MICRO 2016

Graphicionado

A High-Performance and Energy-Efficient Graph Analytics Accelerator

Tae Jun Ham
Lisa Wu
Narayanan Sundaram
Nadathur Satish
Margaret Martonosi

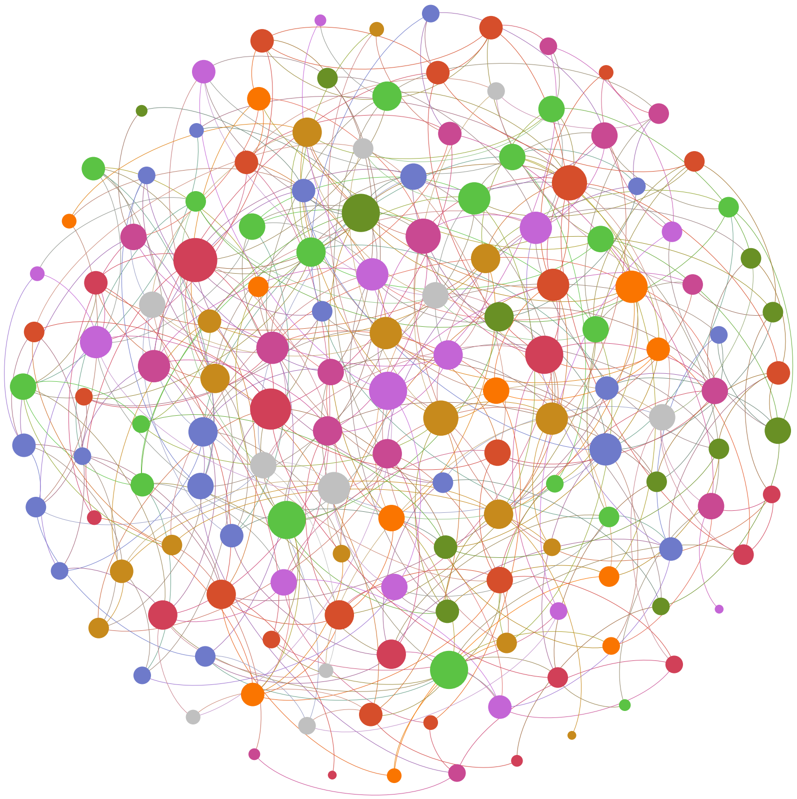
PRINCETON UNIVERSITY
intel

Slide: http://tiny.cc/graphicionado

Graph Analytics
– A key workload of modern-day computing

- Application Domains
  - Social Networks, Bioinformatics, Healthcare, Cybersecurity, Simulation, etc.

- Algorithms
  - Path Analysis (e.g., BFS, SSSP)
  - Centrality Analysis (e.g., PageRank, Betweenness Centrality)
  - Recommendation systems (e.g., Collaborative Filtering)
  - Community Analysis



Graph Analytics is difficult to code

- Graph Analytics has …
  - Irregular data access patterns
  - Extremely low computation-to-communication ratio
  - Poor spatial/temporal data locality
  - Difficult-to-extract parallelism
  - Memory-bound performance
- Solution: Software Graph Processing Frameworks
  - Programmers express graph algorithms using a programming abstraction (i.e., vertex program)
  - Frameworks orchestrate data movements for given computation using efficient backend SW

Industry

Google	Pregel	facebook	Extended Giraph
intel	GraphMat	NVIDIA	nvGraph
ORACLE	PGX	IBM	System G
Microsoft	Graph Engine	(turi create intelligence)	GraphLab Create

Academia

Pegasus (ICDM 09)	TurboGraph (KDD 13)
Galois (SOSP 13)	GraphX (OSDI 14)
Ligra (PPoPP 13)	Gunrock (PPoPP 16)
X-Stream (SOSP 13)	. . .

Software Frameworks have limitations

= Insufficient tailoring to HW

- Application-oblivious memory system in conventional processors
  - Caches blindly cache all data at a coarse granularity including ...
    - Low temporal locality data
    - Low spatial locality data (i.e., only 4B out of 64B is utilized)

Our measurement shows that SW frameworks consume 2x-27x more memory bandwidth than the optimal communication case

- Expensive data movement in conventional processors
  - Graph analytics has extremely low computation-to-communication ratio

Most algorithms: < 6% of instructions are used for actual computation; 94% used for data movements; results in huge energy consumption.

Graphicionado Approach

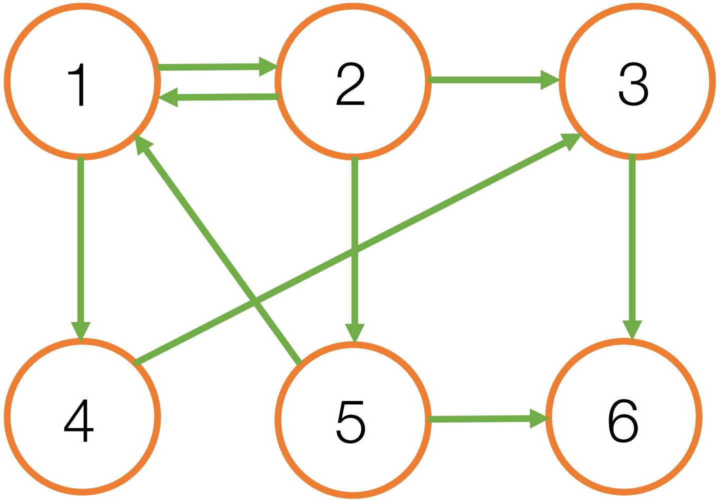
Graphicionado: A high-performance, energy-efficient graph analytics HW accelerator which overcomes the limitations of software frameworks while retaining the programmability benefit of SW frameworks

- Retains Programmability
  - Specialized-while-flexible HW pipeline
- Overcomes Limitation
  - Application-specific memory system
  - Application-specific pipeline design

Presentation Outline

- Why Graphicionado?
- Vertex Programming Abstraction
- Constructing Graphicionado Pipeline from Abstraction
- Optimizing Graphicionado
  - Optimizing Data Movement
  - Scaling On-Chip Memory Usage
  - Parallelization
- Performance/Energy Evaluation
- Conclusions

Graph Data Structure



- Graph consists of vertices & edges
  - Each Vertex has an ID and and a property (or state)
  - Each Edge is defined as a tuple (SRC ID, DST ID, edge property)

- Graph analytics: an iterative computation to calculate the desired vertex property for each vertex

Vertex Programming Abstraction

Processing Phase

```
For each active Vertex U
    For each outgoing edge E(U,V)
        Res = Process_Edge ($E_{weight}$, $U_{prop}$)
        $V_{temp}$ = Reduce ($V_{temp}$, Res)
    End
End
```

Apply Phase

```
For each Vertex V
    $V_{prop}$ = Apply($V_{temp}$, $V_{prop}$)
End
```

Process_Edge
Reduce
Apply

- Vertex programming abstraction is commonly used in SW frameworks (e.g., Intel GraphMat, Google Pregel, etc.)
    - Programmers express graph analytics algorithm with three custom functions
- Graphicionado uses the same abstraction to retain the programmability of SW frameworks

Vertex Programming Abstraction

Processing Phase

```
For each active Vertex U
    For each outgoing edge E(U,V)
        Res = $E_{weight} + U_{prop}$ ——— Process_Edge
        $V_{temp}$ = min($V_{temp}$, Res) ——— Reduce
    End
End
```

Apply Phase

```
For each Vertex V
    $V_{prop}$ = min($V_{temp}$, $V_{prop}$) ——— Apply
End
```

V_{prop}: minimum distance to V on last iteration

V_{temp}: minimum distance to V on this iteration

Single Source Shortest Path (SSSP) Example

Process_Edge: $E_{weight} + U_{prop}$ Reduce: min(V_{temp}, Res) Apply: min(V_{temp}, V_{prop})

Vertex Programming Abstraction

Processing Phase

```
For each active Vertex U
  For each outgoing edge E(U,V)
    Res = $E_{weight} + U_{prop}$
    $V_{temp}$ = min($V_{temp}$, Res)
  End
End
```

Calculate a distance to V through this edge

Update the current minimum distance to V

Apply Phase

```
For each Vertex V
  $V_{prop}$ = min($V_{temp}$, $V_{prop}$)
End
```

Compare the current minimum distance with the value from the last iteration; If it changes its value, add V to the active vertex set for the next iteration

V_{prop}: minimum distance to V on last iteration

V_{temp}: minimum distance to V on this iteration

Single Source Shortest Path (SSSP) Example

Process_Edge: $E_{weight} + U_{prop}$ Reduce: min(V_{temp}, Res) Apply: min(V_{temp}, V_{prop})

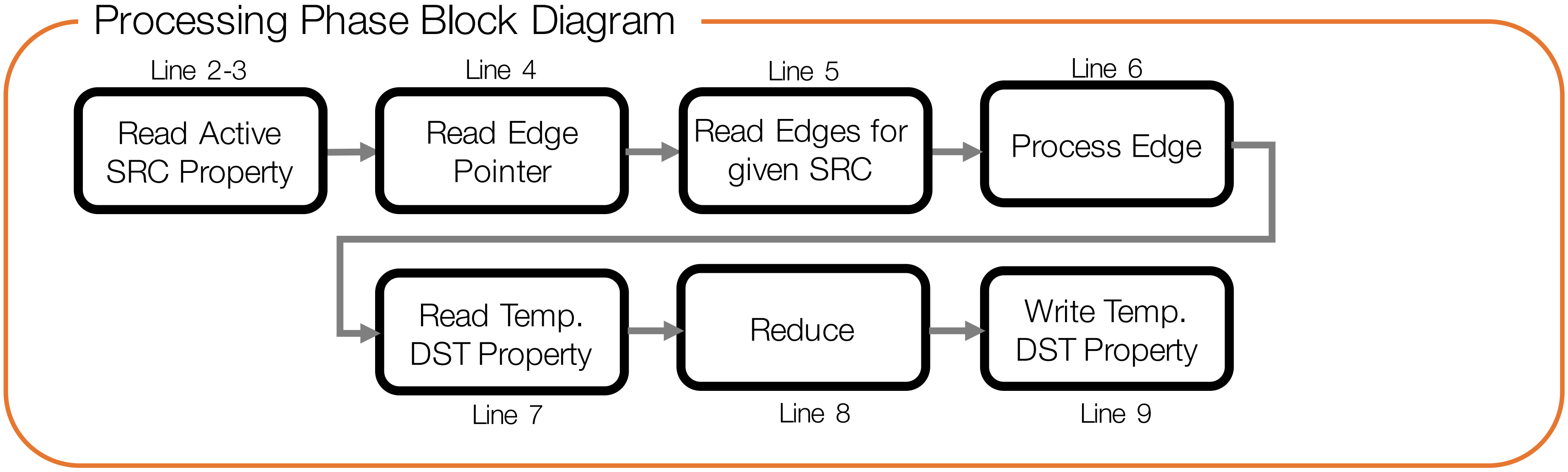
From abstraction to HW

```
 1 for (i=0; i<ActiveVertex.size(); i++) {
 2   vid = ActiveVertexID[i];
 3   vprop = ActiveVertexProp[i];
 4   eptr = PtrToEdgeList[vid];
 5   for (e = Edges[eptr]; e.src == vid; e = Edges[++eptr]) {
 6     res = Process_Edge(e.weight, vprop);
 7     temp = TempVertexProp[e.dst];
 8     temp = Reduce(temp, res);
 9     TempVertexProp[e.dst] = temp;
10   }
11 } // Apply Phase updates ActiveVertexProp with TempVertexProp
```

Read the active (SRC) vertex

Traversing edges of the given active vertex

Updating the destination vertex with the programmer supplied computation



Processing Phase Block Diagram

Line 2-3: Read Active SRC Property

Line 4: Read Edge Pointer

Line 5: Read Edges for given SRC

Line 6: Process Edge

Line 7: Read Temp. DST Property

Line 8: Reduce

Line 9: Write Temp. DST Property

From abstraction to HW

```
1  for (i=0; i<ActiveVertex.size(); i++) {
2    vid = ActiveVertexID[i];
3    vprop = ActiveVertexProp[i];
4    eptr = PtrToEdgeList[vid];
5    for (e = Edges[eptr]; e.src == vid; e = Edges[++eptr]) {
6      res = Process_Edge(e.weight, vprop);
7      temp = TempVertexProp[e.dst];
8      temp = Reduce(temp, res);
9      TempVertexProp[e.dst] = temp;
10   }
11 } // Apply Phase updates ActiveVertexProp with TempVertexProp
```

Sequential (vertex) Memory Access

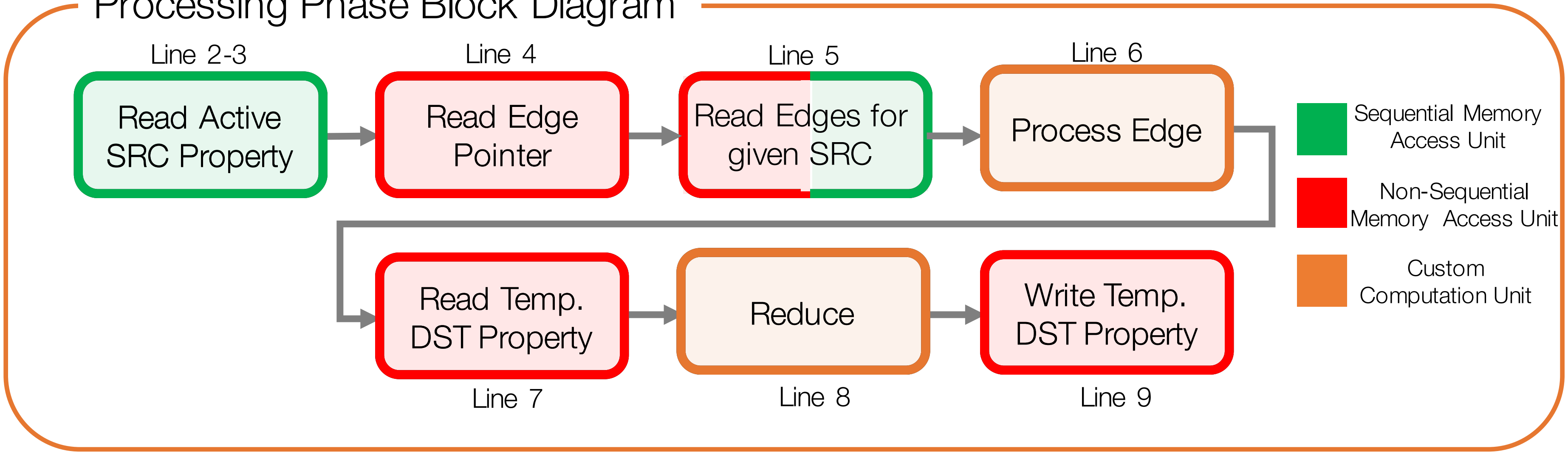
Non-sequential (edge ptr) Memory Access

Non-sequential and then Sequential (edge) Memory Access

Non-sequential (vertex) Memory Access

Custom Computation

Processing Phase Block Diagram



Line 2-3: Read Active SRC Property

Line 4: Read Edge Pointer

Line 5: Read Edges for given SRC

Line 6: Process Edge

Line 7: Read Temp. DST Property

Line 8: Reduce

Line 9: Write Temp. DST Property

🟩 Sequential Memory Access Unit

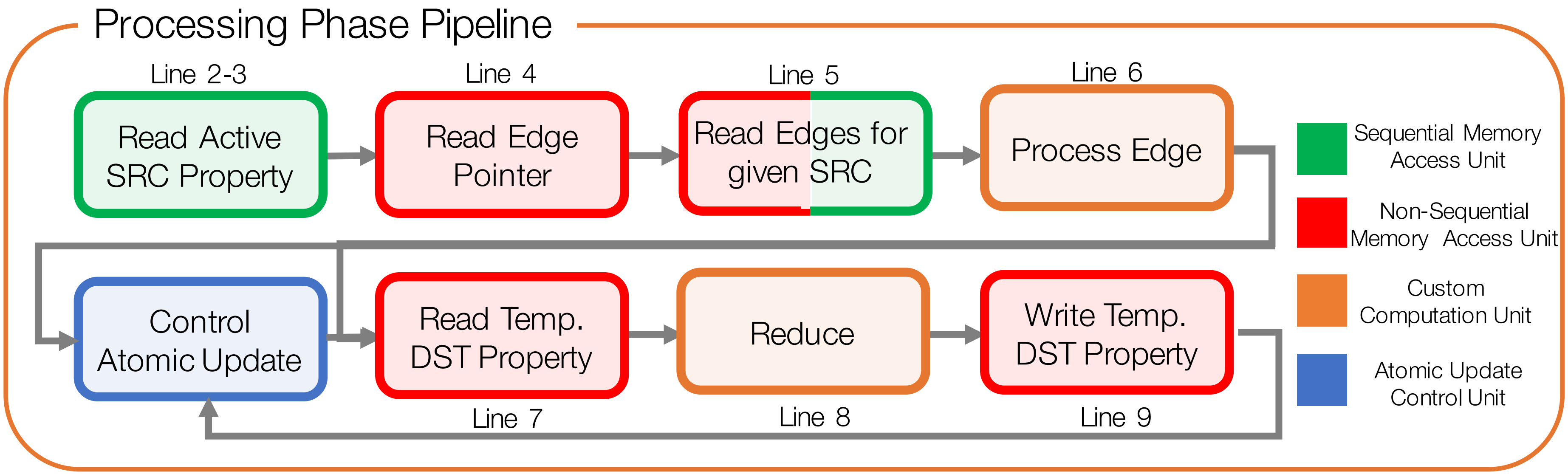
🟥 Non-Sequential Memory Access Unit

🟧 Custom Computation Unit

From abstraction to HW

```
1  for (i=0; i<ActiveVertex.size(); i++) {
2    vid = ActiveVertexID[i];
3    vprop = ActiveVertexProp[i];
4    eptr = PtrToEdgeList[vid];
5    for (e = Edges[eptr]; e.src == vid; e = Edges[++eptr]) {
6      res = Process_Edge(e.weight, vprop);
7      temp = TempVertexProp[e.dst];
8      temp = Reduce(temp, res);
9      TempVertexProp[e.dst] = temp;
10   }
11 } // Apply Phase updates ActiveVertexProp with TempVertexProp
```

Read-Modify-Write Update needs to be atomic for the same destination vertex (e.dst)



Processing Phase Pipeline

Line 2-3: Read Active SRC Property

Line 4: Read Edge Pointer

Line 5: Read Edges for given SRC

Line 6: Process Edge

Control Atomic Update

Line 7: Read Temp. DST Property

Line 8: Reduce

Line 9: Write Temp. DST Property

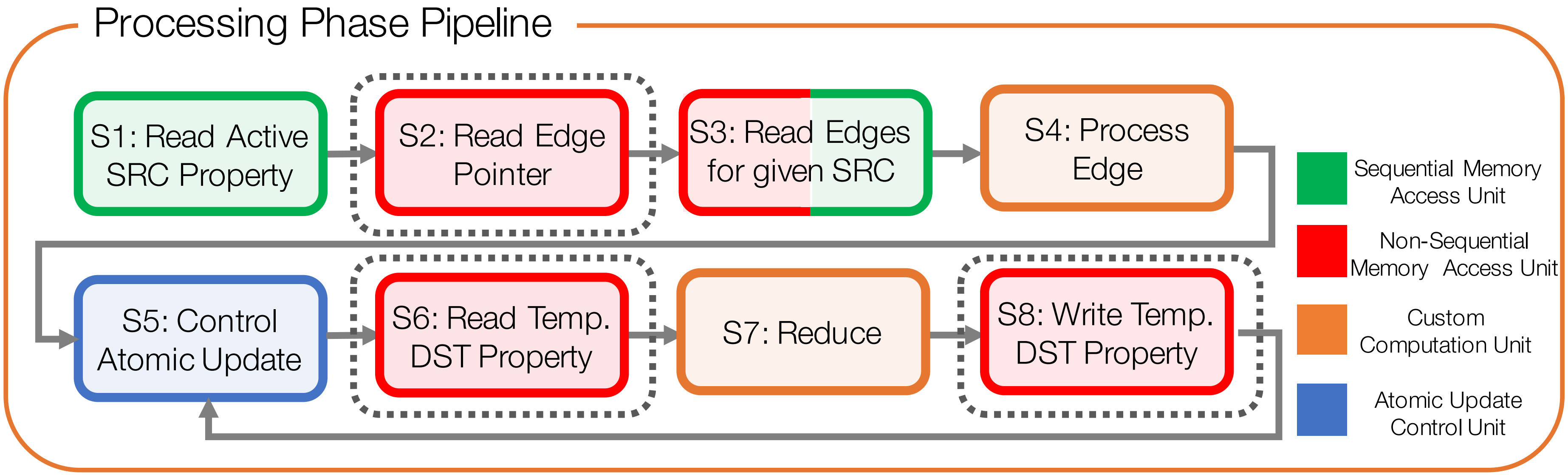
🟩 Sequential Memory Access Unit

🟥 Non-Sequential Memory Access Unit

🟧 Custom Computation Unit

🟦 Atomic Update Control Unit

Optimizing Data Movement

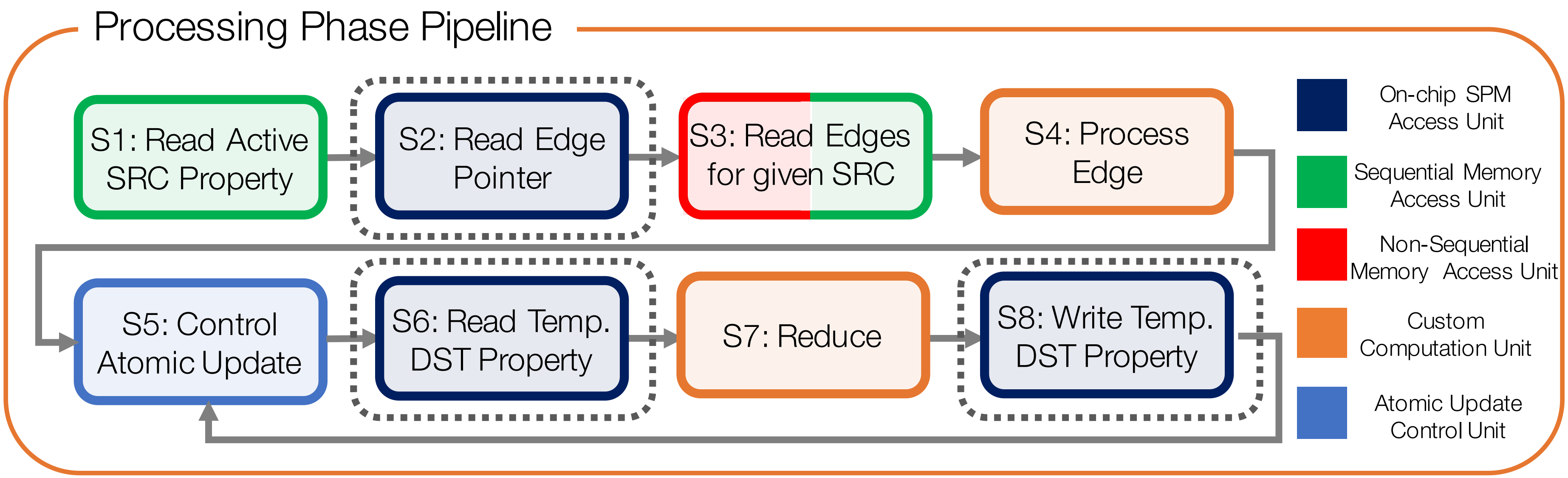

Processing Phase Pipeline
S1: Read Active SRC Property
S2: Read Edge Pointer
S3: Read Edges for given SRC
S4: Process Edge
S5: Control Atomic Update
S6: Read Temp. DST Property
S7: Reduce
S8: Write Temp. DST Property
Sequential Memory Access Unit
Non-Sequential Memory Access Unit
Custom Computation Unit
Atomic Update Control Unit

1. Non-sequential access; low-spatial-locality `PtrtoEdgeList[vid]` `TempVertexProp[e.dst]`
   - Only use 4-16B out of 64B off-chip memory access granularity; leads to off-chip memory BW waste

2. High-temporal locality `TempVertexProp[e.dst]`
   - Data will be used again for edges with the same destination vertex

Use fine-grained on-chip SPM for data used for these modules

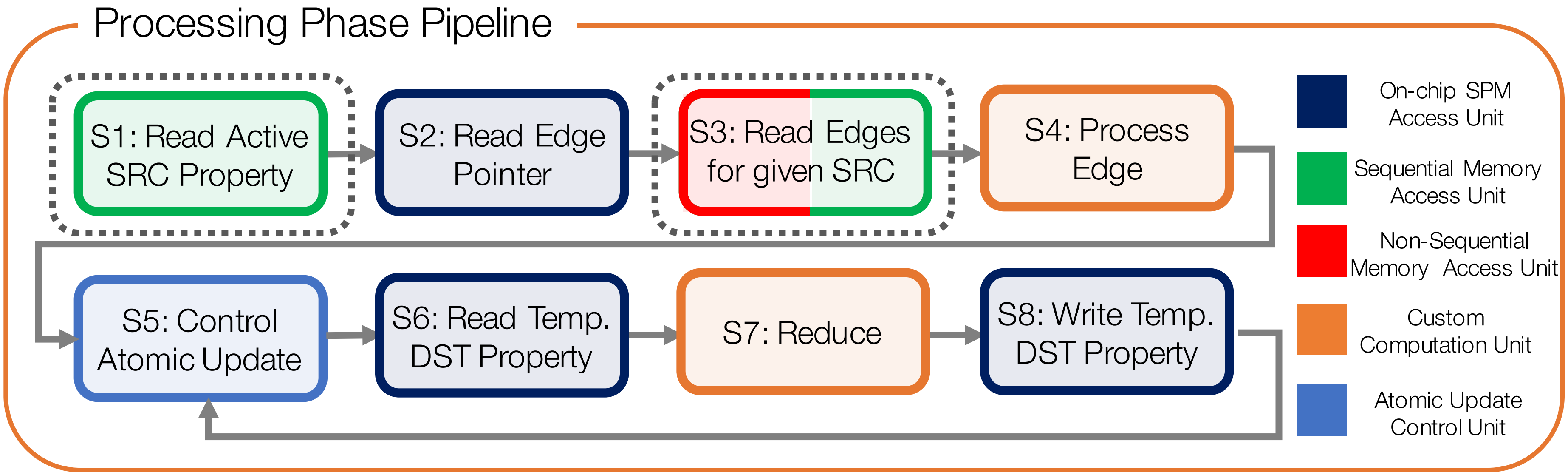
Optimizing Data Movement


Processing Phase Pipeline
S1: Read Active SRC Property
S2: Read Edge Pointer
S3: Read Edges for given SRC
S4: Process Edge
S5: Control Atomic Update
S6: Read Temp. DST Property
S7: Reduce
S8: Write Temp. DST Property
On-chip SPM Access Unit
Sequential Memory Access Unit
Non-Sequential Memory Access Unit
Custom Computation Unit
Atomic Update Control Unit

1. Non-sequential access; low-spatial-locality `PtrtoEdgeList[vid]` `TempVertexProp[e.dst]`
   - Only use 4-16B out of 64B off-chip memory access granularity; leads to off-chip memory BW waste
2. High-temporal locality `TempVertexProp[e.dst]`
   - Data will be used again for edges with the same destination vertex

Use fine-grained on-chip SPM for data used in these modules

Optimizing Data Movement


Processing Phase Pipeline
S1: Read Active SRC Property
S2: Read Edge Pointer
S3: Read Edges for given SRC
S4: Process Edge
S5: Control Atomic Update
S6: Read Temp. DST Property
S7: Reduce
S8: Write Temp. DST Property
On-chip SPM Access Unit
Sequential Memory Access Unit
Non-Sequential Memory Access Unit
Custom Computation Unit
Atomic Update Control Unit

1. Sequentially accessed; high-spatial locality

```
ActiveVertexProp[i]
   Edges[eptr++]
```

   - Has predictable access pattern
   - All 64B loaded from the off-chip memory will be used

2. Low-temporal locality

   - Each active vertex or an edge is processed only once. This data will not be accessed again for the current iteration.

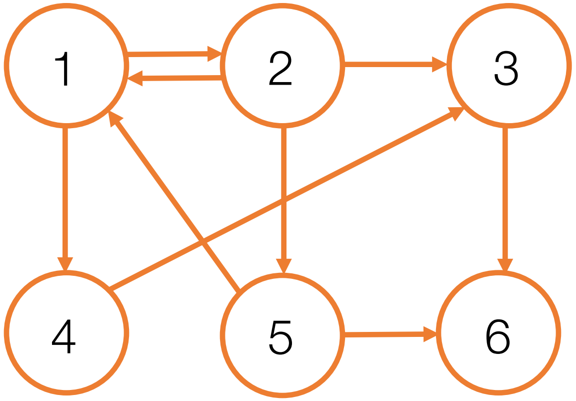
Use prefetcher; Do not use on-chip SPM for data used for these modules

Scaling On-chip Memory Usage

- Graphicionado utilizes an on-chip storage to avoid wasting off-chip BW
  - Often requires 4-16B on-chip storage per vertex

➡ Not enough on-chip storage for 10M+ vertices

- Solution: Partition a graph before the execution; Then, process each subgraph at a time
  - Assign edges to different slices based on their destination vertices



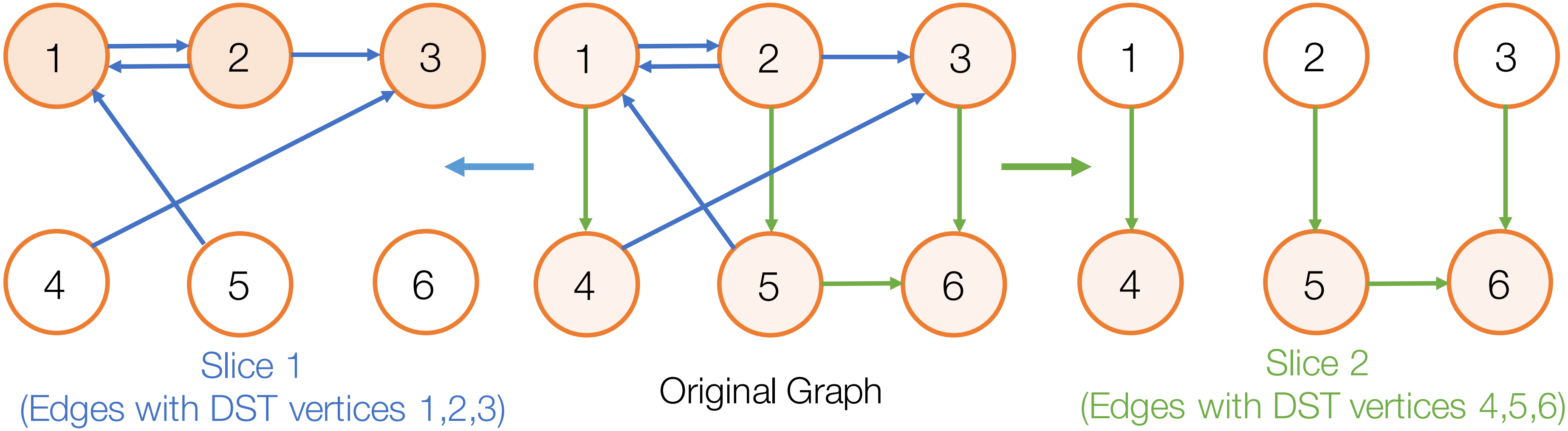
Original Graph

Scaling On-chip Memory Usage

- Graphicionado utilizes an on-chip storage to avoid wasting off-chip BW
  - Often requires 4-16B on-chip storage per vertex

➡ Not enough on-chip storage for 10M+ vertices

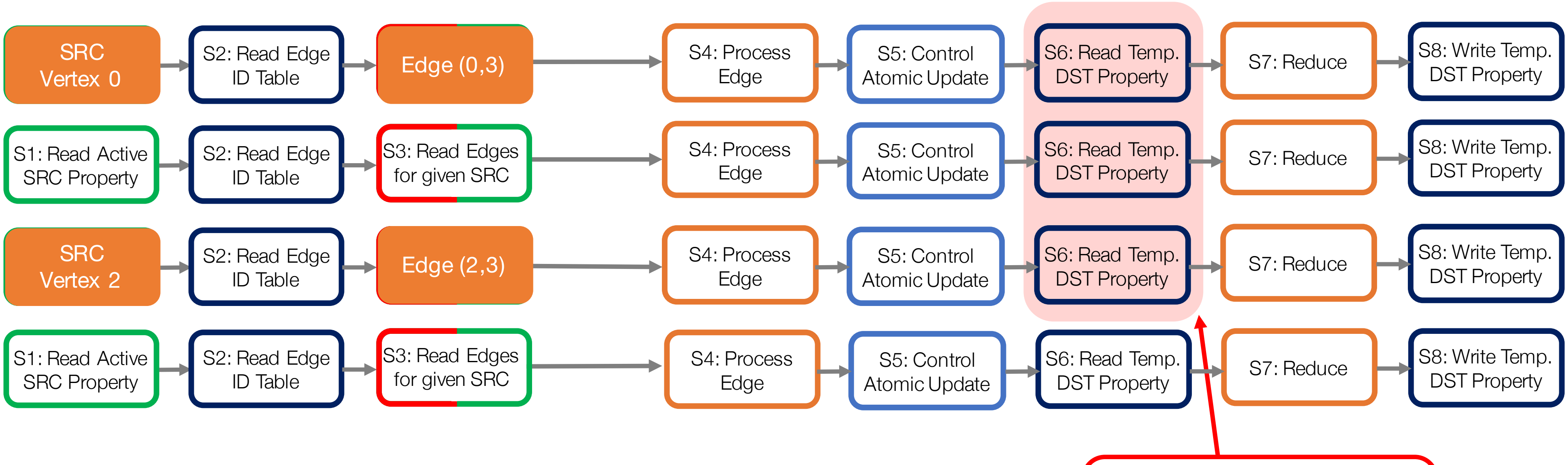
- Solution: Partition a graph before the execution; Then, process each subgraph at a time
  - Assign edges to different slices based on their destination vertices


1
2
3
4
5
6
Slice 1
(Edges with DST vertices 1,2,3)
1
2
3
4
5
6
Original Graph
1
2
3
4
5
6
Slice 2
(Edges with DST vertices 4,5,6)

Only half of the vertices to be stored in on-chip storage at a time

*Paper has additional scaling techniques

Parallelizing Graphicionado Pipeline


SRC Vertex 0
S2: Read Edge ID Table
Edge (0,3)
S4: Process Edge
S5: Control Atomic Update
S6: Read Temp. DST Property
S7: Reduce
S8: Write Temp. DST Property
S1: Read Active SRC Property
S2: Read Edge ID Table
S3: Read Edges for given SRC
S4: Process Edge
S5: Control Atomic Update
S6: Read Temp. DST Property
S7: Reduce
S8: Write Temp. DST Property
SRC Vertex 2
S2: Read Edge ID Table
Edge (2,3)
S4: Process Edge
S5: Control Atomic Update
S6: Read Temp. DST Property
S7: Reduce
S8: Write Temp. DST Property
S1: Read Active SRC Property
S2: Read Edge ID Table
S3: Read Edges for given SRC
S4: Process Edge
S5: Control Atomic Update
S6: Read Temp. DST Property
S7: Reduce
S8: Write Temp. DST Property

Two modules trying to read the same address (`TempVertexProp[3]`)

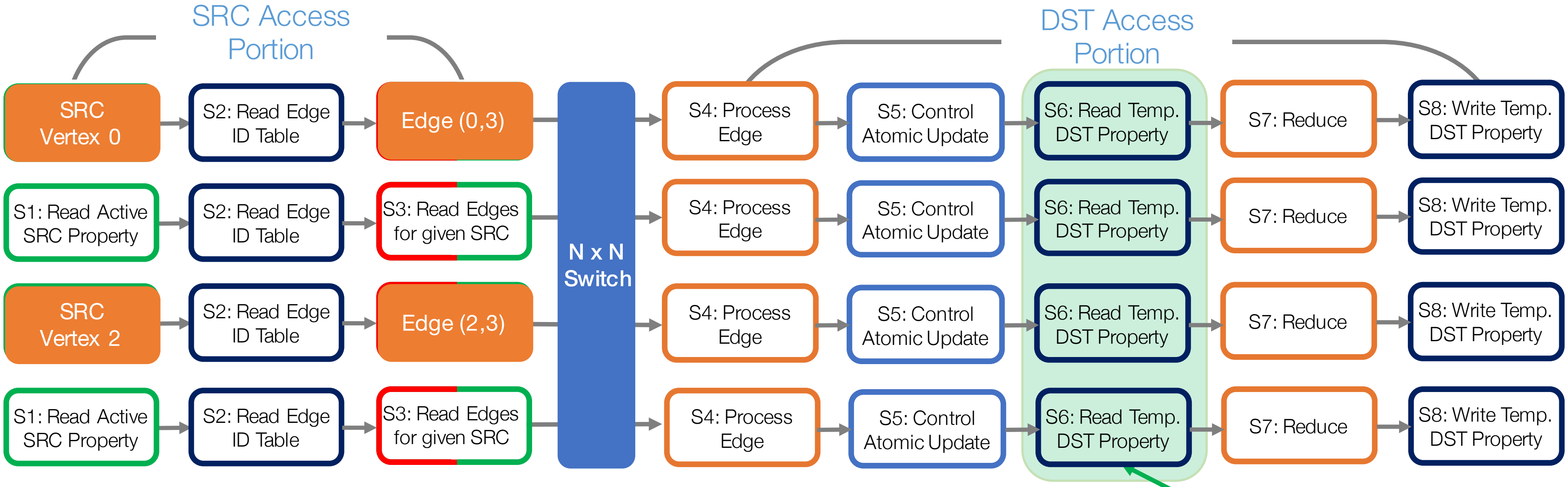
- Naive Parallelization Approach
  - Replicate Pipeline
  - Each pipeline stream processes a portion of the active (SRC) vertices

    (e.g., a modulo of the SRCID determines which stream for processing)

Different streams will try to read/write the same memory address (**`TempVertexProp[]`**) at the same time on S6 & S8

➡ Design complexity (SPM port count) and performance degradation

Parallelizing Graphicionado Pipeline



Each unit accesses an exclusive memory region

- Graphicionado Approach
  - Each pipeline stream is separated to two pieces: SRC access / DST access portion
  - N x N switch routes data to the correct DST stream (e.g., a modulo of the DSTID)

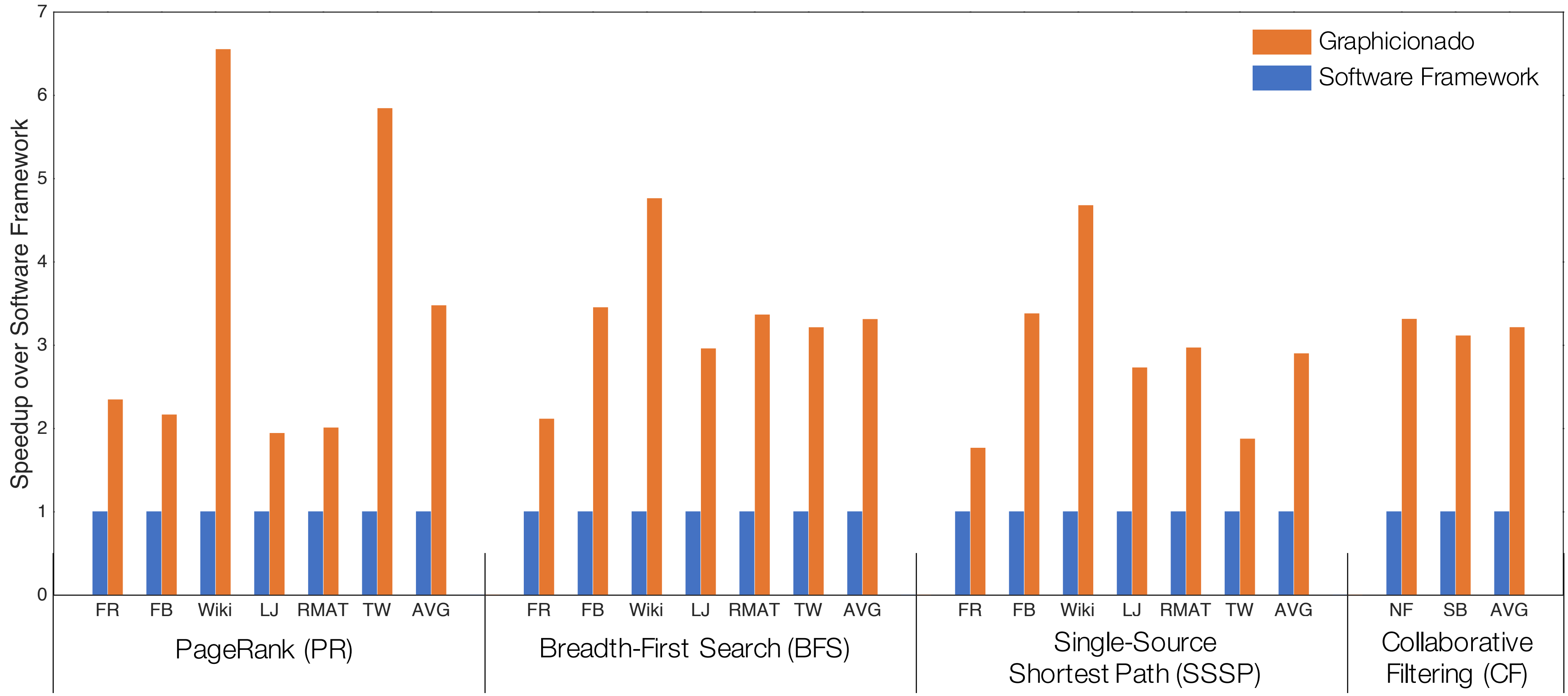
Now each unit access an exclusive portion of the scratchpad memory

➡ Reduction in design complexity and improvement in throughput

Presentation Outline

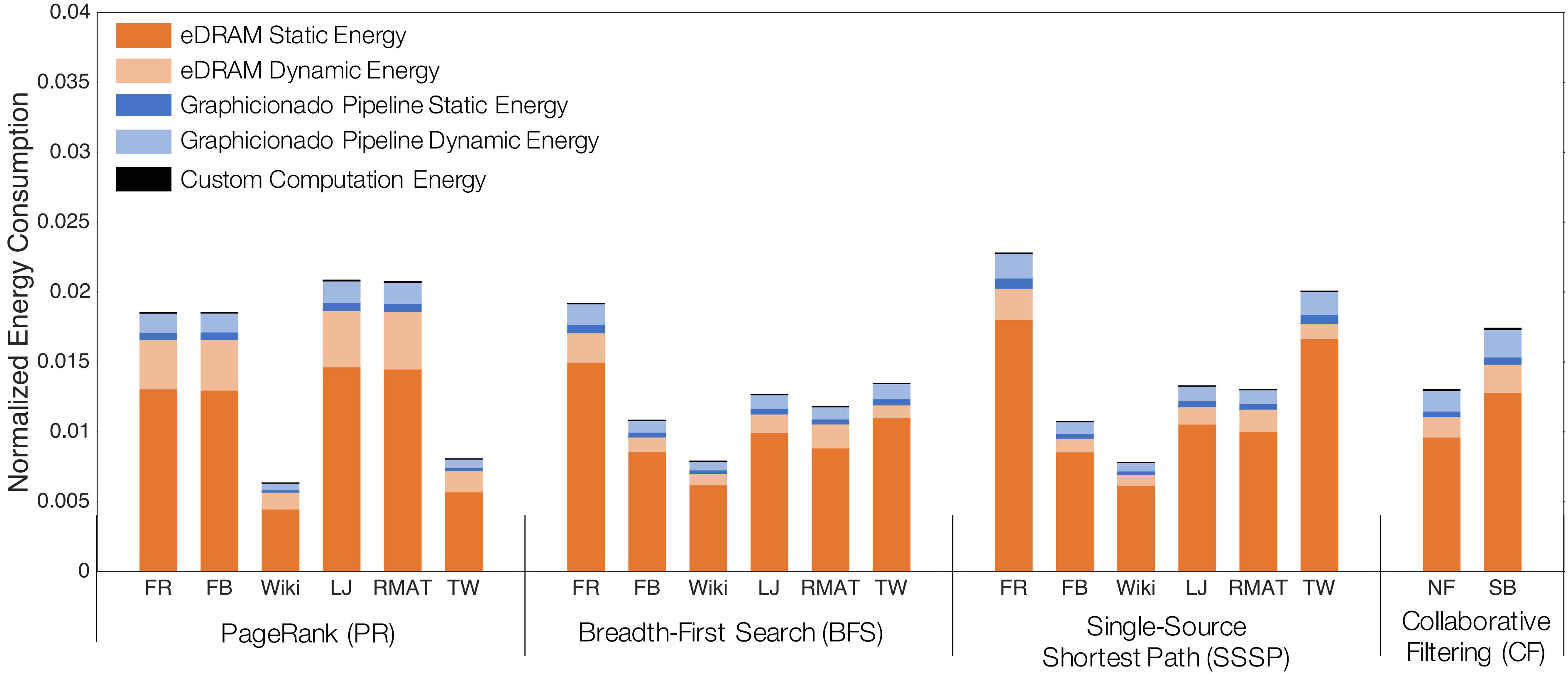
- Why Graphicionado?
- Vertex Programming Abstraction
- Constructing Graphicionado Pipeline from Abstraction
- Optimizing Graphicionado
    - Optimizing Data Movement
    - Scaling On-Chip Memory Usage
    - Parallelization
- Performance/Energy Evaluation
- Conclusions

Overall Graphicionado Performance



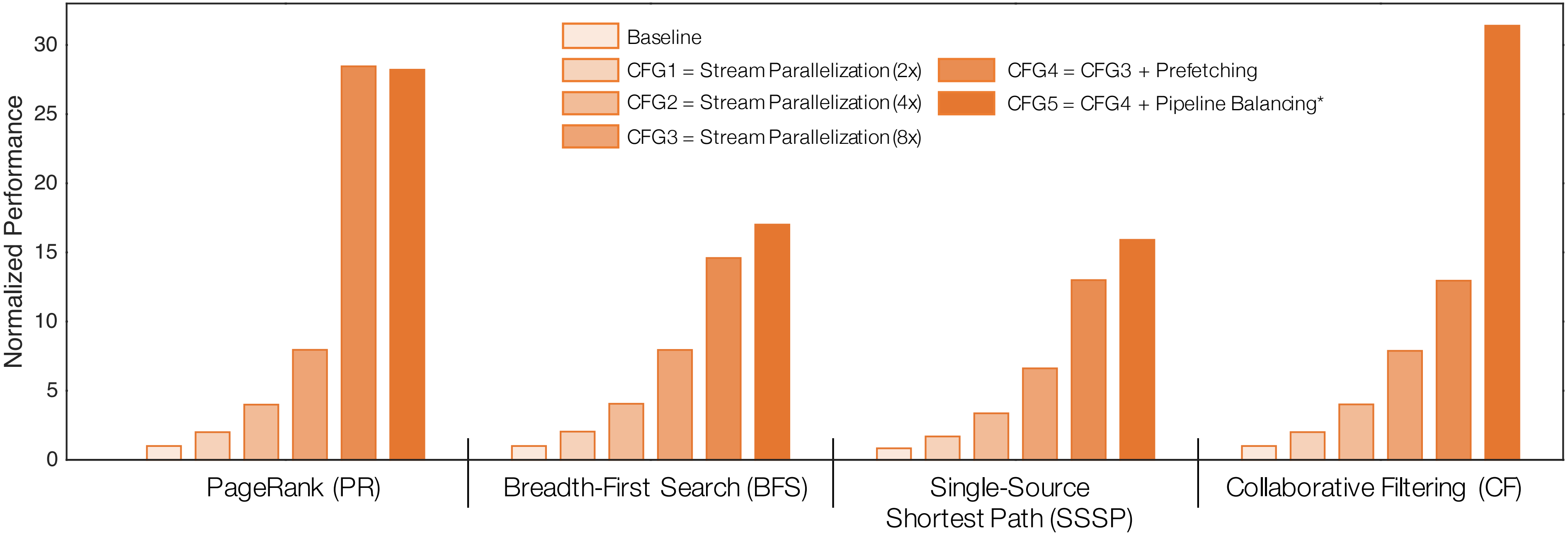
- Graphicionado achieves ~3x speedup over state-of-the-art software framework (Intel GraphMat) across different workloads
  - Software framework was run on a 32-core Haswell Xeon server
  - Both system were provisioned the same theoretical peak memory BW (78GB/s)

Graphicionado Energy Consumption



- Graphicionado consumes < 2% of the energy (50x-100x) compared to the software processing framework
  - Most of the energy (70%+) was spent for the eDRAM static energy
  - 20-25x power saving & 2-5x speedup

Effect of Parallelization & Optimization



- The parallelization and optimization of Graphicionado achieves up to 27x speedup over the baseline single-stream pipeline
  - Without effective parallelizations and optimizations, simply using HW does not necessarily bring any benefit

Conclusions

- Software graph processing frameworks has limitations
  - Inefficient on-chip memory (cache) usage
  - Expensive data movement

- Graphicionado: Specialized HW accelerator for graph analytics
  - Efficient use of on-chip memory
  - Specialized pipeline tailored for graph analytics data movement
  - Effective parallelism

~3x speedup and 50x+ energy saving over state-of-the-art software framework

MICRO 2016

Graphicionado

A High-Performance and Energy-Efficient Graph Analytics Accelerator

Tae Jun Ham
Lisa Wu
Narayanan Sundaram
Nadathur Satish
Margaret Martonosi

PRINCETON UNIVERSITY

intel