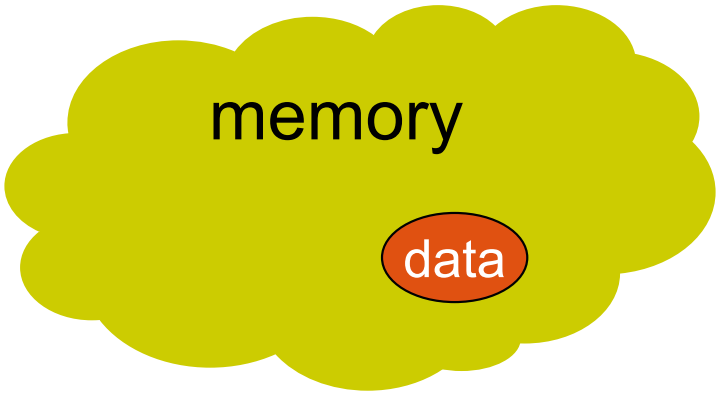
6.189 IAP 2007

Recitations 2-3

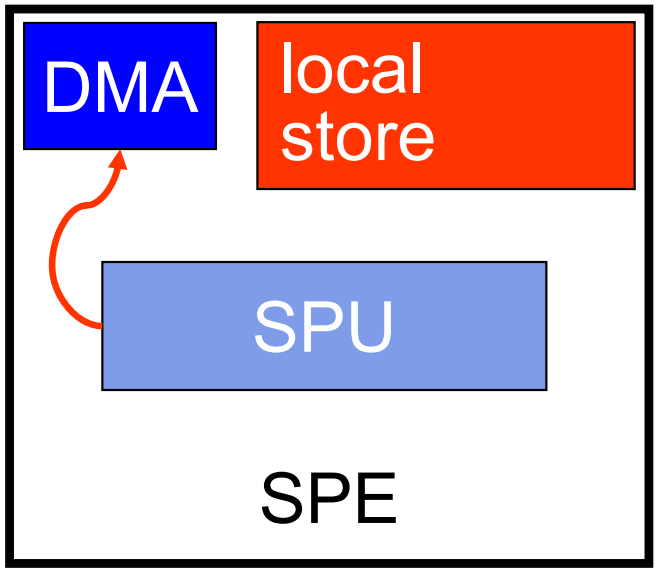
Cell Programming Hands-on

Transferring Data In/Out of SPU Local Store



- SPU needs data

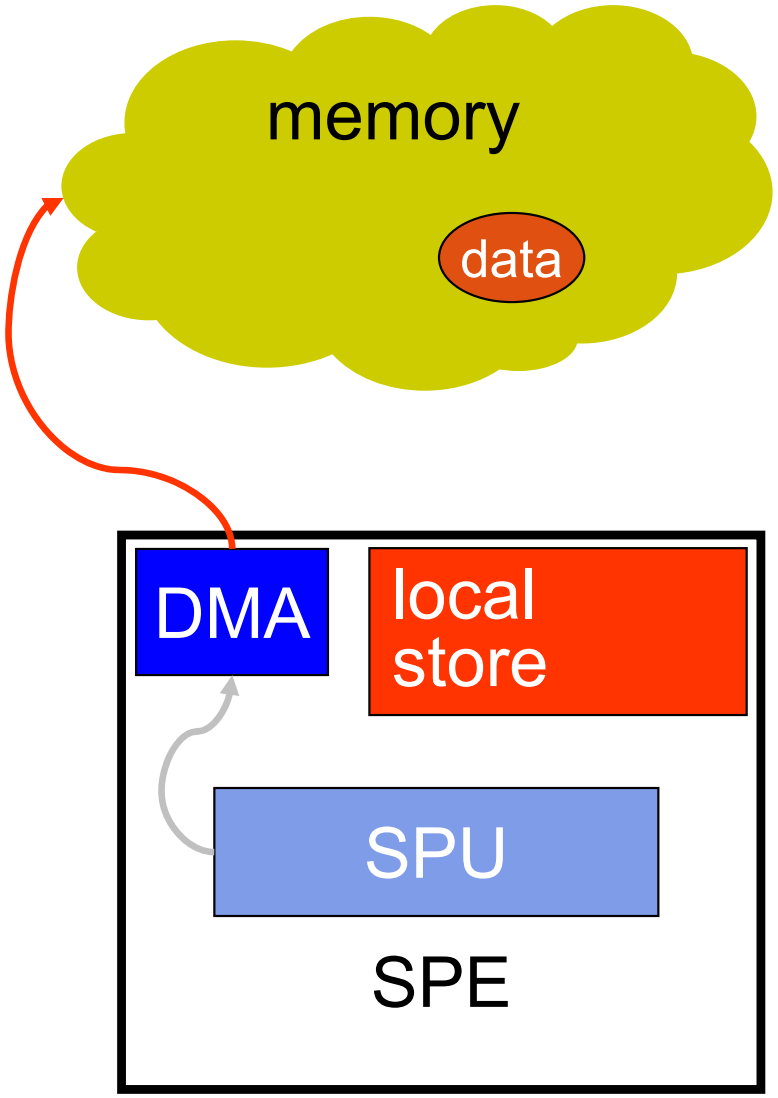
1. SPU initiates DMA request for data



Transferring Data In/Out of SPU Local Store

- SPU needs data

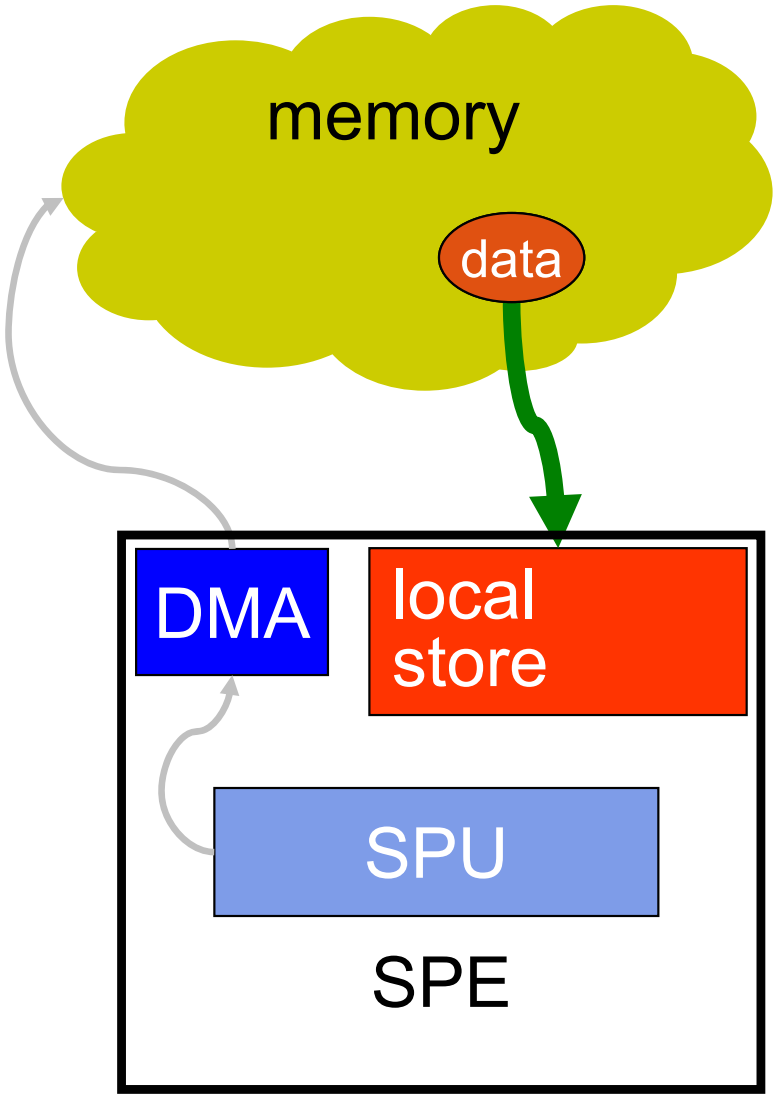
1. SPU initiates DMA request for data
2. DMA requests data from memory



Transferring Data In/Out of SPU Local Store

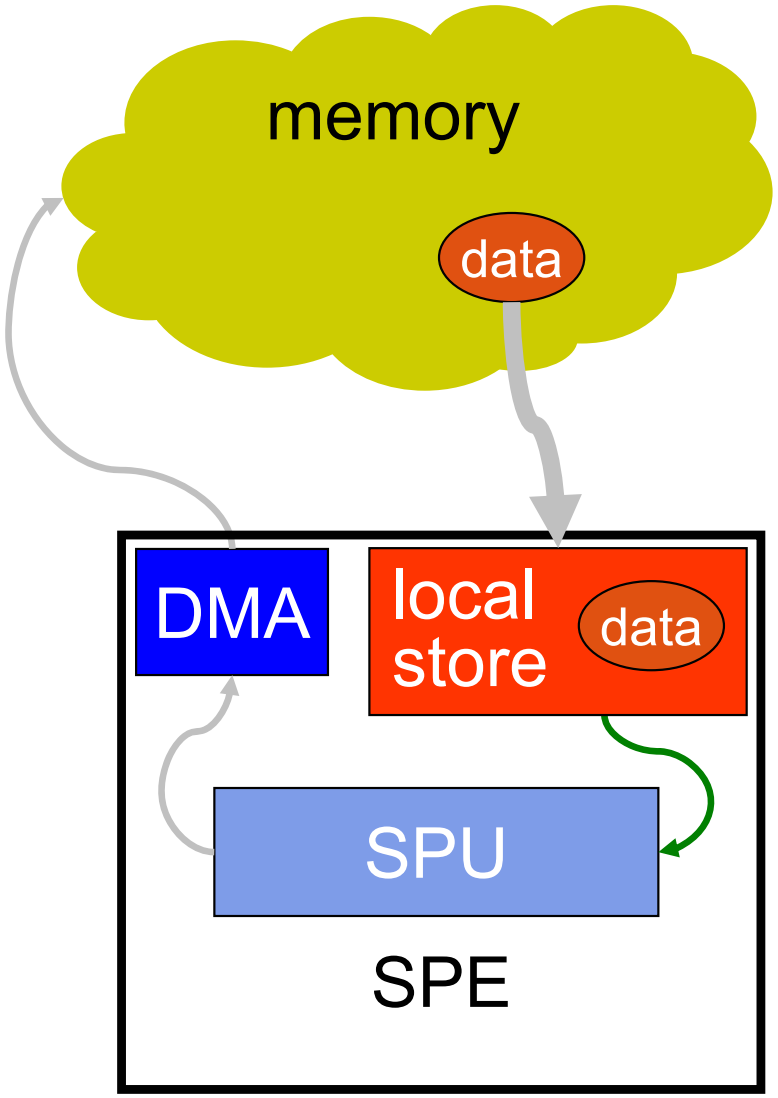
- SPU needs data

1. SPU initiates DMA request for data
2. DMA requests data from memory
3. Data is **copied** to local store



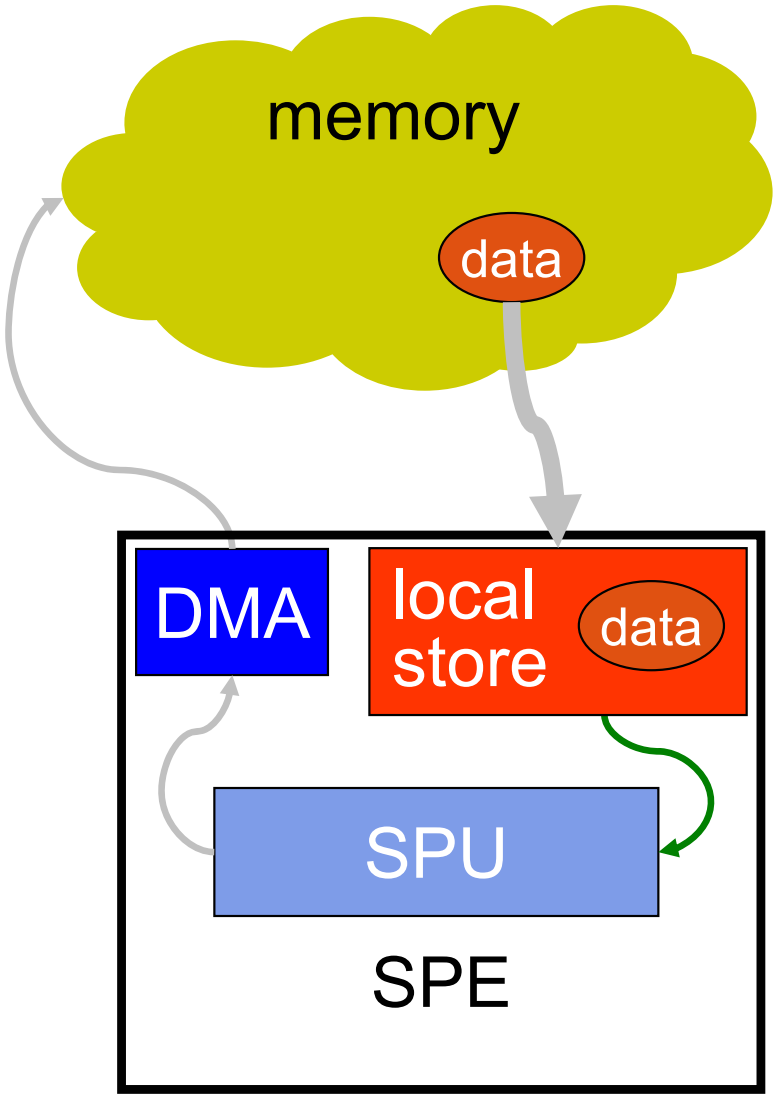
Transferring Data In/Out of SPU Local Store

- SPU needs data
1. SPU initiates DMA request for data
2. DMA requests data from memory
3. Data is copied to local store
4. SPU can access data from local store



Transferring Data In/Out of SPU Local Store

- SPU needs data
1. SPU initiates DMA request for data
2. DMA requests data from memory
3. Data is copied to local store
4. SPU can access data from local store
- SPU operates on data then **copies** data from local store back to memory in a similar process

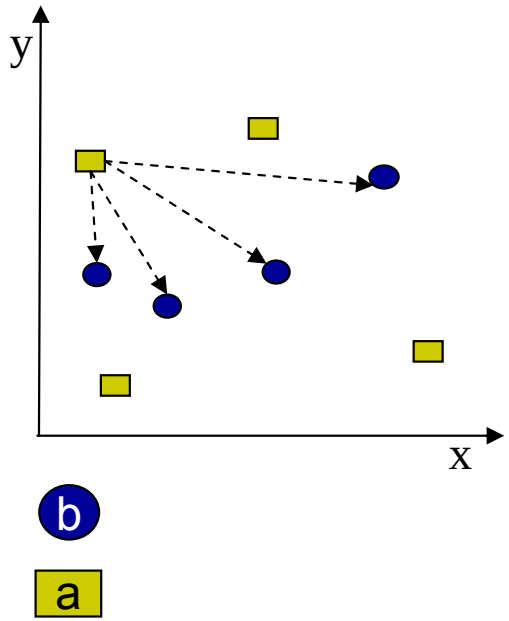


Recitation Plan

- Lab 1 solution
- SPE-SPE communication
- Patterns for mapping computation to SPEs
- Cell-ifying a sample program
  - Hands on programming
  - Optimizations
- Useful resources

Lab 1 Solution

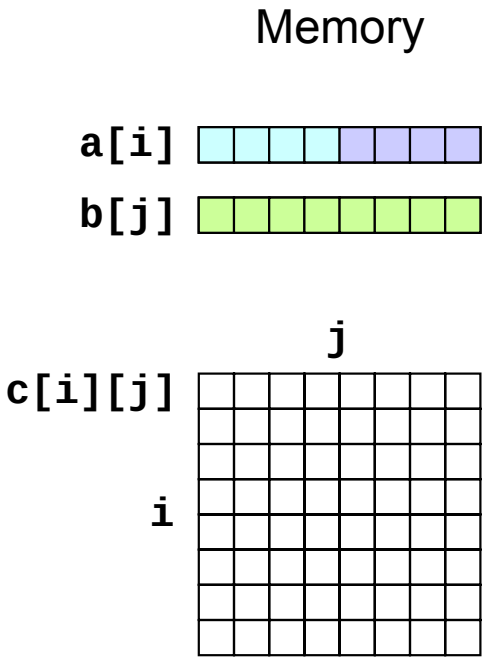
- Calculate distance from each point in a[...] to each point in b[...] and store result in c[...][...]

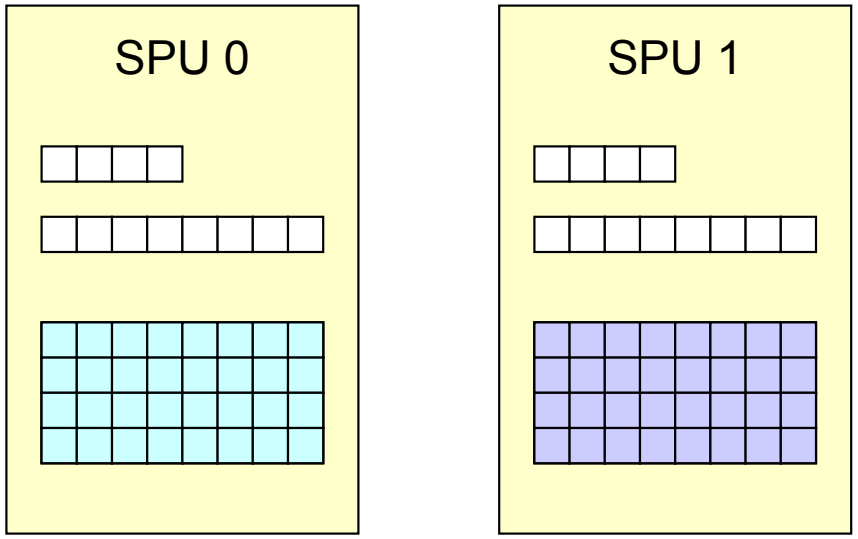



```
for (i = 0; i < NUM_POINTS; i++) {
  for (j = 0; j < NUM_POINTS; j++) {
    c[i][j] = distance(a[i], b[j]);
```

- Divide work among 2 SPEs

Lab 1 Solution





Same code for each SPU

```
typedef struct {
  uintptr32_t a_addr;
  uintptr32_t b_addr;
  uintptr32_t c_addr;
  uint32_t padding;
} CONTROL_BLOCK;
```

Lab 1 Solution

- Can start multiple DMA transfers with the same tag and wait for all of them to complete

```
mfc_get(a, cb.a_addr, HALF_POINTS, 5, 0, 0);
mfc_write_tag_mask(1 << 5);
mfc_read_tag_status_all();

mfc_get(b, cb.b_addr, NUM_POINTS, 5, 0, 0);
mfc_write_tag_mask(1 << 5);
mfc_read_tag_status_all();
```

```
mfc_get(a, cb.a_addr, HALF_POINTS, 5, 0, 0);
mfc_get(b, cb.b_addr, NUM_POINTS, 5, 0, 0);

mfc_write_tag_mask(1 << 5);
mfc_read_tag_status_all();
```

- Can wait for multiple transfers with different tags

```
mfc_write_tag_mask((1 << 5) | (1 << 6));
mfc_read_tag_status_all();
```

- Don’t allocate large buffers on the stack

Recitation Plan

- Lab 1 solution
- SPE-SPE communication
- Patterns for mapping computation to SPEs
- Cell-ifying a sample program
  - Hands on programming
  - Optimizations
- Useful resources

SPE-SPE DMA and Communication

- Streaming data from SPE to SPE
- Example from recitation 1:

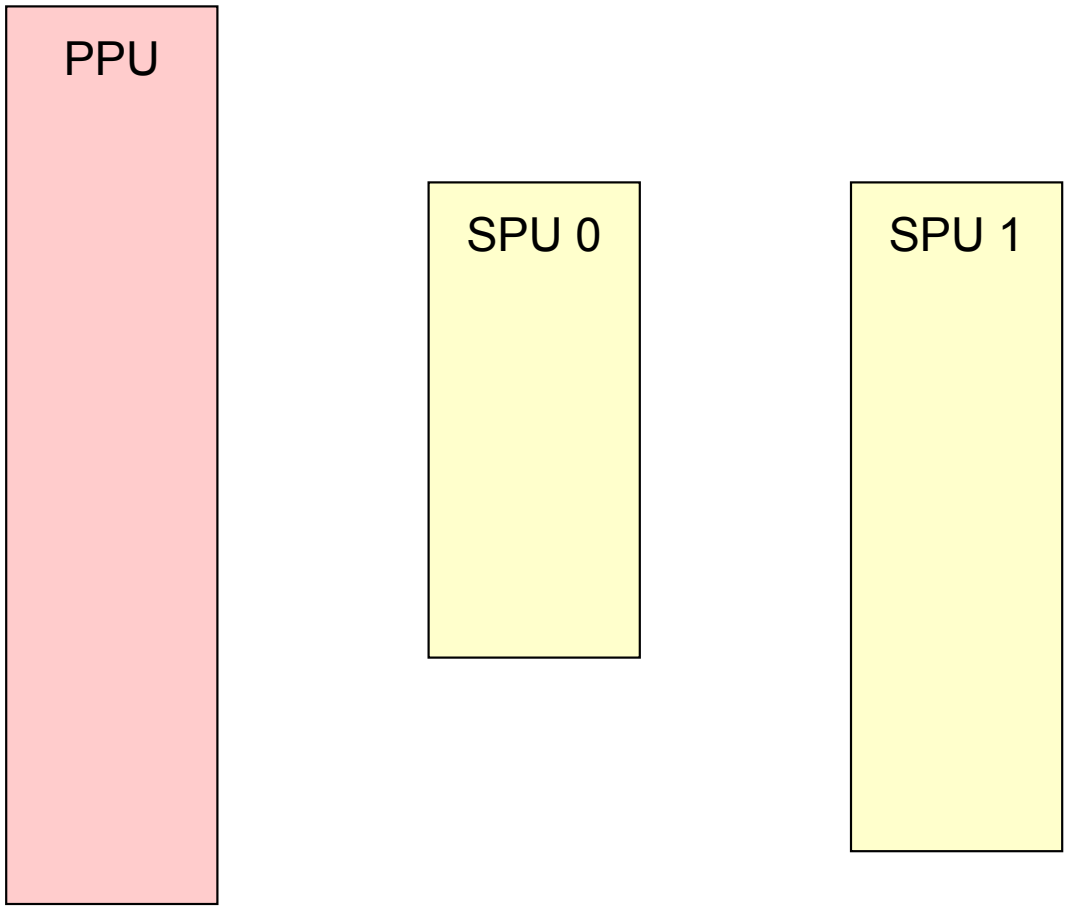
```
for (int i = 0; i < NUM_ELEMENTS; i++) {
  data[i] = data[i] * MUL_FACTOR + ADD_FACTOR;
}
```

- Divide up so one SPE does multiplication, another does addition
- Keep actual data transfer local store to local store
- How to handle synchronization messages?
  - All messages go through PPE
  - SPEs talk to each other directly via mailboxes/signals

SPE-SPE Example

- SPEs that communicate with each other need to know:
  - Addresses of local stores
  - Addresses of MMIO registers
- Only PPU program (via libspe) has access to this information
  - Create SPE threads
  - Gather local store and MMIO register addresses
    - spe_get_ls
    - spe_get_ps_area
  - Send addresses to all SPEs

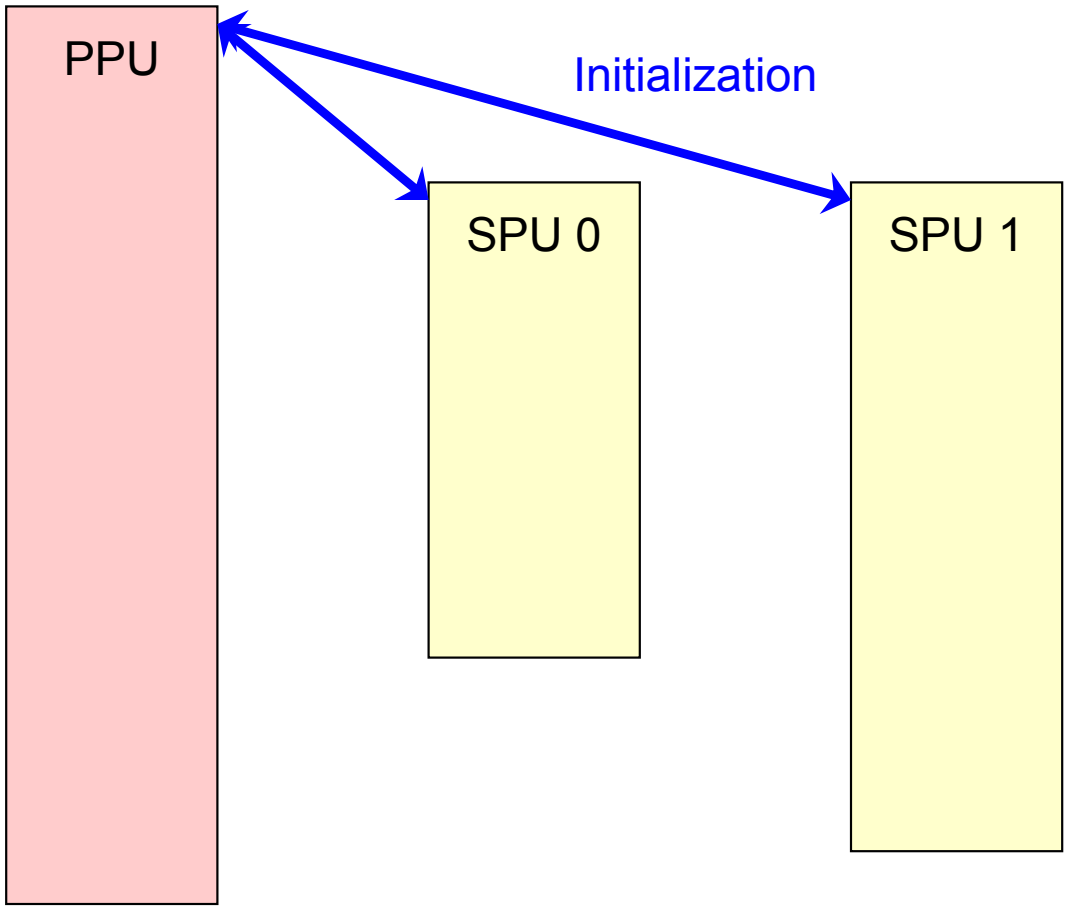
SPE-SPE Example



PPU

```
// Create SPE threads for multiplier and
// adder
id[0] = spe_create_thread(0, &dma_spu0, 0,
                          NULL, ...);
id[1] = spe_create_thread(0, &dma_spu1, 1,
                          NULL, ...);
```

SPE-SPE Example



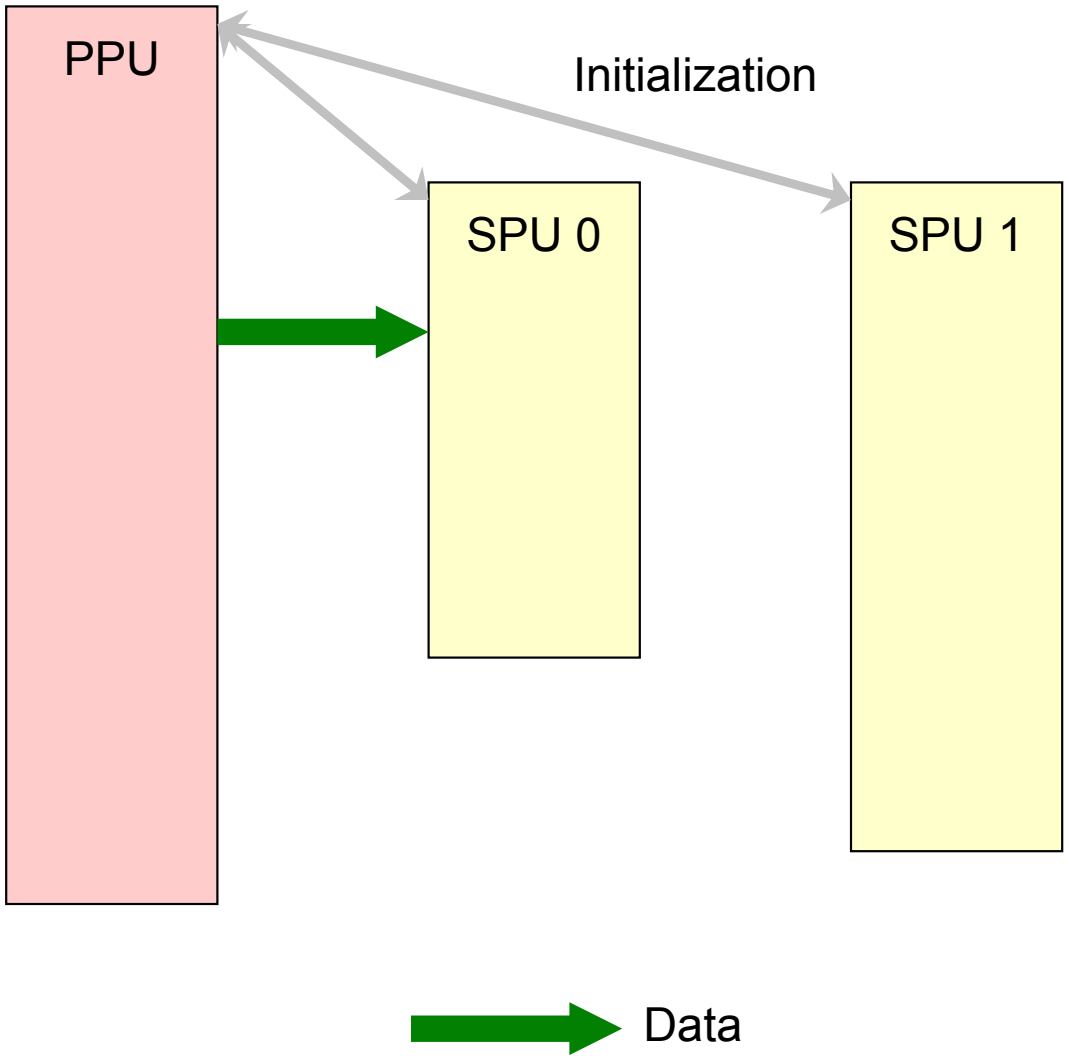
PPU

```
typedef struct {
  uintptr32_t spu_ls[2];
  uintptr32_t spu_control[2];
  ...
} CONTROL_BLOCK;

// Fill in control block
for (int i = 0; i < 2; i++) {
  cb.spu_ls[i] = spe_get_ls(id[i]);
  cb.spu_control[i] =
    spe_get_ps_area(id[i], SPE_CONTROL_AREA);
}
...

// Send control block address to all SPUs
for (int i = 0; i < 2; i++) {
  spe_write_in_mbox(id[i], &cb);
}
```

Both SPUs

```
// Wait for control block address from PPU
cb_addr = spu_read_in_mbox();

// DMA over control block and wait until done
mfc_get(&cb, cb_addr, sizeof(cb), 5, ...);
mfc_write_tag_mask(1 << 5);
mfc_read_tag_status_all();
```

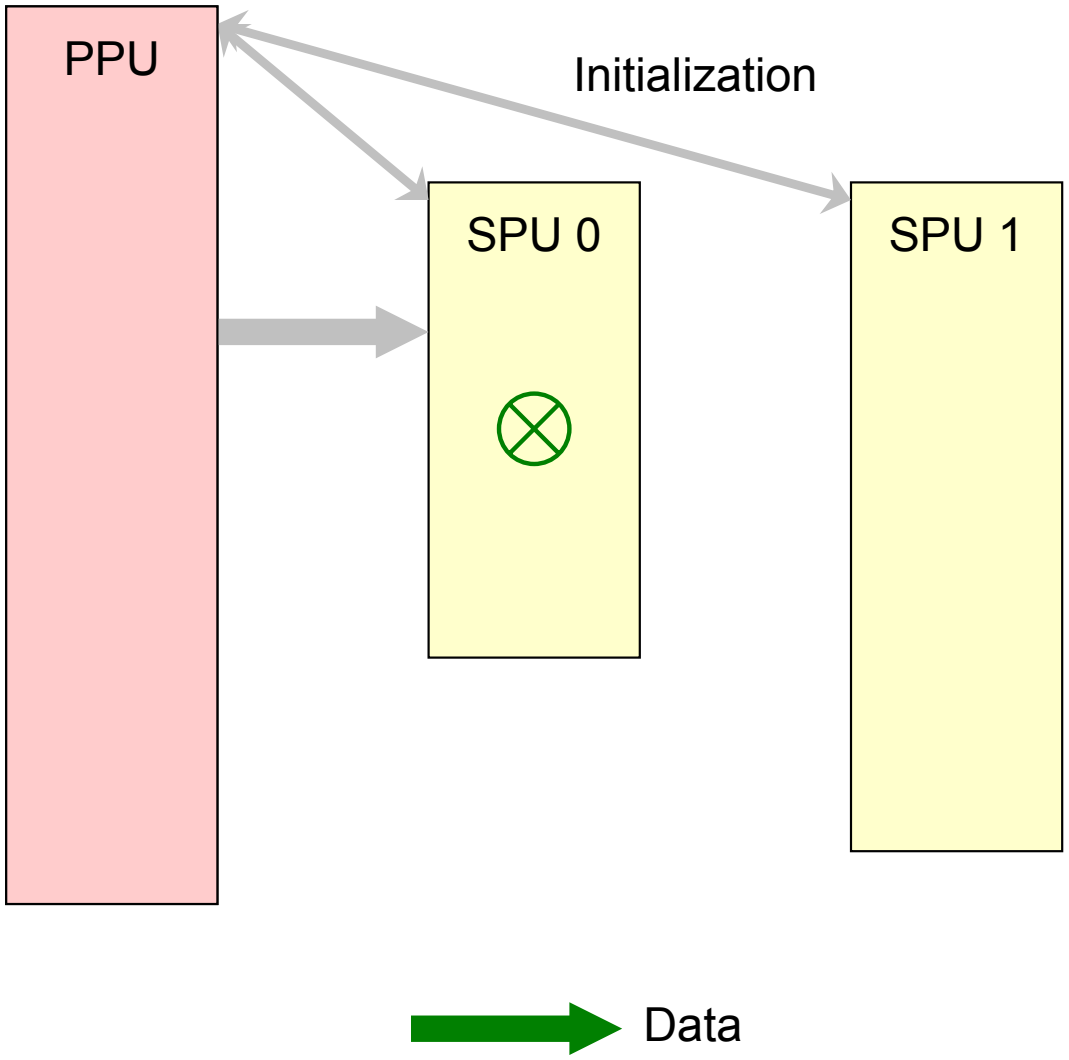
SPE-SPE Example



SPU 0

```
// DMA in data from memory and wait until
// complete
mfc_get(data, cb.data_addr, data_size, ...);
mfc_read_tag_status_all();
```

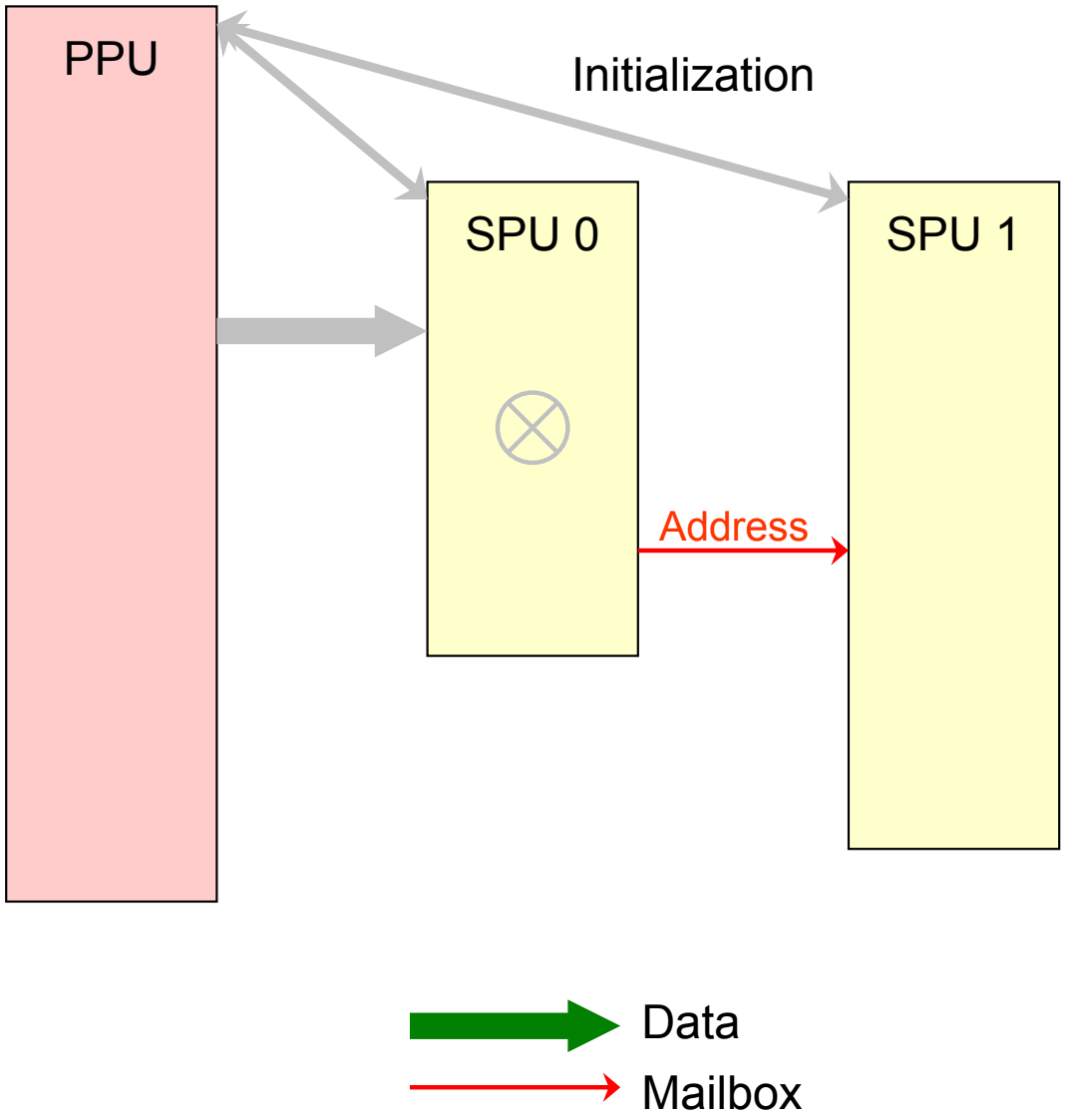
SPE-SPE Example



SPU 0

```
// Process data
...
```

SPE-SPE Example



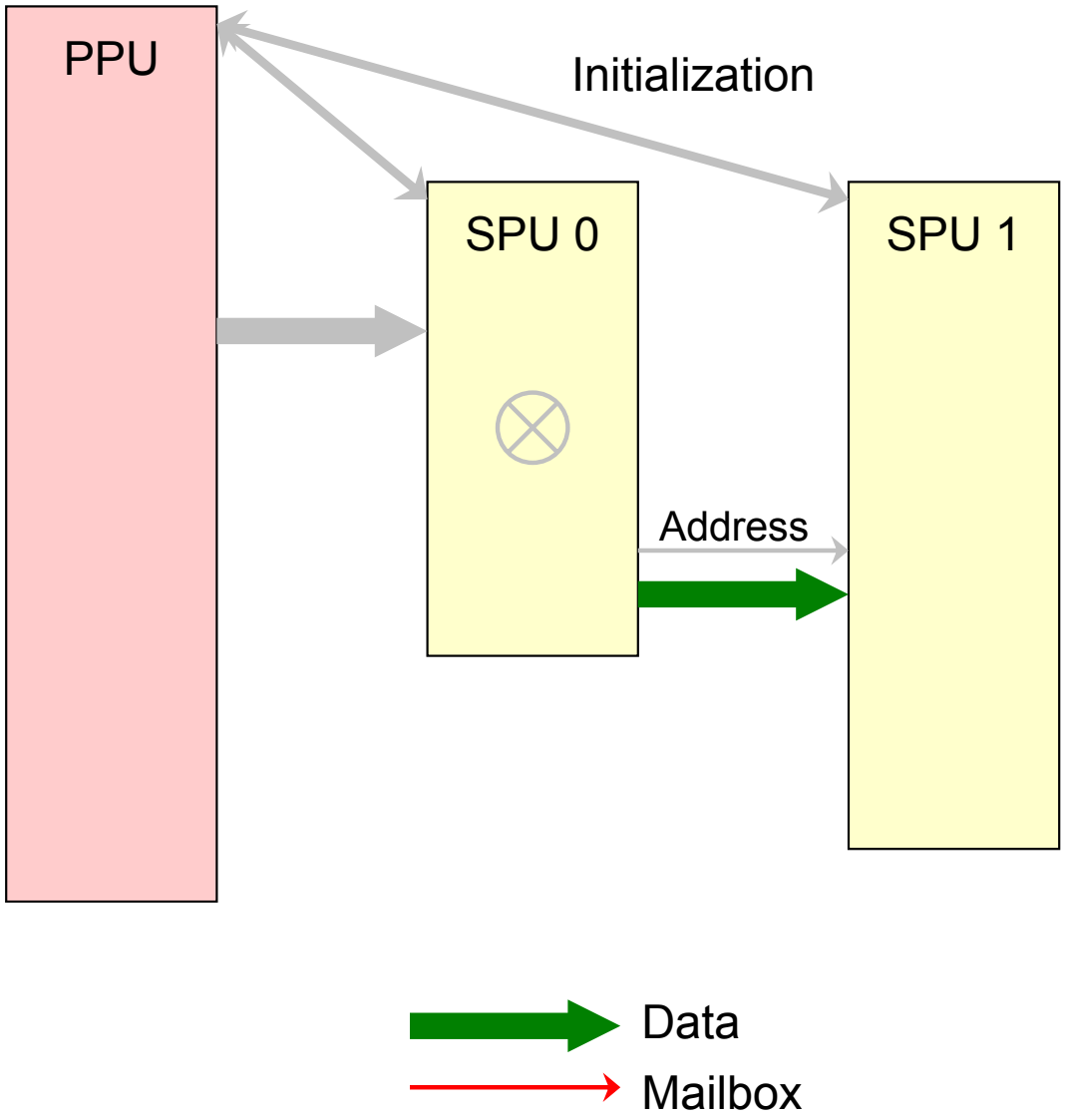
SPU 0

```
// Temporary area used to store values to be
// sent to mailboxes with proper alignment
struct {
  uint32_t padding[3];
  uint32_t value;
} next_mbox __attribute__((aligned(16)));

// Notify SPU 1 that data is ready. Send over
// virtual address so SPU 1 can copy it out.
next_mbox.value = cb.spu_ls[0] + data;
mfc_put(&next_mbox.value,
        cb.spu_control[1] + 12,
        4,
        ...);
```

SPU 1

```
// Wait for mailbox message from SPU 0
// indicating data is ready
data_addr = spu_read_in_mbox();
```

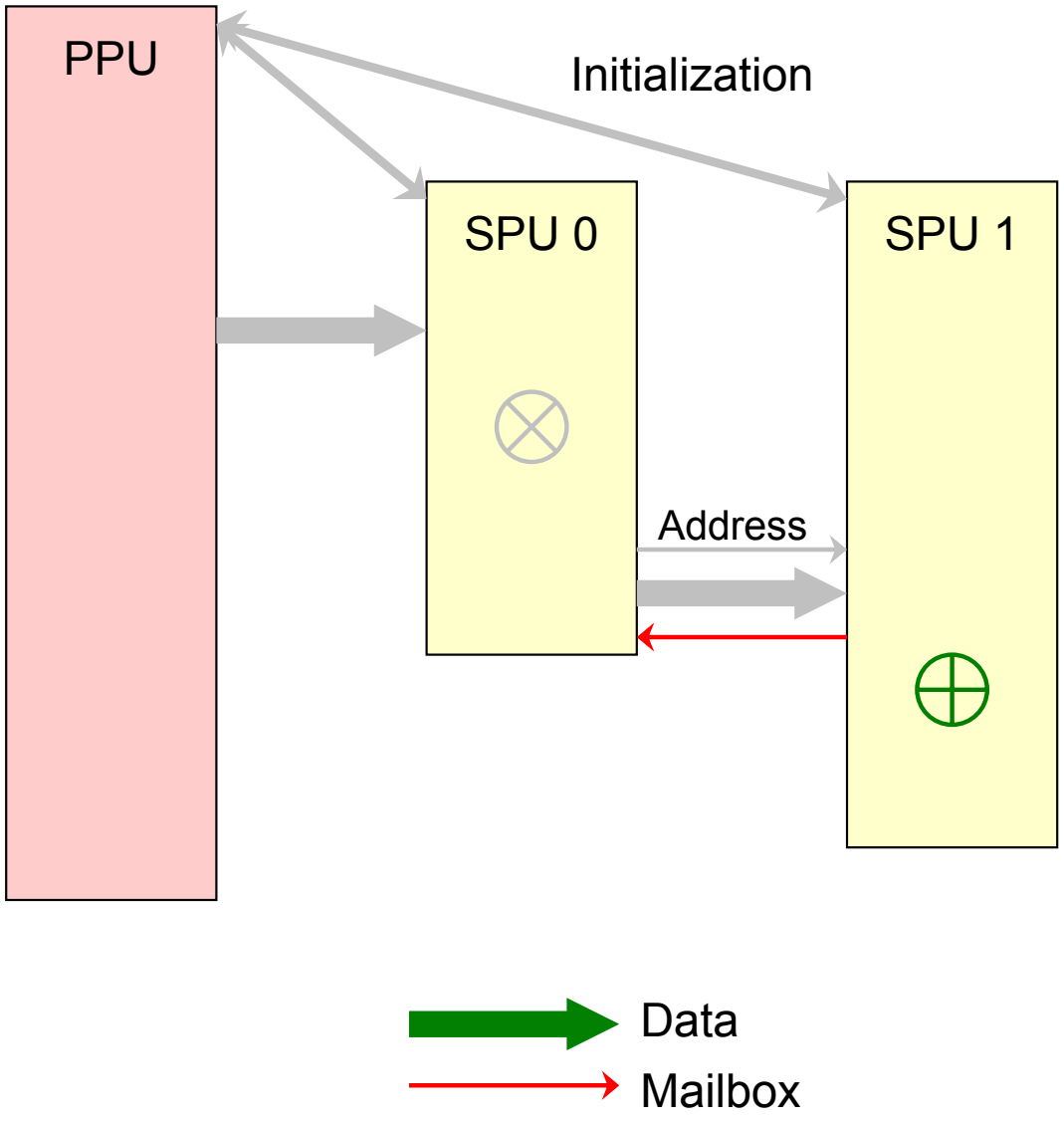
SPE-SPE Example



SPU 1

```
// DMA in data from SPU 0 local store and
// wait until complete
mfc_get(data, data_addr, data_size, ...);
mfc_read_tag_status_all();
```

SPE-SPE Example



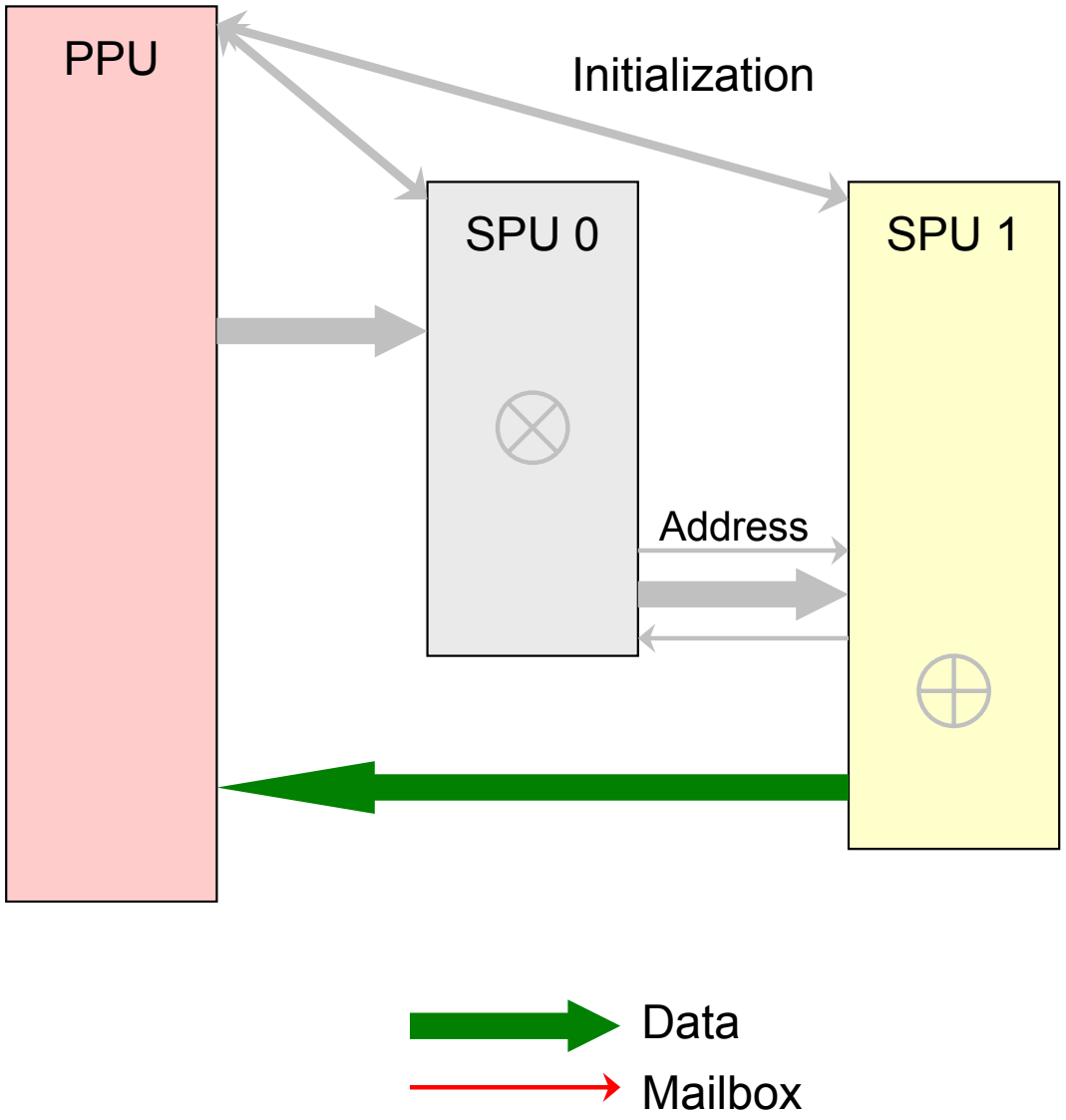
SPU 1

```
// Notify SPU 0 that data has been read
mfc_put(<garbage>,
        cb.spu_control[0] + 12,
        4,
        ...);

// Process data
...
```

SPU 0

```
// Wait for acknowledgement from SPU 1
spu_read_in_mbox();
return 0;
```

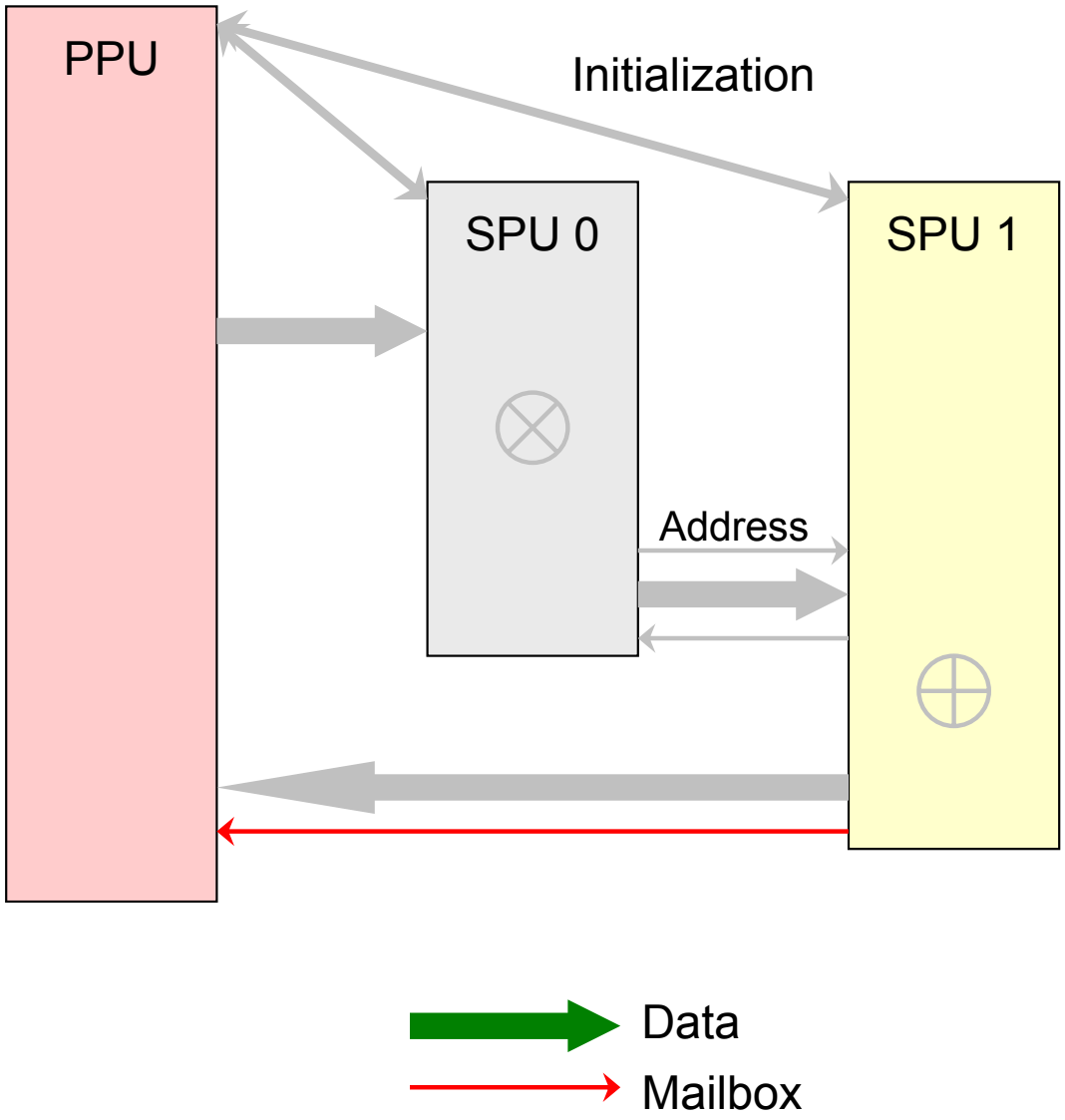
SPE-SPE Example



SPU 1

```
// DMA processed data back to memory and wait
// until complete
mfc_put(data, cb.data_addr, data_size, ...);
mfc_read_tag_status_all();
```

SPE-SPE Example



SPU 1

```
// Notify PPU
spu_write_out_mbox(0);
return 0;
```

PPU

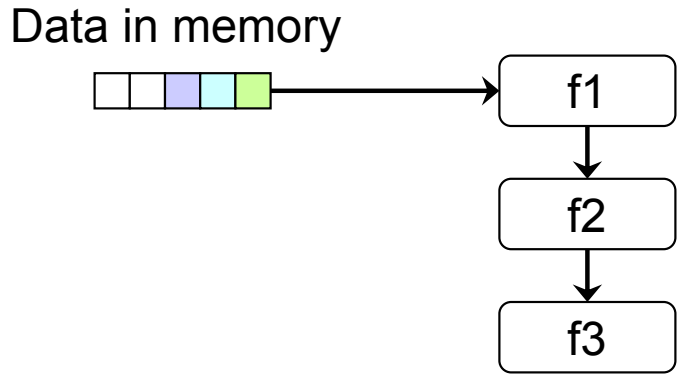
```
// Wait for mailbox message from SPU 1
while (spe_stat_out_mbox(id[1]) == 0);
spe_read_out_mbox(id[1]);
```

Recitation Plan

- Lab 1 solution
- SPE-SPE communication
- Patterns for mapping computation to SPEs
- Cell-ifying a sample program
  - Hands on programming
  - Optimizations
- Useful resources

Mapping Computation to SPEs

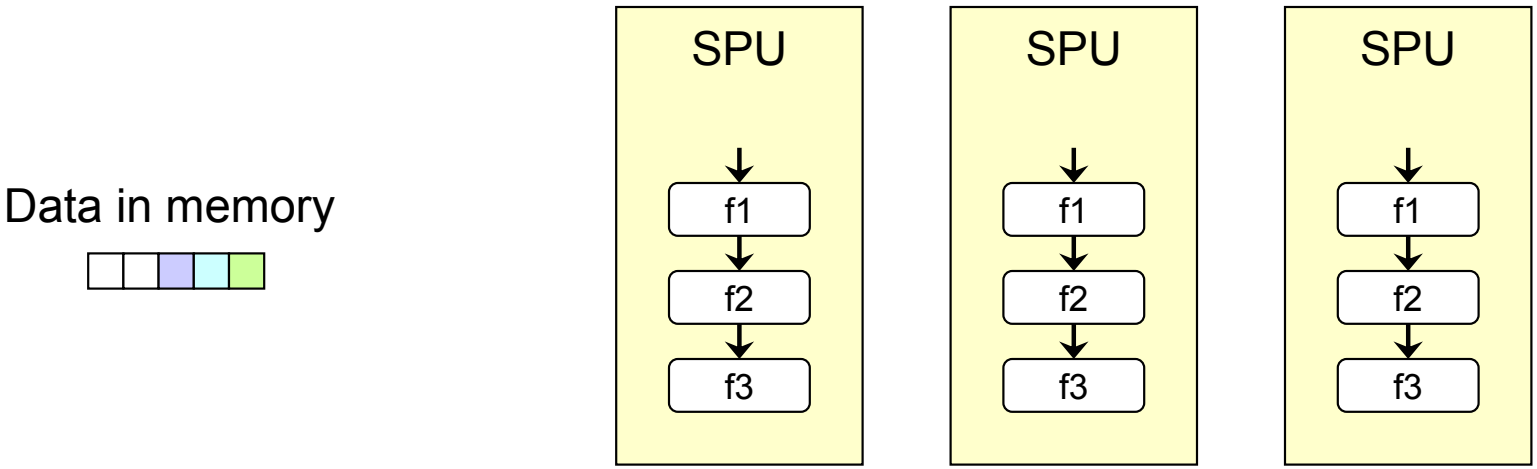
- Original single-threaded program performs many computation stages on data



- How to map to SPEs?

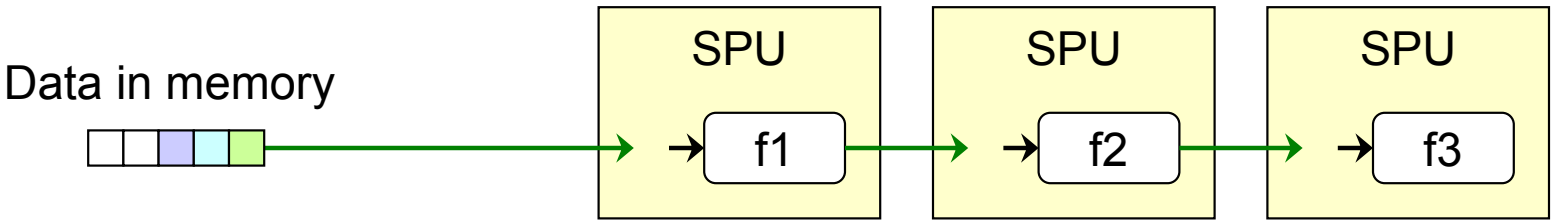
Mapping Computation to SPEs

- Each SPE contains all computation stages
- Split up data and send to different SPEs



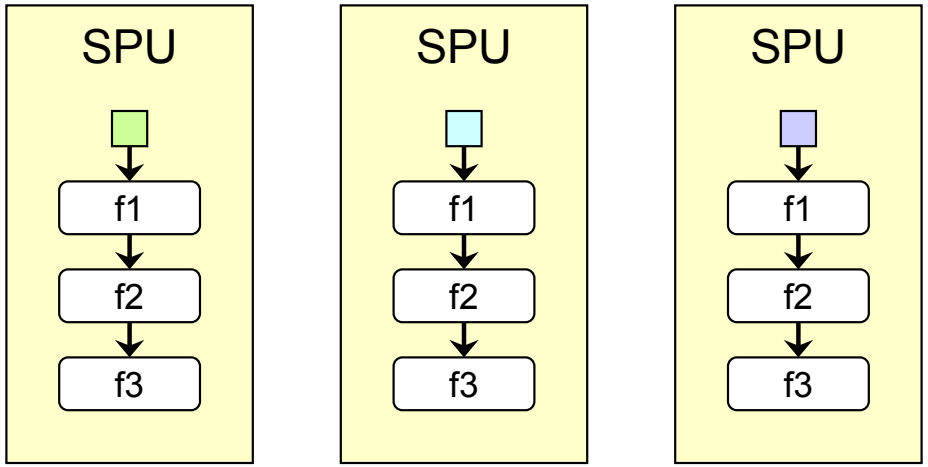
Mapping Computation to SPEs

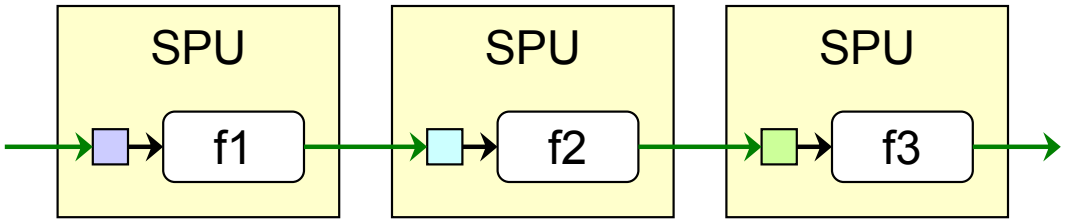
- Map computation stages to different SPEs
- Use DMA to transfer intermediate results from SPE to SPE in pipeline fashion



Mapping Computation to SPEs

- Mixed approaches, other approaches are possible
- Depends on problem
- More detail in future lectures





Recitation Plan

- Lab 1 solution
- SPE-SPE communication
- Patterns for mapping computation to SPEs
- Cell-ifying a sample program
  - Hands on programming
  - Optimizations
- Useful resources

Cell-ifying a Program

- Simple 3D gravitational body simulator
- *n* objects, each with mass, initial position, initial velocity

```
float mass[NUM_BODIES];
VEC3D pos[NUM_BODIES];
VEC3D vel[NUM_BODIES];
```

- Simulate motion using Euler integration

Single-threaded Version

- For each step in simulation
  - Calculate acceleration of all objects
    - For each pair of objects, calculate the force between them and update accelerations accordingly

```
VEC3D acc[NUM_BODIES] = 0;

for (i = 0; i < NUM_BODIES - 1; i++) {
  for (j = i + 1; j < NUM_BODIES; j++) {
    // Displacement vector
    VEC3D d = pos[j] – pos[i];
    // Force
    t = 1 / sqr(length(d));
    // Components of force along displacement
    d = t * (d / length(d));

    acc[i] += d * mass[j];
    acc[j] += -d * mass[i];
  }
}
```

  - Update positions and velocities

```
VEC3D acc[NUM_BODIES] = 0;

for (i = 0; i < NUM_BODIES; i++) {
  pos[i] += vel[i] * TIMESTEP;
  vel[i] += acc[i] * TIMESTEP;
}
```

Exercise (5 minutes)

- Run single-threaded version
  - Get the recitation 2 tarball
    - wget http://cag.csail.mit.edu/ps3/recitation2/rec2.tar.gz
    - tar zxf rec2.tar.gz
  - Build the program
    - cd rec2/sim
    - make
  - Run the program
    - ./sim
  - Amount of time taken to perform each simulation step is printed
    - n = 3072 objects
  - Stop program by pressing Ctrl+C

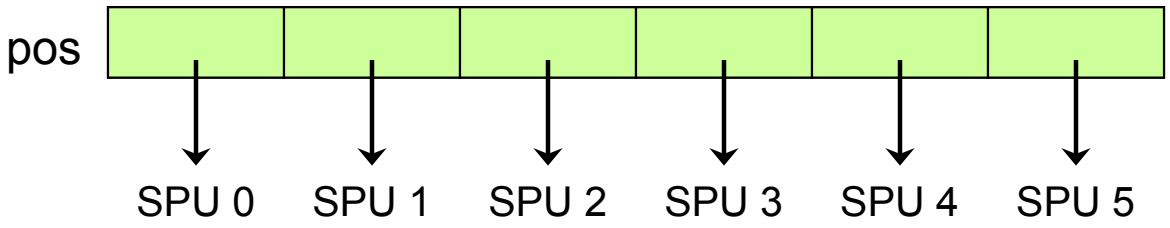
Single-threaded Version

- Slow!
- Speed it up by using SPEs
  - Linux on PS3 has 6 physical SPEs available

Ideas?

Cell-ified Version

- Divide objects into 6 sections (n = 3072 = 6 * 512)



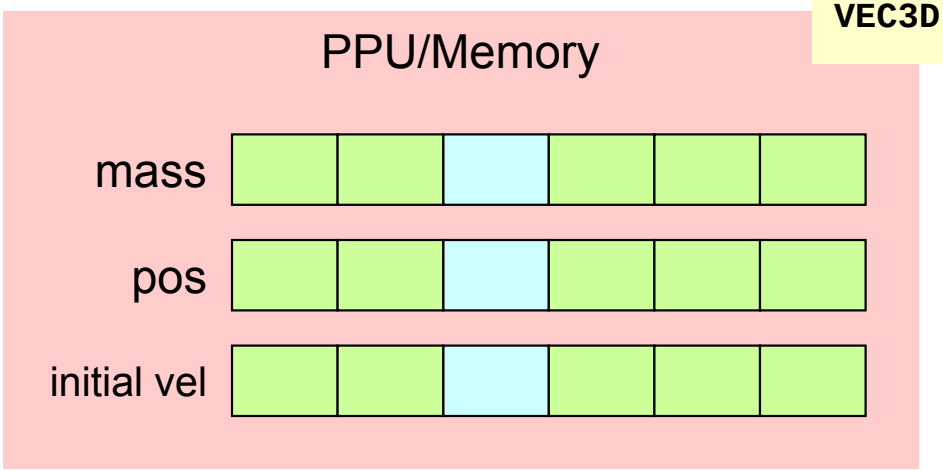
- Each SPU is responsible for calculating the motion of one section of objects
  - SPU still needs to know mass, position of all objects to compute accelerations
  - SPU only needs to know and update velocity of the objects it is responsible for
- Everything fits in local store
  - Positions for 3072 objects take up 36 KB

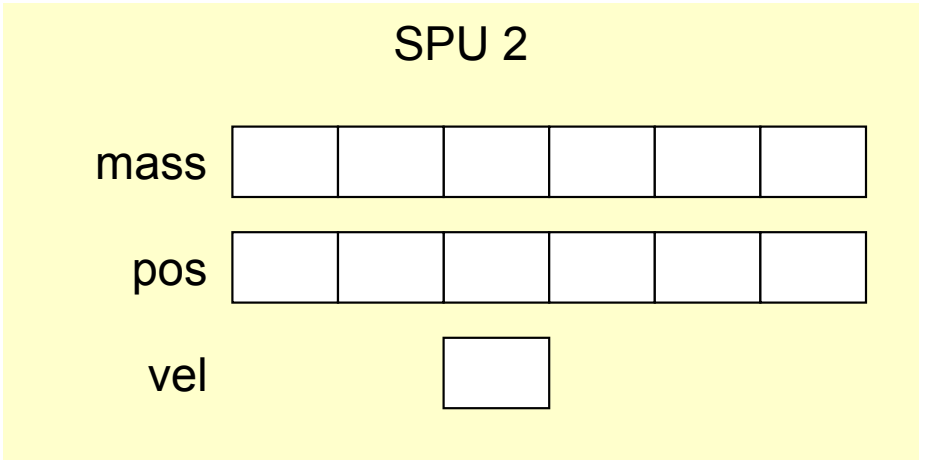
Cell-ified Version

- Initialization
  - PPU tells SPU which section of objects it is responsible for

```
// Pass id in envp
id = envp;
own_mass = mass[id];
own_pos = pos[id];
```

```
// Index [i] stores mass/position of objects SPU i
// is responsible for
float mass[6][SPU_BODIES];
VEC3D pos[6][SPU_BODIES];

// The section of objects this SPU is responsible for
int id;
// Pointer to pos[id]
VEC3D *own_pos;
// Velocity for this SPU's objects
VEC3D own_vel[SPU_BODIES];
```





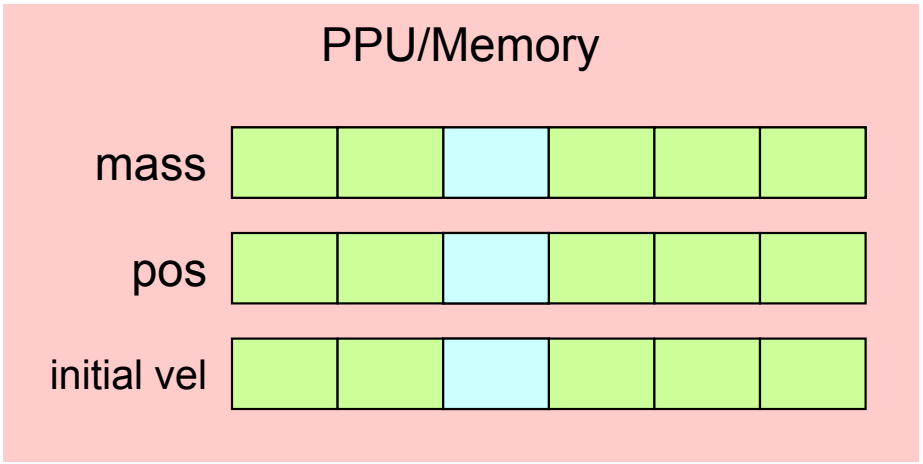
Cell-ified Version

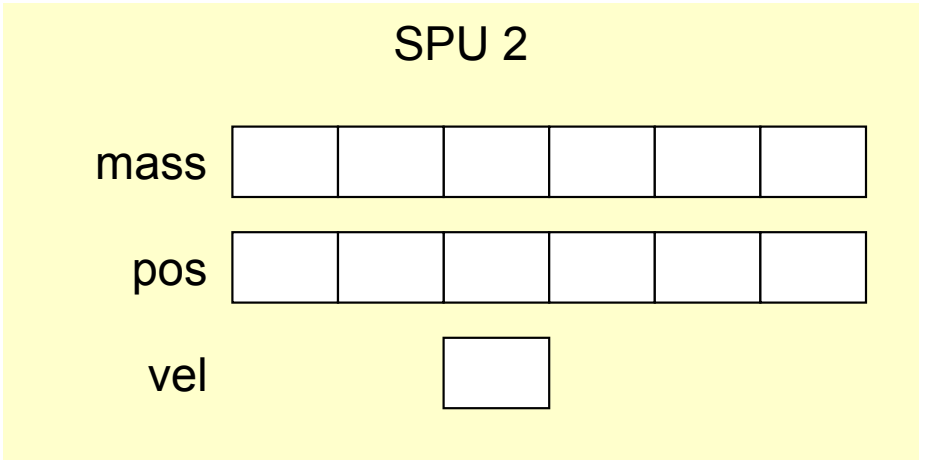
- SPU copies in mass of all objects

```
mfc_get(mass, cb.mass_addr, sizeof(mass), ...);
```

- SPU copies in initial position, velocity of its objects

```
mfc_get(own_pos, cb.pos_addr + id * sizeof(pos[0]), sizeof(pos[0]), ...);
mfc_get(own_vel, cb.vel_addr + id * sizeof(own_vel), sizeof(own_vel), ...);
```

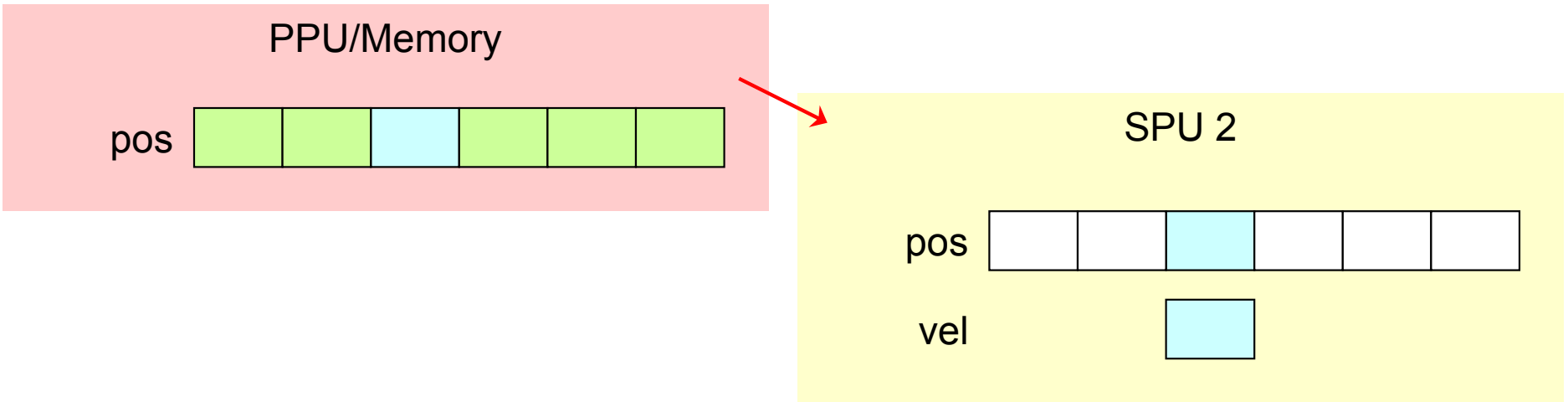




Cell-ified Version

- Simulation step
  - PPU sends message telling SPU to simulate one step

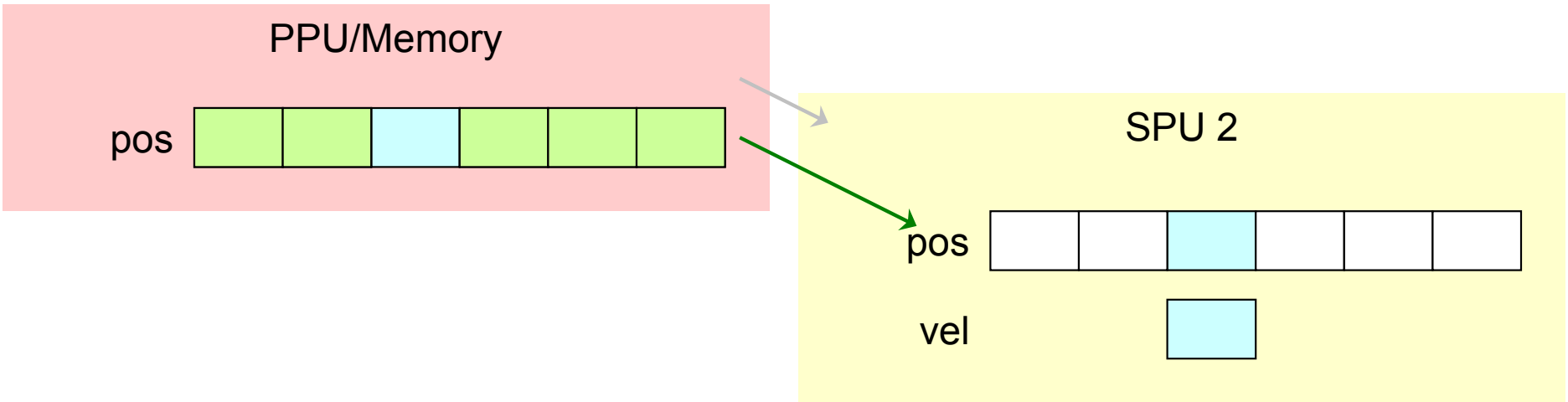
```
spu_read_in_mbox();
```



Cell-ified Version

- SPU copies in updated positions of other objects

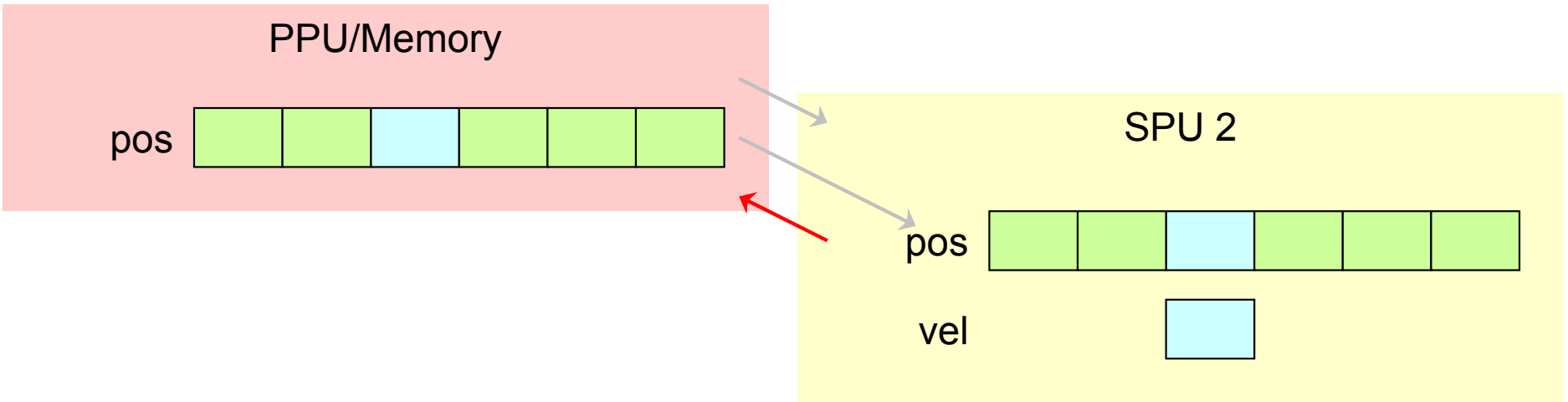
```
if (id != 0) {
  mfc_get(pos, cb.pos_addr + id * sizeof(pos[0]), id * sizeof(pos[0]), ...);
};
if (id != 5) {
  mfc_get(pos[id + 1], cb.pos_addr + (id + 1) * sizeof(pos[0]),
          (5 – id) * sizeof(pos[0]), ...);
}
```



Cell-ified Version

- SPU sends message to PPU indicating it has finished copying positions
  - PPU waits for this message before it can tell other SPUs to write back positions at end of simulation step

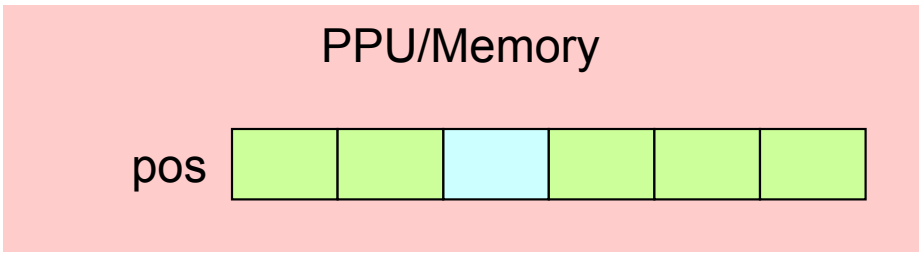
```
spu_write_out_mbox(0);
```

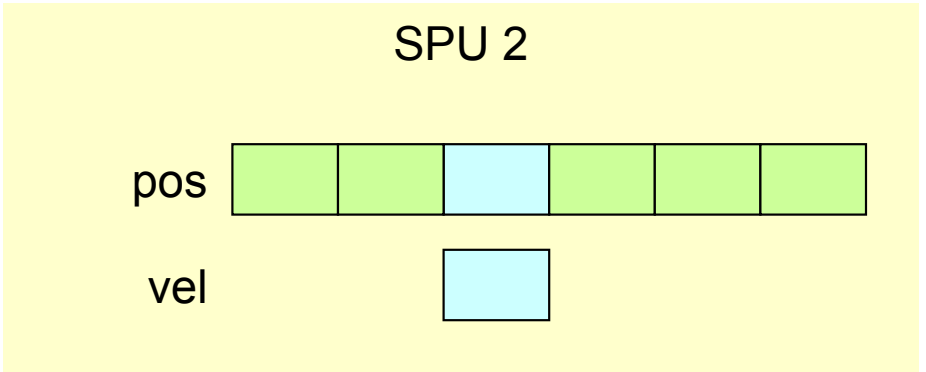


Cell-ified Version

- SPU calculates acceleration and updates position and velocity of its objects

```
// Process interactions between this SPU's objects
process_own();
// Process interactions with other objects
for (int i = 0; i < 6; i++) {
  if (i != id) {
    process_other(pos[i], mass[i]);
  }
}
```

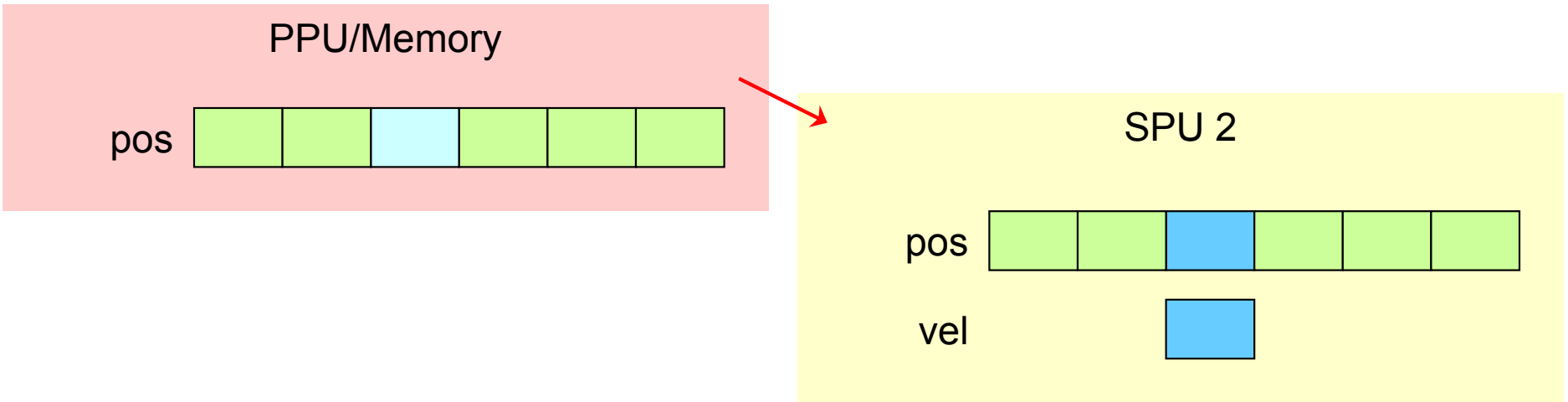




Cell-ified Version

- SPU waits for message from PPU indicating it can write back updated positions

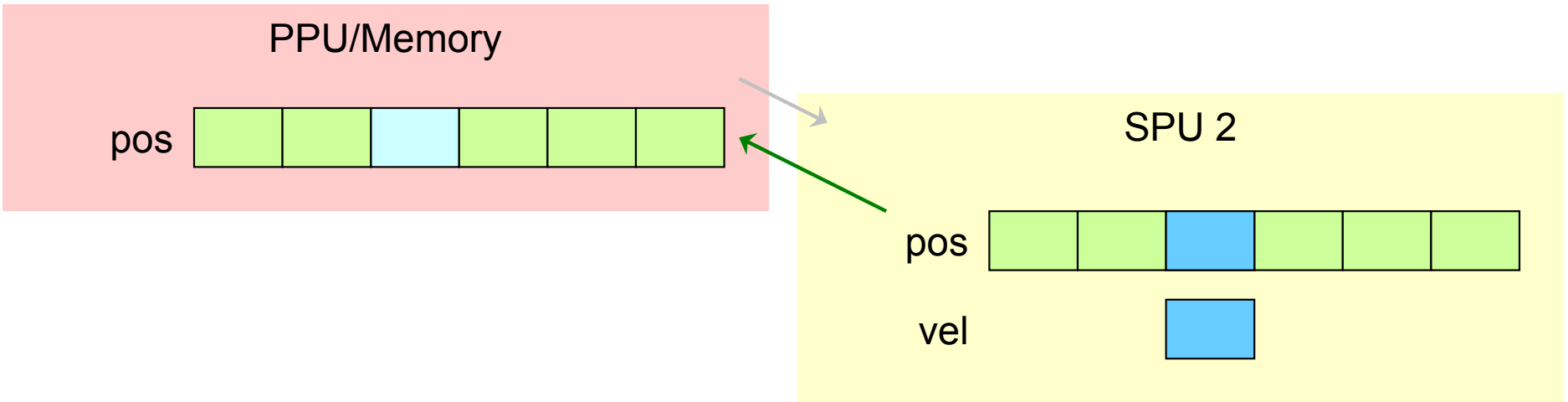
```
spu_read_in_mbox();
```



Cell-ified Version

- SPU writes back updated positions to PPU

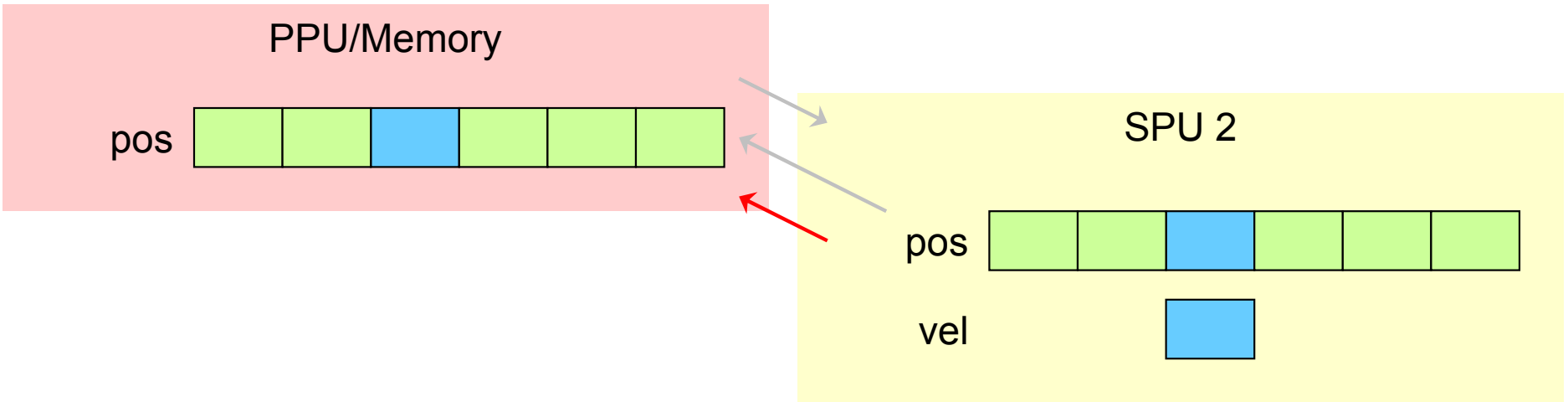
```
mfc_put(own_pos, cb.pos_addr + id * sizeof(pos[0]), sizeof(pos[0]), ...);
```



Cell-ified Version

- SPU sends message to PPU indicating it is done simulation step

```
spu_write_out_mbox(0);
```



Cell-ified Version

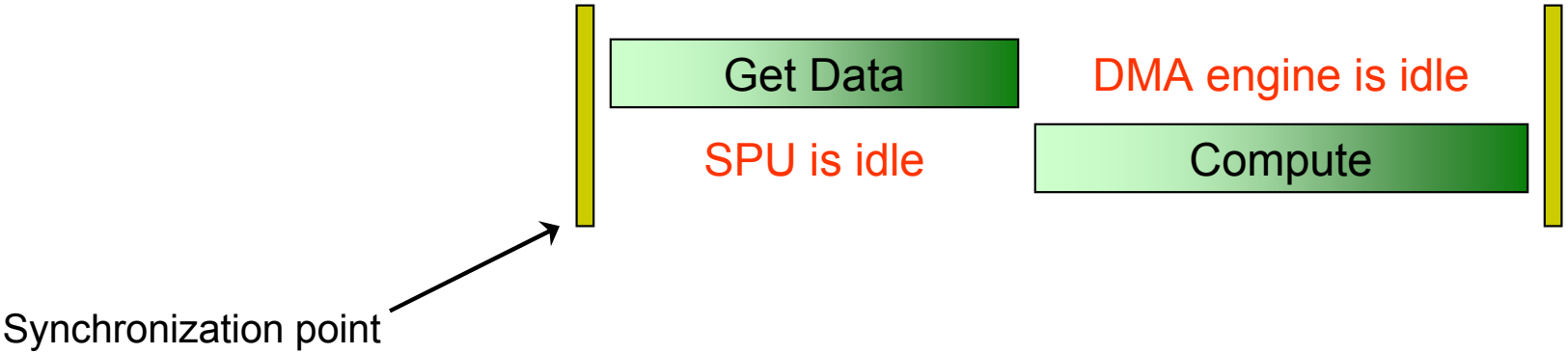
- Acceleration calculation
  - In single-threaded version, we calculate distance between each pair of objects once
  - In Cell-ified version, if 2 SPUs are responsible for objects *i* and *j*, each SPU will calculate the distance separately
- With 6 SPEs, what speedup can we expect?

Exercise (15 minutes)

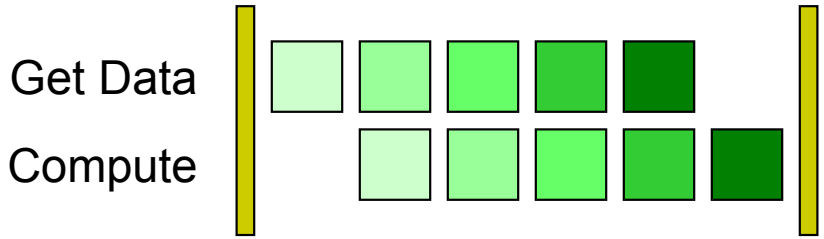
- Read over the code for the Cell-ified version
  - rec2/sim2/spu/sim_spu.c
  - rec2/sim2/sim.c
- Build and run the program
  - cd rec2/sim2
  - make
  - ./sim
- What happened?
  - DMA size limit exceeded
- Fix it
  - Build and run it again

Overlapping DMA and Computation

- We are currently doing this:

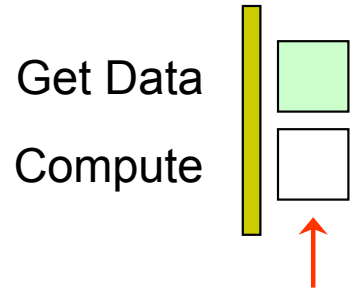


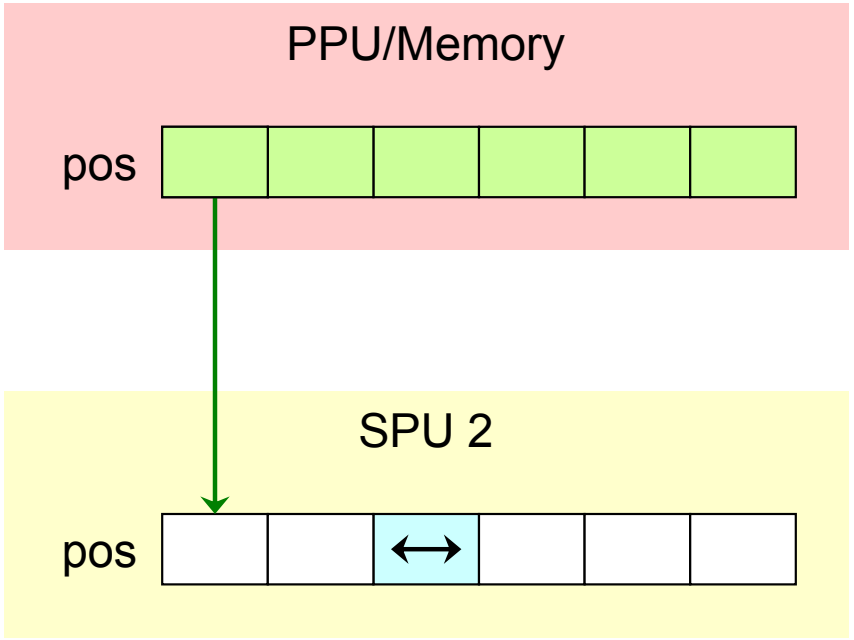
- We can use pipelining to achieve communication-computation concurrency
  - Start DMA for next piece of data while processing current piece



Overlapping DMA and Computation

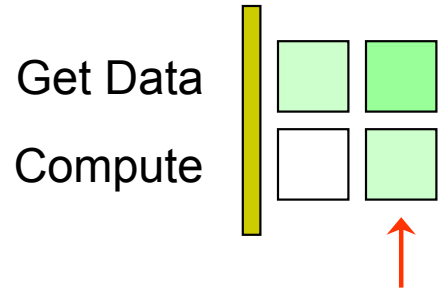
```
// pos[i] stores positions of objects SPU i is
// responsible for
VEC3D pos[6][SPU_BODIES];


// Start transfer for first section of positions
i = 0;
tag = 0;
mfc_get(pos[i],
        cb.pos_addr + i * sizeof(pos[0]),
        sizeof(pos[0]),
        tag,
        ...);
tag ^= 1;

// Process interactions between objects this SPU
// is responsible for
process_own();
```

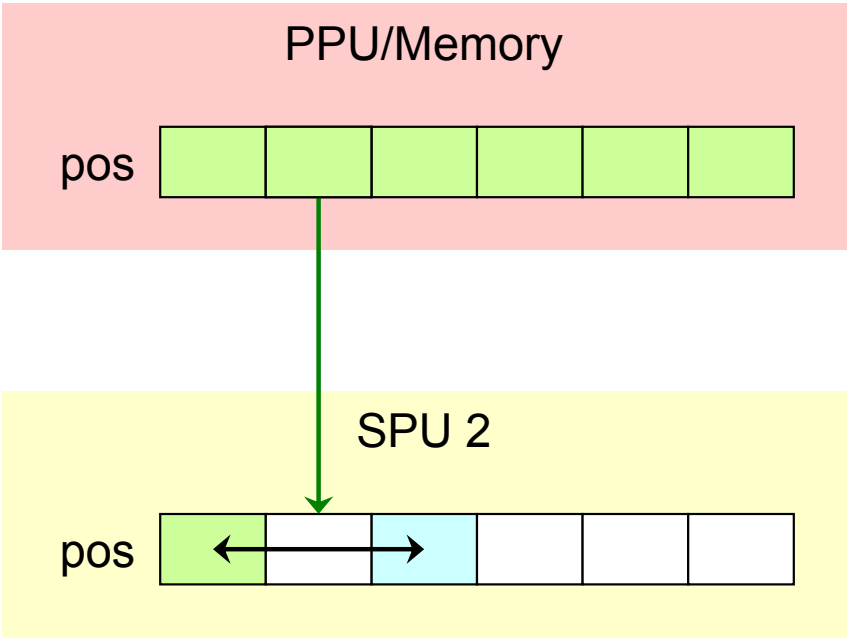




Overlapping DMA and Computation

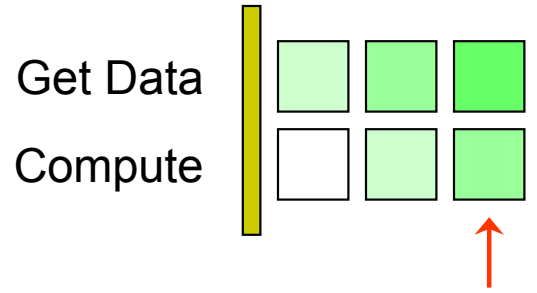
```
while (!done) {
  // Start transfer for next section of positions
  mfc_get(pos[next_i],
          cb.pos_addr + next_i * sizeof(pos[0]),
          sizeof(pos[0]),
          tag,
          ...);

  // Wait for current section of positions to
  // finish transferring
  tag ^= 1;
  mfc_write_tag_mask(1 << tag);
  mfc_read_tag_status_all();

  // Process interactions
  process_other(pos[i], mass[i]);

  i = next_i;
}
```

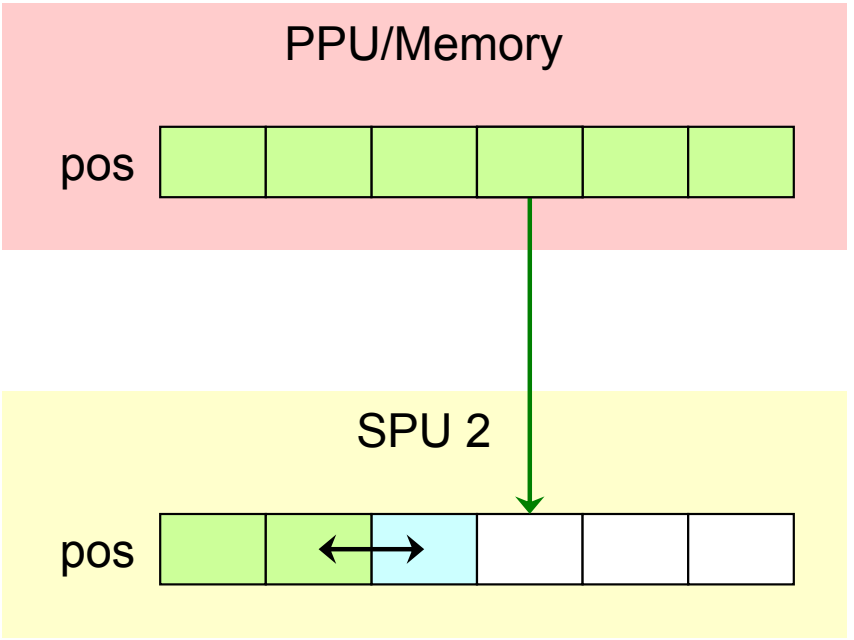




Overlapping DMA and Computation

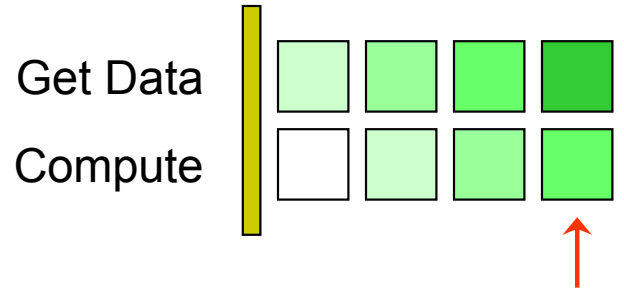
```
while (!done) {
  // Start transfer for next section of positions
  mfc_get(pos[next_i],
          cb.pos_addr + next_i * sizeof(pos[0]),
          sizeof(pos[0]),
          tag,
          ...);

  // Wait for current section of positions to
  // finish transferring
  tag ^= 1;
  mfc_write_tag_mask(1 << tag);
  mfc_read_tag_status_all();

  // Process interactions
  process_other(pos[i], mass[i]);

  i = next_i;
}
```

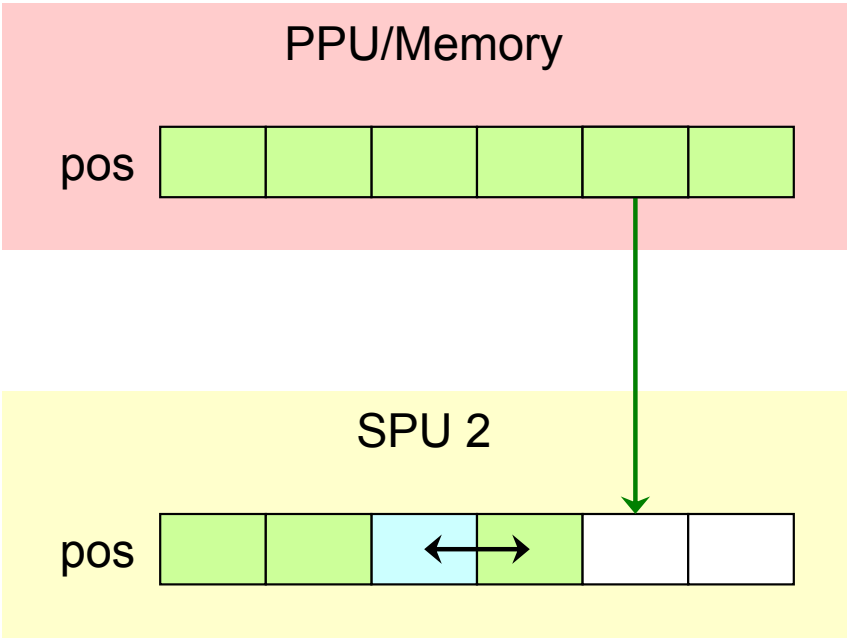




Overlapping DMA and Computation

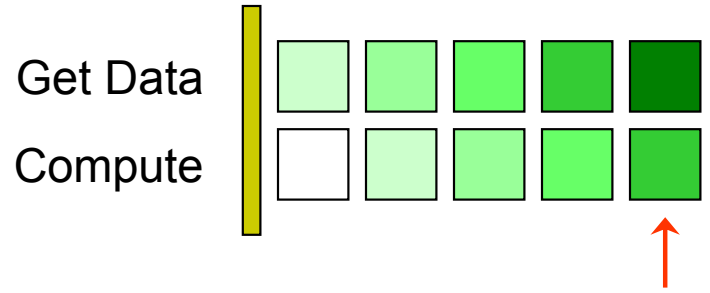
```
while (!done) {
  // Start transfer for next section of positions
  mfc_get(pos[next_i],
          cb.pos_addr + next_i * sizeof(pos[0]),
          sizeof(pos[0]),
          tag,
          ...);

  // Wait for current section of positions to
  // finish transferring
  tag ^= 1;
  mfc_write_tag_mask(1 << tag);
  mfc_read_tag_status_all();

  // Process interactions
  process_other(pos[i], mass[i]);

  i = next_i;
}
```

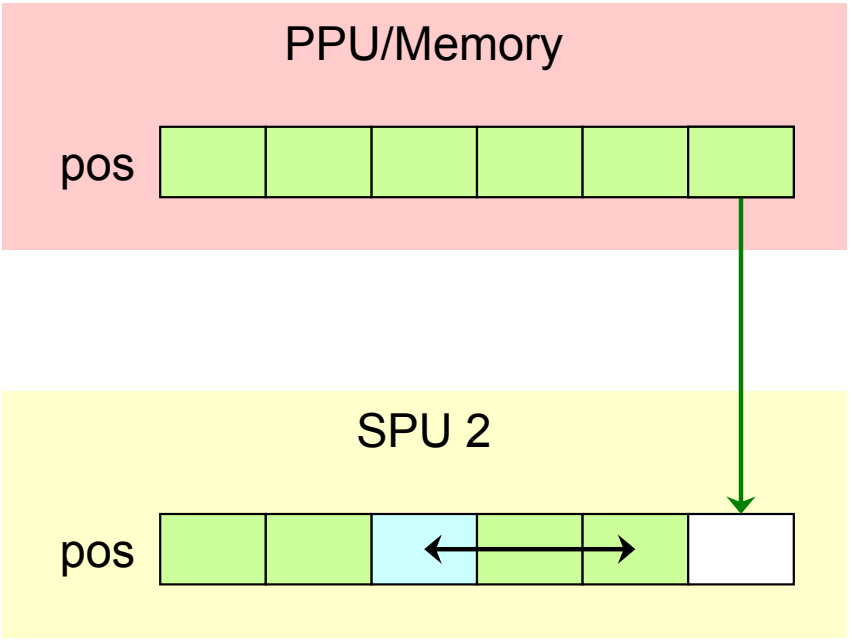




Overlapping DMA and Computation

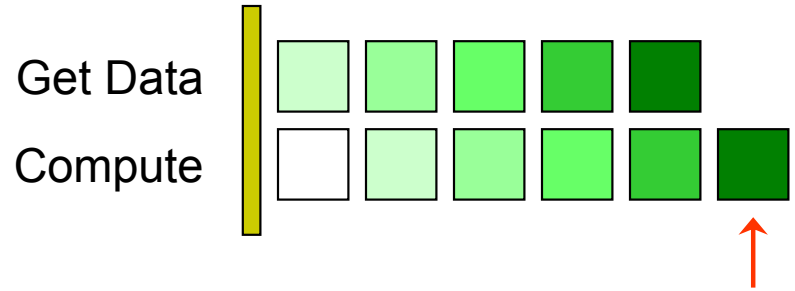
```
while (!done) {
  // Start transfer for next section of positions
  mfc_get(pos[next_i],
          cb.pos_addr + next_i * sizeof(pos[0]),
          sizeof(pos[0]),
          tag,
          ...);

  // Wait for current section of positions to
  // finish transferring
  tag ^= 1;
  mfc_write_tag_mask(1 << tag);
  mfc_read_tag_status_all();

  // Process interactions
  process_other(pos[i], mass[i]);

  i = next_i;
}
```

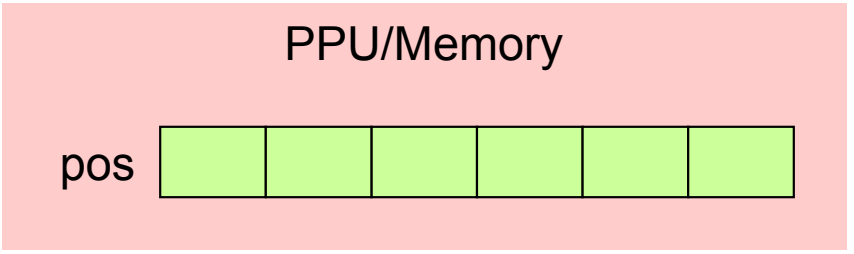


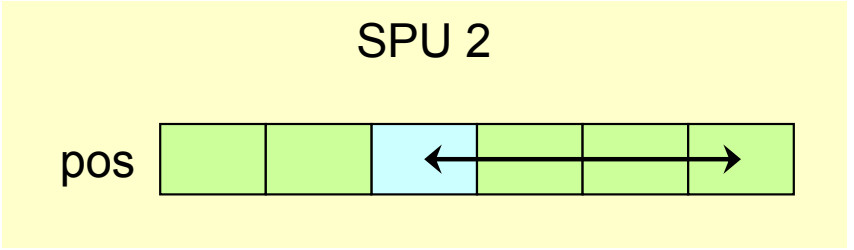


Overlapping DMA and Computation

```
// Wait for last section of positions to finish
// transferring
tag ^= 1;
mfc_write_tag_mask(1 << tag);
mfc_read_tag_status_all();

// Notify PPU that positions have been read
spu_write_out_mbox(0);

// Process interactions
process_other(pos[i], mass[i]);
```

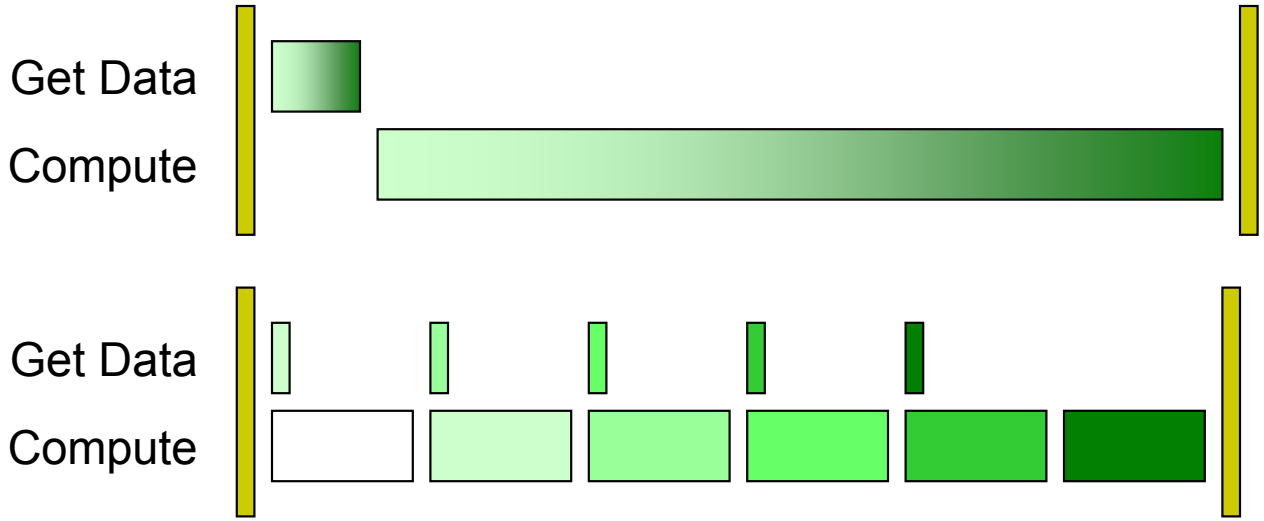






Overlapping DMA and Computation

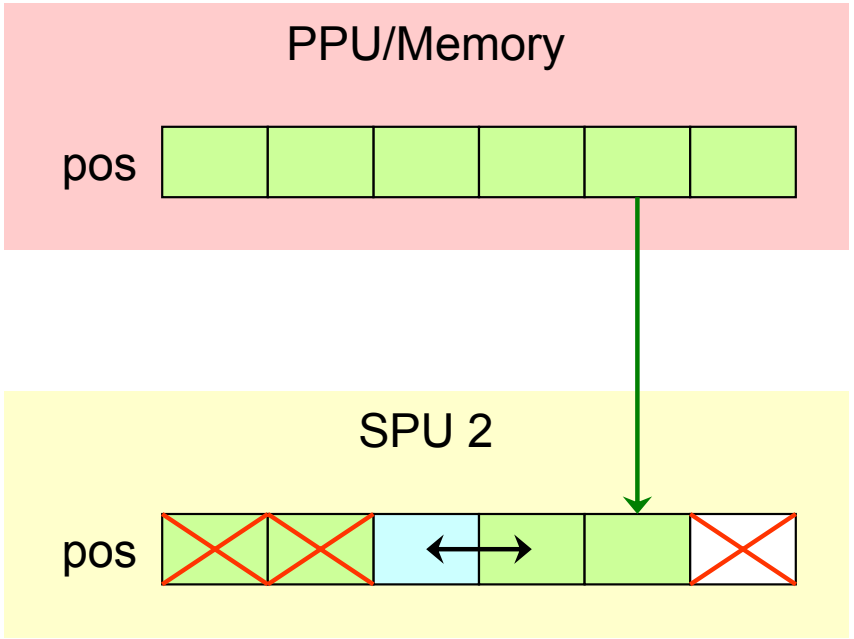
- Performance improvement?
- For this program, computation >> communication
  - $O(n^2)$ computation, $O(n)$ communication



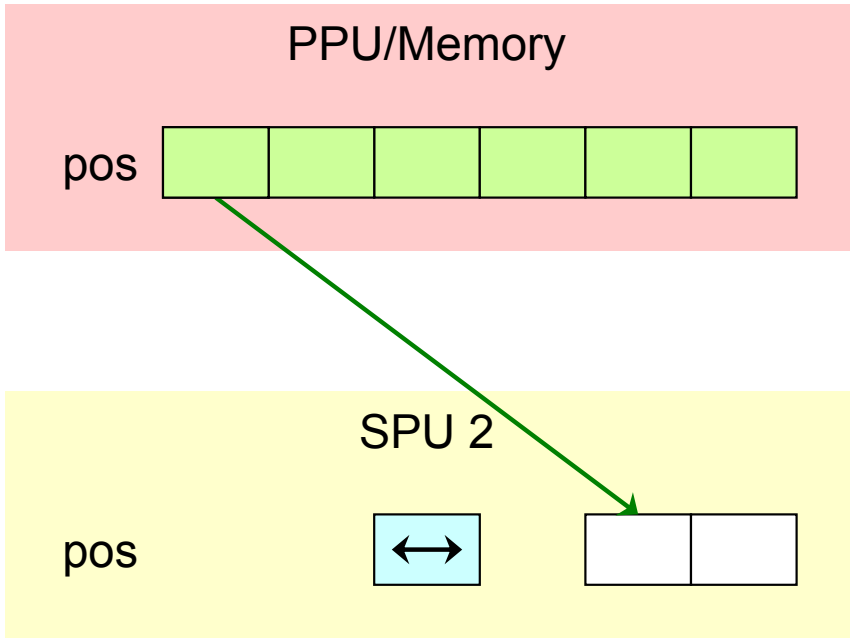
- Pipelining can improve performance by a lot, or not by much
  - Depends on specifics of program
  - Can avoid optimizing parts that don't greatly affect performance

Double-buffering

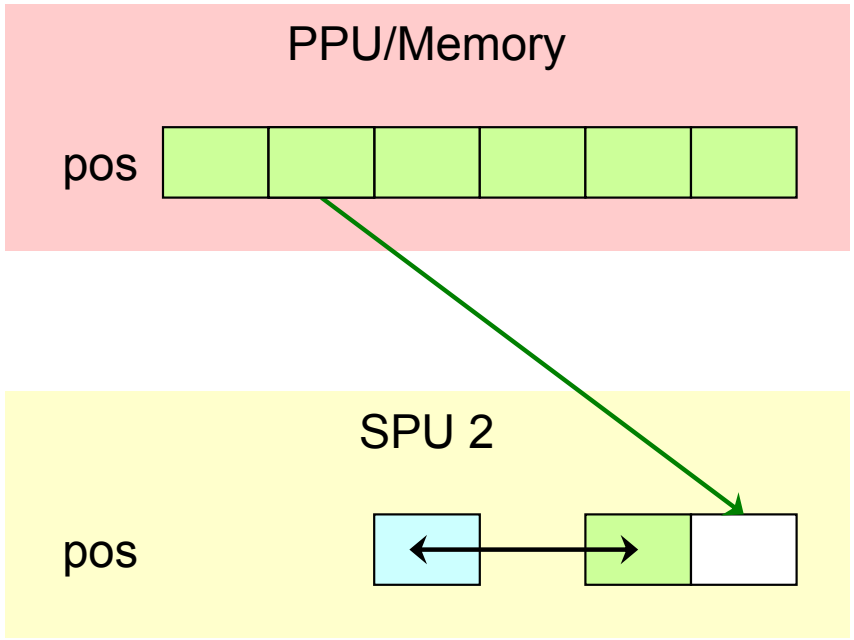
- We're wasting local store space
- Keep 2 buffers
  - Start data transfer into one
  - Process data in other
  - Swap buffers for next transfer



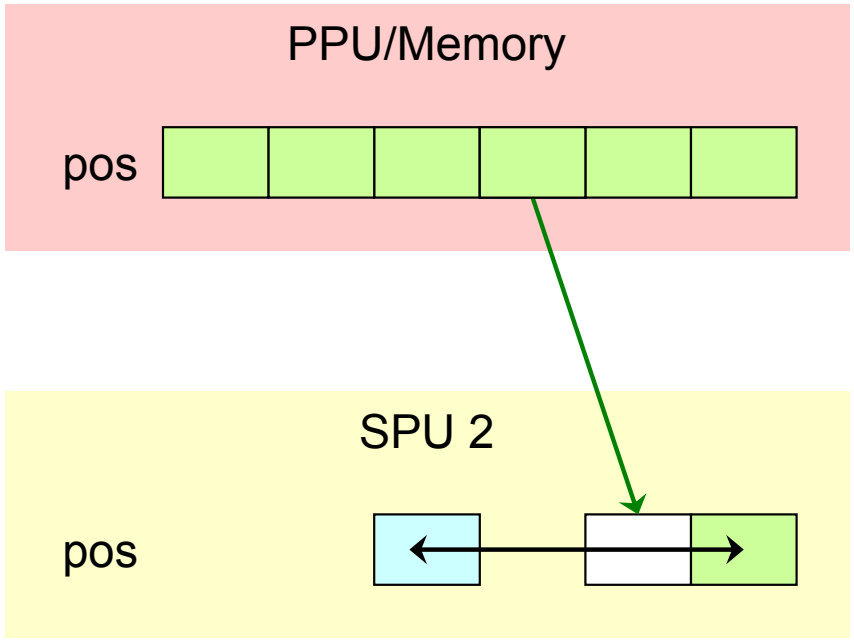
Double-buffering



Double-buffering



Double-buffering



Exercise (20 minutes)

- Read over the code for the pipelined version
  - rec2/sim3/spu/sim_spu.c
- Implement double-buffering
  - Start from the pipelined version

Recitation Plan

- Lab 1 solution
- SPE-SPE communication
- Patterns for mapping computation to SPEs
- Cell-ifying a sample program
  - Hands on programming
  - Optimizations
- Useful resources

Other Performance Optimizations

- Keep data transfer on-chip
  - Copy updated positions of other objects from the local store of the SPU responsible for them instead of memory
  - Most likely will not affect performance by much
- SIMD-ization
- Loop unrolling
- Future lectures and recitations

SPE Threads

- Not the same as “normal” threads
- SPE does not have protection, can only run one thread at a time
  - PPU can “forcibly” context-switch a SPE by saving context, copying out old local store/context, copying in new
- Current SDK does not support context switching SPEs
  - SPE threads are run on physical SPEs in FIFO order
  - If more threads than SPEs, additional threads will wait for running threads to exit before starting
- Don’t create more threads than physical SPEs
- Cell processor has 8 physical SPEs, 6 enabled on PS3 Linux

More Useful Functions

- PPU (libspe)
  - SPE thread groups and events
    - Wait or poll for interrupting mailbox messages and other events from all threads in a group
    - spe_create_group, spe_get_event
- SPU
  - SPE events
    - Wait or poll for mailbox/signals/DMA completion/other by mask
    - spu_write_event_mask, spu_read_event_status, spu_stat_event_status, spu_write_event_ack
  - SPU decrementer
    - Register that counts down at a fixed rate
    - Accurately time parts of SPU program
    - spu_read_decrementer, spu_write_decrementer

List of Useful Functions

PPU (libspe)

```
spe_create_thread
spe_wait

spe_write_in_mbox
spe_stat_in_mbox

spe_read_out_mbox
spe_stat_out_mbox

spe_write_signal

spe_get_ls
spe_get_ps_area

spe_mfc_get
spe_mfc_put
spe_mfc_read_tag_status

spe_create_group
spe_get_event
```

SPU

```
mfc_get
mfc_put
mfc_stat_cmd_queue
mfc_write_tag_mask
mfc_read_tag_status_all/any/immediate

spu_read_in_mbox
spu_stat_in_mbox

spu_write_out_mbox, spu_write_out_intr_mbox
spu_stat_out_mbox, spu_stat_out_intr_mbox

spu_read_signal1/2
spu_stat_signal1/2

spu_write_event_mask
spu_read_event_status
spu_stat_event_status
spu_write_event_ack

spu_read_decrementer
spu_write_decrementer
```