High-Performance Physics Solver Design for Next Generation Consoles

Vangelis Kokkevis
Steven Osman
Eric Larsen

Simulation Technology Group

**Sony Computer Entertainment America
US R&D**

This Talk

- Optimizing physics simulation on a multi-core architecture.
  - Focus on CELL architecture
- Variety of simulation domains
  - Cloth, Rigid Bodies, Fluids, Particles
- Practical advice based on real case-studies
- Demos!

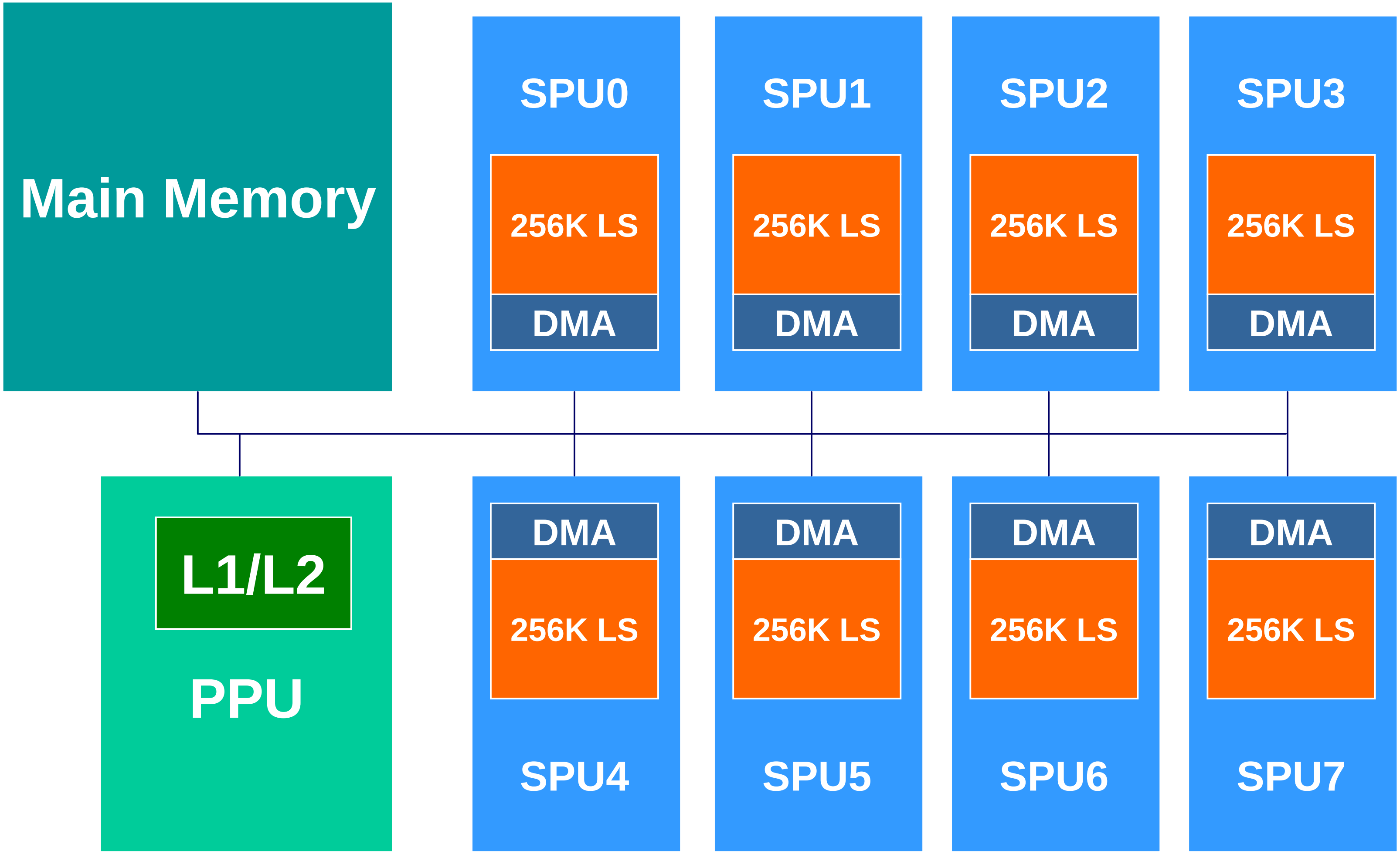
Basic Issues

- Looking for opportunities to parallelize processing
  - High Level – Many independent solvers on multiple cores
  - Low Level – One solver, one/multiple cores
- Coding with small memory in mind
  - Streaming
  - Batching up work
  - Software Caching
- Speeding up processing within each unit
  - SIMD processing, instruction scheduling
  - Double-buffering
- Parallelizing/optimizing existing code

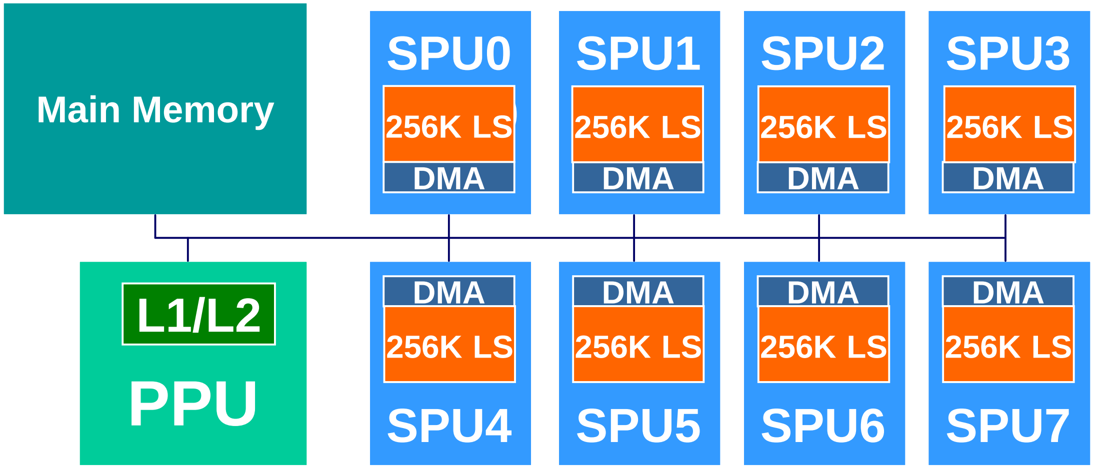
What is not in this talk?

- Details on specific physics algorithms
  - Too much material for a 1-hour talk
  - Will provide references to techniques
- Much insight on non-CELL platforms
  - Concentrate on actual results
  - Concepts *should* be applicable beyond CELL

The Cell Processor Model



Physics on CELL



- Physics should happen mostly on SPUs
  - There's more of them!
  - SPUs have greater bandwidth & performance
  - PPU is busy doing other stuff

SPU Performance Recipe

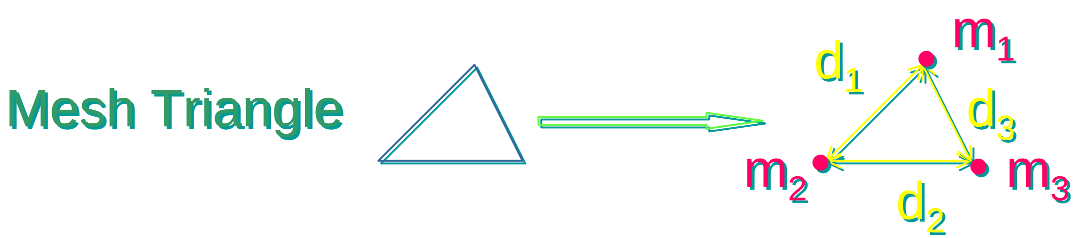
- Large bandwidth to and from main memory
- Quick (1-cycle) LS memory access
- SIMD instruction set
- Concurrent DMA and processing
- Challenges:
  - Limited LS size, shared between code and data
  - Random accesses of main memory are slow

Cloth Simulation

GameDevelopers Conference

Cloth Simulation

- Cloth mesh simulated as point masses (vertices) connected via distance constraints (edges).



- References:
  - T.Jacobsen,*Advanced Character Physics*, GDC 2001
  - A.Meggs,*Taking Real-Time Cloth Beyond Curtains*,GDC 2005

Simulation Step

1. Compute external forces, f^E,per vertex
2. Compute new vertex positions [Integration]:

$$p^{t+1} = (2p^t - p^{t-1}) + \frac{1}{2} f^E * \frac{1}{m} * \Delta t^2$$

3. Fix edge lengths
   - Adjust vertex positions
4. Correct penetrations with collision geometry
   - Adjust vertex positions

How many vertices?

- How many vertices fit in 256K (less actually)?
  - A lot, surprisingly…
- Tips:
  - Look for opportunities to stream data
  - Keep in LS only data required for each step

Integration Step

$$p^{t+1} = (2p^t - p^{t-1}) + \frac{1}{2} f^E * \frac{1}{m} * \Delta t^2$$

16 + 16 + 16 + 4 = 52 bytes / vertex

- Less than 4000 verts in 200K of memory
- We don't need to keep them all in LS
- Keep vertex data in main memory and bring it in in blocks

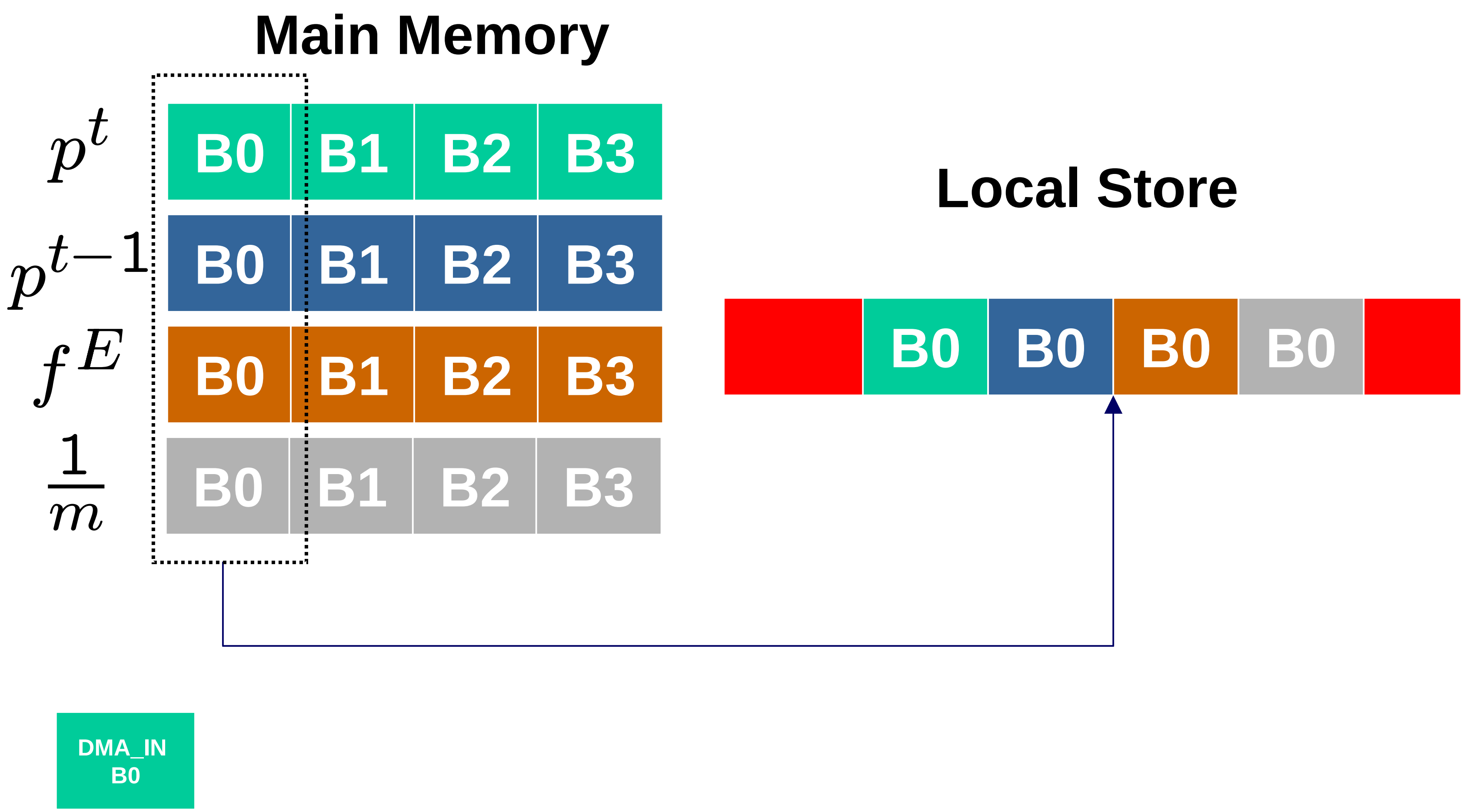
Streaming Integration

Main Memory

p^t	B0	B1	B2	B3
p^{t-1}	B0	B1	B2	B3
f^E	B0	B1	B2	B3
$\frac{1}{m}$	B0	B1	B2	B3

Local Store

Streaming Integration



Streaming Integration

Main Memory

p^t	B0	B1	B2	B3
p^{t-1}	B0	B1	B2	B3
f^E	B0	B1	B2	B3
$\frac{1}{m}$	B0	B1	B2	B3

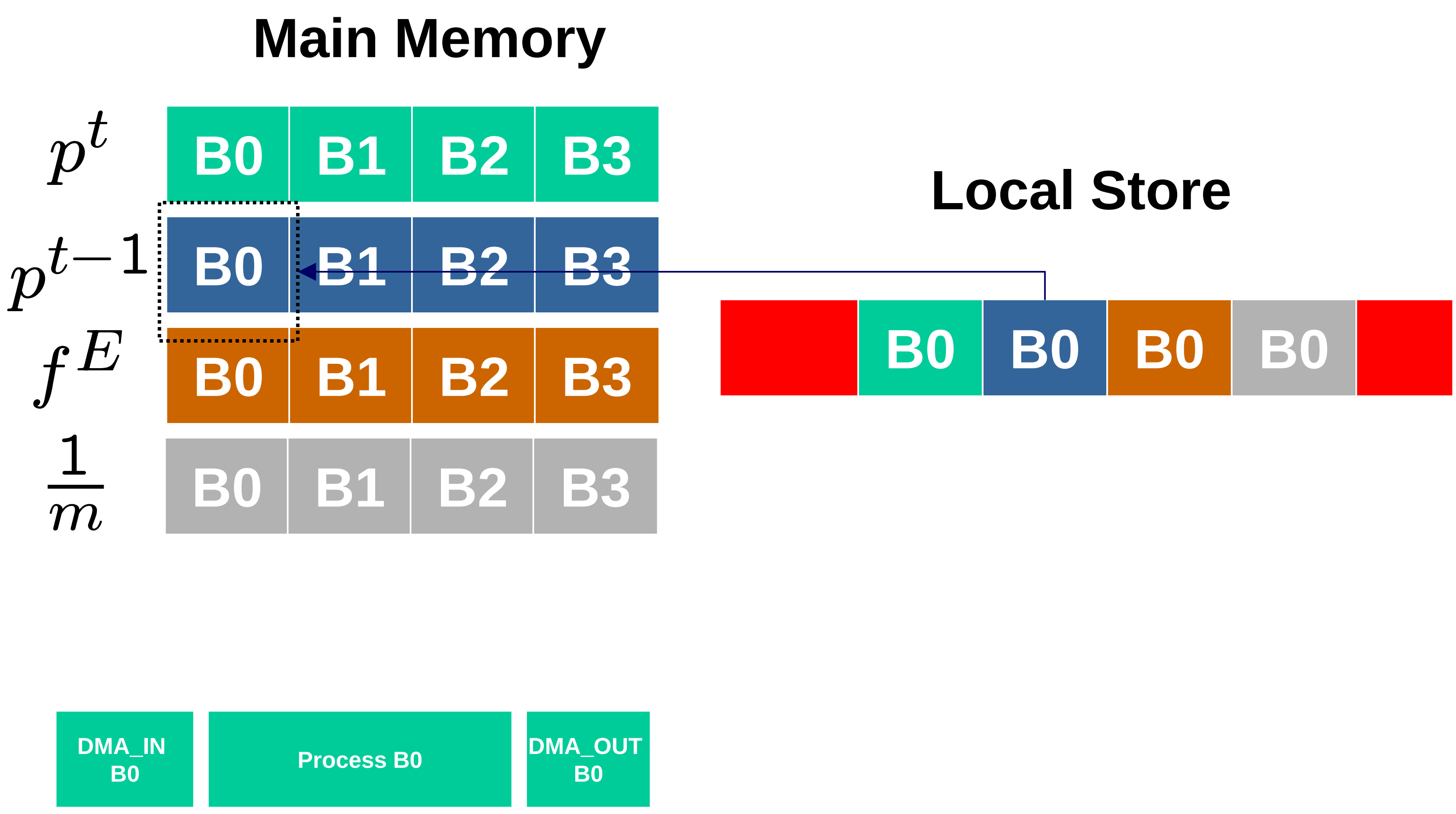
Local Store

	B0	B0	B0	B0	

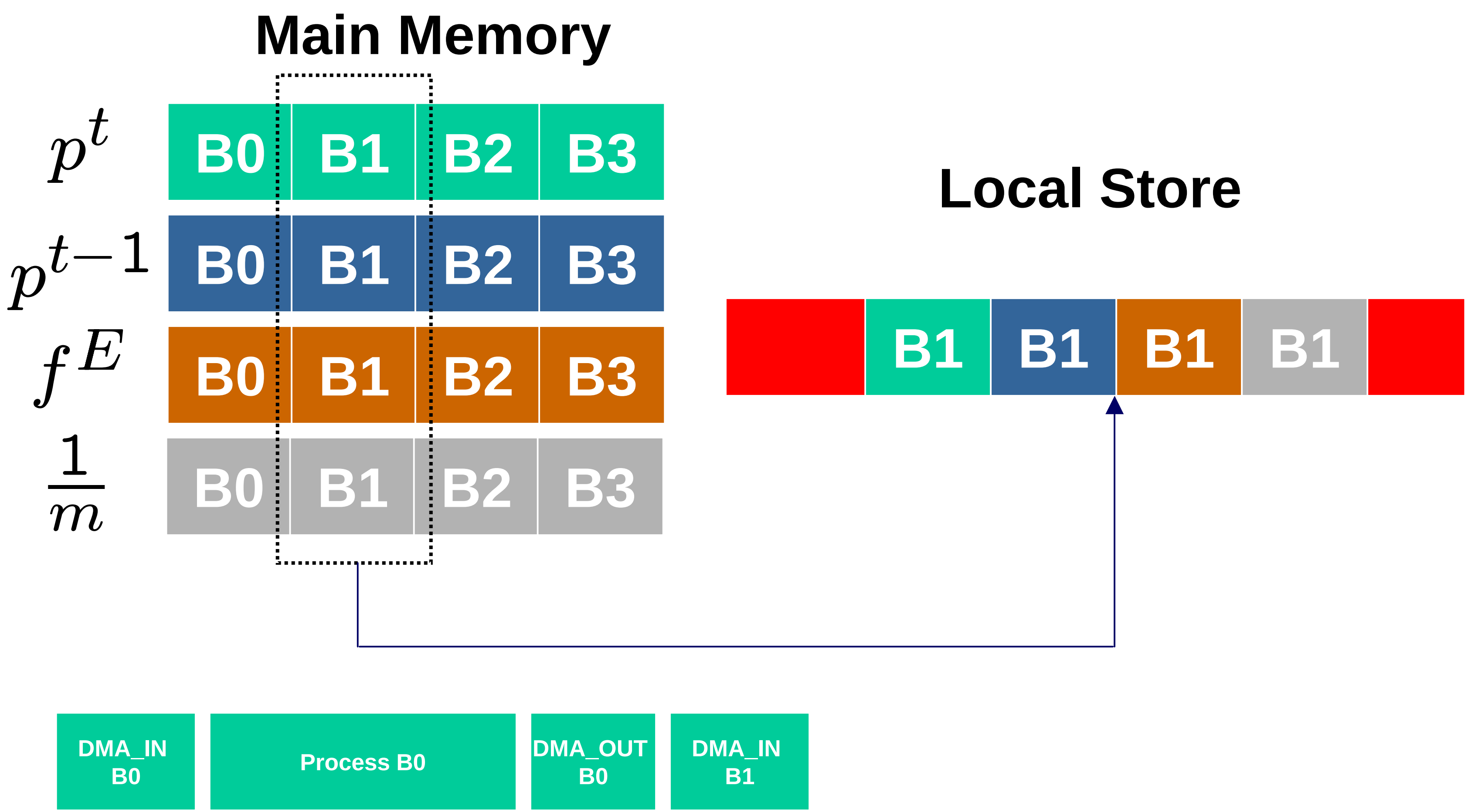
DMA_IN B0

Process B0

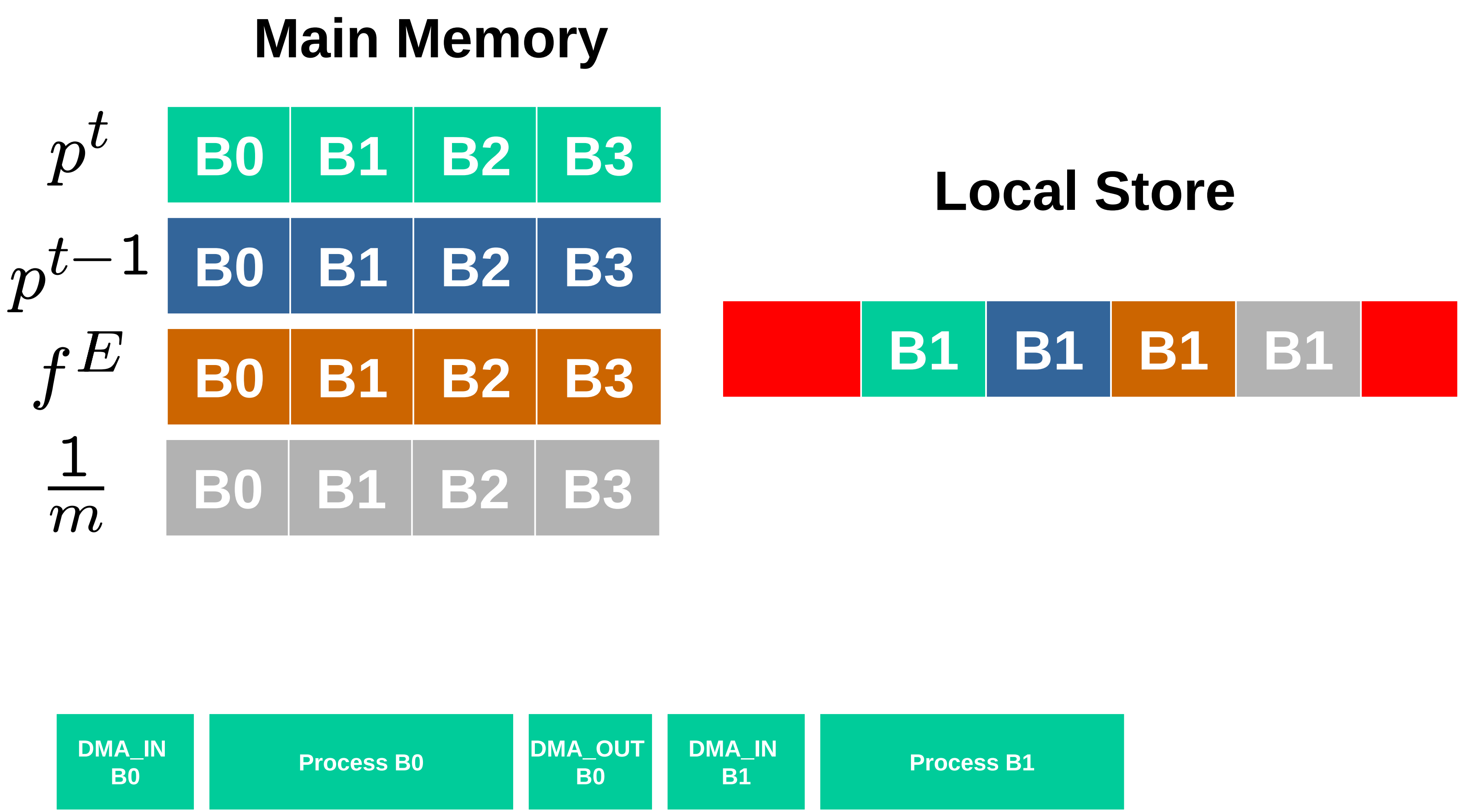
Streaming Integration



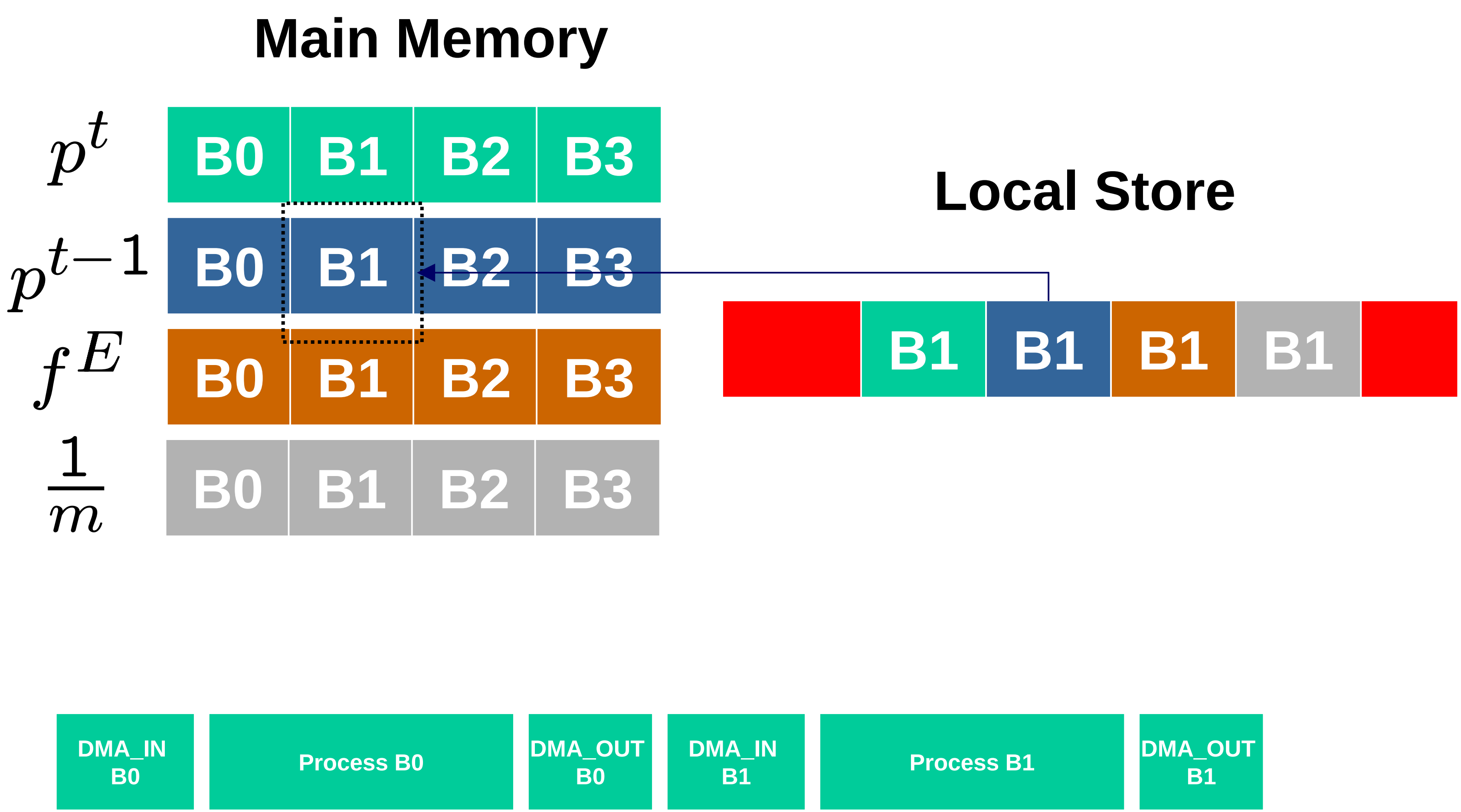
Streaming Integration



Streaming Integration



Streaming Integration



Streaming Integration

Main Memory

p^t	B0	B1	B2	B3
p^{t-1}	B0	B1	B2	B3
f^E	B0	B1	B2	B3
$\frac{1}{m}$	B0	B1	B2	B3

Local Store

	B1	B1	B1	B1	

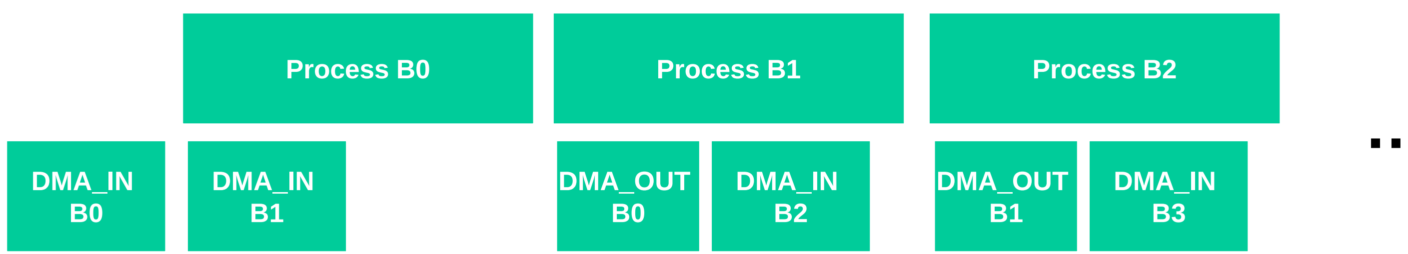
DMA_IN B0	Process B0	DMA_OUT B0	DMA_IN B1	Process B1	DMA_OUT B1	...

Double-buffering

- Take advantage of concurrent DMA and processing to hide transfer times

Without double-buffering:

With double-buffering:



Streaming Data

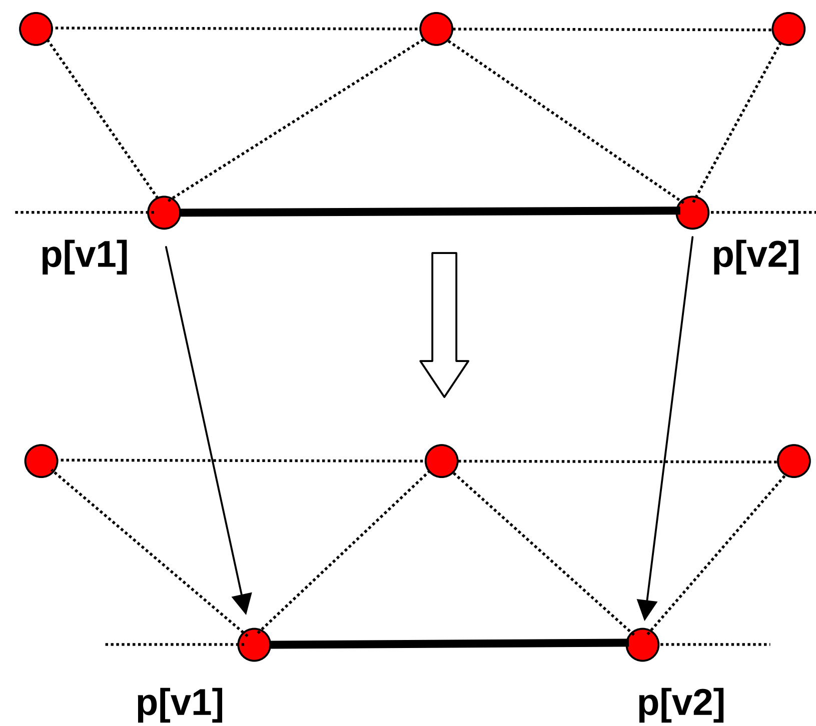
- Streaming is possible when the data access pattern is simple and predictable (e.g. linear)
  - Number of verts processed per frame depends on processing speed and bandwidth but not LS size
- Unfortunately, not every step in the cloth solver can be fully streamed
  - Fixing edge lengths requires random memory access…

Fixing Edge Lengths

- Points coming out of the integration step don't necessarily satisfy edge distance constraints

```
struct Edge
{
    int v1;
    int v2;
    float restLen;
}
```

```
Vector3 d = p[v2] – p[v1];
float len = sqrt(dot(d,d));
diff = (len-restLen)/len;
p[v1] -= d * 0.5 * diff;
p[v2] += d * 0.5 * diff;
```



Fixing Edge Lengths

- An iterative process: Fix one edge at a time by adjusting 2 vertex positions
- Requires random access to particle positions array
- Solution:
  - Keep all particle positions in LS
  - Stream in edge data
  - In 200K we can fit 200KB / 16B > 12K vertices

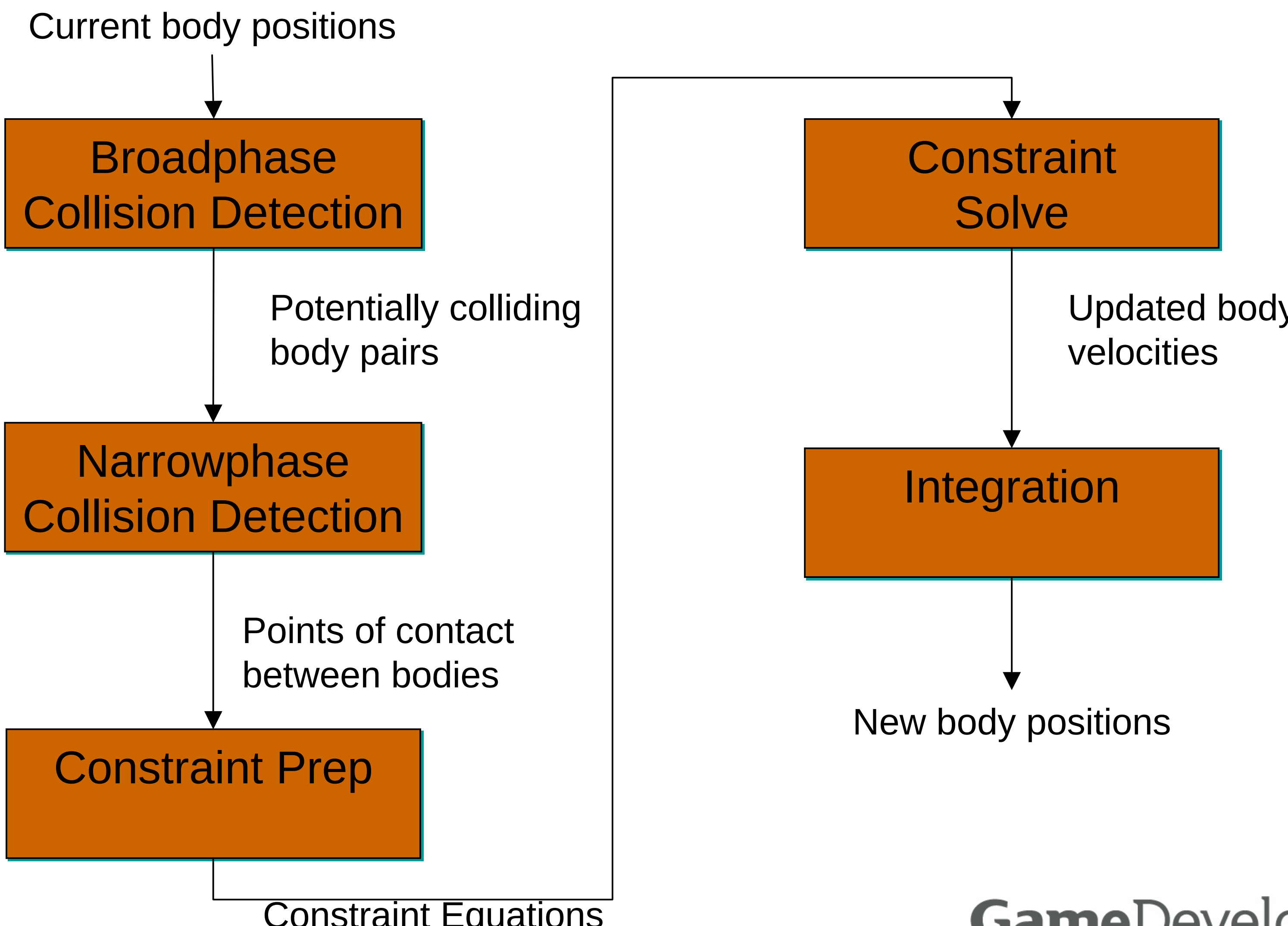
Rigid Bodies

- Our group is currently porting the AGEIA™ PhysX™ SDK to CELL
- Large codebase written with a PC architecture in mind
  - Assumes easy random access to memory
  - Processes tasks sequentially (no parallelism)
- Interesting example on how to port existing code to a multi-core architecture

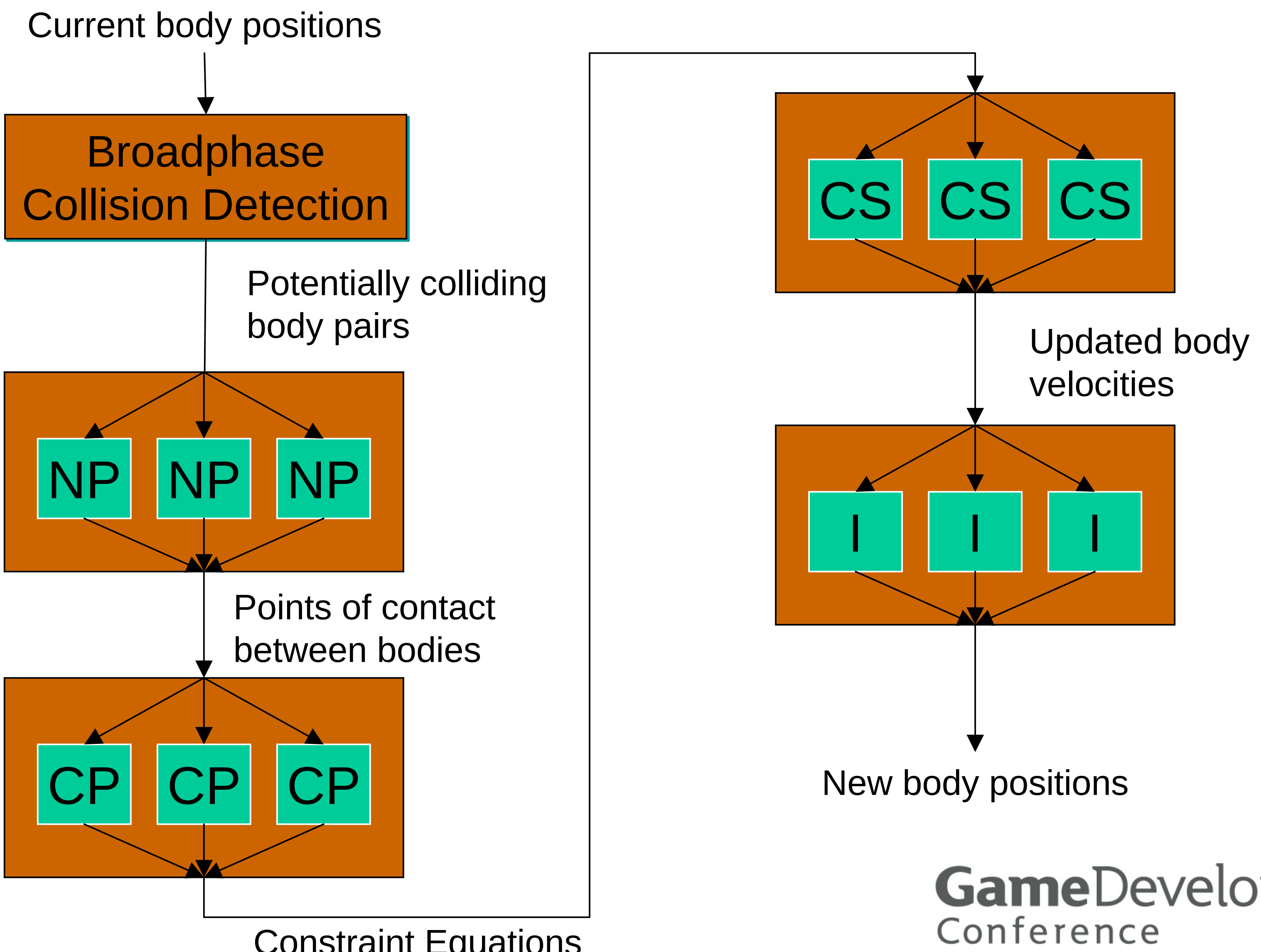
Starting the Port

- Determine all the stages of the rigid body pipeline
- Look for stages that are good candidates for parallelizing/optimizing
- Profile code to make sure we are focusing on the right parts

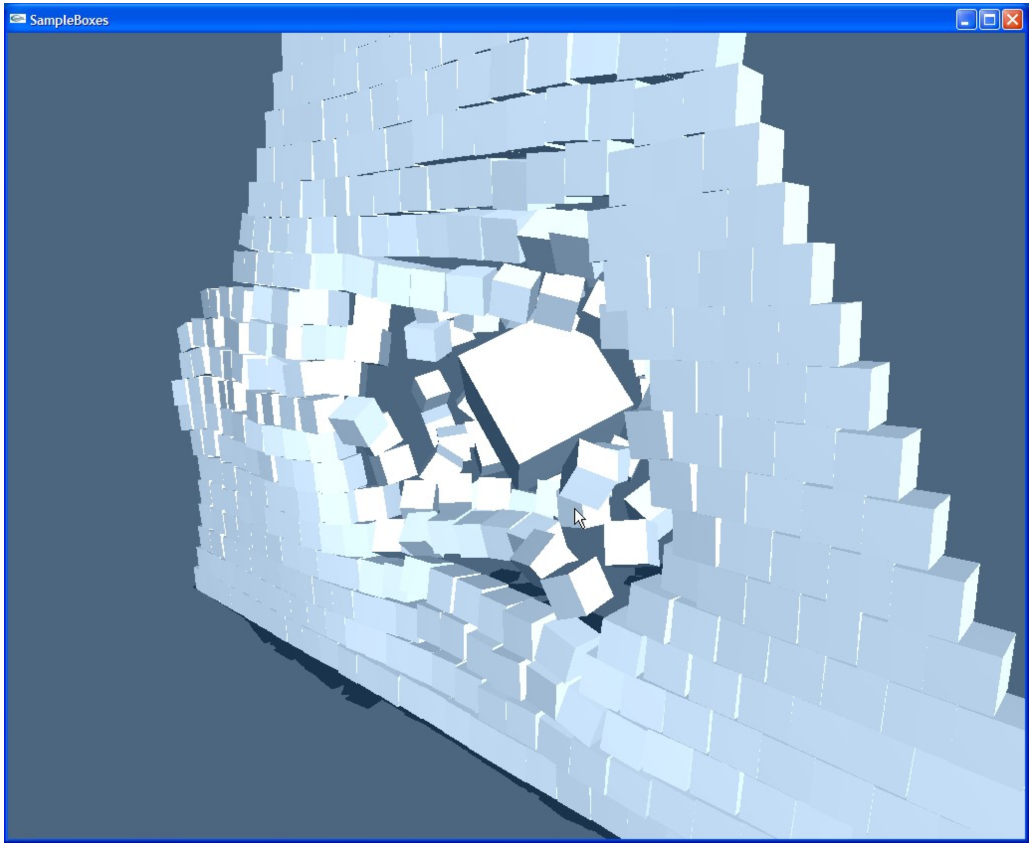
Rigid Body Pipeline



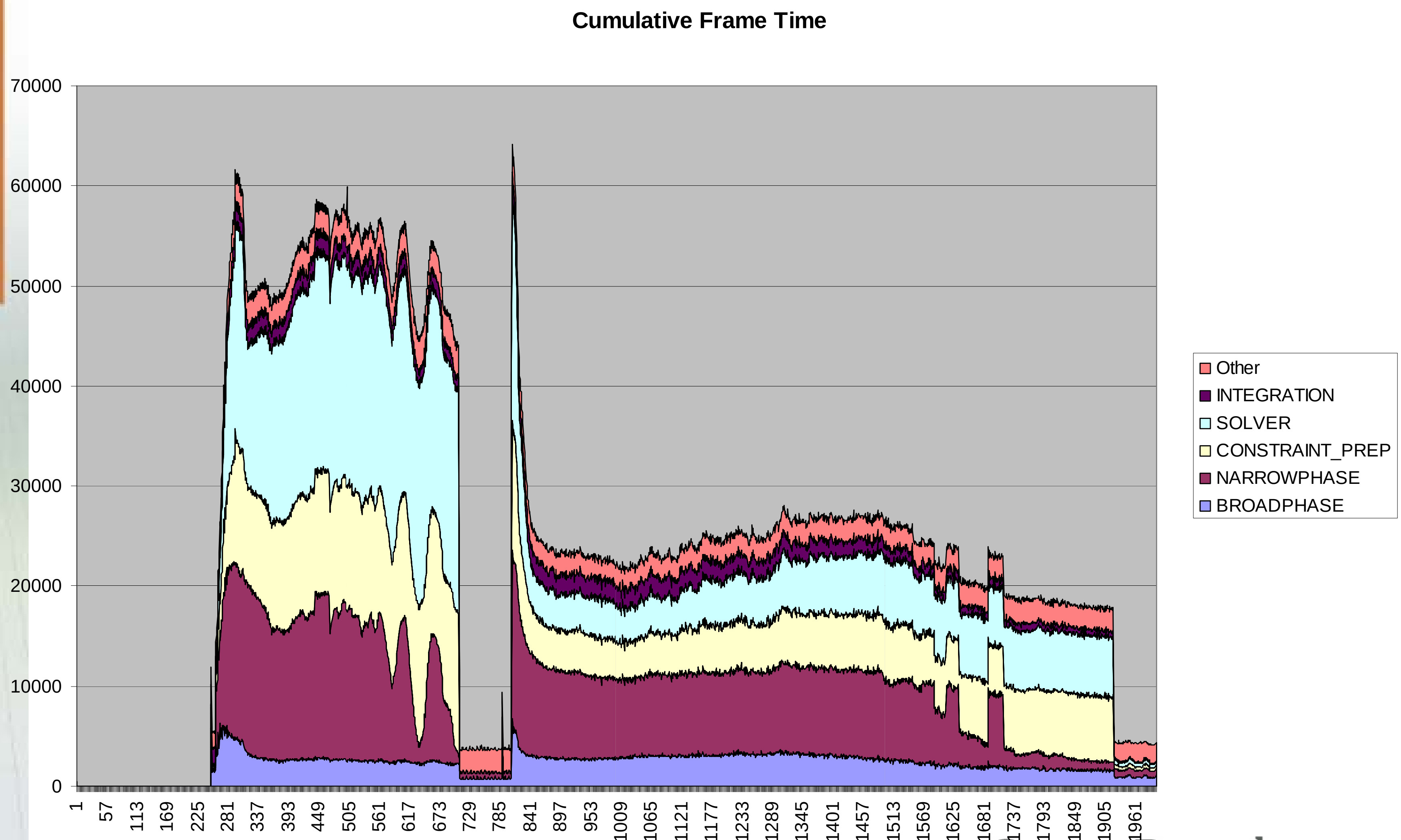
Rigid Body Pipeline



Profiling Scenario



Profiling Results



Running on the SPUs

- Three steps:
  1. (PPU) *Pre-process*
     - "Gather" operation (extract data from PhysX data structures and pack it in MM)
  2. (SPU) *Execute*
     - DMA packed data from MM to LS
     - Process data and store output in LS
     - DMA output to MM
  3. (PPU) *Post-process*
     - "Scatter" operation (unpack output data and put back in PhysX data structures)

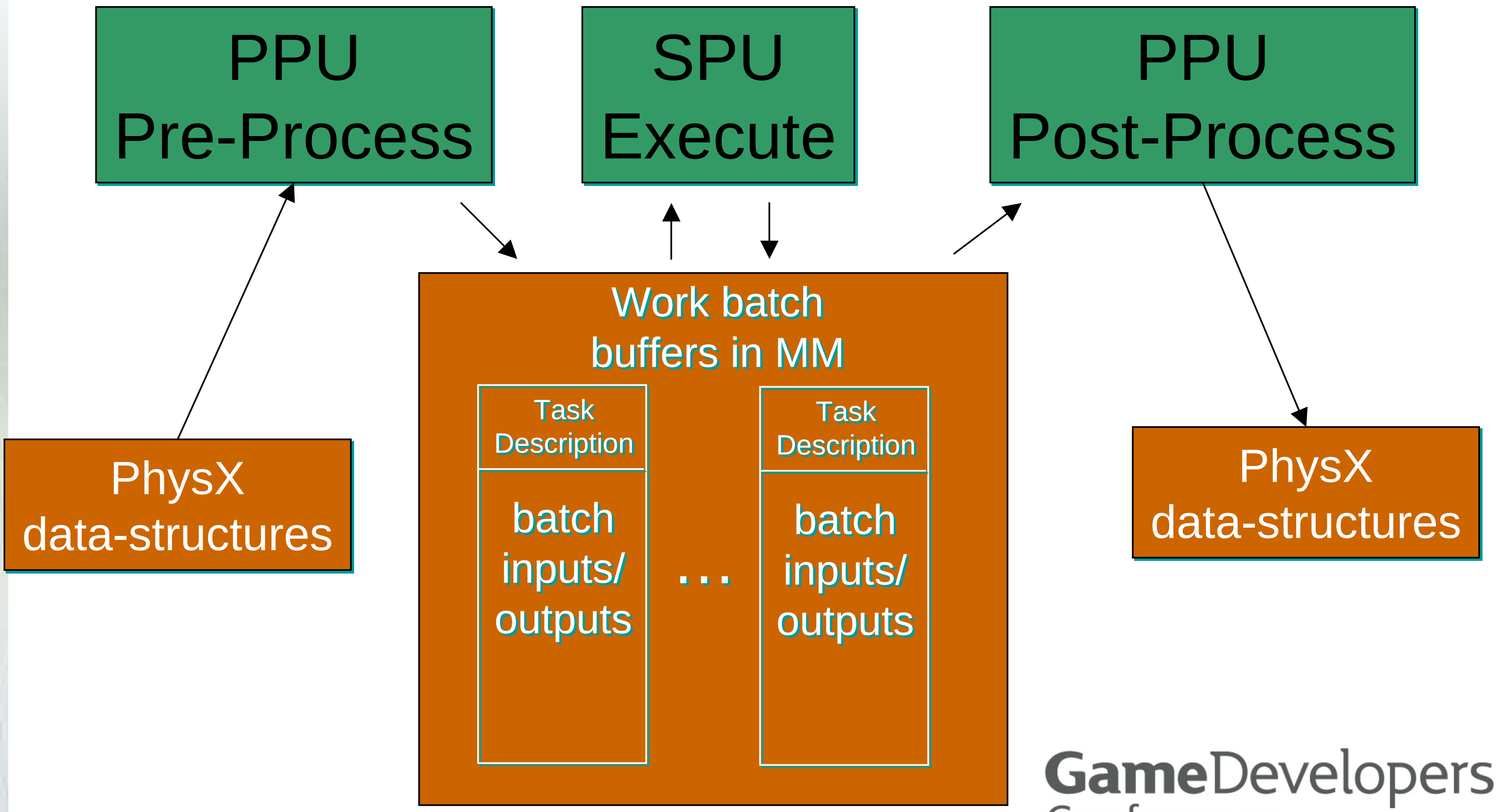
Why Involve the PPU?

- Required PhysX data is not conveniently packed
- Data is often not aligned
- We need to use PhysX data structures to avoid breaking features we haven't ported

- Solutions:
  - Use list DMAs to bring in data
  - Modify existing code to force alignment
  - Change PhysX code to work with new data structures

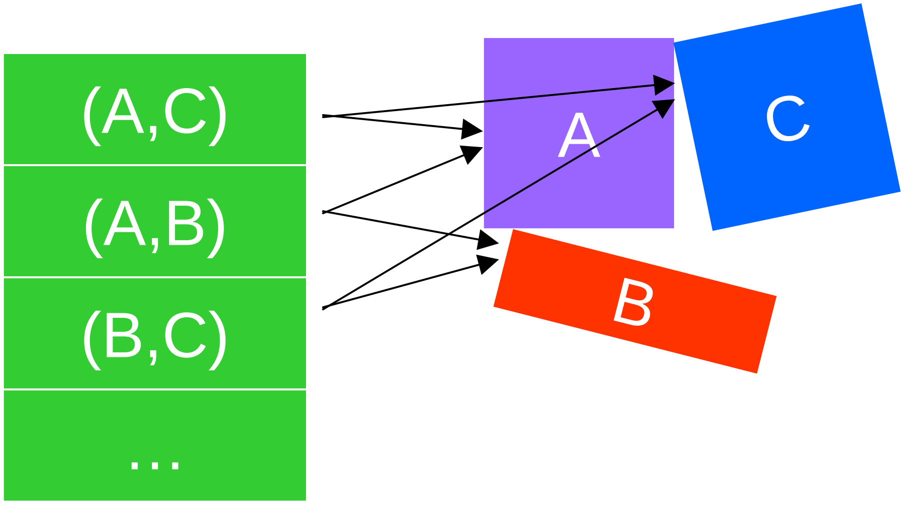
Batching Up Work

- Create work batches for each task



Narrow-phase Collision Detection

- Problem:
  - A list of object pairs that may be colliding
  - Want to do contact processing on SPUs
  - Pairs list has references to geometry



Narrow-phase Collision Detection

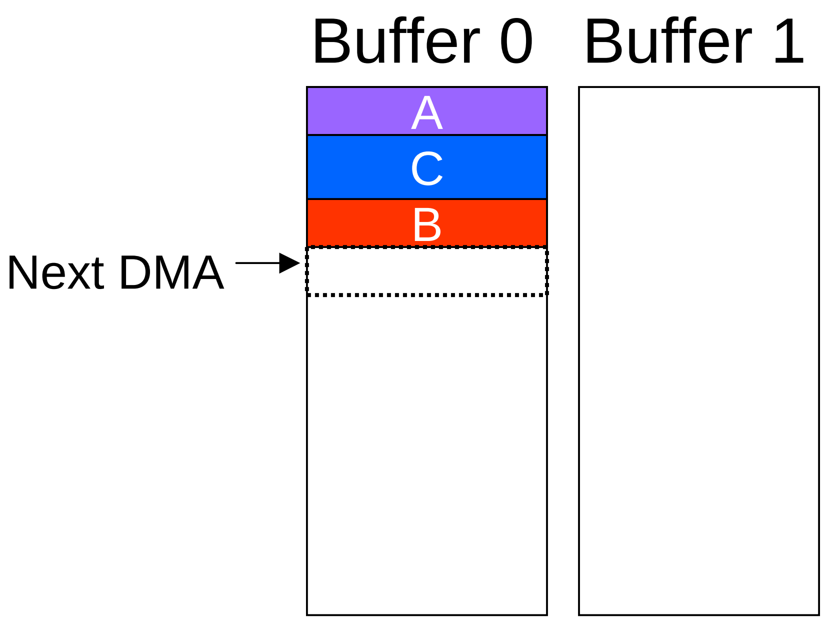
- Data locality
  - Same bodies may be in several pairs
  - Geometry may be instanced for different bodies
- SPU memory access
  - Can only access main memory with DMA
  - No hardware cache
  - Data reuse must be explicit

Software Cache

- Idea: make a (read-only) software cache
  - Cache entry is one geometric object
  - Entries have variable size
- Basic operation
  - SPU checks cache for object
  - If not in cache, object fetched with DMA
  - Cache returns a local address for object

Software Cache

- Data Structures
  - Two entry buffers
  - New entries appended to “current” buffer
    - Hash-table used to record and find loaded entries

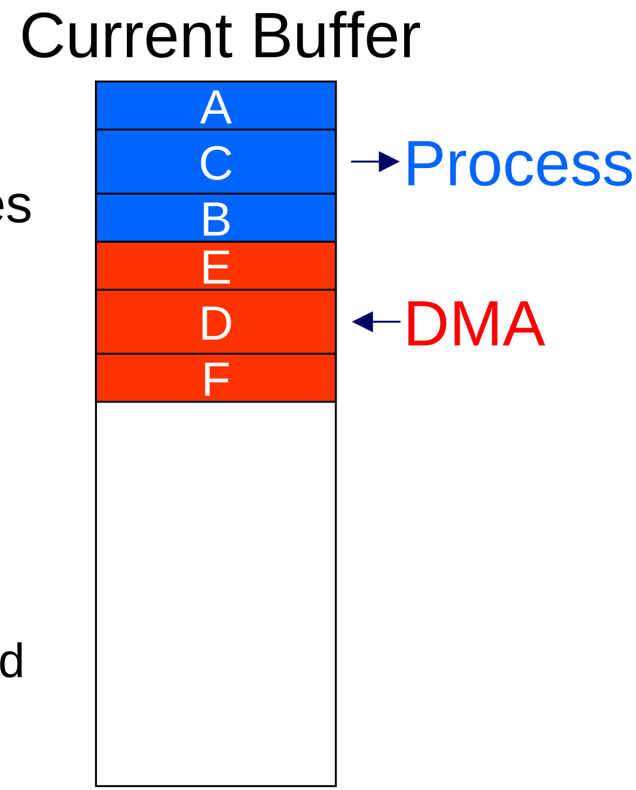


Software Cache

- Data Replacement
  - When space runs out in a buffer
    - Overwrite data in second buffer
  - Considerations
    - Does not fragment memory
    - No searches for free space
    - But does not prefer frequently used data

Software Cache

- Hiding the DMA latency
  - Double-buffering
    - Start DMA for un-cached entries
    - Process previously DMA'd entries
  - Process/pre-fetch batches
    - Fetch and compute times vary
      - Batching may improve balance
    - DMA-lists useful
      - One DMA command
      - Multiple chunks of data gathered



Software Caching

- Conclusions
  - Simple cache is practical
    - Used for small convex objects in PhysX
  - Design considerations
    - Tradeoff of cache-logic cycles vs. bandwidth saved
    - Pre-fetching important to include

Single SPU Performance

- PPU only:

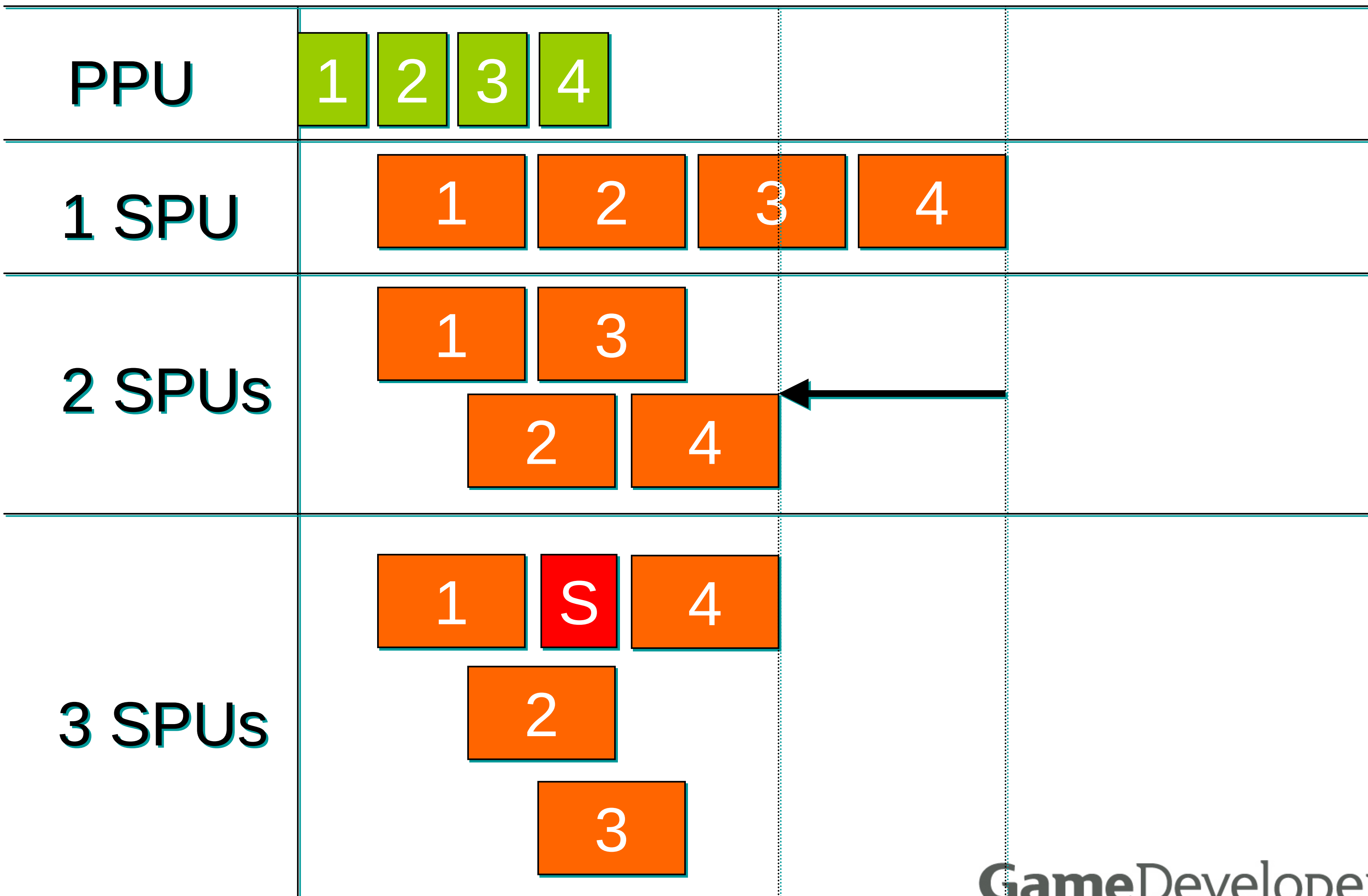
- PPU + SPU:

SPU Exec < PPU Exec: SIMD + fast mem access

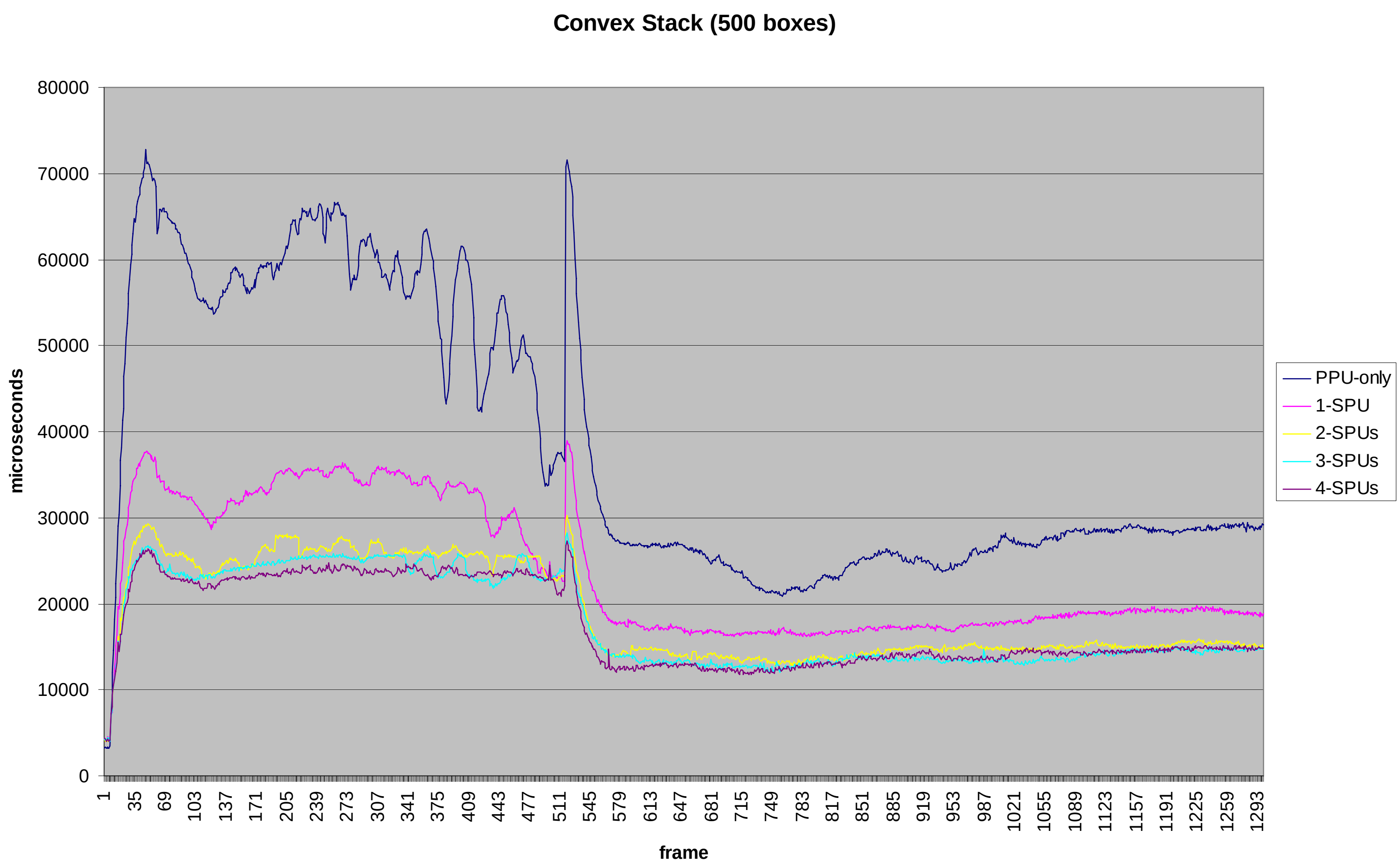
Multiple SPU Performance

- Pre- and Post- processing times determine how many SPUs can be used effectively

Multiple SPU Performance



PPU vs SPU comparisons



Duck Demo

- One of our first CELL demos (spring 2005)
- Several interacting physics systems:
  - Rigid bodies (ducks & boats)
  - Height-field water surface
  - Cloth with ripping (sails)
  - Particle based fluids (splashes + cups)

Duck Demo (Lots of Ducks)

Duck Demo

- Ambitious project with short deadline
- Early PC prototypes of some pieces
- Most straightforward way to parallelize:
  - Dedicate one SPU for each subsystem
- Each piece could be developed and tested individually

Duck Demo Resource Allocation

- PU – main loop
  - SPU thread synchronization, draw calls
- SPU0 – height field water (<50%)
- SPU1 – splashes iso-surface (<50%)
- SPU2 – cloth sails for boat 1 (<50%)
- SPU3 – cloth sails for boat 2 (<50%)
- SPU4 – rigid body collision/response (95%)

Parallelization Recipe

One three-step approach to code parallelization:

1. Find independent components
2. Run them side-by-side
3. Recursively apply recipe to components

Challenges

Step 1: Find independent components

- Where do you look?
- Maybe you need to break apart and overlap your data?
  - → e.g. Broad phase collision detection
- Maybe you need to break apart your loop into individual iterations?
  - → e.g. Solving cloth constraints

Broad Phase Collision Detection

Need to test 600 rigid bodies against each other.

600 Objects

200 Objects A | 200 Objects B | 200 Objects C

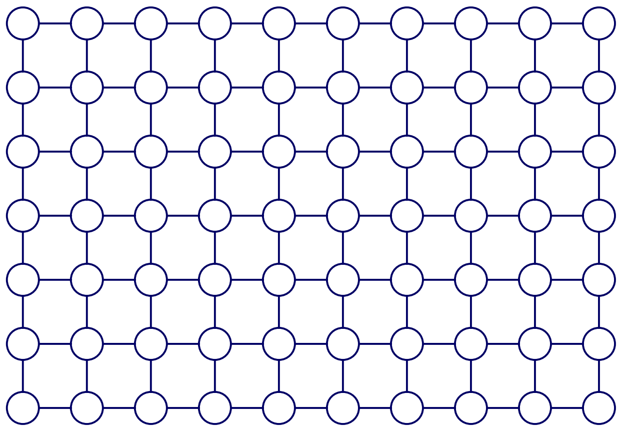
200 Objects A vs 200 Objects B

200 Objects A vs 200 Objects C

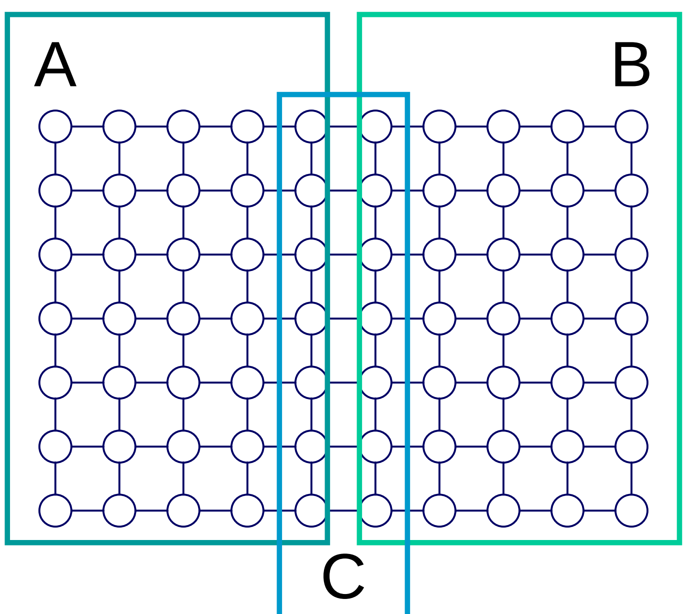
200 Objects B vs 200 Objects C

We can execute all three of these simultaneously

Cloth Solving



```
for (i=1 to 5) {
    cloth=solve(cloth)
}
```




```
for (i=1 to 5) {
    solve_on_proc1(a);
    solve_on_proc2(b);
    wait_for_all()
    solve_on_proc1(c);
    wait_for_all();
}
```

…challenges

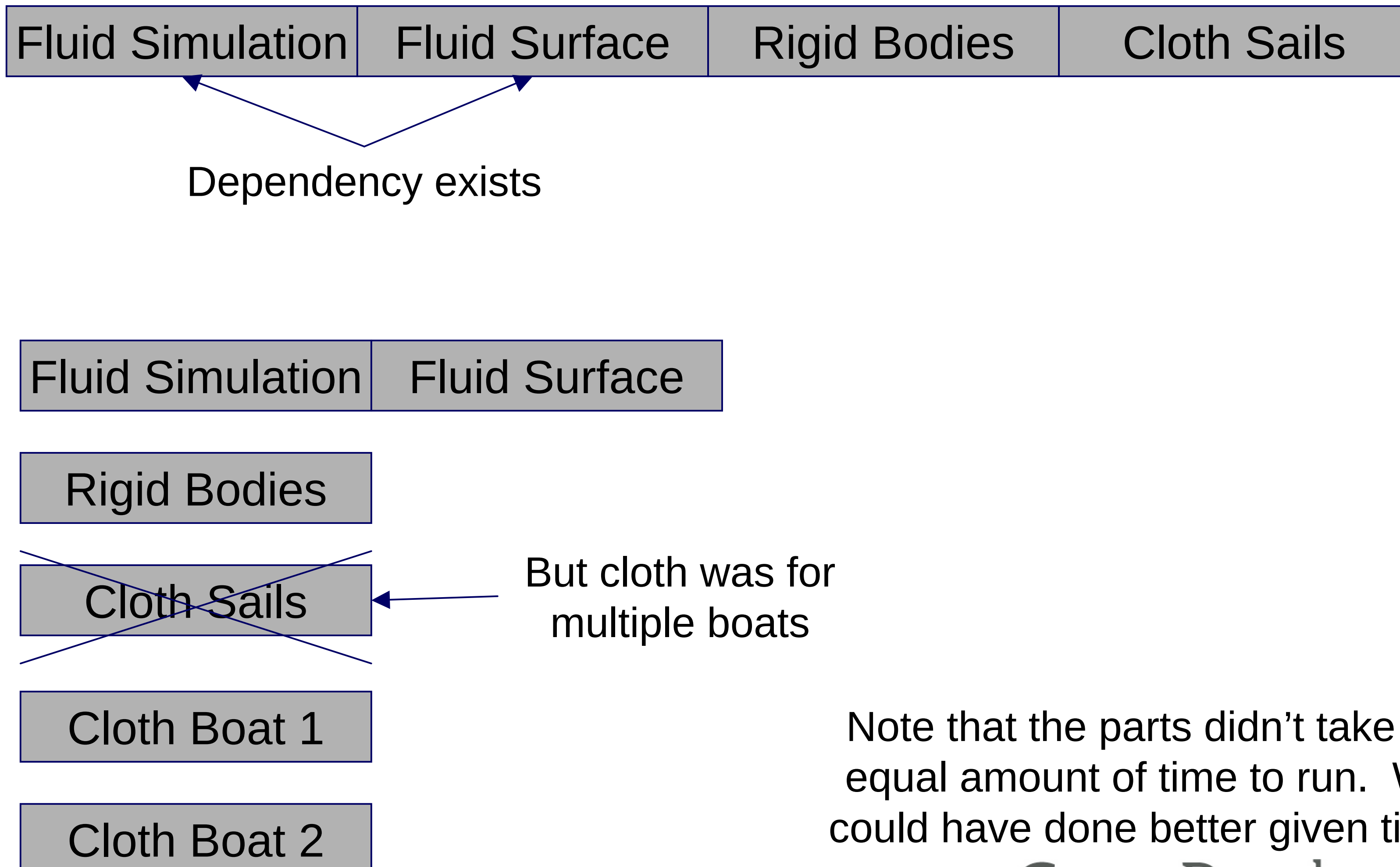
Step 2: Run them side-by-side

- Bandwidth and cache issues
  → Need good data layout to avoid thrashing cache or bus
- Processor issues
  → Need efficient processor management scheme
- What if the job sizes are very different?
  e.g. a suit of cloth and a separate neck tie
  → Need further refinement of large jobs, or you only save on the small neck tie time

…challenges

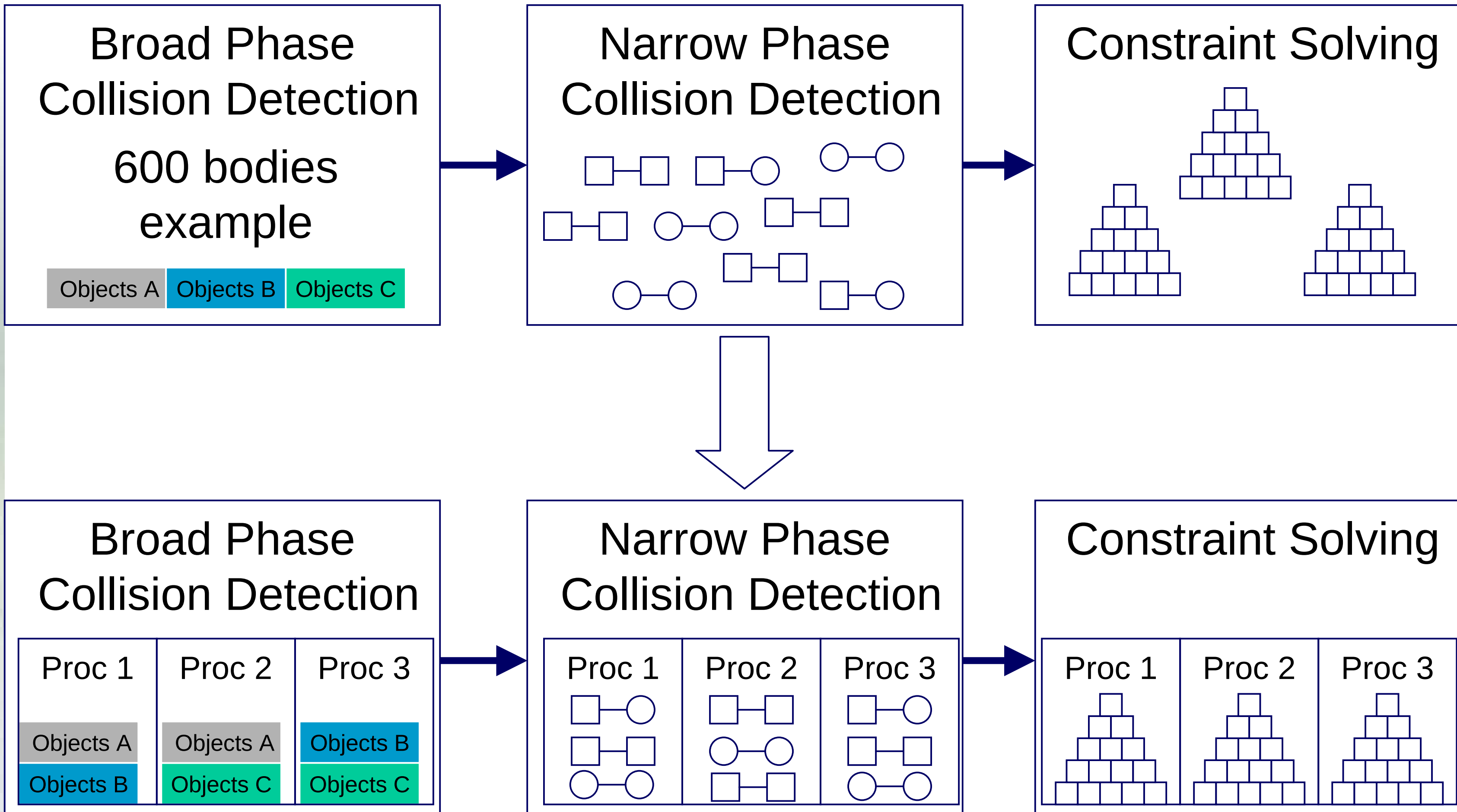
- Step 3: Recurse
- When do you stop?
  - → Overhead of launching smaller jobs
  - → Synchronization when a stage is done
    - e.g. Gather results from all collision detection before solving
- But this *can* go down to the instruction level
  - e.g. Using Structure-of-Arrays, transform four independent vectors at once

High Level Parallelization: Duck Demo



Note that the parts didn't take an equal amount of time to run. We could have done better given time!

Lower Level Parallelization
Rigid Body Simulation



Structure of Arrays

Array of Structures or “AoS”

Data[0]	X_0	Y_0	Z_0	W_0
Data[1]	X_1	Y_1	Z_1	W_1
Data[2]	X_2	Y_2	Z_2	W_2
Data[3]	X_3	Y_3	Z_3	W_3
Data[4]	X_4	Y_4	Z_4	W_4
Data[5]	X_5	Y_5	Z_5	W_5
Data[6]	X_6	Y_6	Z_6	W_6
Data[7]	X_7	Y_7	Z_7	W_7

1 AoS Vector

Structure of Arrays or “SoA”

Data[0]	X_0	X_1	X_2	X_3
Data[1]	Y_0	Y_1	Y_2	Y_3
Data[2]	Z_0	Z_1	Z_2	Z_3
Data[3]	W_0	W_1	W_2	W_3
Data[4]	X_4	X_5	X_6	X_7
Data[5]	Y_4	Y_5	Y_6	Y_7
Data[6]	Z_4	Z_5	Z_6	Z_7
Data[7]	W_4	W_5	W_6	W_7

1 SoA Vector

Bonus!
Since W is almost always 0 or 1, we can eliminate it with a clever math library and save 25% memory and bandwidth!

Lowest Level Parallelization:

Structure-of-Array processing of Particles

Given:

$p_n(t)$=position of particle n at time t

$v_n(t)$=velocity of particle n at time t

$$p_1(t_i)=p_1(t_{i-1}) + v_1(t_{i-1}) * dt + 0.5 * G * dt^2$$

$$p_2(t_i)=p_2(t_{i-1}) + v_2(t_{i-1}) * dt + 0.5 * G * dt^2$$

...

Note they are independent of each other

So we can run four together using SoA

$$p_{\{1\text{-}4\}}(t_i)=p_{\{1\text{-}4\}}(t_{i-1}) + v_{\{1\text{-}4\}}(t_{i-1}) * dt + 0.5 * G * dt^2$$

Failure Case

Gauss Seidel Solver

Consider a simple position-based solver that uses distance constraints. Given:

- **p**=current positions of *all* objects
- solve(c_n, **p**) takes **p** and constraint c_n and computes a new **p** that satisfies c_n

p=solve(c_0, **p**)

p=solve(c_1, **p**)

…

Note that to solve c_1, we need the result of c_0. Can't solve c_0 and c_1 concurrently!

Failure Case
Possible Solutions

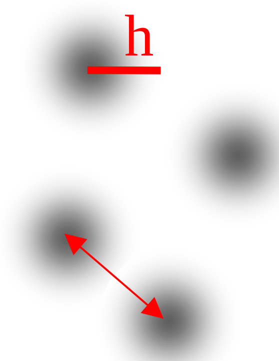
Generally, it's you're out of luck, but…

- Some cases have very limited dependencies
  e.g. particle-based cloth solving
  - → Solution: Arrange constraints such that no four adjacent constraints share cloth particles
- Consider a different solver
  e.g. Jacobi solvers don't use updated values until all constraints have been processed once
  - But they need more memory ($\mathbf{p}_{new}$ and $\mathbf{p}_{current}$)
  - And may need more iterations to converge

Duck Demo (EyeToy + SPH)

Smoothed Particle Hydrodynamics (SPH) Fluid Simulation

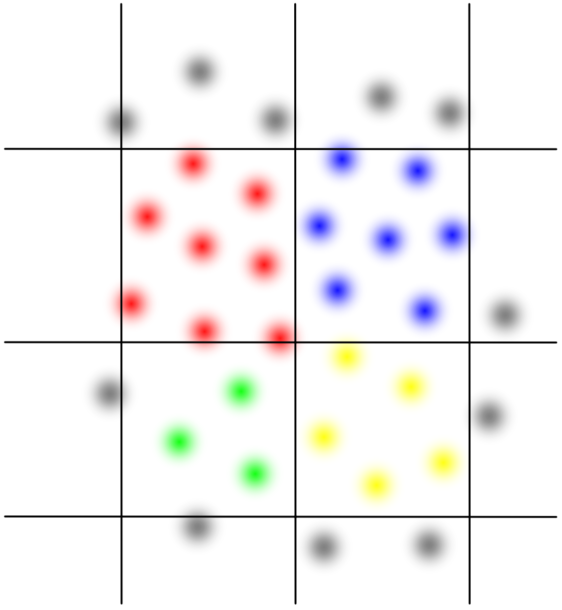
- Smoothed-particles
  - Mass distributed around a point
  - Density falls to 0 at a radius *h*



- Forces between particles closer than 2*h*

SPH Fluid Simulation

- High-level parallelism
  - Put particles in grid cells
  - Process on different SPUs
  - (Not used in duck demo)



- Low-level parallelism
  - SIMD and dual-issue on SPU
  - Large *n* per cell may be better
    - Less grid overhead
    - Loops fast on SPU

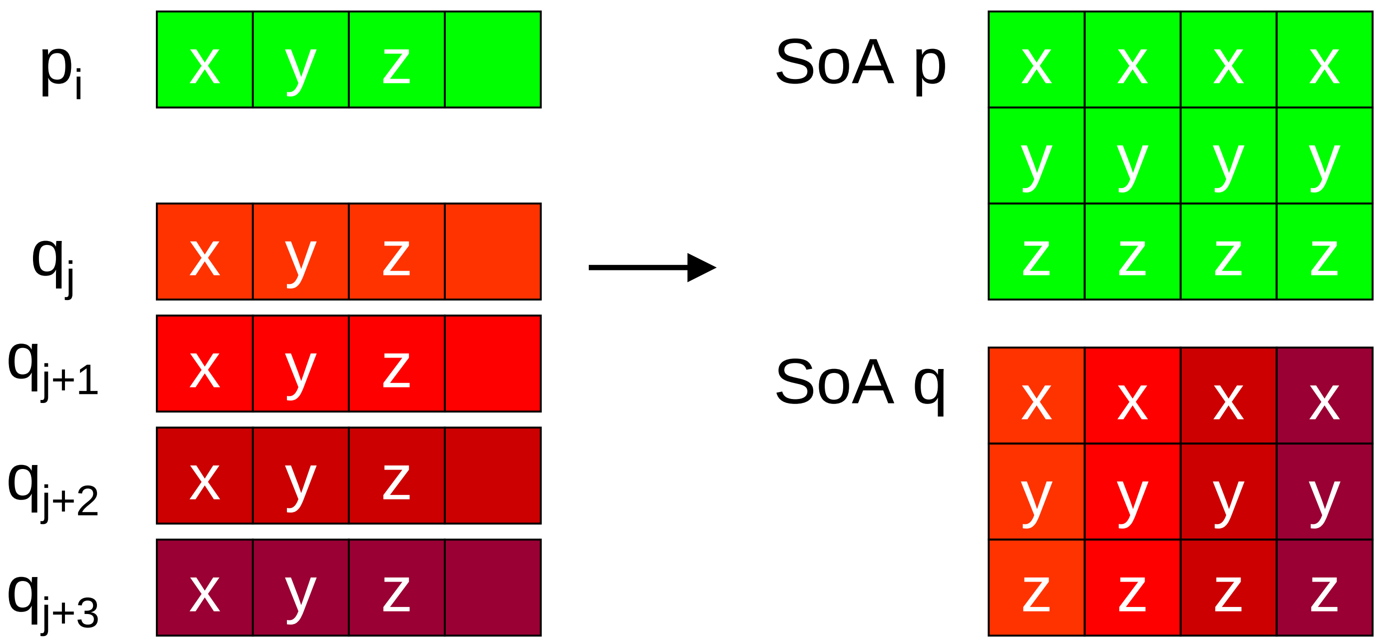
SPH Loop

- Consider two sets of particles P and Q
  - E.g., taken from neighbor grid cells
  - $O(n^2)$ problem
- Can unroll (e.g., by 4)

```
for (i = 0; i < numP; i++)
    for (j = 0; j < numQ; j+=4)
        Compute force (p_i, q_j)
        Compute force (p_i, q_j+1)
        Compute force (p_i, q_j+2)
        Compute force (p_i, q_j+3)
```

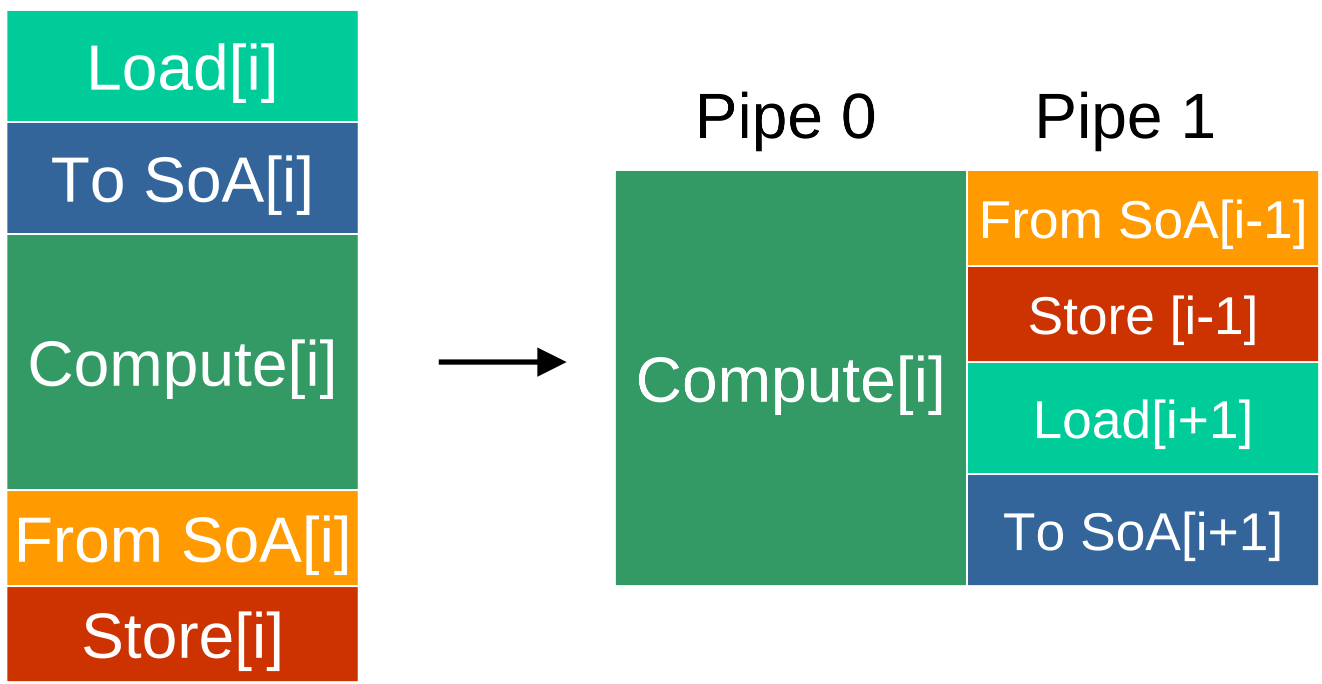
SPH Loop, SoA

- Idea:
  - Increase SIMD throughput with structure-of-arrays
  - Transpose and produce combinations



SPH Loop, Software Pipelined

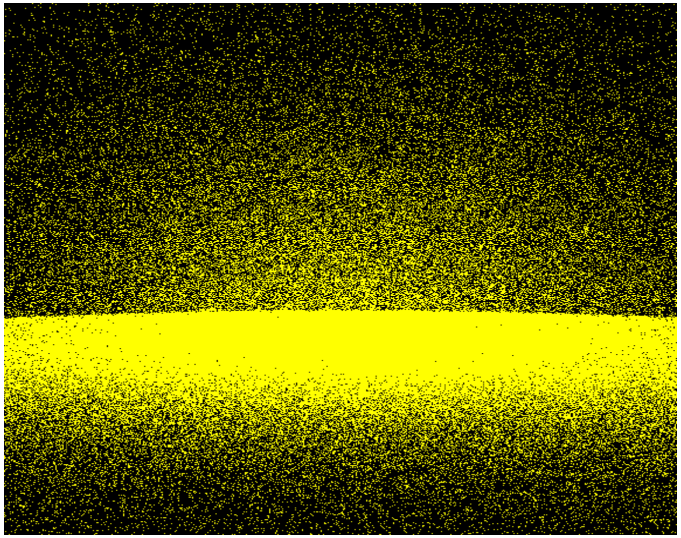
- Add software pipelining
  - Conversion instructions can dual-issue with math

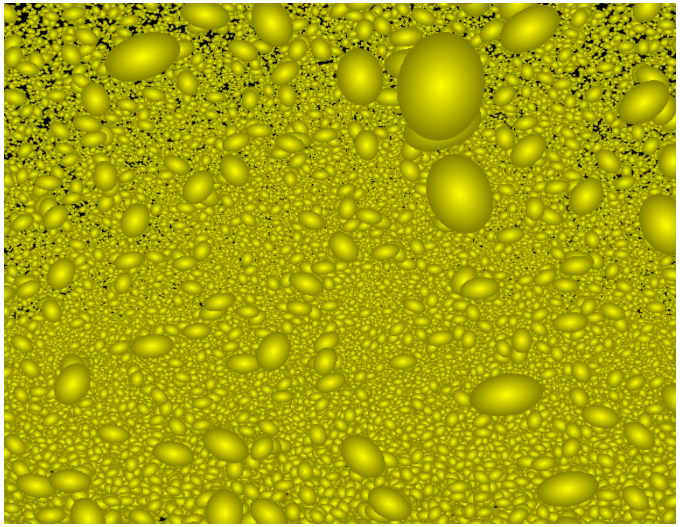


Recap

- Finding independence is hard!
  - Across subsystems or within subsystems?
  - Across iterations or within iterations?
  - Data level independence?
  - Instruction level independence?
  - How about “bandwidth level” independence?
- Parallelization overhead
  - Sometimes running serially wins over overhead of parallelization

Particle Simulation Demo





Questions?

http://www.research.scea.com/

Contacts:

Vangelis Kokkevis: vangelis_kokkevis@playstation.sony.com

Eric Larsen: eric_larsen@playstation.sony.com

Steven Osman: steven_osman@playstation.sony.com