Case Studies

- Two simple examples of applying PCAM
- Not much detail, but they are actual solutions to real problems

Finite Difference

- Often works well for regular models based on physics
  - Navier-Stokes fluid dynamics
  - Maxwell's equations
- Also used as a general numerical technique
  - Initial boundary problems: solve for steady state within volume
  - Time domain marching: how does field change with time

- Convert continuous-domain problem into discrete-domain
  - Sample scalar or vector field at regular Cartesian co-ordinates
  - Iteratively update values in the field

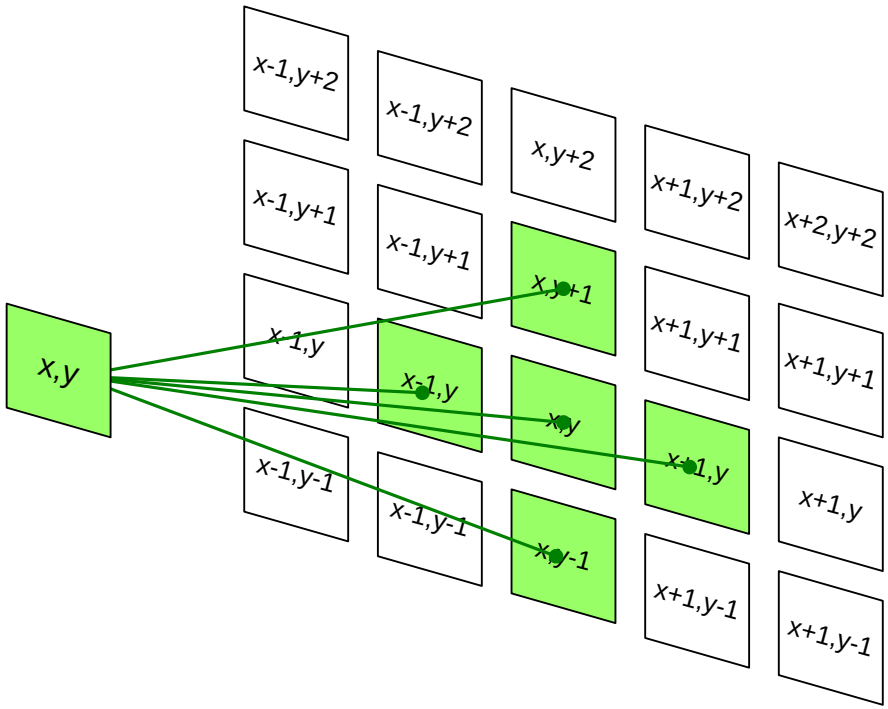
Warning: this is just an example, **not** a recommendation or hint

Parallelism

- 2D inner loop
  - Relaxes over n^2 grid
  - Apply 3x3 stencil
  - No dependencies
- 1D outer loop
  - Steps forward in time
  - Each step based on previous step
  - True dependency

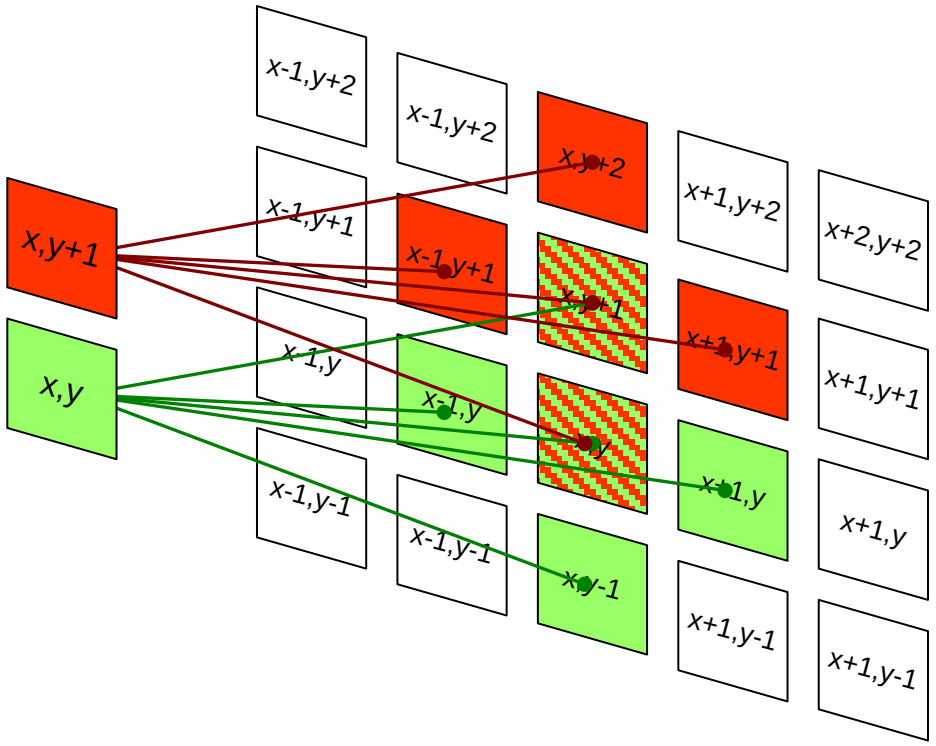
```
void FiniteDifferenceStep(
    int n,
    const double *f,
    double *nf
){
  for(int x=0;x<n;x++){
    for(int y=0;y<n;y++){
      double tmp =
                    f[x*n+y+1]
 + f[x*n+y-n] + f[x*n+y  ] + f[x*n+y+n] +
                    f[x*n+y-1];

      nf0[x*n+y]=tmp*0.2;
    }
}
```

Communication



- Each point depends on current value plus four neighbours

Communication

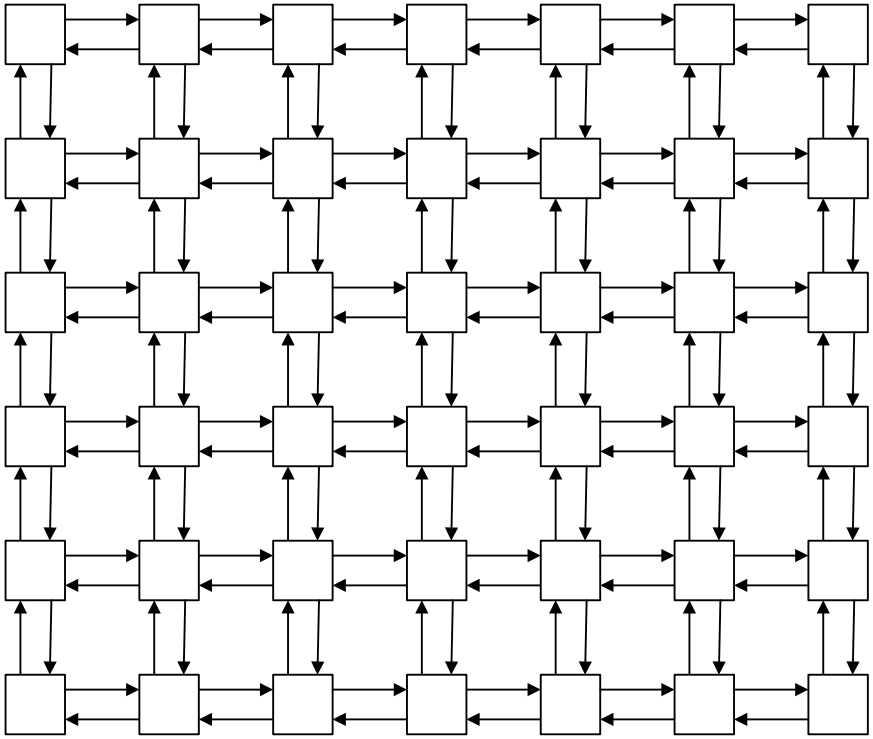


- Each point depends on current value plus four neighbours

There is shared data between adjacent elements

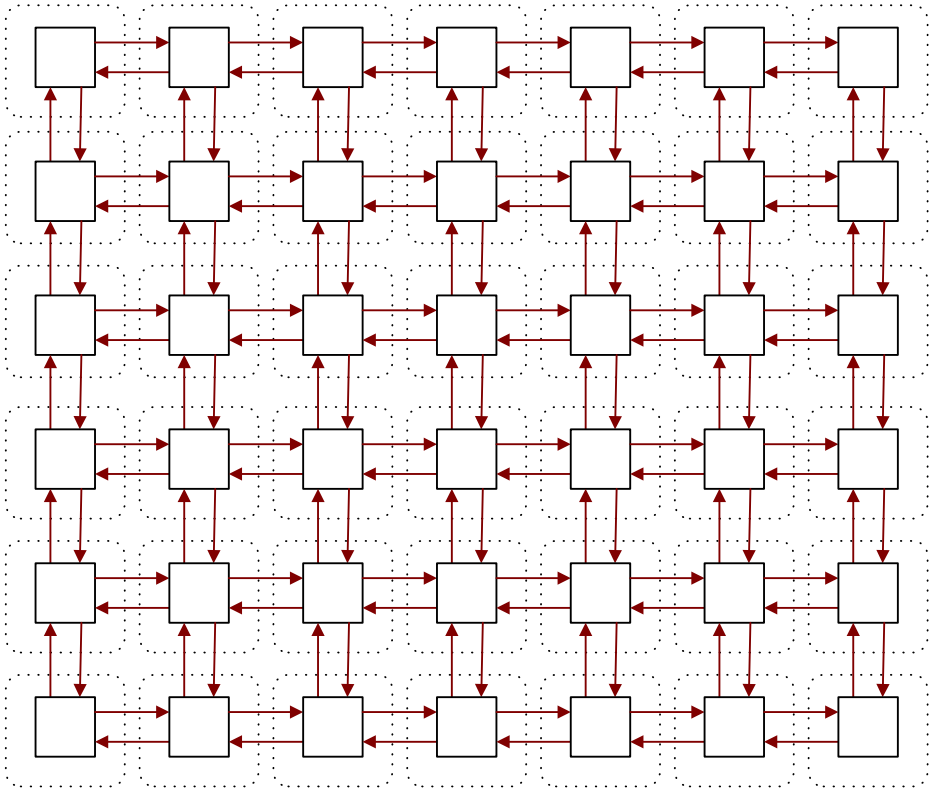
The shared data is read-only

Communication



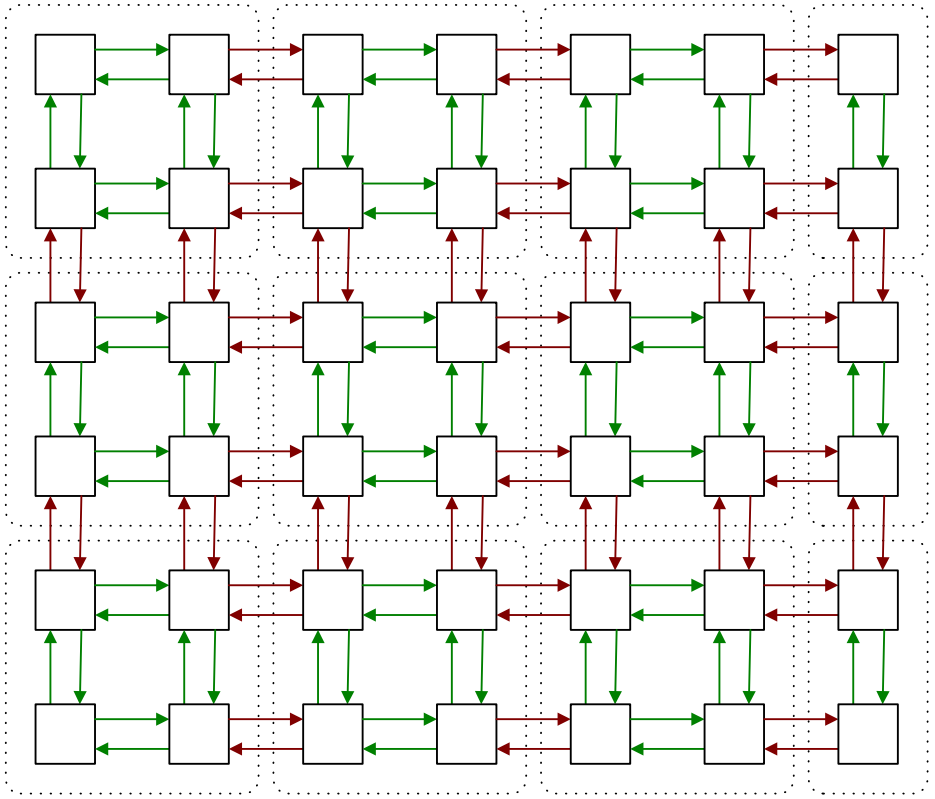
- Very nice communication: **structured**, **local**, **static**

Agglomeration



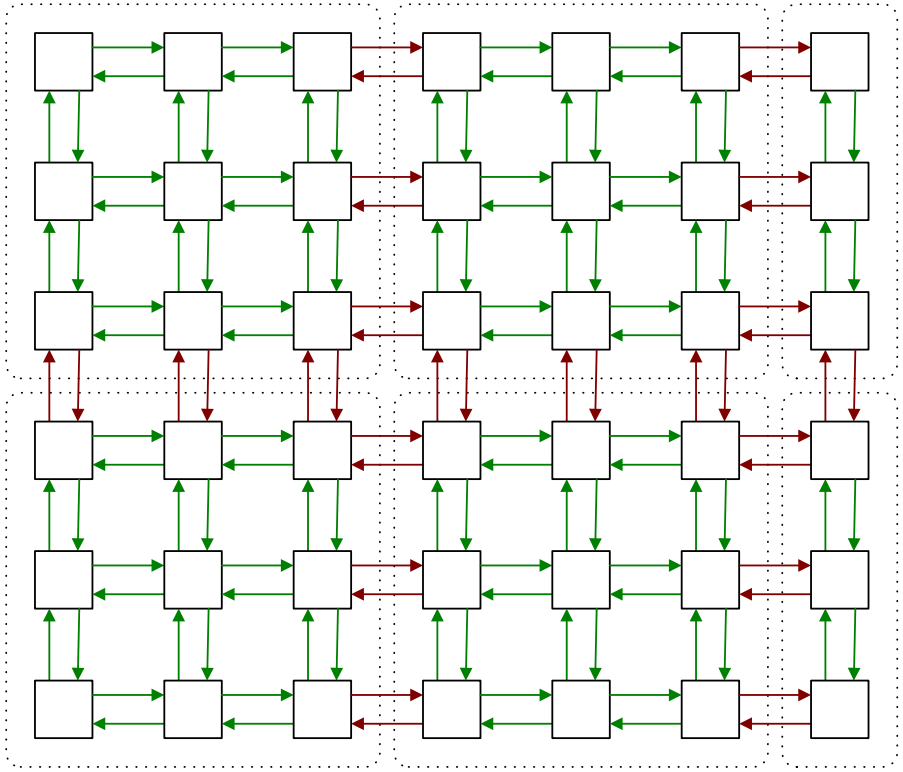
- Can run each element as a single task
- Tasks per step = (7*6) = 42, Transfers per step = (6*5)*4 = 120

Agglomeration



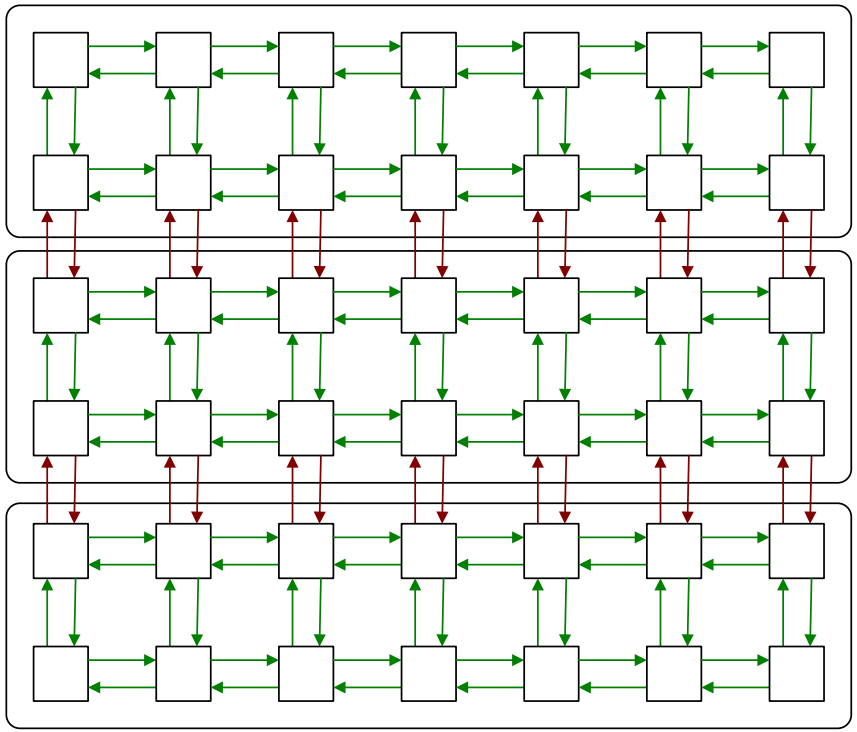
- Group into blocks of 2x2
  - Need to handle any tasks that don't quite fit
- Transfers per step = [(3*6)+(7*2)]*2 = 64, Tasks per step=12

Agglomeration



- Transfers per step = [(6*2)+7]*2 = 38, Tasks per step = 6

Agglomeration



- Transfers per step = (7*2)*2 = 28, Tasks per step=3

Agglomeration

Pattern	Tasks	TxRx	Work/Task	Work/TxRx
1x1	42			
2x2	12			
3x3	6			
7x3	3			

Mapping

- Nice regular structure, works well in many platforms
- CPU choices are fairly simple
  - `parallel_for` to schedule group of tasks within step
  - `blocked_range_2d` to split into sub-groups
  - Sequential for loop to step though time

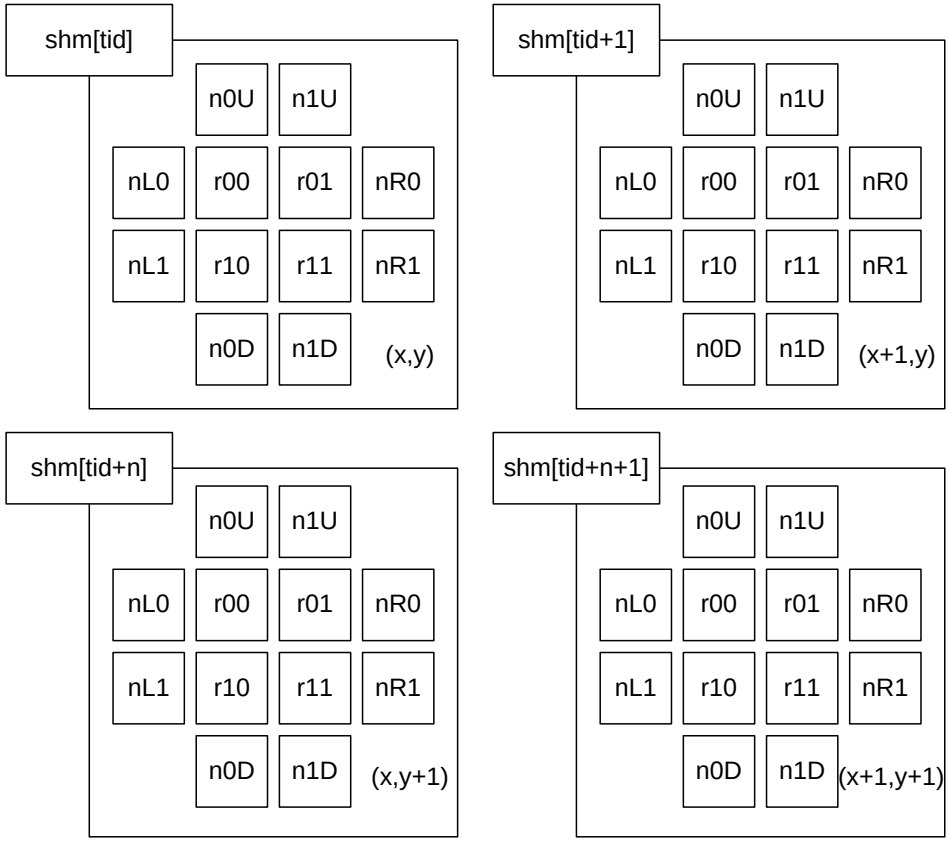
- Any more advanced options?

- What about GPUs?

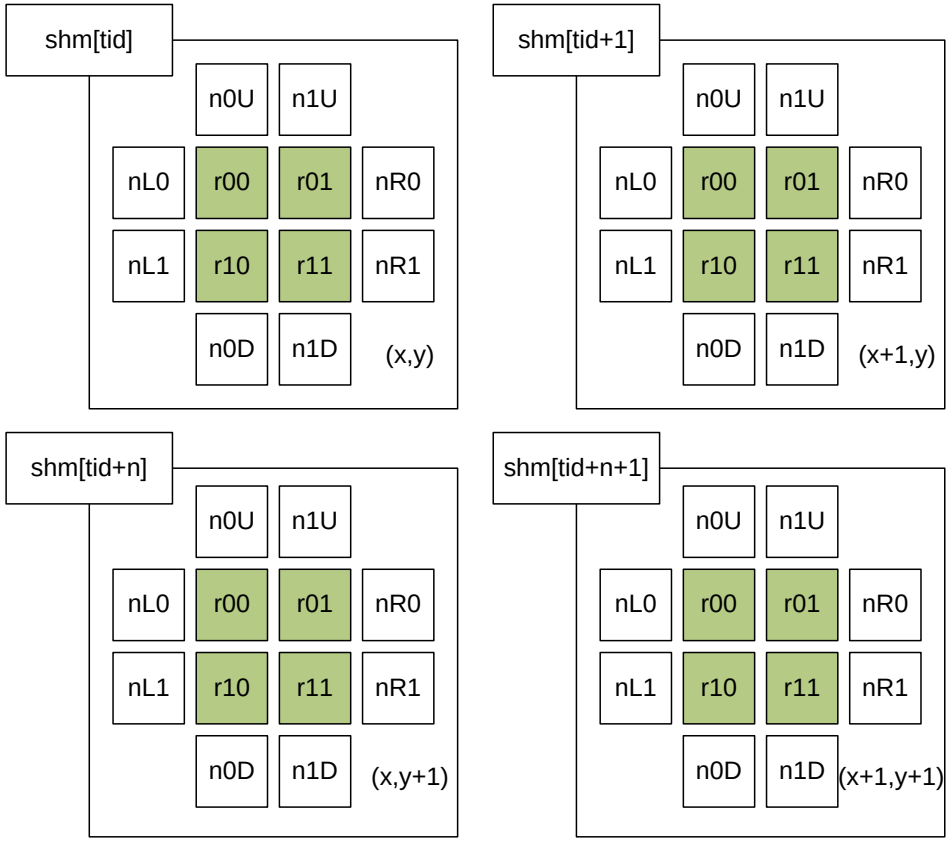
- Map each cell to a thread?
  - Very fast...
- Size of field is restricted
  - Only have block-level syncs
  - Field must be same size as shared memory
- Cost per element
  - Five memory accesses
  - Two barriers (syncs)
  - One function evaluation

```
__kernel void FiniteDifference(
    int t, float *field
){
 int tid=wId.y*wDim.x+wIdx.x;

 // State for (x,y) in a register
 float curr=field[tid];

 for(int i=0;i<t;i++){
     // Read the neighbouring field
     float left=field[tid-1];
     float right=field[tid+1];
     float up=field[tid-bDim.x];
     float down=field[tid+bDim.x];
     // Calculate the new value
     curr=(curr+left+right+up+down)/5.0f;
     barrier(...);

     // Update the value in the field
     field[tid]=curr;
     barrier(...);
}}
```

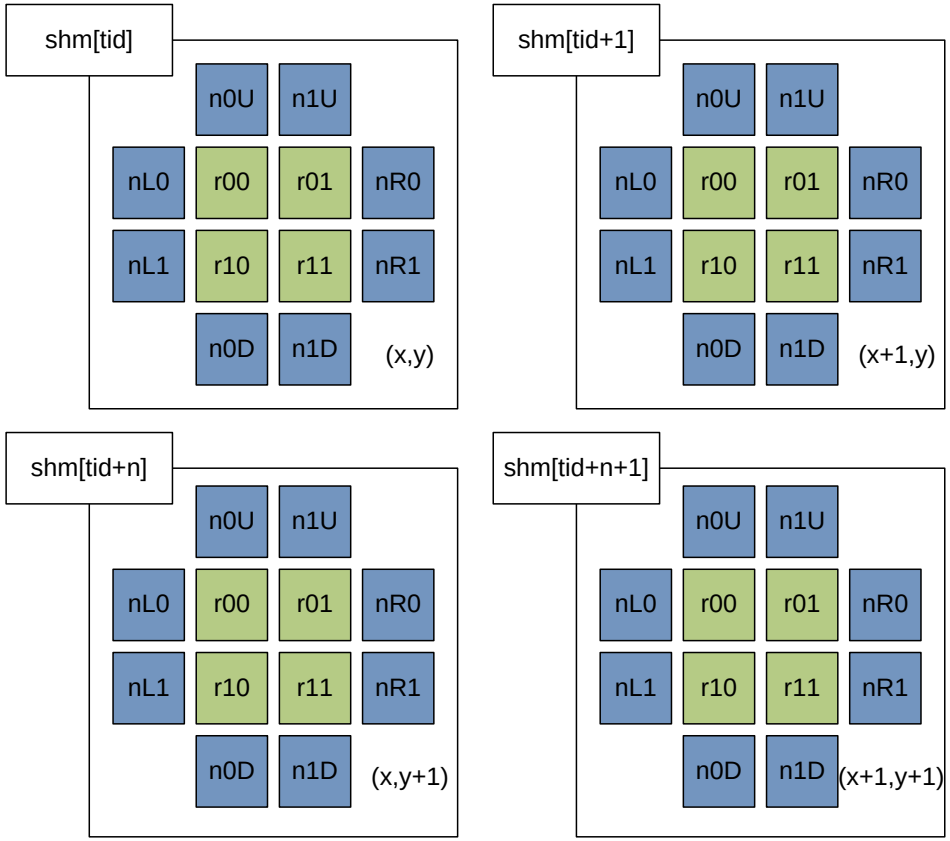
- Two elements per thread?
  - Need the neighbourhood
- Each thread has 3 parts



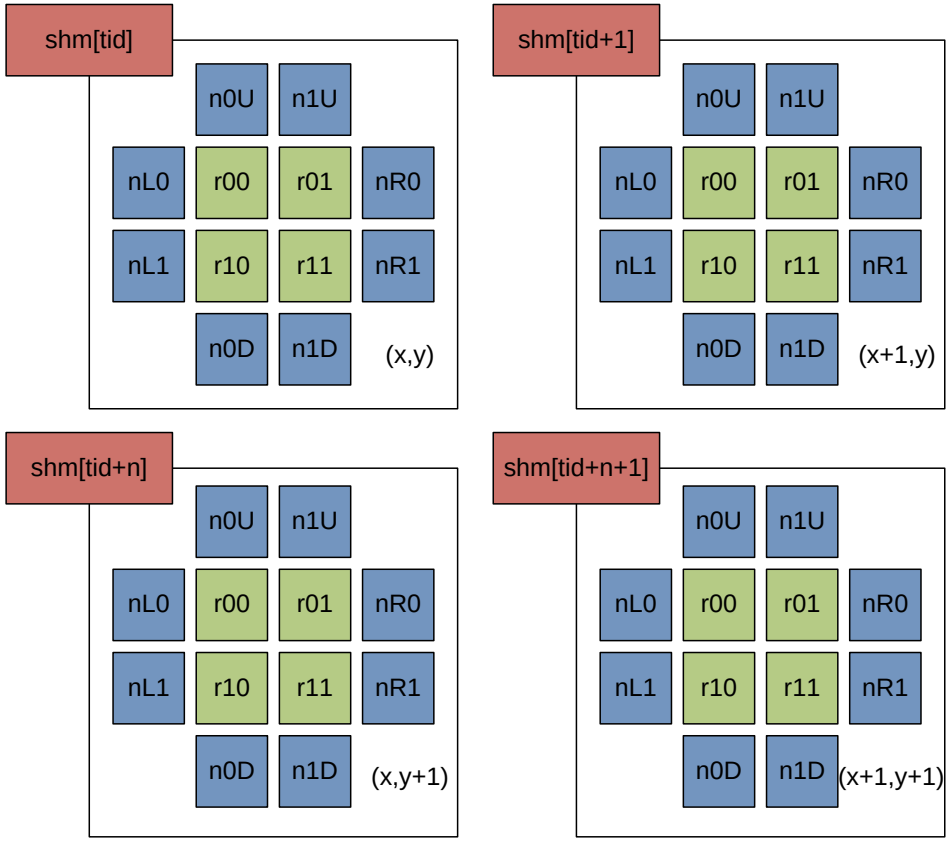
- Two elements per thread?
  - Need the neighbourhood
- Each thread has 3 parts
  - Registers for field values



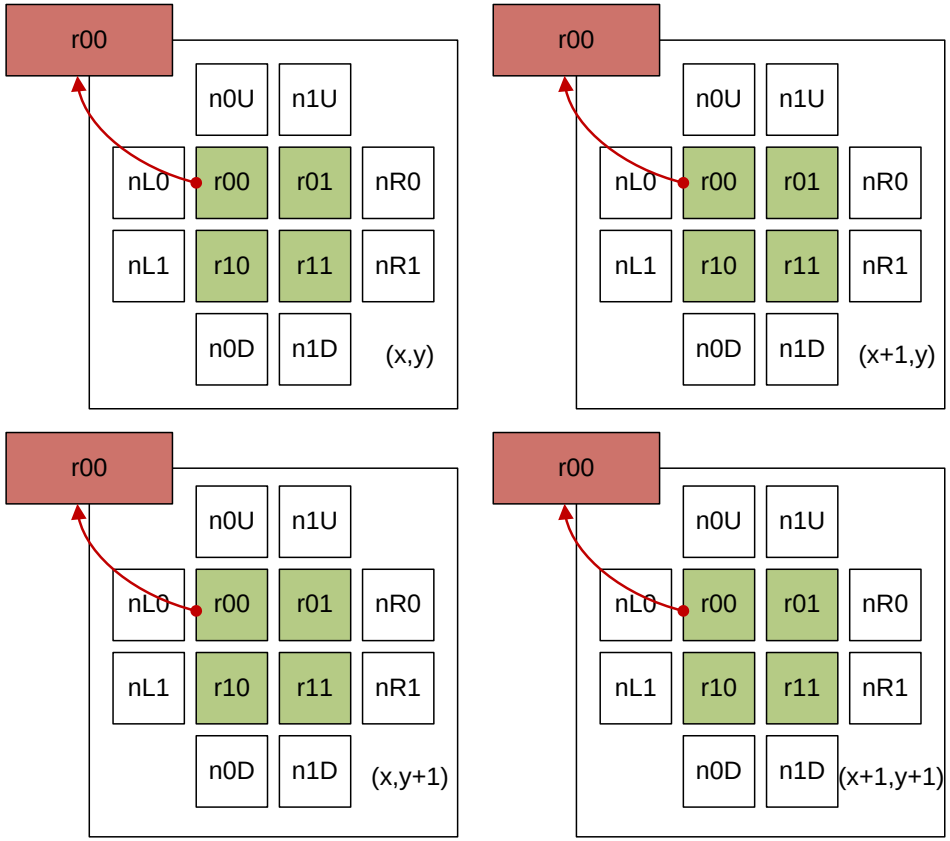
- Two elements per thread?
  - Need the neighbourhood
- Each thread has 3 parts
  - Registers for field values
  - Registers for neighbours



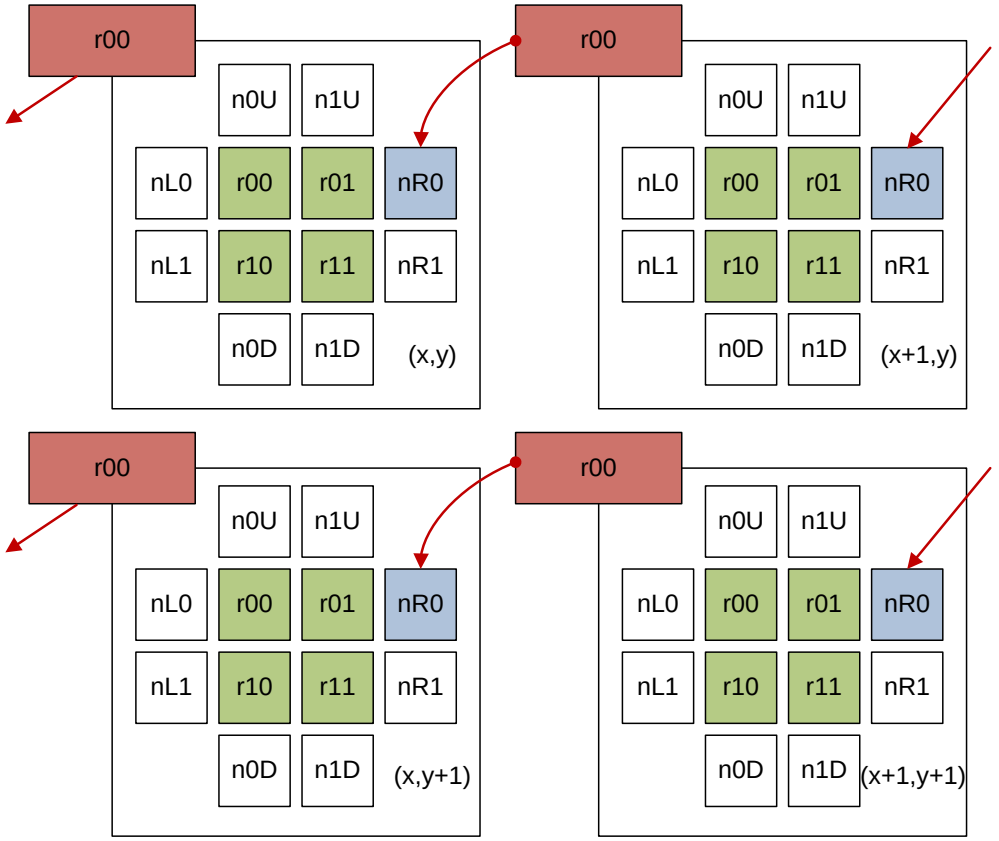
- Two elements per thread?
  - Need the neighbourhood
- Each thread has 3 things
  - Registers for field values
  - Registers for neighbours
  - One shared mem location
- Field size = 4*shared mem
  - But... communication?



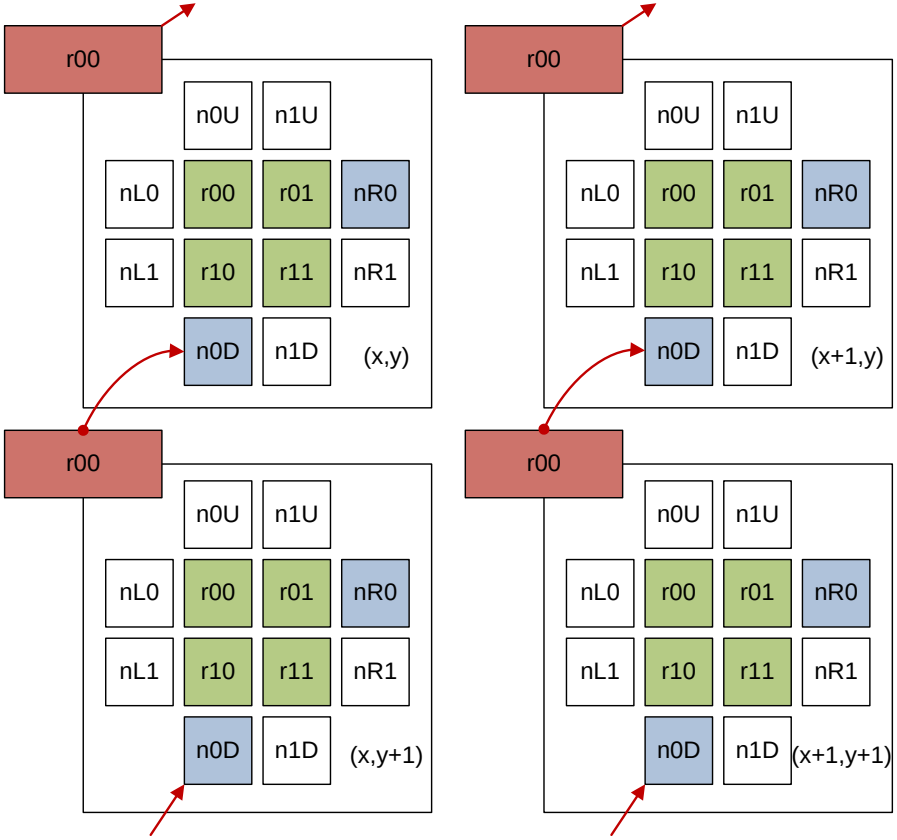
- Two elements per thread?
  - Need the neighbourhood
- Each thread has 3 things
  - Registers for field values
  - Registers for neighbours
  - One shared mem location
- Field size = 4*shared mem
  - But... communication?
- Four-stage broadcast
  - Send r00 to shared mem



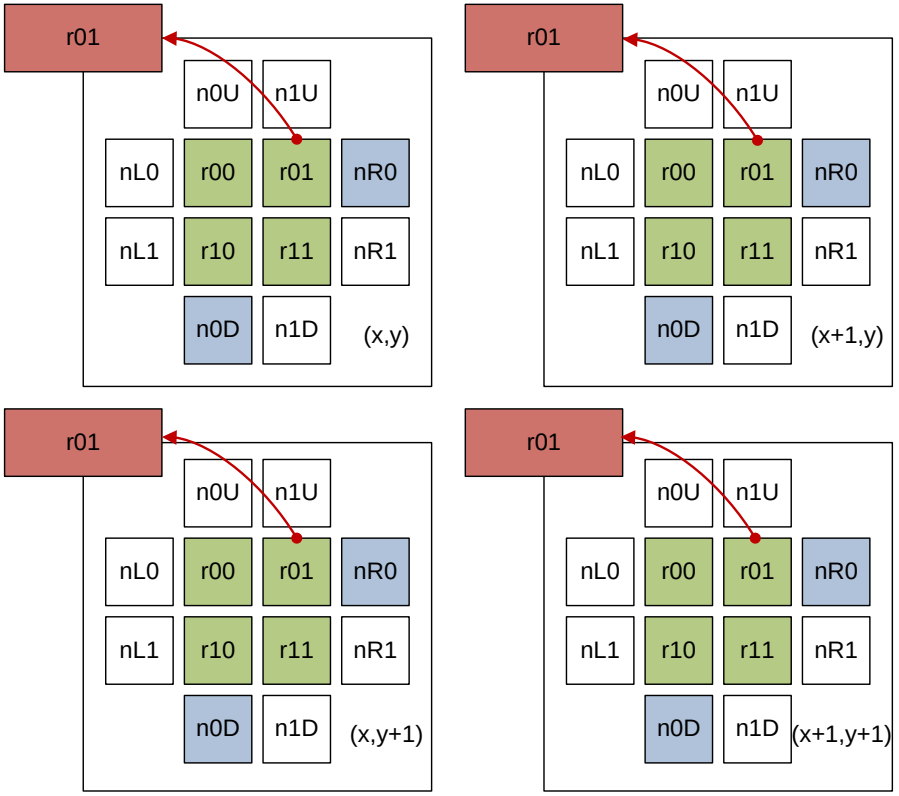
- Two elements per thread?
  - Need the neighbourhood
- Each thread has 3 things
  - Registers for field values
  - Registers for neighbours
  - One shared mem location
- Field size = 4*shared mem
  - But... communication?
- Four-stage broadcast
  - Send r00 to shared mem
  - Neighbours capture it



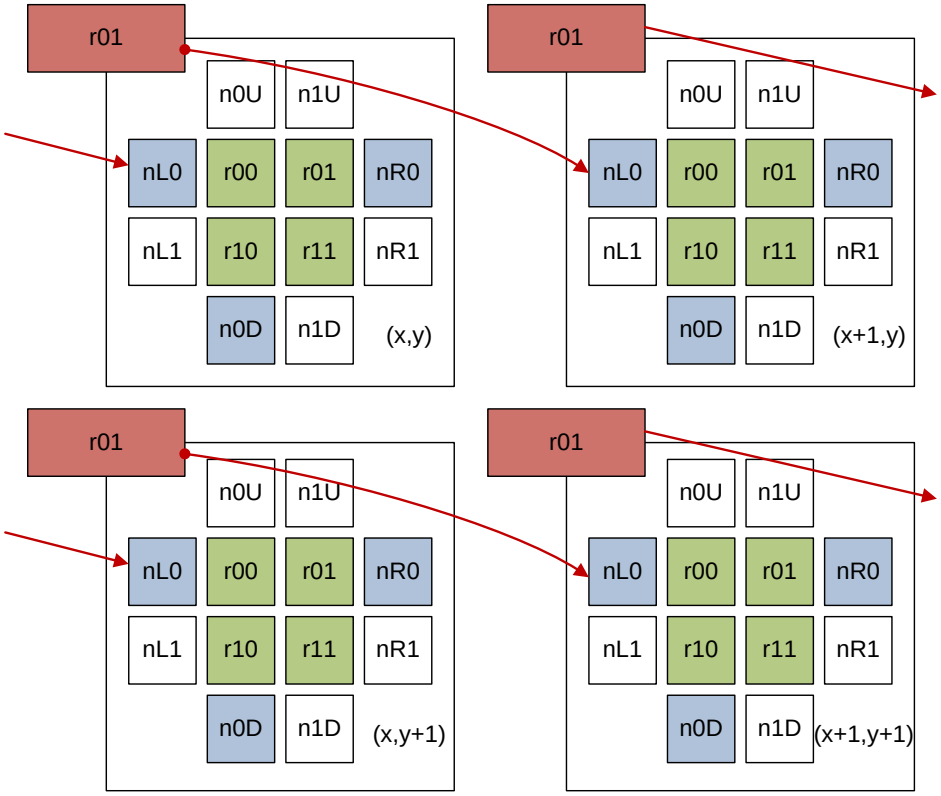
- Two elements per thread?
  - Need the neighbourhood
- Each thread has 3 things
  - Registers for field values
  - Registers for neighbours
  - One shared mem location
- Field size = 4*shared mem
  - But... communication?
- Four-stage broadcast
  - Send r00 to shared mem
  - Neighbours capture it



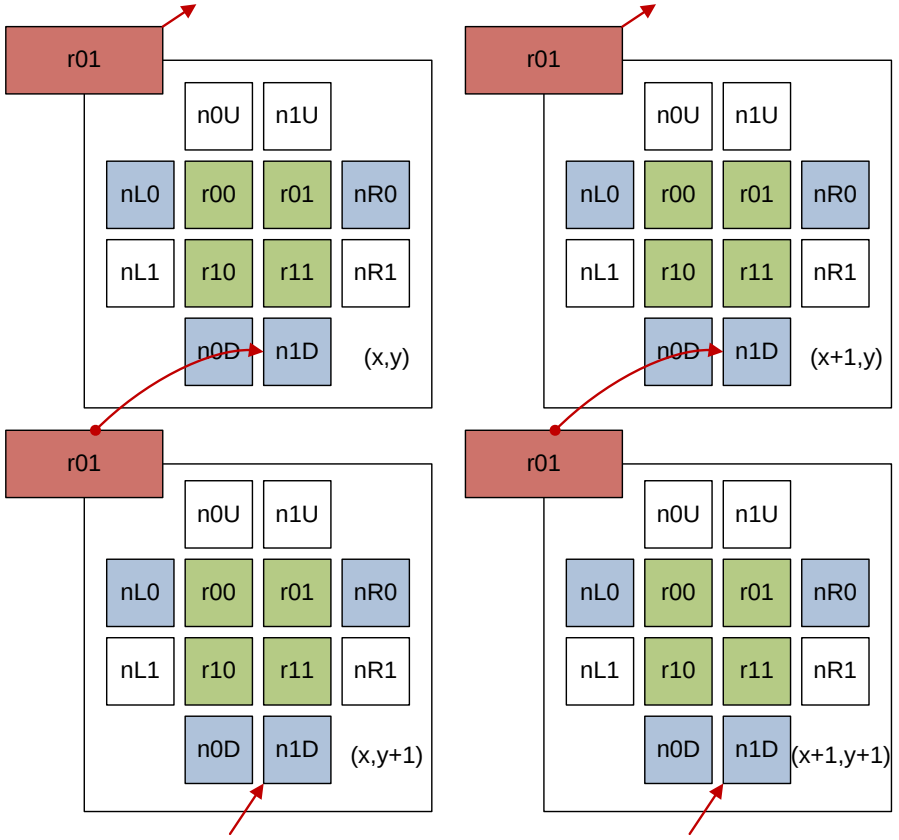
- Two elements per thread?
  - Need the neighbourhood
- Each thread has 3 things
  - Registers for field values
  - Registers for neighbours
  - One shared mem location
- Field size = 4*shared mem
  - But... communication?
- Four-stage broadcast
  - Send r00 to shared mem
  - Neighbours capture it
  - Send r01 to shared mem



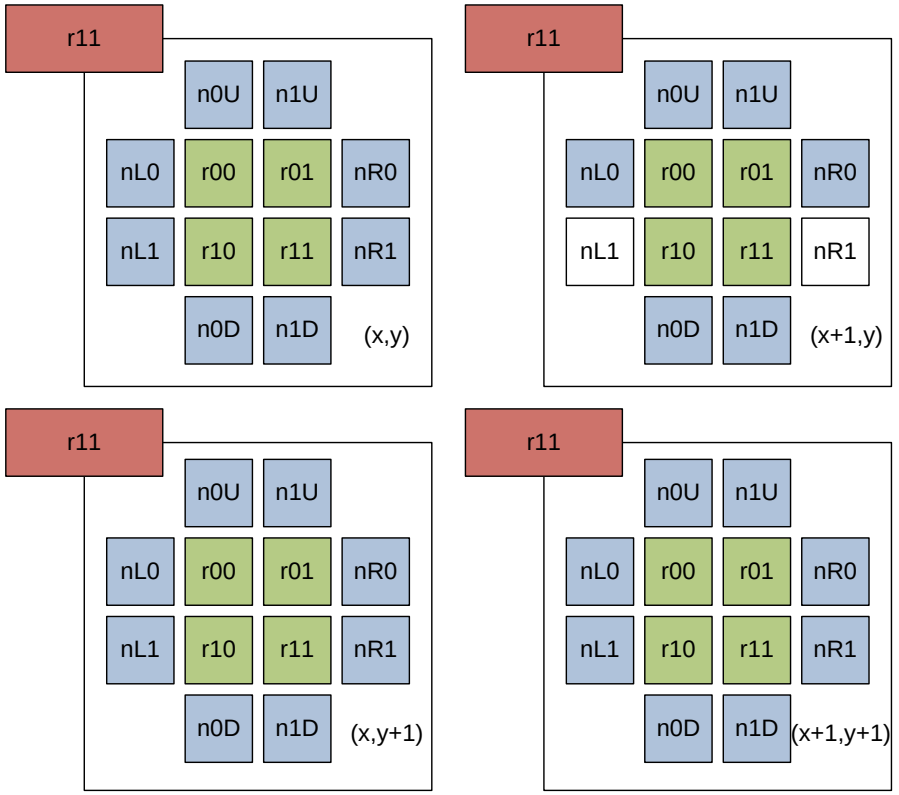
- Two elements per thread?
  - Need the neighbourhood
- Each thread has 3 things
  - Registers for field values
  - Registers for neighbours
  - One shared mem location
- Field size = 4*shared mem
  - But... communication?
- Four-stage broadcast
  - Send r00 to shared mem
  - Neighbours capture it
  - Send r01 to shared mem
  - Neighbours capture it



- Two elements per thread?
  - Need the neighbourhood
- Each thread has 3 things
  - Registers for field values
  - Registers for neighbours
  - One shared mem location
- Field size = 4*shared mem
  - But... communication?
- Four-stage broadcast
  - Send r00 to shared mem
  - Neighbours capture it
  - Send r01 to shared mem
  - Neighbours capture it



- Two elements per thread?
  - Need the neighbourhood
- Each thread has 3 things
  - Registers for field values
  - Registers for neighbours
  - One shared mem location
- Field size = 4*shared mem
  - But... communication?
- Four-stage broadcast
  - Send r00 to shared mem
  - Neighbours capture it
  - Send r01 to shared mem
  - Neighbours capture it
- Eventually all is done
  - Update the field!



Does it make sense?

- Simple, but laborious to write
  - Quite difficult to get right…
- What are the requirements?
  - 8 sync, 12 read/write
  - Per element: 2 sync, 3 read/write
- Do the advantages outweigh the problems?
  - Pro: four times the field size; less comms.
  - Con: three times the register usage

- What happens with a 3x3 grid?
- What about using the same approach with global memory?

```
__kernel void FiniteDifference(int t, float *f,
  __local float *scratch)
{
   int tid=(tIdx.y*2)*(bDim.x*2)+(tIdx.x*2);
   int sid=tid/2;

   float r00=f[tid], r01=f[tid+1];
   float r10=f[tid+2*bDim.x], r11=f[tid+2*bDim.x+1];

   for(int i=0;i<t;i++){
      // Store data for current top-left
      scratch[tid]=r00;
      barrier(...);
      nR0 = scratch[tid+1];
      n0D = scratch[tid+bDim.x];
      barrier(...);

      // Store data for current top-right
      scratch[tid]=r01;
      barrier(...);
      nL0 = scratch[tid-1];
      n1D = scratch[tid+bDim.x];
      barrier(...);

      // Store data for current bottom-left
      scratch[tid]=r10;
      barrier(...);
      nR1 = scratch[tid+1];
      n0U = scratch[tid-bDim.x];
      barrier(...);

      // Finally, do the bottom-right
      scratch[tid]=r11;
      barrier(...);
      nL1 = scratch[tid-1];
      n1U = scratch[tid-bDim.x];
      barrier(...);

      // Finally the neighbourhood is done! Calculate
      // into temporary variables to preserve originals
      float t00=(r00+nL0+n0U+r01+r10)/5.0f;
      float t01=(r01+r00+n1U+nR0+r11)/5.0f;
      float t10=(r10+nL1+r00+r11+n0D)/5.0f;
      float t11=(r11+r10+r01+nR1+n1D)/5.0f;

      // And update the actual variables...
      r00=t00;
      r01=t01;
      r10=t10;
      r11=t11;
   }}
```

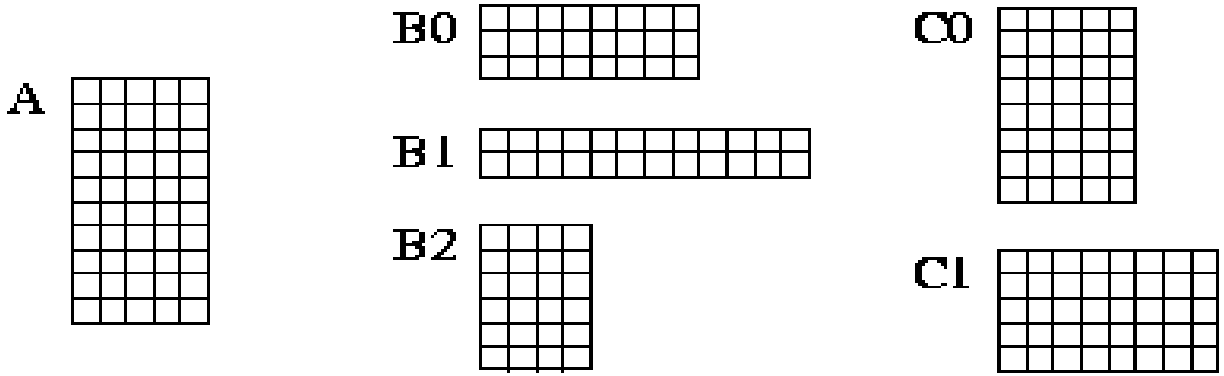
ASIC Floorplan Optimisation

- Circuits consist of high-level functionality and interconnect
  - "*This adder adds the output of this multiplier and that register*"
  - Leaves the low-level design of each component flexible
  - Doesn't specify the physical layout of all the components

- Target is often to minimise the total silicon area required
  - Each component has a number of possible 2D implementations
  - Connectivity specifies which components must be adjacent
  - Find the design with the smallest 2D bounding box

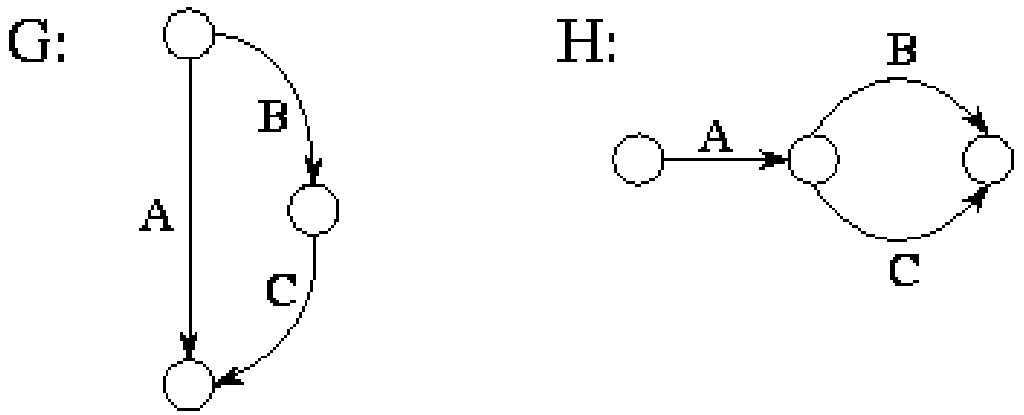
Example and figures from: http://www.mcs.anl.gov/~itf/dbpp/text/node21.html

Problem Specification: Components



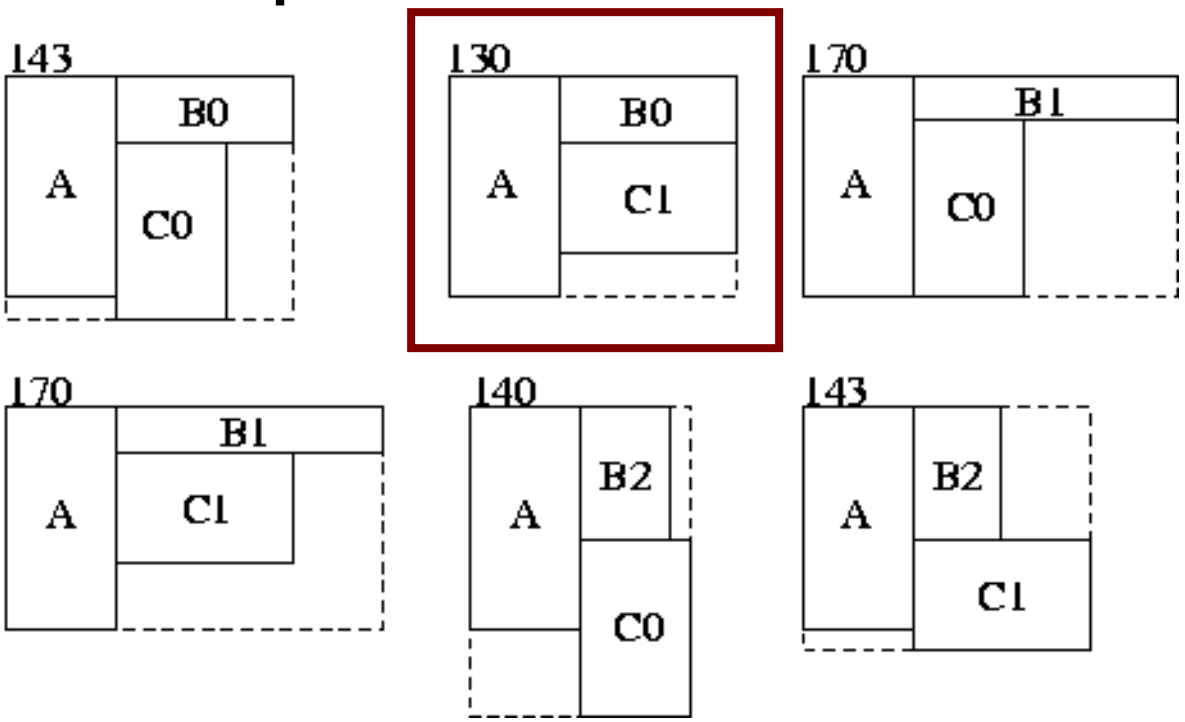
- Three components A, B, and C
- Each component has a number of implementations
  - Each implementation has a different layout and area in silicon

Problem Spec.: Adjacency Graphs



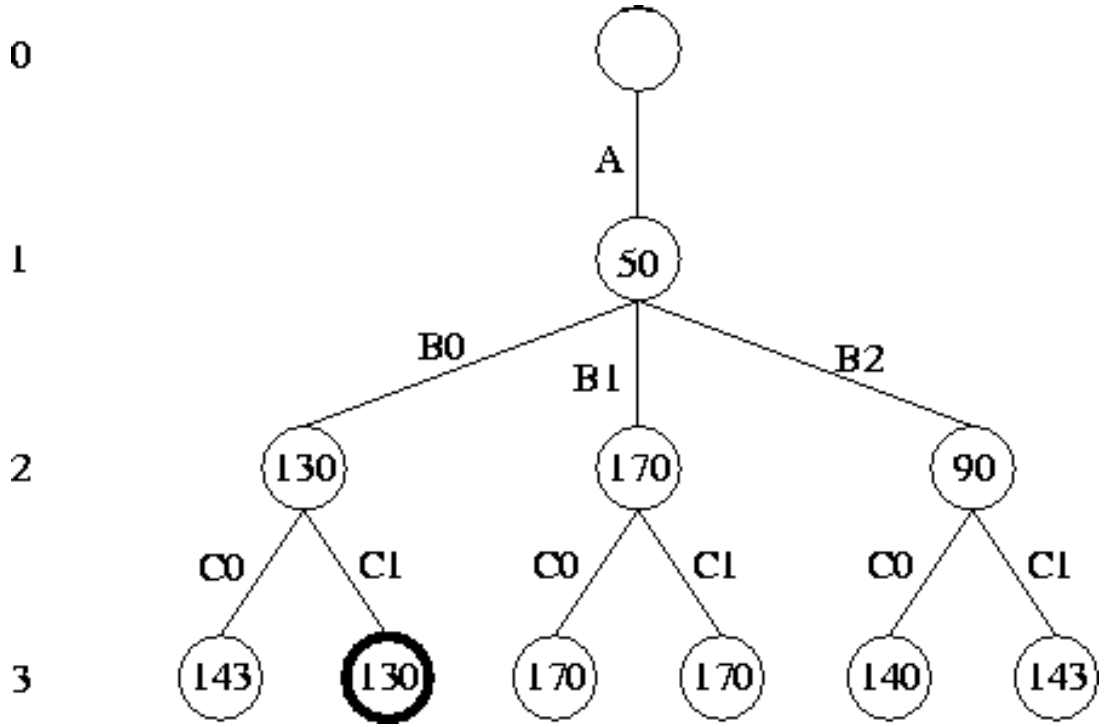
- Components need to communicate with each other
- Graph G: constraints in vertical direction
  - "*Lower edge of B must touch upper edge of C*"
- Graph H: constraints in horizontal direction
  - "*Right edge of A must touch left edge of both B and C*"

Problem Spec.: Possible Solutions



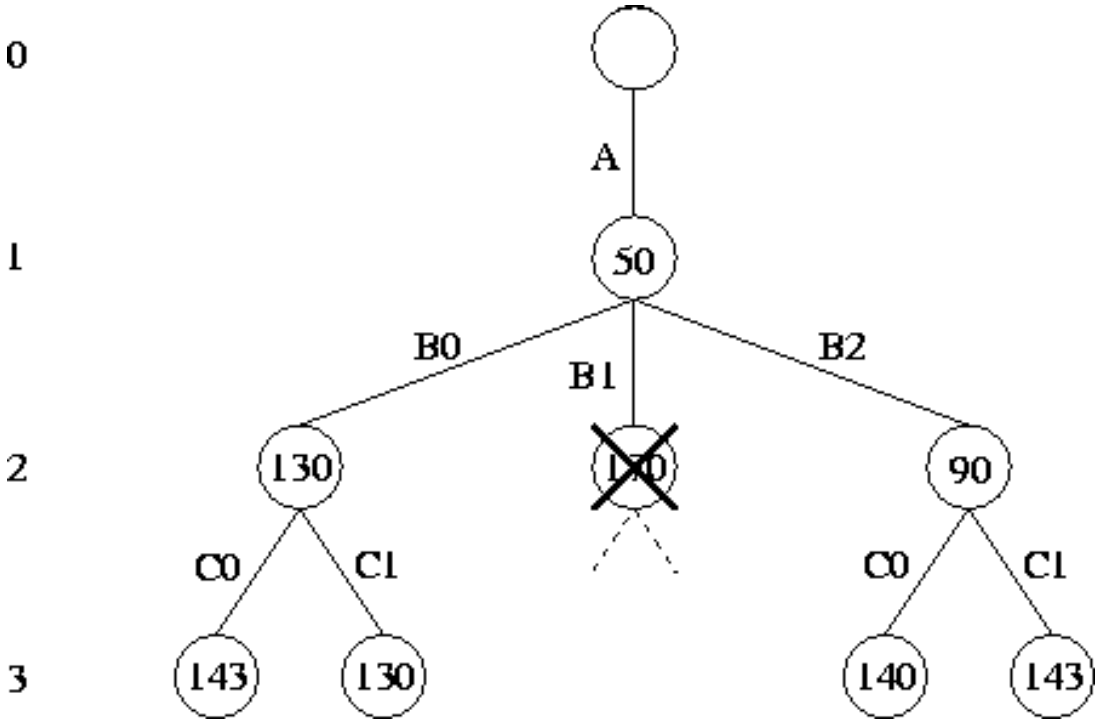
- Need to consider all solutions which meet constraints
  - Must use methodical method to enumerate all possibilities
  - Potential for ***combinatorial*** explosion of potential solutions

Problem Spec.: Search Tree



- Start from top and left of G and H graph
- Recursively expand graph, trying all component implementations
- Find leaf node with the lowest cost

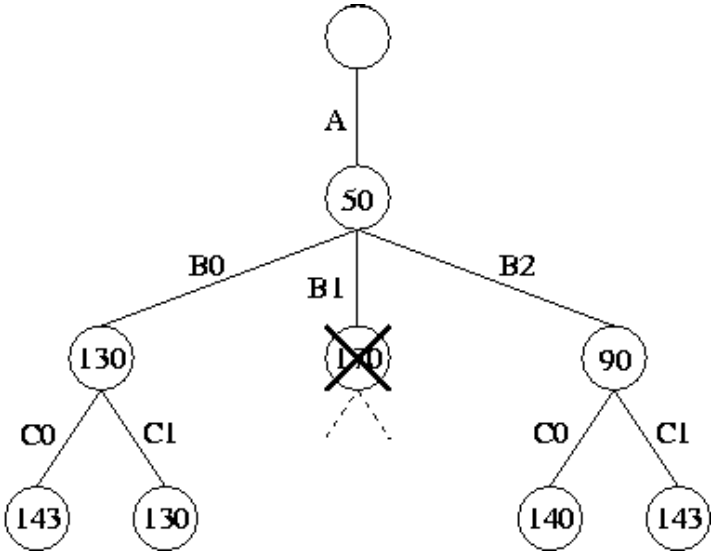
Problem Spec.: Branch-and-Bound



- Try to eliminate useless branches without expanding them
- If intermediate node has higher cost than leaf node, prune it
- We must know the cost of at least one leaf to enable pruning

Partitioning

- There are no distinct stages to the problem
  - Pretty much just recursive expansion of nodes
  - Apply the same operation until all leaves have been considered
- There is significant parallelism available
  - Task based decomposition at each branch in the tree
- Need to be careful though
  - Scheduler should operate depth first
  - Need to get to leaves for pruning cost
- Pruning requires shared minimum
  - Should be “Non-blocking”
  - Most not limit parallelism

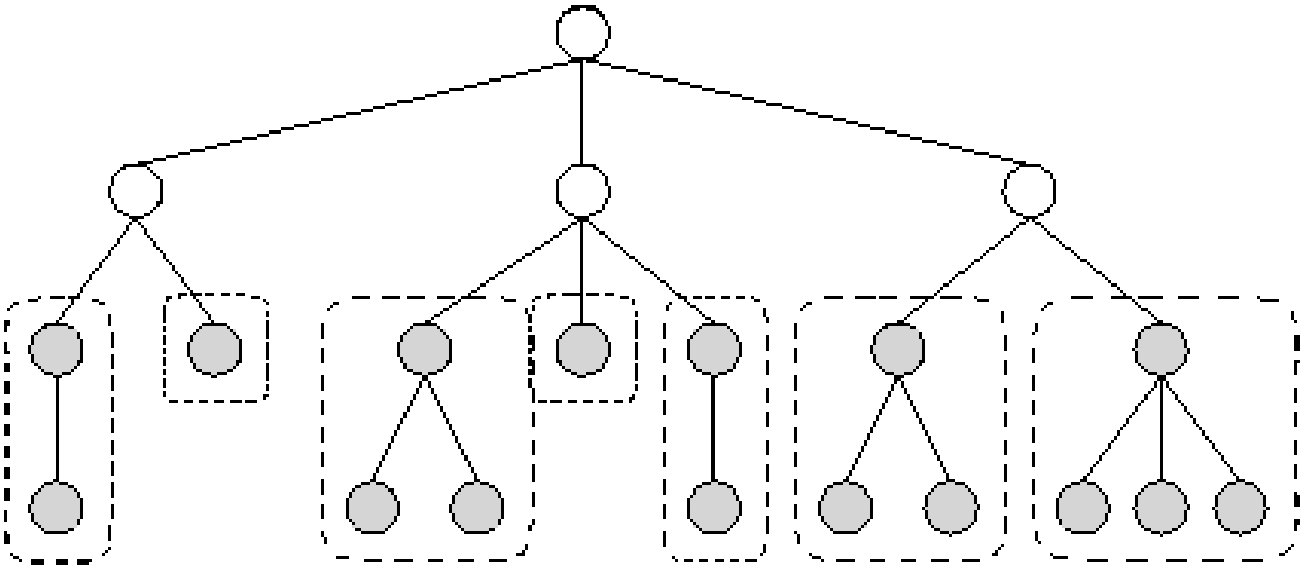


Communication

- Need to maintain the best minimum seen so far
  - *During execution*: used to control pruning
  - *End of execution*: reflects the cost of the minimum answer
- What if we ***don't*** correctly maintain shared minimum?
  - *During execution*: do a bit more work than necessary (oh well...)
  - *End of execution*: return the ***wrong*** answer

- Can take an *eventual consistency* approach
  - Maintain local shared minimum within task: *very cheap*
  - Periodically update global minimum: *reduce overhead*
  - *Eventually* global minimum reflects minimum of all locals
- Tradeoff extra work versus reduced communication overhead

Agglomeration



- Standard agglomeration problem: too many tasks, slow down
  - Switch to serial below a certain depth in the tree
- But also a pruning specific agglomeration problem
  - Need to make sure that we explore down as well as out
  - Must ensure scheduler will reach leaf nodes fairly often
  - Something to bear in mind during Mapping stage

Mapping

- Would this work well in a GPU?
  - Computation: does a SIMD architecture look appropriate?
  - Parallelism: does a grid/block system look appropriate?

- Would this work well in a CPU?
  - What systems have the right kind of scheduling... TBB?
  - How would you implement the shared communication?
  - Is there a good argument for parallel_reduce vs task?

Finishing up

Where we are

- The department (tries) to make you good engineers
  - Computers are a fact of life in any engineering field
  - In your careers all computers will be parallel

- **If** asked to speed something up, you'll know where to start
  - But: correctness is more important than speed
  - Though correctness is often a sliding scale

- If someone asks you to do the impossible, you can say no
  - Many acceleration projects are hopeless
  - At the very least you can leave the project/company/startup

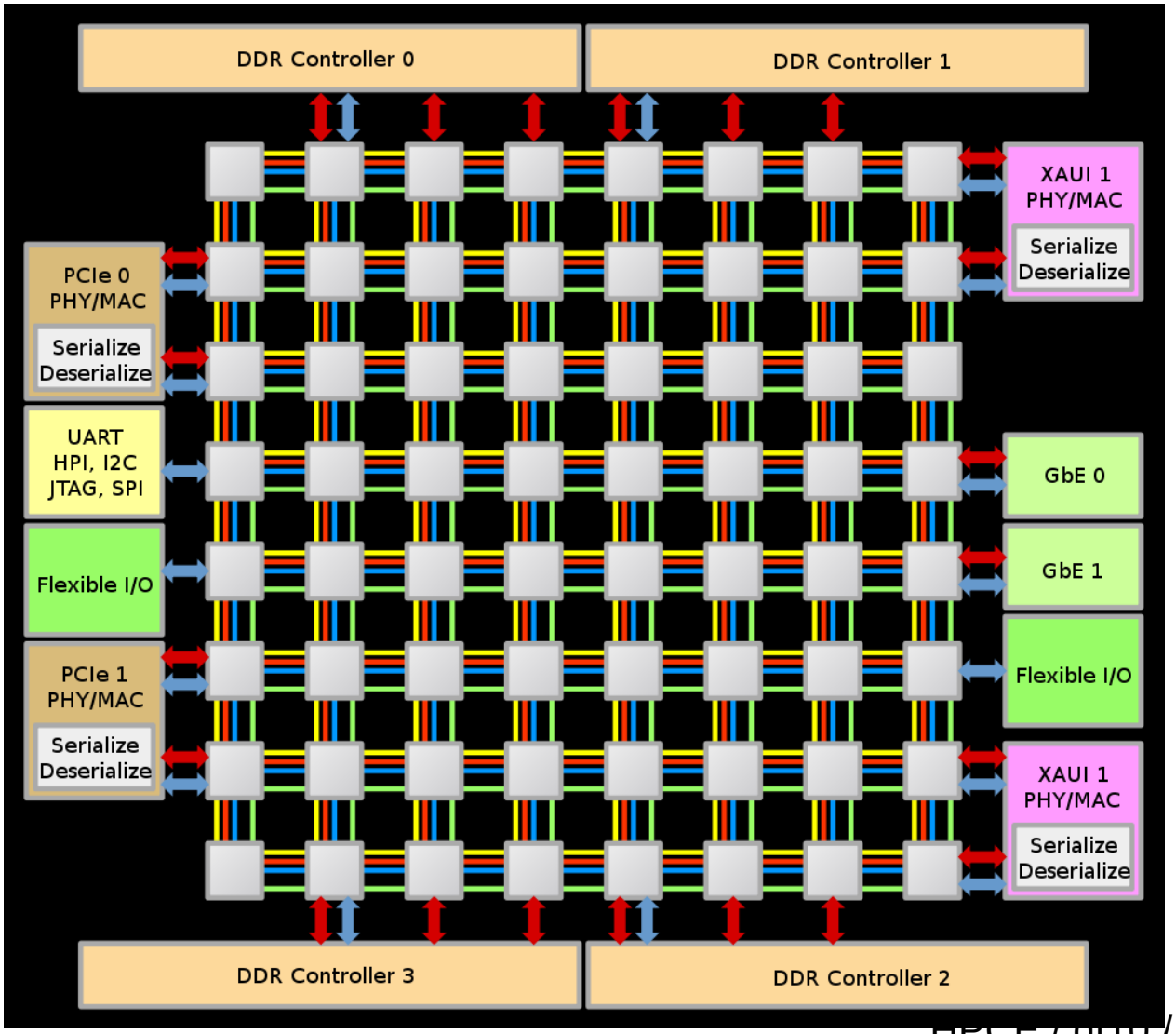
Practical Skills

- These systems are all around you
  - The department doesn't have very exciting systems
  - Have multi-core and GPUs by accident

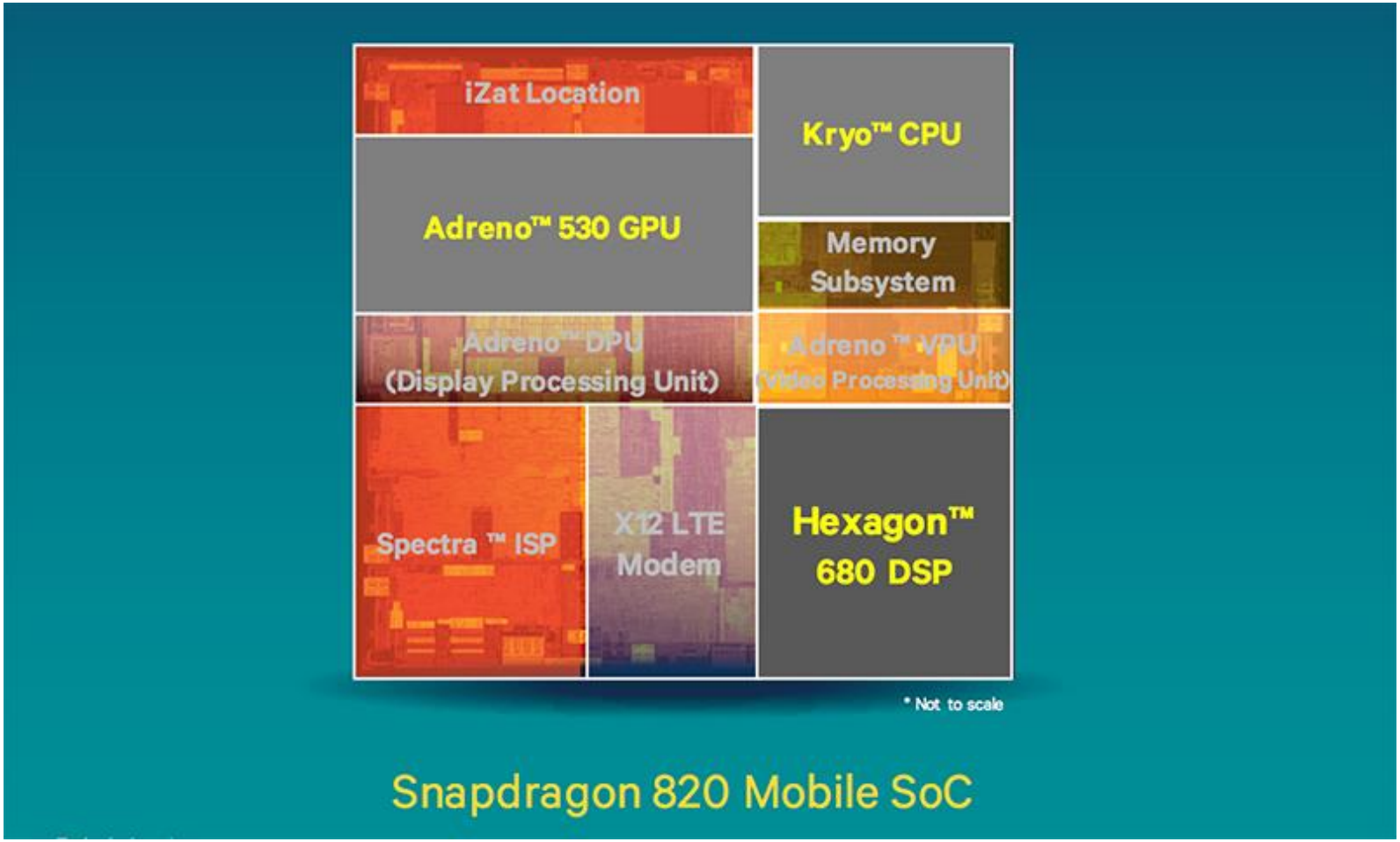
- Can you make a program run on 16 or 32 cores?
  - A 4x speed-up on 16 cores is still very useful
  - Many things can be done with parallel_for and parallel_reduce

- Can you port a basic numerical program to a GPU?
  - Lots of problems fit easily in the iteration space model
  - Many problems can be converted to pipelines

TILE64 : 64 x 64-bit processors



Mobile SoCs


iZat Location
Kryo™ CPU
Adreno™ 530 GPU
Memory Subsystem
Adreno™ DPU
(Display Processing Unit)
Adreno™ VPU
(Video Processing Unit)
Spectra™ ISP
X12 LTE Modem
Hexagon™ 680 DSP
* Not to scale
Snapdragon 820 Mobile SoC

Theoretical Knowledge

- Practical skills are good, but understanding is better
  - Why do you have to program in this way rather than that way?
  - Why do languages not let you do certain things?

- Theoretical knowledge is important for acceleration
  - It's easy to create parallelism and occupy all the cores
  - It's more difficult to create parallelism **and** get a speed-up
  - Need to understand things like agglomeration and atomics

- Architectural knowledge is becoming more important…
  - Very difficult to program GPUs well without architecture
  - Continuing trend for multi-core SoCs and computers

CW5

- We’re nearly there…

- Lot’s of problems with AWS instances crashing
  - Something is causing machines to crash and eventually reboot
  - Any ideas?
- If you make the machine crash, it won’t mean you get zero
- Feedback schedule (this is the “exam” part):
  - During coursework: managed three feedback runs
  - Performance/correctness: over the next couple of weeks
  - Detailed feedback: sometime after Christmas
  - Grades: targeting start of Feb

CW6

- Tradition suggests a sanity gap between CW5 and CW6
  - Will issue at 9:00 on Mon (Nov 27th)
  - Due at 22:00 on Fri Dec 15th (19 days total)
- You can keep your group or mix it up
  - New spreadsheet here
  - XXXX (link hidden for slides)
  - Can change up until 17:00 on Nov 27th when repos are created
    - Remember, you can always work in a local repo till then
- Target is a real and rather large application
  - Don't expect massive speedups – be realistic
  - There are no deliberate optimisation opportunities
  - Repos will be created at 9:00 on Monday

Strengths: Github feedback going well

- *From your end*: A lot of good interaction on issues this year
  - Thanks to all those who posted questions
  - Thanks to all those who answered questions
  - Thanks to those finding bugs and solutions
- *From my end*: went quite well in terms of response rate
  - CW1-CW4 : ~110 responses total
  - CW5: 70 responses so far
  - ~7.5 mins per response; ~22.5 hours, or 11.25 mins/student
  - 85% <1 day, 94% <2 days
  - Much better mechanism than email

Strengths: Real-time feedback

- Self assessment for CW1 works well
  - Everyone knows what mark they are getting
  - Achieved marks ~100% for everyone

- Intermittent assessment for CW2-CW3 quite effective
  - CW2: 80% knew they would get 90%+ one day before deadline
  - CW3: 75% knew they would get 90%+ one day before deadline
  - CW4: 67% knew they would get 90%+ one day before deadline
  - Actually better than giving out test script?
    - Encourages people to start earlier
  - Led to good questions about why/how assessments worked

Weaknesses: marks (!= feedback)

- Turning feedback scores into “official” marks taking too long
  - Some quadratic processes that didn’t scale
  - Lots of people started the course then dropped it
  - Too many submission routes
- Possible solutions: *remove safety nets*
  - No intervention for submissions that are broken
  - One single source of submission (i.e. github)
  - Final feedback score is exactly the grade
- Possible solutions: *zero marks allocated to CW1-CW4*
  - If students don’t do it… ¯_(ツ)_/¯ (CW5+CW6 would be fun)
- ~~Possible solutions: *get GTAs to do it*~~
  - GTAs shouldn’t be used for admin, only for teaching

Weaknesses(?) : generating graphs

- I often ask students to generate certain graphs
  - There are good reasons for asking for them
  - Can’t really say whether one is good or bad

- Are they just taking up time, or is the value clear?

Weaknesses: becoming a MOOC

- Class attendance drops over time
  - Lots of invisible students taking the course
  - About 60% of the class attending via panopto
  - *(Not unique to this module)*
- Possible solutions: *give up and be a MOOC*
  - Business school have done it
  - Potentially allows more people to take it…
- Possible solutions: *reduce class size, use workshops*
  - Much more interesting for me
  - Need to cut the number of students back to 60, but how?

Weaknesses/strength: scope creep

- The course is slowing changing from the original intent
  - Used to be 50% theory (exam) and 50% skills (practise)
  - Theoretical side is dwindling to a small part
  - More and more about software
- What is this course actually for:
  - Practical GPU skills?
  - General optimisation?
  - Preparing for research?
  - Preparing for projects?
  - Improving software skills?
  - Gaining software skills?
  - Automation and infrastructure?
- What do you think it is for?

Conclusion

- You should now know *a bit* about a number of technologies
  - Multi-core, GPU, pipelines, scripting, cloud services, github
  - Everyone should at last be… competent
- Hopefully you also know something about performance
  - Applied asymptotic analysis
  - Intuition for the cost/benefit tradeoffs when parallelising
  - A rough idea about what to look for when optimising
- There are now diminishing returns…
  - People are busy
  - Not much point to piling on more knowledge

Thank-you all for attending: good luck with CW6