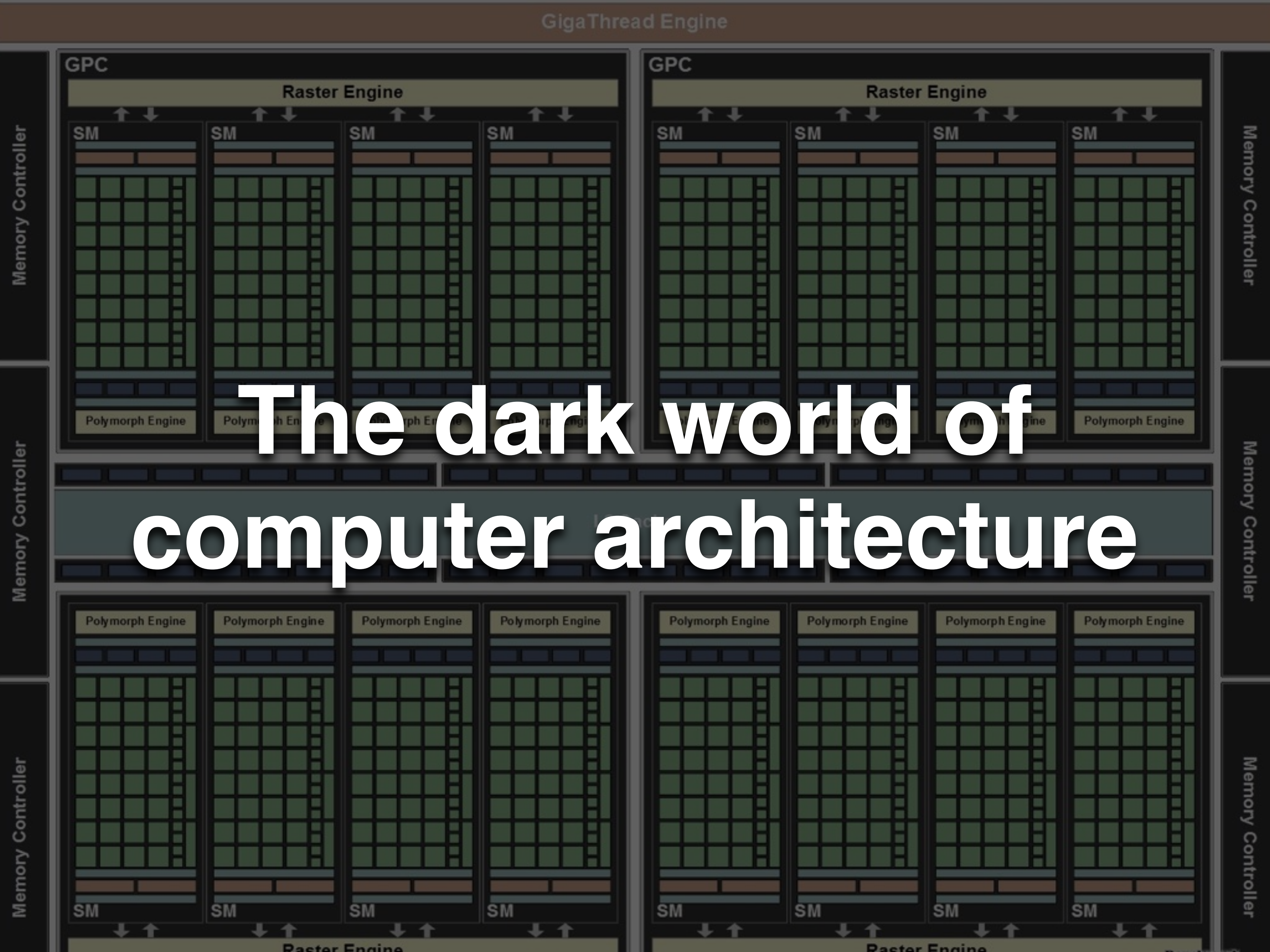
Sparse & irregular processing on GPUs using functional languages

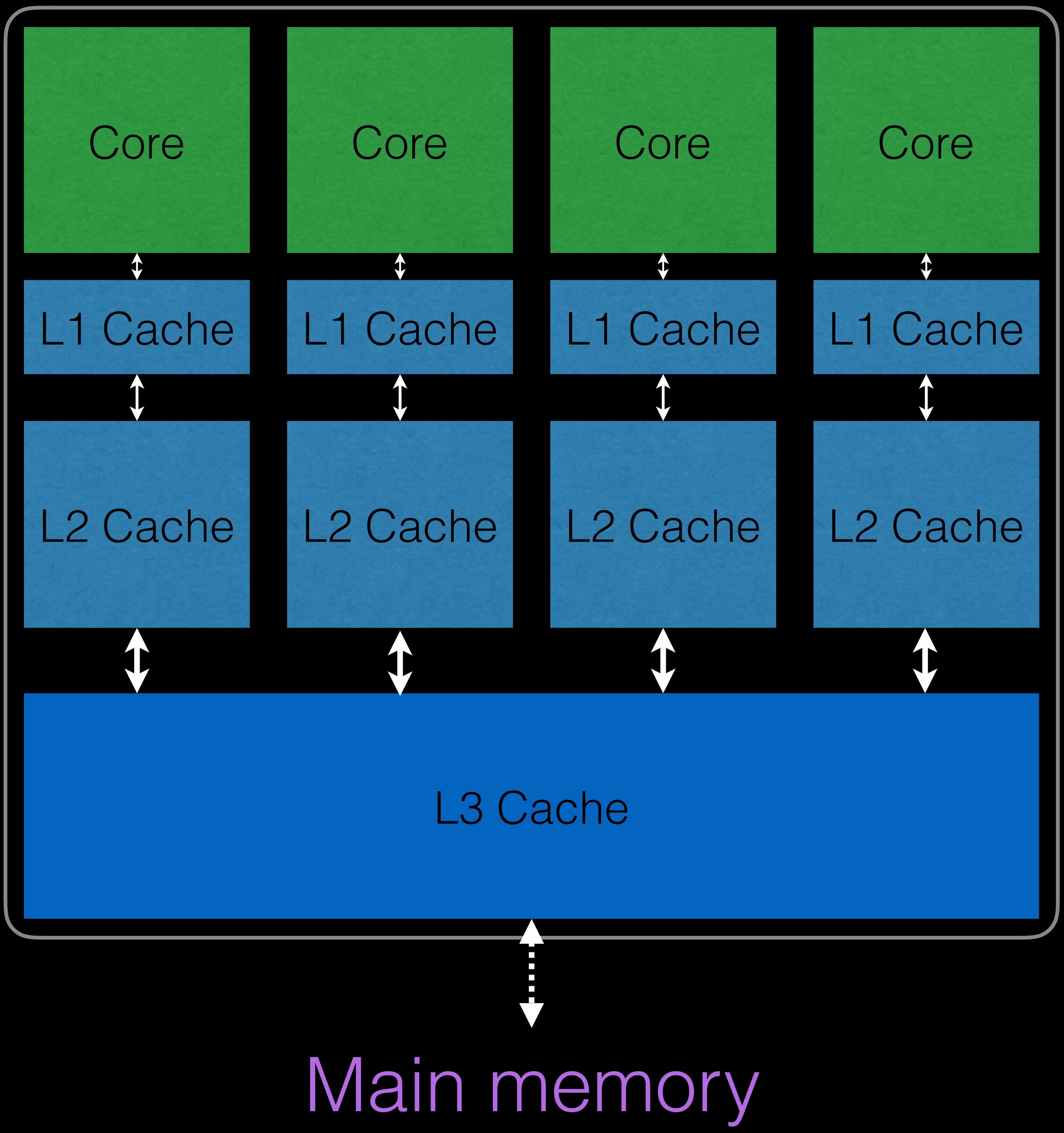
Adam Harries

School of Informatics, University of Edinburgh

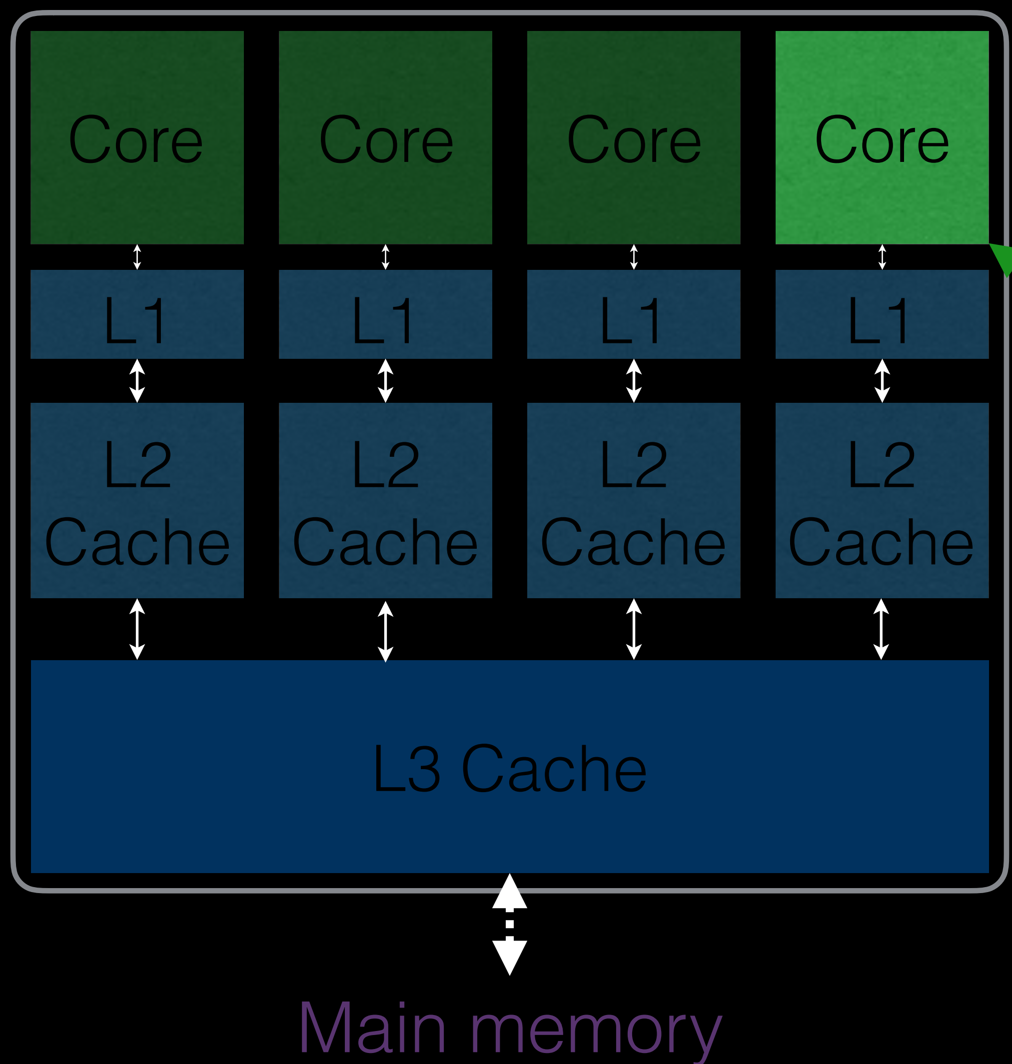
The dark world of computer architecture



Current CPUs

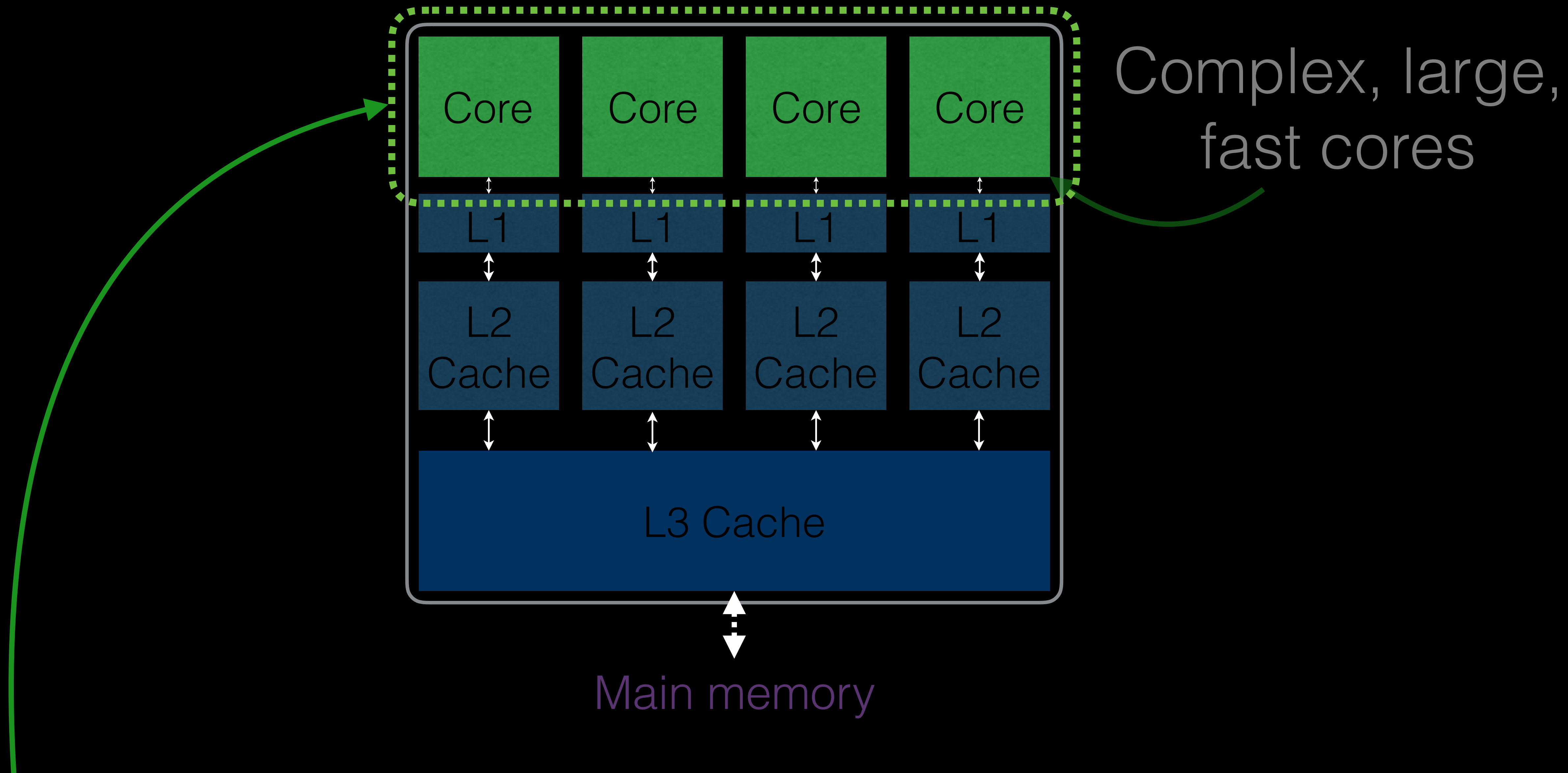


Current CPUs



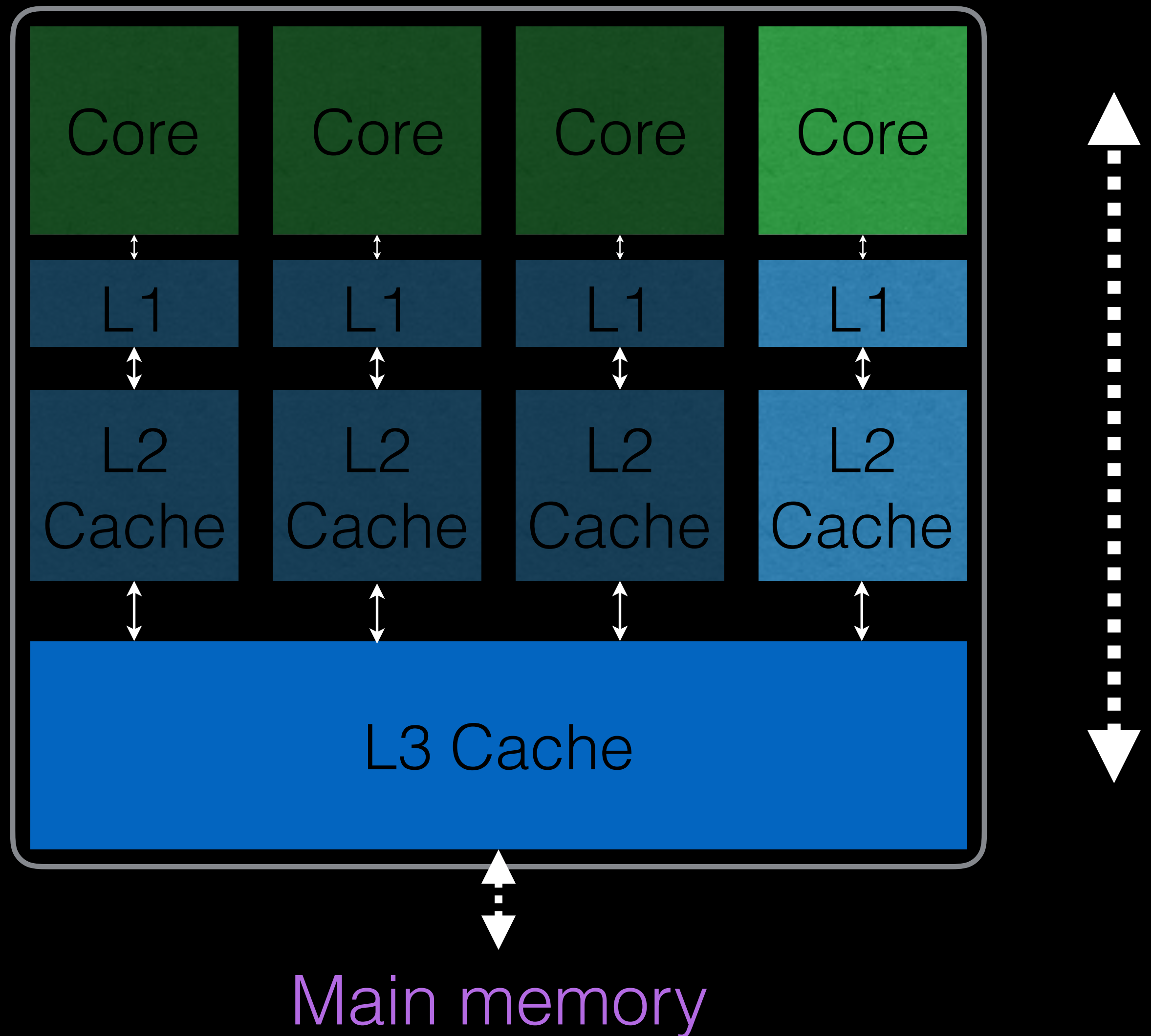
Complex, large, fast cores

Current CPUs



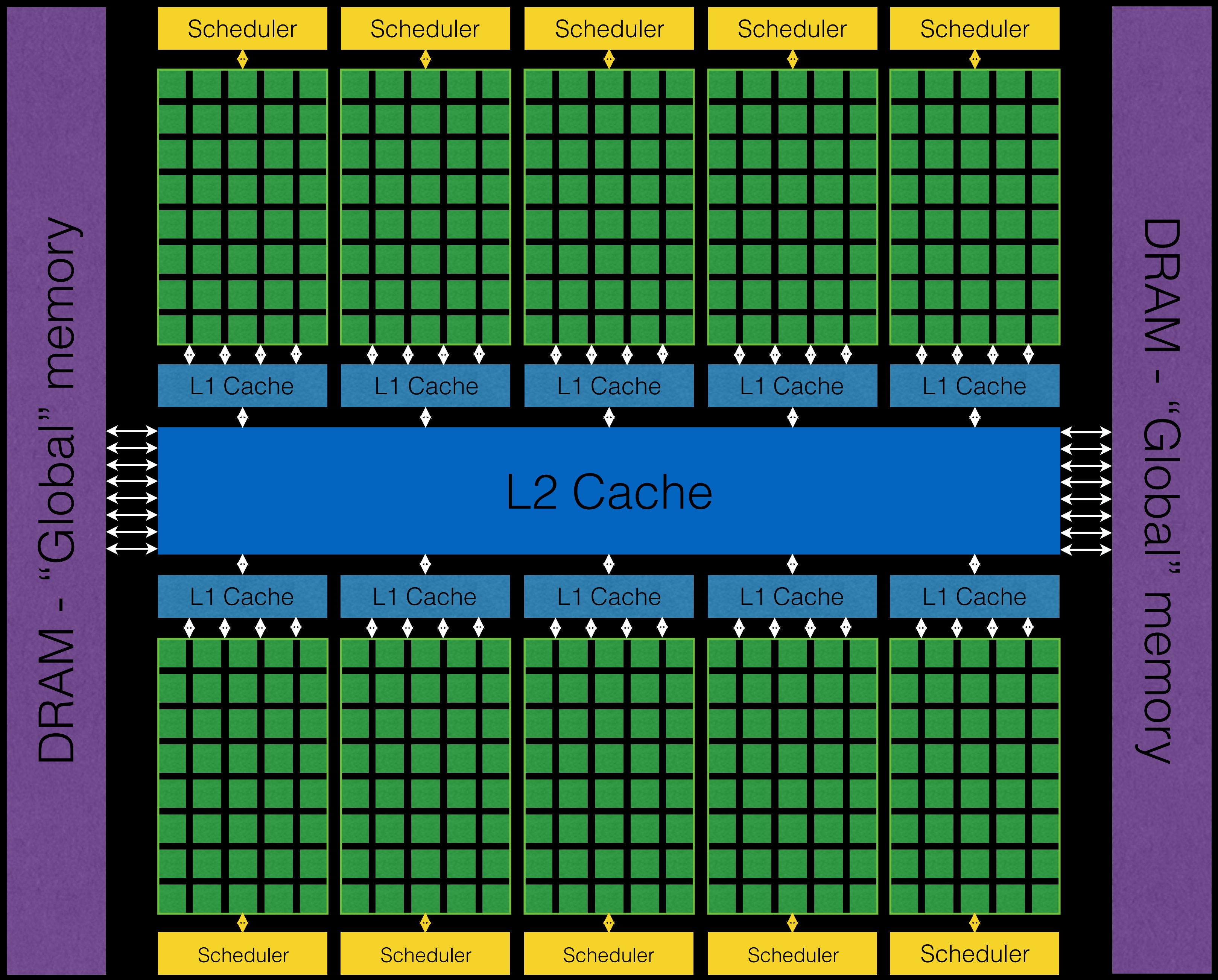
Coarse parallelism: **low thread count** and **heavyweight threads**

Current CPUs

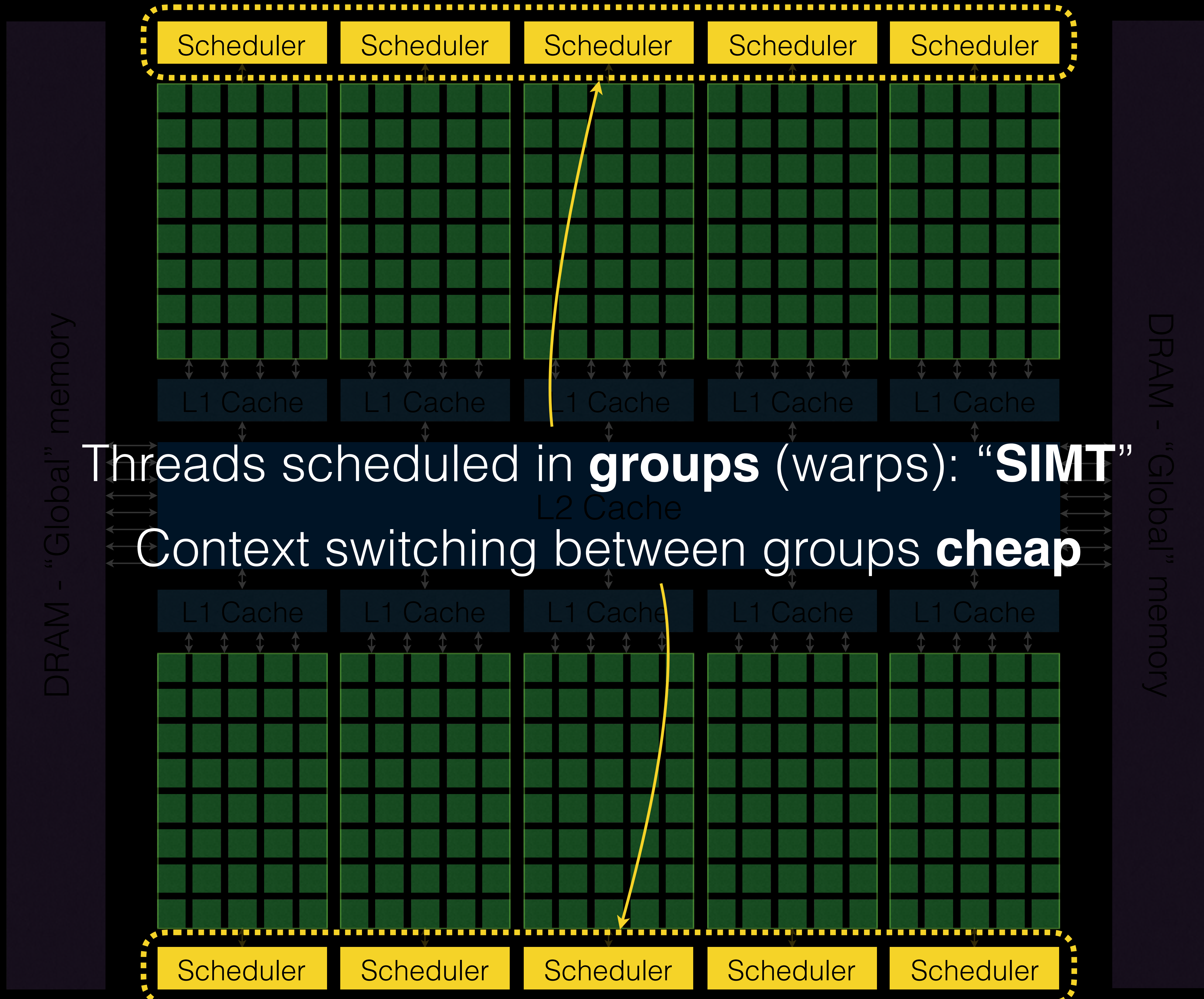


Context switching between threads is **expensive**:
architecture optimised for single threaded performance

Current GPUs



Huge numbers of **lightweight**, **simple** cores
= massive parallelism



Threads scheduled in **groups** (warps): “**SIMT**”

Context switching between groups **cheap**

Conclusion: GPUs give you *greater*, more *flexible* parallelism

Conclusion: GPUs give you *greater*, more *flexible* parallelism…right?

Conclusion: GPUs give you *greater*, more *flexible* parallelism…right?

Why don't we use them for everything?

No caches: Programmers must be careful with intelligently fetching data

Grouped Parallelism: Programmers should avoid thread divergence within groups

Simple cores: Programmers must parallelise with fine grained *and statically* distributed workloads

No caches: Programmers must be careful with intelligently fetching data

Grouped Parallelism: Programmers should avoid thread divergence within groups

Simple cores: Programmers must parallelise with fine grained *and statically* distributed workloads

GPUs encourage **static** *array based* parallelism

No caches: Programmers must be careful with intelligently fetching data

Grouped Parallelism: Programmers should avoid thread divergence within groups

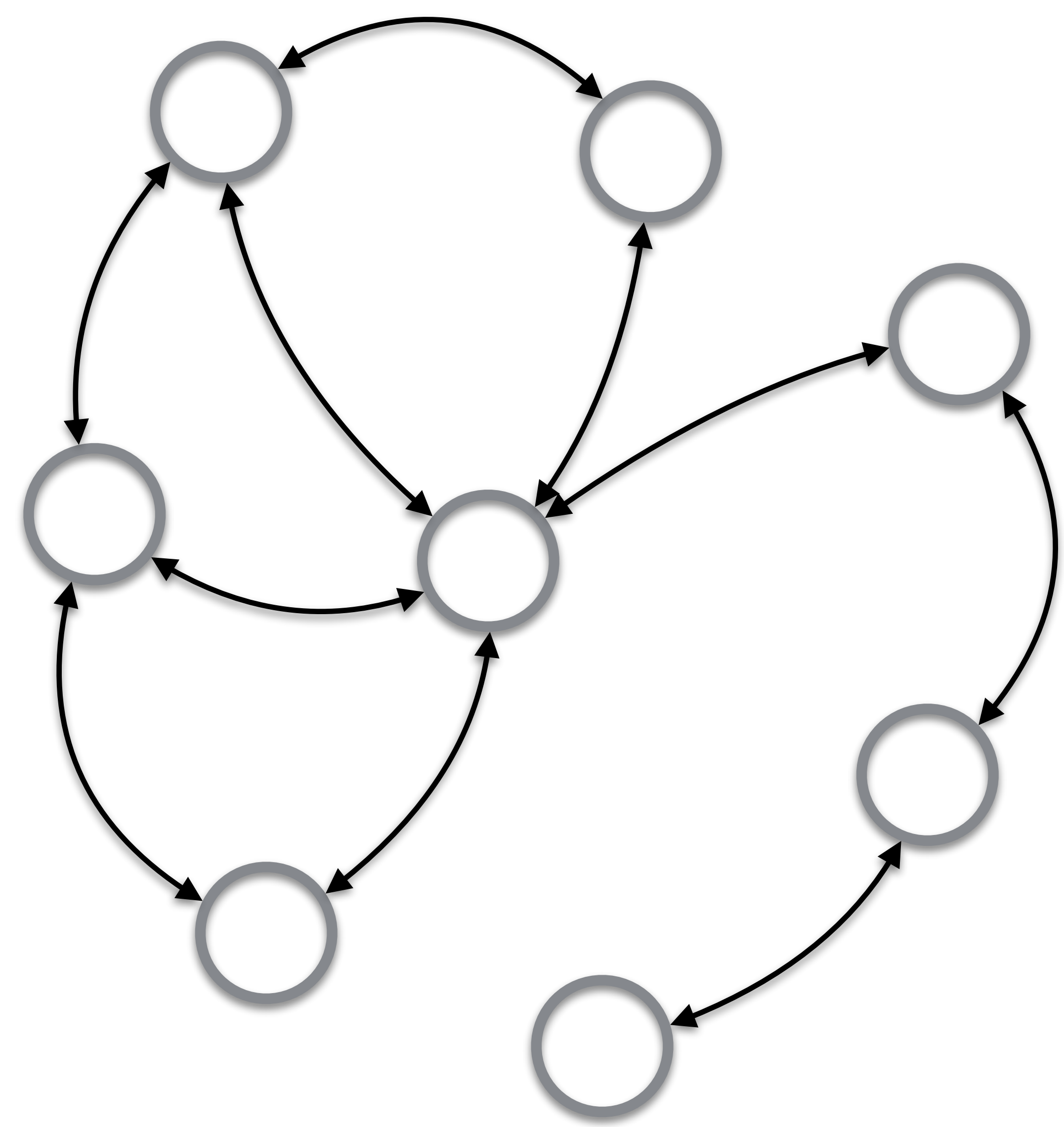
Simple cores: Programmers must parallelise with fine grained *and statically* distributed workloads

GPUs encourage **static** *array based* parallelism
How can we **efficiently** parallelise *other domains*?

Domain of interest:
Graph algorithms

Aim: accelerate processing using **GPU parallelism**

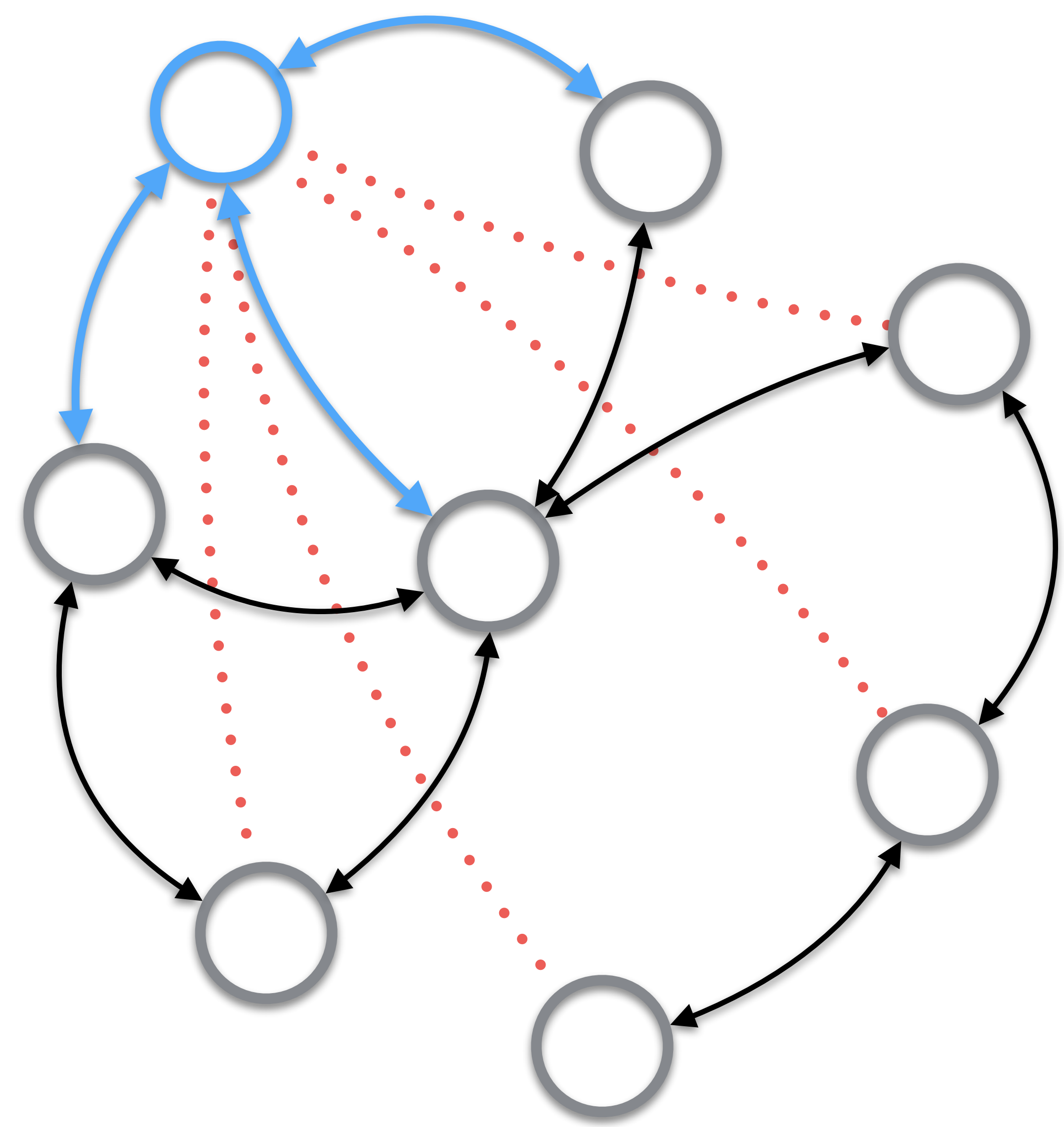
What are the **problems**?



What are the **problems**?

Sparsity:

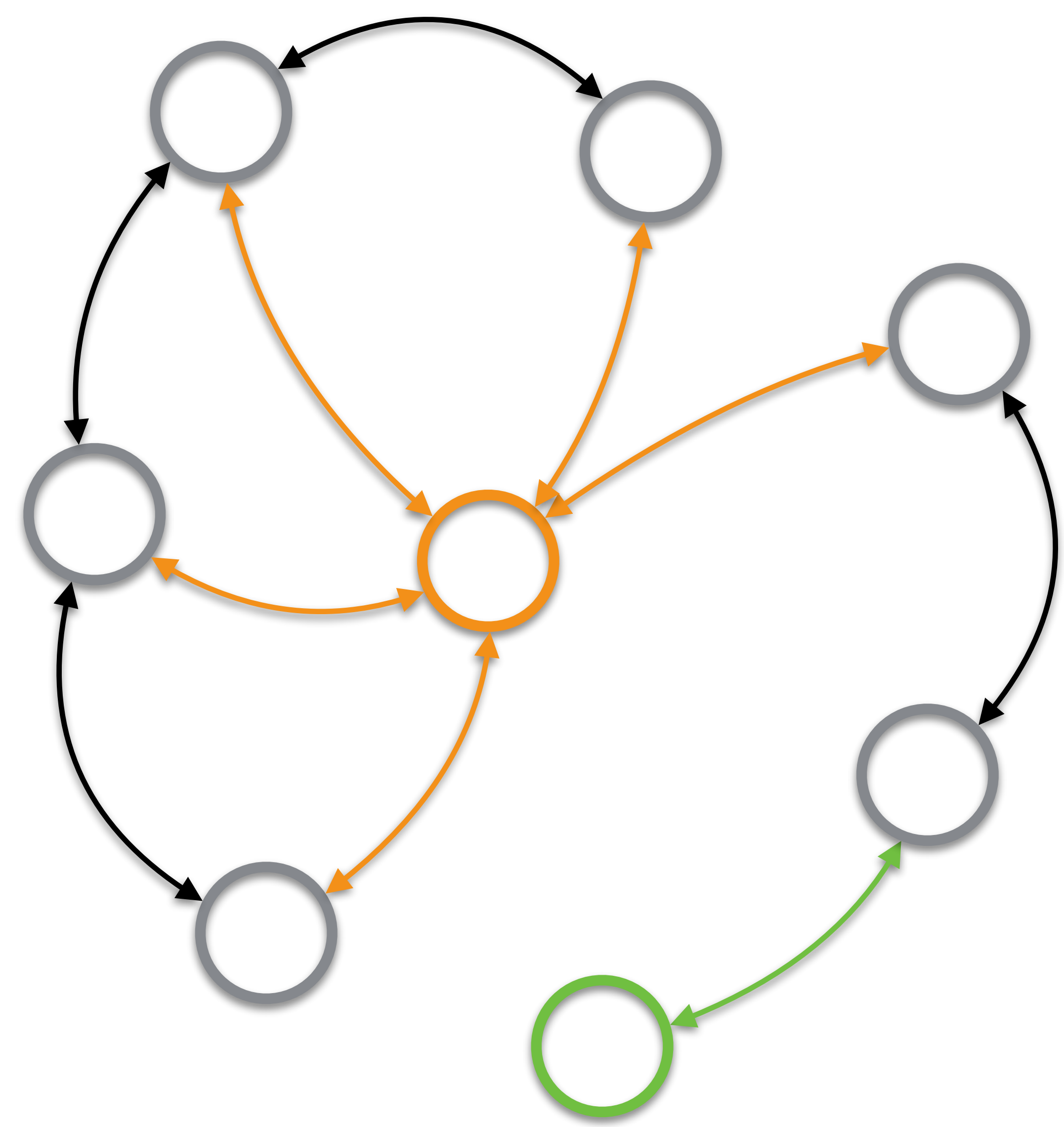
- Complex data structures
- Random data access patterns
- Low compute to data ratio



What are the **problems**?

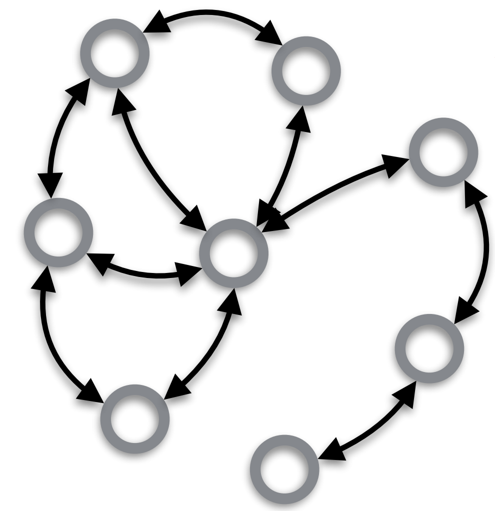
Irregularity:

- Unbalanced workload within data sets
- Control flow divergence
- Coarse granularity of parallelism



Our proposal:

Translate problem to simpler domain: (sparse) **Linear Algebra**



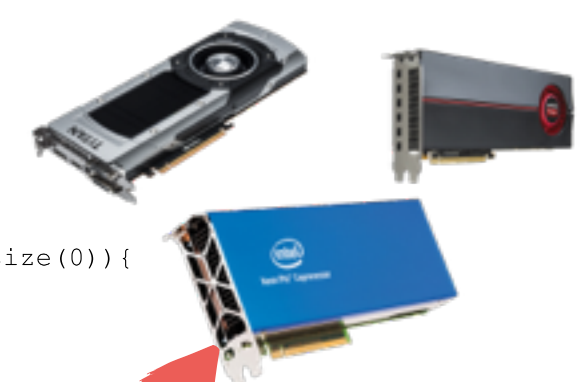
$$\sum_{i=0} a_i \times b_i$$

Express algorithm with **high level parallelism** (aka, functional languages)

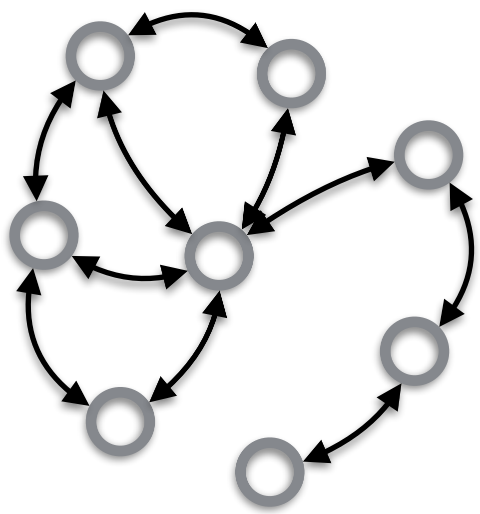
```
Reduce(add, 0) o Map(mult)
```

Explore space of implementations to find **high performance** solution

```
__kernel void dotProduct(
    __global const elem_t* a,
    __global const elem_t* b,
    __global const elem_t* res,
    __global const int len){
    int id = get_global_id(0);
    elem_t tmp;
    for(int i = id;i<len;i+=get_global_size(0)){
        tmp = a[i]*b[i];
        atomic_add(res, tmp);
    }
}
```



Our proposal:

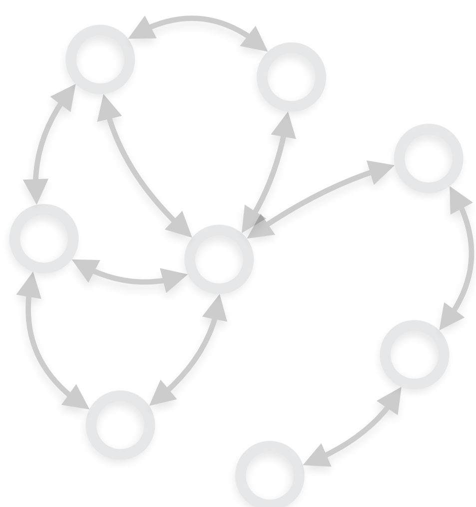


$$\sum_{i=0} a_i \times b_i$$

Translate problem to simpler domain:
(sparse) **Linear Algebra**

(intuition: operations on adjacency matrix of graph)

Our proposal:



$$\sum_{i=0} a_i \times b_i$$

```
Reduce(add, 0) o Map(mult)
```

Implement linear algebra in parallel using a high level **functional** language

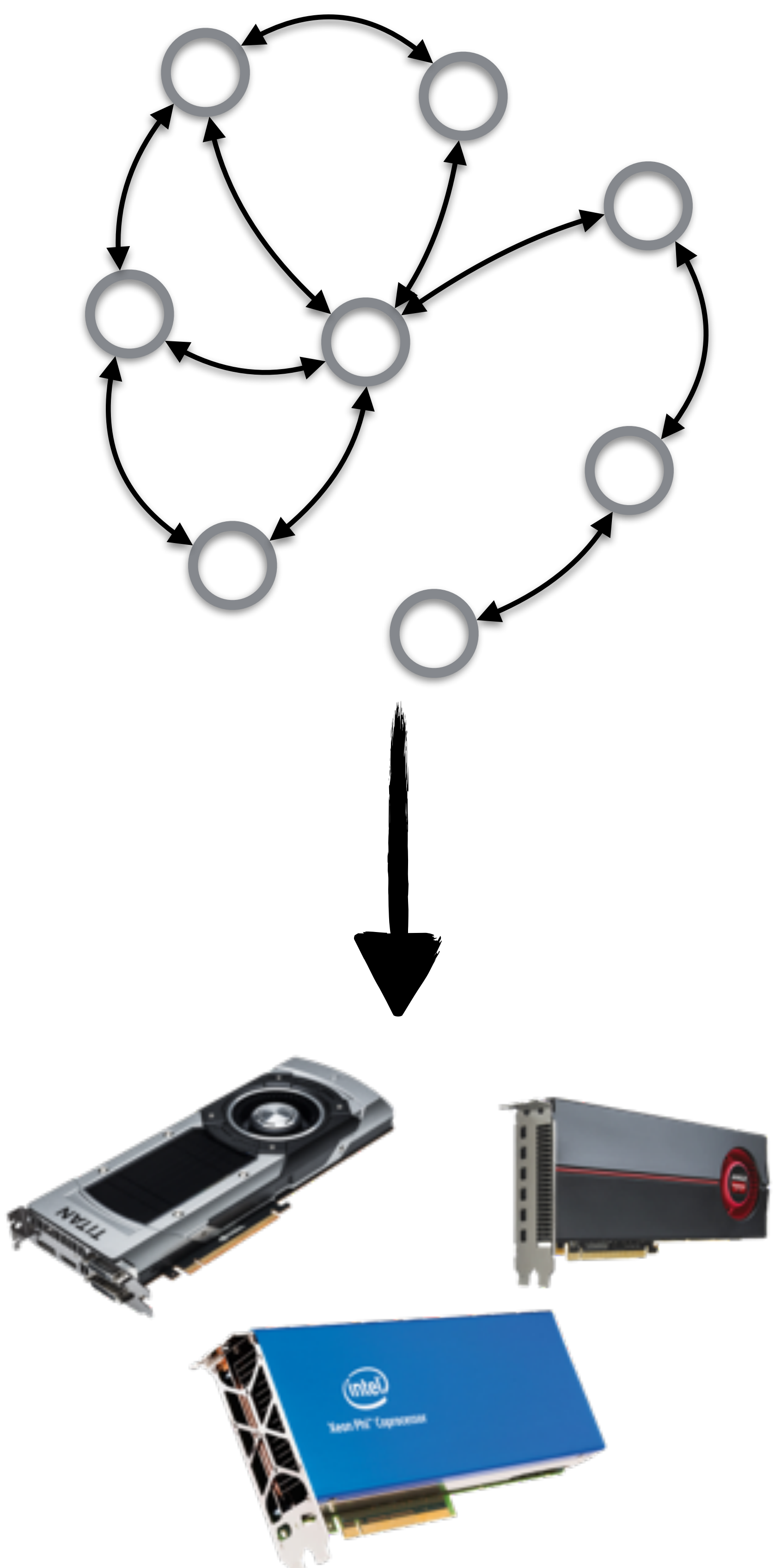
Our proposal:

Use rewrite rules to automatically generate **high performance** OpenCL code

$$\sum_{i=0} a_i \times b_i$$

```
Reduce(add, 0) o Map(mult)
```

```
__kernel void dotProduct(
    __global const elem_t* a,
    __global const elem_t* b,
    __global const elem_t* res,
    __global const int len){
    int id = get_global_id(0);
    elem_t tmp;
    for(int i = id;i<len;i+=get_global_size(0)){
        tmp = a[i]*b[i];
        atomic_add(res, tmp);
    }
}
```

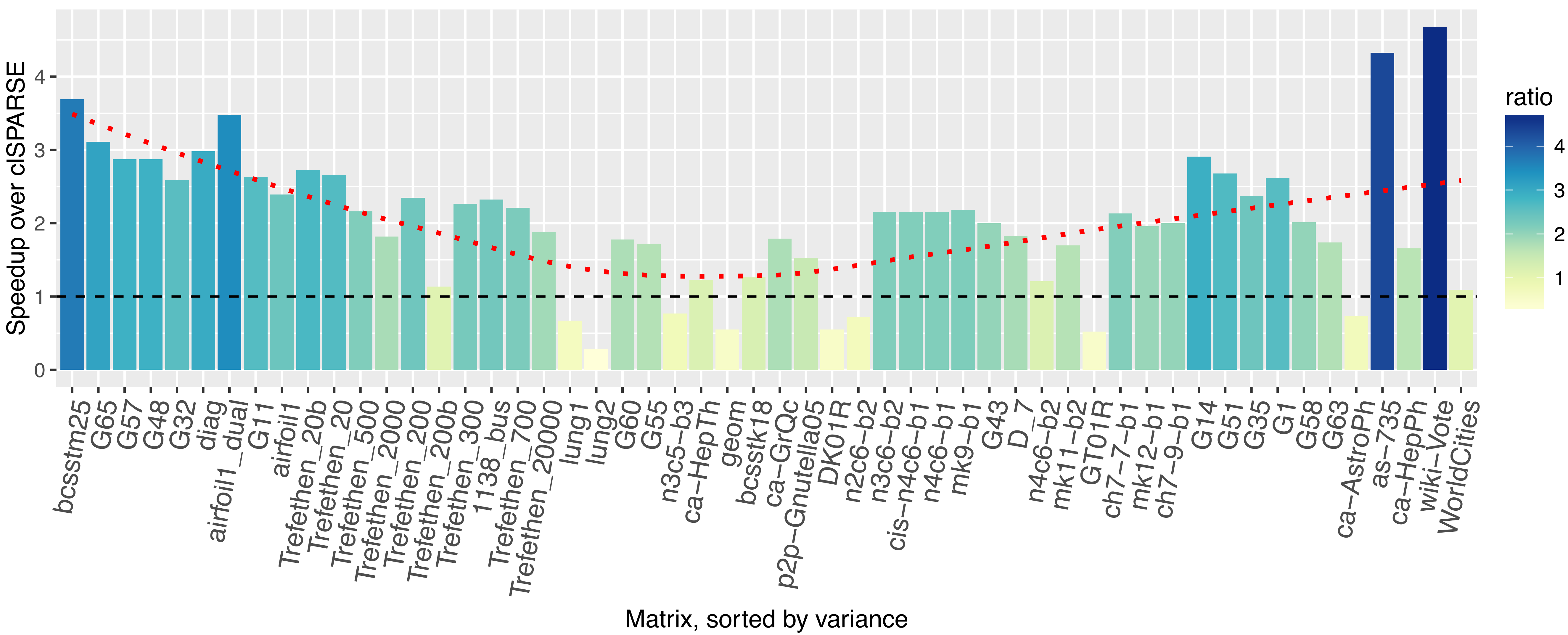


Sparsity: Translate complex sparse domain to simpler dense domain

Irregularity: Regularise code using intelligent compilation techniques

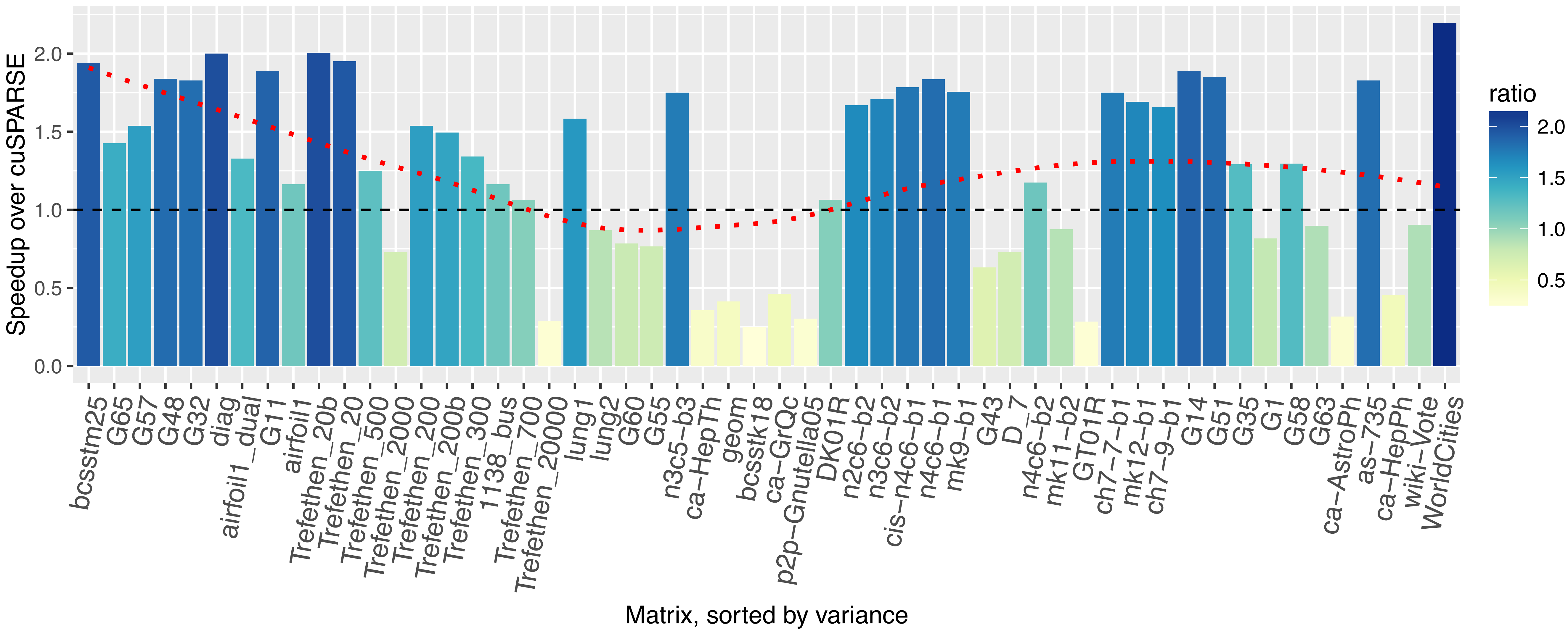
---

Current work: extending to and evaluating on other domains.

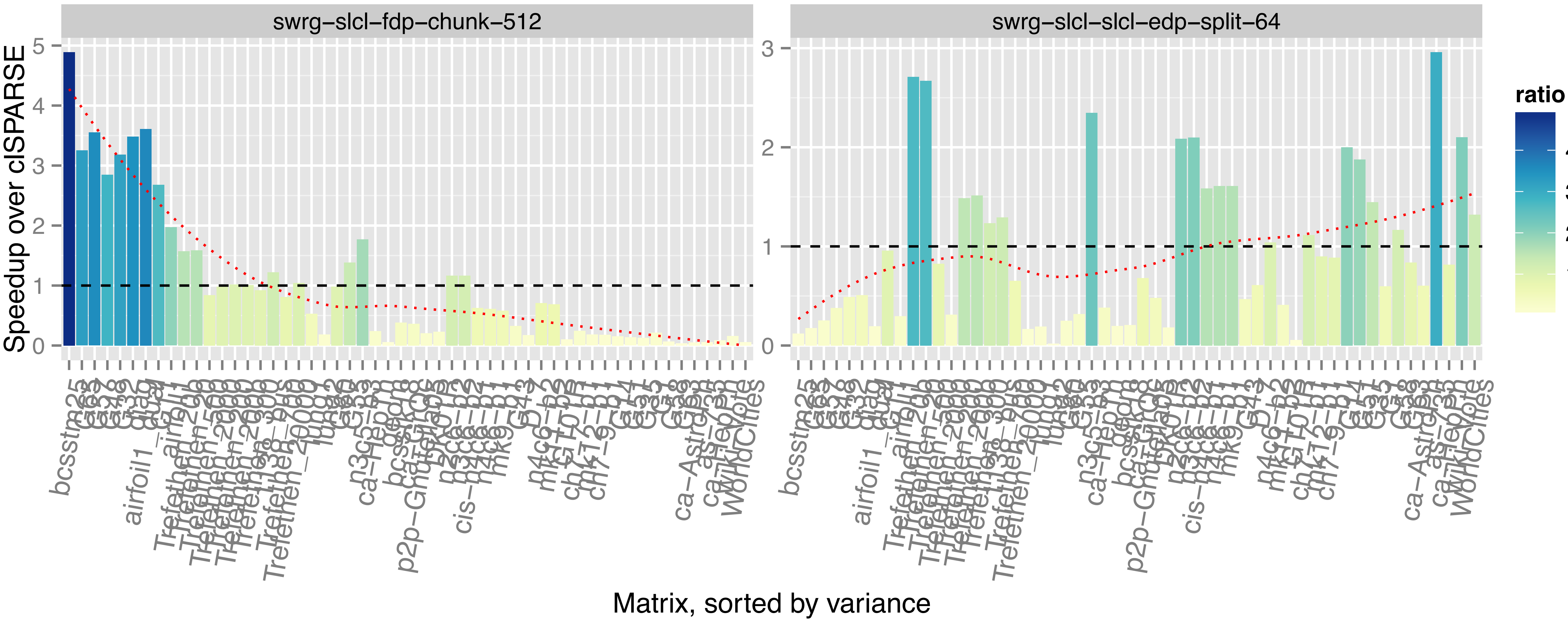


Speedup over clSPARSE
Matrix, sorted by variance
ratio
bcsstm25
G65
G57
G48
G32
diag
airfoil1_dual
G11
airfoil1
Trefethen_20b
Trefethen_20
Trefethen_500
Trefethen_2000
Trefethen_200
Trefethen_200b
Trefethen_300
1138_bus
Trefethen_700
Trefethen_20000
lung1
lung2
G60
G55
n3c5−b3
ca−HepTh
geom
bcsstk18
ca−GrQc
p2p−Gnutella05
DK01R
n2c6−b2
n3c6−b2
cis−n4c6−b1
n4c6−b1
mk9−b1
G43
D_7
n4c6−b2
mk11−b2
GT01R
ch7−7−b1
mk12−b1
ch7−9−b1
G14
G51
G35
G1
G58
G63
ca−AstroPh
as−735
ca−HepPh
wiki−Vote
WorldCities

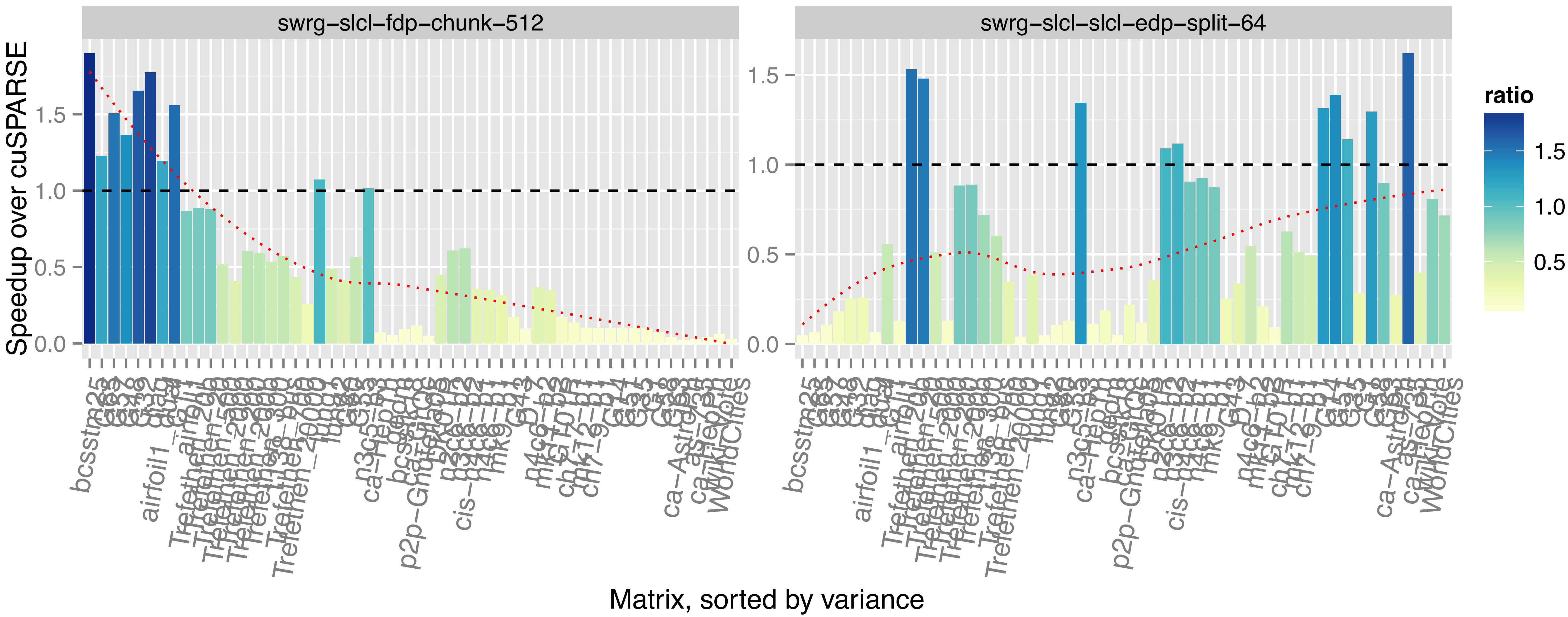
AMD


Speedup over cuSPARSE
0.0
0.5
1.0
1.5
2.0
ratio
bcsstm25
G65
G57
G48
G32
diag
airfoil1_dual
G11
airfoil1
Trefethen_20b
Trefethen_20
Trefethen_500
Trefethen_2000
Trefethen_200
Trefethen_200b
Trefethen_300
1138_bus
Trefethen_700
Trefethen_20000
lung1
lung2
G60
G55
n3c5−b3
ca−HepTh
geom
bcsstk18
ca−GrQc
p2p−Gnutella05
DK01R
n2c6−b2
n3c6−b2
cis−n4c6−b1
n4c6−b1
mk9−b1
G43
D_7
n4c6−b2
mk11−b2
GT01R
ch7−7−b1
mk12−b1
ch7−9−b1
G14
G51
G35
G1
G58
G63
ca−AstroPh
as−735
ca−HepPh
wiki−Vote
WorldCities
Matrix, sorted by variance

NVIDIA


swrg−slcl−fdp−chunk−512
swrg−slcl−slcl−edp−split−64
Speedup over clSPARSE
Matrix, sorted by variance
ratio

AMD


swrg–slcl–fdp–chunk–512
swrg–slcl–slcl–edp–split–64
Speedup over cuSPARSE
0.0
0.5
1.0
1.5
Matrix, sorted by variance
ratio
0.5
1.0
1.5

NVIDIA