AMG on the GPU

Exposing fine-grained parallelism

Copper 2011

Nathan Bell, Nvidia
Steven Dalton, University of Illinois
Luke Olson, University of Illinois

Motivation 1: potential

1M d.o.f.
10 nonzero neighbors
1.5 cycle complexity
15M computations

2 for a V(1,1)-cycle
20 iterations
600M for solution

16 bytes for (data,row,col)
10 Gbytes of data to process

125 Gbyte/sec SpMV on the GPU?
~ **0.08 sec**

25 Gbyte/sec SpMV on host?
~ **0.4 sec**

Motivation 2: software + hardware

real acceleration units: dedicated computation, double precision, high speeds

useable software: CUDA + Thrust + Cusp

AMG "asks" for acceleration:

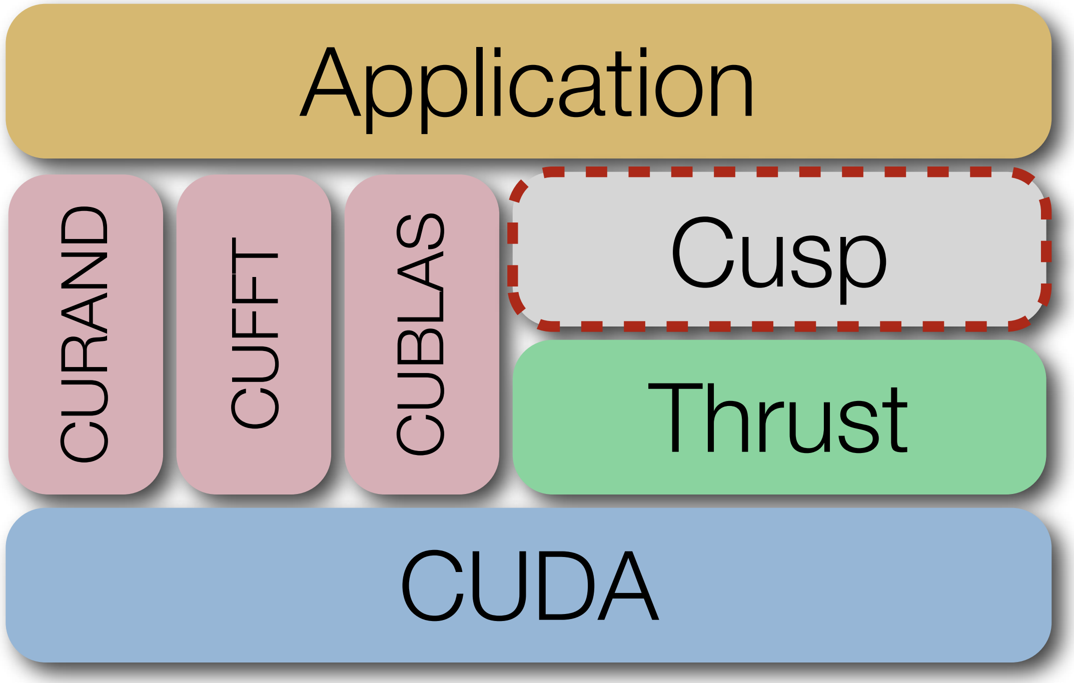
✓adaptive
✓multiple candidates, cycles
✓less/more aggressive coarsening
✓multiple RHS

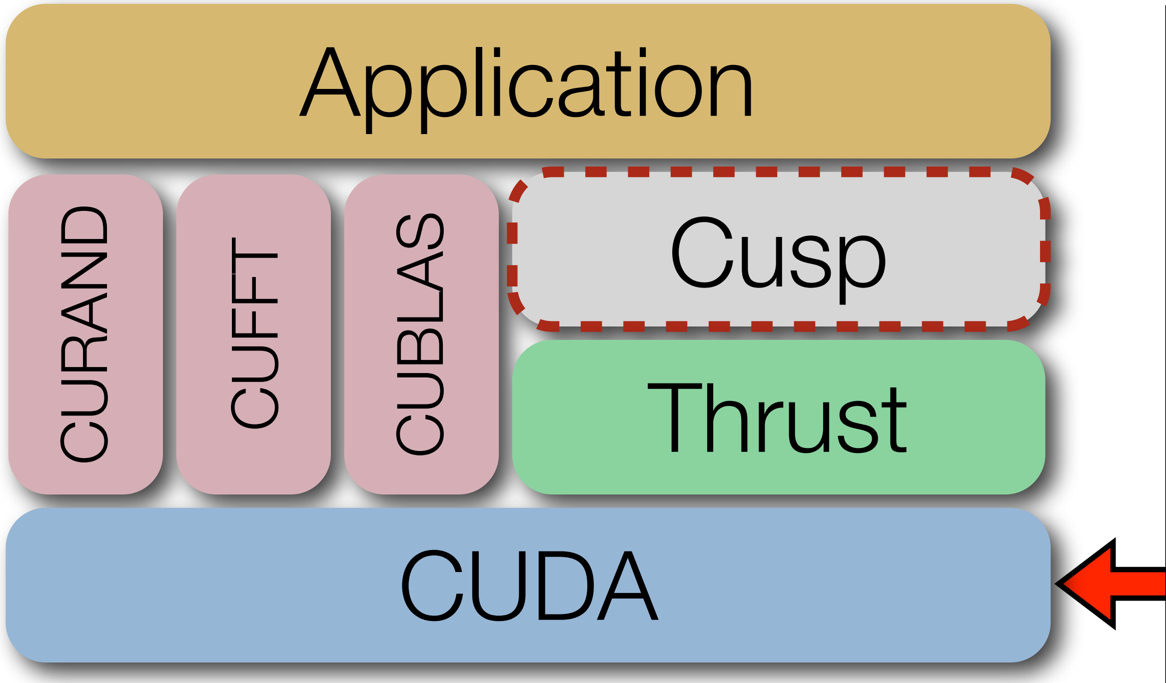
I. some basics about the software

II. some details about parallelism

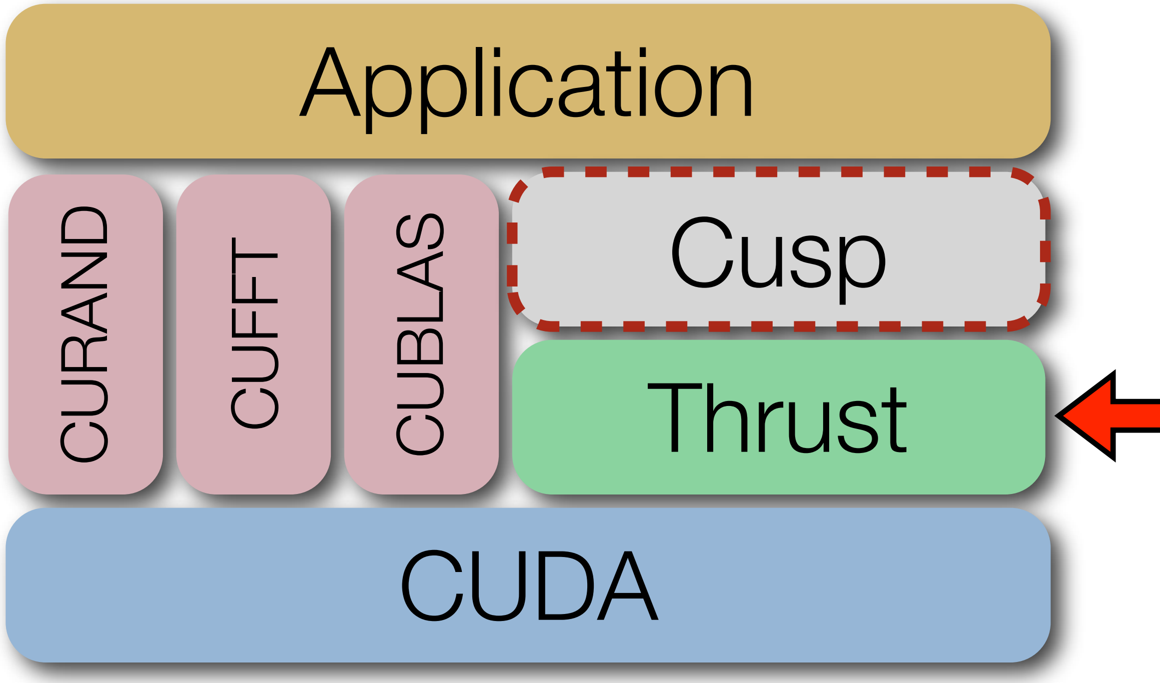
III. some results on performance

goal: an efficient, useable, extensible AMG algorithm on the GPU


Application
CURAND
CUFFT
CUBLAS
Cusp
Thrust
CUDA



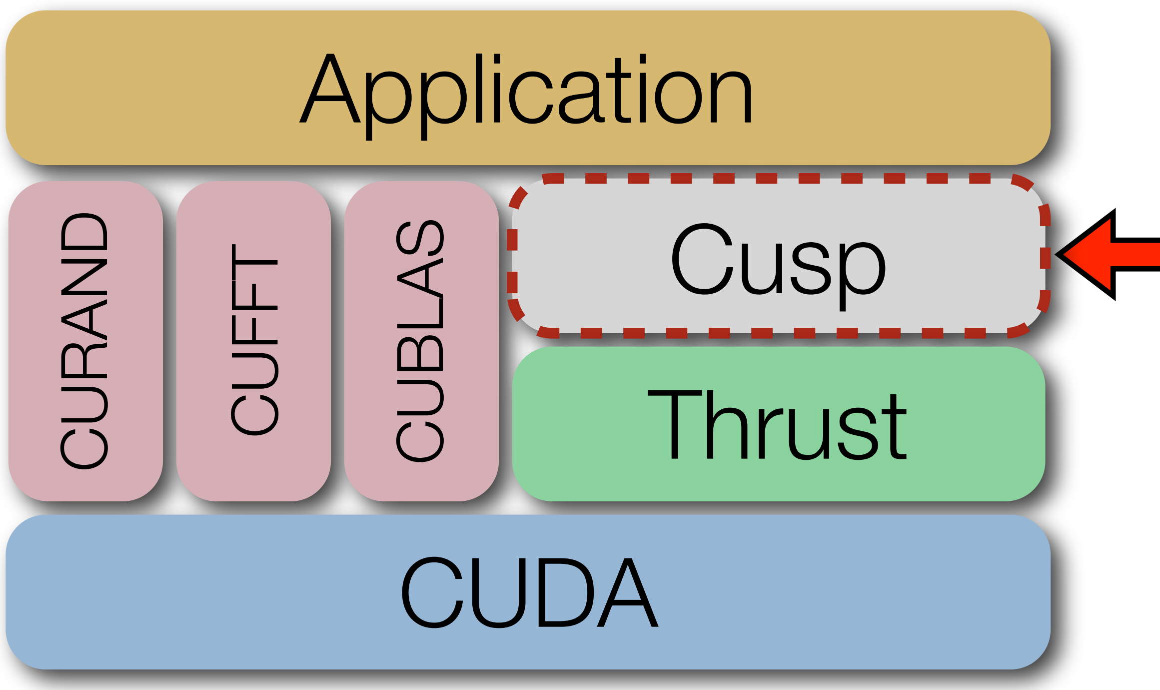

```
__global__ void function(float *a)
{
  // perform action on device
}

int main( int argc, char * argv[] )
{
  float *a_h, *a_d;
  float *b_h;
  int N = 10000;
  size_t size = N * sizeof(float);
  a_h = (float *)malloc(size);      // allocate size bytes on host
  cudaMalloc((void **) &a_d, size);// allocate same on device

  // initialize a_h

  // copy data to device
  cudaMemcpy(a_d, a_h,
             sizeof(float)*size, cudaMemcpyHostToDevice);

  int blockSize = 4;                   // threads per block
  int nBlocks = N/blockSize +...;      // # of blocks needed
  incrementArrayOnDevice <<< nBlocks, blockSize >>> (a_d);
  cudaMemcpy(b_h, a_d,
             sizeof(float)*N, cudaMemcpyDeviceToHost);

  free(a_h); free(b_h); cudaFree(a_d);  // deallocate
}
```



- fast development
- low overhead
- open source

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <cstdlib.h>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 * 1024 * 1024);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (846M keys per sec on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

- like C++ STL for CUDA
- usability
- containers and algorithms on host and device



- fast development
- low overhead
- open source

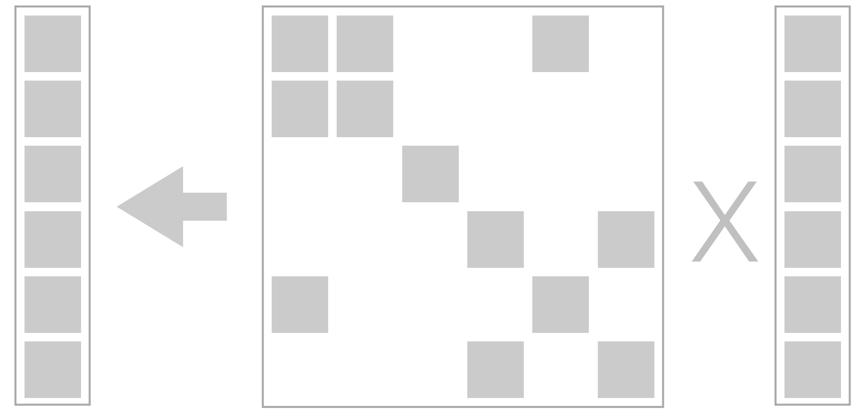
```
#include <cusp/hyb_matrix.h>
#include <cusp/io/matrix_market.h>
#include <cusp/krylov/cg.h>

int main(void)
{
    // create an empty sparse matrix structure (HYB format)
    cusp::hyb_matrix<int, float, cusp::device_memory> A;

    // load a matrix stored in MatrixMarket format
    cusp::io::read_matrix_market_file(A, "5pt_10x10.mtx");

    // allocate storage for solution (x) and right hand side (b)
    cusp::array1d<float, cusp::device_memory> x(A.num_rows, 0);
    cusp::array1d<float, cusp::device_memory> b(A.num_rows, 1);

    // solve the linear system A * x = b
    // with the Conjugate Gradient method
    cusp::krylov::cg(A, x, b);

    return 0;
}
```

- sparse matrix support
- solvers
- io
- “views” of other memory

Cusp

- Sparse matrix containers: COO, CSR, DIA, ELL, HYB
- Sparse Matrix-Vector multiply (SpMV)
- Sparse Matrix-Matrix multiply (SpMM)
- Transpose and format conversions
- Maximal independent sets
- Solvers

SpMV



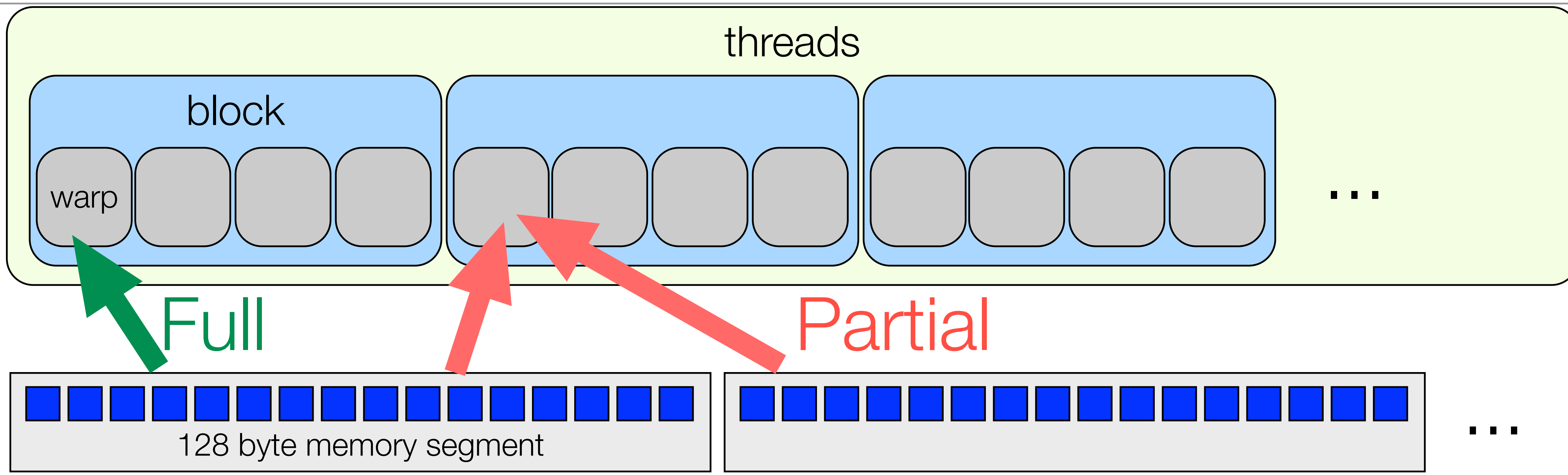
- *memory-bound*
- low arithmetic intensity (flop/mem. transfer)
- Tesla C2050 peak (double):

515 GFLOP/s performance
144 GB/s bandwidth
3.57 FLOP/byte

- Typical SpMV:

5-20 GFLOP/s performance
100 GB/s bandwidth
0.1 FLOP:bytes (2:16)

SpMV: memory coalescing and parallelism



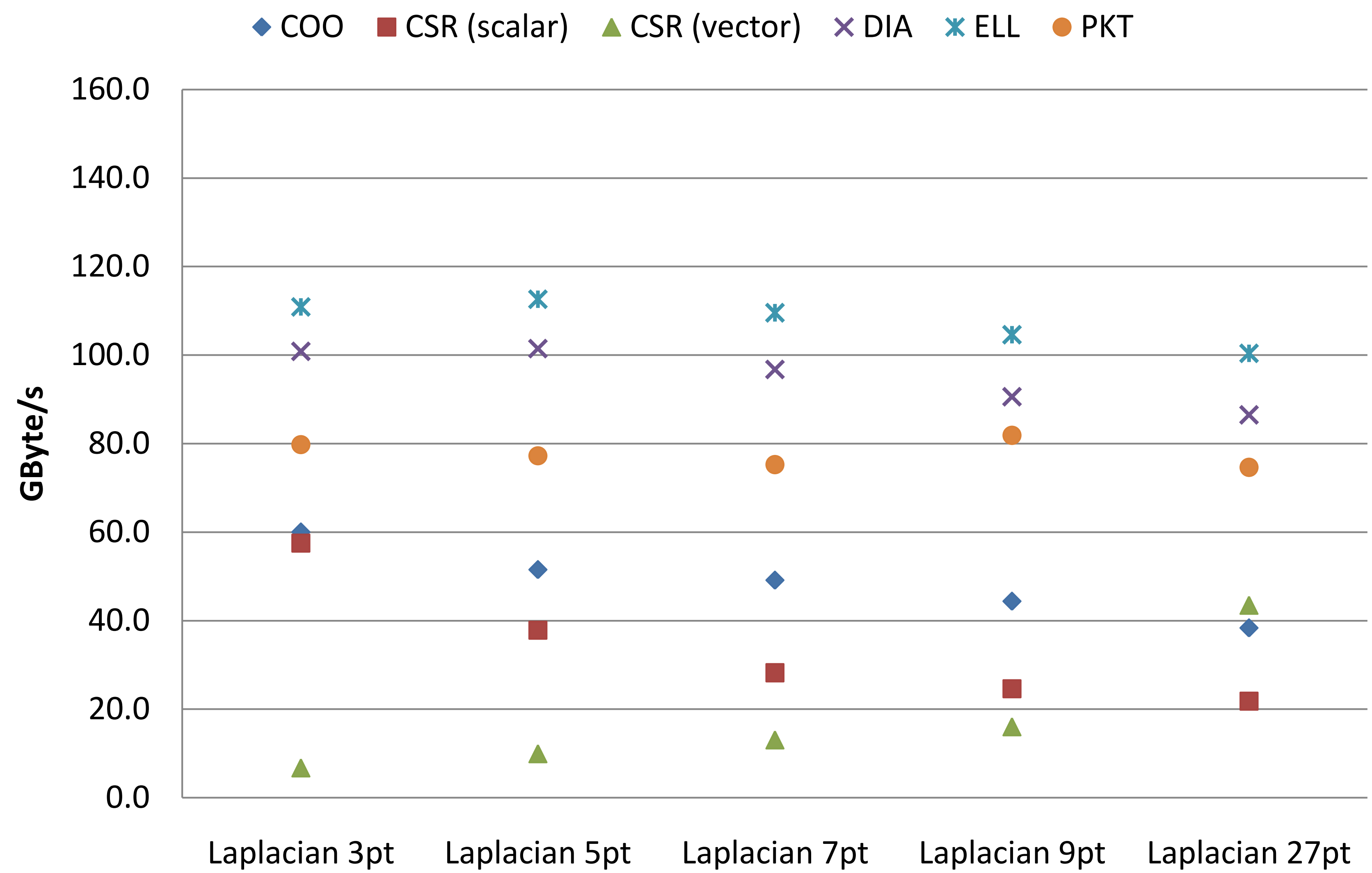
- **DIA, ELL, CSR(scalar)**
  - one thread per row
- **CSR(vector)**
  - one warp per row
- **COO**
  - one thread per nonzero

} full, needs many rows

} partial, few rows sufficient

} low, insensitive to # of rows

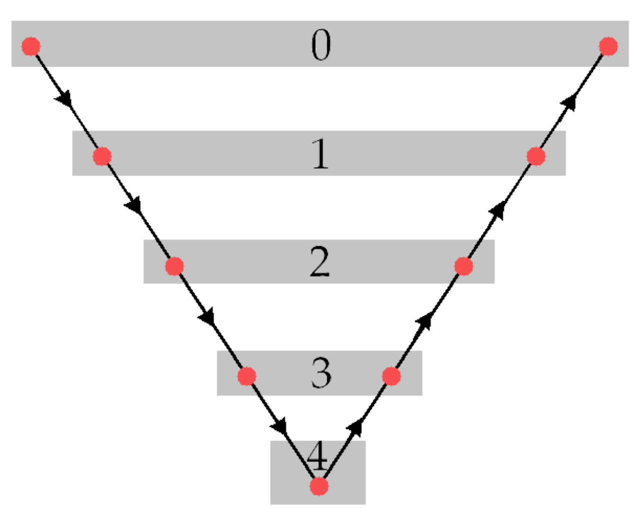
SpMV results

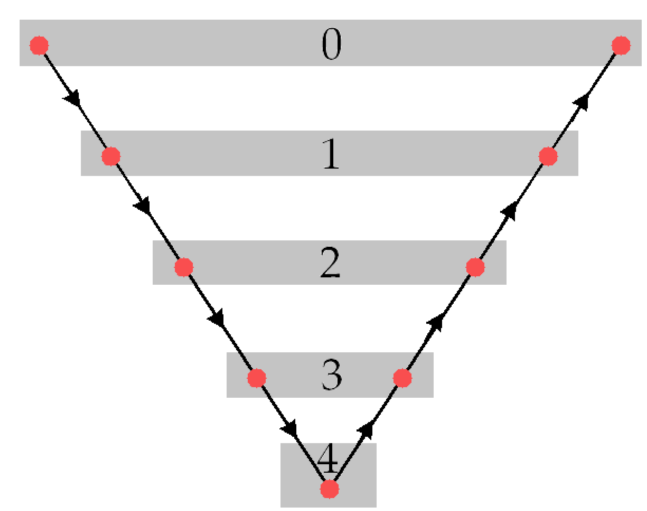


- Summary: structure formats do better with structure
- Results are often a toss-up as structure is lost (think coarse grids)

AMG Components: what we need

- a strength routine S_{ij}
- a threaded aggregation method $S \rightarrow Agg$
- an interpolation assembly $B, Agg \rightarrow T$
- smoothing interpolation $T \rightarrow P$
- coarse-grid operator $P^T A P$
- cycling





Setup

Solve

Strength-of-connection

parameters: A
return: S

$D \leftarrow 0$
for $n \in [0, nnz]$ **do**
    if $I_n = J_n$
        $D(I_n) = V_n$
    if $|V_n| > \theta\sqrt{|D(I_n)| \cdot |D(J_n)}$
        $S_{ij} = 1$

- simplifies to stream reduction (contraction)
- use

```
thrust::copy_if(V, V + N, result, is_even());
```

Aggregation

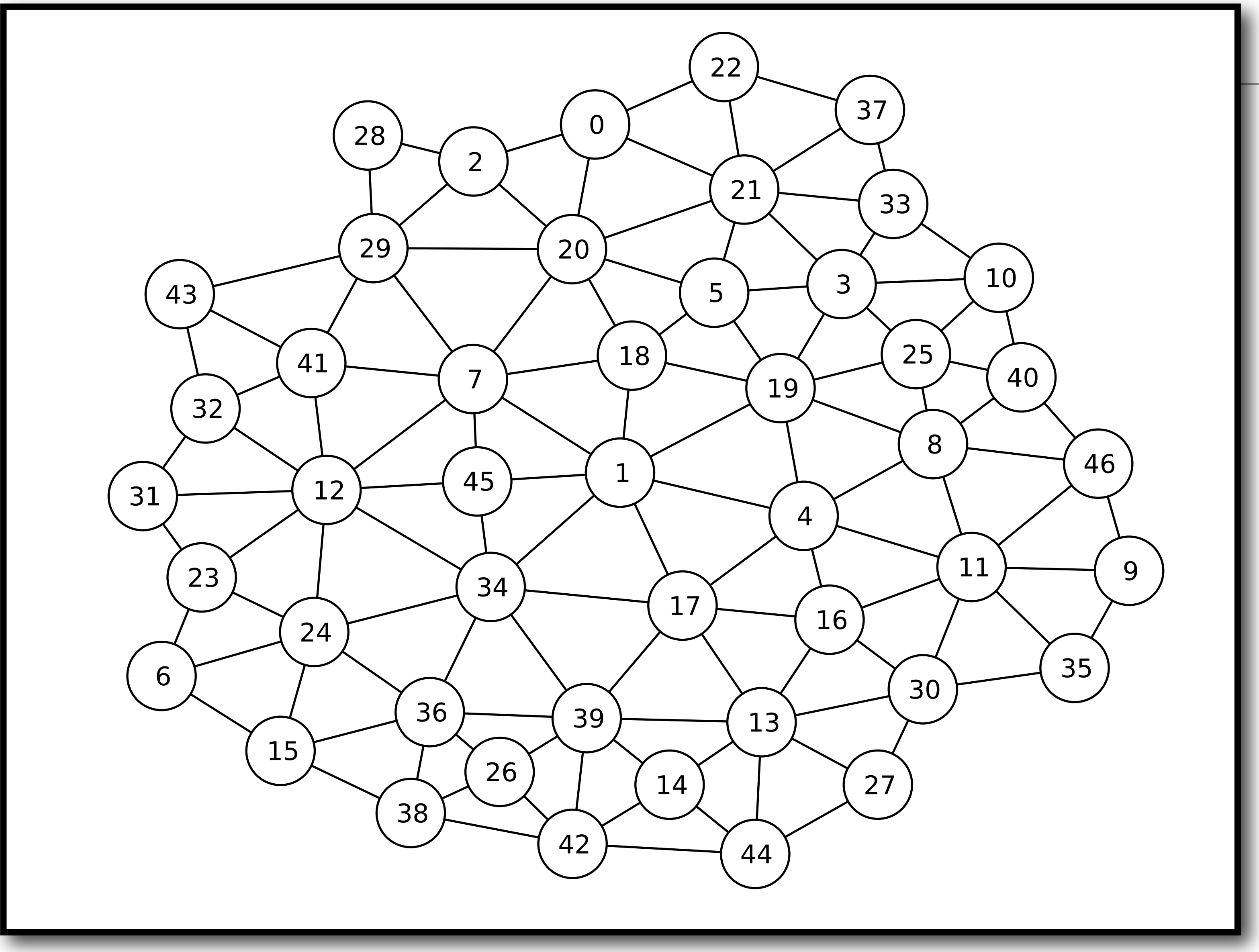
- standard aggregation: greedy, sequentially dependent
- “new” threaded aggregation: generalized modified independent sets

standard = threaded

(up to matrix permutation)

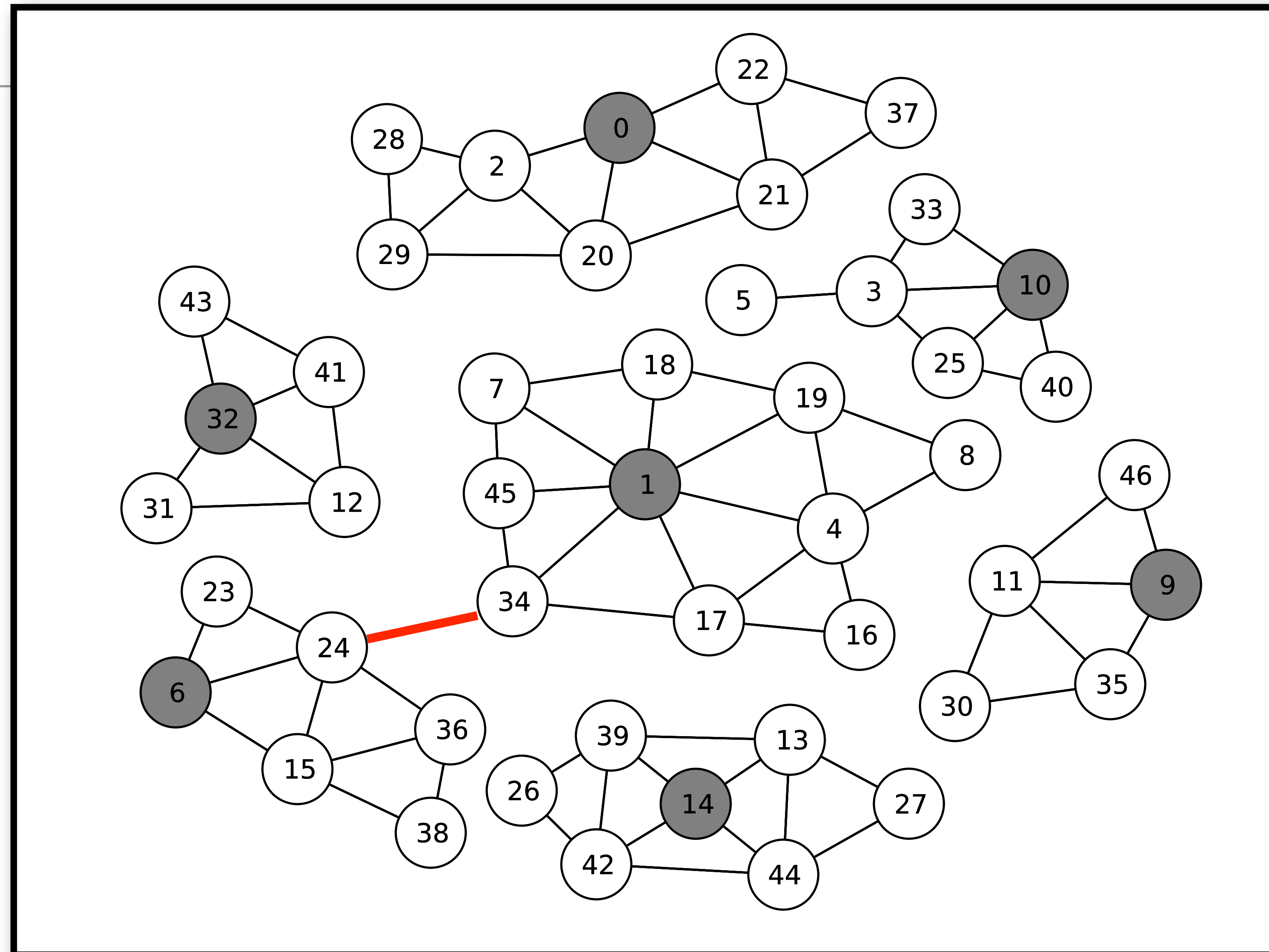
- memory access and parallelism resembles SpMV

MIS(2)



22
37
0
28
2
21
33
29
20
10
43
5
3
25
41
18
40
7
19
32
8
46
45
1
31
12
4
11
9
23
34
17
16
24
35
6
30
36
39
13
15
26
14
27
38
42
44

MIS(2)



independent

- root nodes more than 2 edges apart (> distance-2)
- an unaggregated node more than 2 edges from a root can become a root

MIS(2)

maximal

MIS(2)

1. Given a MIS(2) of {0,1} (N nodes)

2. prefix scan to enumerate

```
thrust::exclusive_scan(set, set + N, set, init)
```

3. communicate to neighbors aggregate index (**SpMV**)

~ std. first pass

4. communicate index to unaggregated neighbors (another **SpMV**)

~ std. second pass

- Generalizes to MIS(*k*)
- allows for variable coarsening

MIS(2)

1. Given a MIS(2) of {0,1} (N nodes)

2. prefix scan to enumerate

```
thrust::exclusive_scan(set, set + N, set, init)
```

3. communicate to neighbors
   aggregate index (**SpMV**)

} ~ std. first pass

4. communicate index to unaggregated
   neighbors (another **SpMV**)

} ~ std. second pass

- Generalizes to MIS(*k*)
- allows for variable coarsening

Transpose, Smoothing, spectral radius

- Smoothing: all **SpMV**s with GS or wJ
- Spectral radius: all **SpMV**s with Arnoldi***
- Transpose: **sort_by_key** (millions integer keys/sec)

std::sort	tbb:parallel_sort	thrust::sort
10.6	35.1	804.8

  - keys: column index
  - values: (row,data) tuple

SpMM

- SMMP algorithm: very sequential
  - requires O(ncol) storage to determine entries of each sparse row
  - parallelism would require O(ncol) memory per thread
- Consider $C = A * B$

$$A = \begin{bmatrix} 5 & 10 & 0 \\ 15 & 0 & 20 \end{bmatrix}, = \begin{bmatrix} (0,0,\ \ 5) \\ (0,1,10) \\ (1,0,15) \\ (1,2,20) \end{bmatrix}, \quad B = \begin{bmatrix} 25 & 0 & 30 \\ 0 & 35 & 40 \\ 45 & 0 & 50 \end{bmatrix}, = \begin{bmatrix} (0,0,25) \\ (0,2,30) \\ (1,1,35) \\ (1,2,40) \\ (2,0,45) \\ (2,2,50) \end{bmatrix},$$

1. form intermediate view of C
2. sort C by row, col
3. contract C by summing duplicates

SpMM

$$A = \begin{bmatrix} 5 & 10 & 0 \\ 15 & 0 & 20 \end{bmatrix}, B = \begin{bmatrix} 25 & 0 & 30 \\ 0 & 35 & 40 \\ 45 & 0 & 50 \end{bmatrix},$$

- expand with $A(i, j) * B(i, :)$

$$C = \begin{bmatrix} (0,0, & 125) \\ (0,2, & 150) \\ (0,1, & 350) \\ (0,2, & 400) \\ (1,0, & 375) \\ (1,2, & 450) \\ (1,0, & 900) \\ (1,2, & 1000) \end{bmatrix}$$

- all parallel gather, scatter, scan

```
thrust::gather(map, map + 10, input, out.begin())
thrust::scatter(input, input + 10, map, out.begin())
thrust::inclusive_scan(data, data + 6, data)
```

- sort by column keys

$$C = \begin{bmatrix} (0,0, & 125) \\ (0,1, & 350) \\ (0,2, & 150) \\ (0,2, & 400) \\ (1,0, & 375) \\ (1,0, & 900) \\ (1,2, & 450) \\ (1,2, & 1000) \end{bmatrix}$$

```
thrust::stable_sort(A, A + N);
```

SpMM

- parallel reduction

$$C = \begin{bmatrix} (0,0, & 125) \\ (0,1, & 350) \\ (0,2, & 550) \\ (1,0, & 1275) \\ (1,2, & 1450) \end{bmatrix} = \begin{bmatrix} 125 & 350 & 550 \\ 1275 & 0 & 1450 \end{bmatrix}.$$

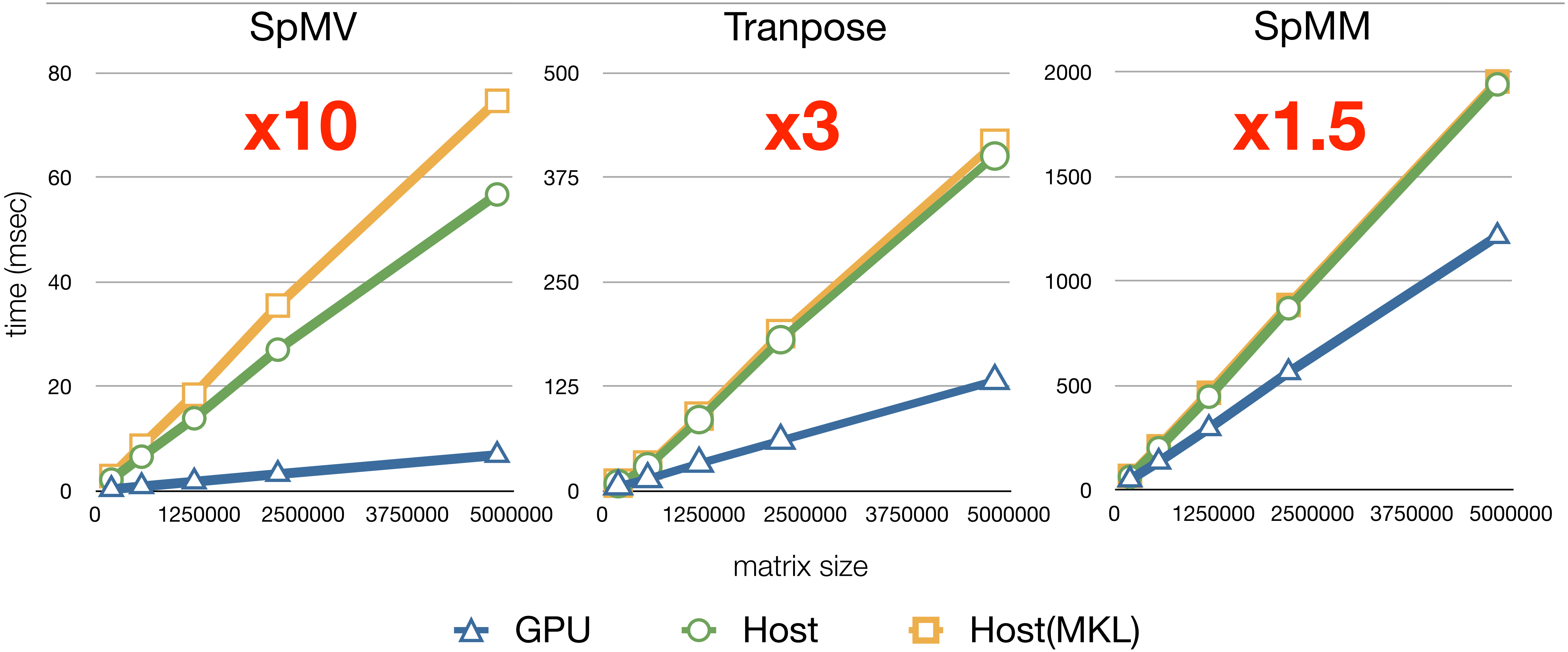
```
thrust::reduce_by_key(A, A + N, B, C, D)
```

- insensitive to irregularity of input

- same “work” as SMMP

- storage cost can be large for intermediate (reduce by subdividing)

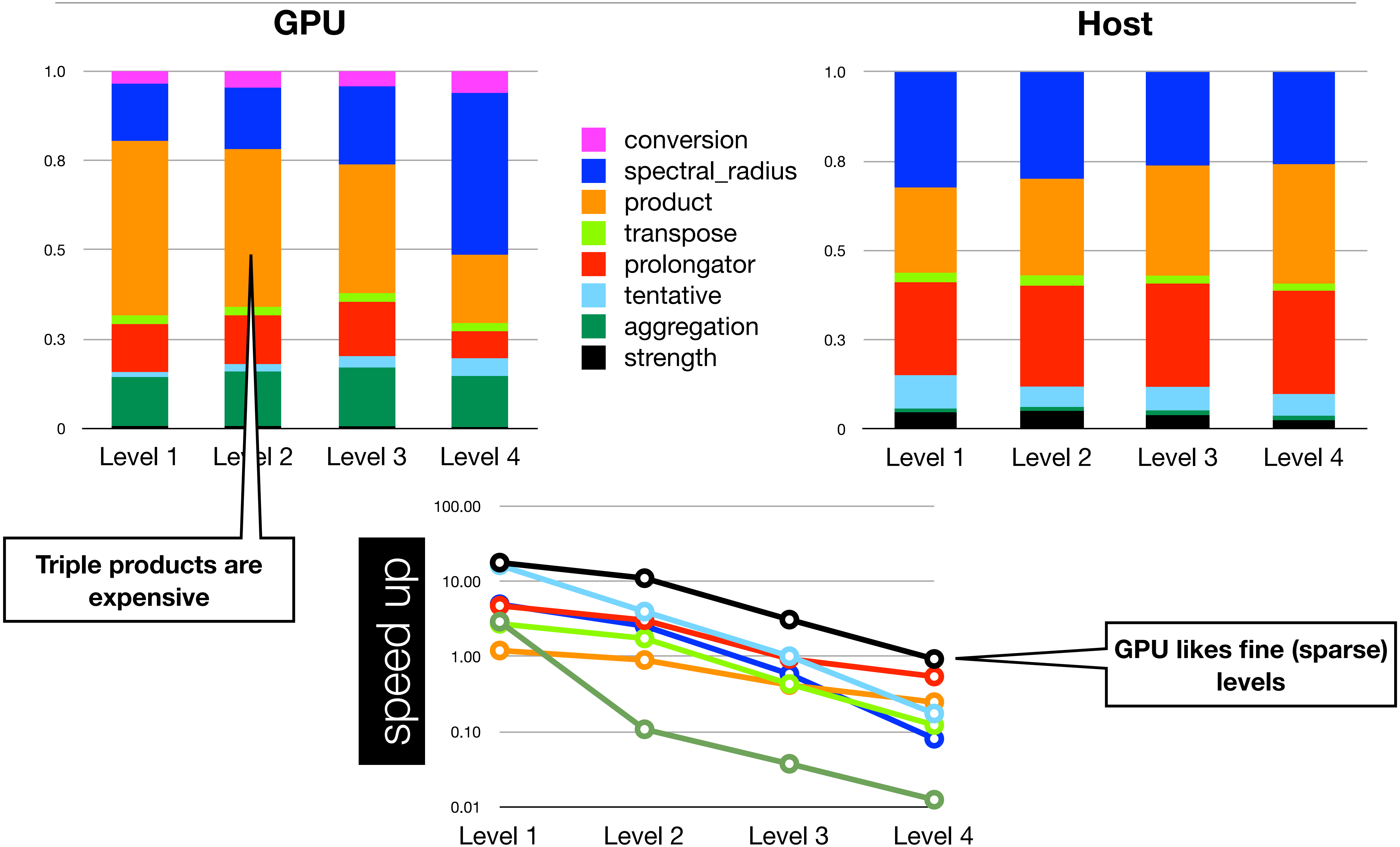
Setup kernels: SpMM, Transpose, SpMV



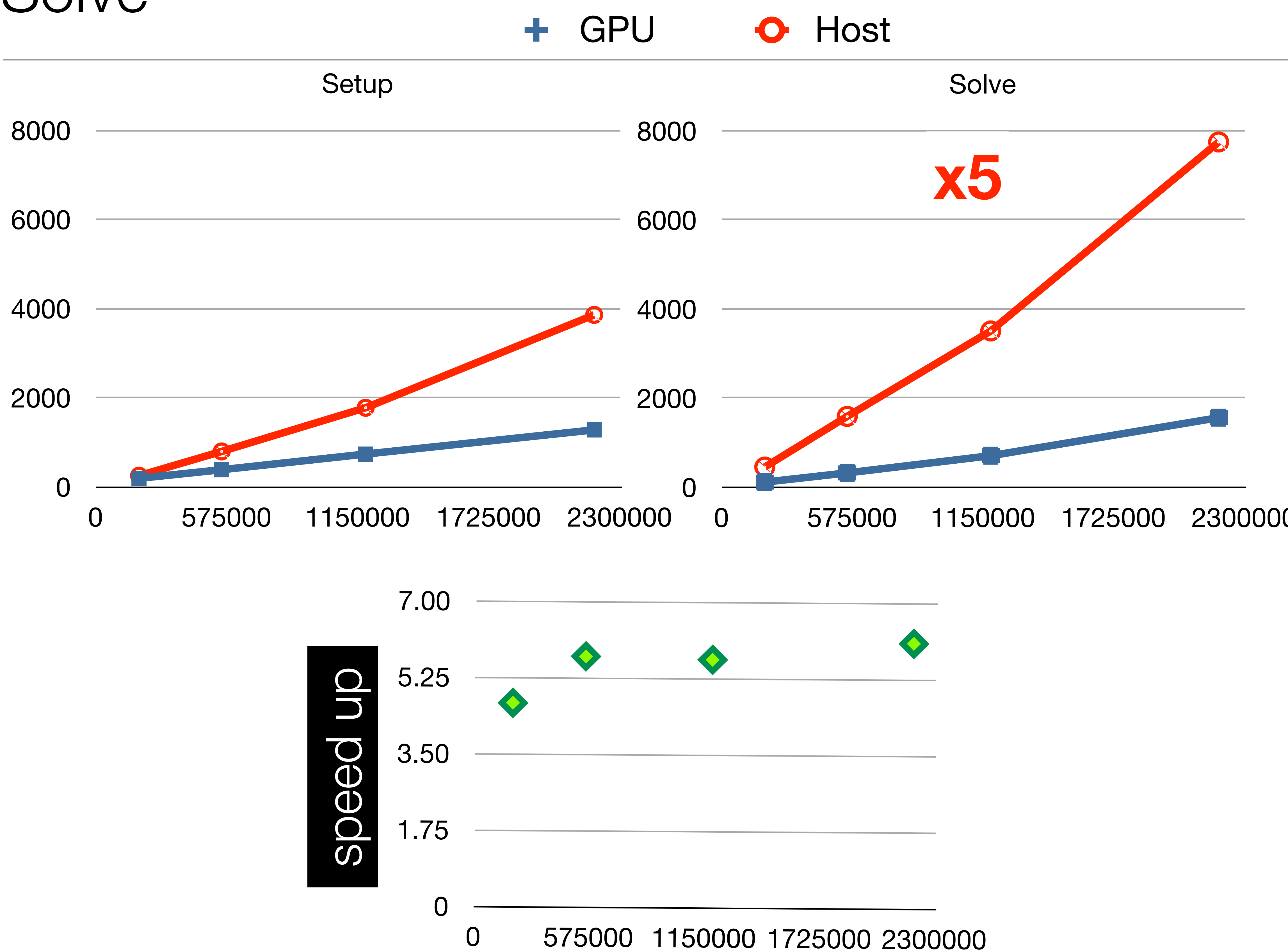
speed up

size	A*B	BT*(A*B)	BT*A*B	Tranpose	SpMV
5E+06	1.8	1.4	1.6	3.2	10.8
2E+06	1.7	1.4	1.6	3.1	10.9
1E+06	1.7	1.3	1.6	2.7	10.1
6E+05	1.7	1.3	1.5	2.1	9.5
2E+05	1.6	1.0	1.4	1.6	7.9

Setup components: total time



Solve


GPU
Host
Setup
Solve
8000
6000
4000
2000
0
0
575000
1150000
1725000
2300000
x5
speed up
7.00
5.25
3.50
1.75

Summary

exposed SA to **GPU kernels**

tangible speedups

identified **practical** directions for AMG to take advantage of fine grained parallelism

thanks to NVIDIA.

Sparse matrix conversion times (msec)

- sort, gather, scatter, scan

unstructured

2D
1M dof
8M nnz

From\To	COO	CSR	ELL	HYB
COO	5	6	20	24
CSR	8	4	22	25
ELL	17	18	6	22
HYB	63	69	83	4

3D
1M+ dof
17M nnz

From\To	COO	CSR	ELL	HYB
COO	10	12	57	56
CSR	15	8	60	59
ELL	72	74	19	71
HYB	139	152	196	10