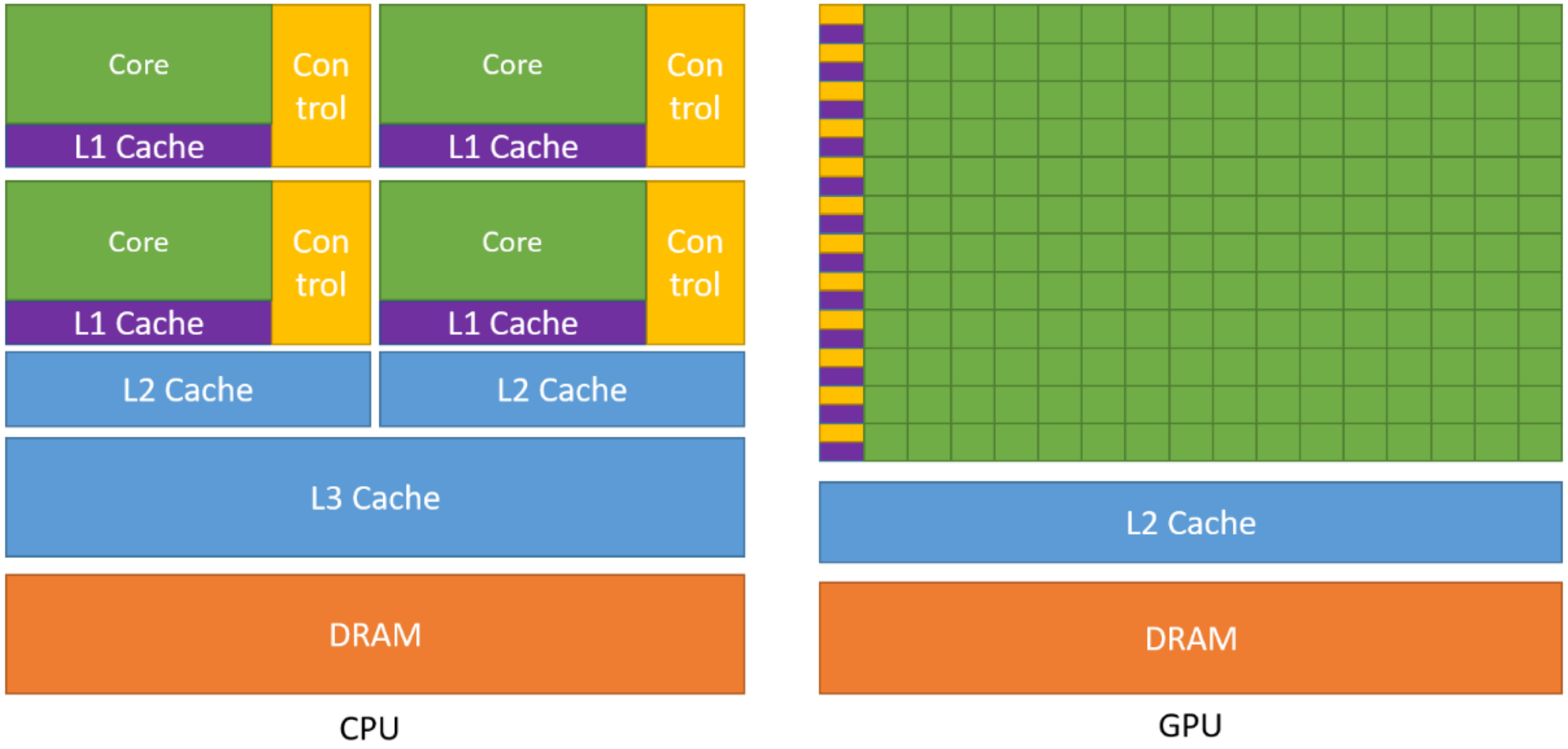
Compiling Graph Applications for GPUs with GraphIt

Ajay Brahmakshatriya, Yunming Zhang, Changwan Hong, Shoaib Kamil,
Julian Shun, Saman Amarasinghe

CPU vs. GPU

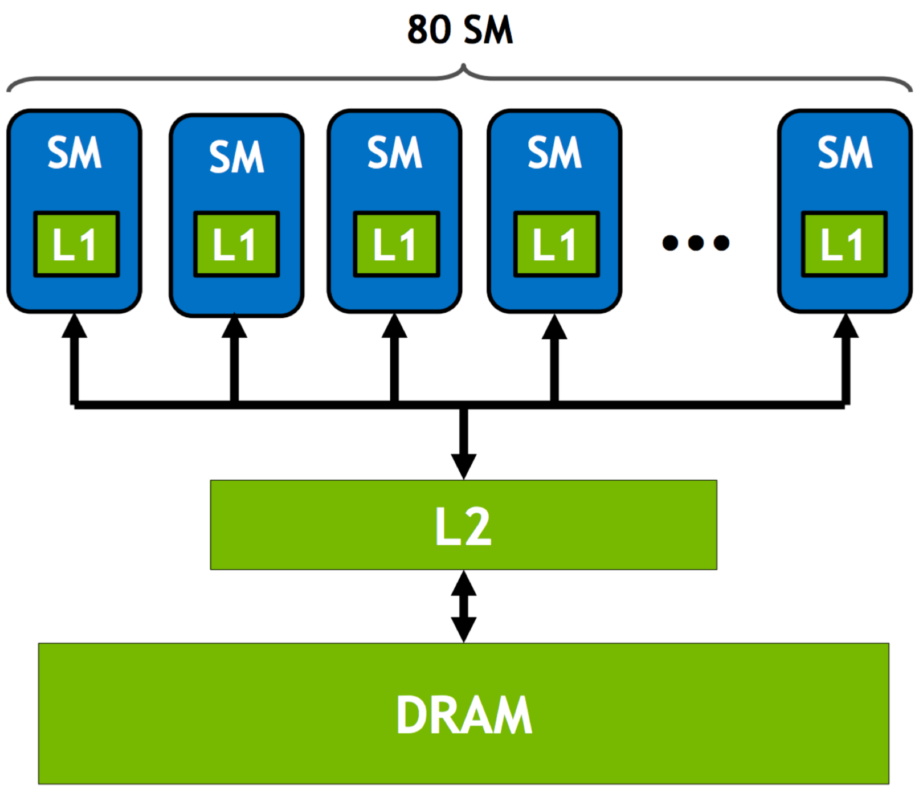


Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

- CPU is designed to execute one thread fast
- GPU is designed to execute many (slower) threads in parallel, achieving higher throughput

GPU Architecture

- GPU has multiple streaming multiprocessors (SMs)
  - For example, Tesla V100 has 80 SMs
- Work in GPUs is organized into thread blocks (CTAs), and dynamically assigned to SMs
- Each SM has its own data cache, which can be partitioned between L1 cache and shared memory
  - 128KB on Tesla V100
- There is global DRAM (16 or 32 GB on V100) and a global L2 cache (6144 KB on V100)



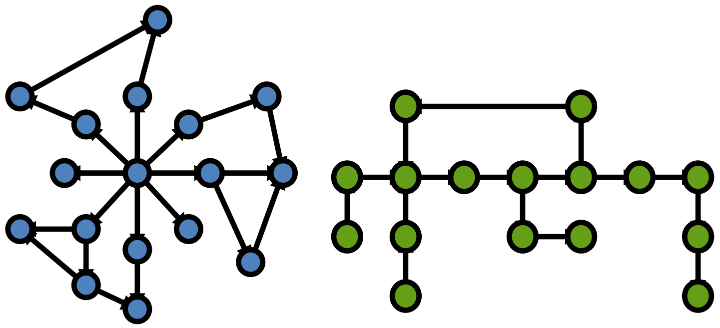
Source: https://on-demand.gputechconf.com/gtc/2018/presentation/s81006-volta-architecture-and-performance-optimization.pdf

GPU Architecture

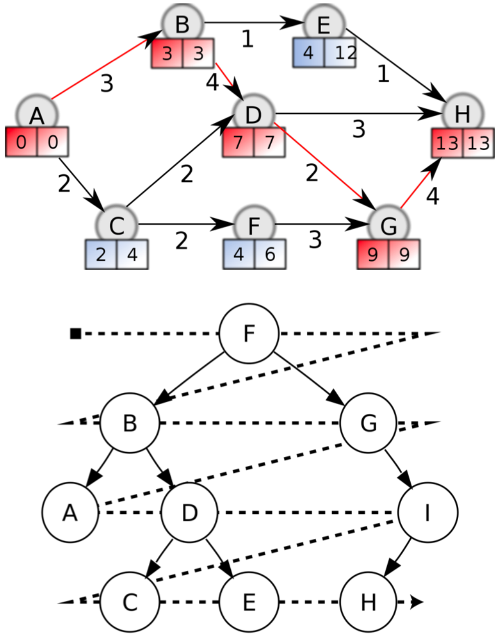
- Each SM schedules warps (groups of 32 threads) to execute the same computation together
  - Tesla V100 has 4 warp schedulers and instruction units, and can execute 4 warps at a time

- Each instruction unit has its own cores for arithmetic, and L0 instruction cache

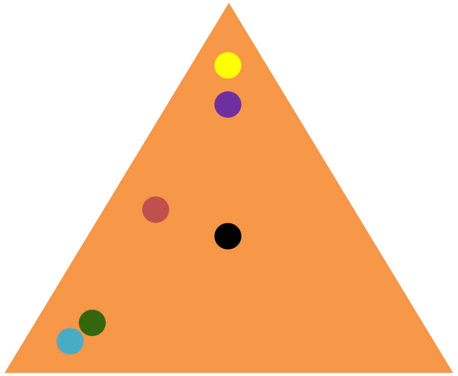
Graph Optimization Tradeoff Space



Graphs



Algorithms



Optimizations

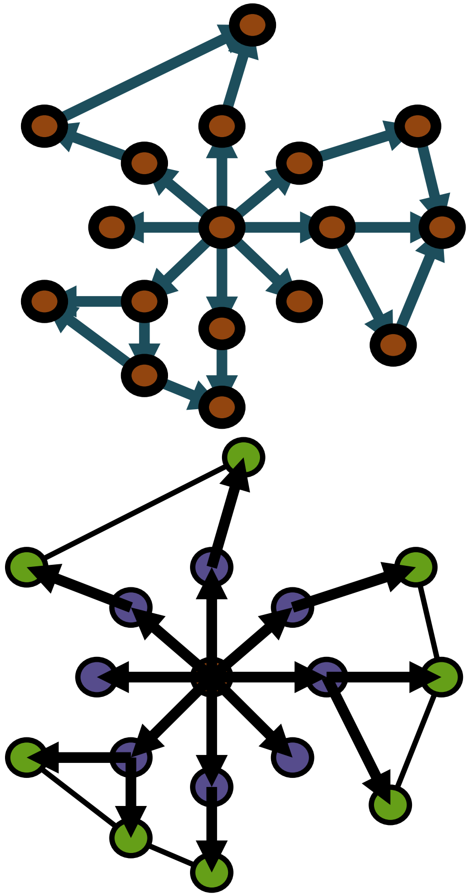
Hardware

GraphIt – A Domain-Specific Language for Graph Analytics

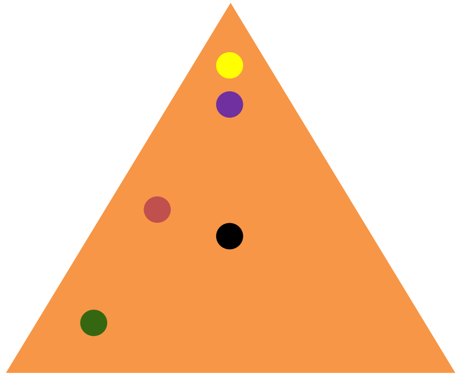
- Decouple Algorithm from Optimization
  - Algorithm language: What to compute
  - Scheduling (optimization) language: How to compute

- Scheduling representation
  - Easy to use for users to try different combinations of optimizations without changing the algorithm

GraphIt – A Domain-Specific Language for Graph Analytics



Algorithm Language



Optimization Representation

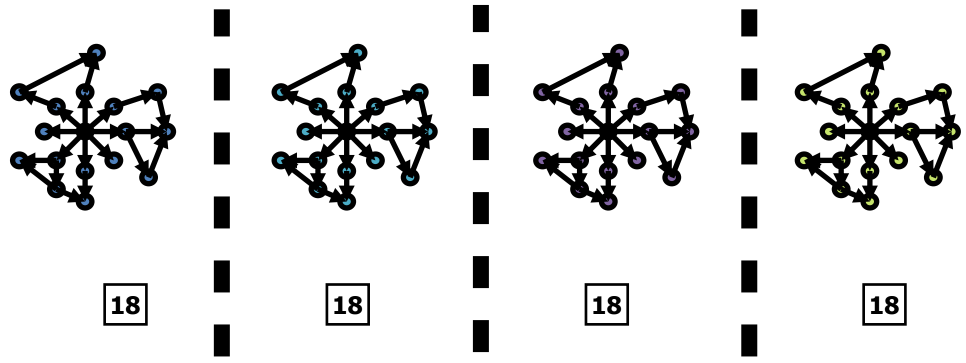
- Scheduling Language
- Schedule Representation (e.g., Graph Iteration Space)

Autotuner

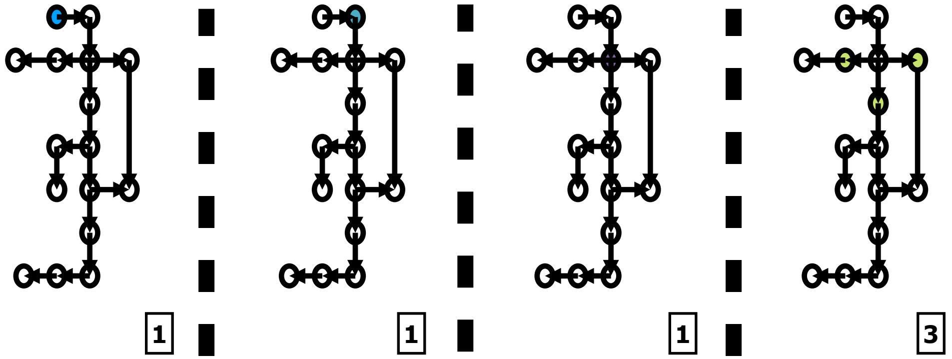
Hardware Variations?

PageRank on social networks

Vertices Processed



Vertices Processed



SSSP on road graphs

GPU

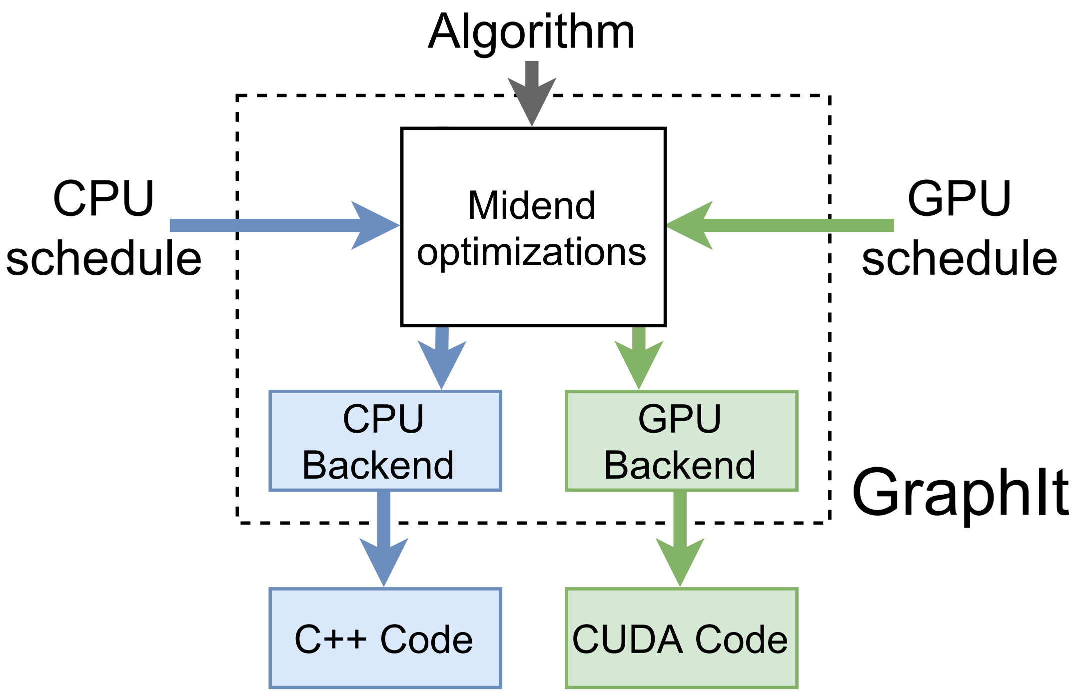
- Massive parallelism
- Coalesced memory accesses

CPU

- Limited parallelism
- Large number of iterations

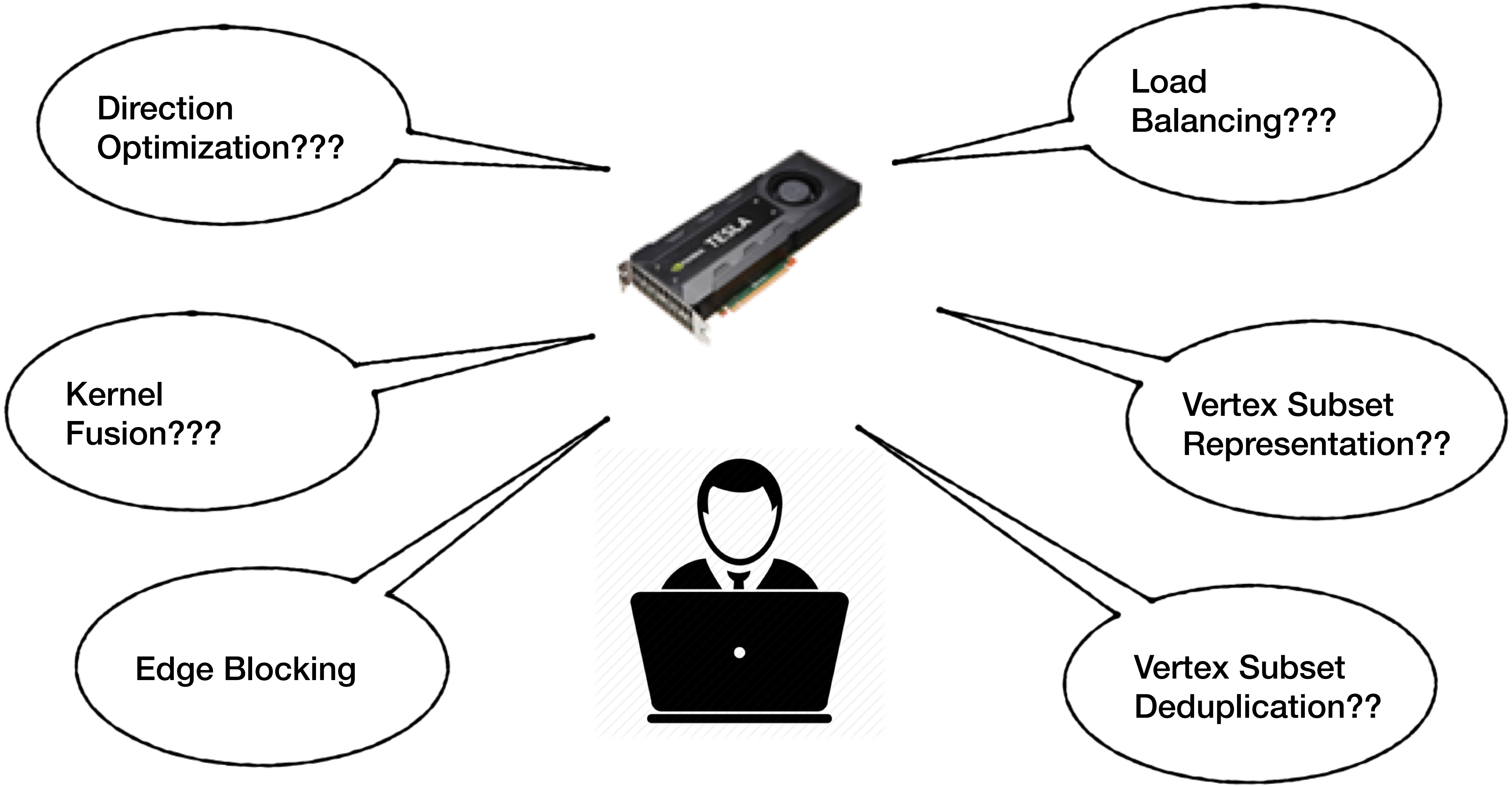
- No existing framework for generating both CPU and GPU code prior to GraphIt

GraphIt Backend Support



- GraphIt currently has backends for multicore CPUs and GPUs
- First framework to support code generation for both CPUs and GPUs with the same algorithm specification

Key GPU Optimizations



GPU Scheduling Language

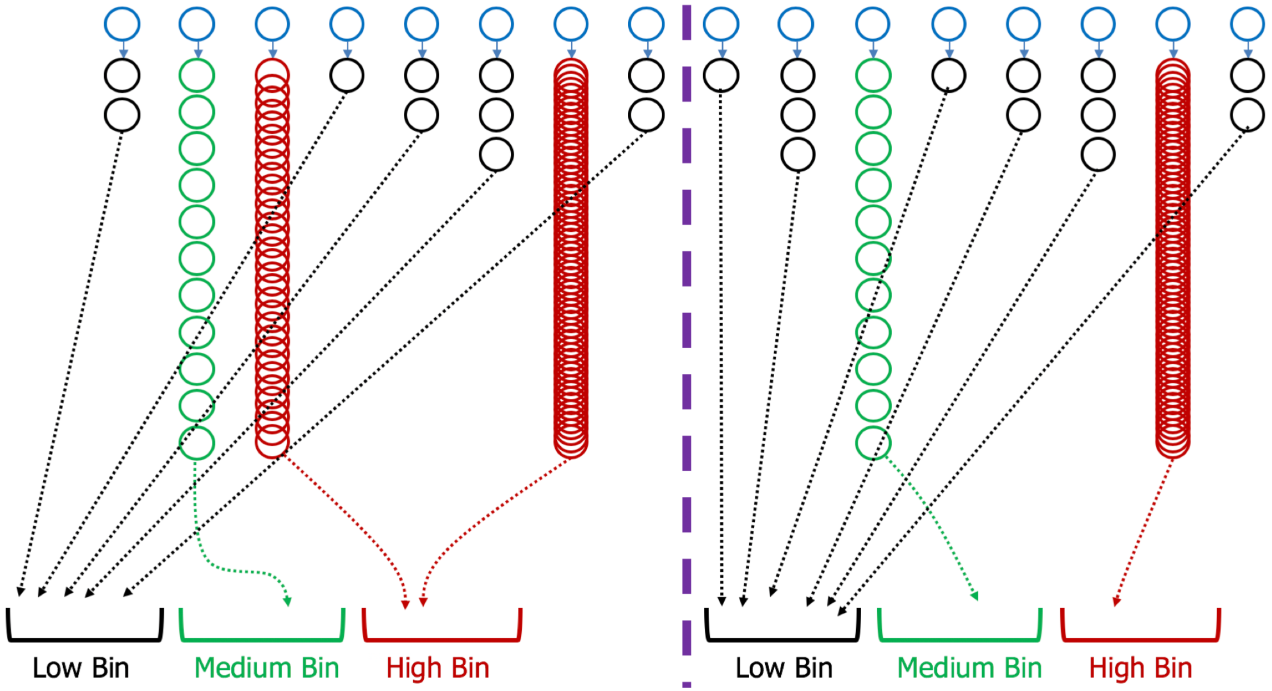
- Load Balancing Strategy — `configLoadBalance(CM|WM|TWC|TWCE|EB|VB|STRICT)`
- Iteration direction — `configDirection(PUSH|PULL)`
- Representation of output frontier — `configFrontierCreation(FUSED|UNFUSED, BITMAP|BOOLMAP)`
- Deduplication of output frontier — `configDeduplication(ENABLED|DISABLE, BITMAP|BOOLMAP|MONOTONIC_COUNTERS)`
- Fusing multiple CUDA kernels — `configKernelFusion(ENABLED|DISABLED)`
- Graph partitioning for cache utilization — `configEdgeBlocking(ENABLED|DISABLED)`
- Runtime combinations of above — `HybridGPUSchedule`

Comparison of Optimizations with Other Frameworks

Optimization	Gunrock	GSWITCH	SEP-Graph	GraphIt
Load Balancing	VERTEX BASED, EDGE BASED, TWC	CM, WM, TWC, STRICT	VERTEX BASED	ETWC, TWC, STRICT, CM, WM, VERTEX BASED, EDGE BASED
Edge Blocking	Not supported	Not supported	Not supported	Supported
Vertex Set Creation	Fused/Unfused	Fused/Unfused	Fused	Sparse Queue / Bitmap / Boolean Array
Direction Optimization	Push/Pull/Hybrid	Push/Pull/Hybrid	Push/Pull/Hybrid	Push/Pull/Hybrid
Deduplication	Supported	Not supported	Supported	Supported
Vertex Ordering	Supported	Supported	Supported	Supported
Kernel Fusion	Supported	Not supported	Supported	Supported
Total combinations	48	32	16	**576**

New Optimization: ETWC Load Balancing

- Edge-based Thread Warps CTAs load-balancing (ETWC)
  - First, equally partitions vertices across CTAs
  - Then, partitions edges of a vertex into low, medium, high bins to be processed by a thread, warp, and entire CTA, respectively
  - Trades off load balance for reducing load balancing overhead

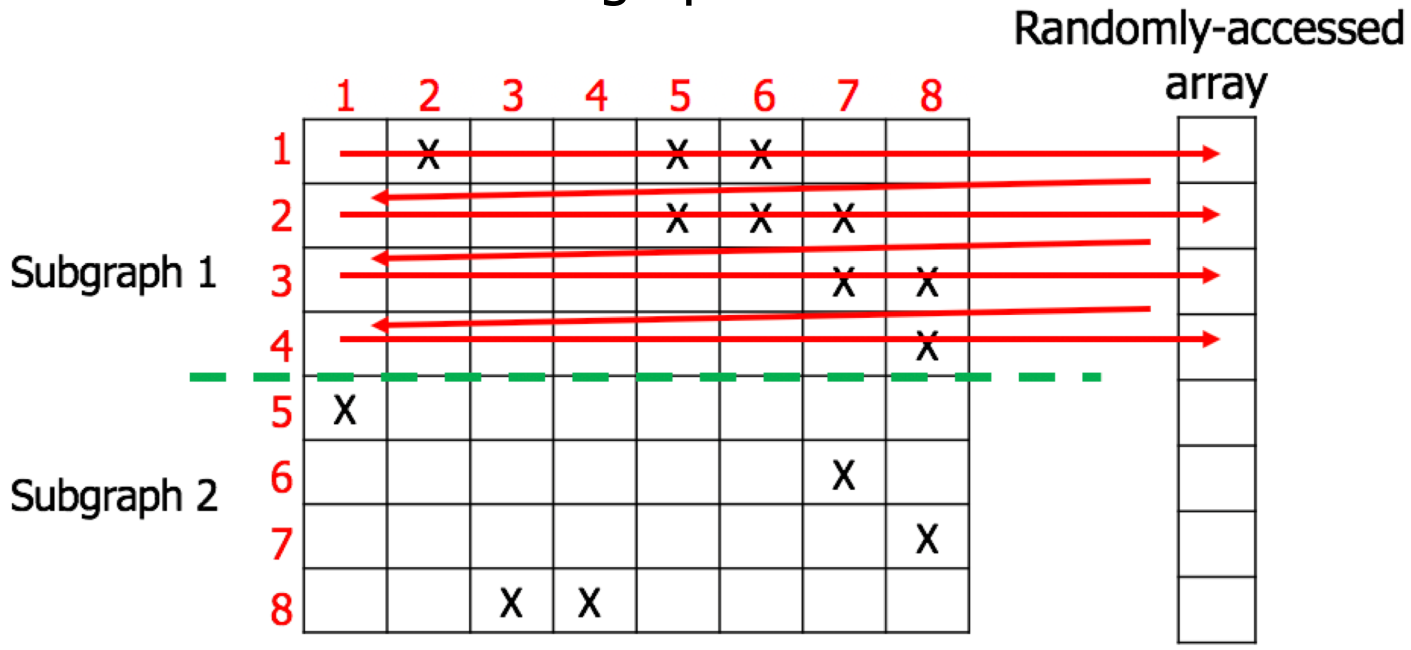


Graph	ETWC	TWC	CM
OK	43.58	**40.69**	42.24
TW	**106.11**	107.57	116.06
LJ	19.72	20.03	**18.42**
SW	**226.35**	230.00	230.03
HW	**4.94**	5.79	8.17
IC	**11.38**	11.50	22.16
RU	**136.64**	255.89	168.90
RC	**91.20**	162.54	109.89
RN	**13.10**	25.77	16.25

Times (ms) of ETWC on breadth-first search, compared with existing strategies TWC and CM. Fastest time is **bolded**.

New Optimization: Edge Blocking

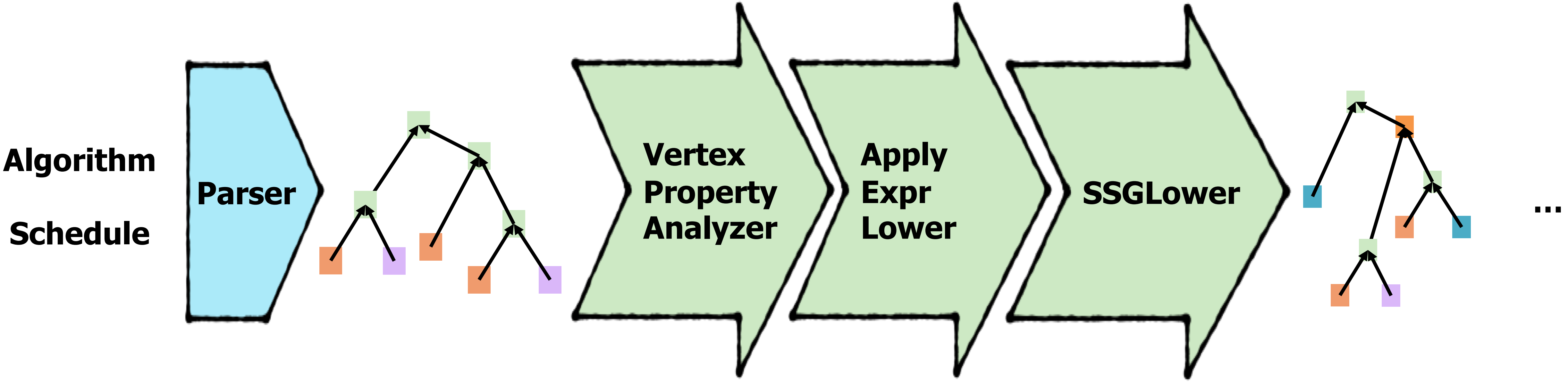
- Edge blocking (EB)
  - Tiles edges into subgraphs such that the random accesses for each subgraph fit in L2 cache
  - Process each subgraph one at a time



Graph	Without EB	With EB	Speedup
OK	41.75	**14.18**	2.94x
TW	88.25	**77.86**	1.13x
LJ	15.67	**7.68**	2.04x
SW	144.88	**102.11**	1.41x
HW	**7.01**	7.02	0.99x
IC	**18.24**	19.55	0.93x
RU	8.35	**6.32**	1.35x
RC	8.39	**5.56**	1.50x
RN	0.44	**0.43**	1.02x

Times (ms) and speedup of Edge blocking (EB) on PageRank

GraphIt Compilation



- Whole program analysis / transformations

Example: Frontier Reuse Analysis

```
...
while (frontier.getVertexSetSize() != 0)
    output = edges.from(frontier).to(toFilter).
       applyModified(updateEdge, parent);
    delete frontier;
    frontier = output;
  end
...
```

```
while (frontier.getVertexSetSize() != 0) {
    cudaMalloc(output, ...);
    ApplyModified<<<, >>>(frontier, output, …);
    ...
    cudaFree(frontier);
    frontier = output;
}
```

Allocations and freeing on GPUs are costly unlike CPUs

- Do we really need to allocate and free on every round?
- We should reuse the memory that was allocated for `frontier`
- Is it always safe to do so?
  - What if the old frontier is used again?

Liveness Analysis!

Example: Frontier Reuse Analysis

- Constructs live ranges for each Vertexsubset variable (used to represent frontiers)

```
...
while (frontier.getVertexSetSize() != 0)
    output = edges.from(frontier)…applyModified(…);
    delete frontier;
    frontier = output;
end
...
```

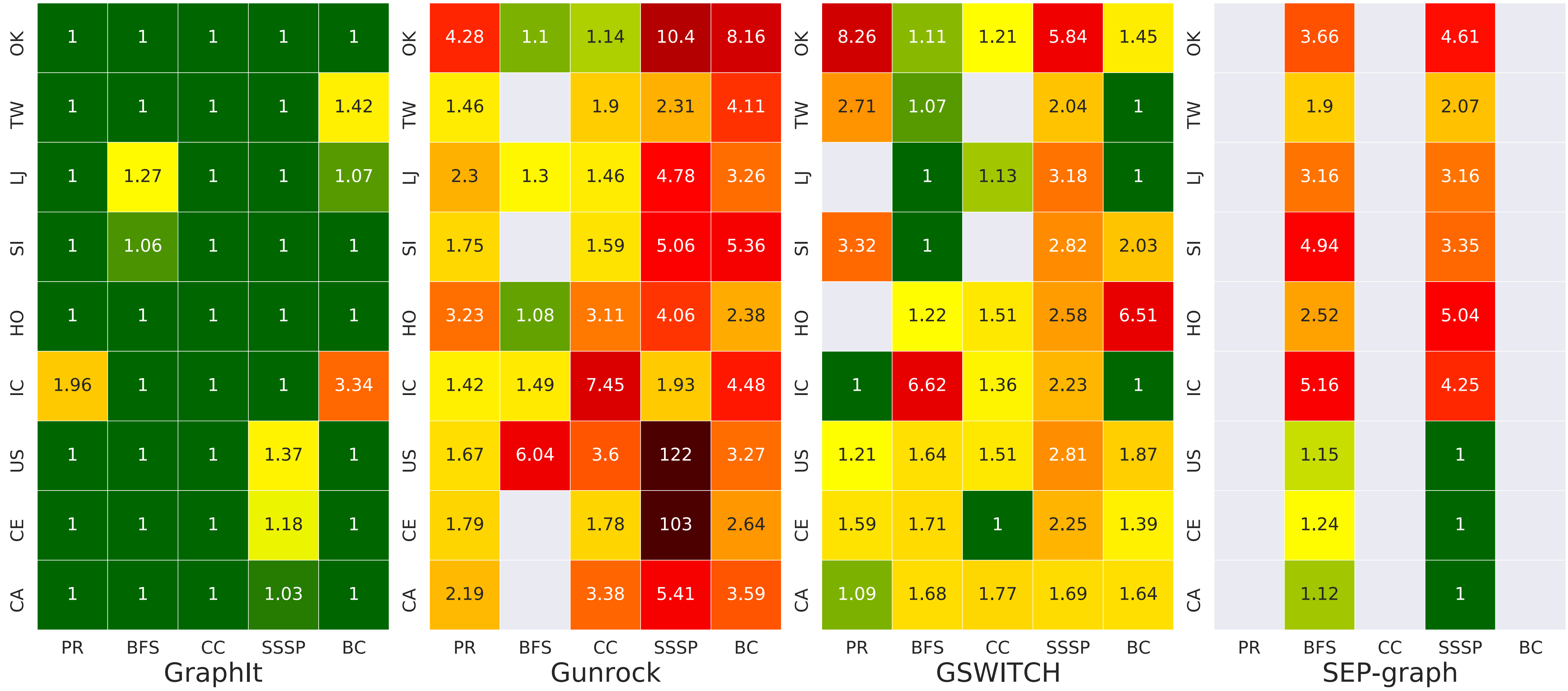
- Disjoint live ranges allows memory to be reused

Experimental Evaluation

- Compare to 3 state-of-the-art frameworks (Gunrock [Wang et al. 2017], GSWITCH [Meng et al. 2019], SEP-graph [Wang et al. 2019])
- 5 algorithms: breadth-first search, single-source shortest paths (Delta-stepping), connected components, betweenness centrality, and PageRank
- 9 datasets: social networks, Web graphs, and road networks
- 2 generations of NVIDIA GPUs: Pascal and Volta

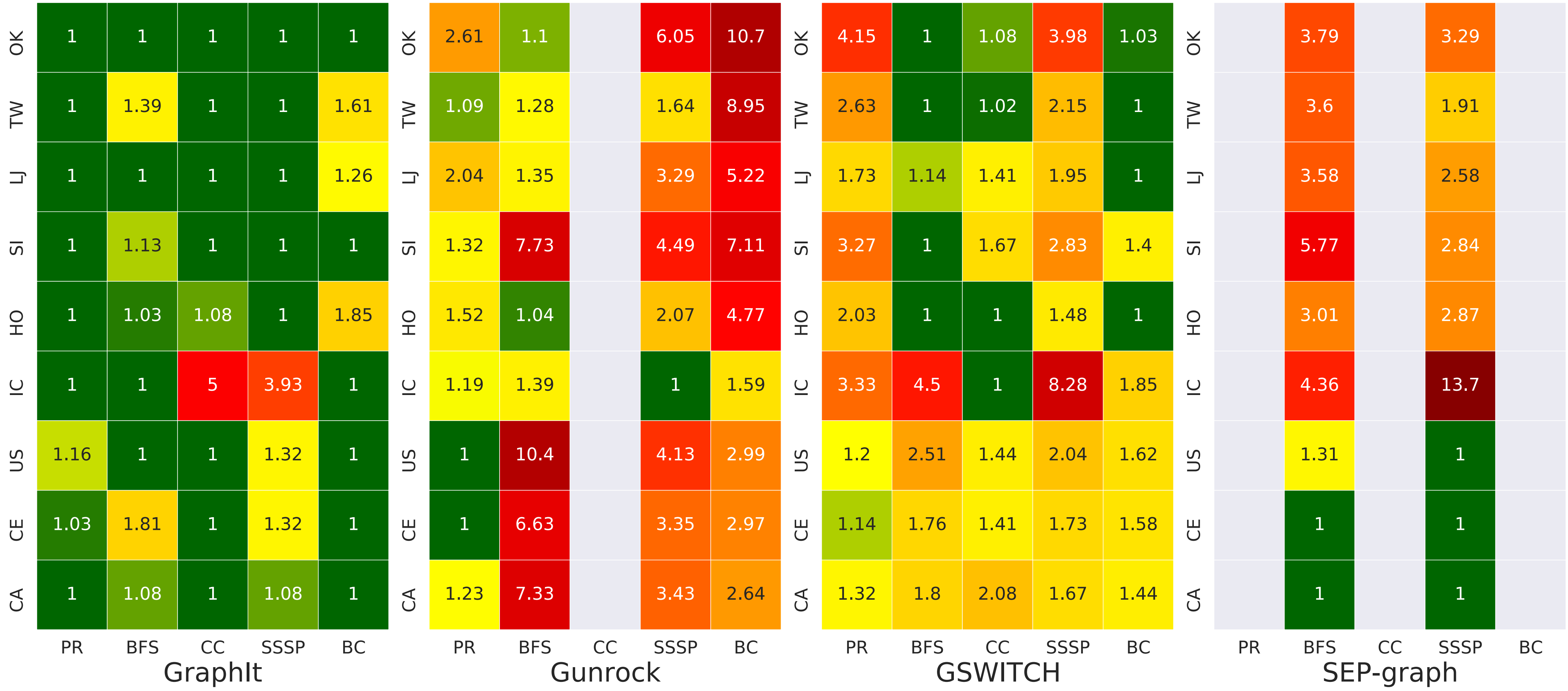
State of the Art and GraphIt on Titan Xp (Pascal) GPU

Slowdowns relative to the fastest implementation



State of the Art and GraphIt on V-100 (Volta) GPU

Slowdowns relative to the fastest implementation



GraphIt achieves a speedup of up to 5.11x due to searching through a much larger space of optimizations

Programmability

- Lines of code for each algorithm in each framework

Algorithm	Gunrock	GSWITCH	SEP-Graph	**GraphIt (Algorithm+Schedule)**
Breadth First Search	2189	164	481	**66**
PageRank	2207	159	-	**61**
Connected Components	3014	160	-	**62**
Betweenness Centrality	1792	280	-	**128**
SSSP with Delta Stepping	1438	203	473	**50**

CPU vs. GPU

- Compared GPU implementations with CPU implementations in GraphIt on a 24-core machine
  - PageRank, BFS, betweenness centrality, and connected components were faster on the GPU
  - Delta-Stepping on road graphs was faster on the CPU
  - CPUs can process much larger graphs

- It is critical to be able to choose between CPU and GPU for each application!

Summary

- GraphIt DSL and compiler to generate high-performance CPU and GPU code from the same high-level algorithm representation
- New GPU-specific scheduling language options and optimizations
- The GPU algorithms from GraphIt outperform state-of-the-art GPU frameworks while requiring fewer lines of code
- Open source: https://graphit-lang.org