A Study of the Potential of Locality-Aware Thread Scheduling for GPUs

Cedric Nugteren (presenter)
Gert-Jan van den Braak
Henk Corporaal

Eindhoven University of Technology

MuCoCoS-7 at Euro-Par '14
August 26, 2014

A GPU schedules threads in a particular order

Can we find a better thread schedule?

Poor programming can cause bad performance

Can we hide this behind smart hardware/software?

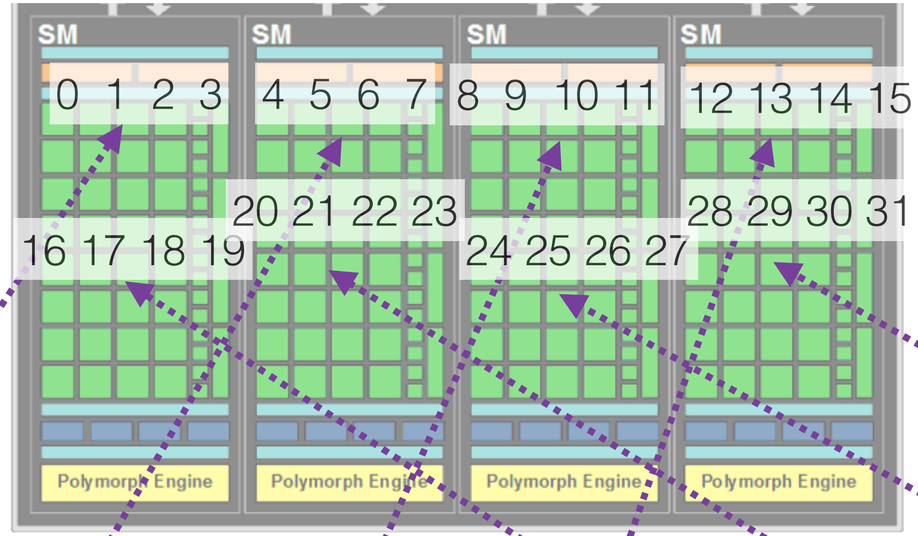
Thread scheduling affects performance

How much performance can we gain?

Thread scheduling

Warp **x** mod 4
onto SM **x**

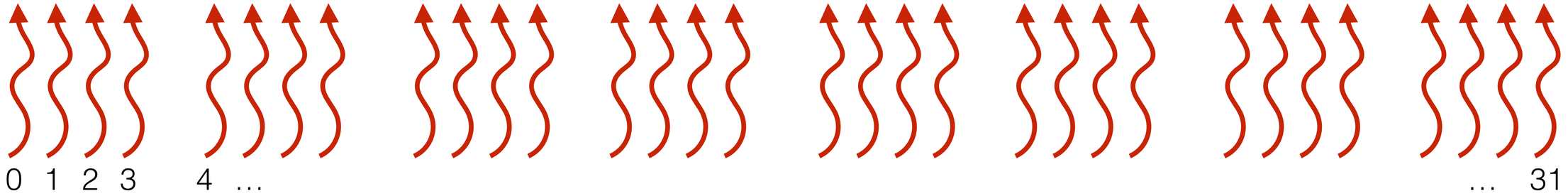
(example warp: 4 threads)



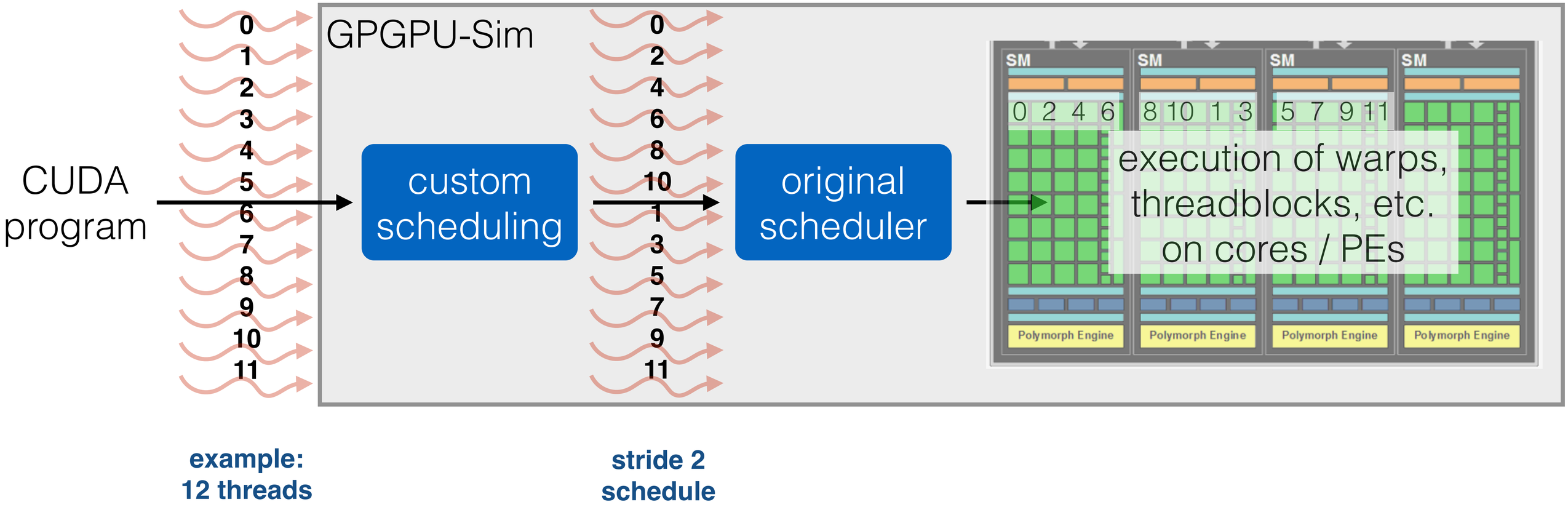
A different schedule affects:

- Cache-line locality (spatial)
- L1 and L2 locality (temporal)
- Memory coalescing

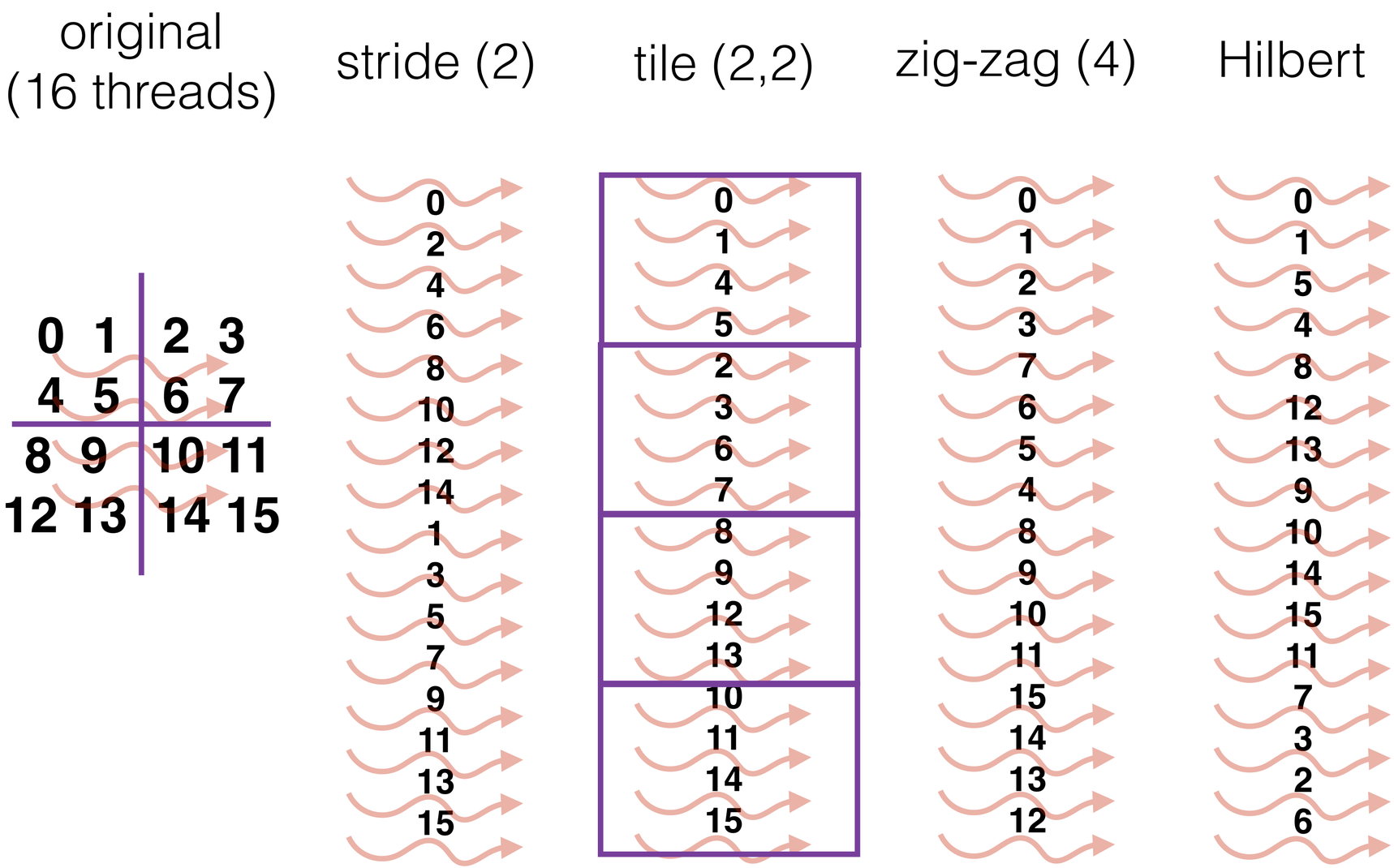
threads



Experimental set-up



Evaluated schedules



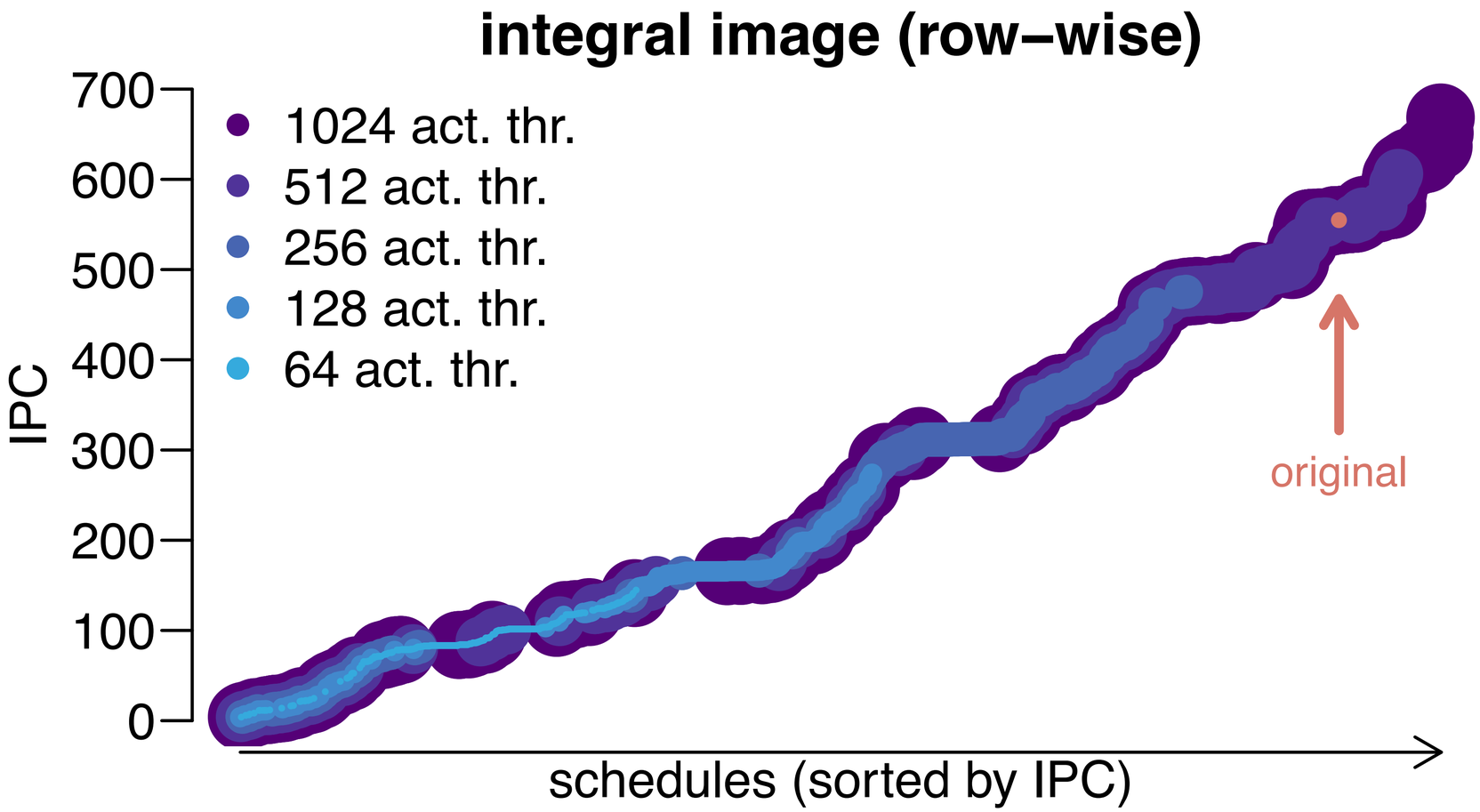
At granularity of:

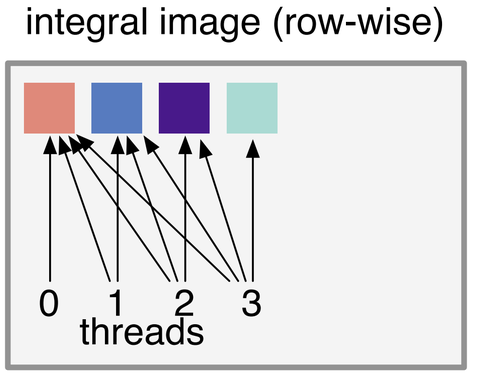
- Threads
- Half-warps / warps
- Threadblocks
- …

Total: 2000+ schedules

Tested with 6 ‘naive’ CUDA benchmarks

Experimental results (1/6)

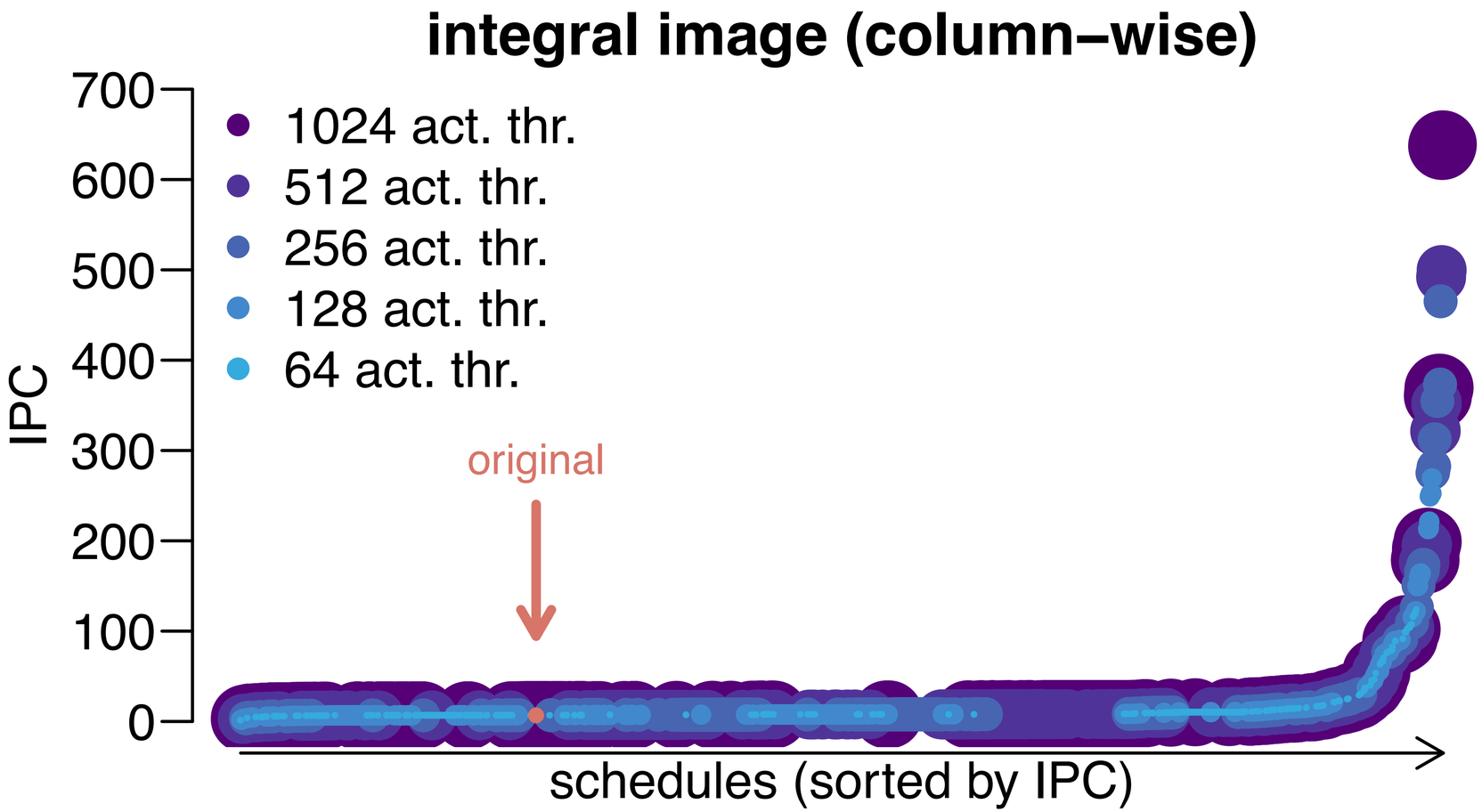




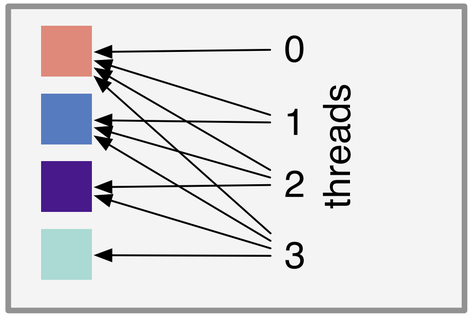
Observations:

- Wide performance range (2 - 700)
- Good original schedule
- 20% to gain by tiling
- Active thread count not too important

Experimental results (2/6)



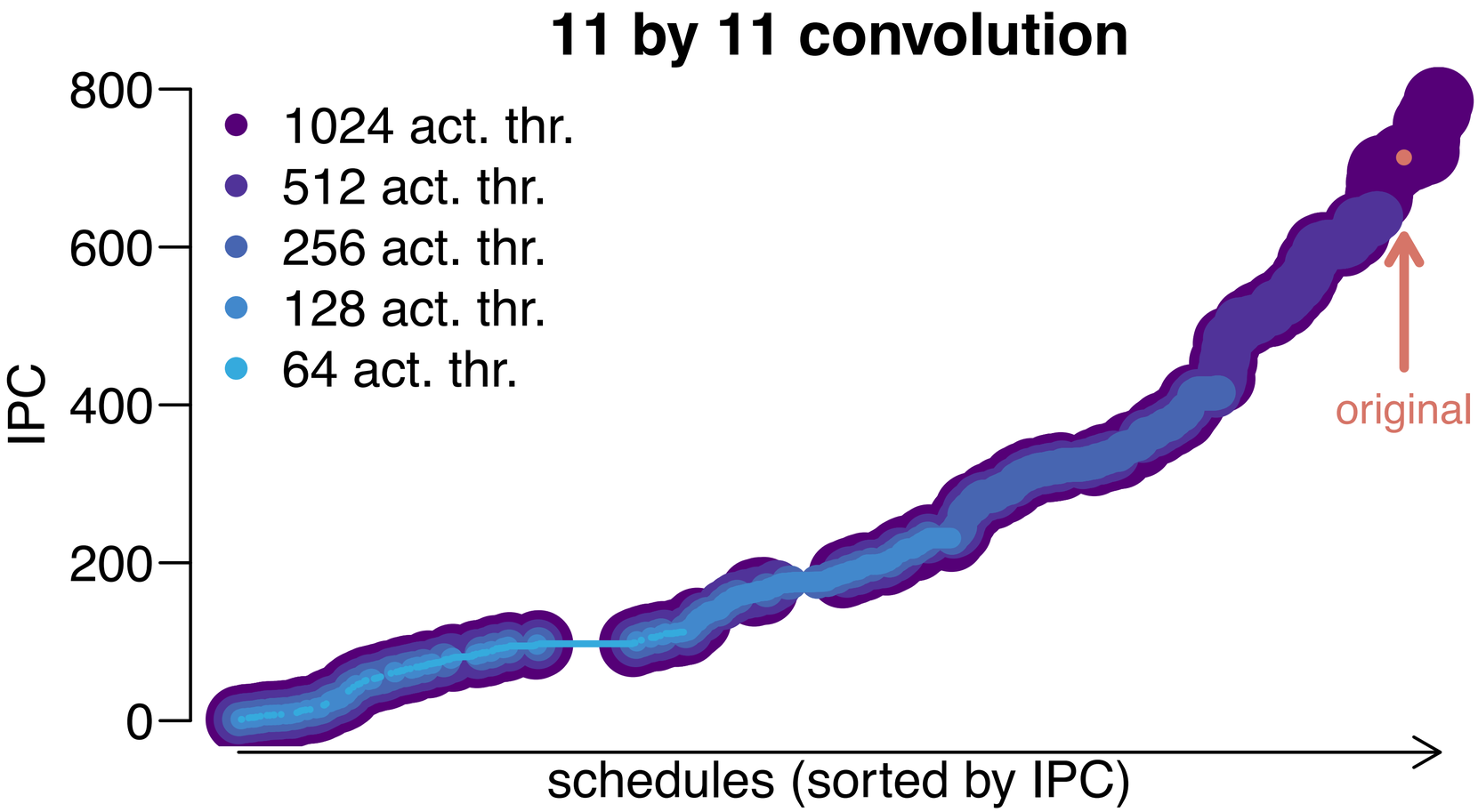
integral image (col-wise)

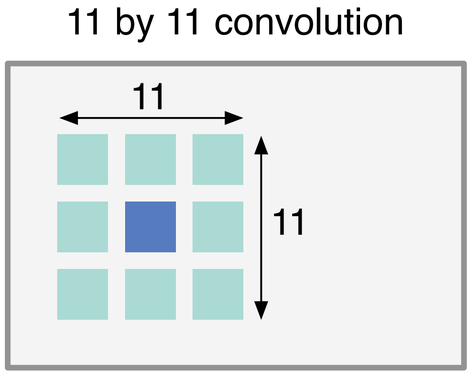


Observations:

- Original schedule with un-coalesced accesses and bad cache locality
- Schedules with a stride bring back the row-wise behaviour (and performance)

Experimental results (3/6)

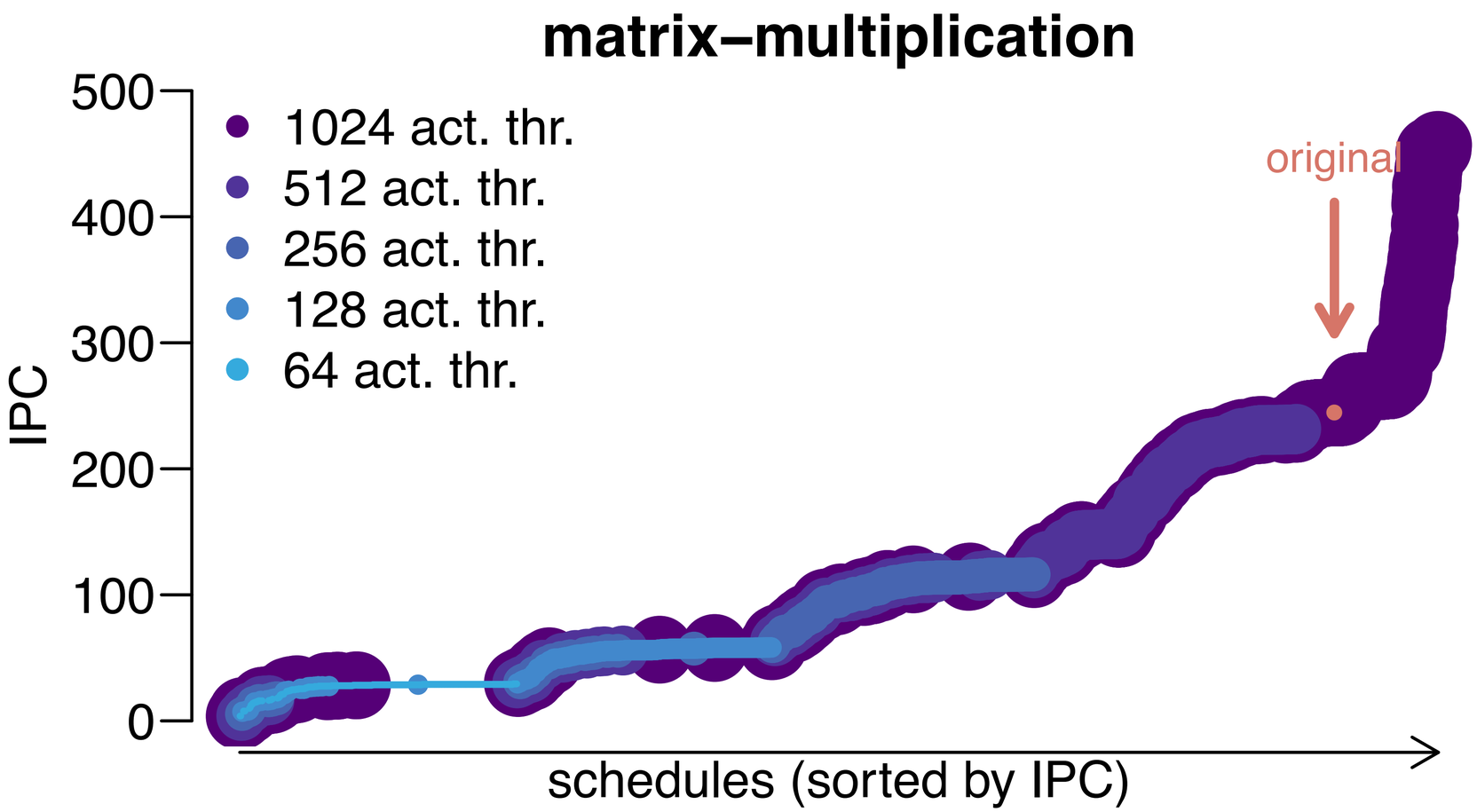




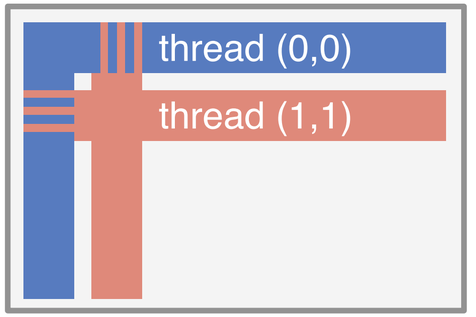
Observations:

- Wide performance range (2 - 800)
- Good ‘original’ schedule
- 10% to gain by using zigzag scheduling

Experimental results (4/6)



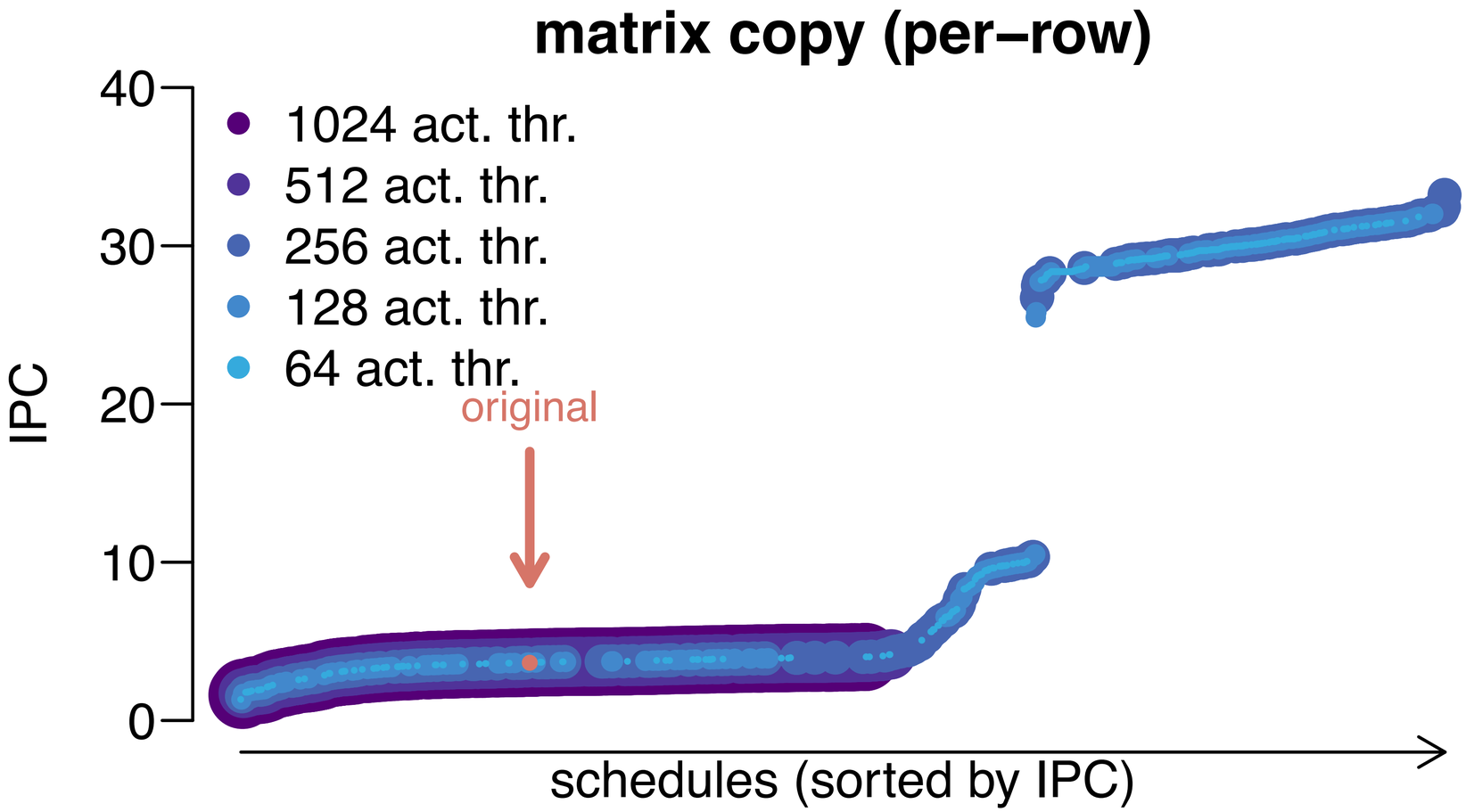
matrix multiplication

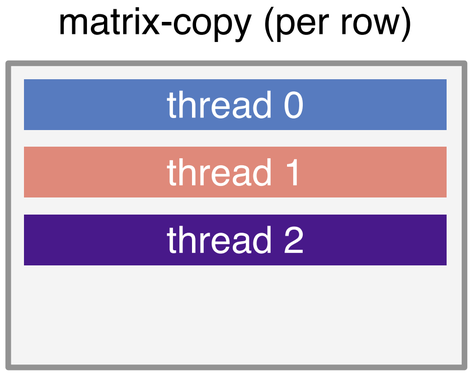


Observations:

- Wide performance range (2 - 500)
- Average original schedule
- 87% to gain by tiling
- Active thread count important

Experimental results (5/6)

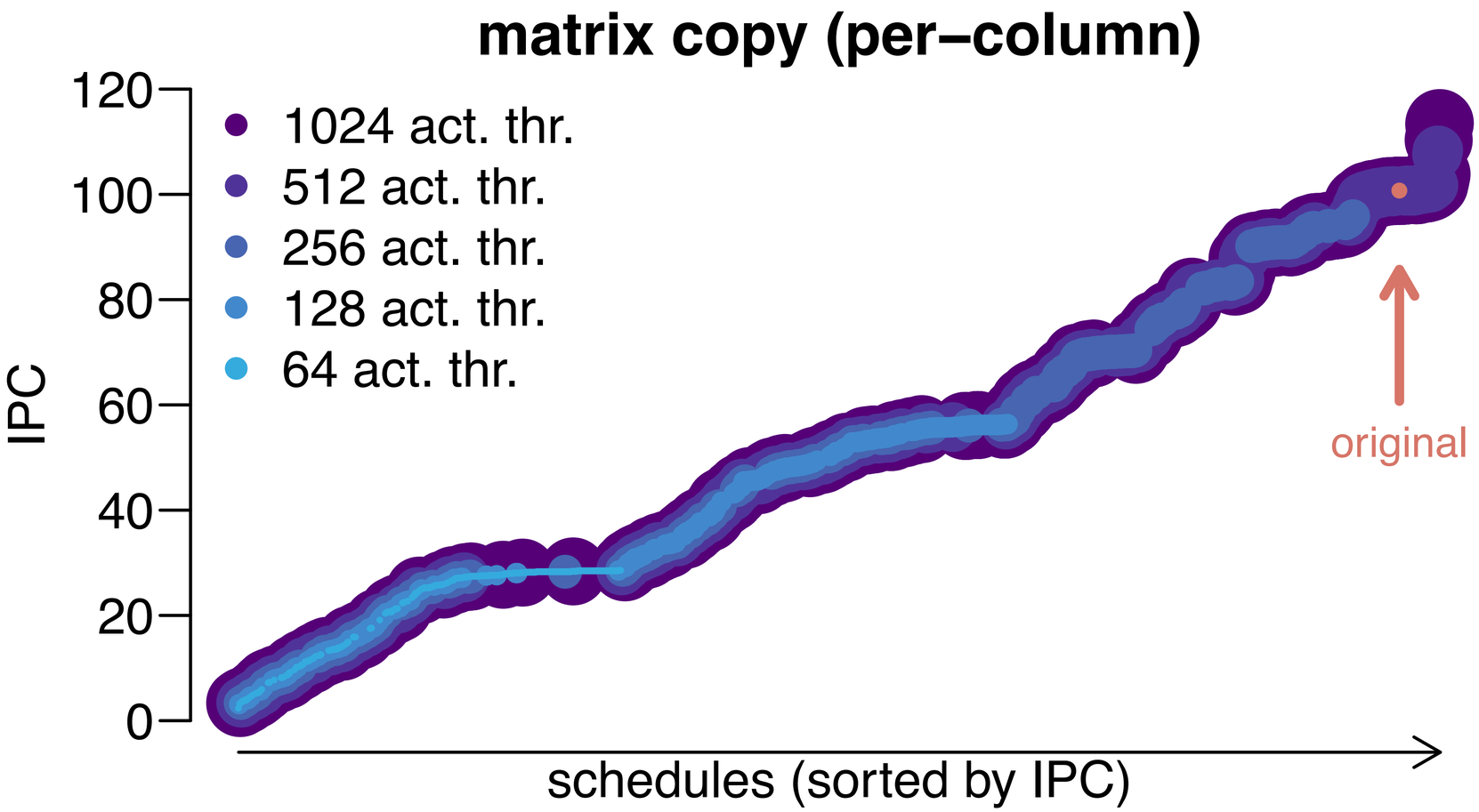


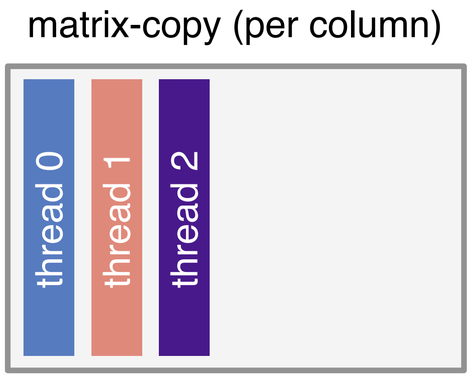


Observations:

- 64, 128, 256 active threads for best performance
- 512 and 1024 active threads suffer from cache contention
- Small IPC range

Experimental results (6/6)

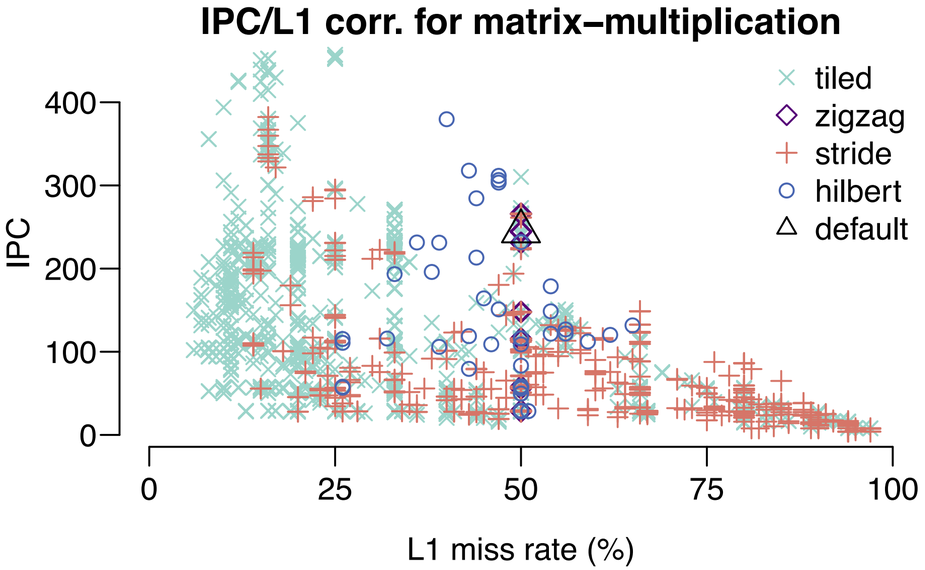


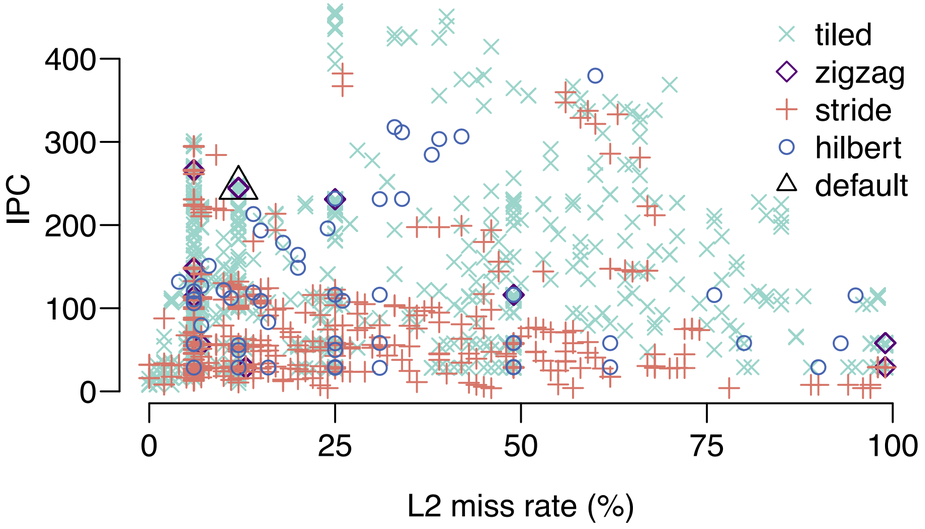


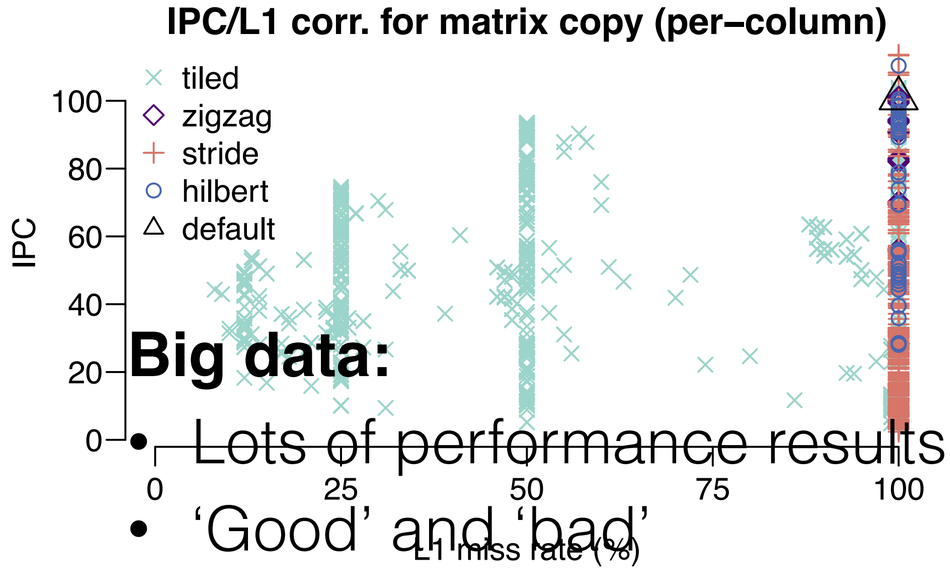
Observations:

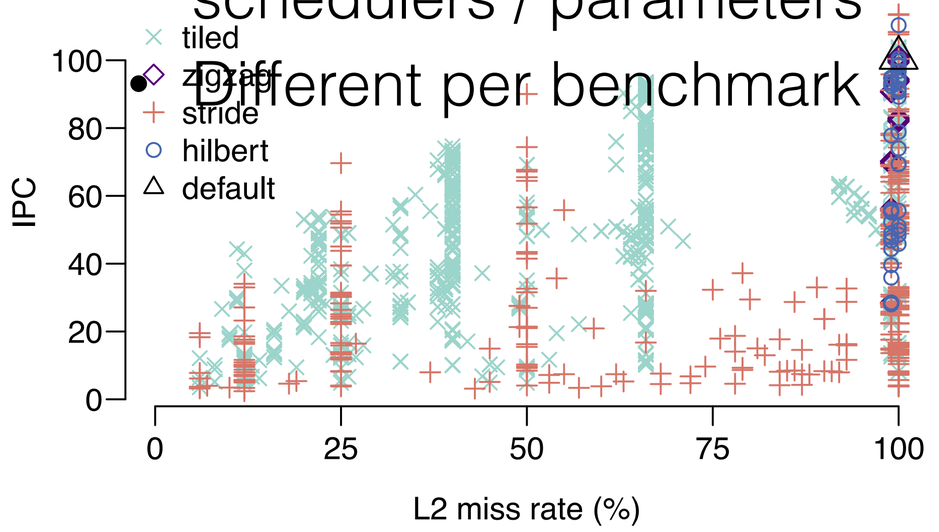
- Good ‘original’ schedule
- 12% to gain by tiling

Detailed results









Big data:

- Lots of performance results
- ‘Good’ and ‘bad’ schedulers / parameters
- Different per benchmark

OK, so there is performance potential

But how do we find a good schedule?

Idea: Calculate best schedule

Calculate best schedule:

- Calculate thread similarity:
  - Use static analysis where possible
  - Estimate/predict behaviour otherwise
- Use shortest path algorithms?
- Use graph-cut algorithms?

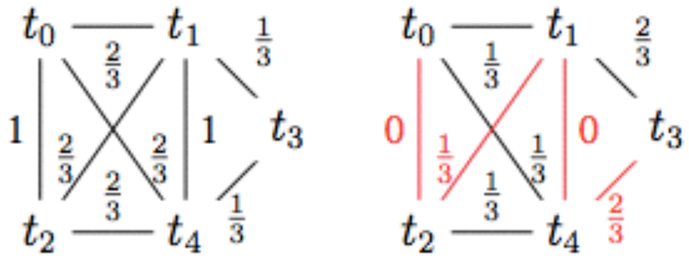
$$\Gamma_{ij\delta} = \frac{1}{\delta+1}\sum_{d=0}^{\delta}\gamma_{ijd}$$

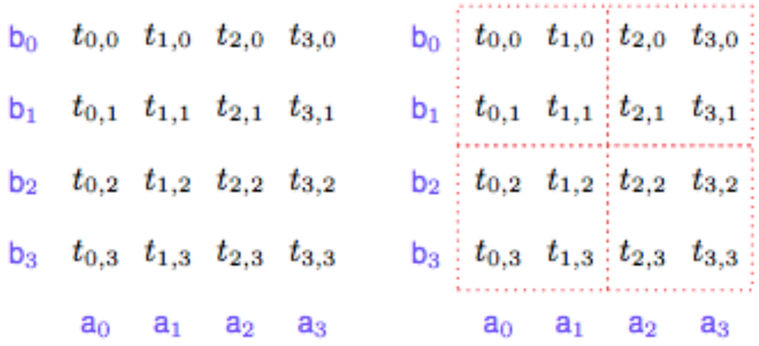
with

$$\gamma_{ij\delta} = \frac{1}{A-\delta}\sum_{k=0}^{A-\delta} equal(i_k, j_{k+\delta})$$

and

$$equal(a,b) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases}$$



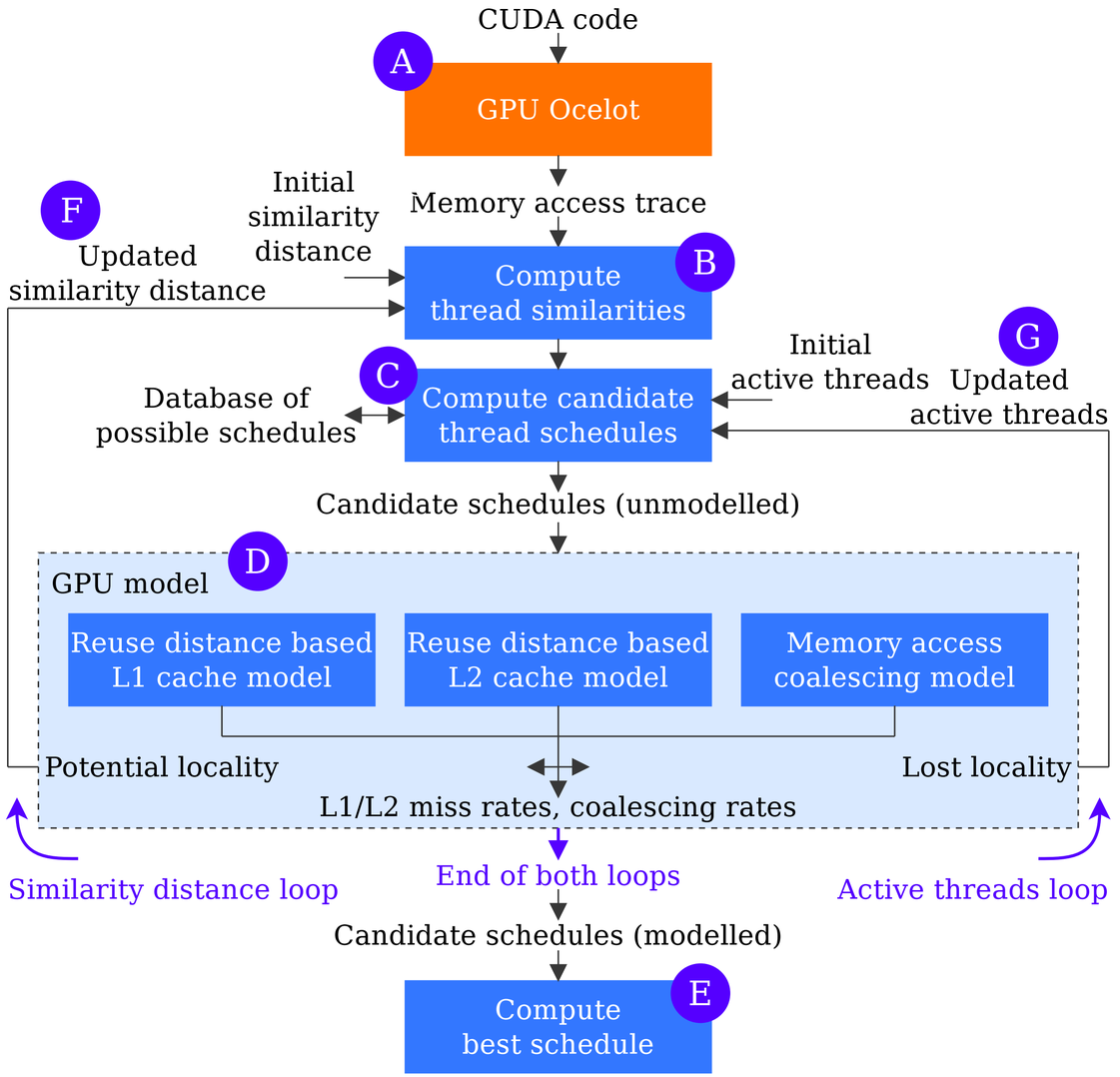


Idea: Model cache behaviour

Model cache behaviour:

- At compile-time
- Run potential schedules through a model:
  - Use a cache-only GPU model?
  - Use an analytical model?
- Use heuristics to limit potential schedules?

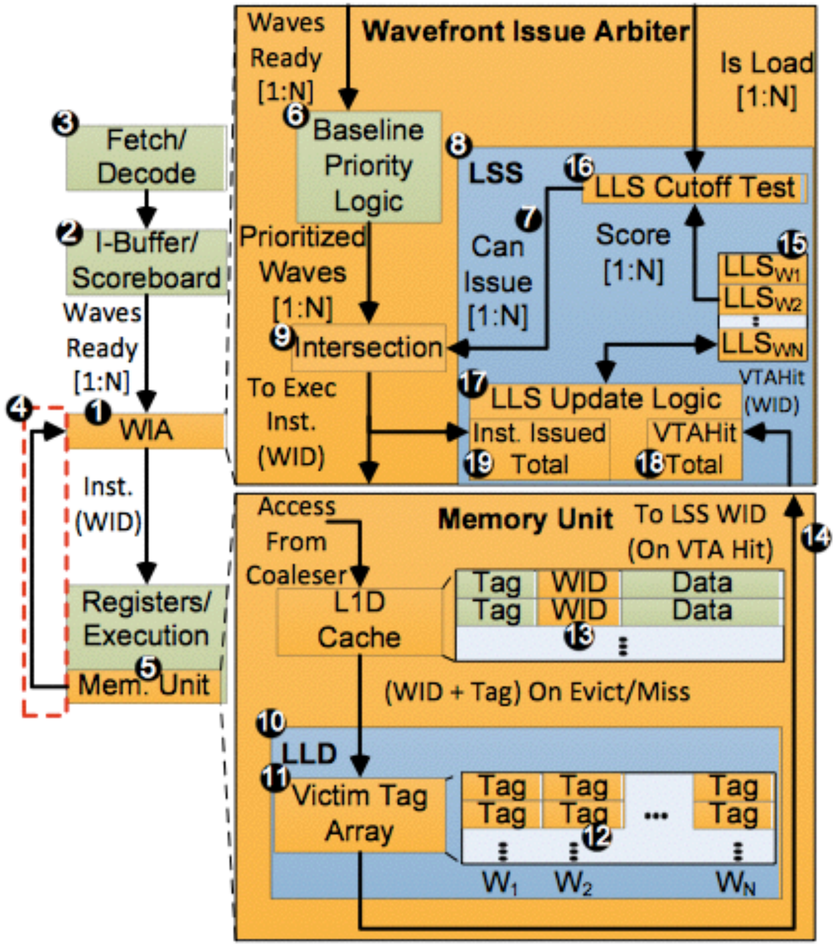
- Use auto-tuning or a neural network to train a model?



Idea: Use hardware counters

Use hardware counters:

- Change schedules on-the-fly
- Measure:
  - Cache hits/misses?
  - Lost locality?
- What is the hardware-overhead?
- How dynamic is the application?



Taken from T. Rogers et al.
“Cache-Conscious Wavefront Scheduling“

Conclusions

Conclusions:

- Locality-aware thread scheduling:
  - Is important for performance
  - Can improve programmability

Future work:

- Investigate how to find a good schedule