Automated architecture-aware mapping of streaming applications onto GPUs

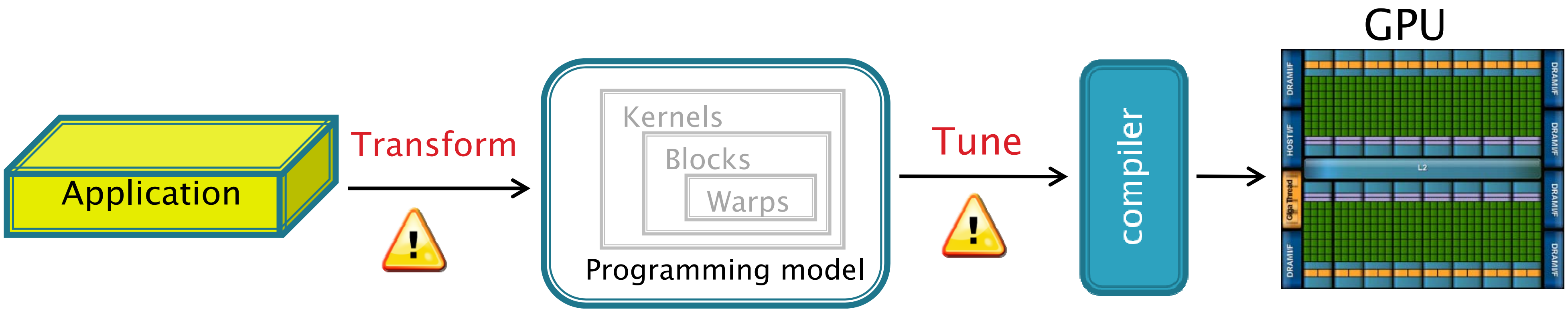
Andrei Hagiescu
Weng-Fai Wong
School of Computing,
National University of Singapore,
Singapore

School *of* Computing

Huynh Phung Huynh
Rick Siow Mong Goh
*A*STAR Institute of*
High Performance Computing,
Singapore

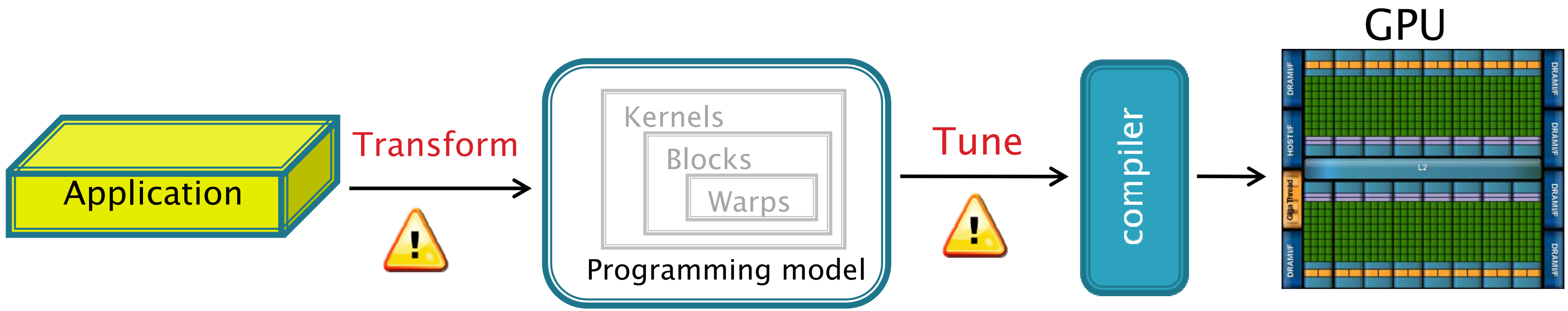
Institute of
High Performance
Computing
A*STAR

Overview

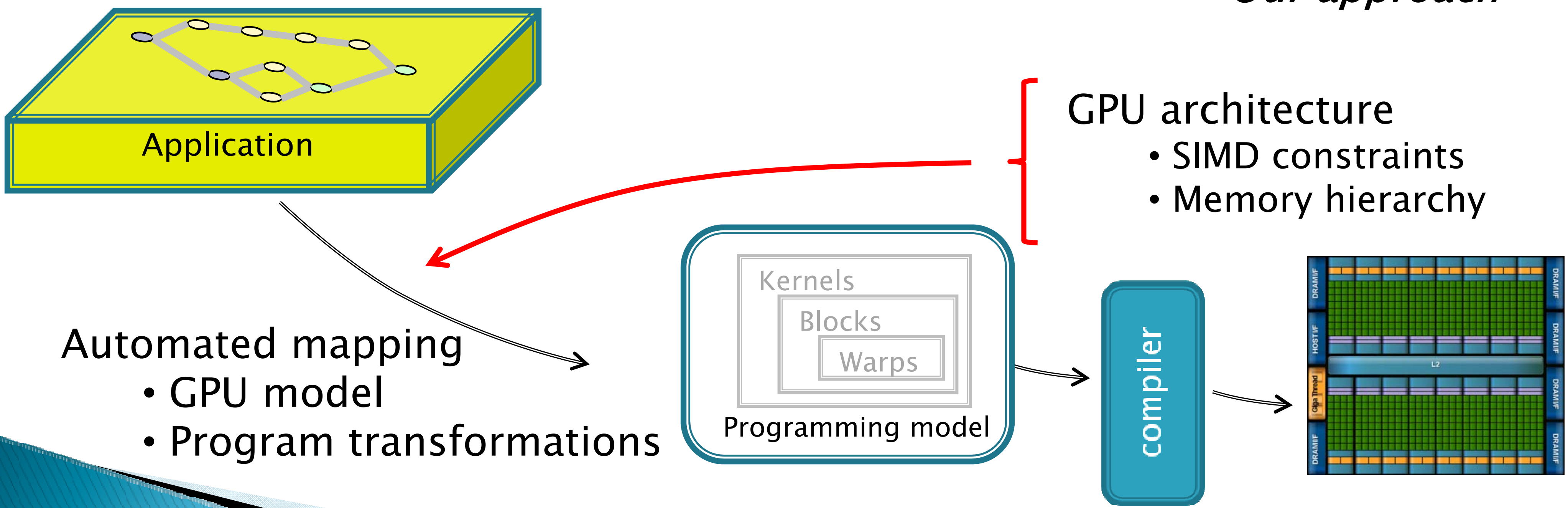


GPU
Application
Transform
Kernels
Blocks
Warps
Programming model
Tune
compiler

Overview

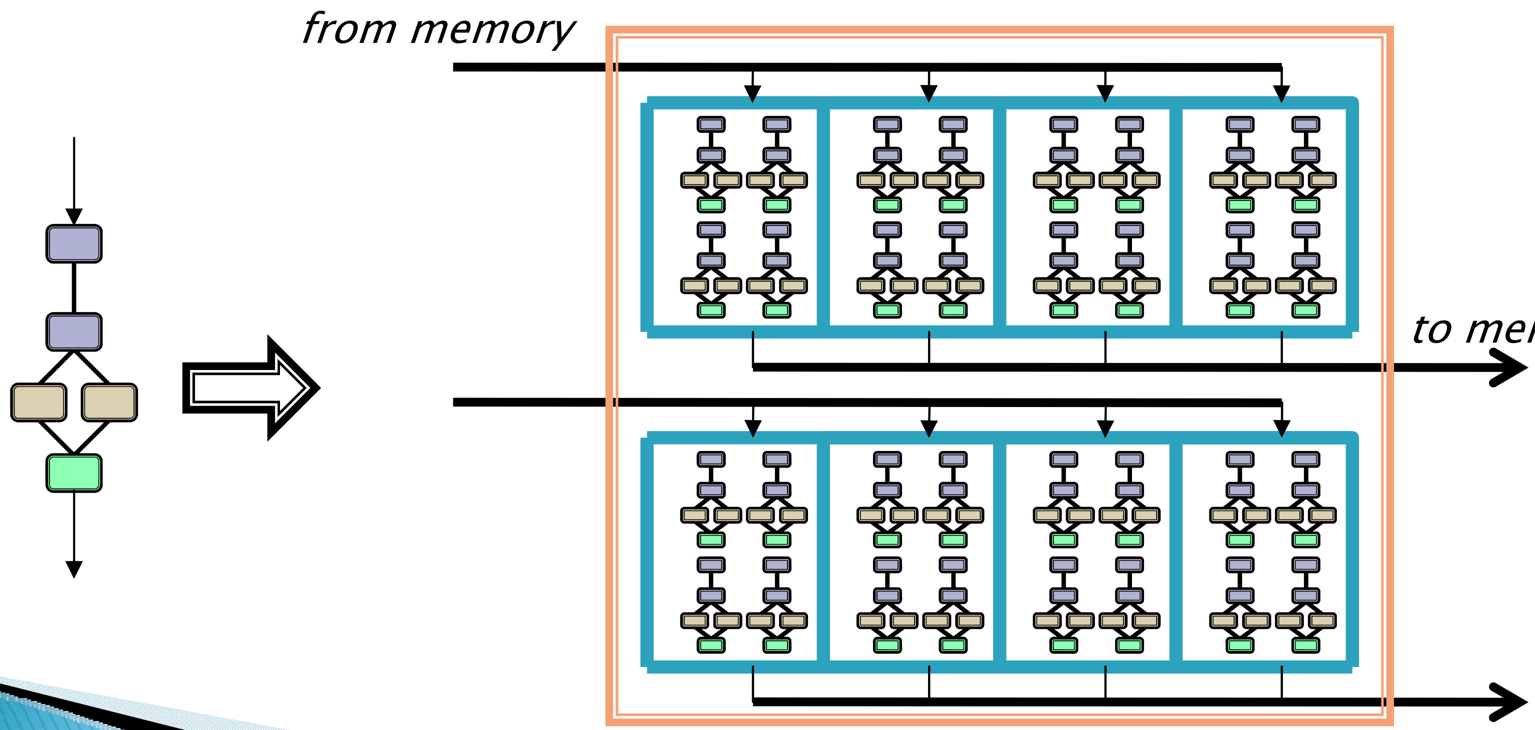


Our approach



Our mapping strategy

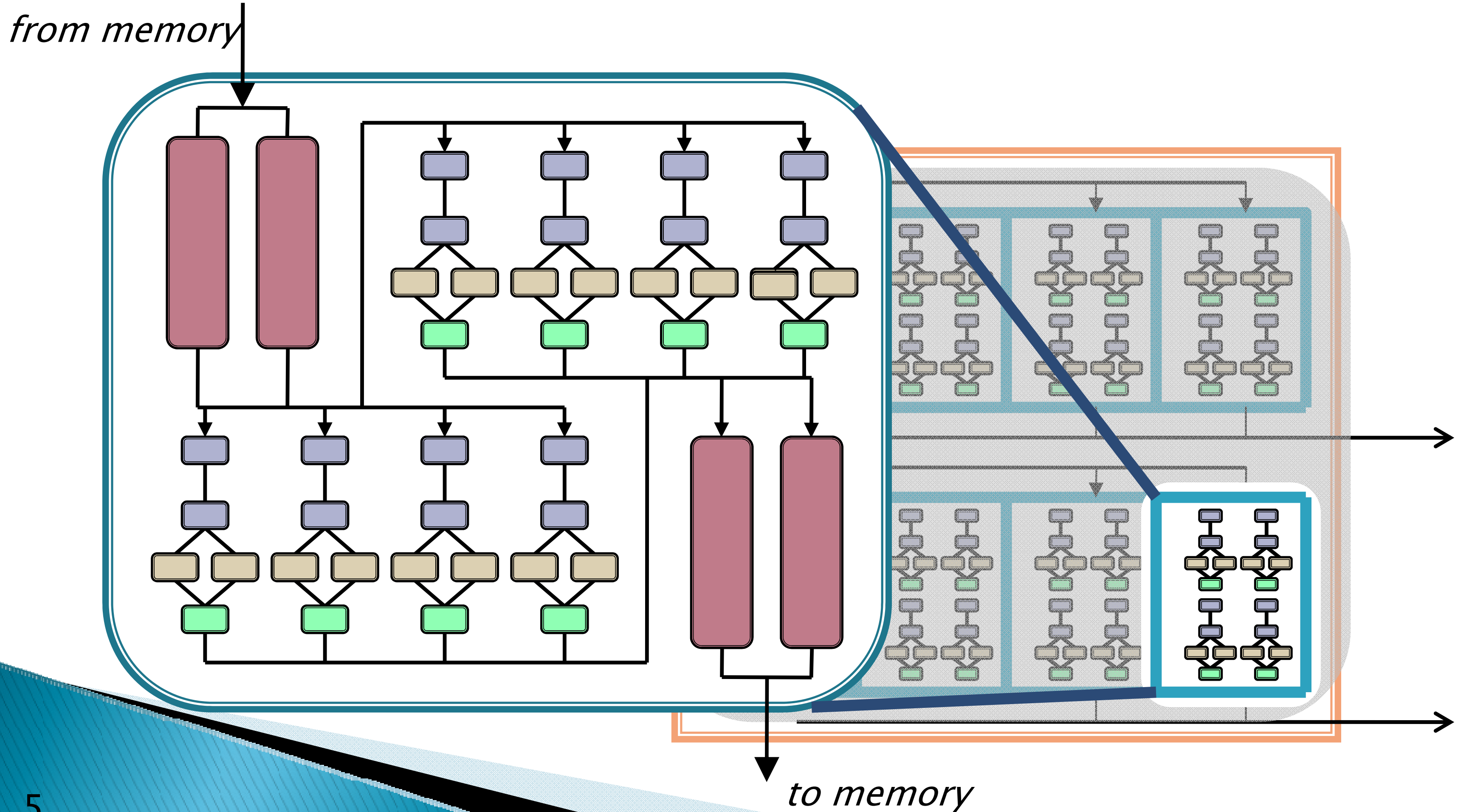
① Stream graph → Parallel instances of the entire graph



Our mapping strategy

① Stream graph → Parallel instances of the entire graph

② Novel memory access scheme

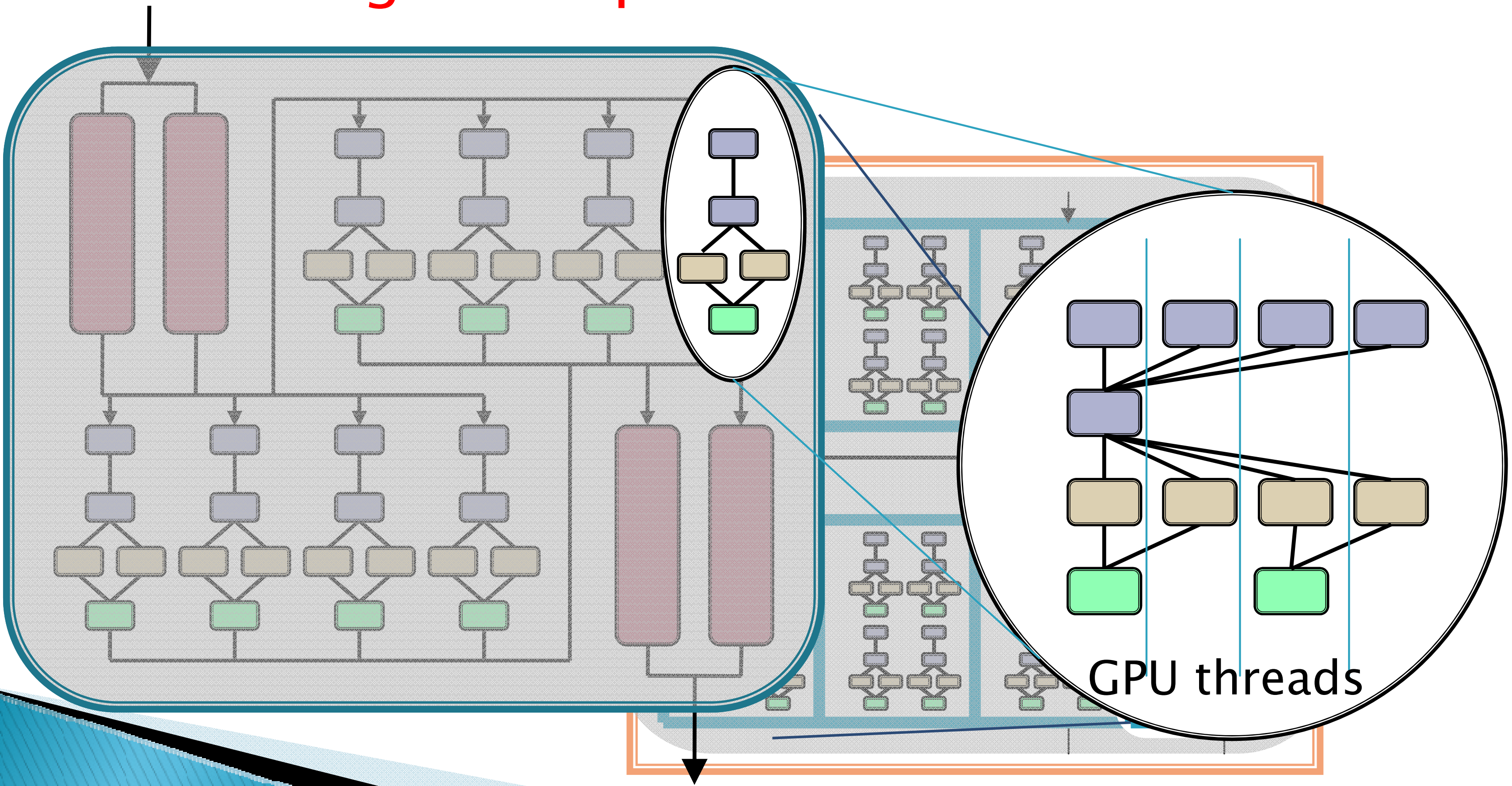


Our mapping strategy

① Stream graph → Parallel instances of the entire graph

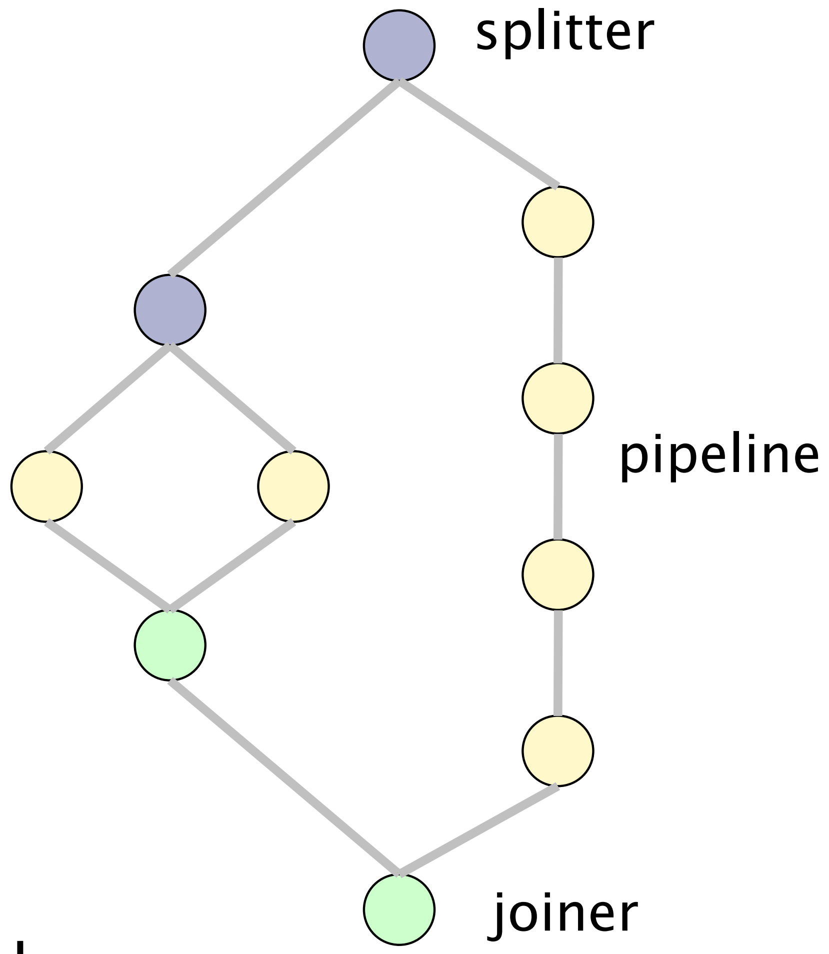
② Novel memory access scheme

③ Utilize fine-grained parallelism



StreamIt

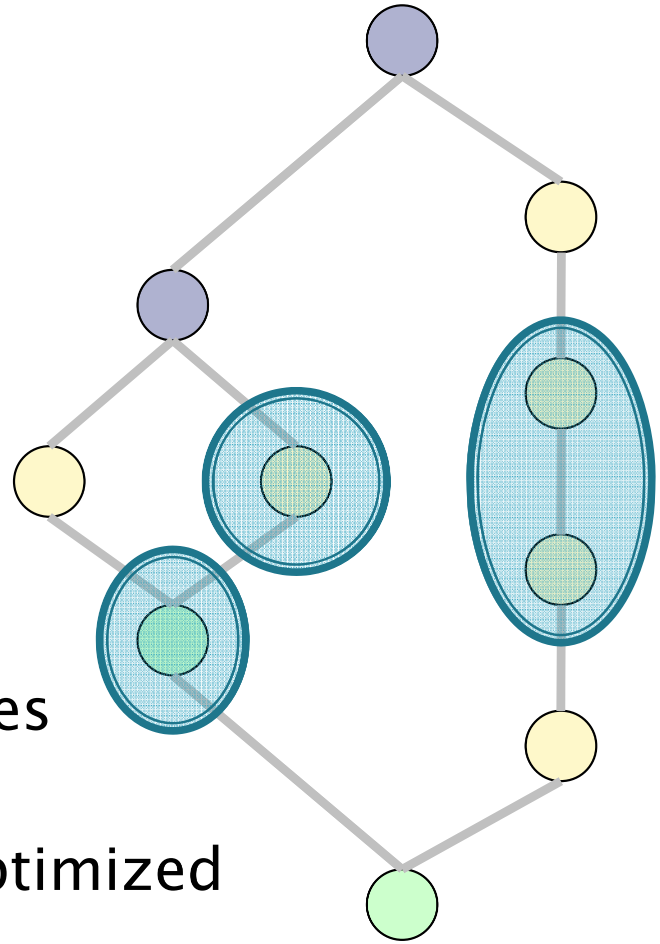
- Hierarchical stream graph
  - Well defined rates
  - Pipeline
  - Splitters / Joiners
  - Mostly *stateless* filters
- StreamIt compiler
  - Schedules
  - Flattens
  - Analyzes
- Peeking
  - Alternative to filters with state
  - Allows access to input consumed by future iteration



Stream graph

Related work on StreamIt to GPU

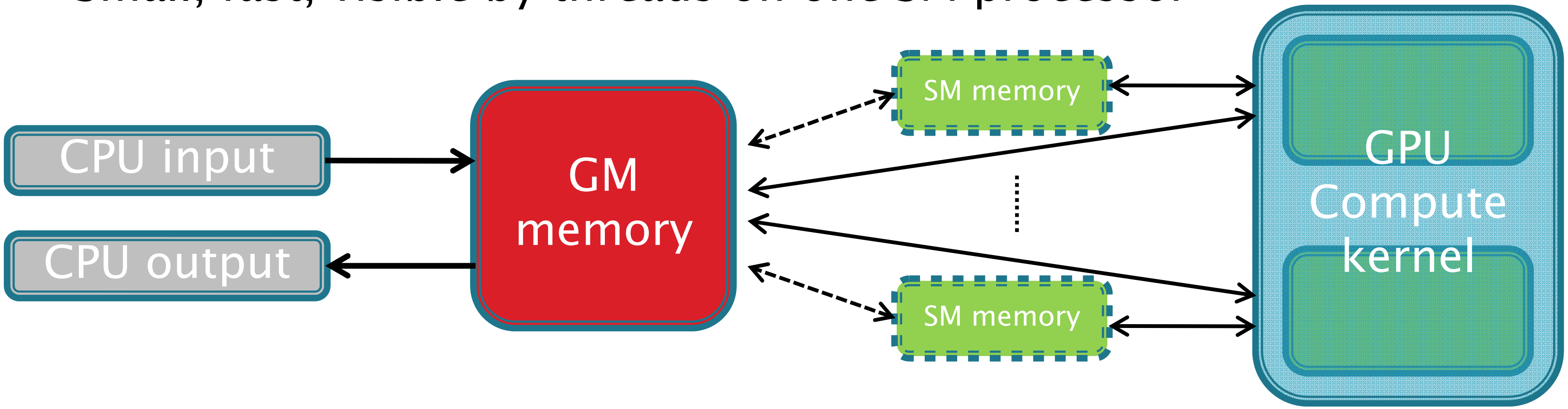
- Udupa et al. (CGO 2009)
  - Software pipelined execution of stream programs on GPUs
    - ⚠ no memory prefetching
    - ⚠ pipeline computation



- Hormati et al. (ASPLOS 2011)
  - Sponge: Portable Stream Programming on Graphics Engines
    - ✔ memory access scheme
    - ⚠ memory traffic not fully optimized (filters fused partially)
    - ⚠ no compression on multiple threads

GPU memory hierarchy

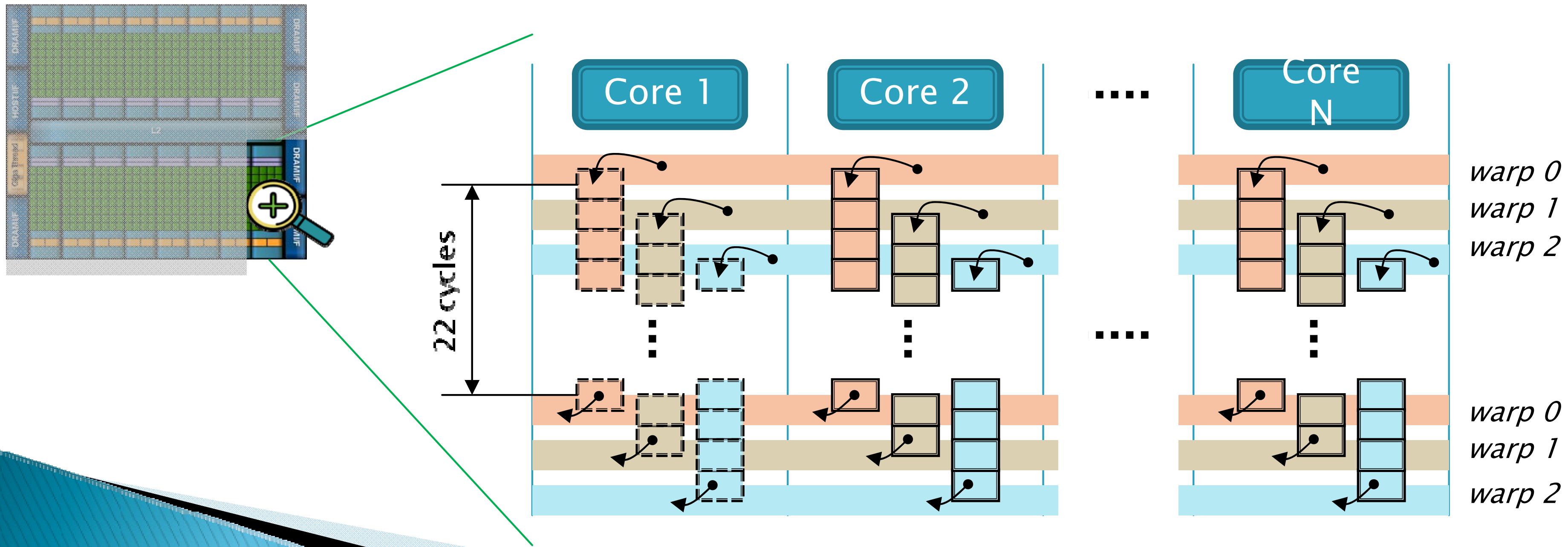
- Global GM memory
  - Large, slow, visible by threads on all SM processors
- Shared SM memory
  - Small, fast, visible by threads on one SM processor



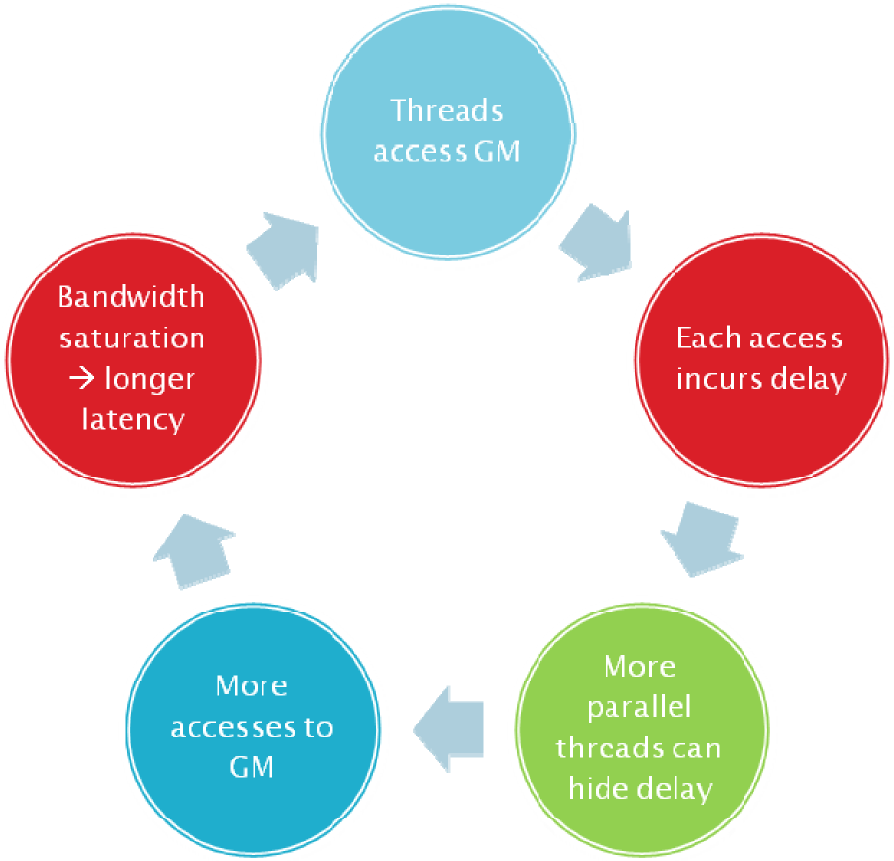
- Other memories, subset of GM
  - Local memory – for registers spilling
  - Texture memory – different access pattern

GPU internals

- Array of SIMD processors (SM processors)
  - Each handles a large pool of threads grouped in *warps*
- Interleaved execution for high throughput
  - operation latency (22 cycles)
  - memory latency (~ 400 cycles)

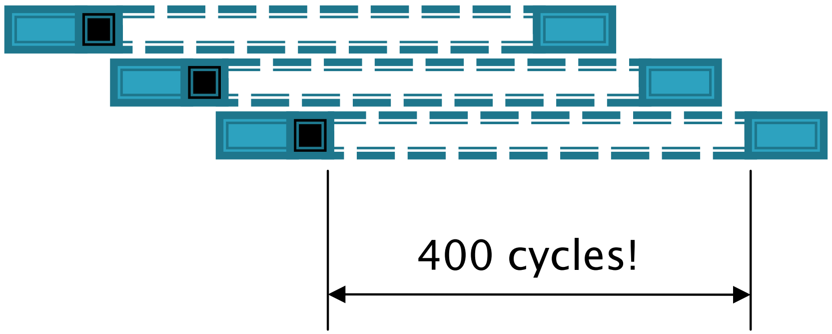


Memory access pattern



Vicious cycle

- Influenced by ratio of:
  - Memory access
  - Computation
- Bandwidth limitation



Specialization

- Workset: GM memory or SM memory?
  - GM → many parallel threads
    ➔ saturate bandwidth
  - SM → prefetching
    - Requires parallel loads / stores
    - SM memory size dictates number of parallel iterations
    ➔ prefetch rate linked to workset size

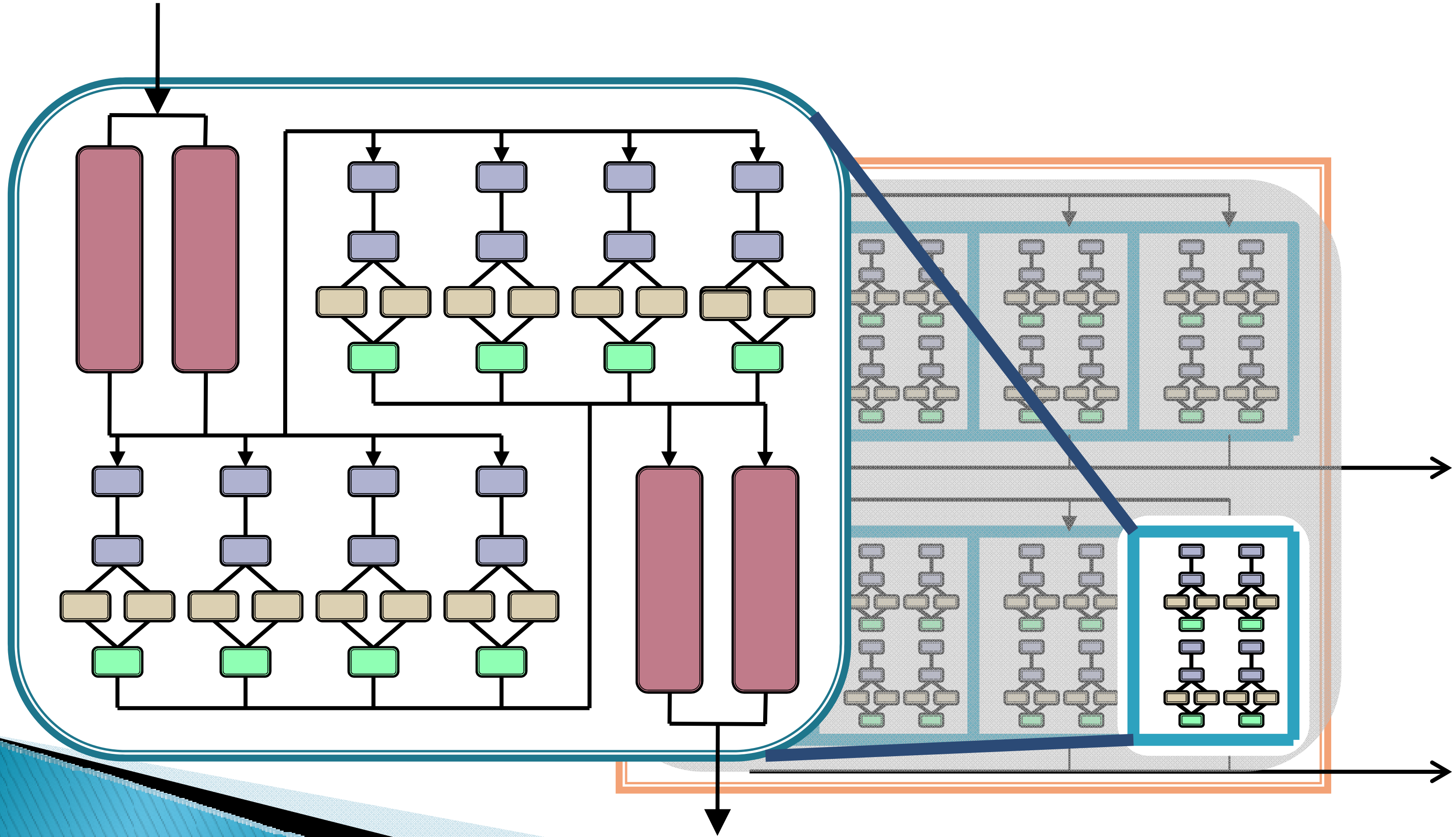
- Separate GM memory accesses from computation
  - Specialize warps
  - Use SM memory to cache the workset
  - Computation-only warps release the workset faster

Mapping strategy

Stream graph → Parallel instances of the entire graph

Novel memory access scheme

Utilize fine-grained parallelism



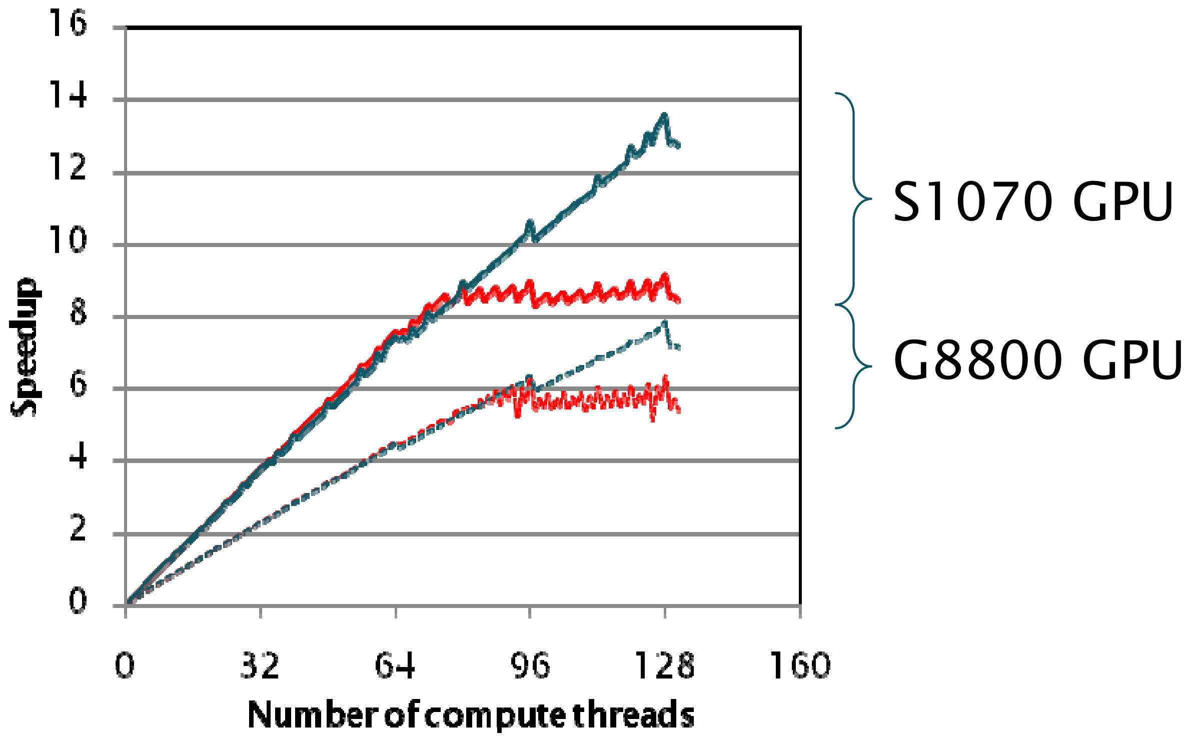
SIMT versus SIMD

- Misconception: Threads should not diverge
  - True only for threads belonging to a warp
- Warp:
  - SIMD execution model
  - Static thread allocation (based on thread ID)
- No penalty for this CUDA code:

```
if (threadIdx.x < warpSize) {
    compute_action();
} else if (threadIdx.x >= warpSize && threadIdx.x < 2 * warpSize)
    memory_access_action();
} else if …
```

Bandwidth limitation

- Insufficient memory access warps limit performance



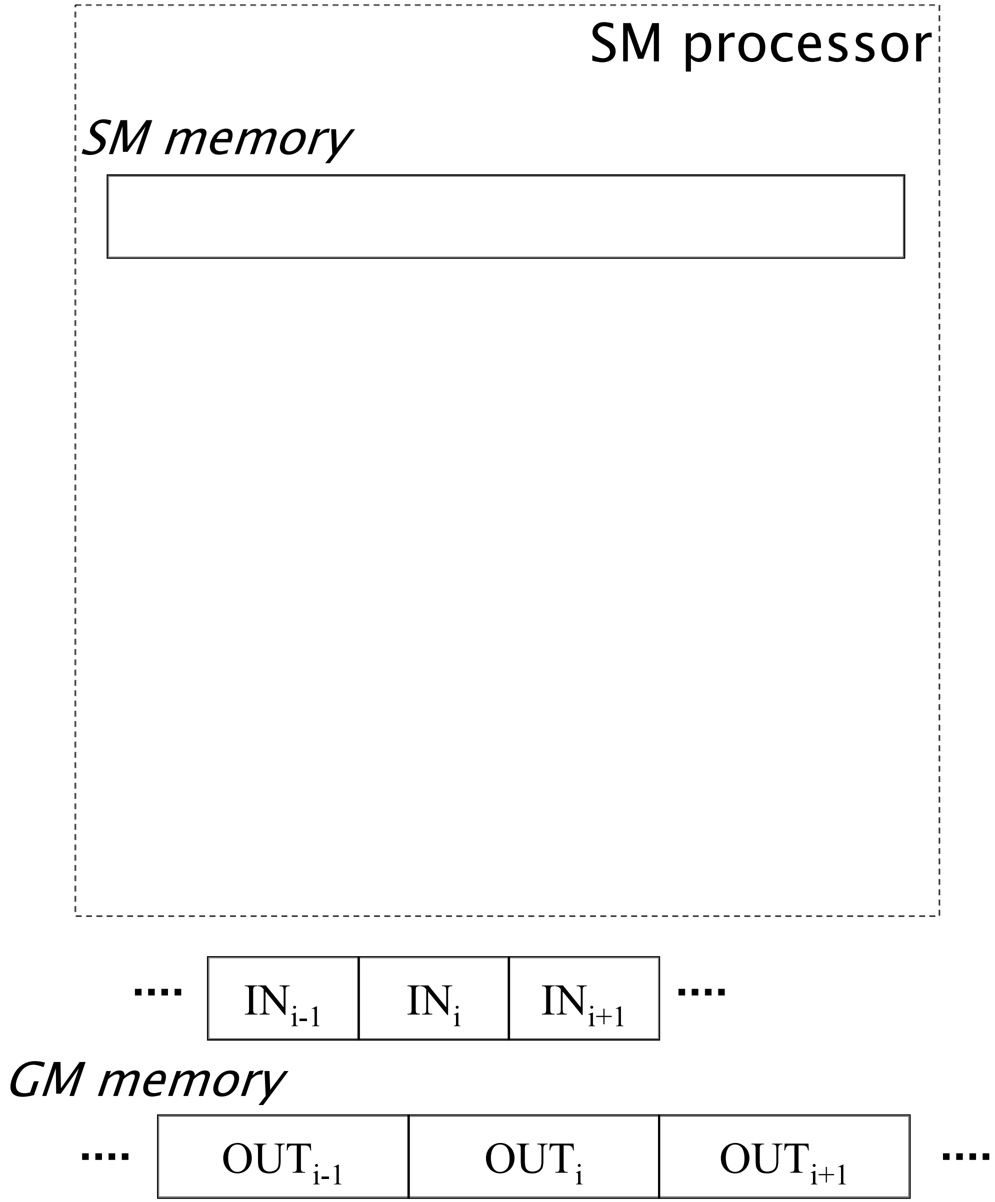
(Bitonic sort)

Insufficient to sustain required bandwidth

Additional warps for memory access

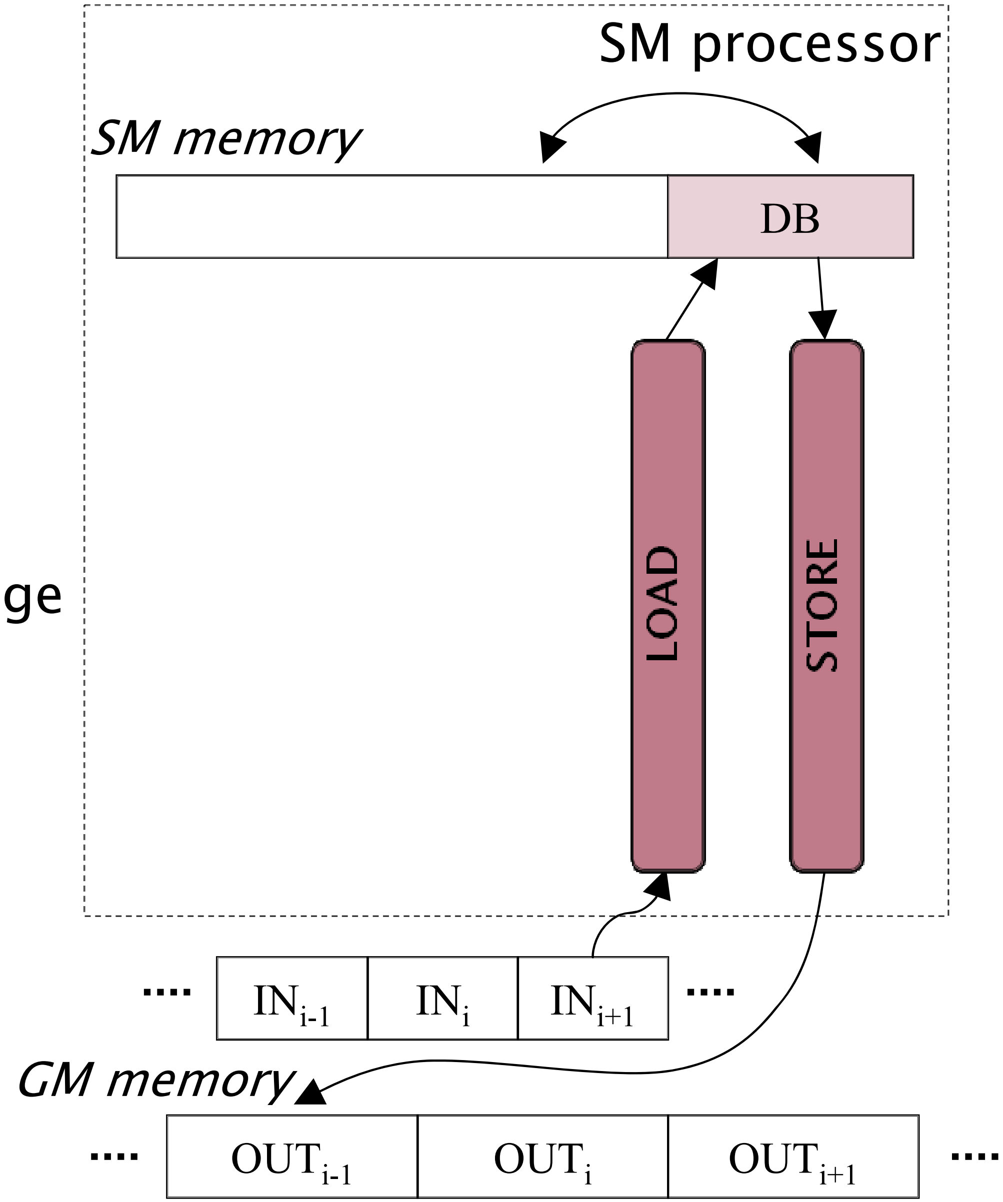
Data movement

- Software pipelining a group of iterations in each SM processor
- I/O streams in GM memory



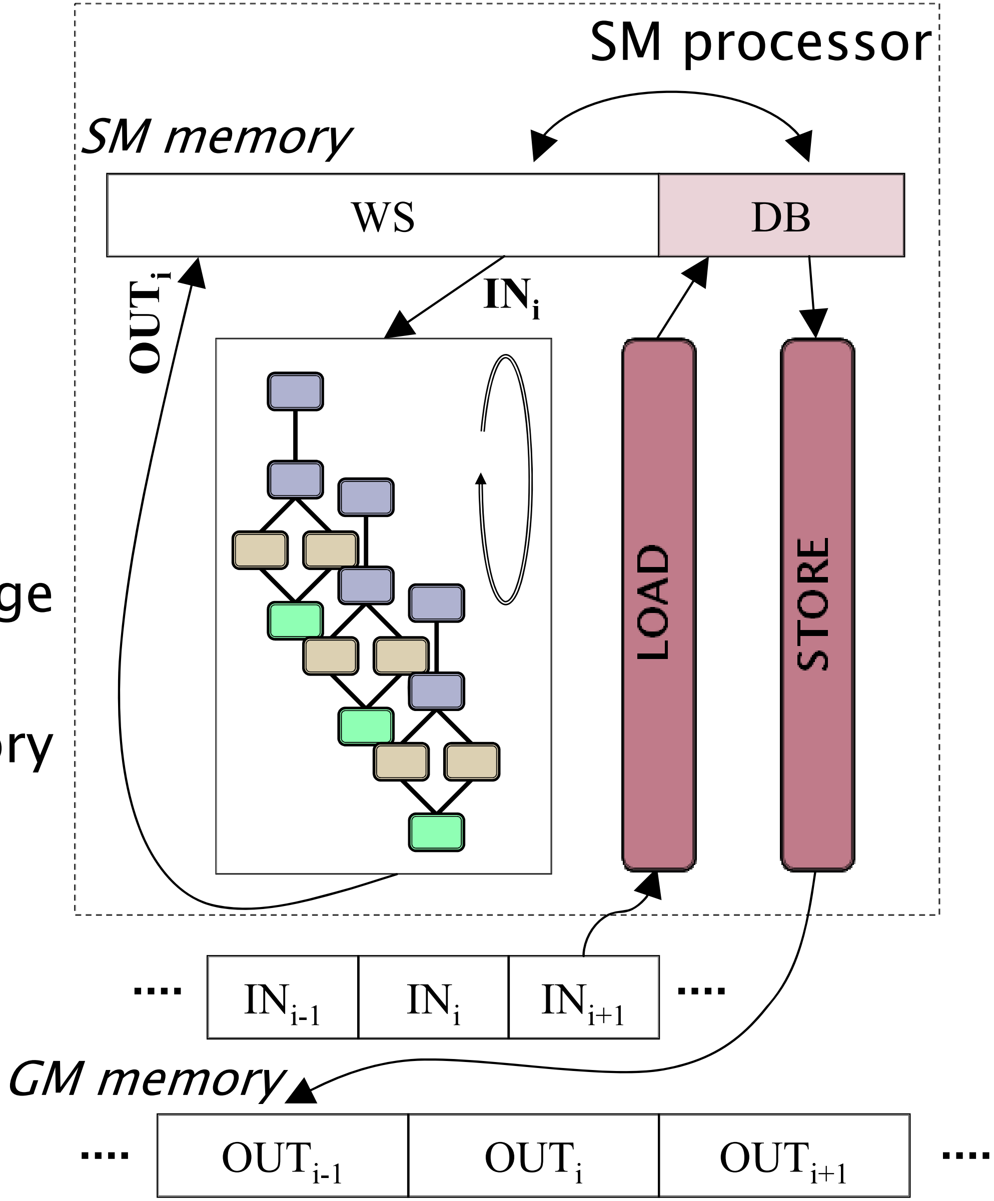
Data movement

- Software pipelining a group of iterations in each SM processor
- I/O streams in GM memory
- Double buffer (DB) for I/O exchange



Data movement

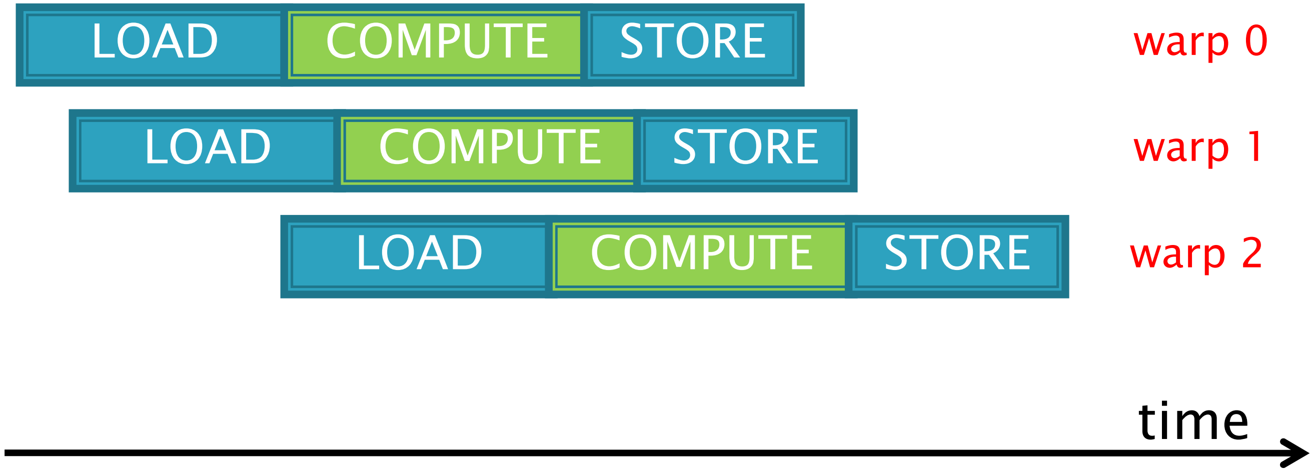
- Software pipelining a group of iterations in each SM processor
- I/O streams in GM memory
- Double buffer (DB) for I/O exchange
- Workset (WS) must fit in SM memory
  - I/O data
  - All intermediate stream data
  - ➔ Limited number of iterations



Prefetching vs Specialization

Prefetching

- 3x COMPUTE warps
- 3x WS memory



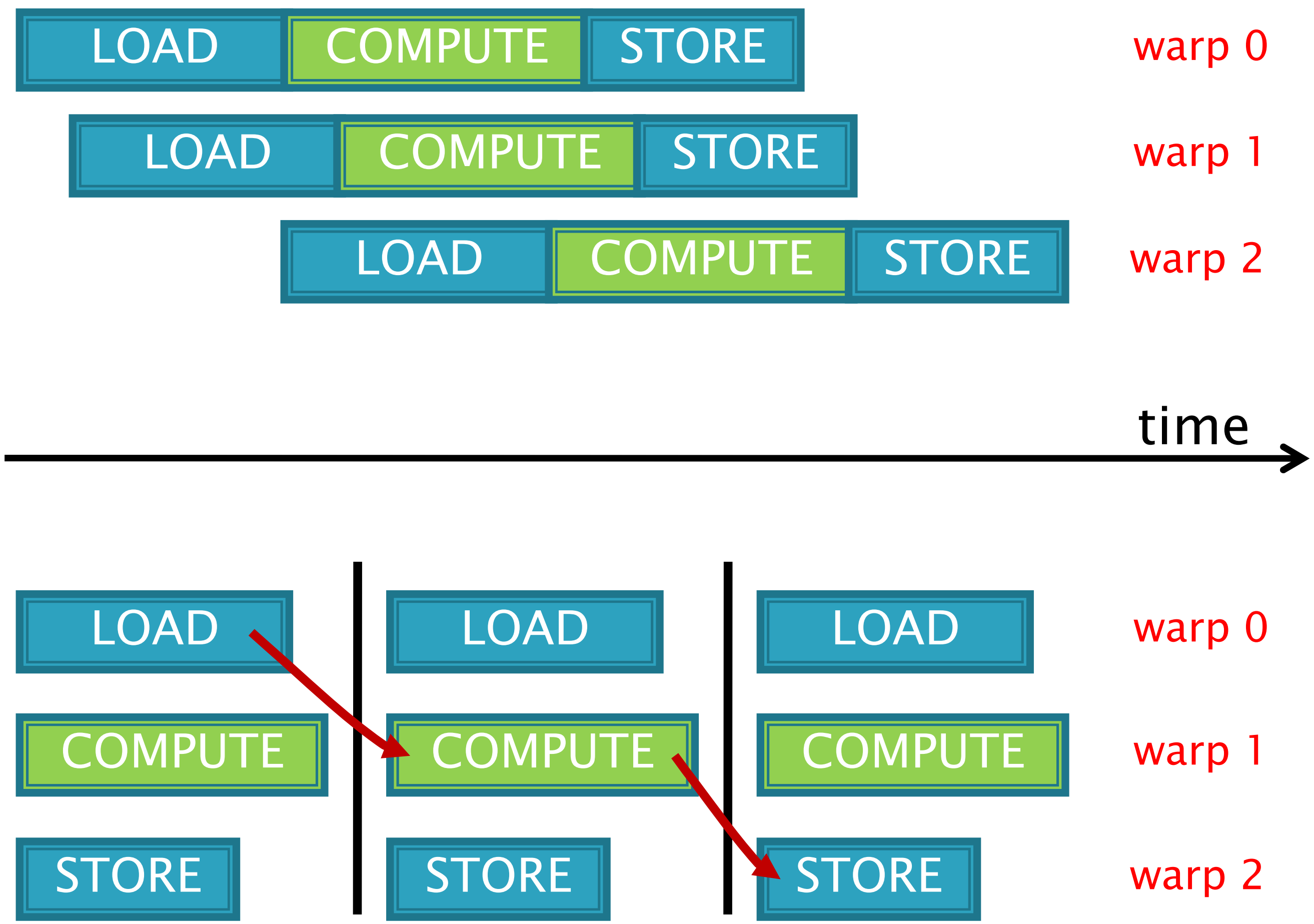
Prefetching vs Specialization

Prefetching

- 3x COMPUTE warps
- 3x WS memory

Specialization

- 1x COMPUTE warp
- 1x (WS+DB) memory

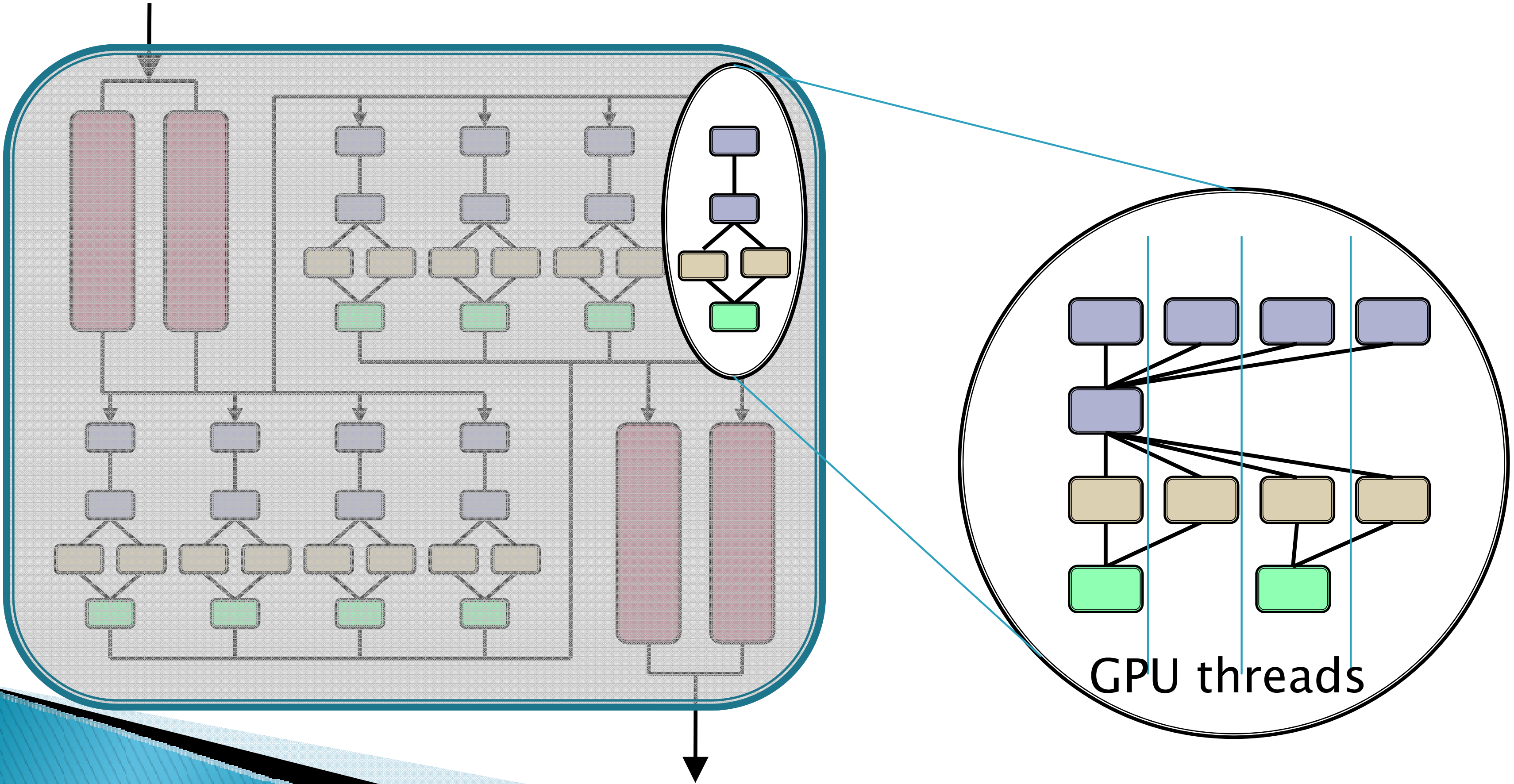


Mapping strategy

Stream graph → Parallel instances of the entire graph

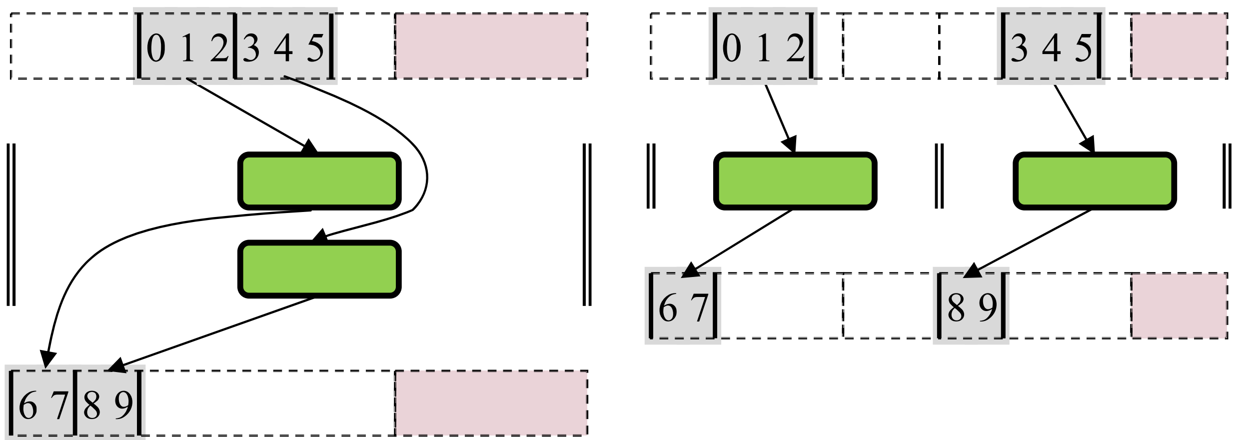
Novel memory access scheme

Utilize fine-grained parallelism

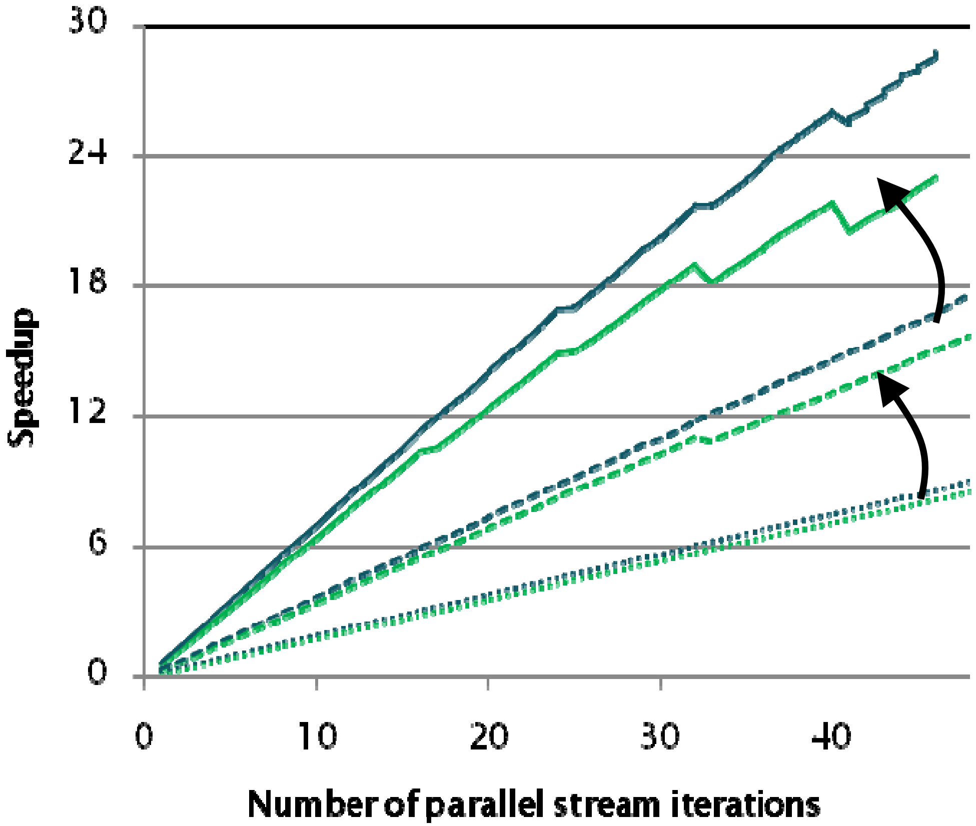


Fine grained parallelism

- Multiple threads / stream iteration
  - Distributed schedule
- Synchronization
  - Lock-step execution of warp threads



Design space characterization



4 threads / iteration

2 threads / iteration

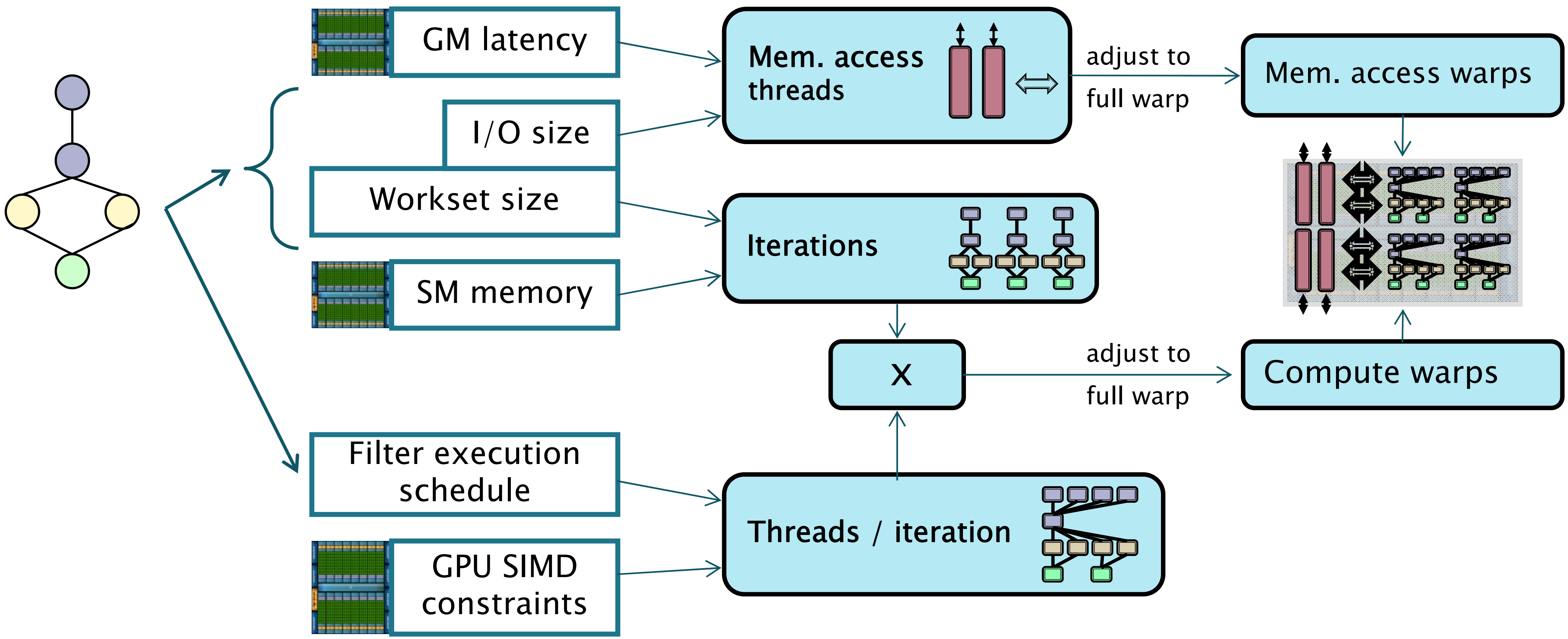
1 thread / iteration

(FilterBank)

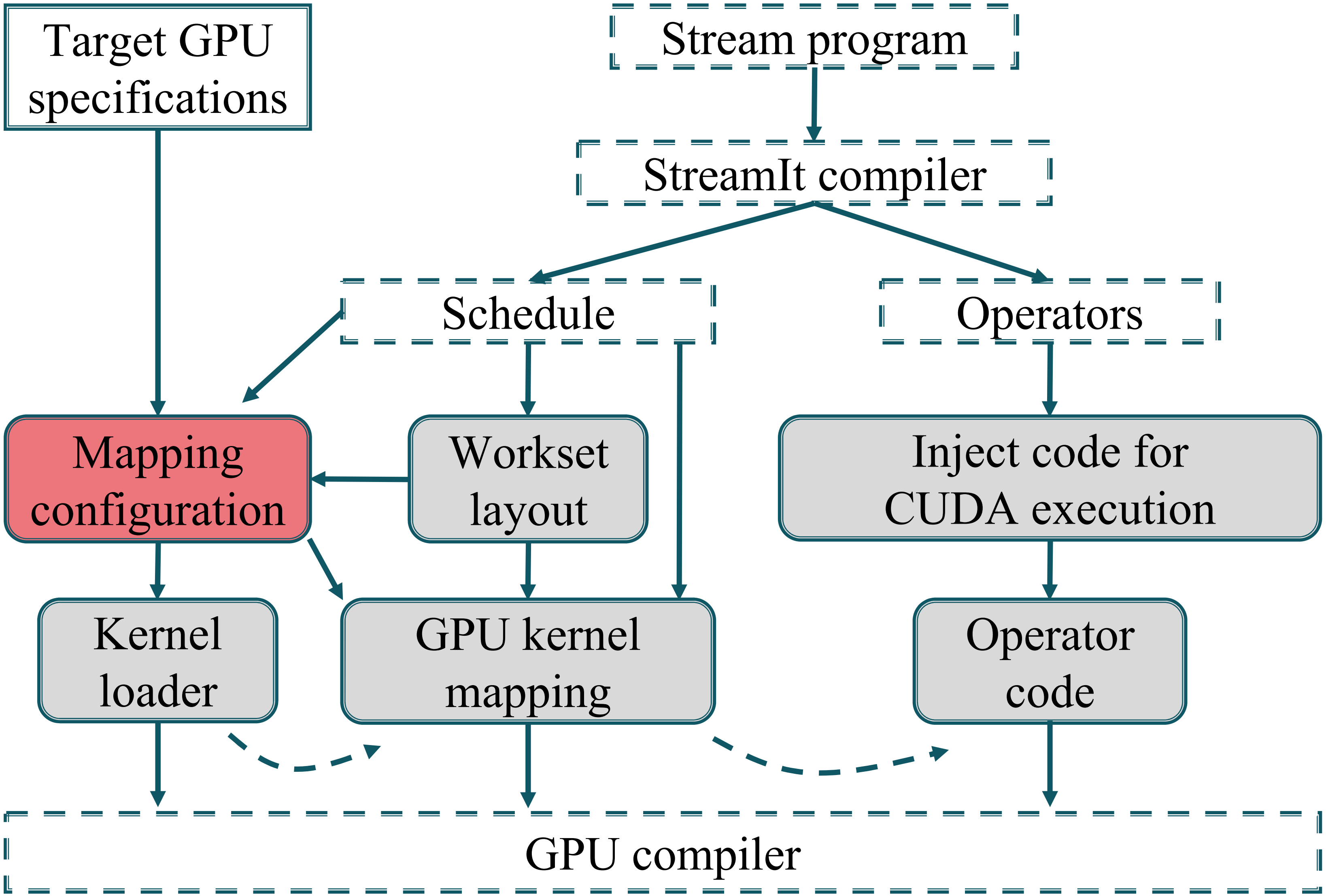
S1070 GPU

G8800 GPU

Design point selection

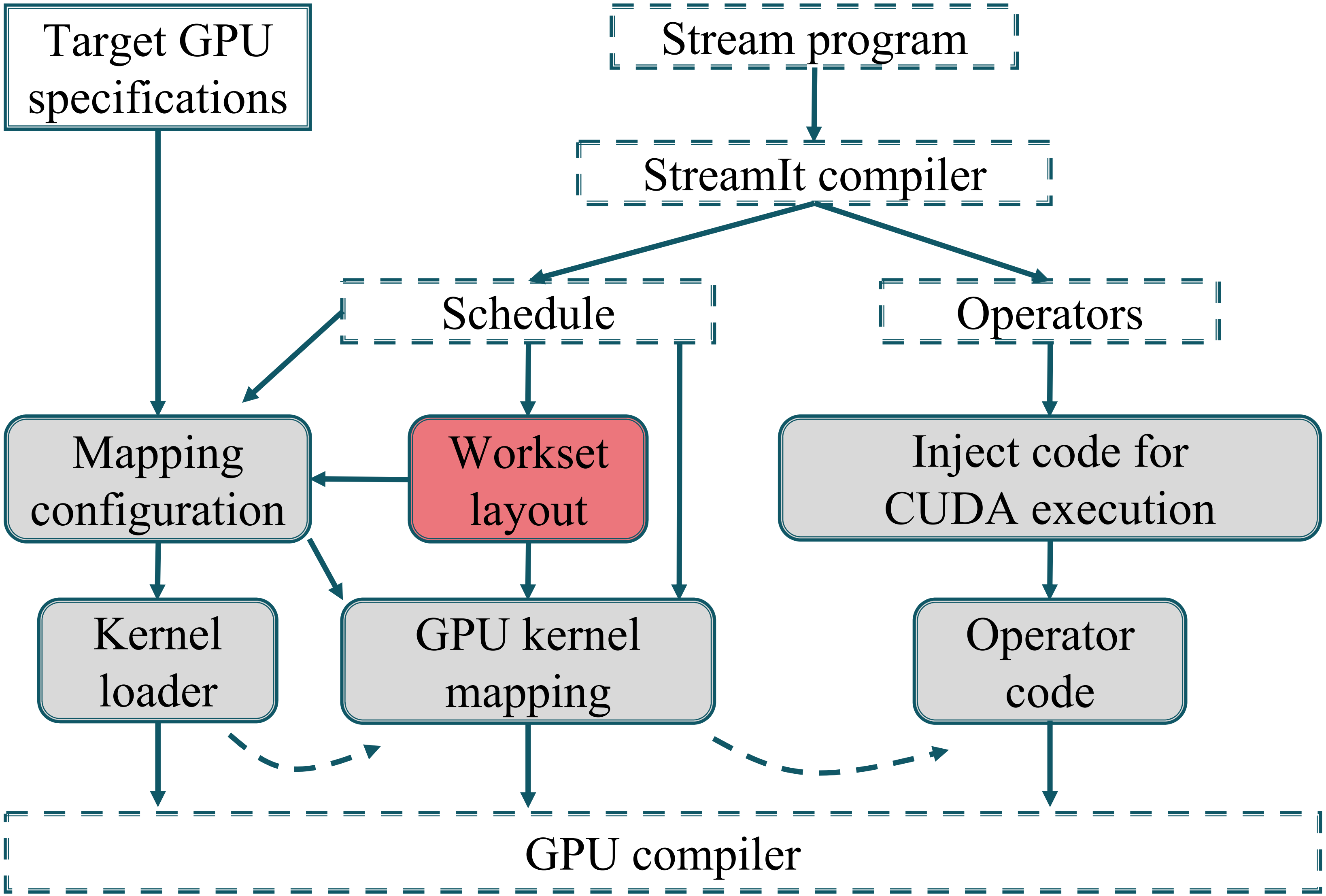


Flow



Target GPU specifications
Stream program
StreamIt compiler
Schedule
Operators
Mapping configuration
Workset layout
Inject code for CUDA execution
Kernel loader
GPU kernel mapping
Operator code
GPU compiler

Flow


Target GPU specifications
Stream program
StreamIt compiler
Schedule
Operators
Mapping configuration
Workset layout
Inject code for CUDA execution
Kernel loader
GPU kernel mapping
Operator code
GPU compiler

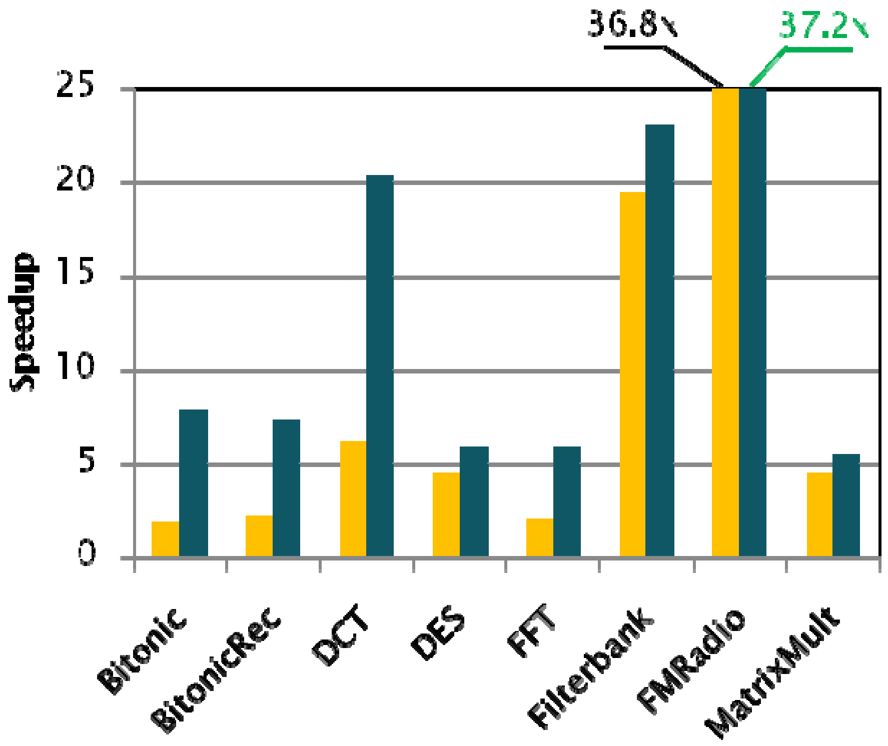
Workset layout

- Fragmentation of workset allocation
  - Small buffers are required between filters
    - Liveness analysis → estimation of workset size
    - Fragmentation → actual allocation may lead to a slight increase
- Coalesced memory access

Peeking

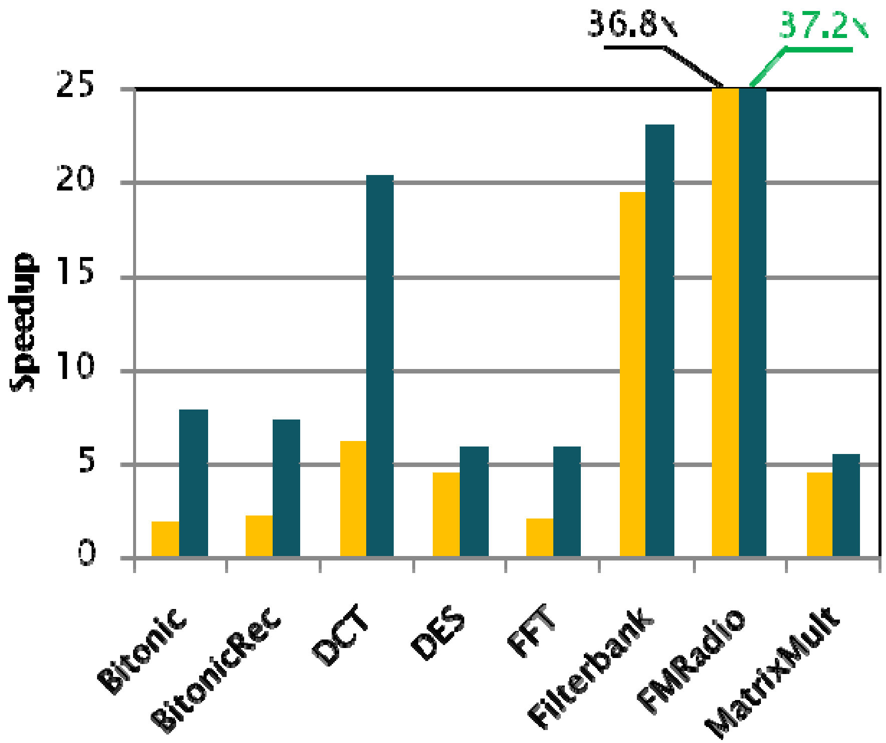
- Inter-iteration dependencies
- Overlap input data to reconstruct the initial elements
  - For each SM processor
  - For each parallel thread group
- Intuition:
  - Warm-up intermediate buffers
  - Threads access previous iterations
- Custom synchronization
  - Only between compute threads
  - Implemented custom barrier

Results

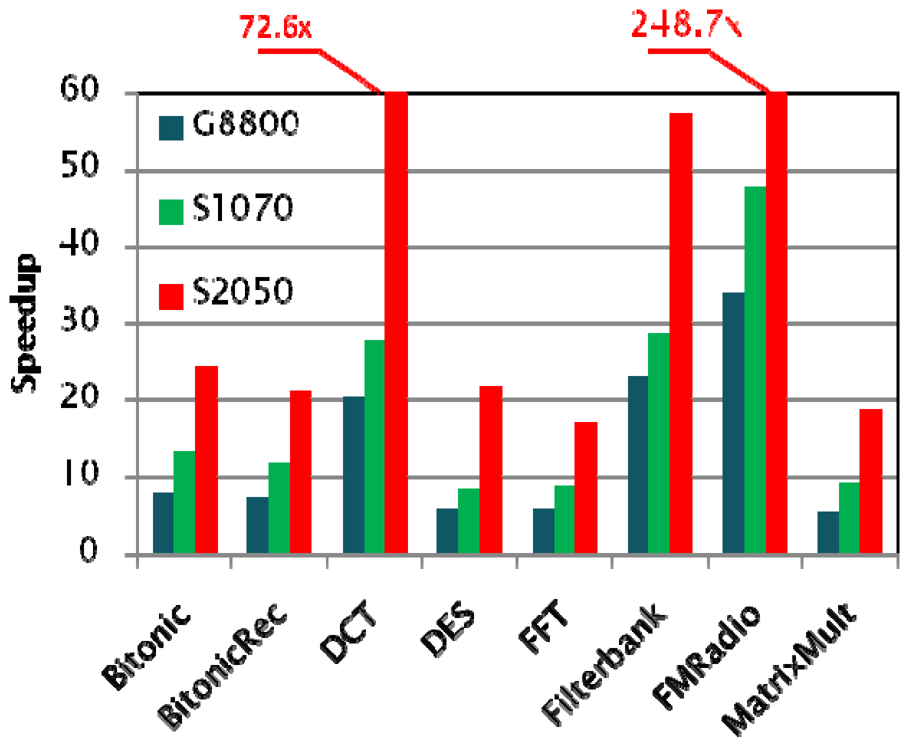


versus CGO 2009

Results



versus CGO 2009



Different GPU architectures

Conclusions

- Novel scheme to execute stream graphs on GPU
- Automatic heuristic for selecting efficient design points
- Novel memory access scheme through software pipelining

Thank you

- Questions?