data_driven:
  for wid in range( start, end ):
    idx = inwl.pull( wid )
    my_node = nodes[idx]
    compute( my_node )
    for all neighbors of my_node:
      if check( neighbor ):
        outwl.push( neighbor.idx )

def main:
  init_wl<<<N>>>( nodes, inwl )
  while not inwl.empty():
    data_driven<<<M>>>( nodes, inwl, outwl )
    swap( outwl, inwl )
```

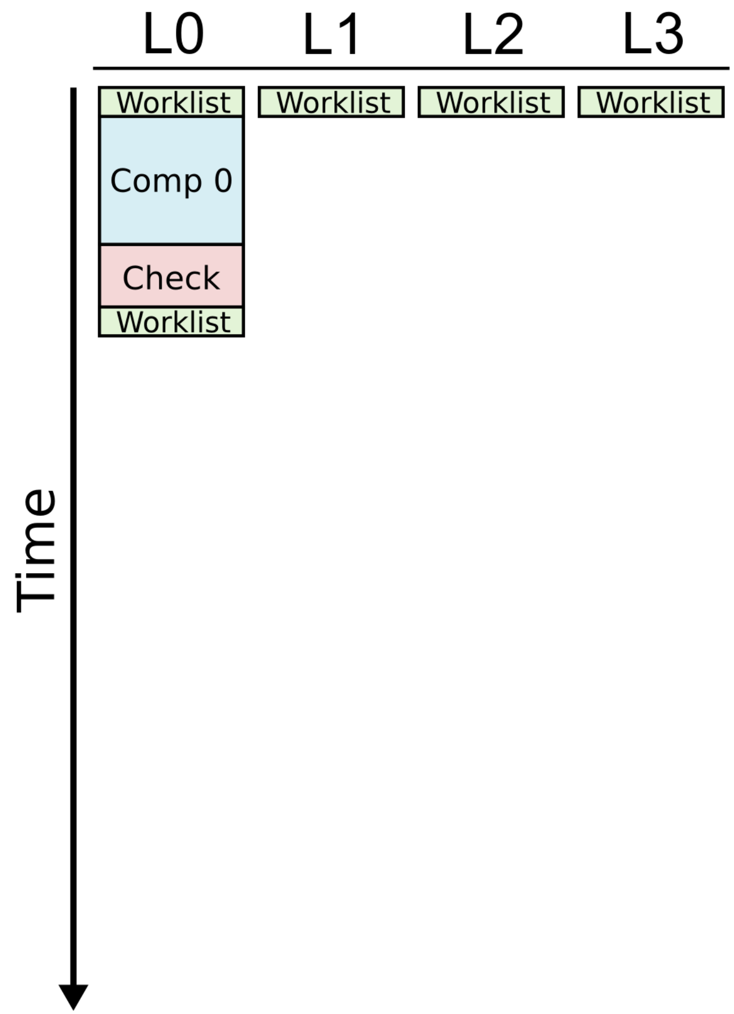


- **Less load balancing!**

Double-Buffered Data-Driven Approach

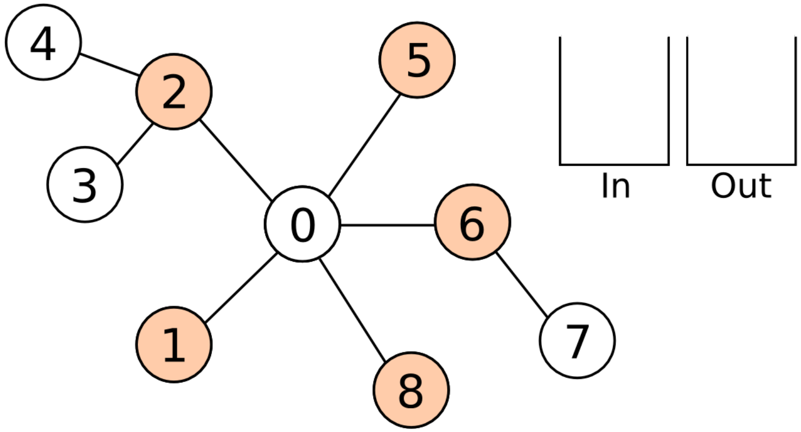



```
def data_driven:
  for wid in range( start, end ):
    idx = inwl.pull( wid )
    my_node = nodes[idx]
    compute( my_node )
    for all neighbors of my_node:
      if check( neighbor ):
        outwl.push( neighbor.idx )

def main:
  init_wl<<<N>>>( nodes, inwl )
  while not inwl.empty():
    data_driven<<<M>>>( nodes, inwl, outwl )
    swap( outwl, inwl )
```

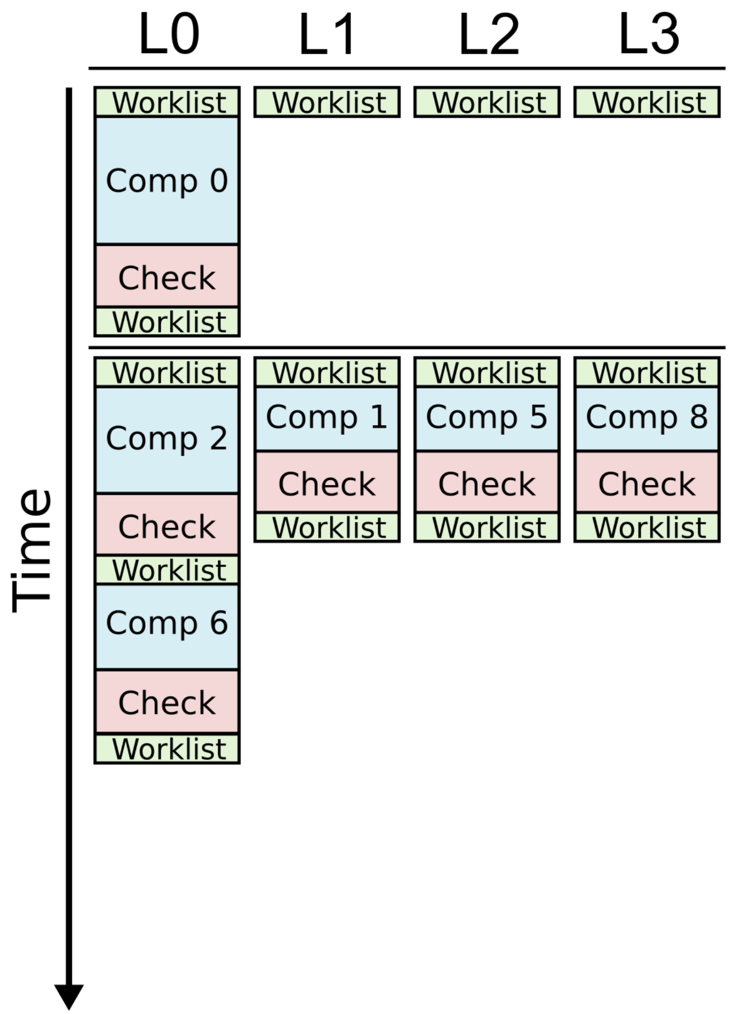


- **Less load balancing!**

Double-Buffered Data-Driven Approach

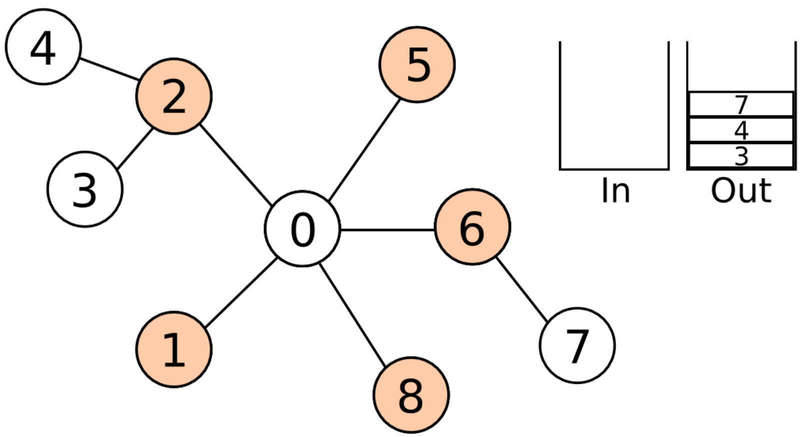



```
def data_driven:
  for wid in range( start, end ):
    idx = inwl.pull( wid )
    my_node = nodes[idx]
    compute( my_node )
    for all neighbors of my_node:
      if check( neighbor ):
        outwl.push( neighbor.idx )

def main:
  init_wl<<<N>>>( nodes, inwl )
  while not inwl.empty():
    data_driven<<<M>>>( nodes, inwl, outwl )
    swap( outwl, inwl )
```

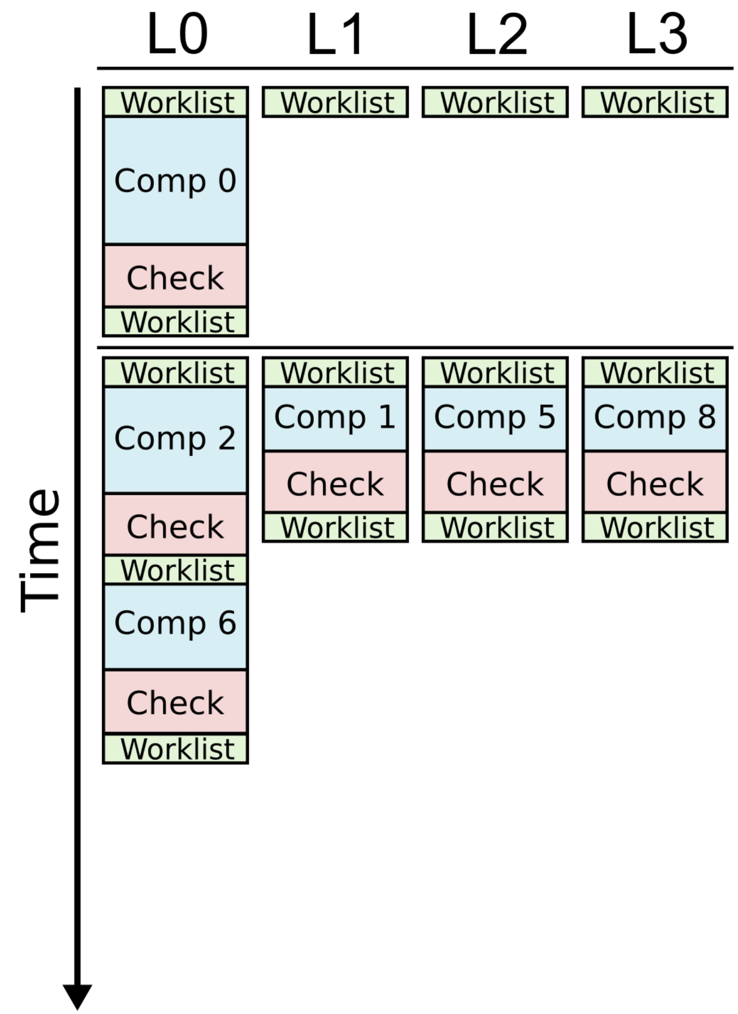


- **Less load balancing!**

Double-Buffered Data-Driven Approach

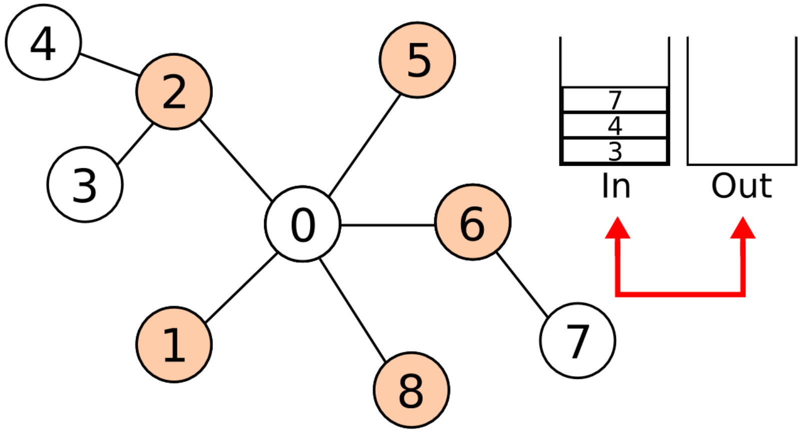



```
def data_driven:
  for wid in range( start, end ):
    idx = inwl.pull( wid )
    my_node = nodes[idx]
    compute( my_node )
    for all neighbors of my_node:
      if check( neighbor ):
        outwl.push( neighbor.idx )

def main:
  init_wl<<<N>>>( nodes, inwl )
  while not inwl.empty():
    data_driven<<<M>>>( nodes, inwl, outwl )
    swap( outwl, inwl )
```

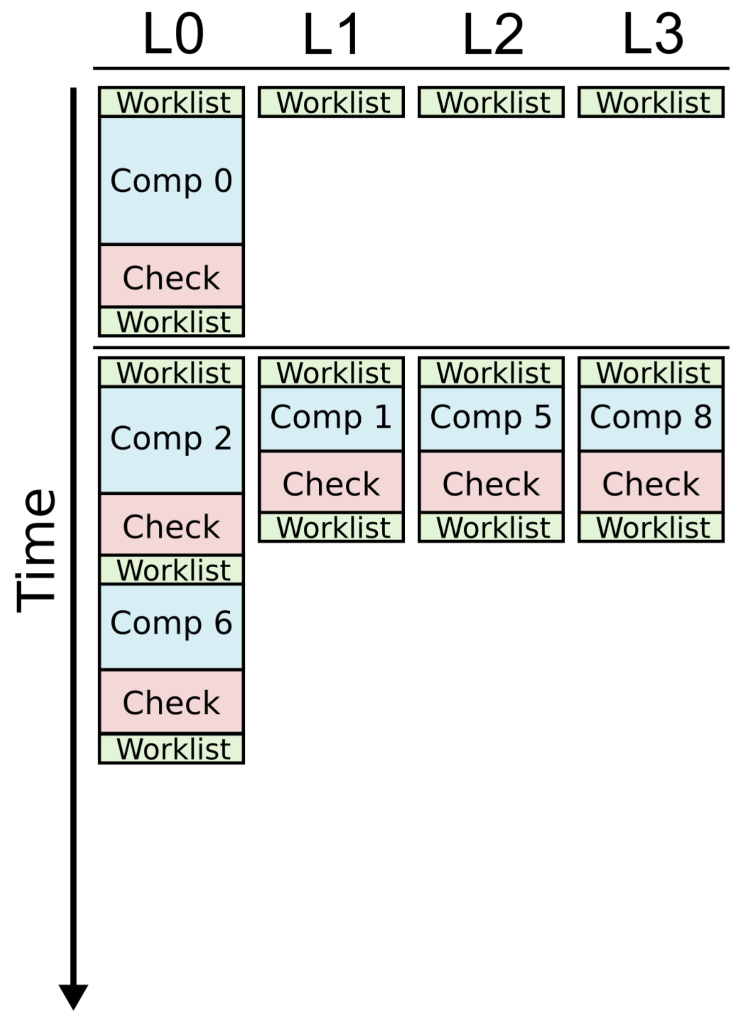


- **Less load balancing!**

Double-Buffered Data-Driven Approach

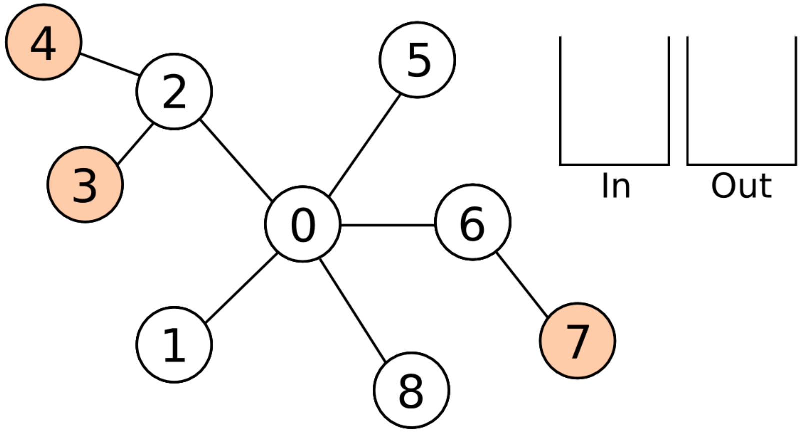



```
def data_driven:
  for wid in range( start, end ):
    idx = inwl.pull( wid )
    my_node = nodes[idx]
    compute( my_node )
    for all neighbors of my_node:
      if check( neighbor ):
        outwl.push( neighbor.idx )

def main:
  init_wl<<<N>>>( nodes, inwl )
  while not inwl.empty():
    data_driven<<<M>>>( nodes, inwl, outwl )
    swap( outwl, inwl )
```

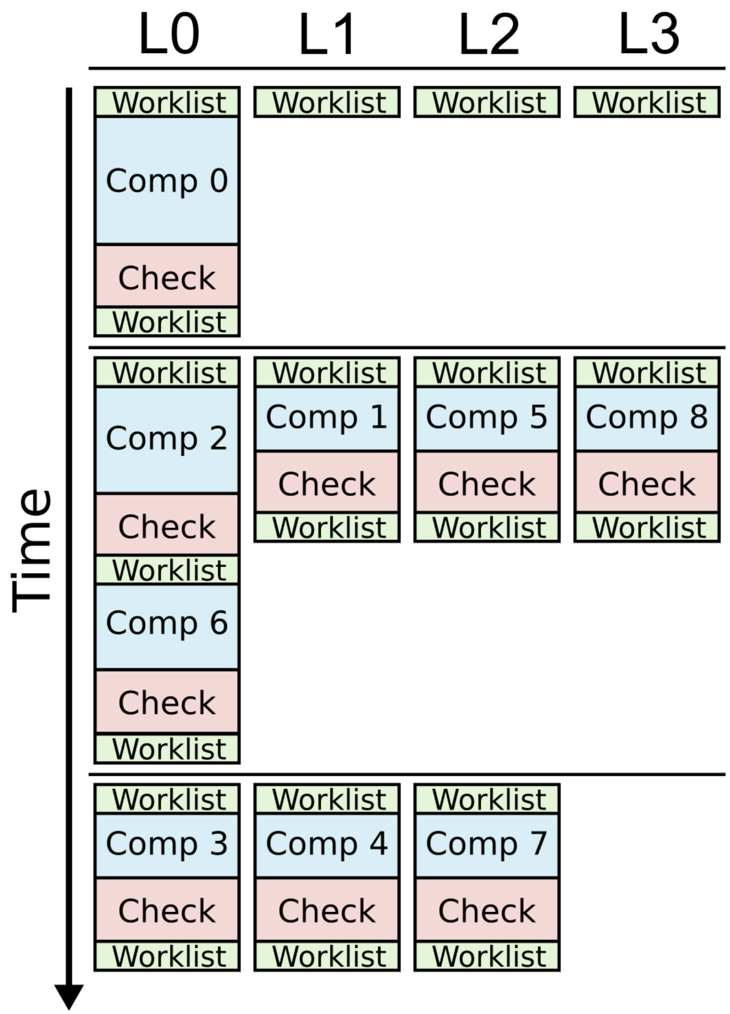


- **Less load balancing!**

Double-Buffered Data-Driven Approach

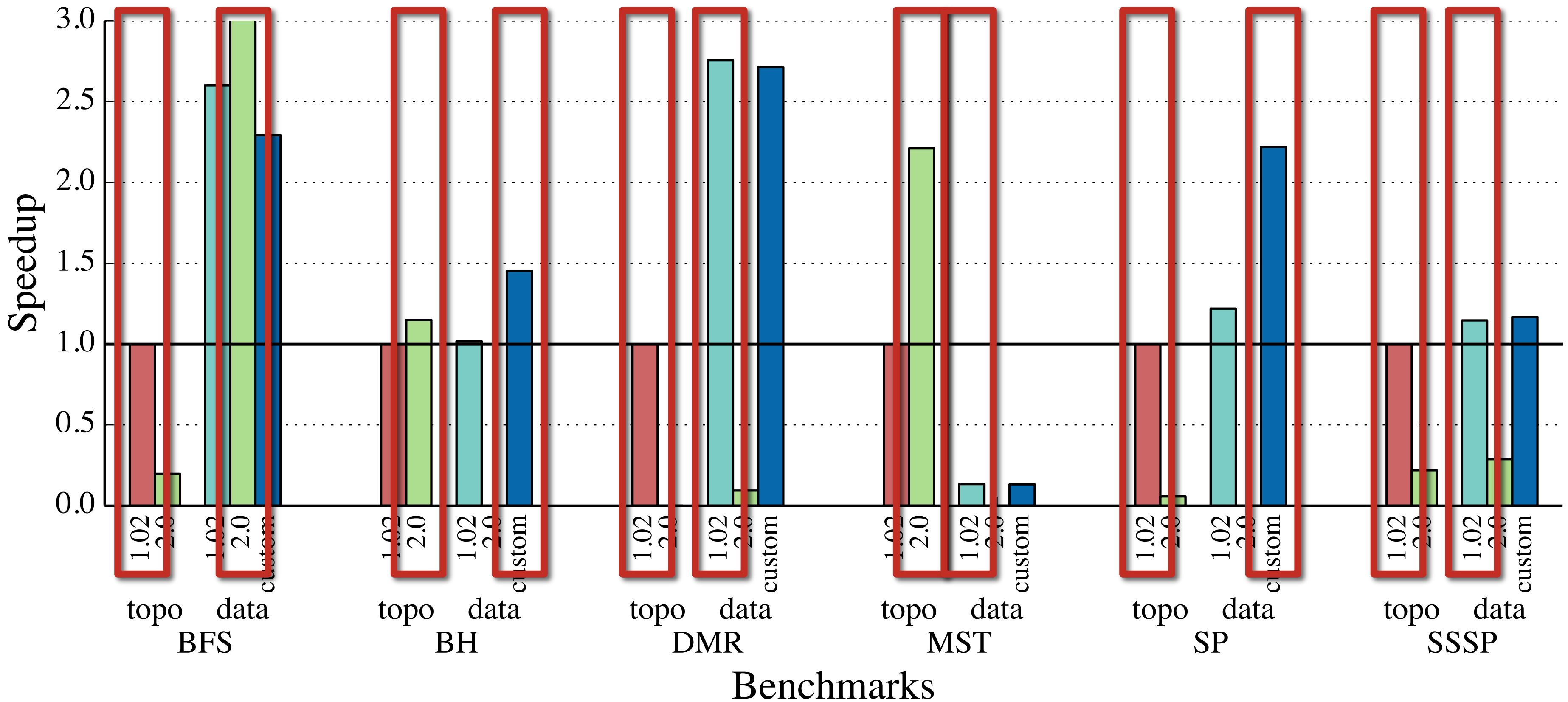



```
def data_driven:
  for wid in range( start, end ):
    idx = inwl.pull( wid )
    my_node = nodes[idx]
    compute( my_node )
    for all neighbors of my_node:
      if check( neighbor ):
        outwl.push( neighbor.idx )

def main:
  init_wl<<<N>>>( nodes, inwl )
  while not inwl.empty():
    data_driven<<<M>>>( nodes, inwl, outwl )
    swap( outwl, inwl )
```



- **Less load balancing!**

Comparison of LonestarGPU Versions



- Experiments on NVIDIA Tesla C2075 GPU
- Choose best topology- and data-driven for each benchmark
- Data-driven outperforms topology-driven in most cases

Room for Improvement

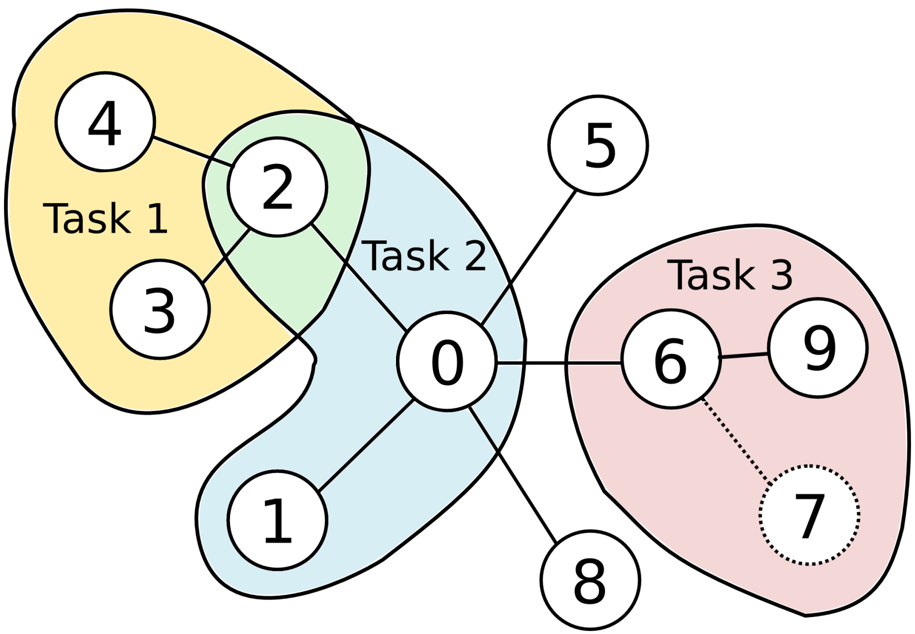
- Even with optimizations, data-driven approaches still have some weaknesses:
  - Memory contention on pushes
  - Suboptimal load balancing
  - SW overhead from worklist

- Significant time and effort to implement optimizations, performance not always guaranteed!

Can we use hardware to address these weaknesses?

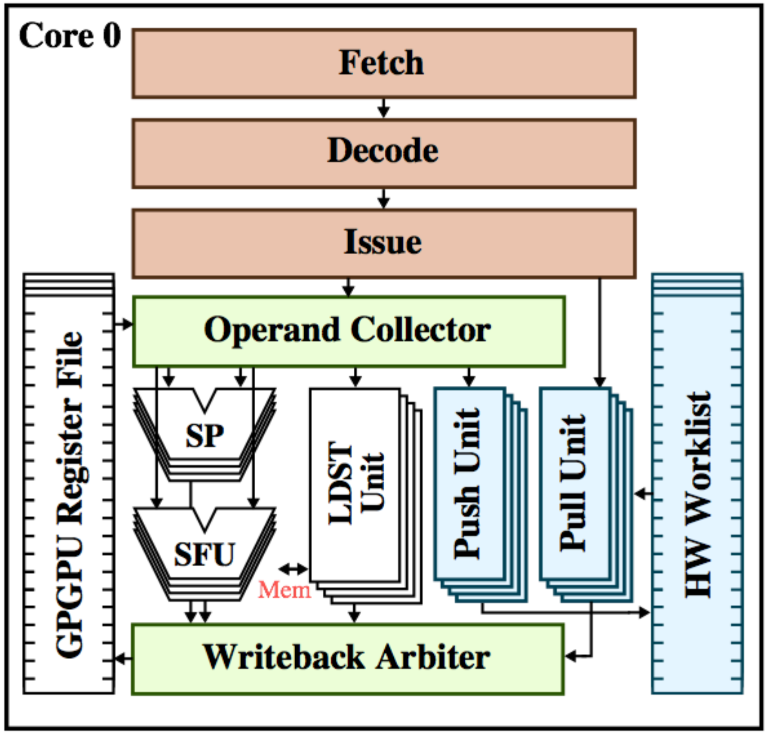
Presentation Outline

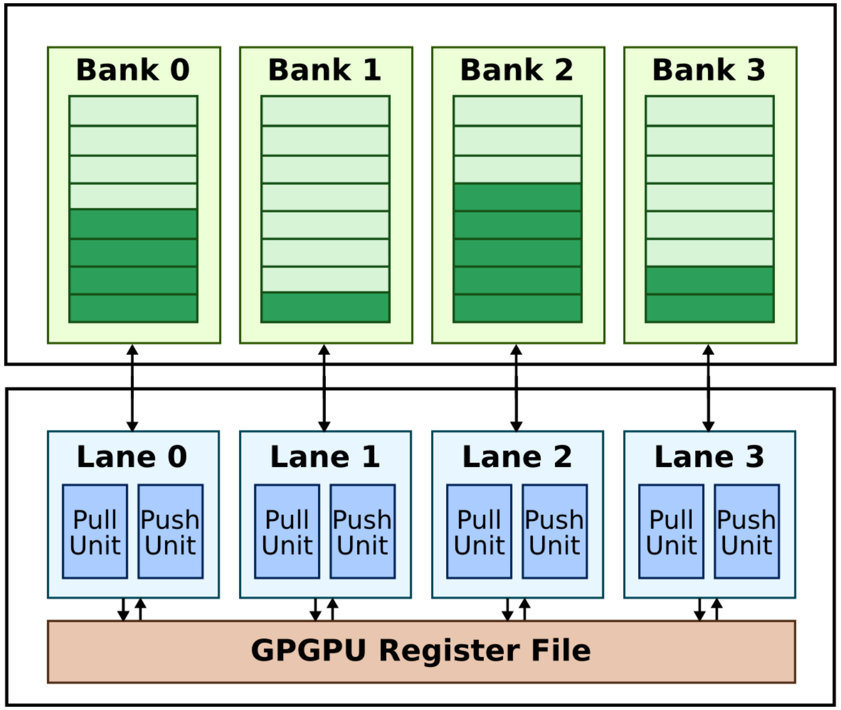
- Motivation
- Mapping Irregular Algorithms to GPGPUs
- Developing Optimized Software Baselines
- **Fine-Grain Hardware Worklists**
- Evaluation



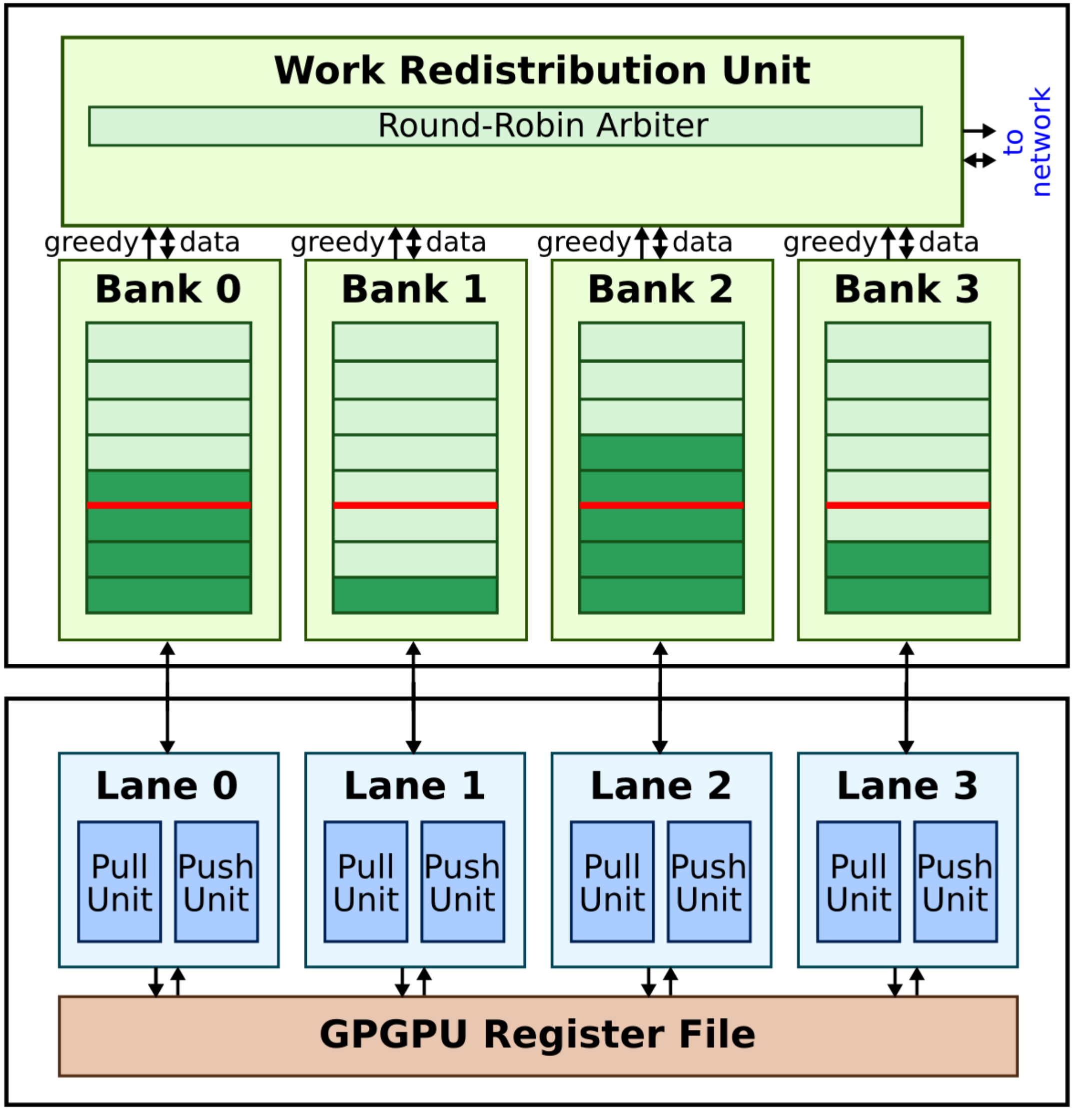
Fine-Grain Hardware Worklist (HWWL) Banks

Instruction	Description
`wlpull`	Pulls work ID from HWWL. If bank is empty: return WAIT if work in other banks, otherwise return DONE.
`wlpush`	Pushes work ID to HWWL, throws exception if overflow buffer is full.



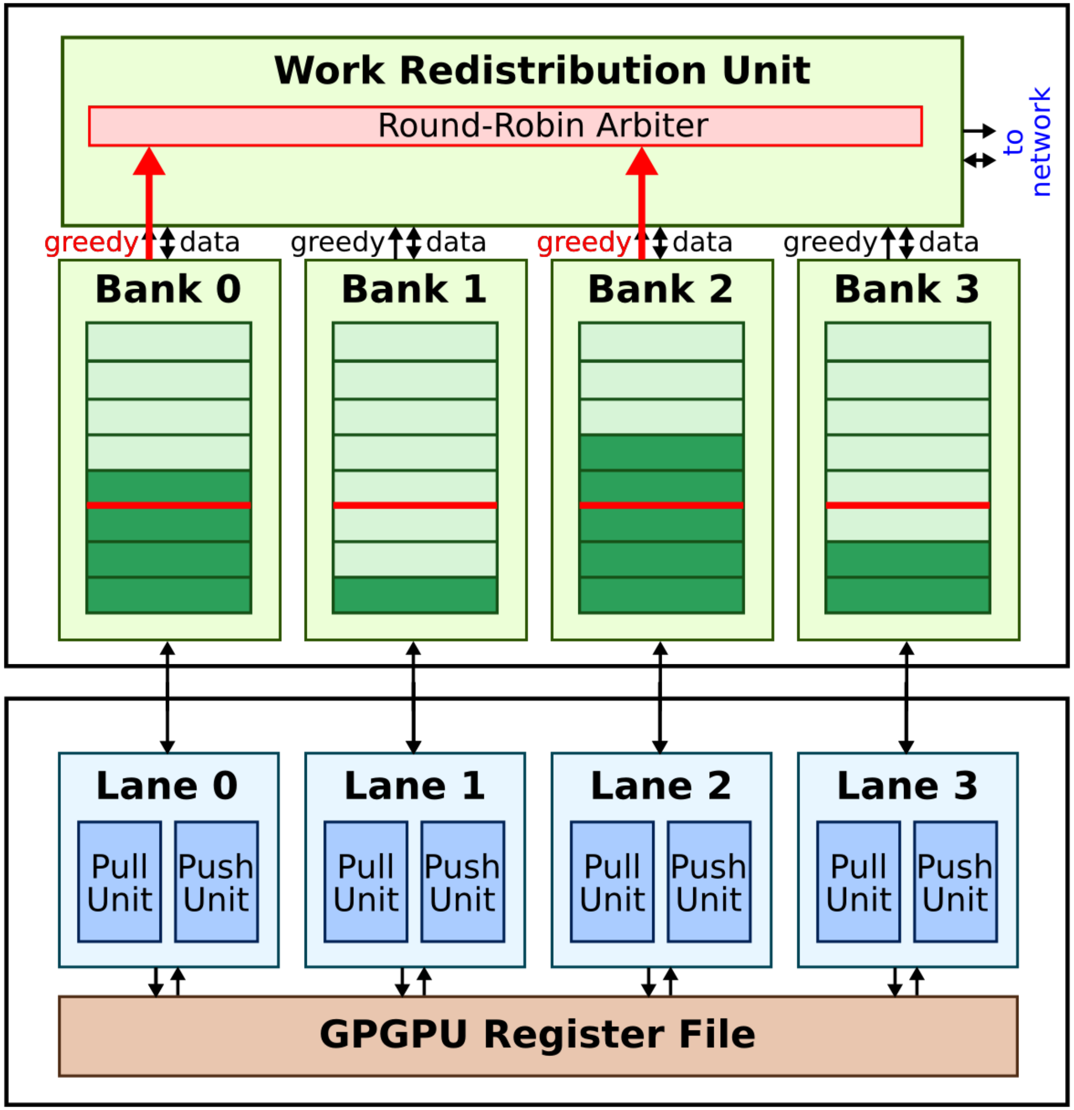


HWWL Intra-Core Work Redistribution (Threshold)



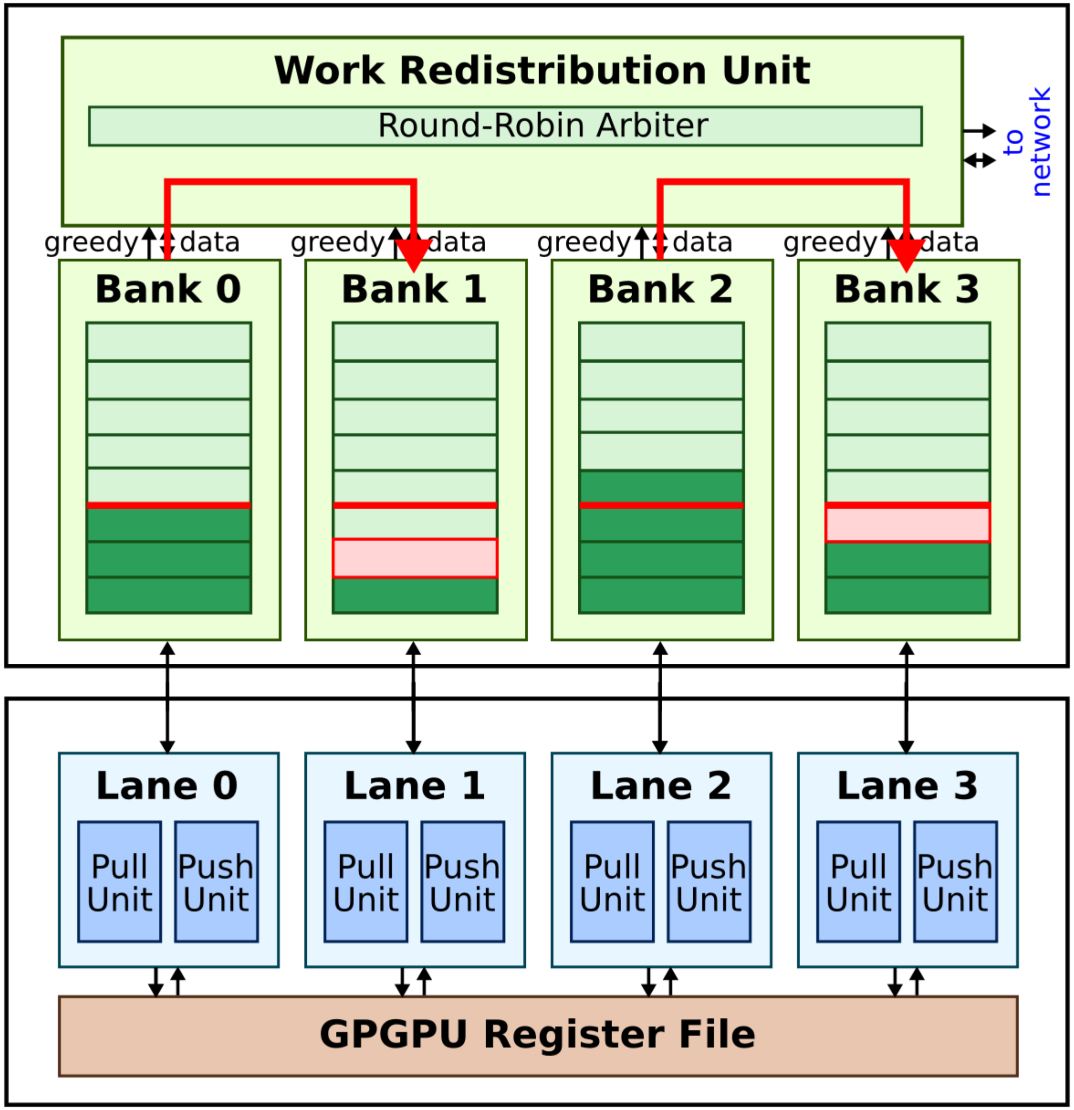
- **Greedy** banks with more work than threshold **donate**
- **Needy** banks with less work than threshold **receive**
- Priority based on round-robin arbitration

HWWL Intra-Core Work Redistribution (Threshold)



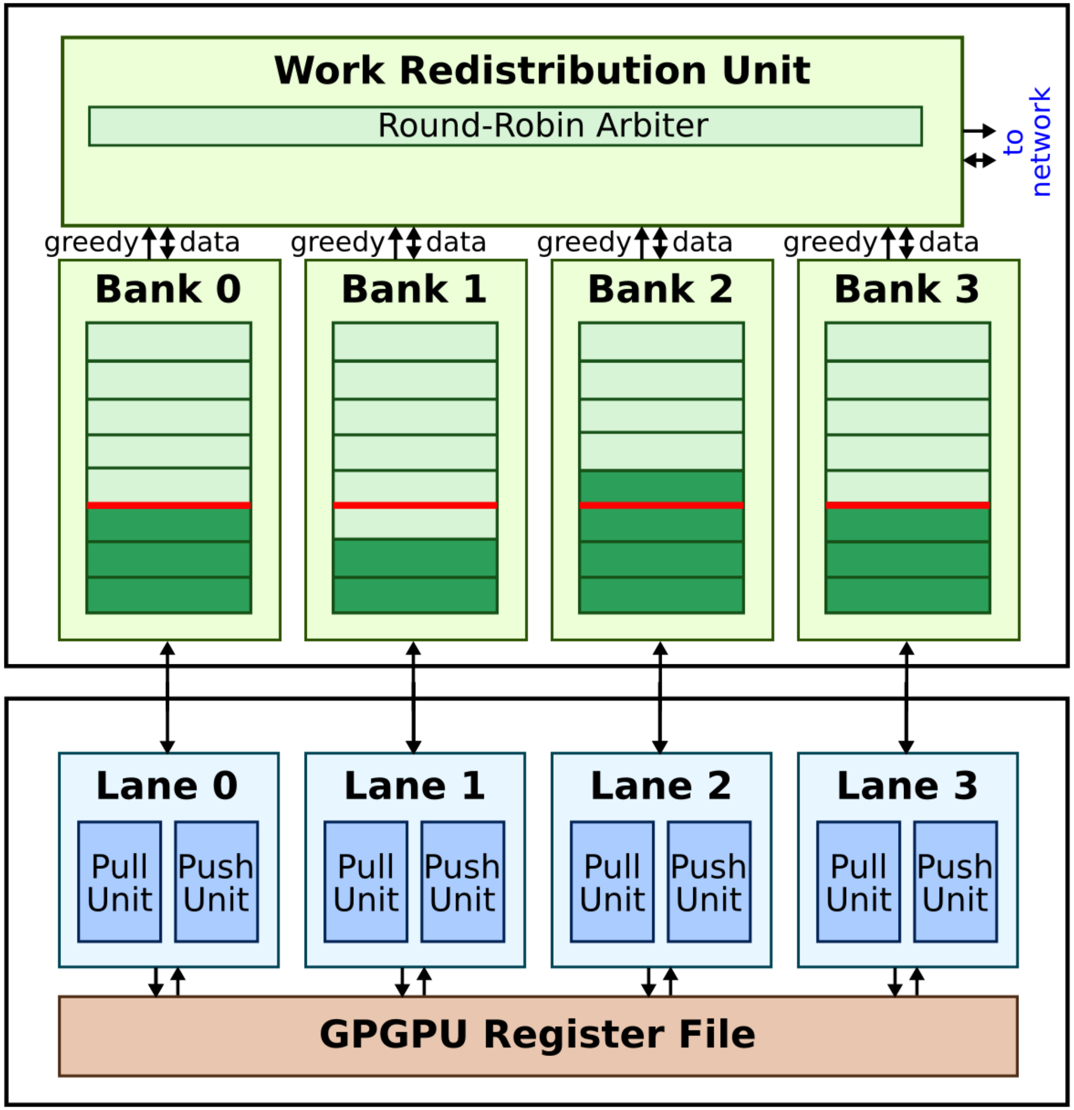
- **Greedy** banks with more work than threshold **donate**
- **Needy** banks with less work than threshold **receive**
- Priority based on round-robin arbitration

HWWL Intra-Core Work Redistribution (Threshold)



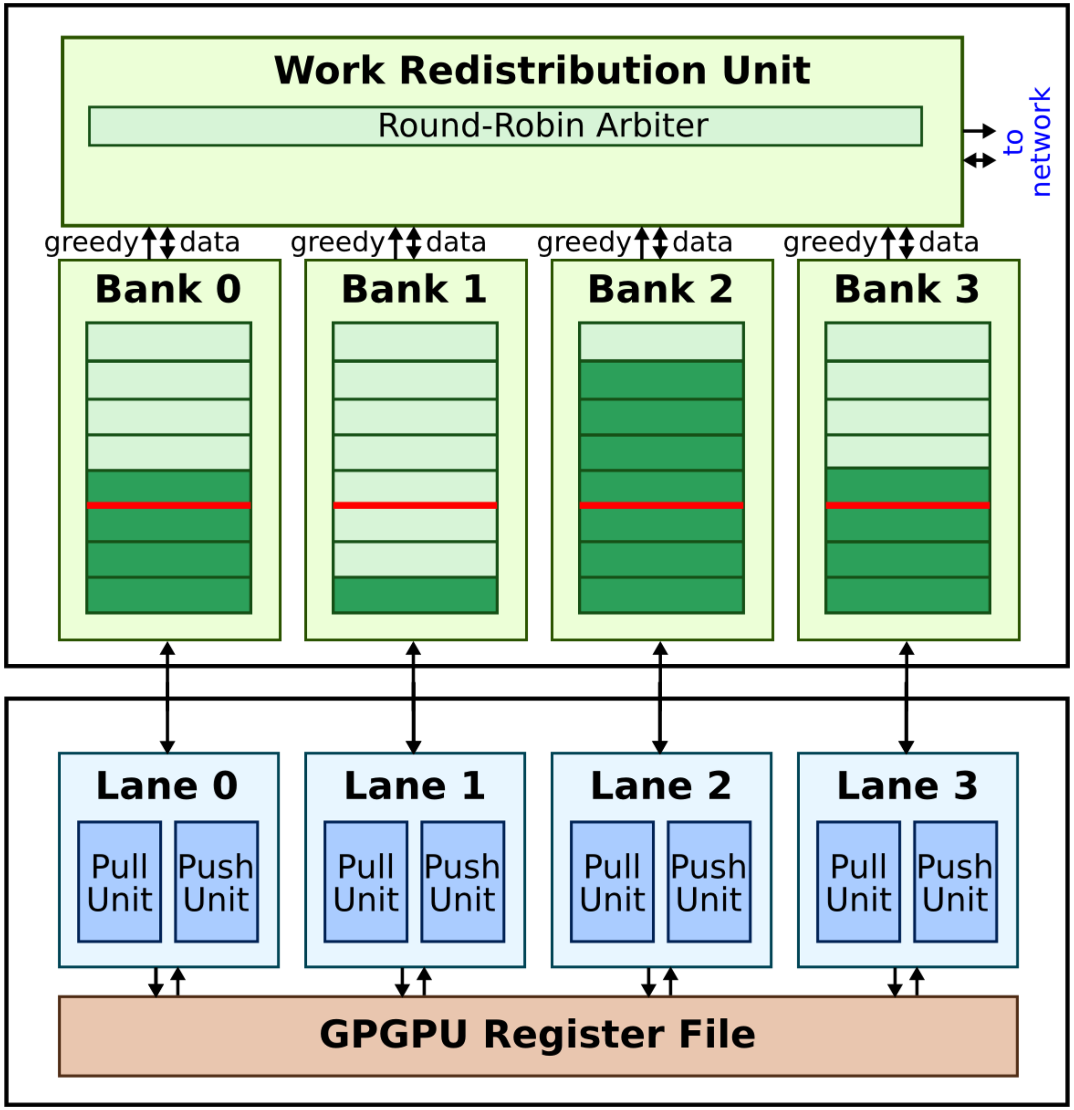
- **Greedy** banks with more work than threshold **donate**
- **Needy** banks with less work than threshold **receive**
- Priority based on round-robin arbitration

HWWL Intra-Core Work Redistribution (Threshold)



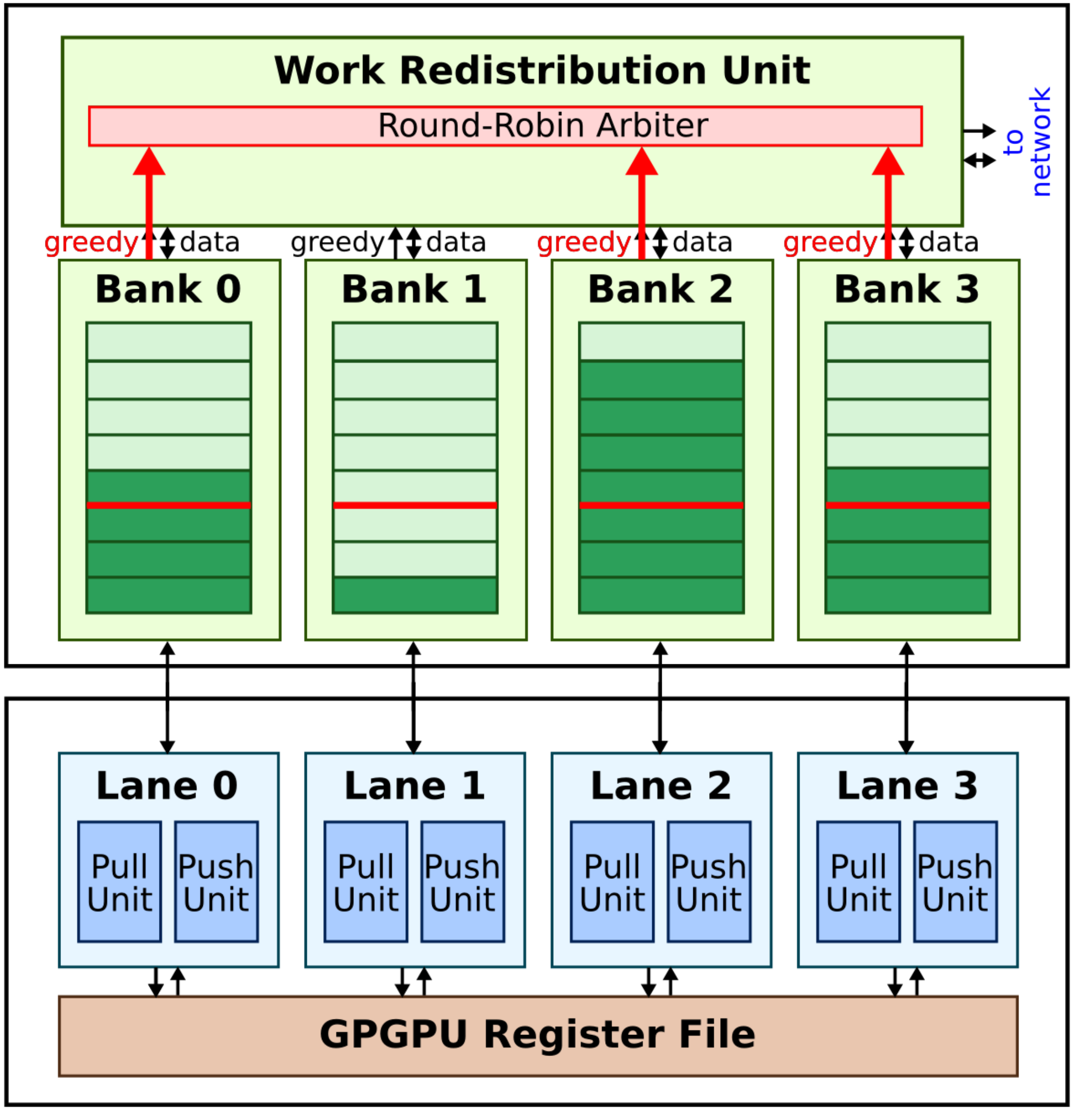
- **Greedy** banks with more work than threshold **donate**
- **Needy** banks with less work than threshold **receive**
- Priority based on round-robin arbitration

HWWL Intra-Core Work Redistribution (Threshold)



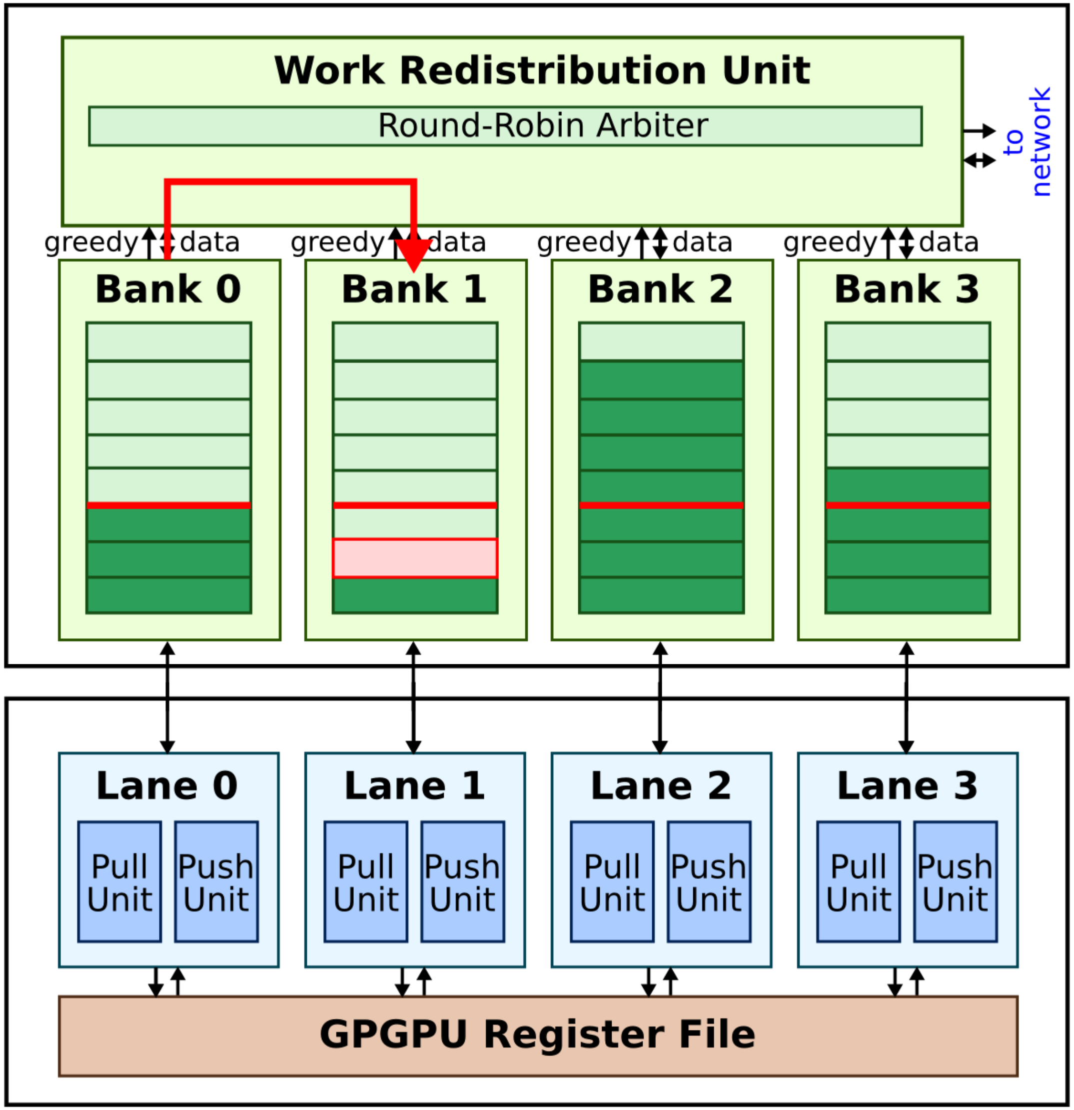
- Simple design, low overhead
- A few banks can monopolize most of the work due to occupancy-agnostic priorities

HWWL Intra-Core Work Redistribution (Threshold)



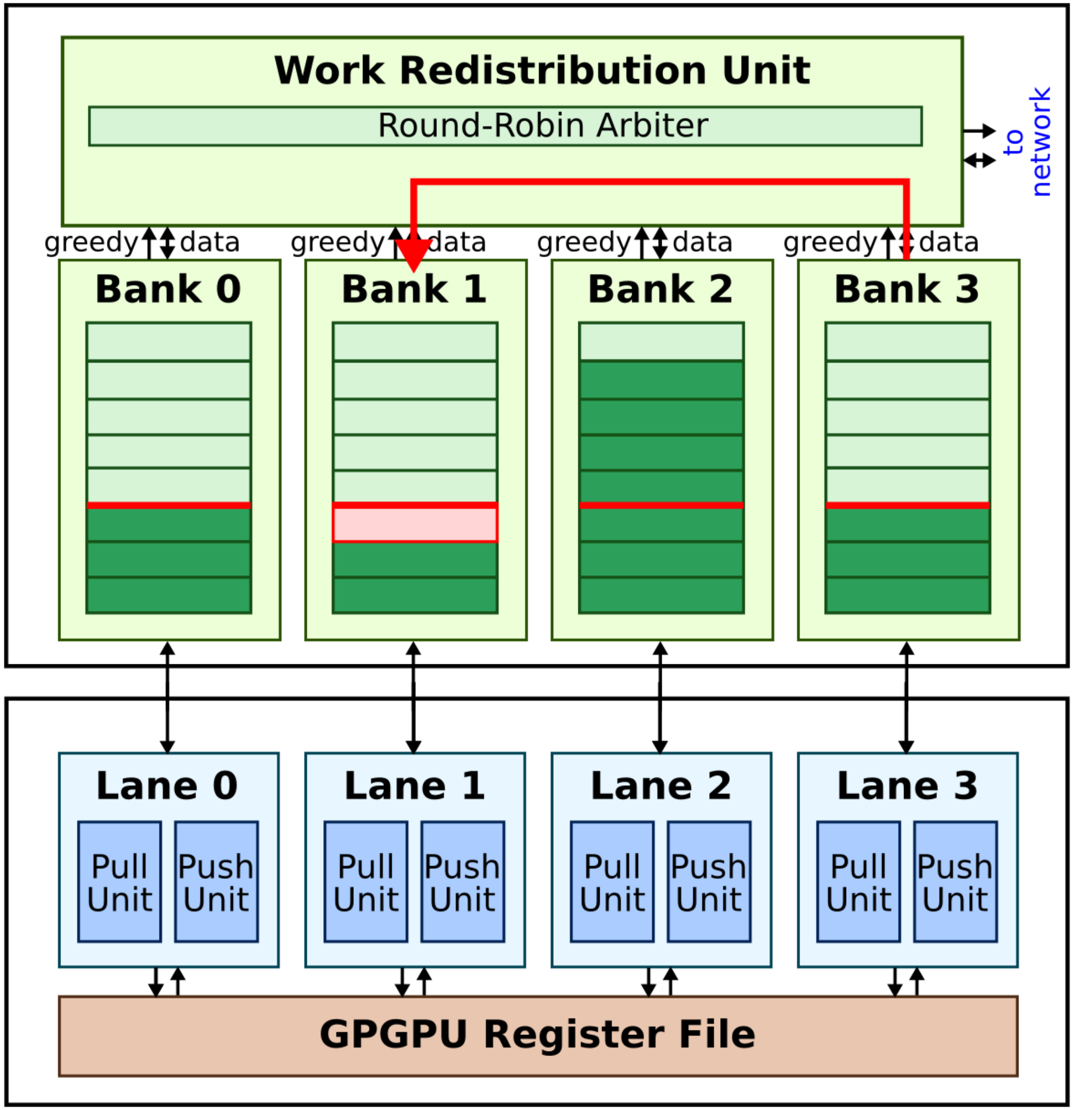
- Simple design, low overhead
- A few banks can monopolize most of the work due to occupancy-agnostic priorities

HWWL Intra-Core Work Redistribution (Threshold)



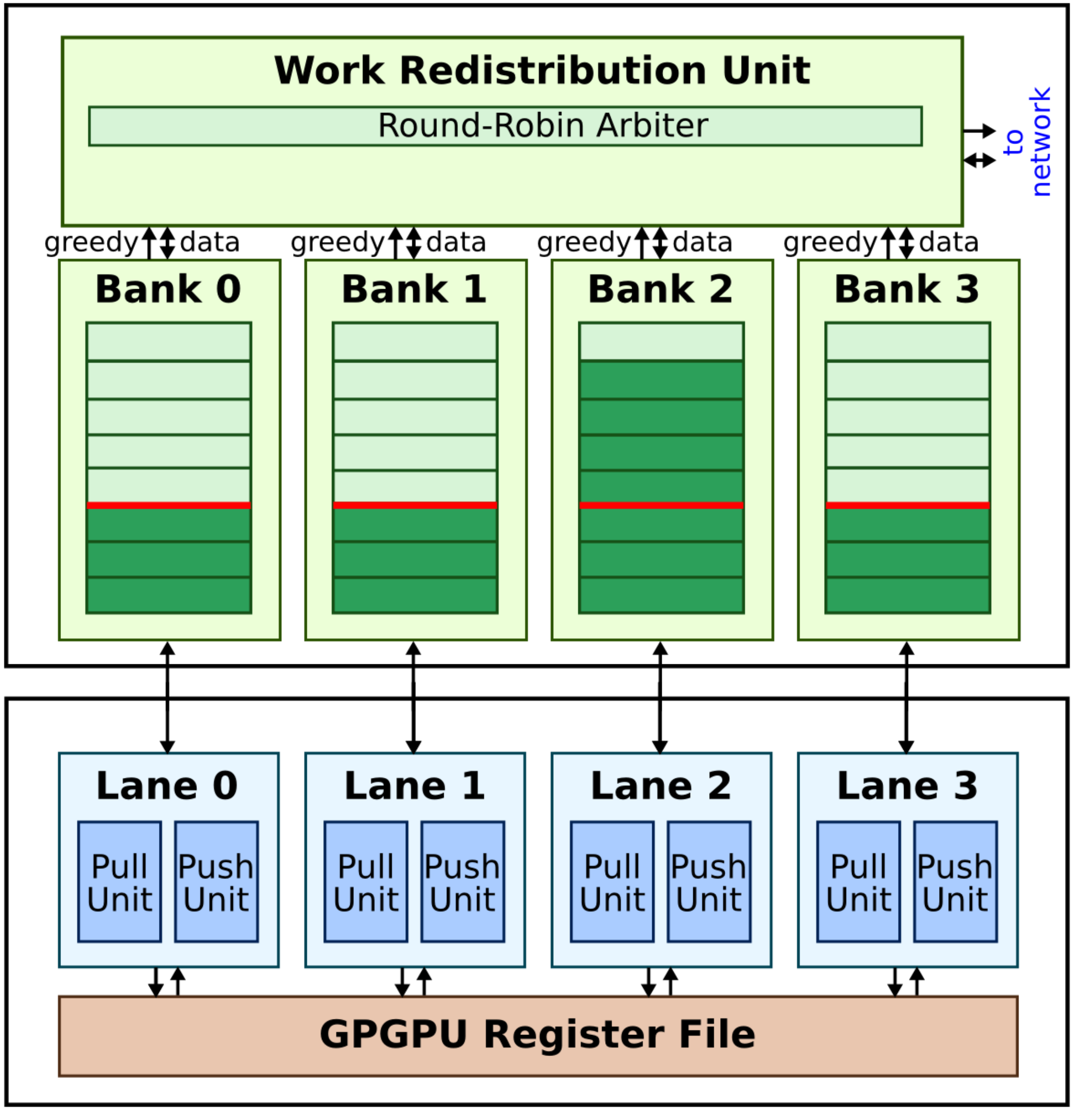
- Simple design, low overhead
- A few banks can monopolize most of the work due to occupancy-agnostic priorities

HWWL Intra-Core Work Redistribution (Threshold)



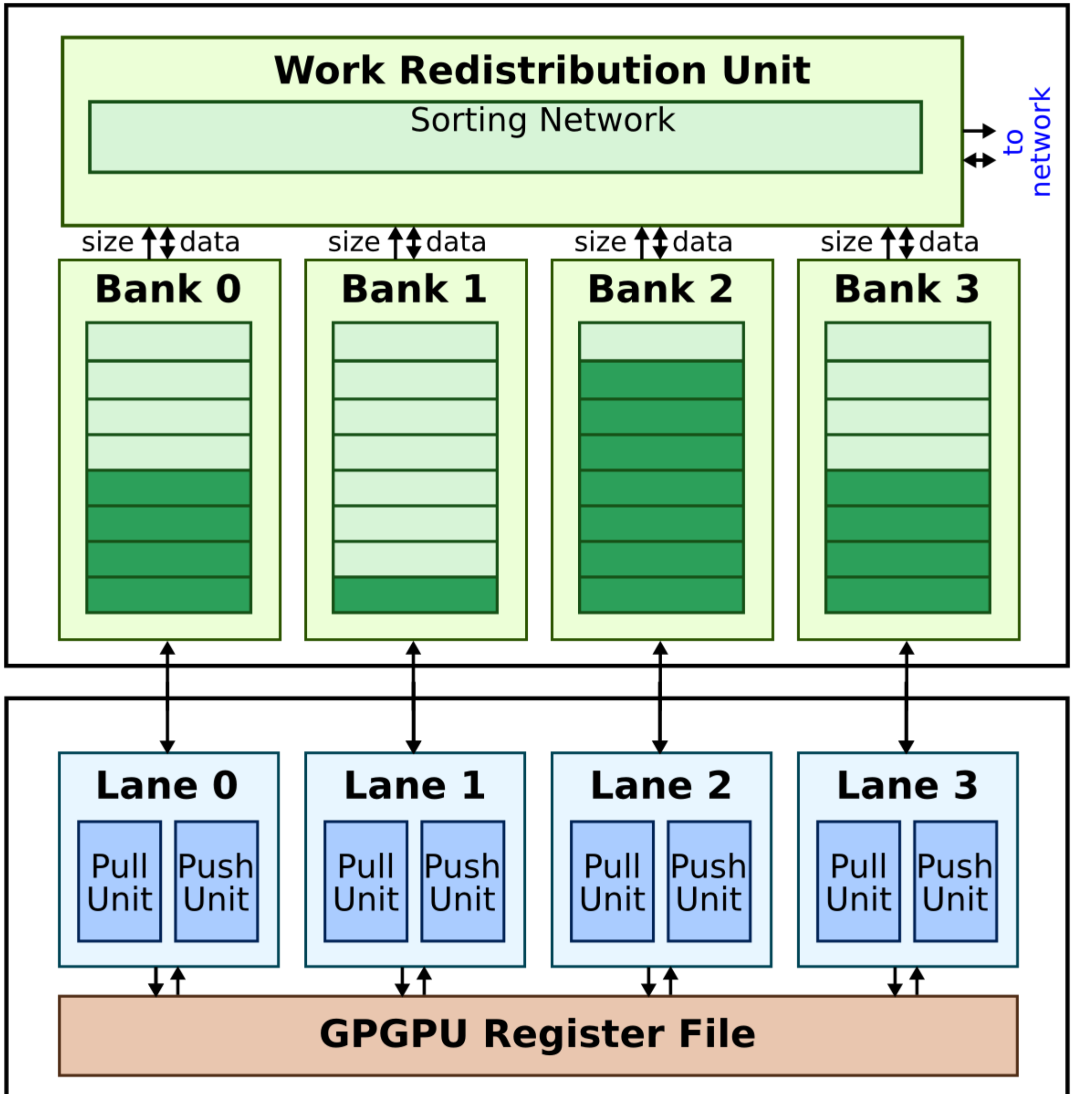
- Simple design, low overhead
- A few banks can monopolize most of the work due to occupancy-agnostic priorities

HWWL Intra-Core Work Redistribution (Threshold)



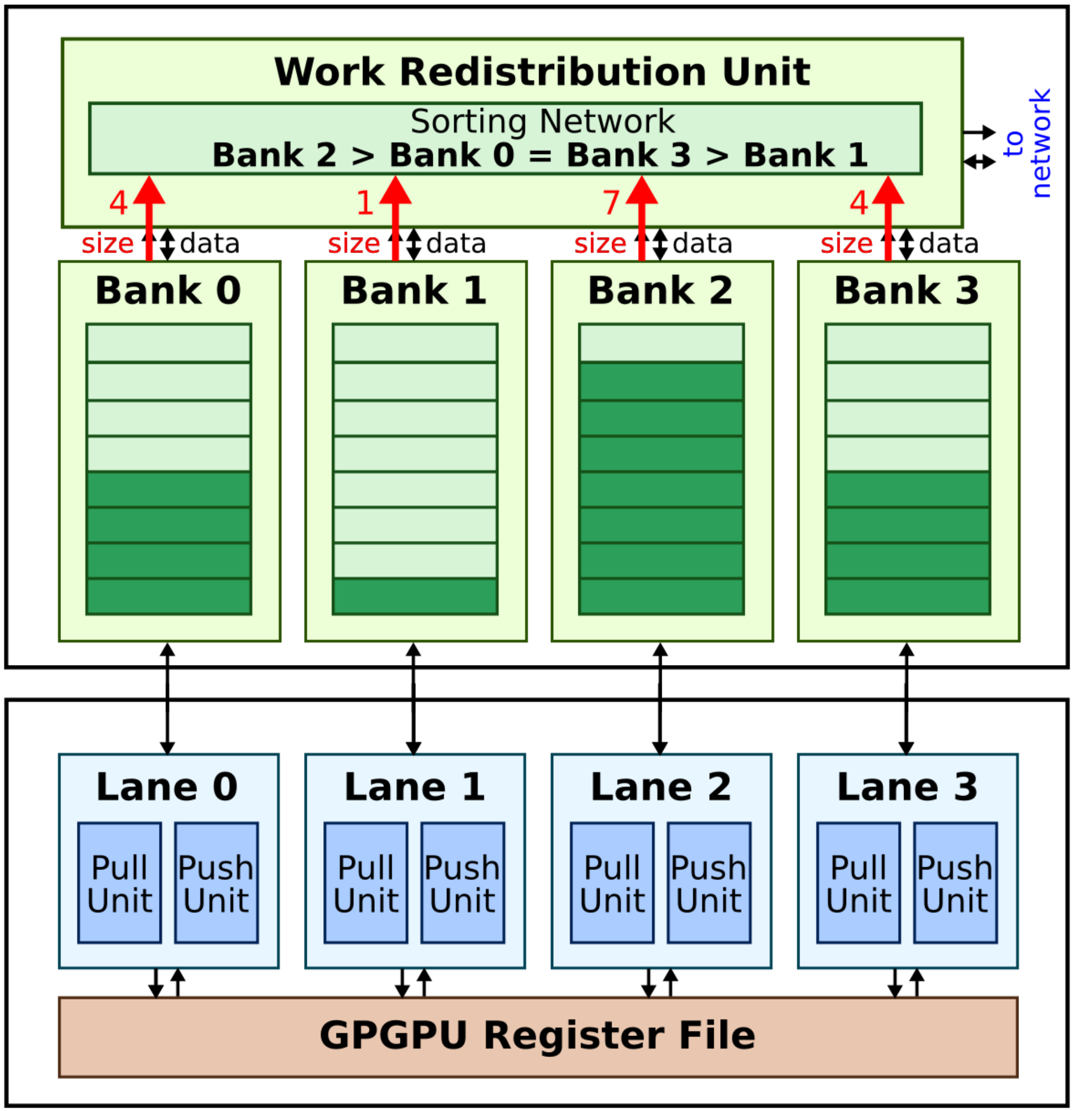
- Simple design, low overhead
- A few banks can monopolize most of the work due to occupancy-agnostic priorities

HWWL Intra-Core Work Redistribution (Sorting)



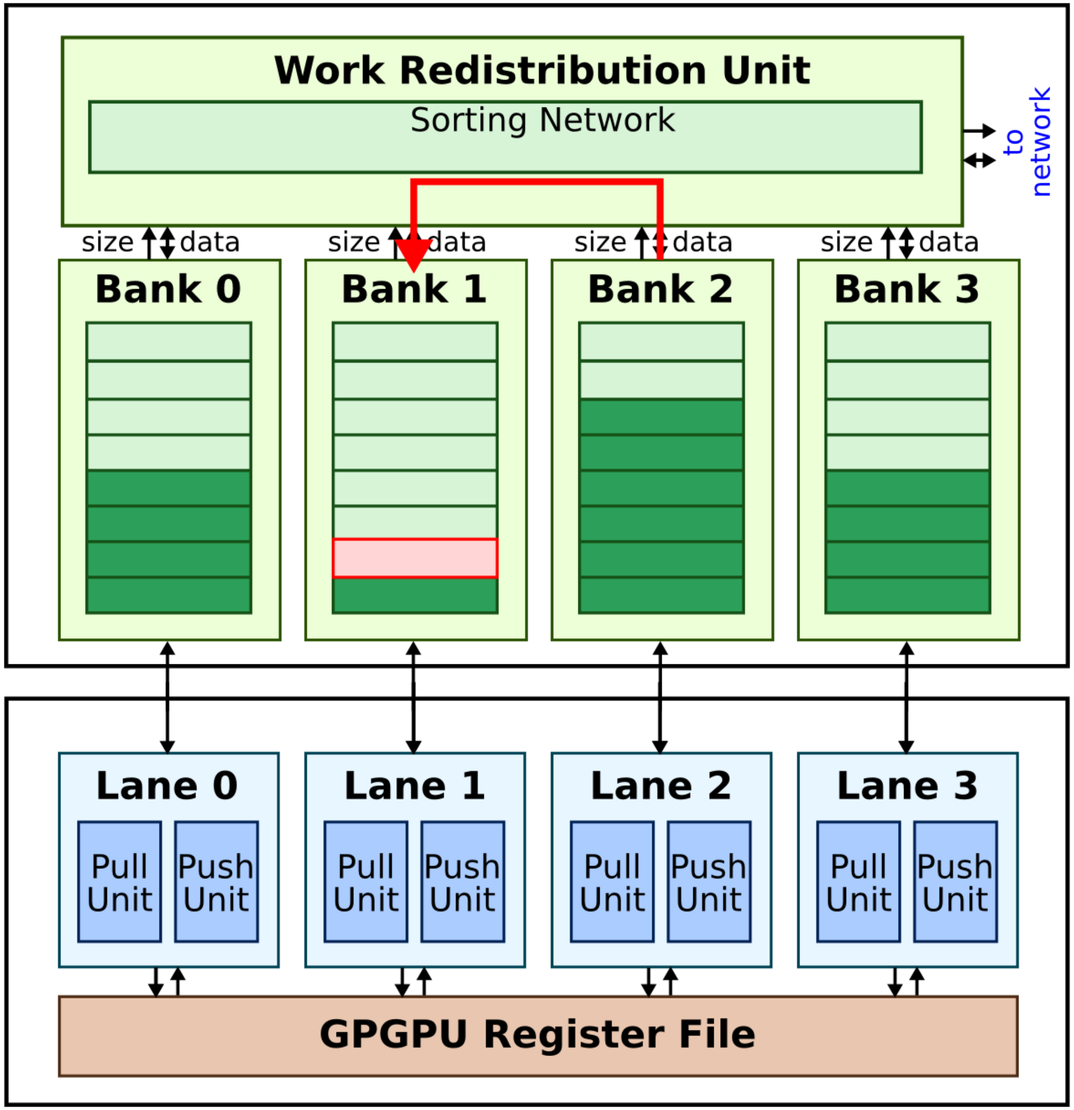
- Tradeoff complexity for better load balancing
- Sort banks based on amount of work
- Banks with most work donate to banks with least work

HWWL Intra-Core Work Redistribution (Sorting)



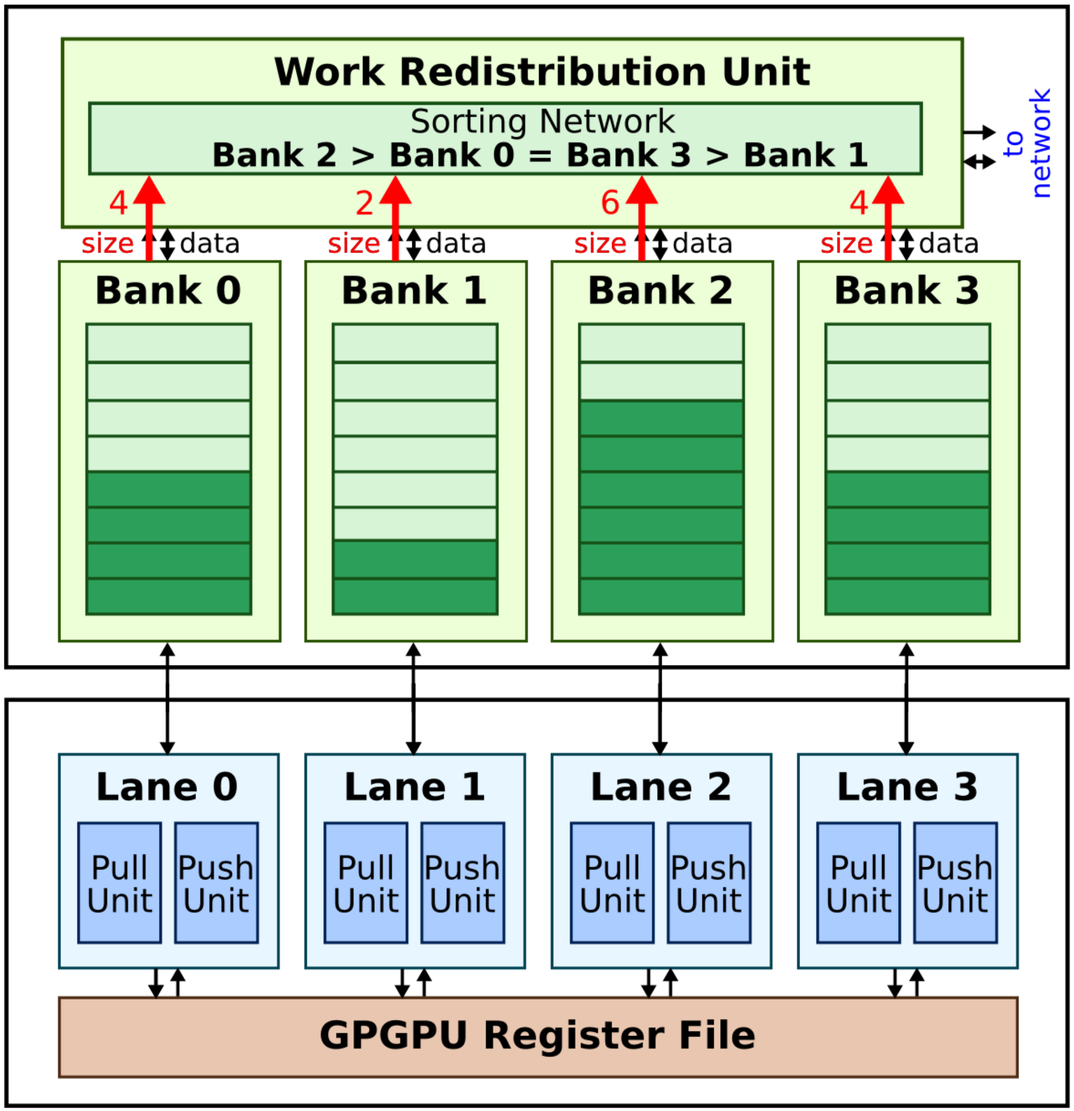
- Tradeoff complexity for better load balancing
- Sort banks based on amount of work
- Banks with most work donate to banks with least work

HWWL Intra-Core Work Redistribution (Sorting)



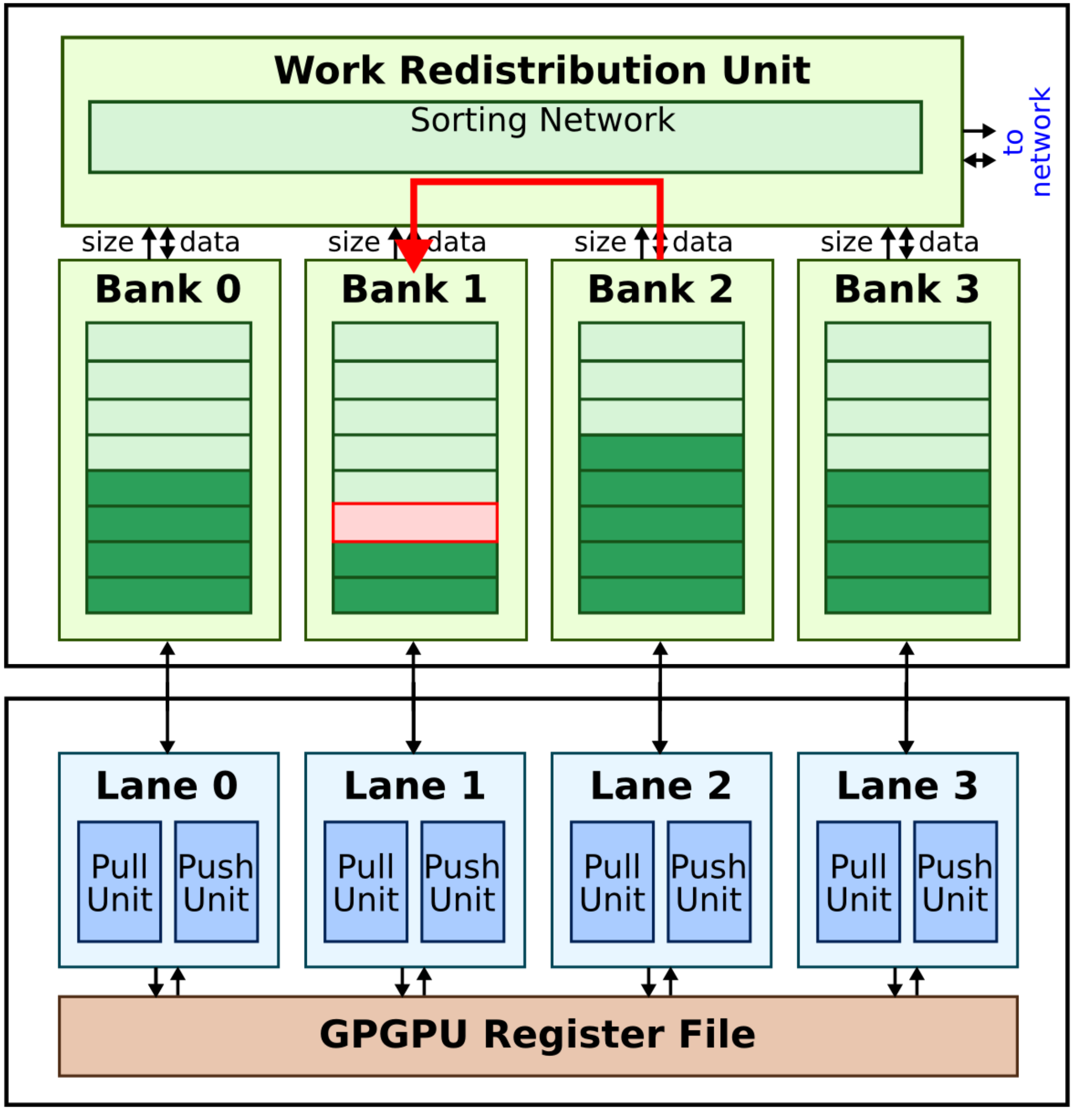
- Tradeoff complexity for better load balancing
- Sort banks based on amount of work
- Banks with most work donate to banks with least work

HWWL Intra-Core Work Redistribution (Sorting)



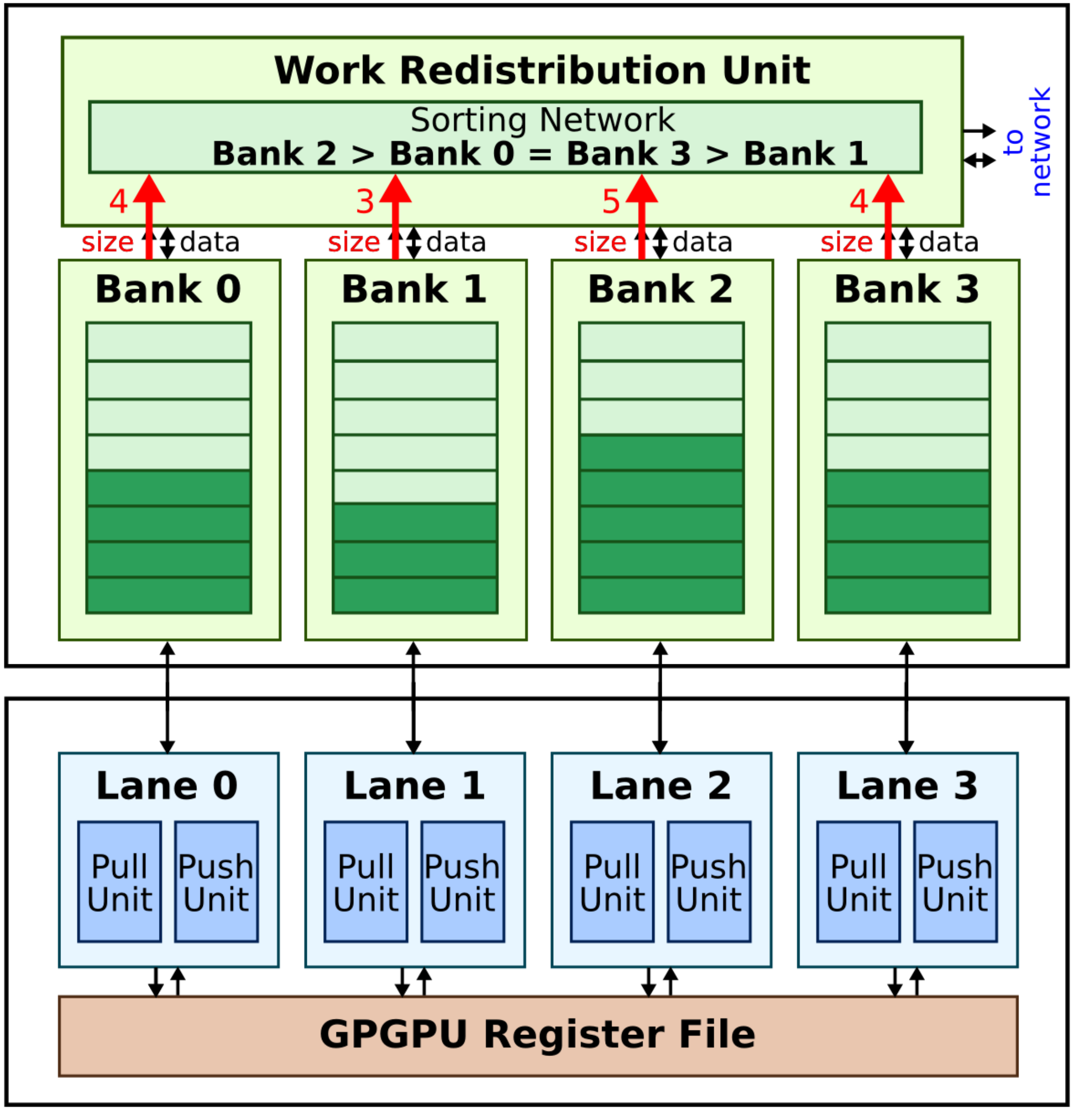
- Tradeoff complexity for better load balancing
- Sort banks based on amount of work
- Banks with most work donate to banks with least work

HWWL Intra-Core Work Redistribution (Sorting)



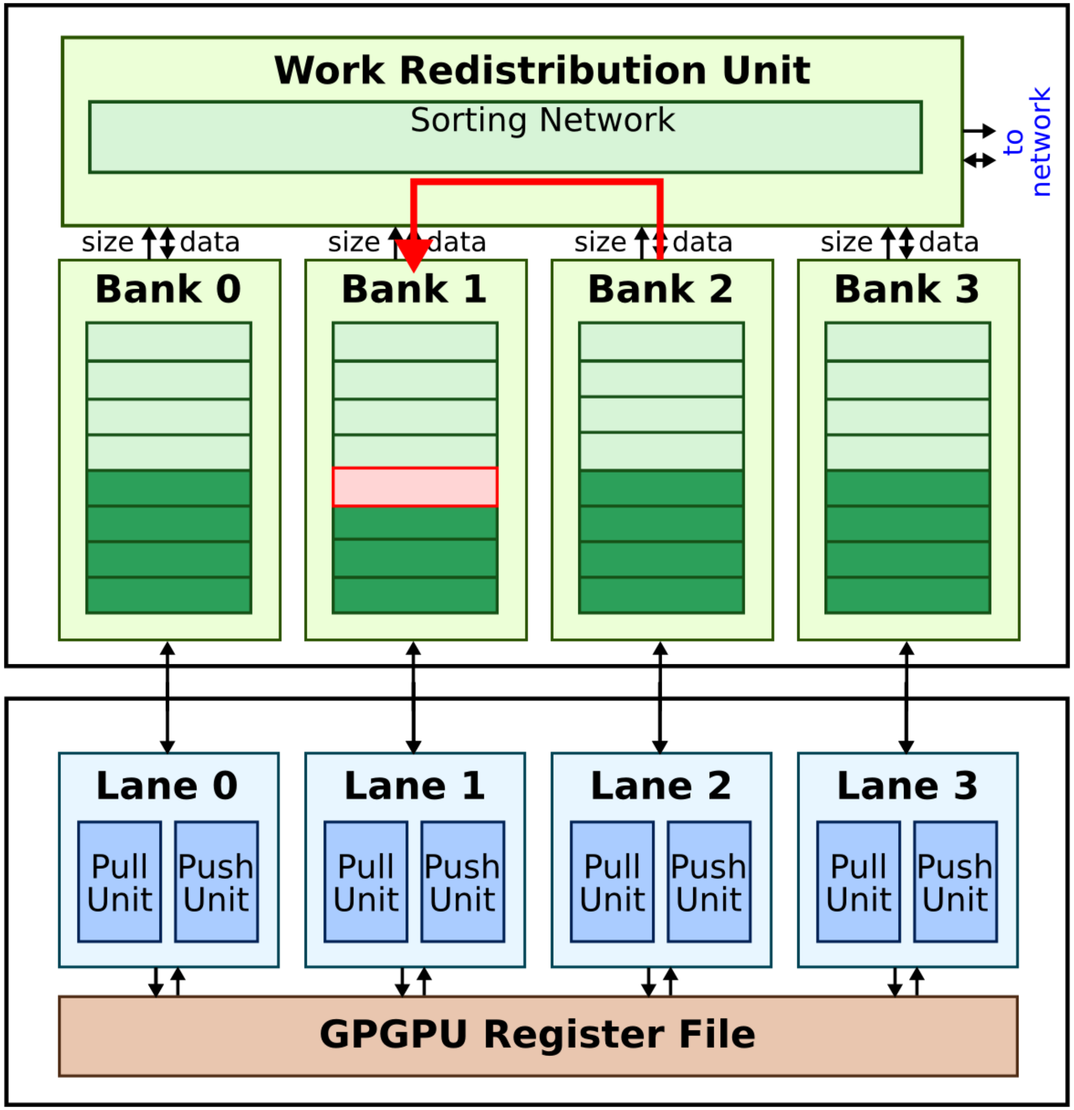
- Tradeoff complexity for better load balancing
- Sort banks based on amount of work
- Banks with most work donate to banks with least work

HWWL Intra-Core Work Redistribution (Sorting)



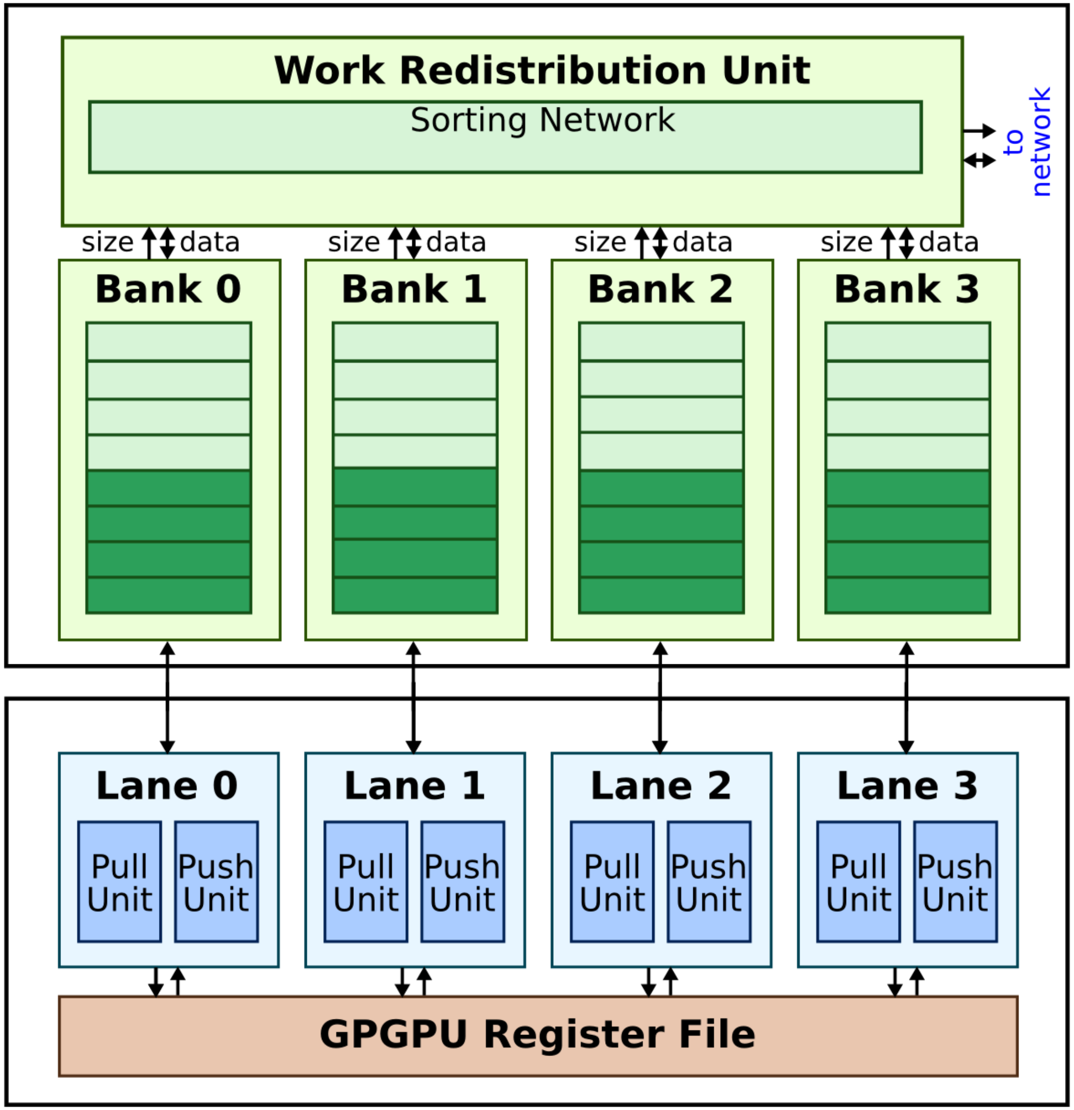
- Tradeoff complexity for better load balancing
- Sort banks based on amount of work
- Banks with most work donate to banks with least work

HWWL Intra-Core Work Redistribution (Sorting)



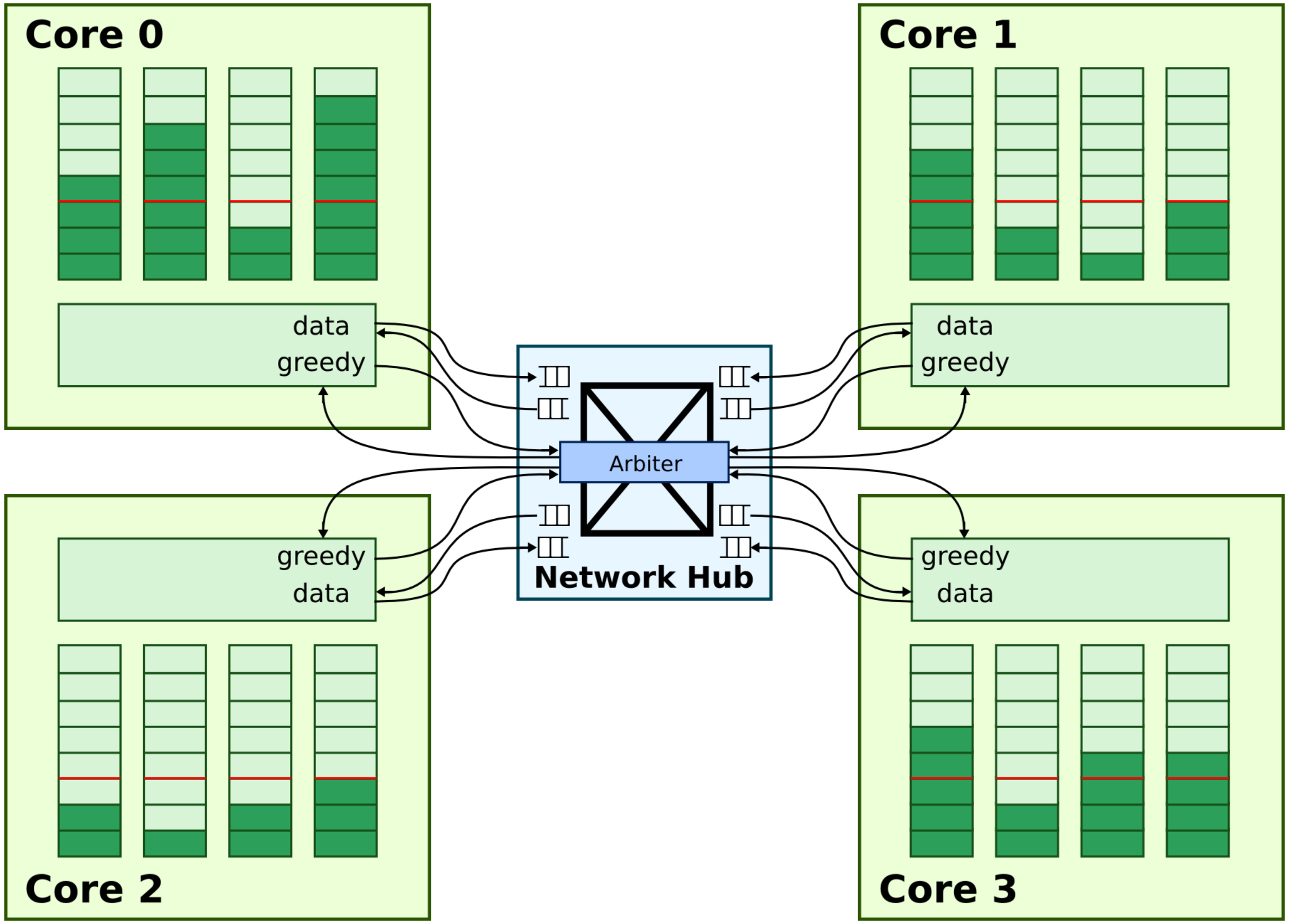
- Tradeoff complexity for better load balancing
- Sort banks based on amount of work
- Banks with most work donate to banks with least work

HWWL Intra-Core Work Redistribution (Sorting)



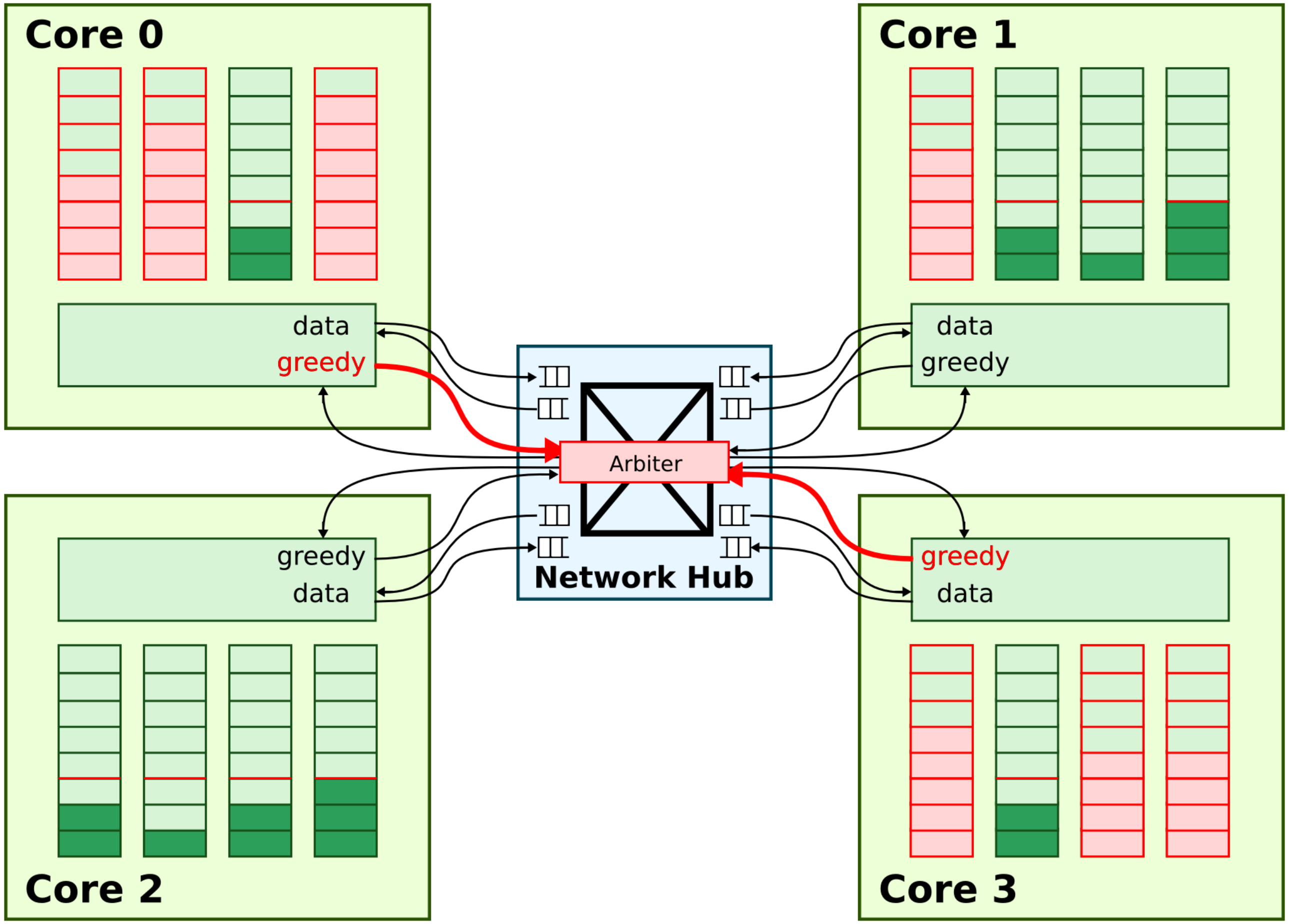
- Tradeoff complexity for better load balancing
- Sort banks based on amount of work
- Banks with most work donate to banks with least work

HWWL Inter-Core Work Redistribution



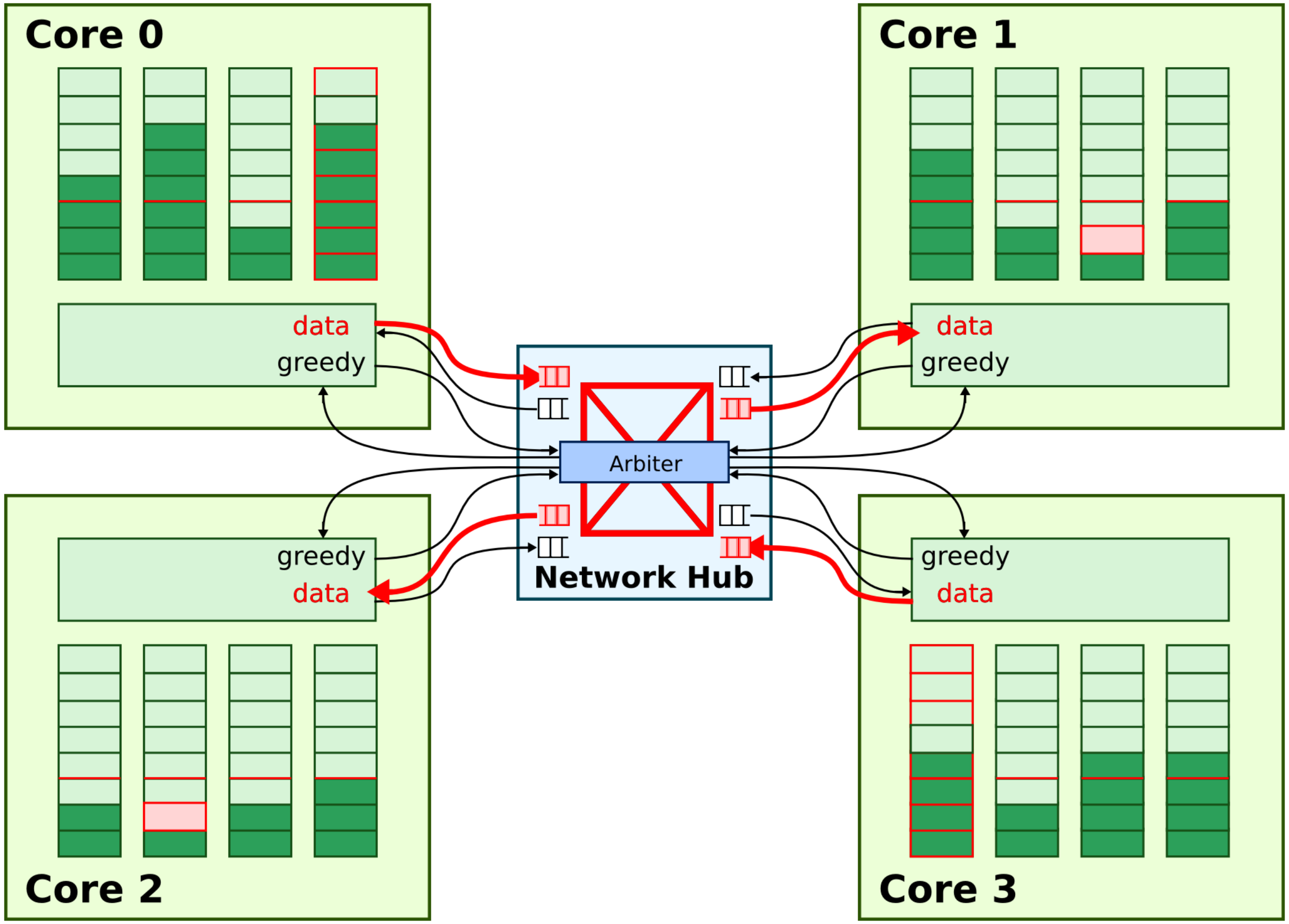
- Inter-core redistribution network with tree topology
  - 2 hops to any destination

HWWL Inter-Core Work Redistribution



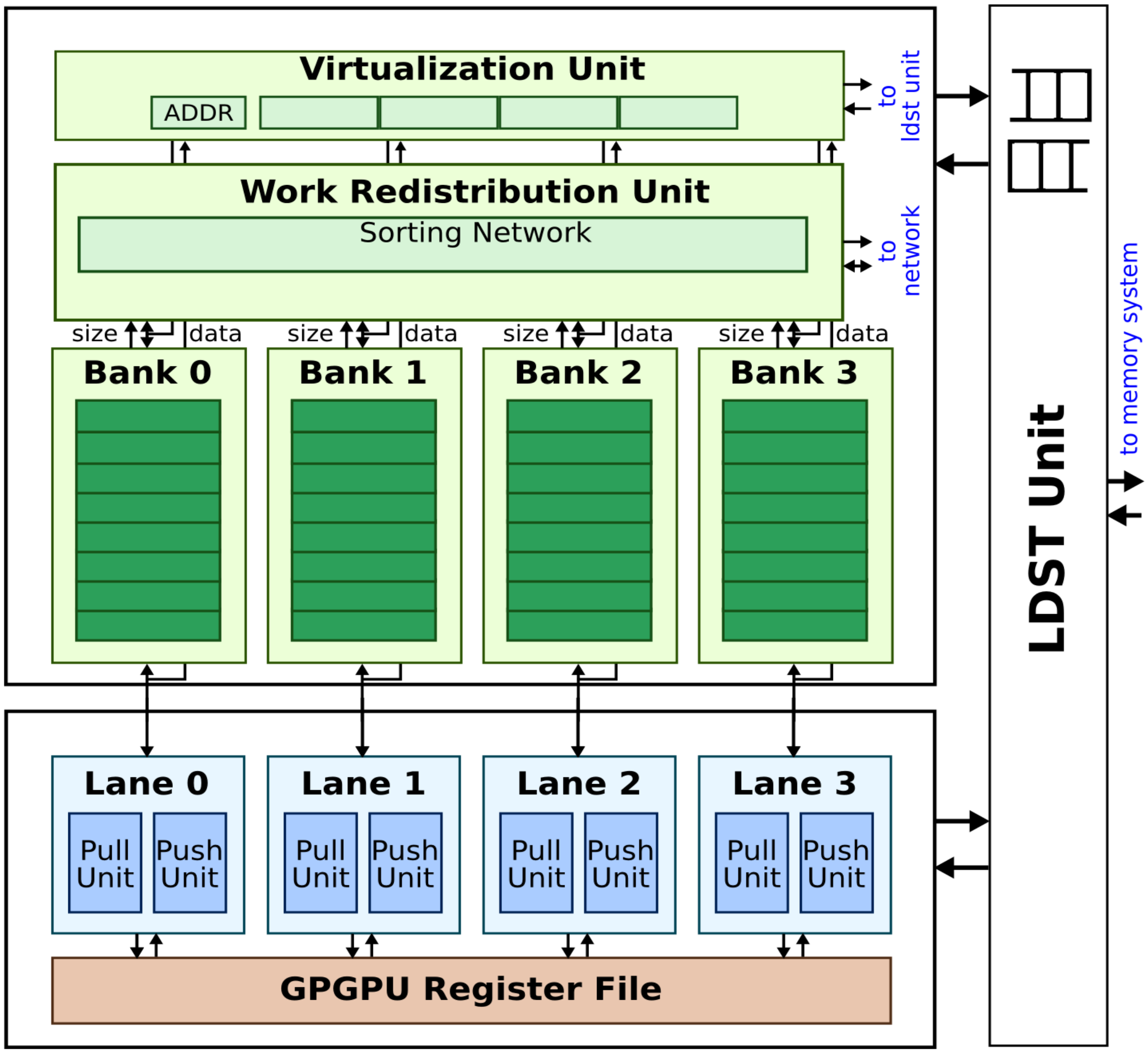
- **Donate** if # greedy banks > # needy banks

HWWL Inter-Core Work Redistribution



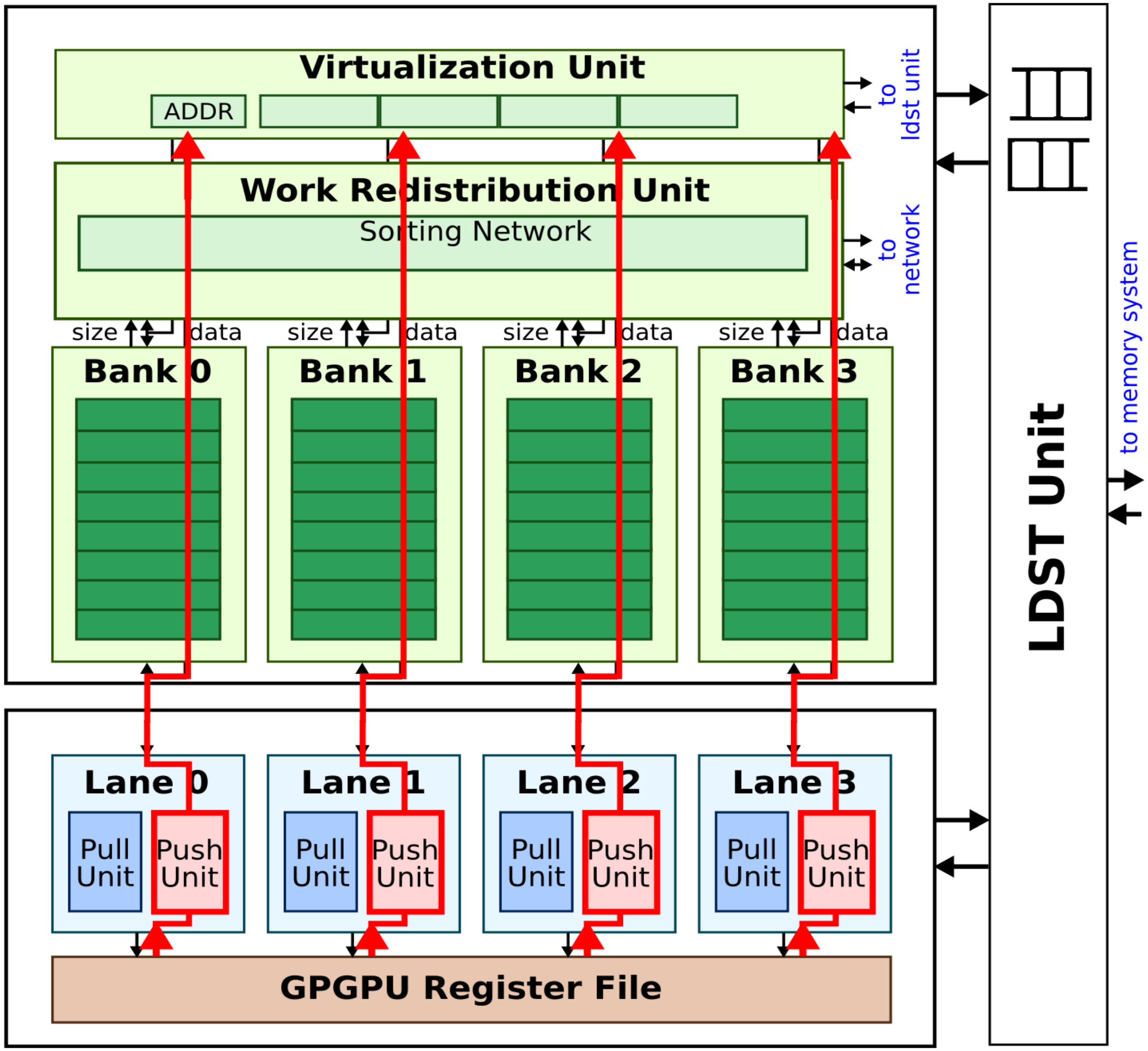
- Also explored monolithic sorting network (global information)

HWWL Work Spilling



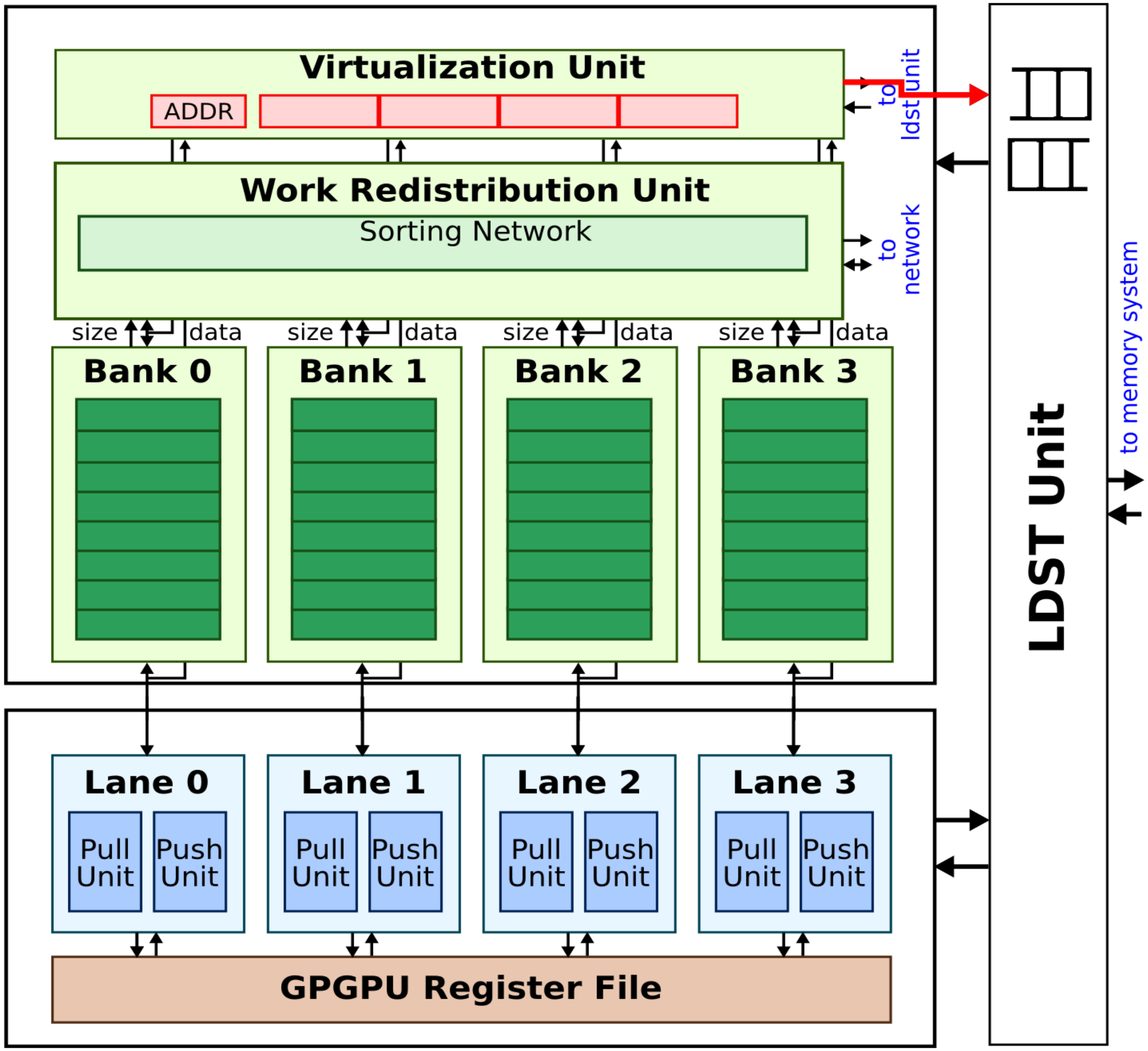
- Virtualization unit manages per-core **overflow buffer**
- If banks are full on a push, inject spill request to load-store queue
- Guaranteed coalescing for spill requests

HWWL Work Spilling



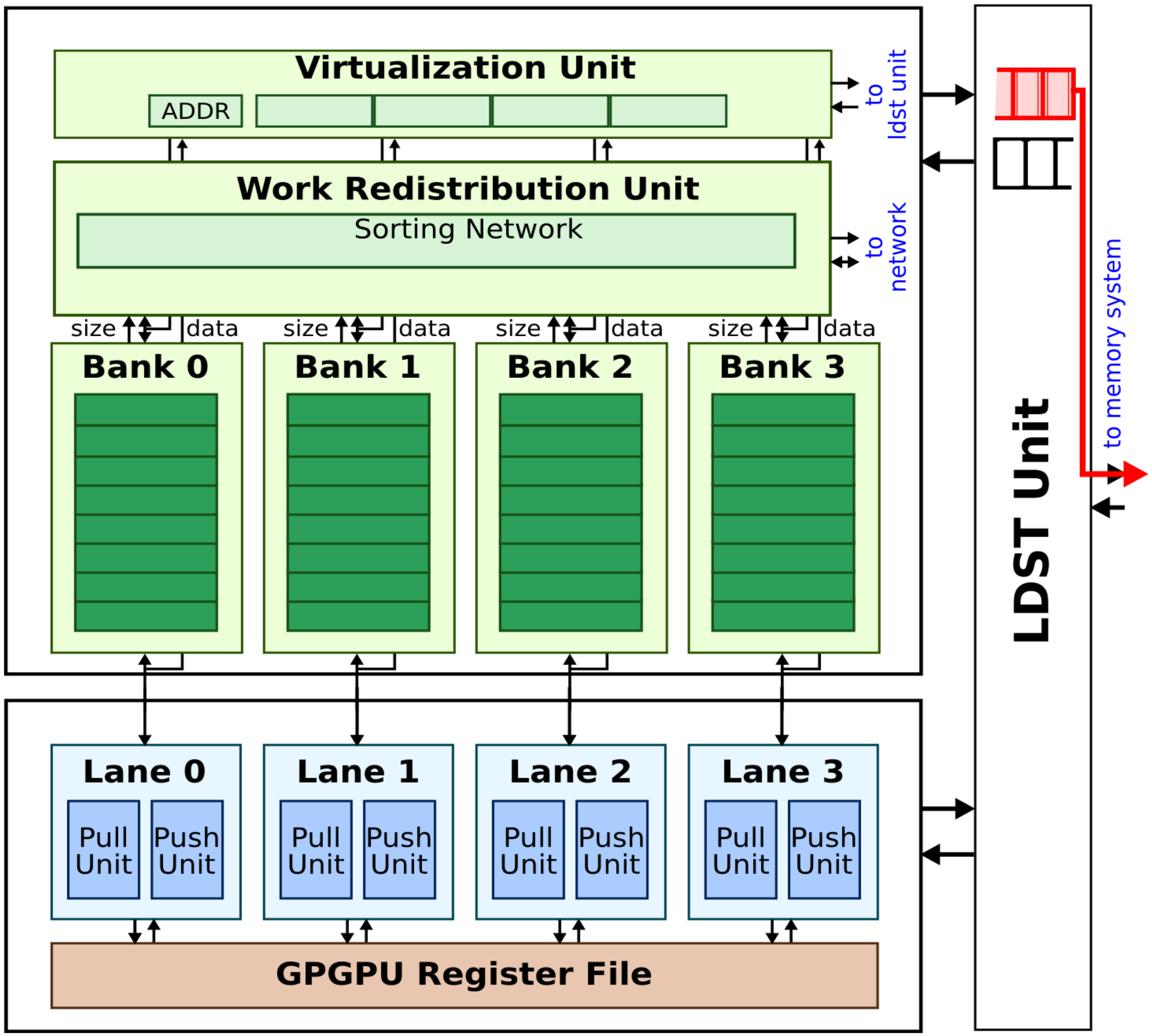
- Virtualization unit manages per-core **overflow buffer**
- If banks are full on a push, inject spill request to load-store queue
- Guaranteed coalescing for spill requests

HWWL Work Spilling



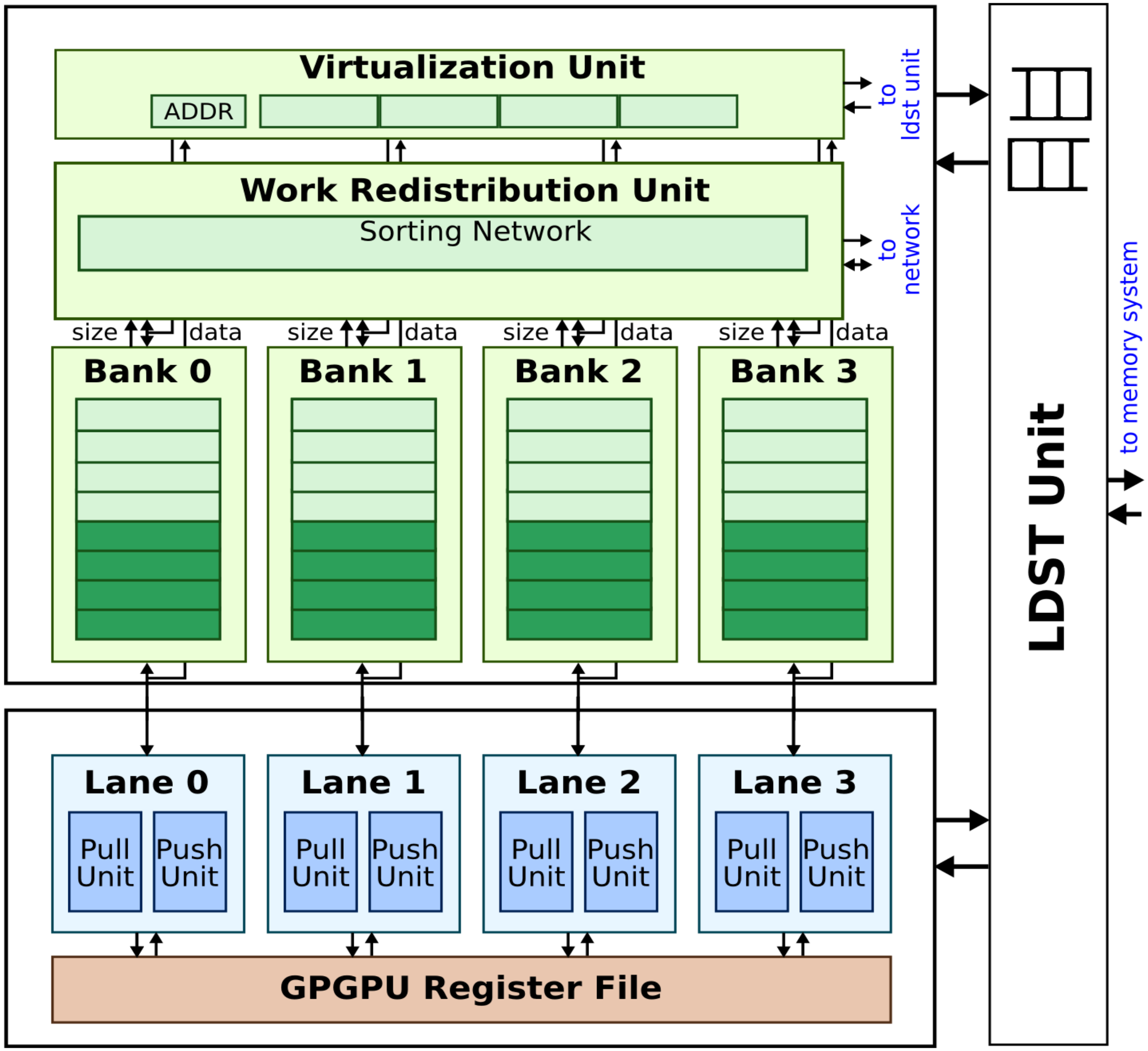
- Virtualization unit manages per-core **overflow buffer**
- If banks are full on a push, inject spill request to load-store queue
- Guaranteed coalescing for spill requests

HWWL Work Spilling



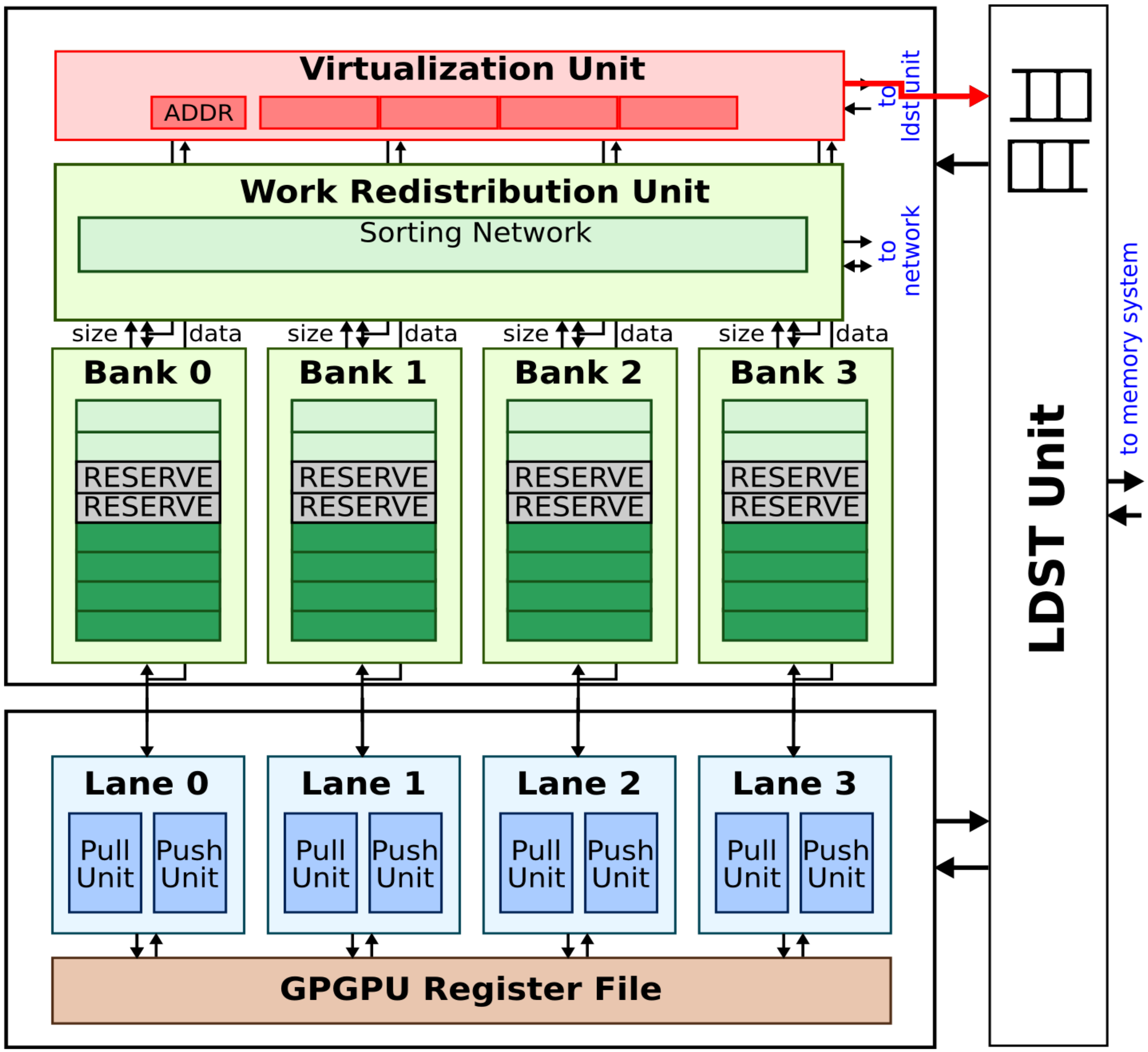
- Virtualization unit manages per-core **overflow buffer**
- If banks are full on a push, inject spill request to load-store queue
- Guaranteed coalescing for spill requests

HWWL Work Refilling (Interval-Based)



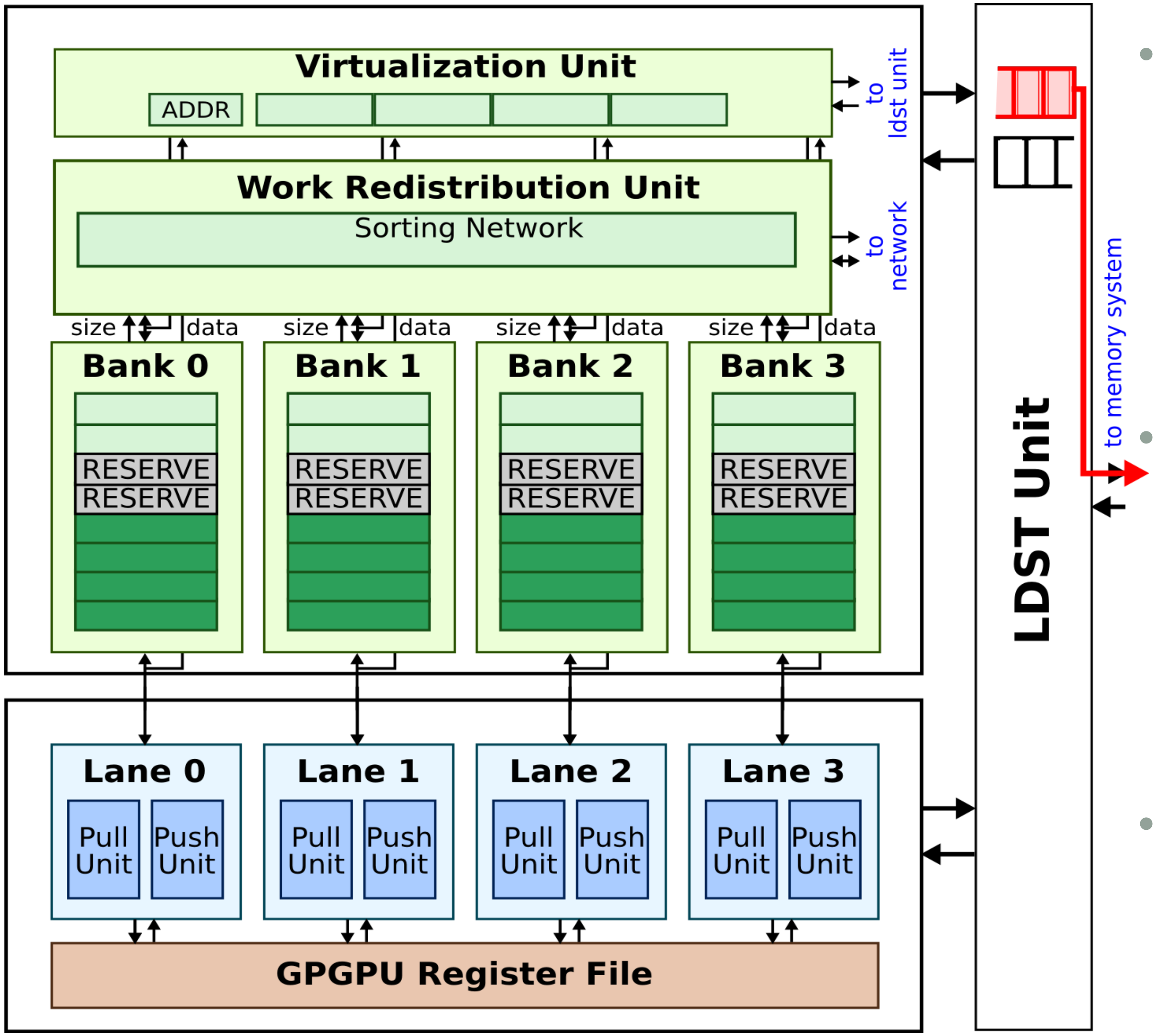
- Periodically check if banks are not full and work is in overflow buffer
- Reserve entries and inject refill request into load-store queue (1-bit to mark as refill)
- Refill responses are routed to virtualization unit for writeback

HWWL Work Refilling (Interval-Based)



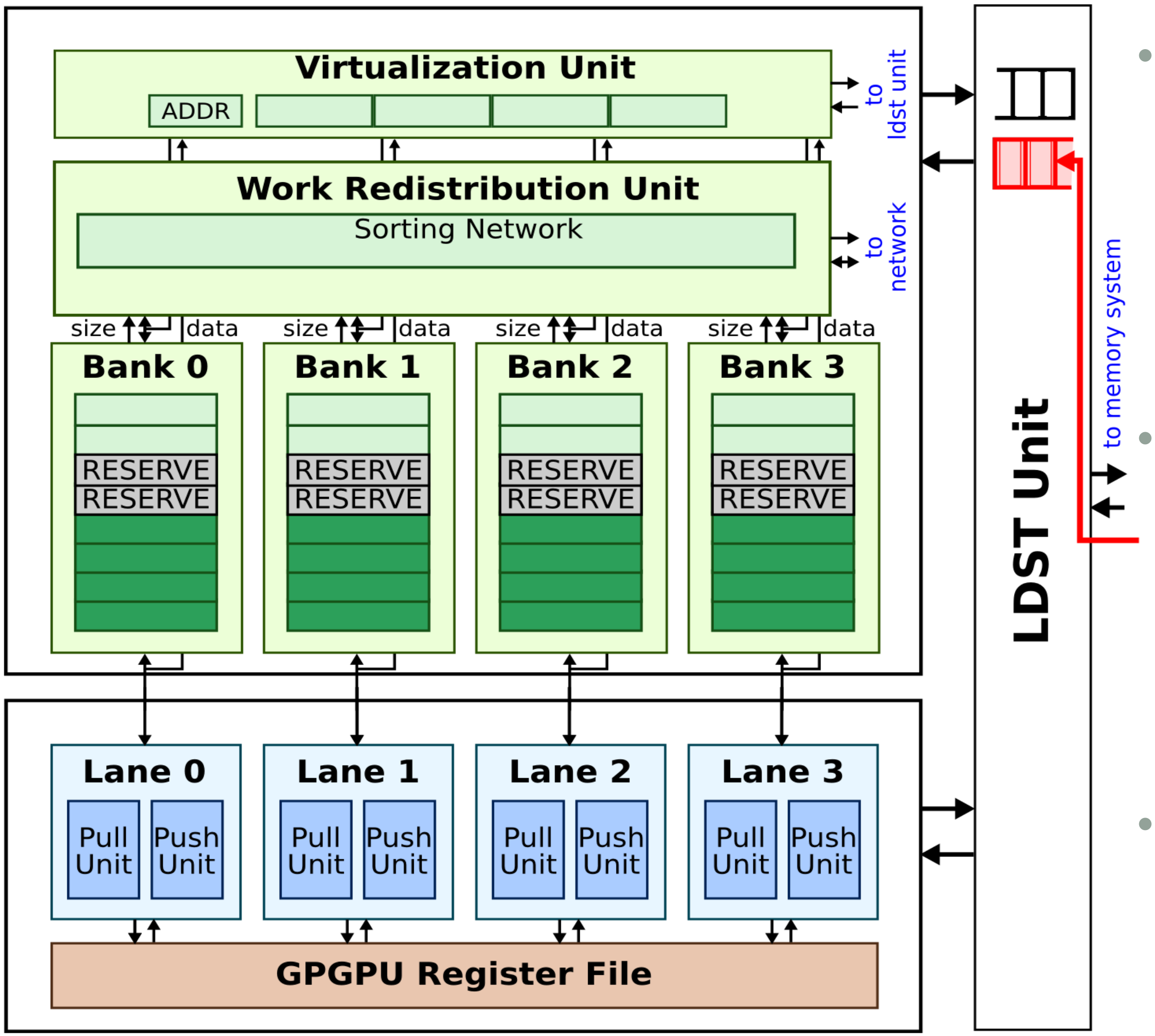
- Periodically check if banks are not full and work is in overflow buffer
- Reserve entries and inject refill request into load-store queue (1-bit to mark as refill)
- Refill responses are routed to virtualization unit for writeback

HWWL Work Refilling (Interval-Based)



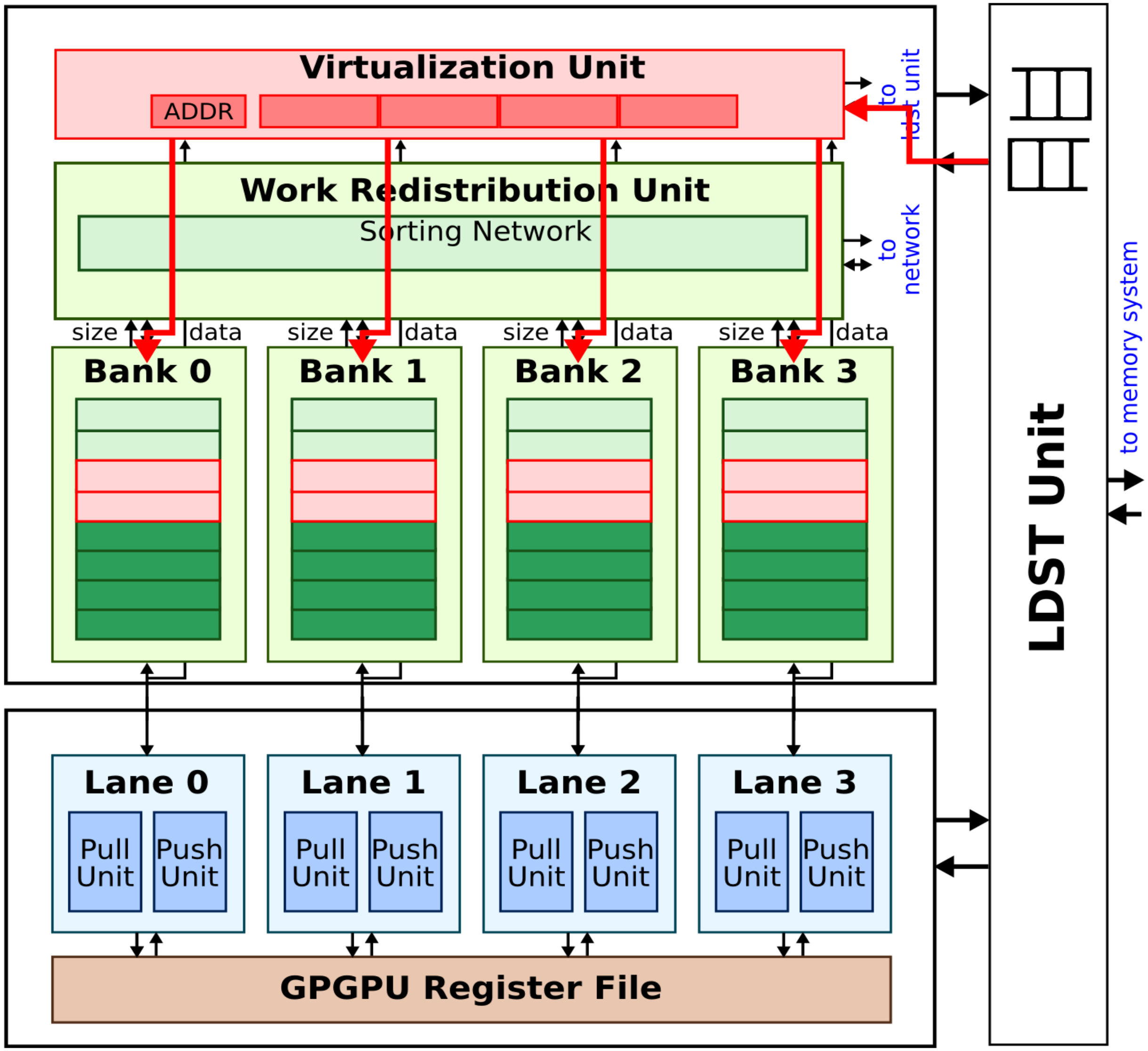
- Periodically check if banks are not full and work is in overflow buffer
- Reserve entries and inject refill request into load-store queue (1-bit to mark as refill)
- Refill responses are routed to virtualization unit for writeback

HWWL Work Refilling (Interval-Based)



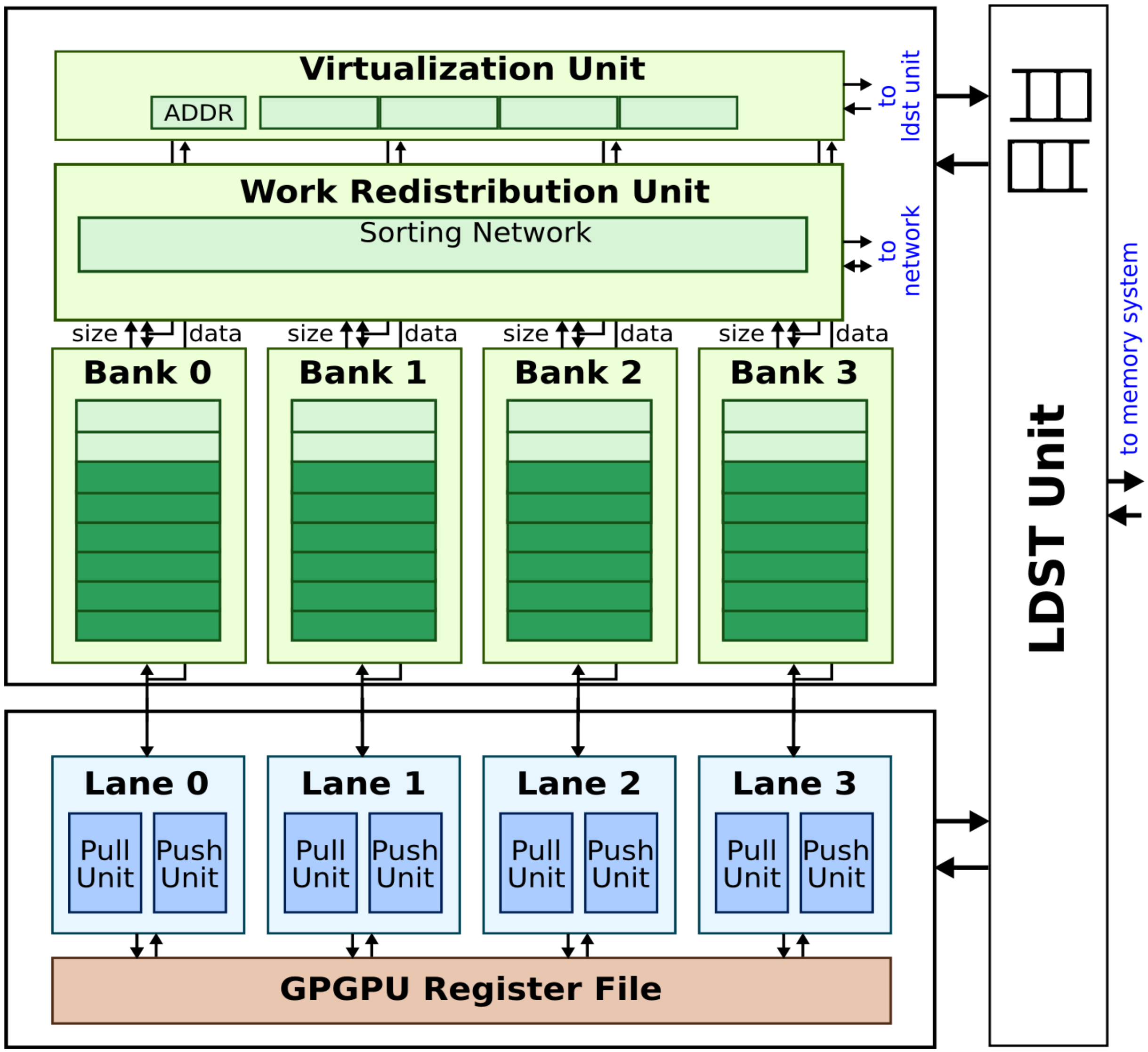
- Periodically check if banks are not full and work is in overflow buffer
- Reserve entries and inject refill request into load-store queue (1-bit to mark as refill)
- Refill responses are routed to virtualization unit for writeback

HWWL Work Refilling (Interval-Based)



- Periodically check if banks are not full and work is in overflow buffer
- Reserve entries and inject refill request into load-store queue (1-bit to mark as refill)
- Refill responses are routed to virtualization unit for writeback

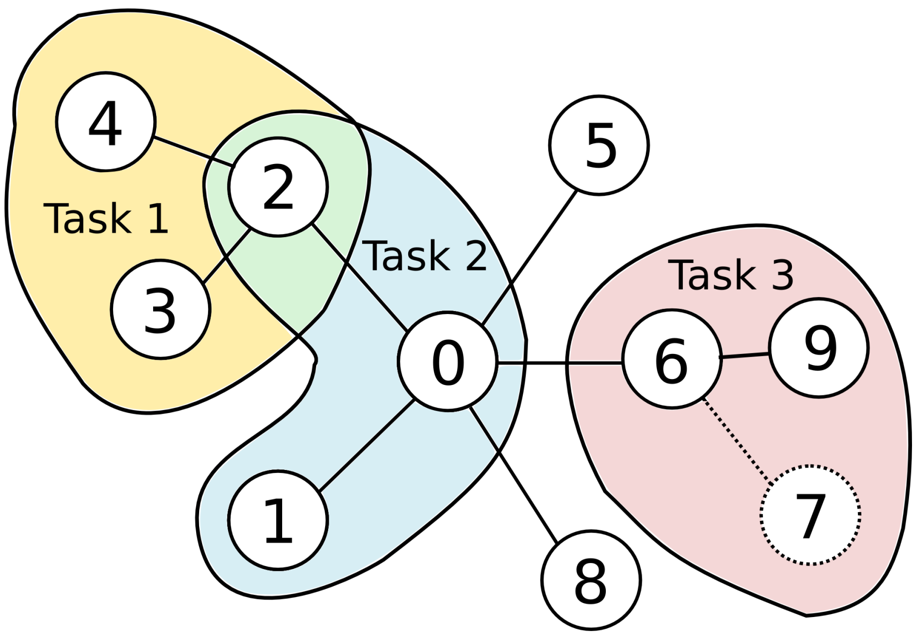
HWWL Work Refilling (Interval-Based)



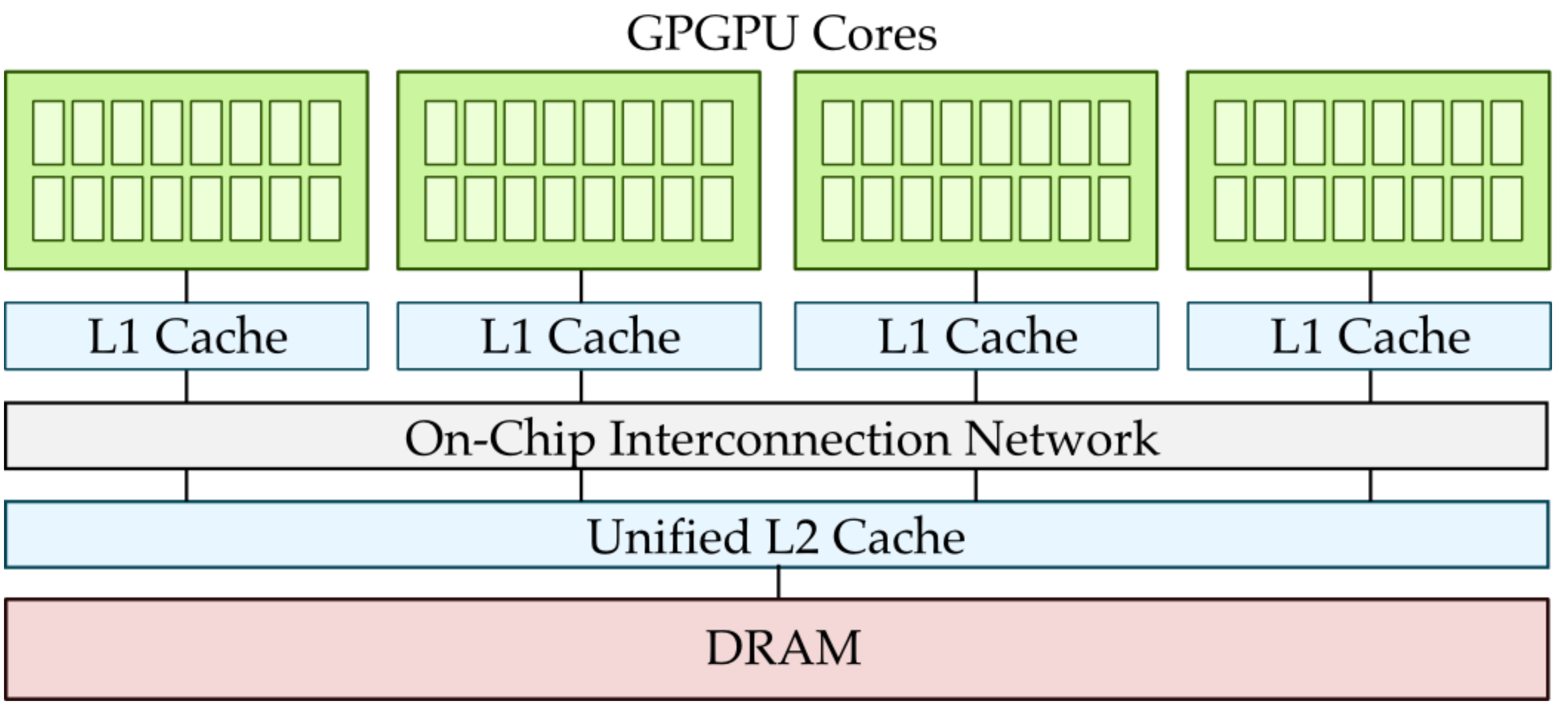
- Periodically check if banks are not full and work is in overflow buffer
- Reserve entries and inject refill request into load-store queue (1-bit to mark as refill)
- Refill responses are routed to virtualization unit for writeback

Presentation Outline

- Motivation
- Mapping Irregular Algorithms to GPGPUs
- Developing Optimized Software Baselines
- Fine-Grain Hardware Worklists
- **Evaluation**

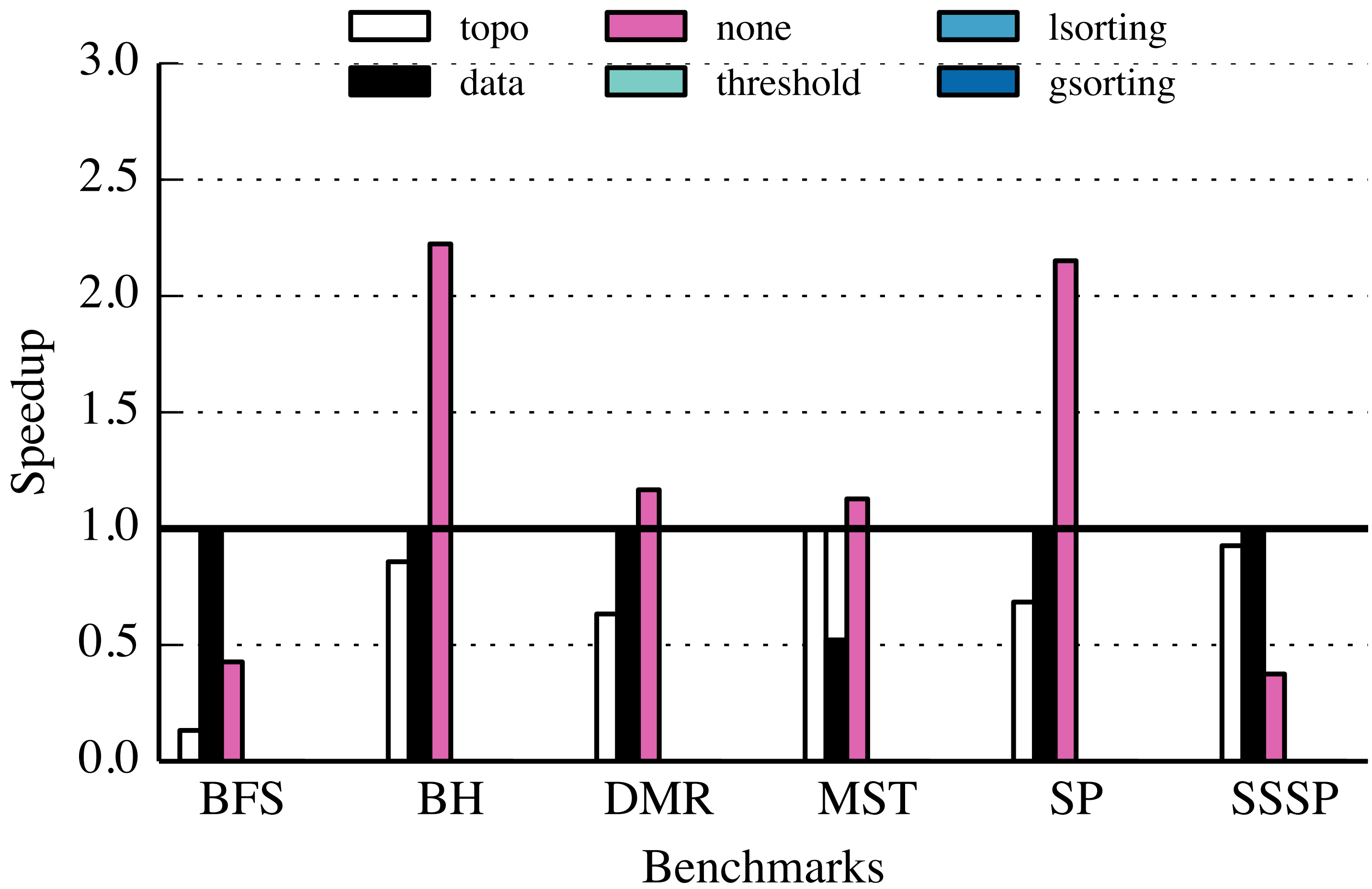


Methodology



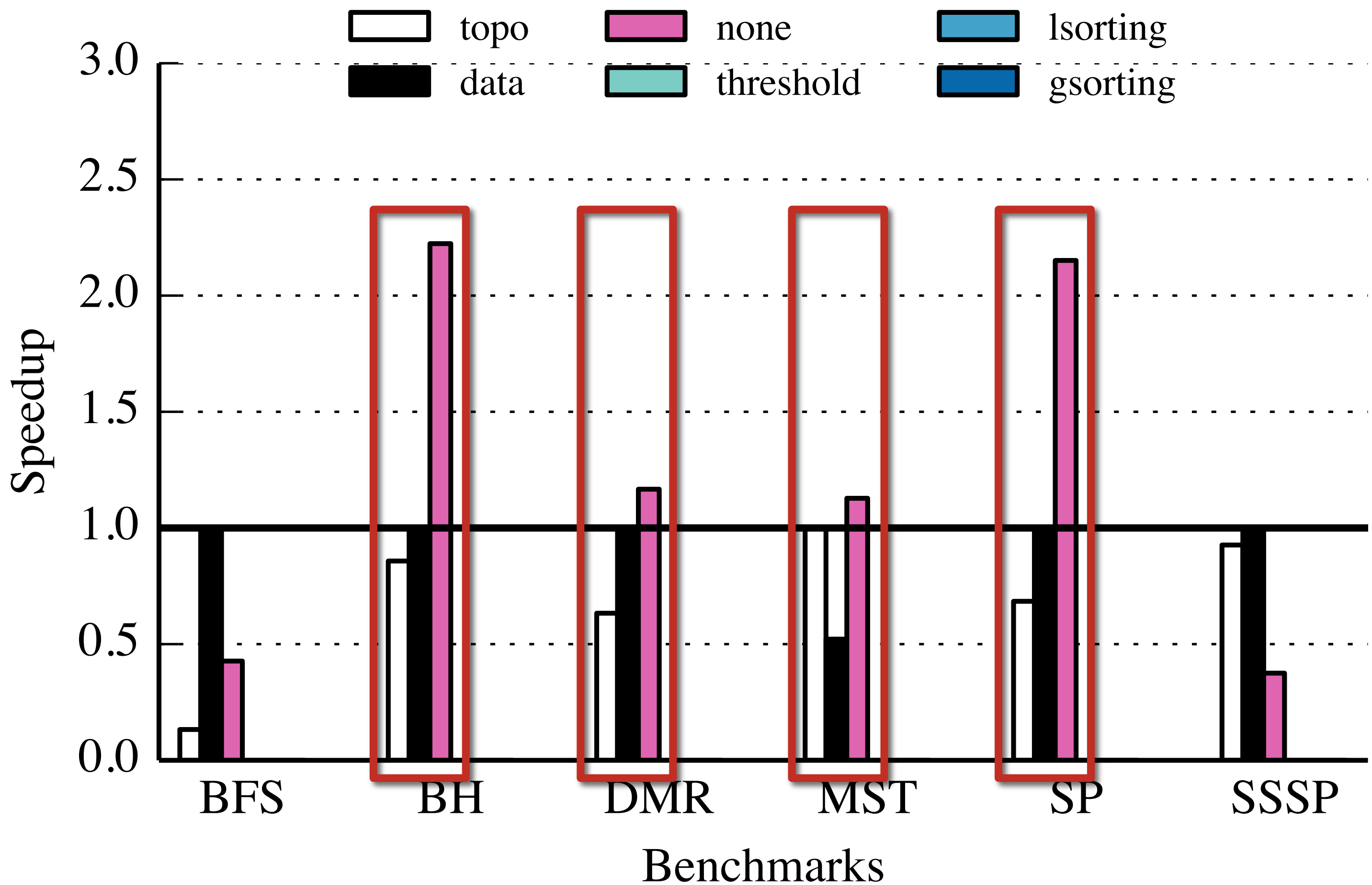
- Evaluate highly optimized LonestarGPU benchmarks on GPGPU-Sim 3.0 (GTX480 configuration)
- 4 cores with 16 lanes each (scalability study in paper)
- Private 16KB L1$, unified 786KB L2$
- FIFO-based DRAM model

Performance: HWWL Banks (No Redistro)



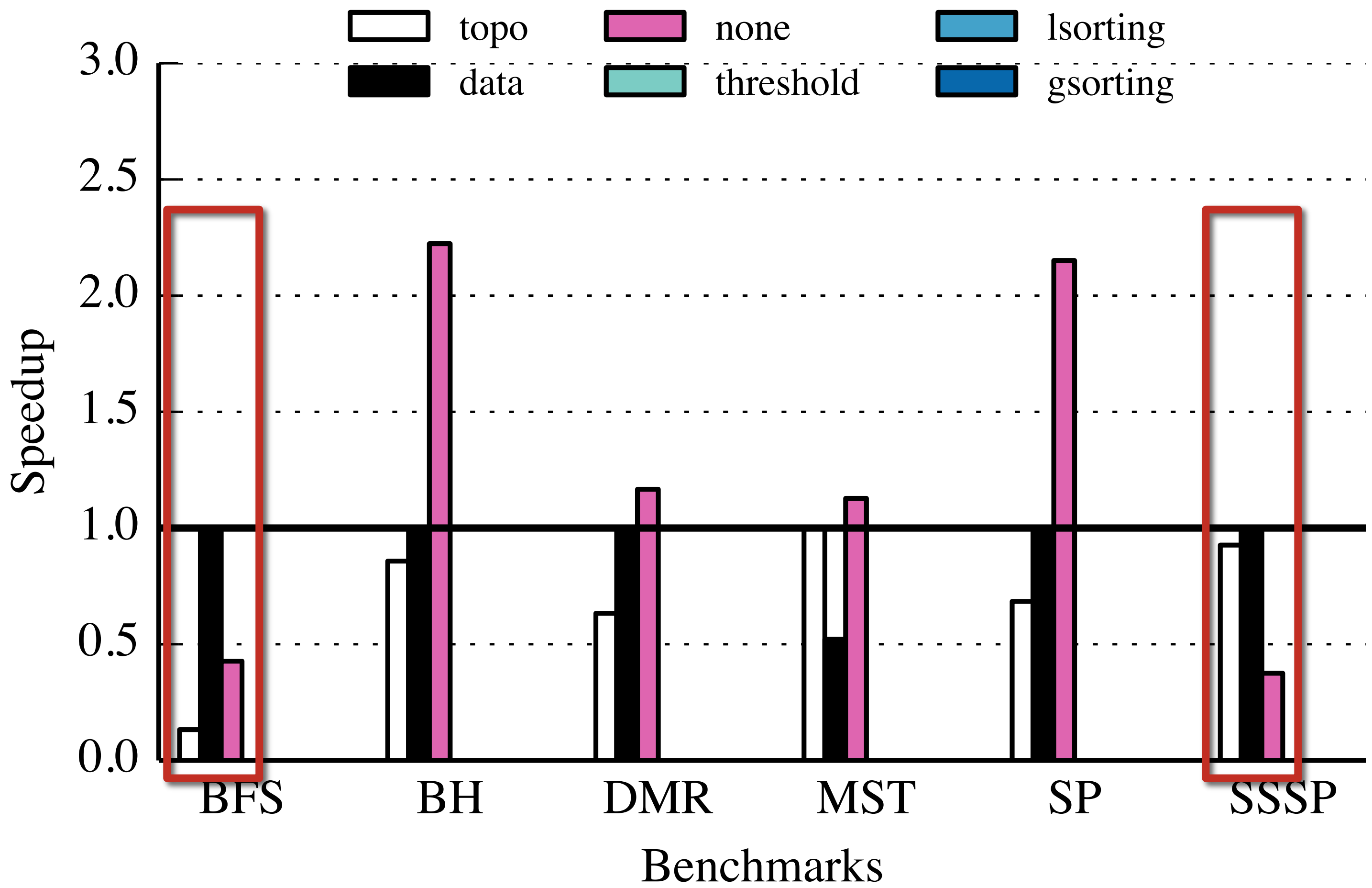
HWWL results normalized to best of topology- or data-driven implementations running on nominal GPGPU

Performance: HWWL Banks (No Redistro)



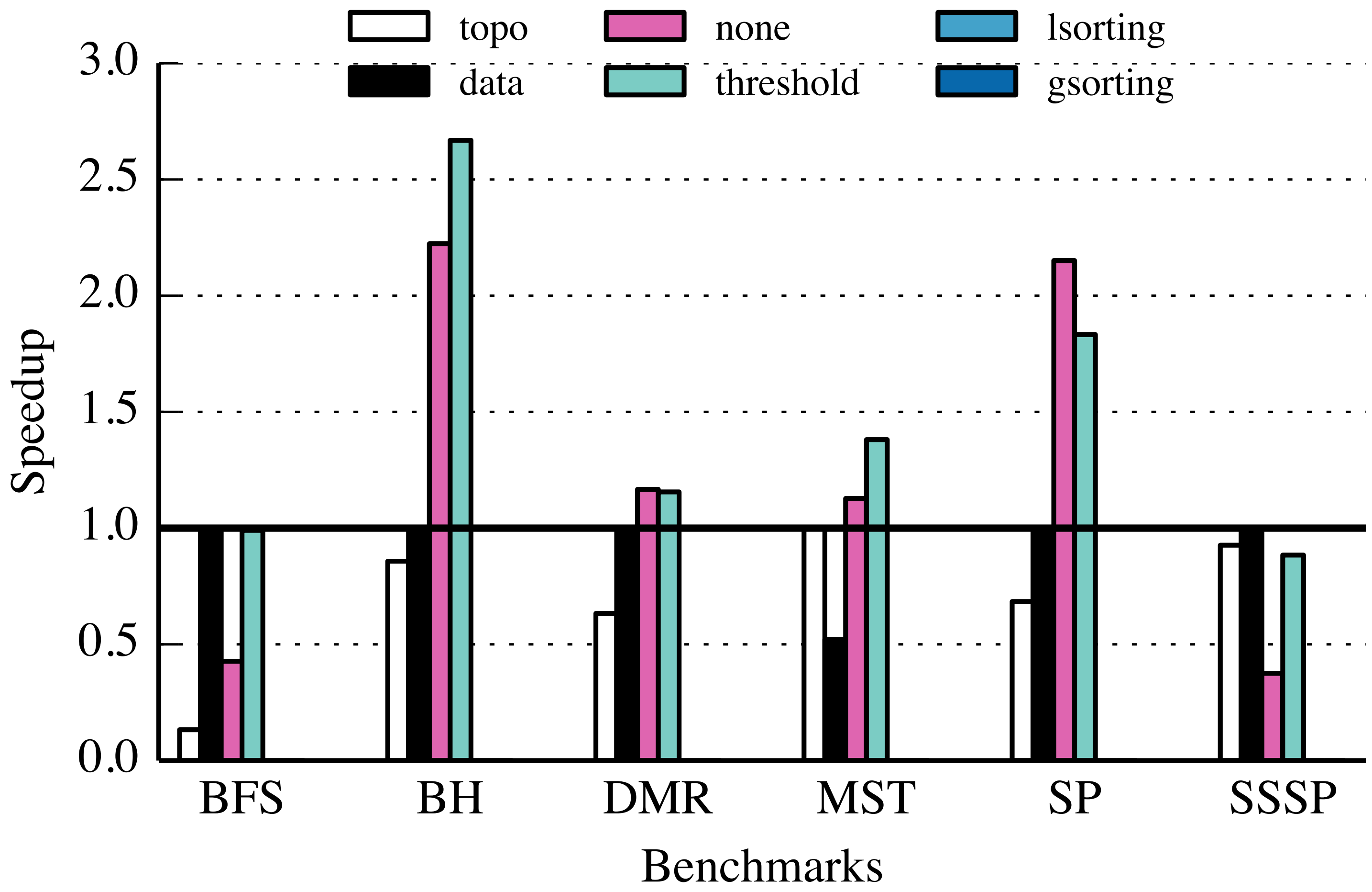
- Up to 67% reduction in memory stalls
- Up to 16% reduction in dynamic instructions

Performance: HWWL Banks (No Redistro)

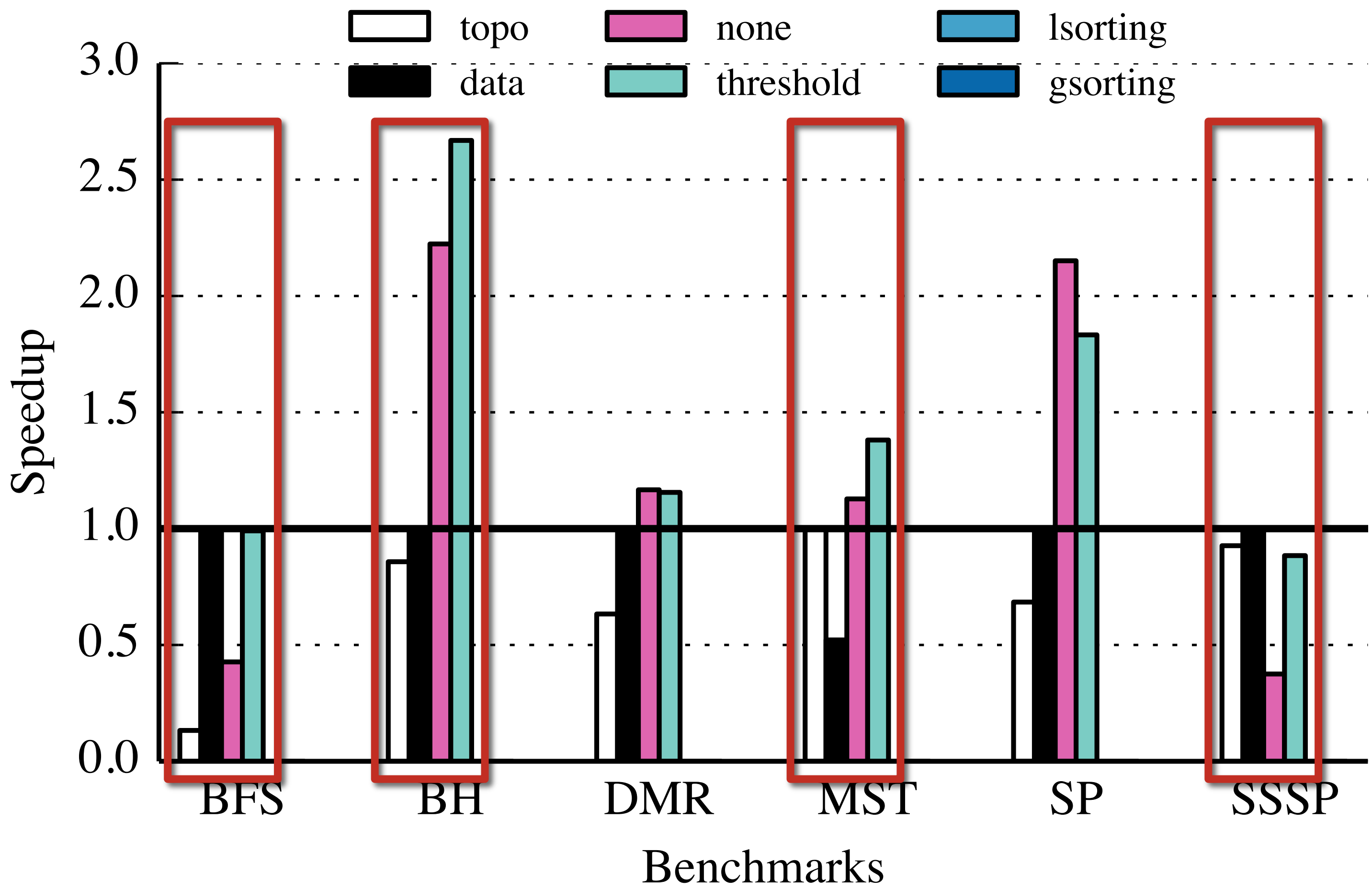


Benchmarks with less inherent load balancing perform worse!

Performance: HWWL Work Redistribution

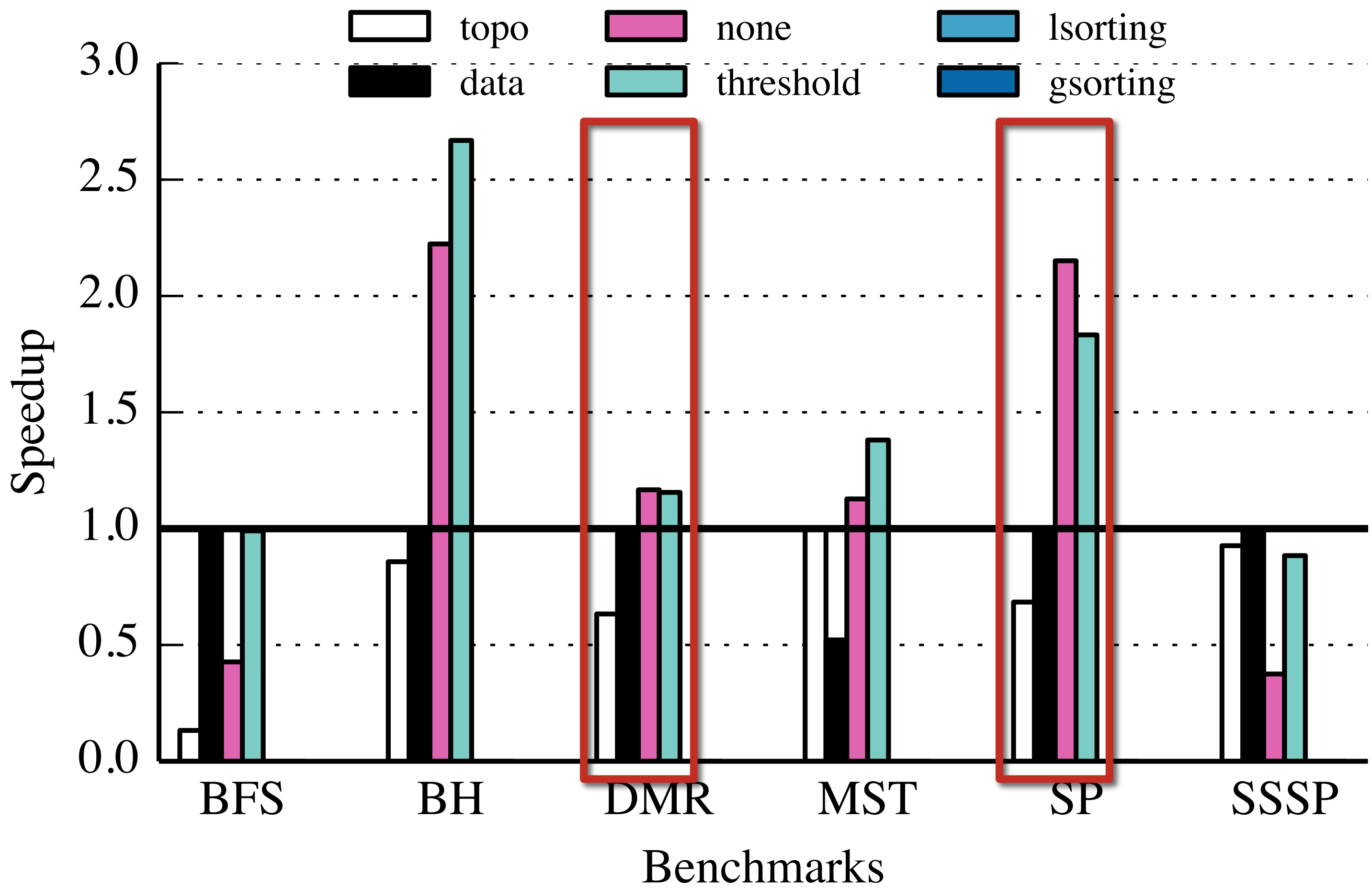


Performance: HWWL Work Redistribution



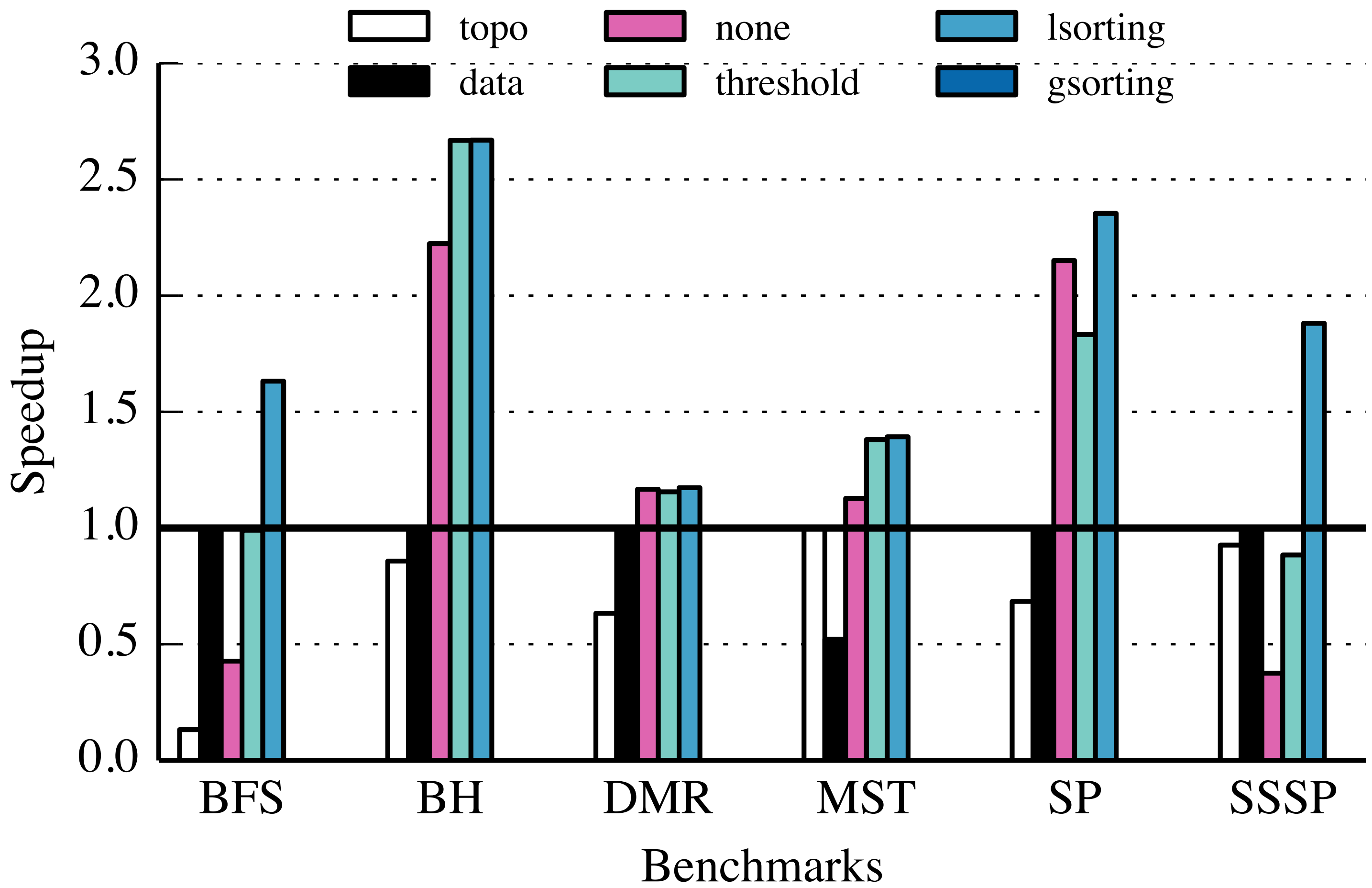
Performance from improved load balancing (order of magnitude decrease in WAIT tokens pulled)

Performance: HWWL Work Redistribution

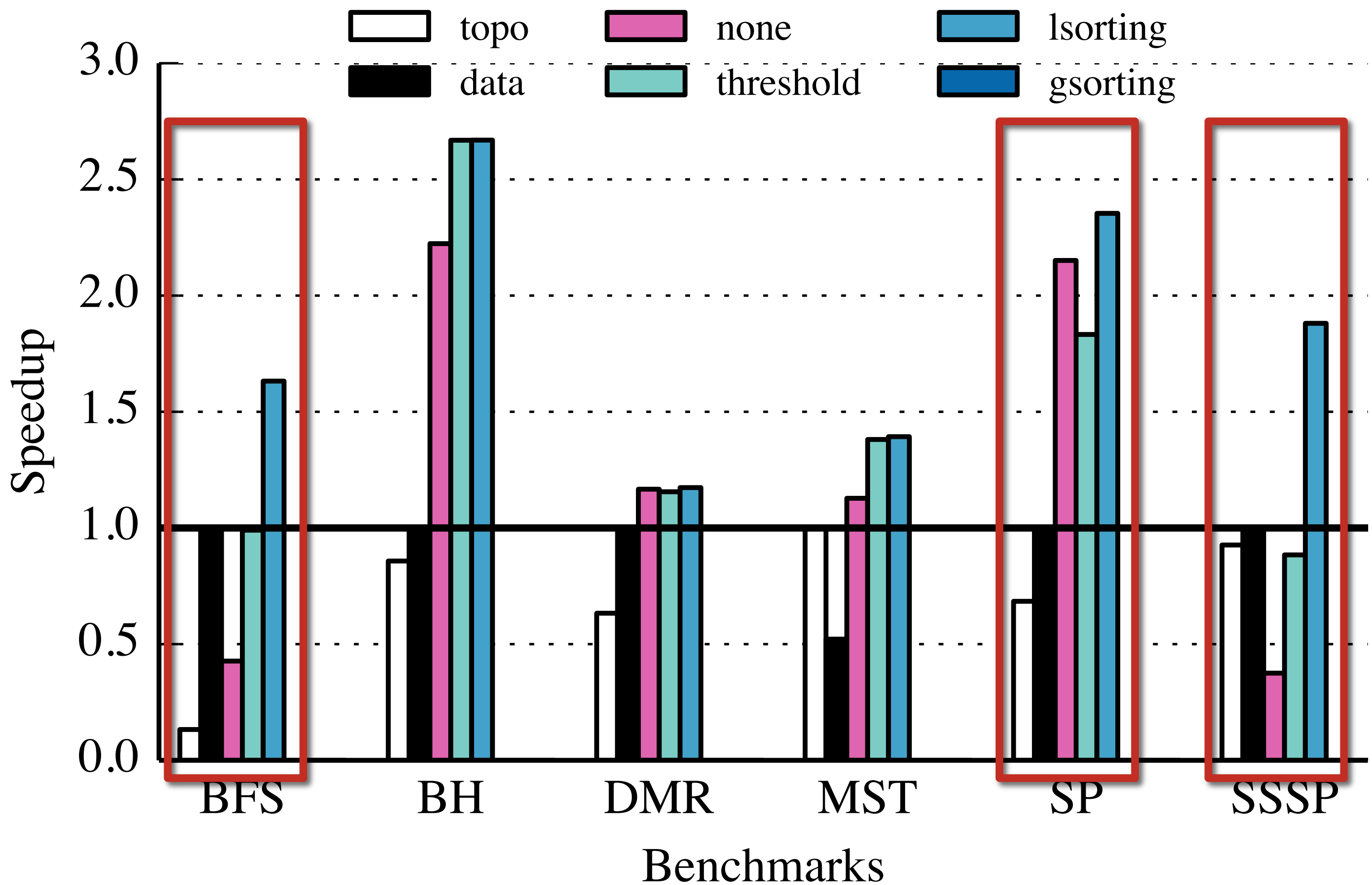


In some cases, threshold-based redistribution yields undesirable work distributions (few banks hog)

Performance: HWWL Work Redistribution

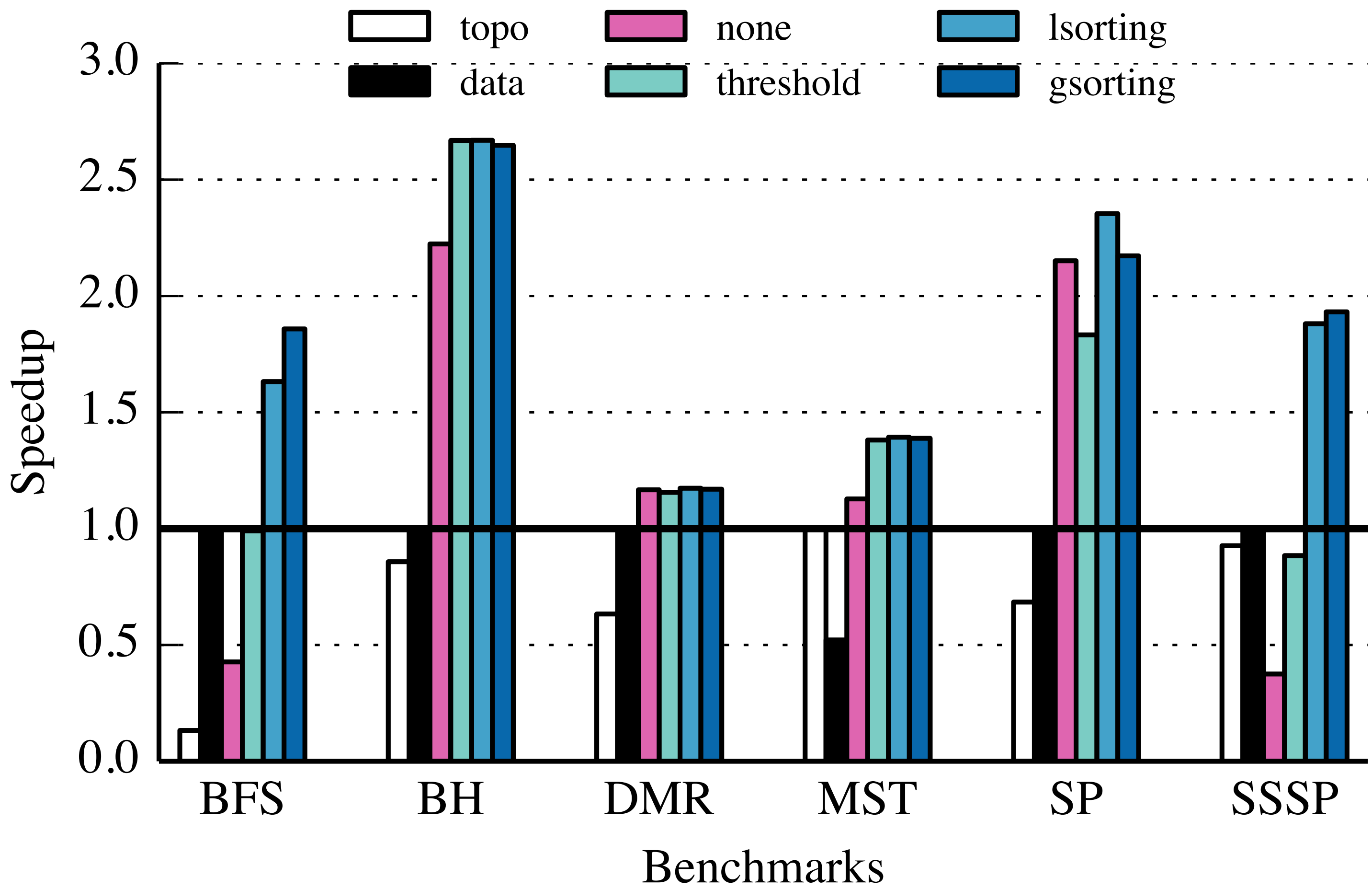


Performance: HWWL Work Redistribution

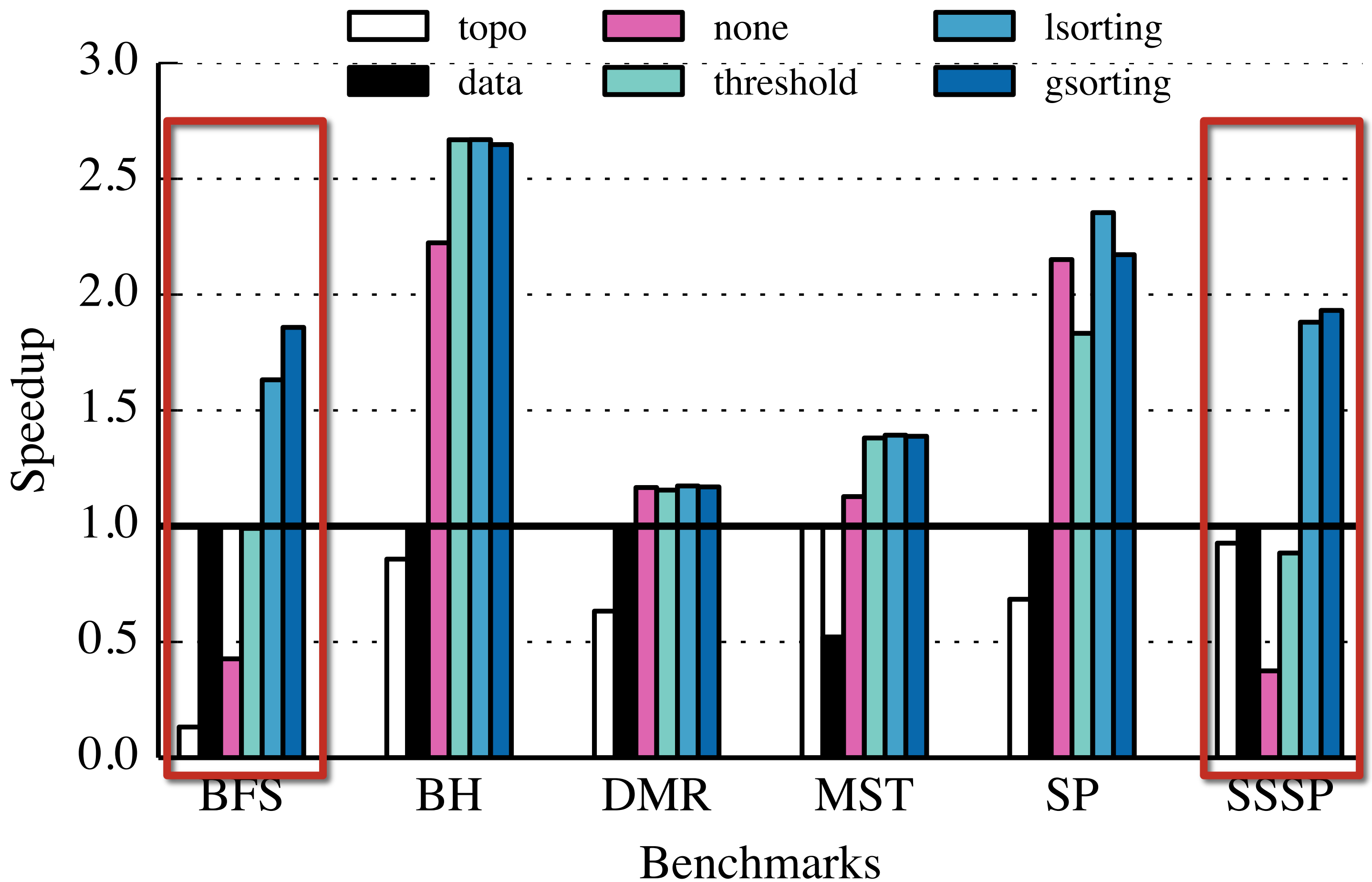


Sorting-based redistribution increases complexity for improved load balancing

Performance: HWWL Work Redistribution

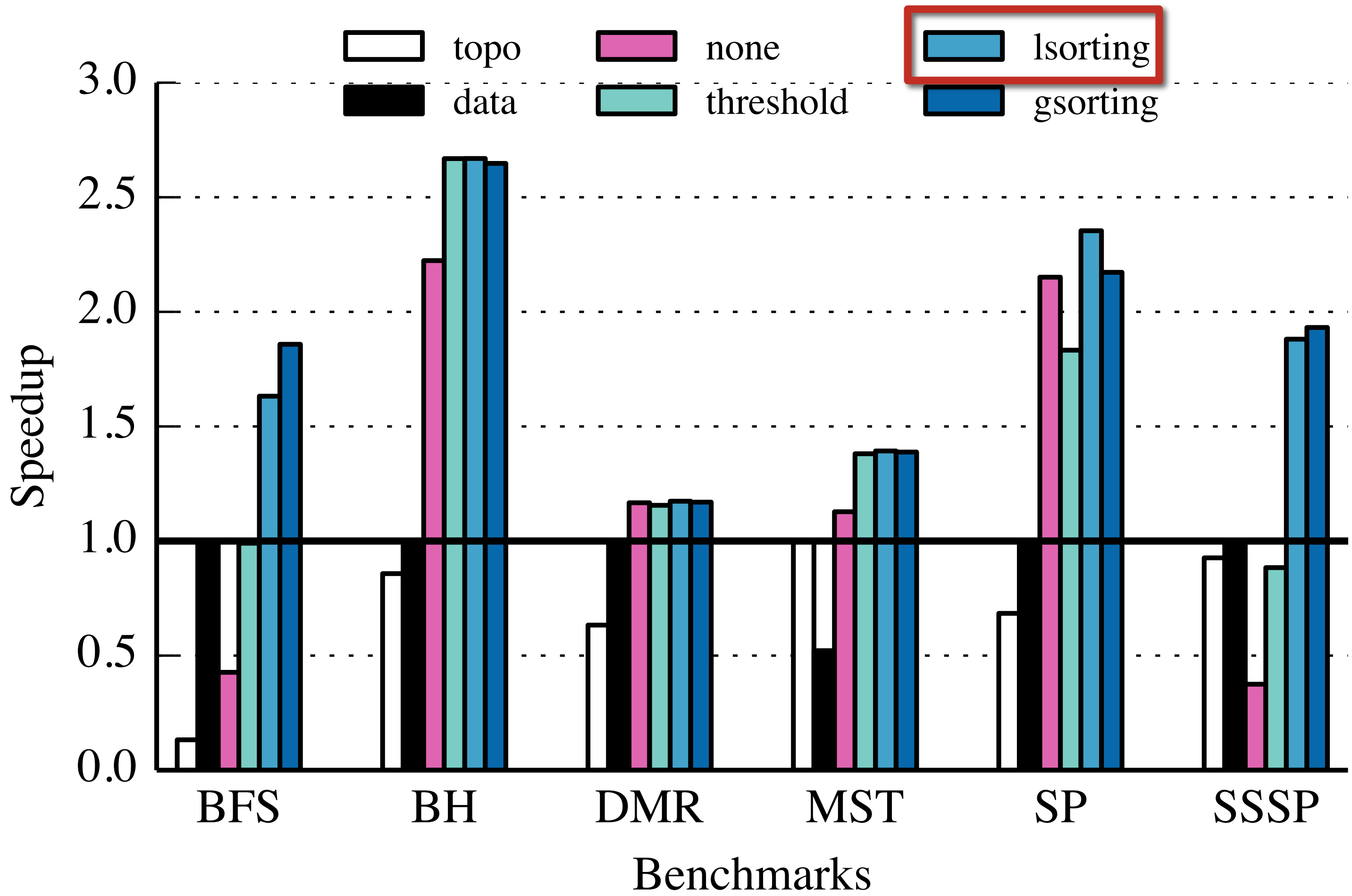


Performance: HWWL Work Redistribution



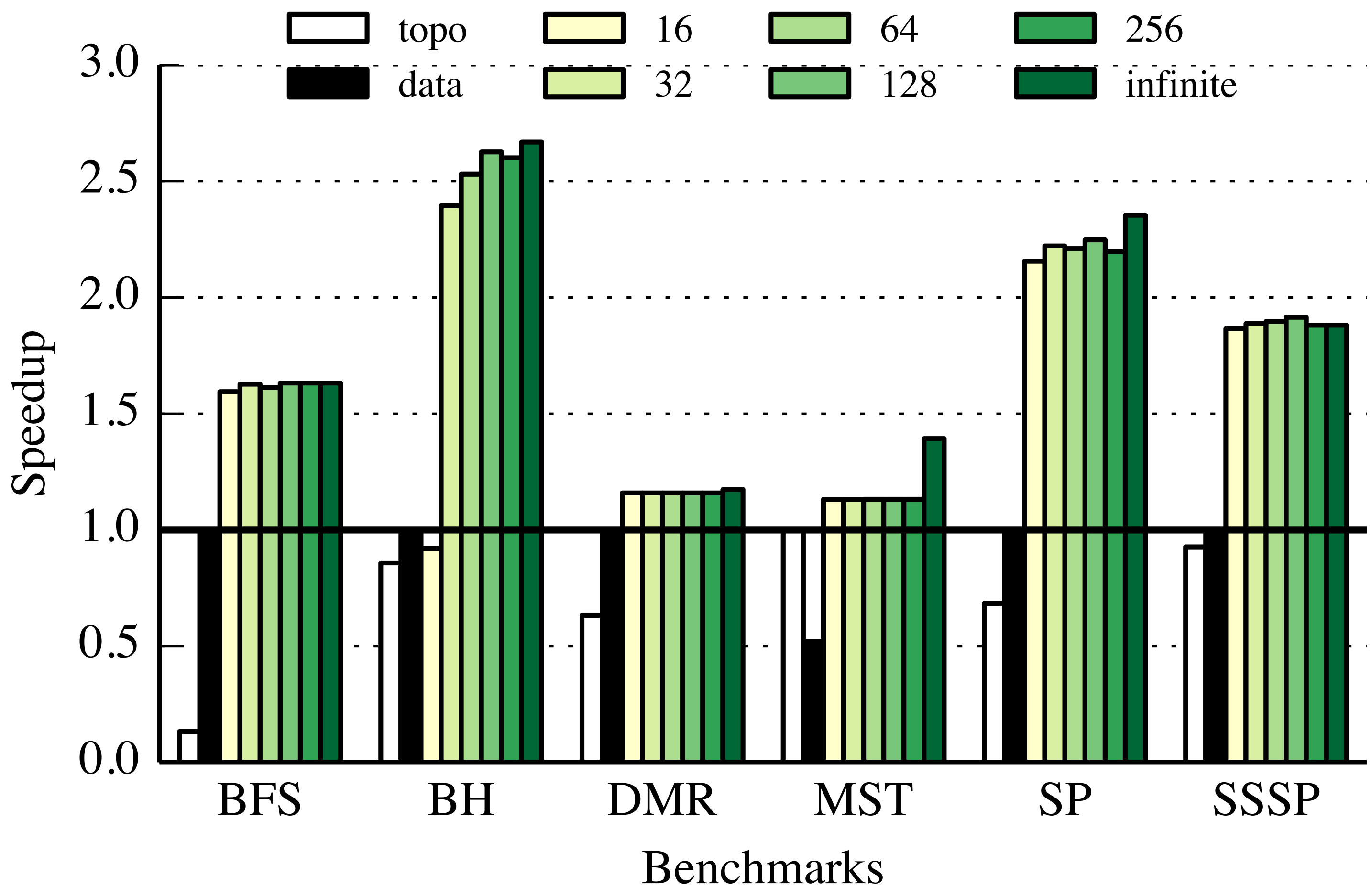
Providing global bank information to monolithic sorter only helps marginally in isolated cases

Performance: HWWL Work Redistribution



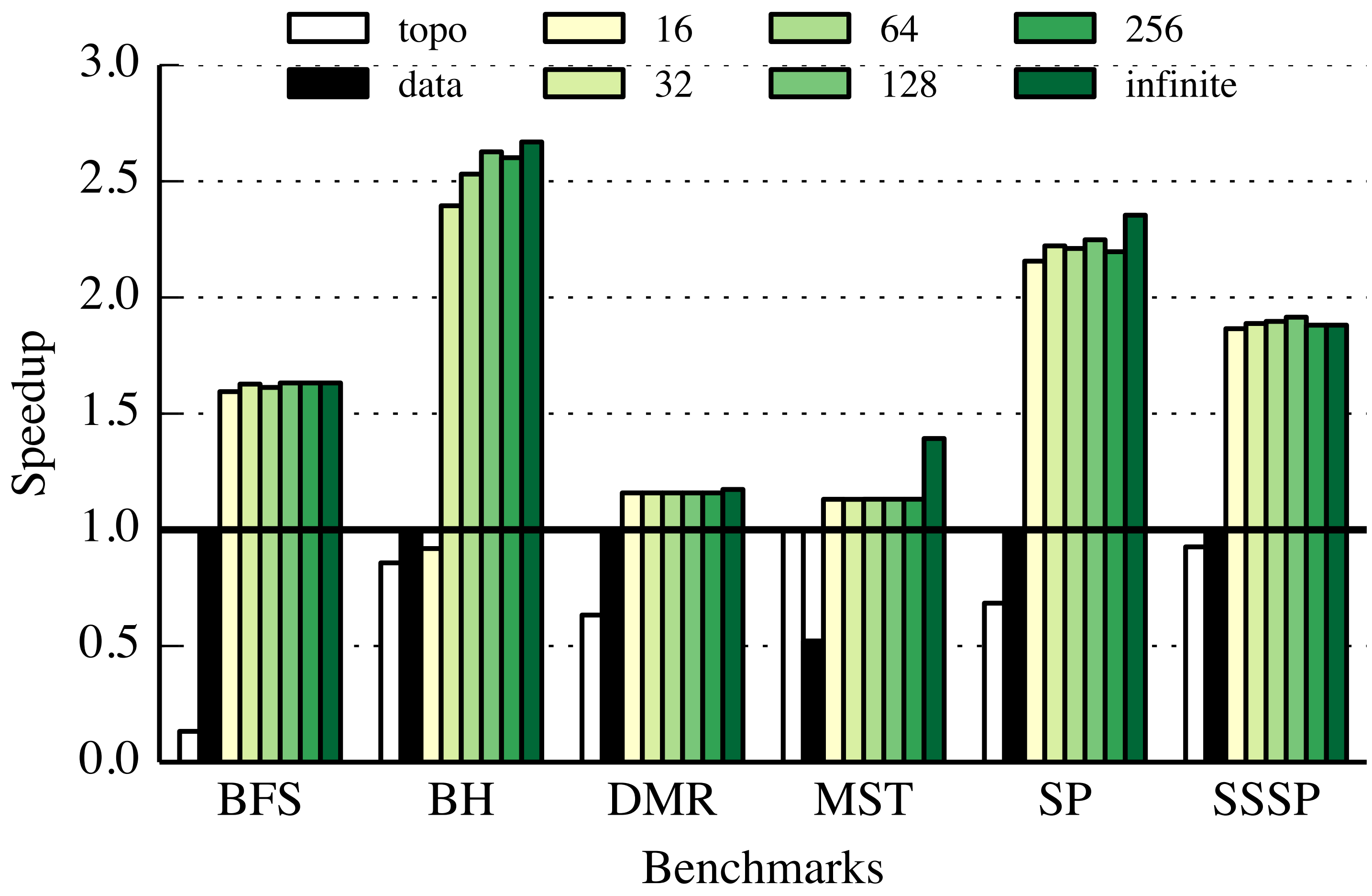
Choose local sorting-based redistribution

Performance: HWWL Spilling/Refilling



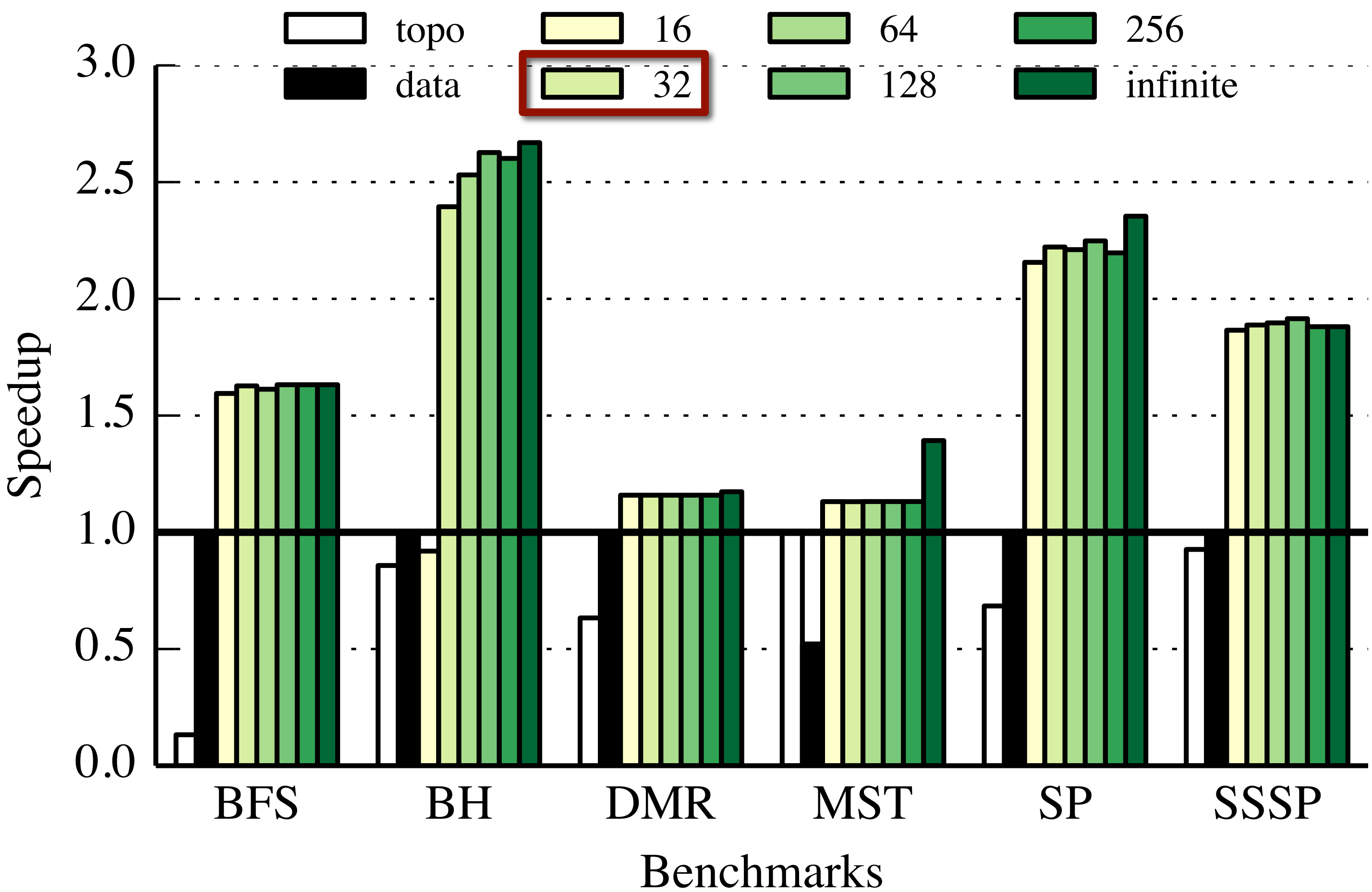
Focus on interval-based virtualization (minimal overhead for improved performance on simpler compute operators)

Performance: HWWL Spilling/Refilling



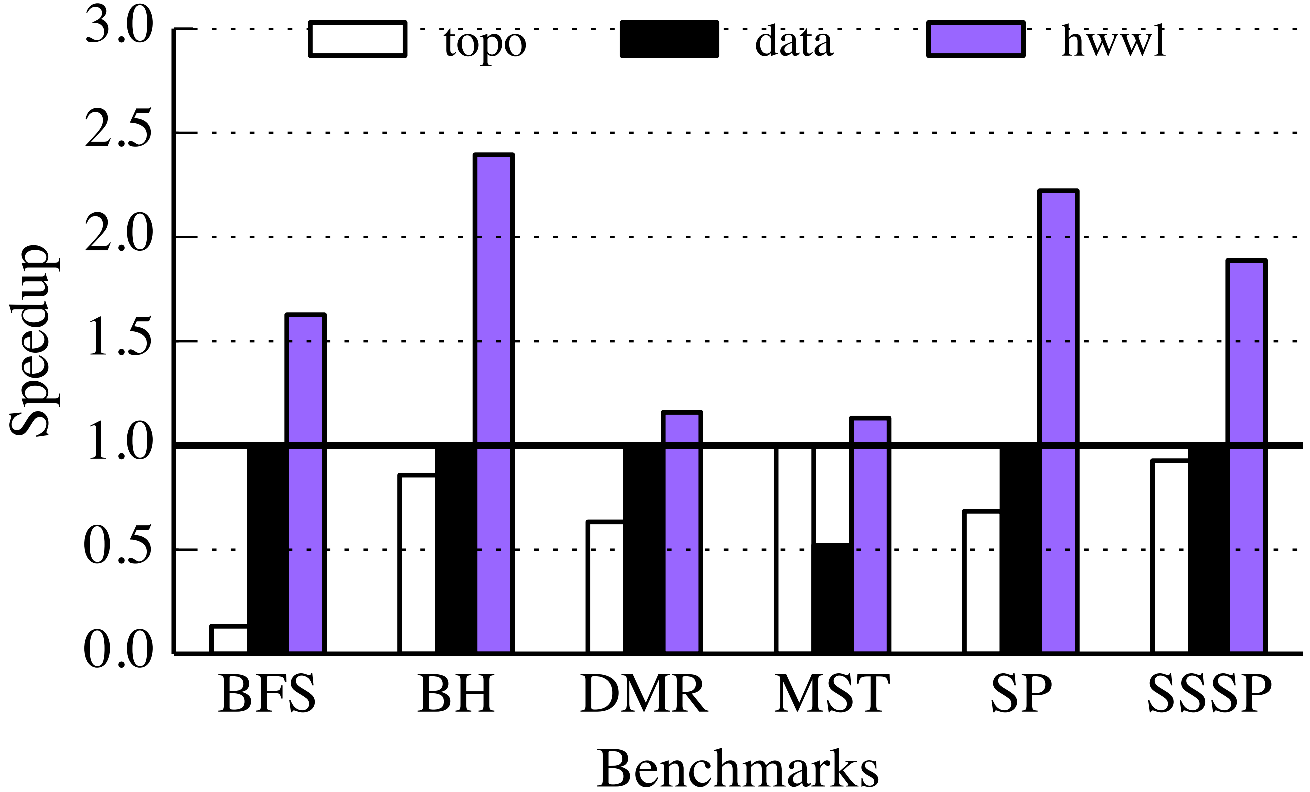
Virtualization does not significantly hurt performance in most cases

Performance: HWWL Spilling/Refilling



32 entries is enough to achieve most of potential performance

Overall HWWL Performance



2.5% of GPGPU regfile area for banks

~160 um^2 for sorting network

- Realistic HWWL with **32 entries per bank**, **local sorting work redistribution**, and **interval-based virtualization**
- Speedups ranging from **1.2—2.4X** over the best SW implementation

Take-Away Points

- Software optimizations can be effective, but require significant programmer effort and time, performance not guaranteed
- Relatively simple hardware support can ease the burden on the programmer while improving performance on algorithms difficult to map to GPGPUs

Sponsored by:
NDSEG Fellowship
NSF CAREER Award
Intel
NVIDIA

Special thanks to LonestarGPU team!