Beyond Gaming: Programming the PS3 Cell Architecture for Cost-Effective Parallel Processing

Rodric Rabbah

IBM Watson Center

Get a PS3, Add Linux

- The PS3 can boot user installed Operating Systems
  - Dual boot: GameOS and Other OS
- Installing Linux on the PS3 is well documented
  - Yellow Dog Linux
  - Fedora Core Linux
  - Other Linux distributions reportedly work as well
  - For recipes: http://cag.csail.mit.edu/ps3/recipes.shtml
- User level access to the PS3 processor: Cell
  - Cell SDK from IBM alphaWorks adds compilers, examples, etc.

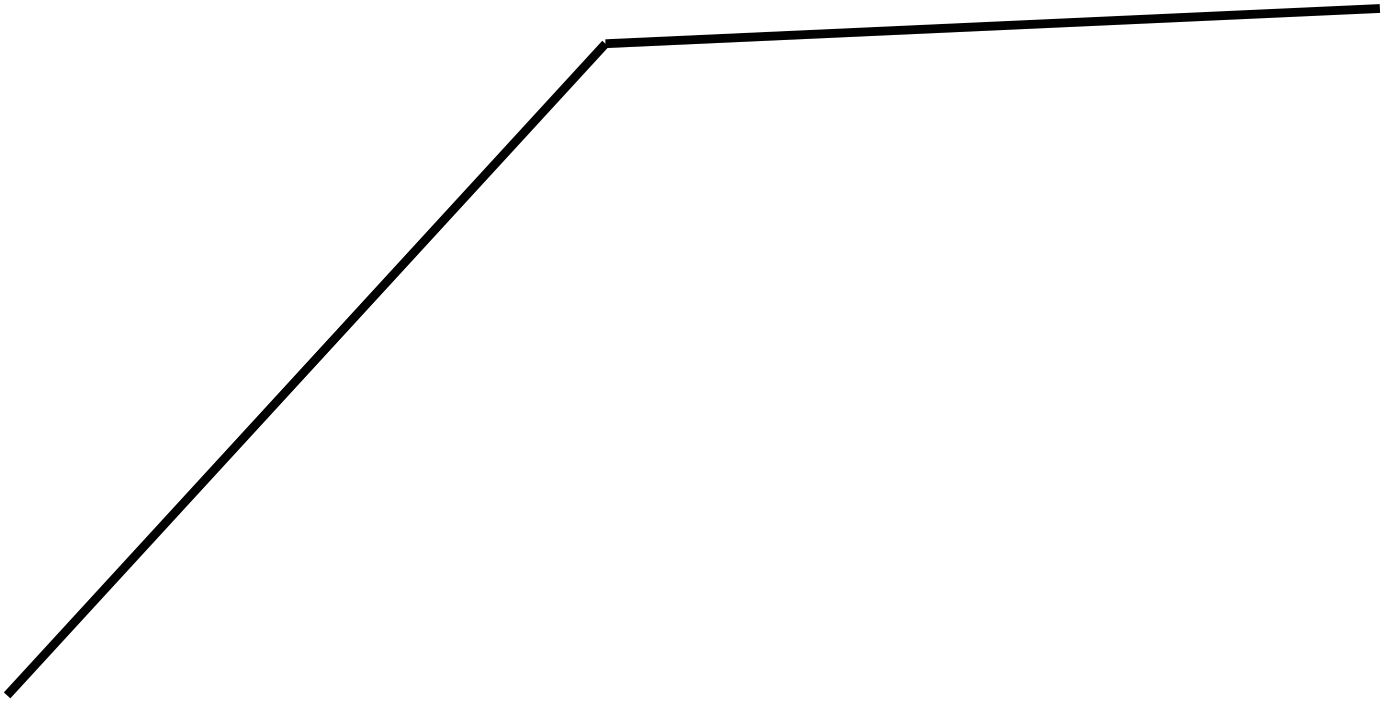
PS3 Cell Processor

- Cell Broadband Engine Architecture
  - Heterogeneous *multicore* architecture with 9 cores
  - 1 general purpose core: Power Processor Element (PPE)
  - 8 accelerator cores: Synergistic Processor Elements (SPEs)

- On the PS3 only 6 SPEs are accessible, and 256MB RAM
  - No access to graphics card

- Cell is unique: one of the first easily accessible (distributed-memory) multicore architectures
  - Distributed-memory, each core has its own local memory
    - SPE can only directly access data in its local store
  - Compared to multicores that shares a cache and can directly access any data in the address space

Obligatory Multicore Slide

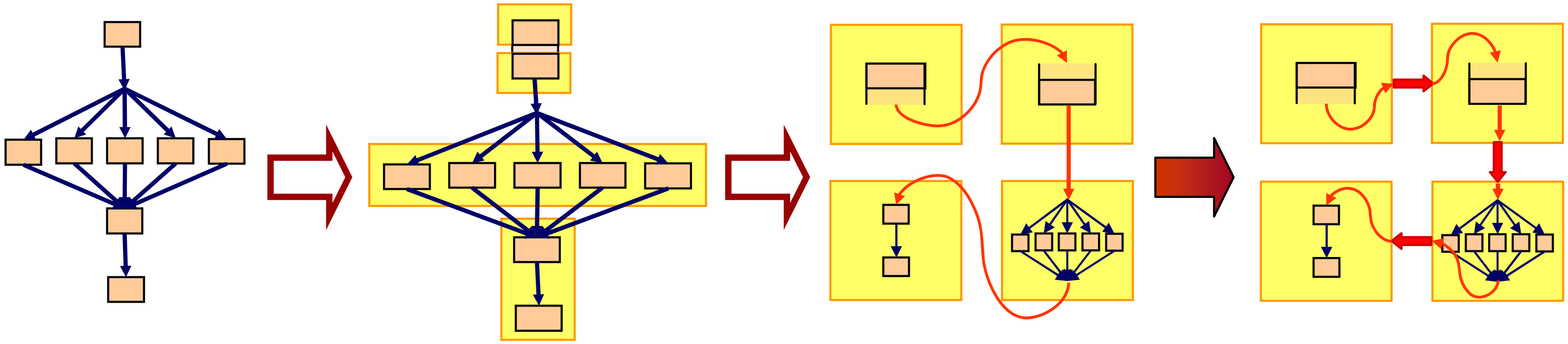


- Monolithic processor design complexity no longer scalable due to power and wire delay limitations
- New design pattern: distribute resources, more cores on a chip

Multicores, how do you program them?

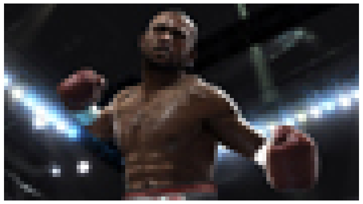
- Painfully!
  But you'll change that…

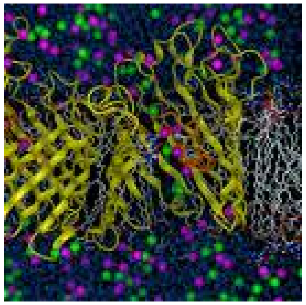
- Multicores require orchestration of concurrent computation across many cores to deliver high performance
  - Cores run in parallel
  - Programming becomes exercise in partitioning, mapping (layout), routing (communication) and scheduling

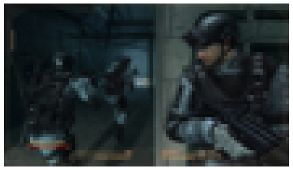


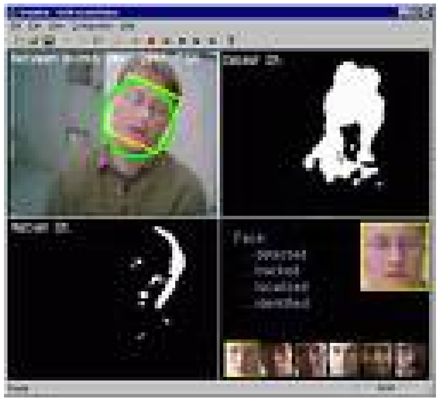
Partitioning **Layout** **Routing and Scheduling**

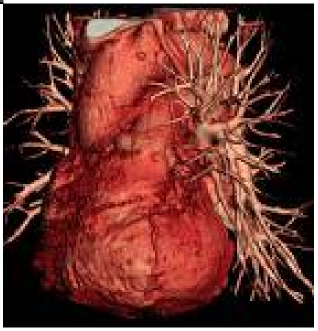
Parallelism Applicable Everywhere

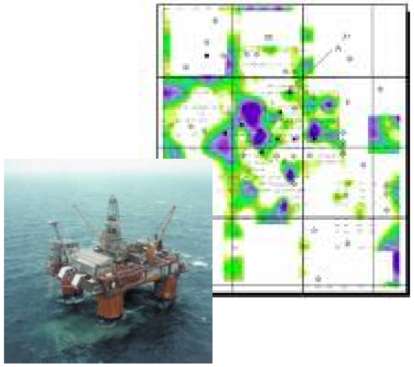


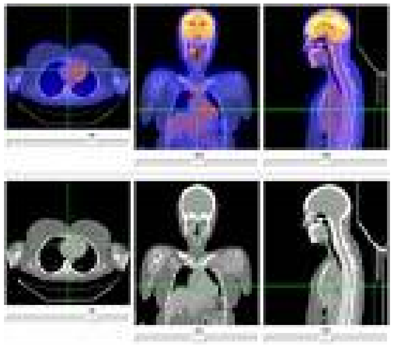


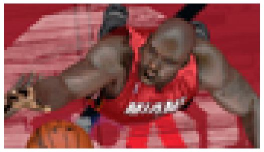




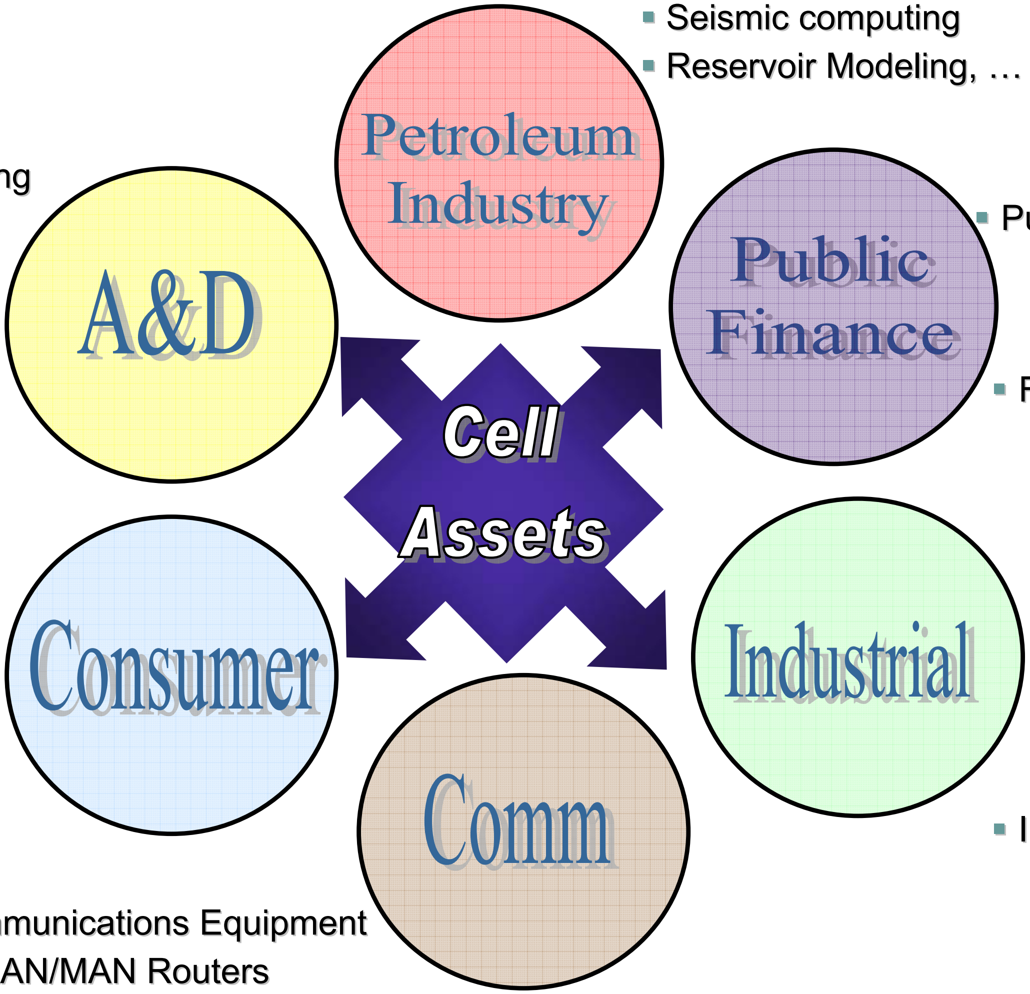








Cell Application Domains



- Petroleum Industry
  - Seismic computing
  - Reservoir Modeling, …

- Aerospace & Defense
  - Signal & Image Processing
  - Security, Surveillance
  - Simulation & Training, …

- Public Sector / Gov't & Higher Educ.
  - Signal & Image Processing
  - Computational Chemistry, …

- Finance
  - Trade modeling

- Consumer / Digital Media
  - Digital Content Creation
  - Media Platform
  - Video Surveillance, …

- Medical Imaging
  - CT Scan
  - Ultrasound, …

- Industrial
  - Semiconductor / LCD
  - Video Conference

- Communications Equipment
  - LAN/MAN Routers
  - Access
  - Converged Networks
  - Security, …

Take Away Messages

- Experience with Cell has demonstrated that good programming models are not optional for multicores

- PS3s offer convenient access to Cell processors and provide a practical platform for research and innovation
  - Many hard problems to solve that are applicable in a more general context

- Using PS3s in an educational setting can provide students with hands on experience that can acclimate them to the parallel programming challenges in a fun and exciting context

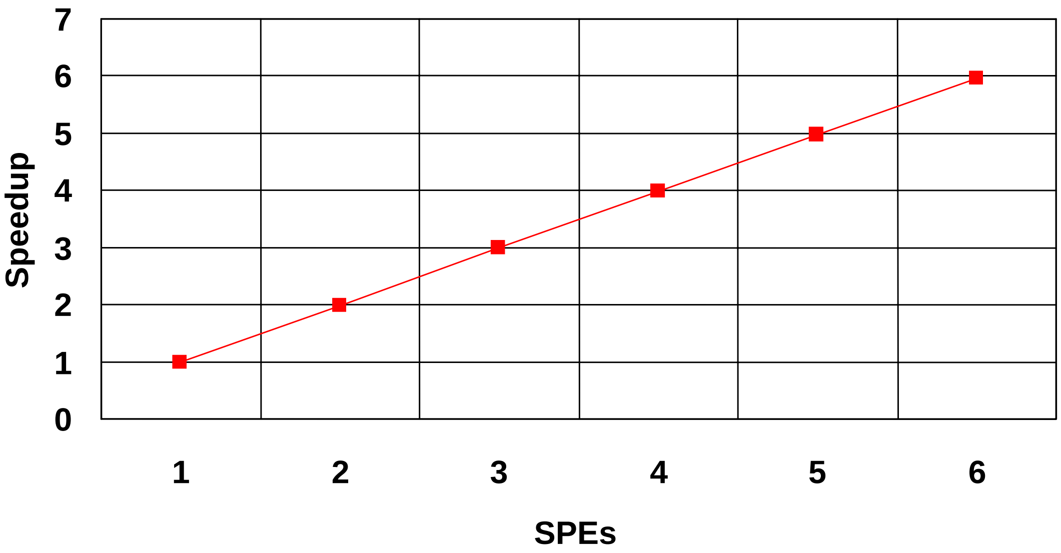
On Teaching Multicore Programming Using PS3s and Cell

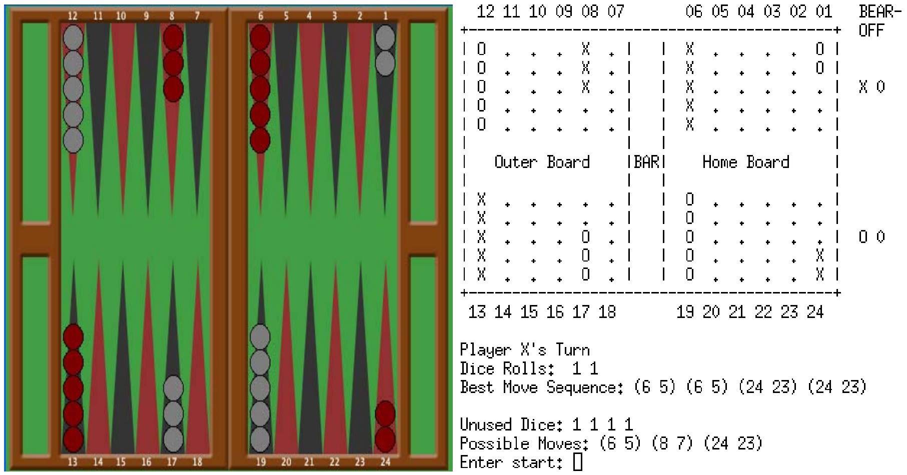
- Multicore programming primer short course at MIT, Jan. 2007
  - Covered parallel programming challenges
    (18 lectures)
  - Offered students hands on parallel programming experience
    (5 recitations, one take-home lab)
  - Culminated in student projects designed and implemented for PS3
    - Students formed teams and determined their own projects
    - Some project source code is available online

- All course material available online
  http://cag.csail.mit.edu/ps3

Sample Student Project: Backgammon

- AI Backgammon player
- 1M board evaluations in ~3 seconds (6 SPEs)
- Data parallel implementation, linear speedup



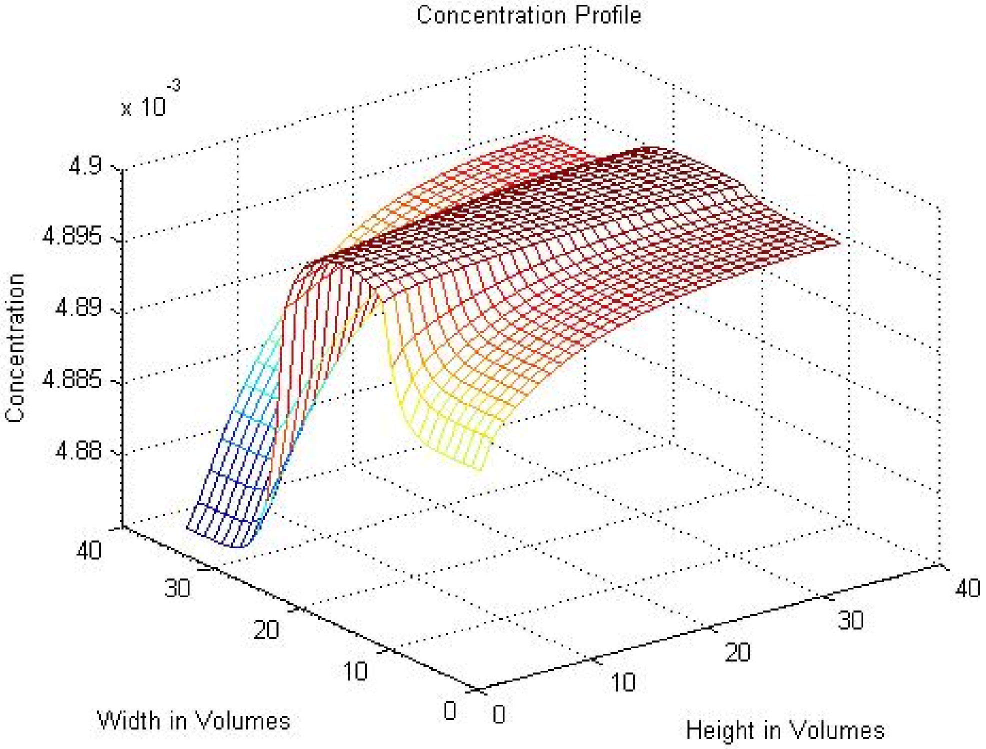


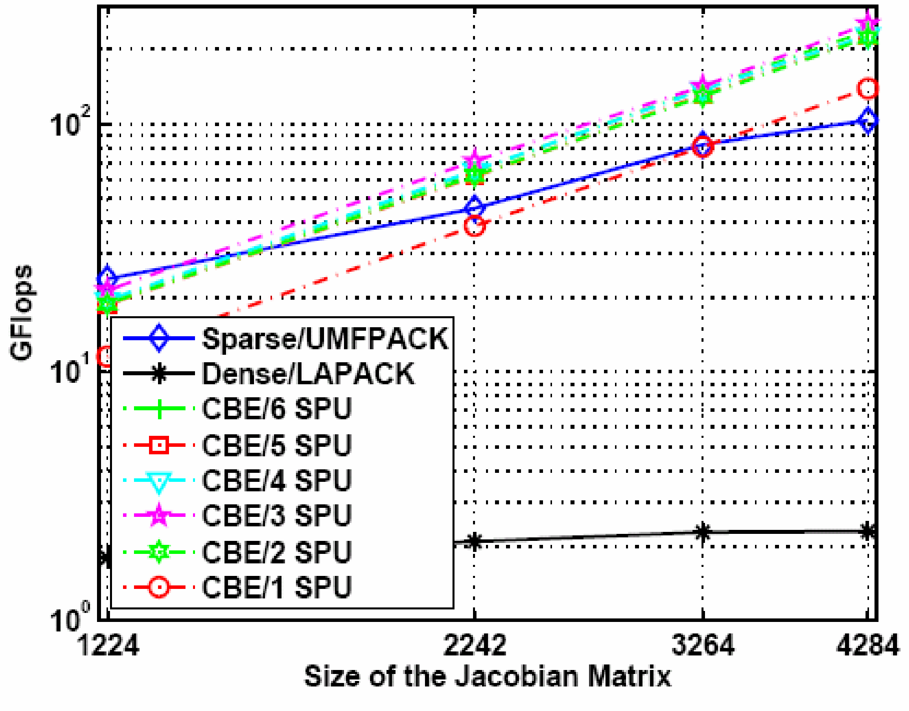
```
  12 11 10 09 08 07       06 05 04 03 02 01   BEAR-
+-------------------------------------------+ OFF
| O  .  .  .  X  .  |    | X  .  .  .  .  O |
| O  .  .  .  X  .  |    | X  .  .  .  .  O |
| O  .  .  .  X  .  |    | X  .  .  .  .  . |  X O
| O  .  .  .  .  .  |    | X  .  .  .  .  . |
| O  .  .  .  .  .  |    | X  .  .  .  .  . |
|                   |    |                  |
|   Outer Board     |BAR|   Home Board      |
|                   |    |                  |
| X  .  .  .  .  .  |    | O  .  .  .  .  . |
| X  .  .  .  .  .  |    | O  .  .  .  .  . |
| X  .  .  .  O  .  |    | O  .  .  .  .  . |  O O
| X  .  .  .  O  .  |    | O  .  .  .  .  X |
| X  .  .  .  O  .  |    | O  .  .  .  .  X |
+-------------------------------------------+
 13 14 15 16 17 18       19 20 21 22 23 24

Player X's Turn
Dice Rolls:  1 1
Best Move Sequence: (6 5) (6 5) (24 23) (24 23)

Unused Dice: 1 1 1 1
Possible Moves: (6 5) (8 7) (24 23)
Enter start: []
```

Eddie Scholtz and Mike Fitzgerald
http://cag.csail.mit.edu/ps3/backgammon.shtml

Sample Student Project: Battery Simulation

- 2D electrochemical model of lead acid battery cell
  - Solves linear system using two solvers
  - Banded LU solver and dense LU solver

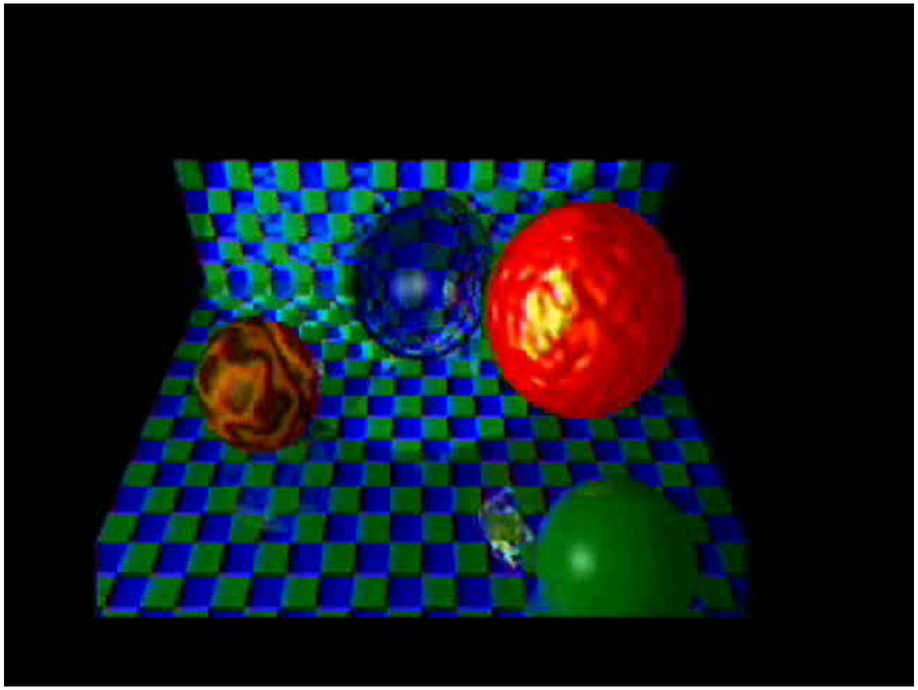




James Geraci, Sudarshan Raghunathan, and John Chu
http://cag.csail.mit.edu/ps3/battery-model.shtml

Sample Student Project: Ray Tracer

- Full ray tracer running on each SPE
- Data parallel implementation



Blue-Steel team (6 students)
http://cag.csail.mit.edu/ps3/blue-steel.shtml

Tutorial Agenda

- Brief overview of Cell performance characteristics

- Programming Cell
  - Cell components
  - Application walk through
  - Inter-core parallelism: structuring computation and communication
  - Orchestration: synchronization mechanisms
  - SIMD for single thread performance: it still matters

- Opportunities for research and innovation, and education
  - Programming Language
  - Parallelizing Compiler
  - Abstract Streaming Layer

Acknowledgements

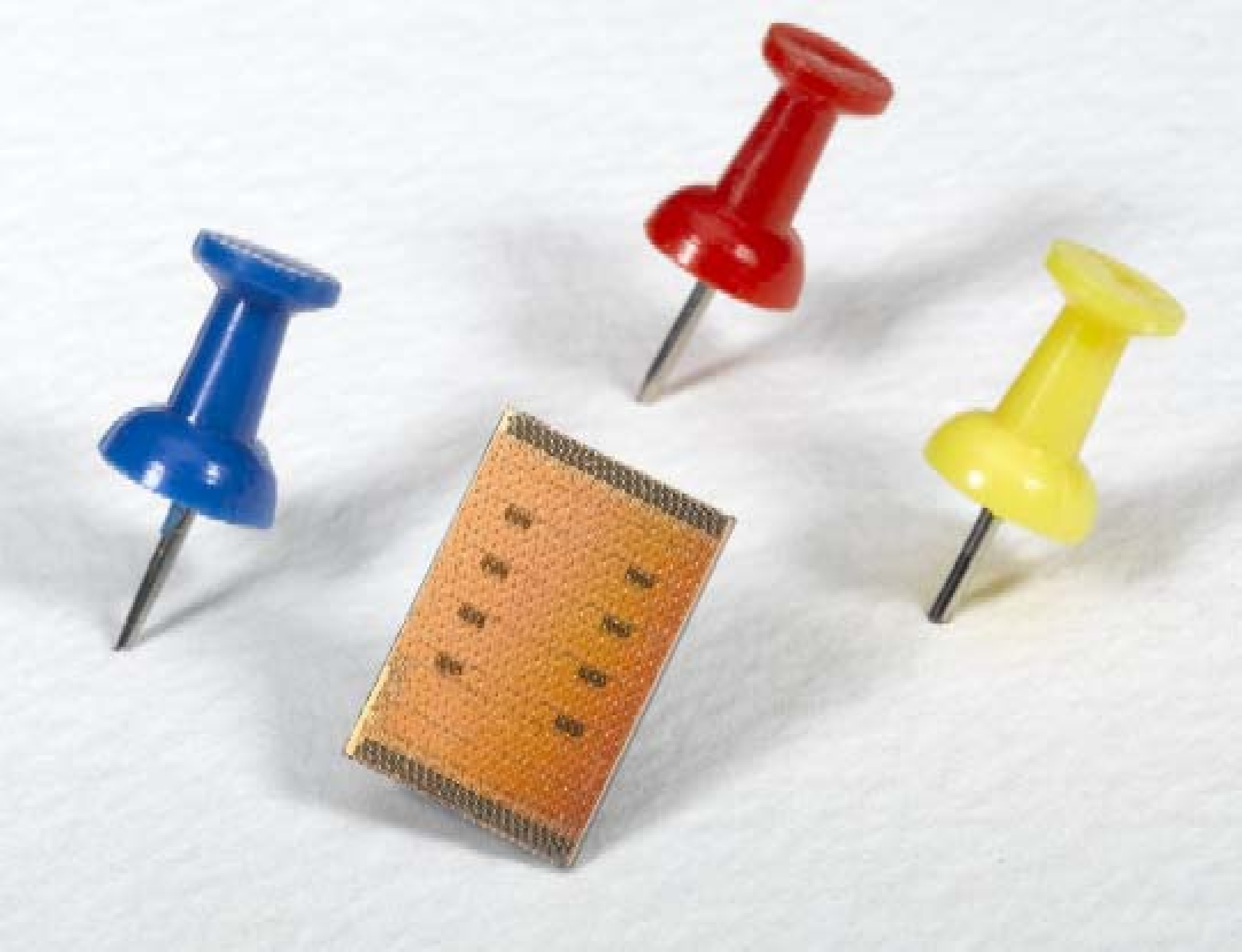
IBM, especially

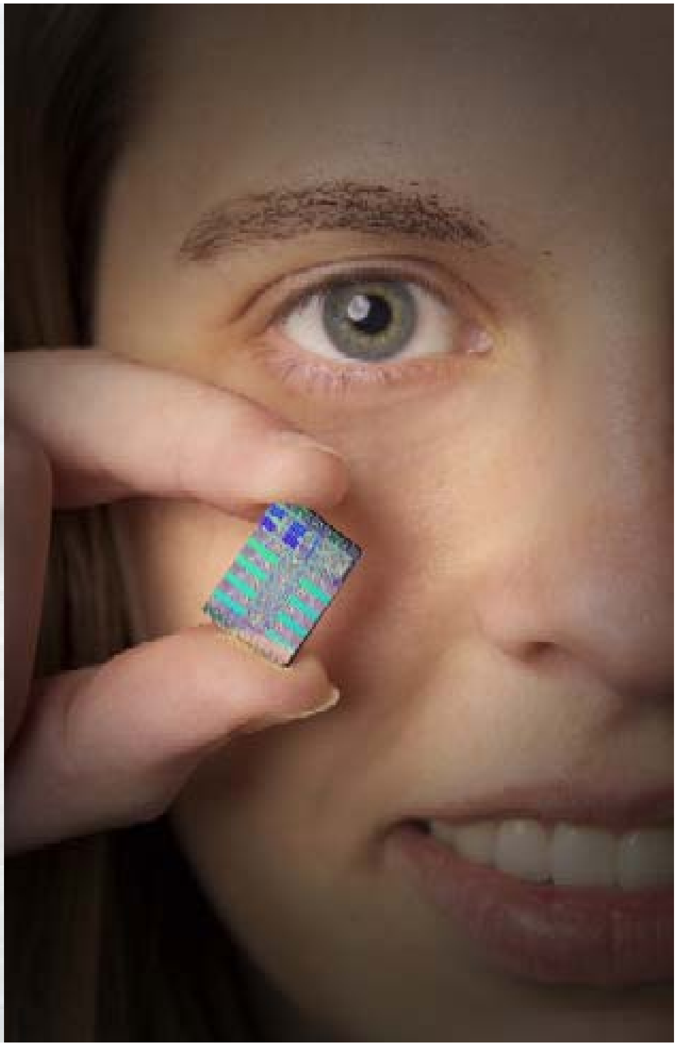
- Dr. Bruce D'Amora
- Dr. Michael Perrone

StreamIt group at MIT, especially

- Phil Sung (MEng)
- David Zhang (MEng)

Cell





Cell History

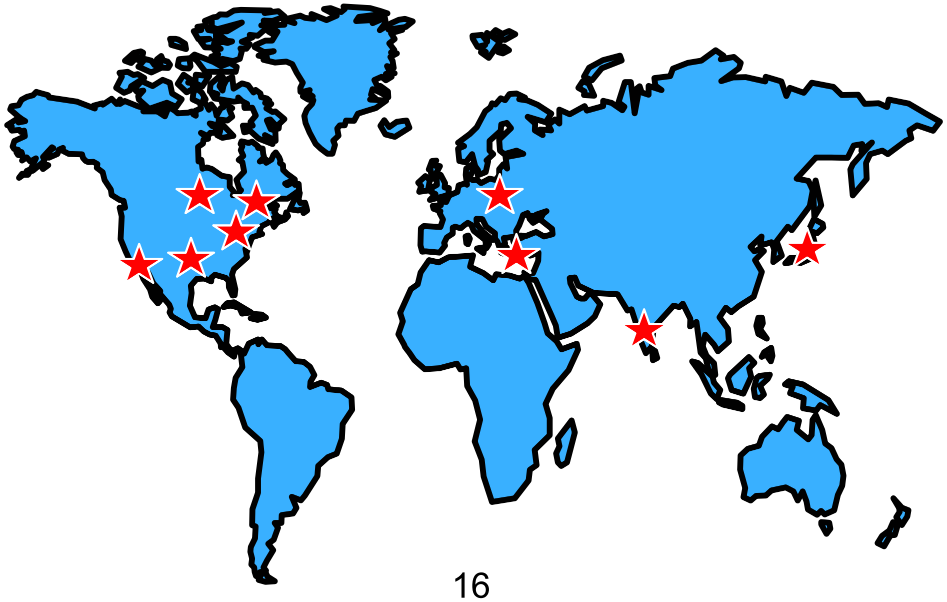
- IBM, SCEI/Sony, Toshiba Alliance formed in 2000
- Design Center opened in March 2001 (based in Austin, Texas)
- Single Cell BE operational Spring 2004
- 2-way SMP operational Summer 2004
- February 7, 2005: First technical disclosures
- October 6, 2005: Mercury Announces Cell Blade
- November 9, 2005: Open Source SDK & Simulator Published
- November 14, 2005: Mercury Announces Turismo Cell Offering
- February 8, 2006 IBM Announced Cell Blade

SONY COMPUTER ENTERTAINMENT

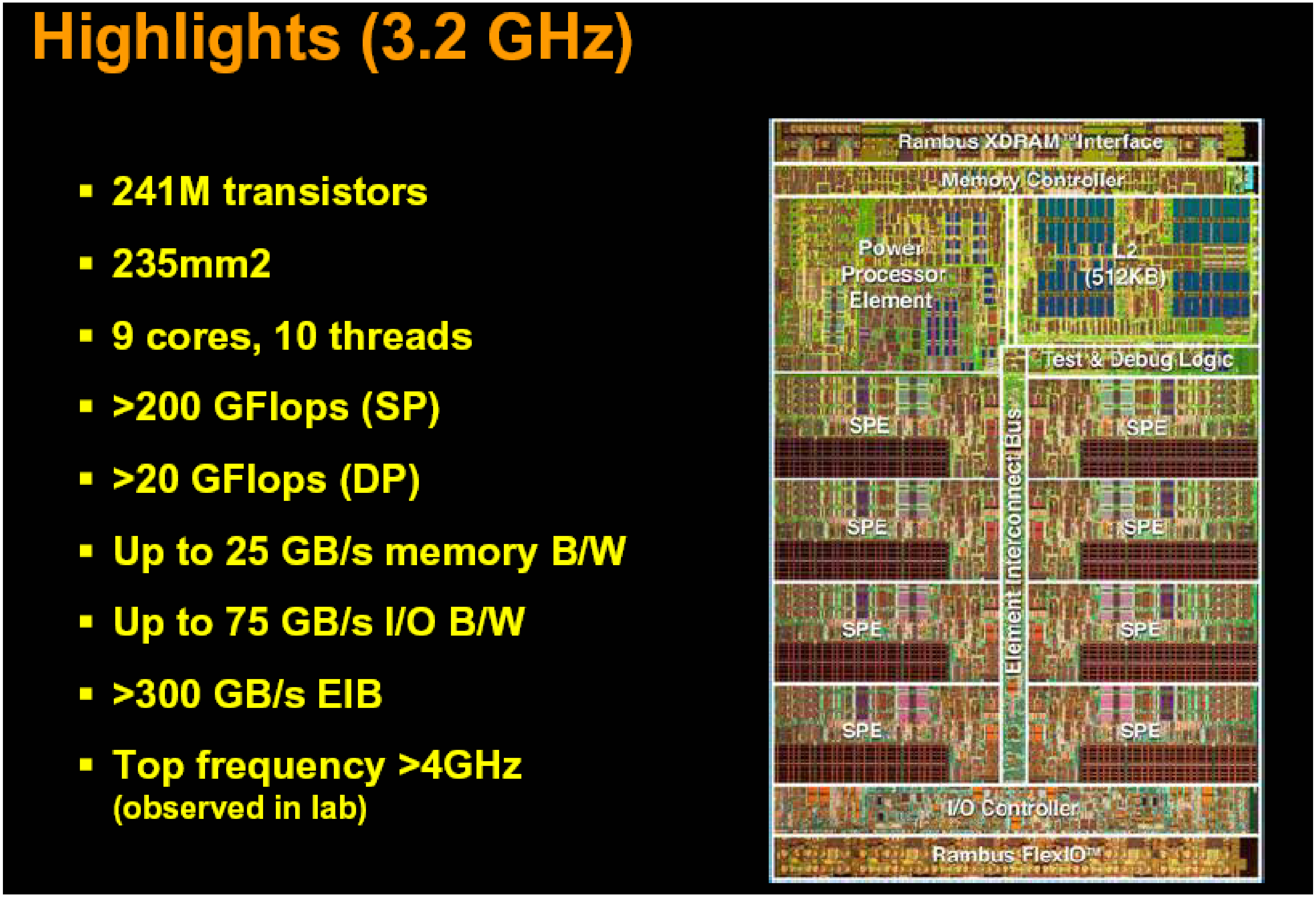
SONY

TOSHIBA

IBM



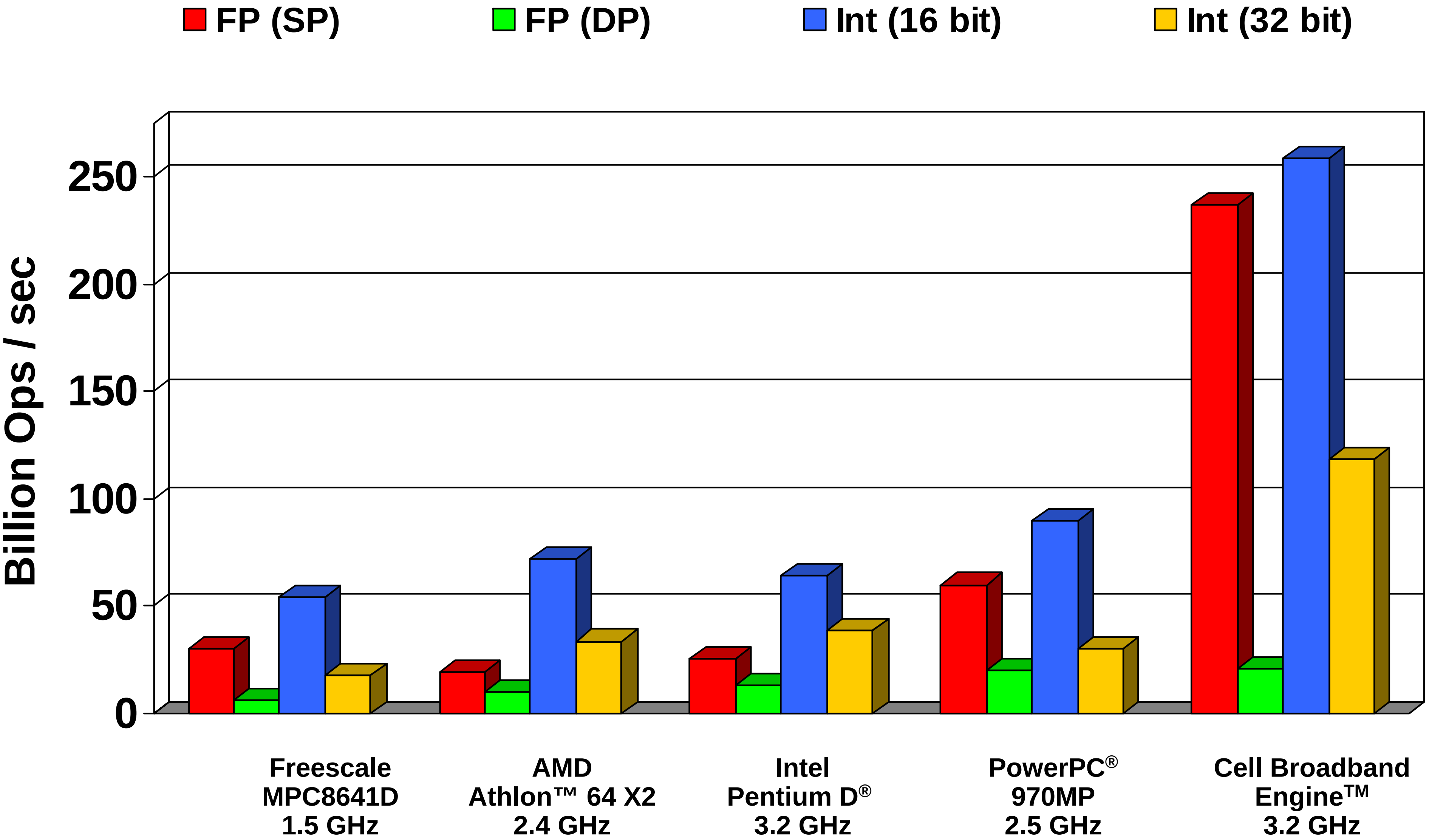
Cell Chip



Highlights (3.2 GHz)

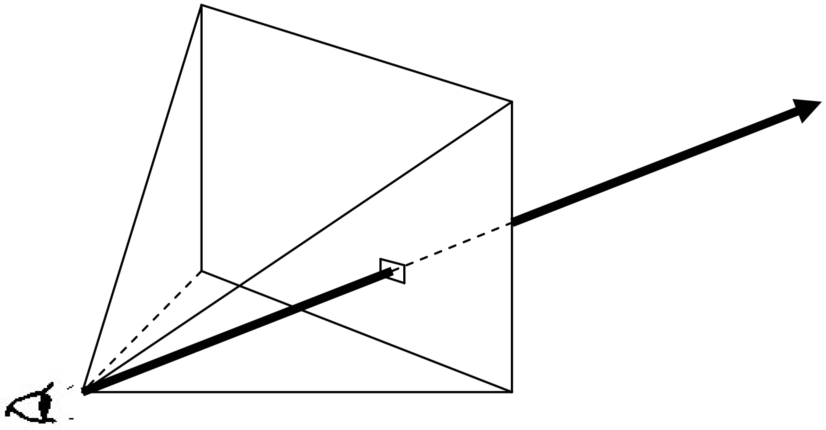
- 241M transistors
- 235mm2
- 9 cores, 10 threads
- >200 GFlops (SP)
- >20 GFlops (DP)
- Up to 25 GB/s memory B/W
- Up to 75 GB/s I/O B/W
- >300 GB/s EIB
- Top frequency >4GHz
  (observed in lab)

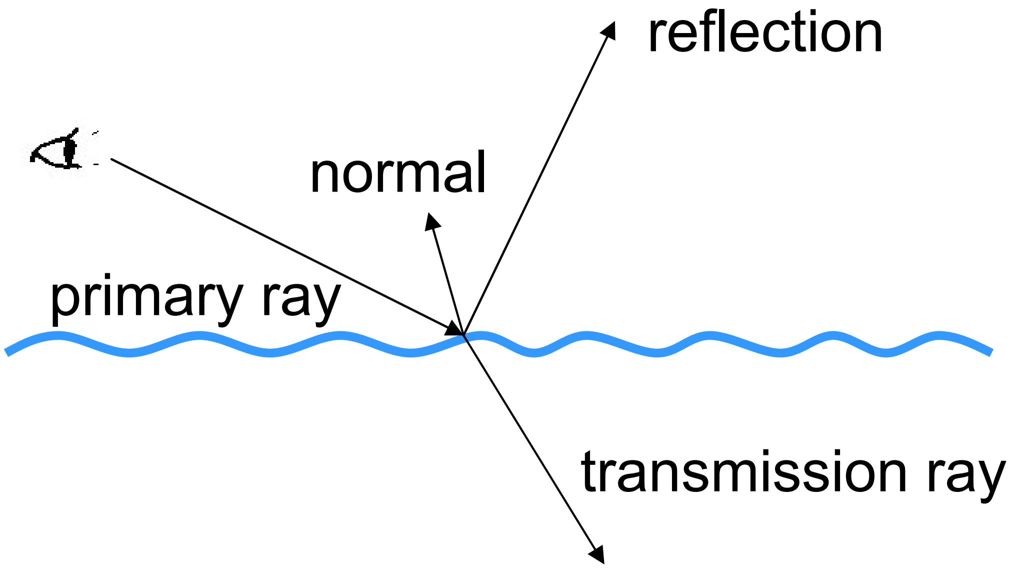
Theoretical Peak Operations



Rendering Scenes by Ray Tracing

- Shoot rays into scene through pixels in image plane and follow their paths
  - Rays bounce around as they strike objects
  - Rays generate new rays
  - Result is color and opacity for that pixel
  - Abundant parallelism (process rays in parallel)





reflection

normal

primary ray

transmission ray

SPEs vs GPU

- Cell 4-5x better performance
  - 7 SPEs used for rendering
  - 1 SPE reserved for image compression
- Renewed interest in ray tracing
  - Real-time ambient occlusion
  - Now practical for real time
  - Visualization of huge digital mode
- Seamless Scale Out
  - More cores → More performance

Courtesy of Barry Minor, IBM

IBM Interactive Ray Tracer (iRT) Demo

http://www.alphaworks.ibm.com/tech/irt

Key Performance Characteristics

- Cell performance ~10x better than GPP for media and other applications that can take advantage of its SIMD capability
  - PPE performance is comparable to a traditional GPP performance
  - SPE performance mostly the same as, or better than, a GPP with SIMD
  - Performance scales with number of SPEs
- Cell sufficiently versatile to cover a wide application
  - Floating point operations
  - Integer operations
  - Data streaming / throughput support
  - Real-time support
- Cell architecture is exposed to software (compilers and applications)
  - Performance gains from tuning can be significant
  - Tools are provided to assist in performance debugging and tuning

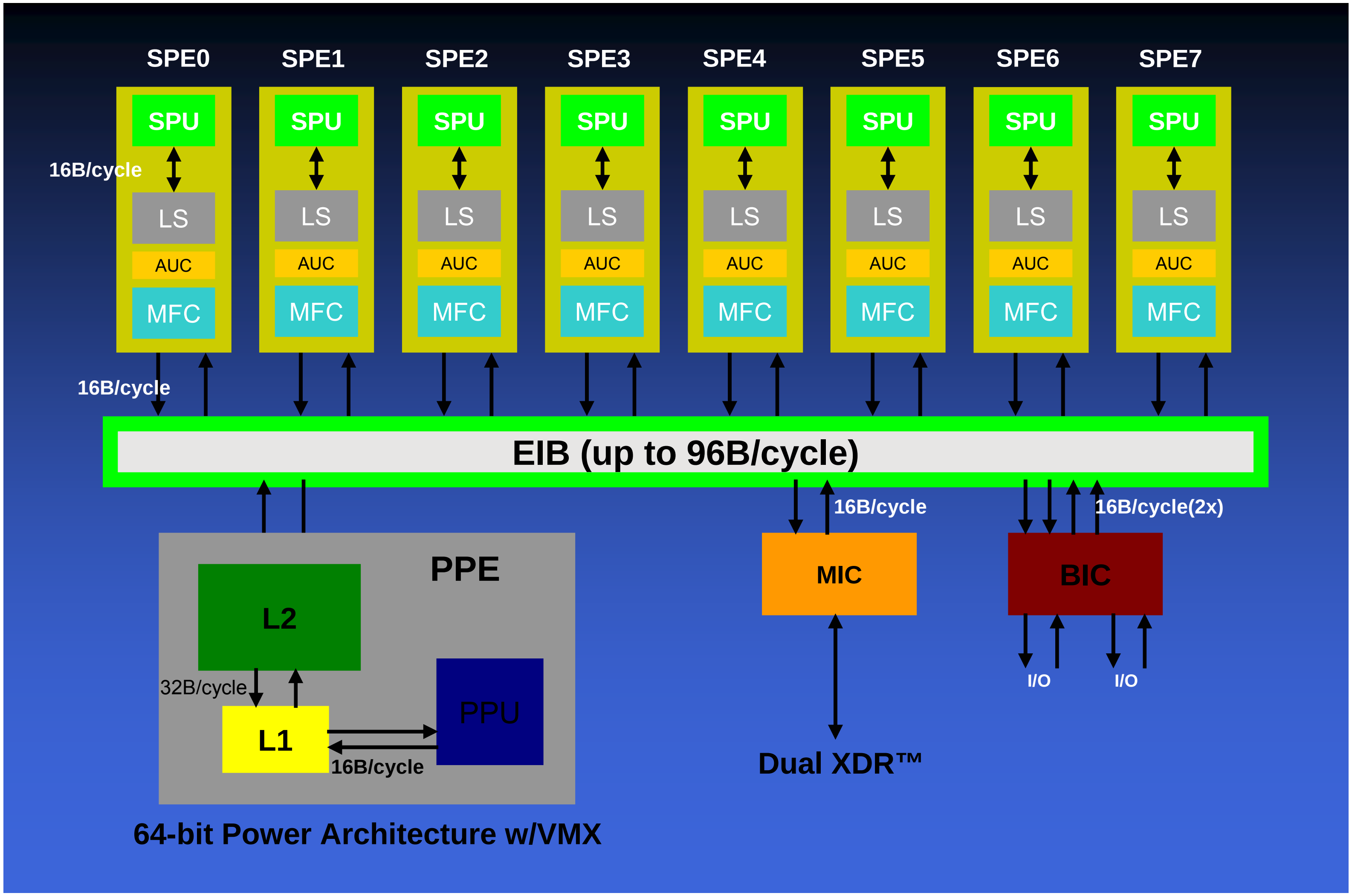
Tutorial Agenda

- Brief overview of Cell performance characteristics

- Programming Cell
  - Cell components
  - Application walk through
  - Inter-core parallelism: structuring computation and communication
  - Orchestration: synchronization mechanisms
  - SIMD for single thread performance: it still matters

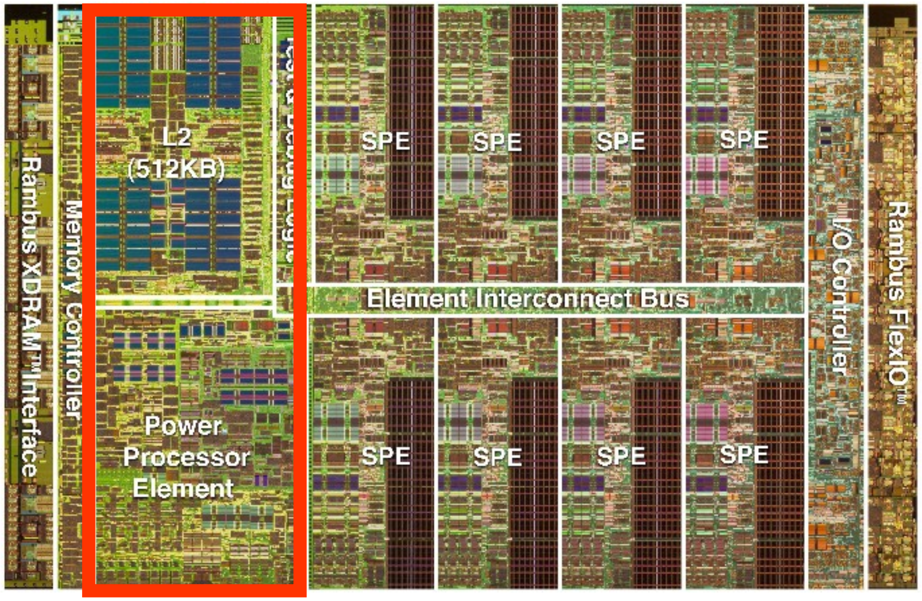
- Opportunities for research and innovation, and education
  - Programming Language
  - Parallelizing Compiler
  - Abstract Streaming Layer

Cell Broadband Engine Architecture



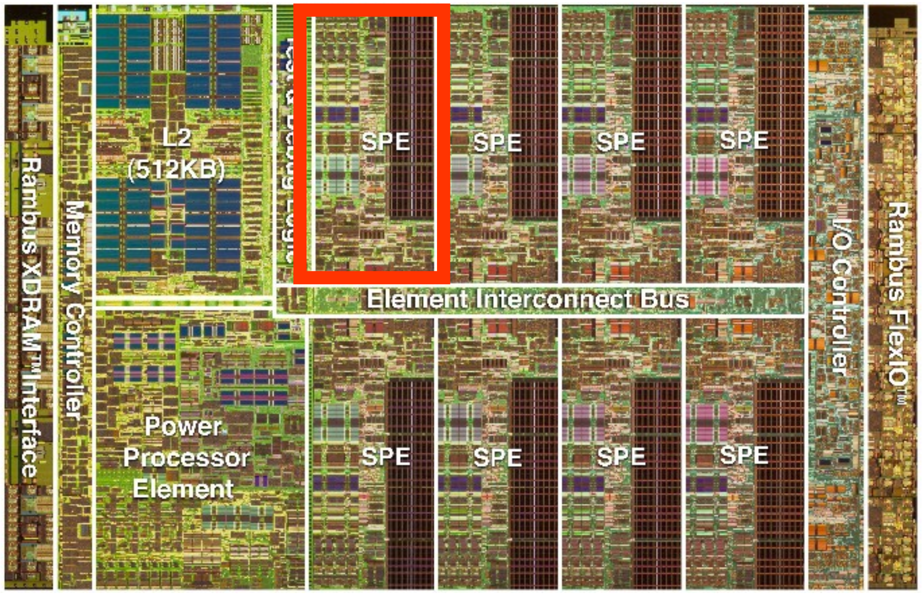
Power Processor Element

- PPE handles operating system and control tasks
- 64-bit Power Architecture with VMX
- In-order, 2-way hardware simultaneous multi-threading (SMT)
- 32KB L1 cache (I & D) and 512KB L2



Synergistic Processor Element

- Specialized high performance core
- Three main components
  - SPU: processor
  - LS: local store memory
  - MFC: memory flow control manages data in and out of SPE



SPU Processing Core

- In-order processor: no speculation or branch prediction
- Greatest compute power is single precision floating point
  - Single precision floating point is not full IEEE compliant, similar to graphics HW
  - Double precision floating point is full IEEE compliant
- 128 unified registers used for all data types
- Can only access (load & store) data in the SPE local store

Local Store (LS)

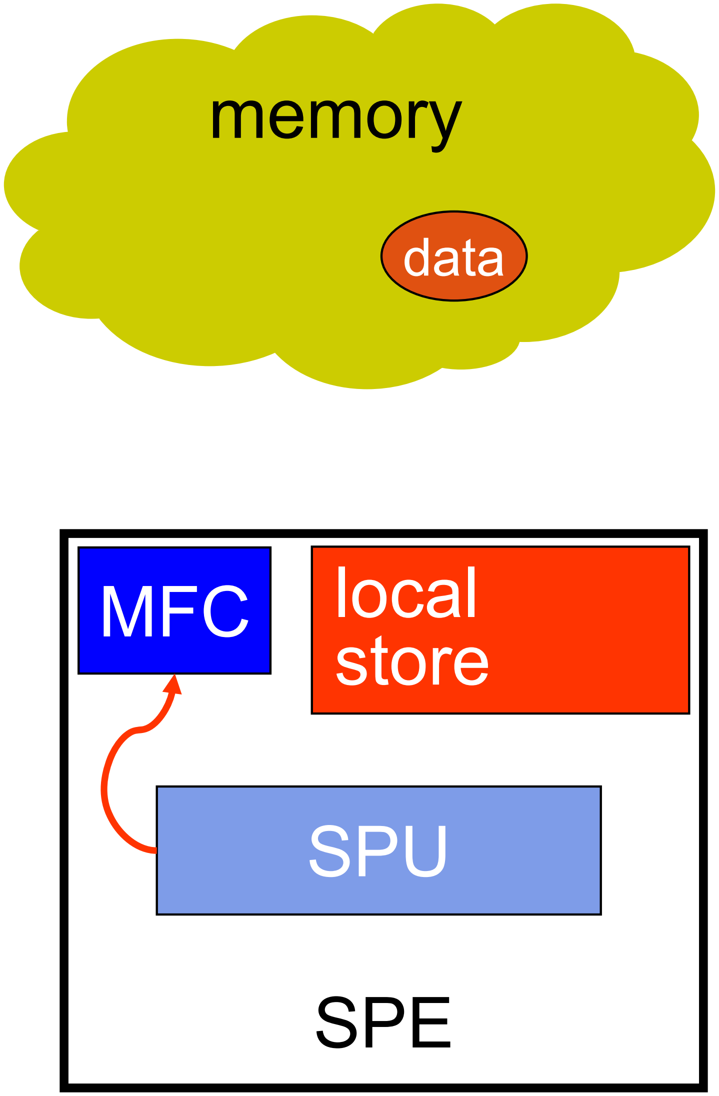
- 256KB of memory per SPE
- Code and data share LS
- SPU can load 16B per cycle from LS

- Data from main memory is explicitly copied to and from the local store since SPU cannot access any other memory locations directly

Data In and Out of the SPE Local Store

- SPU needs data

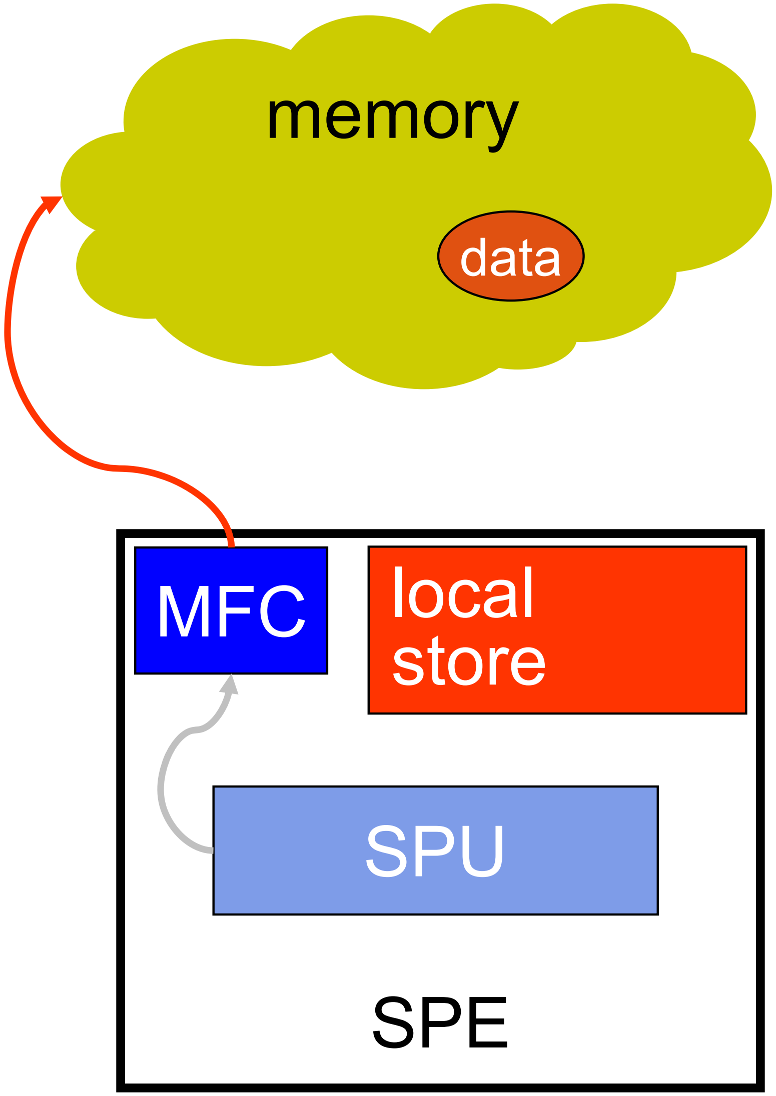
1. SPU initiates MFC request for data



Data In and Out of the SPE Local Store

- SPU needs data

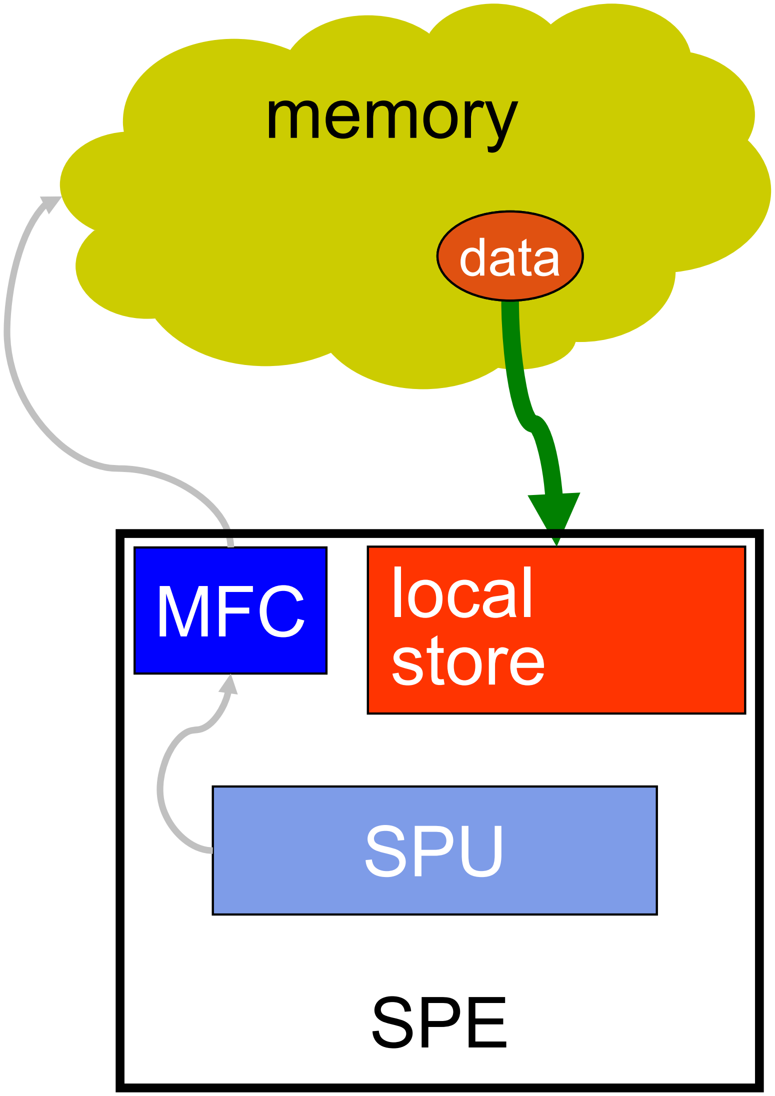
1. SPU initiates MFC request for data
2. MFC requests data from memory



Data In and Out of the SPE Local Store

- SPU needs data

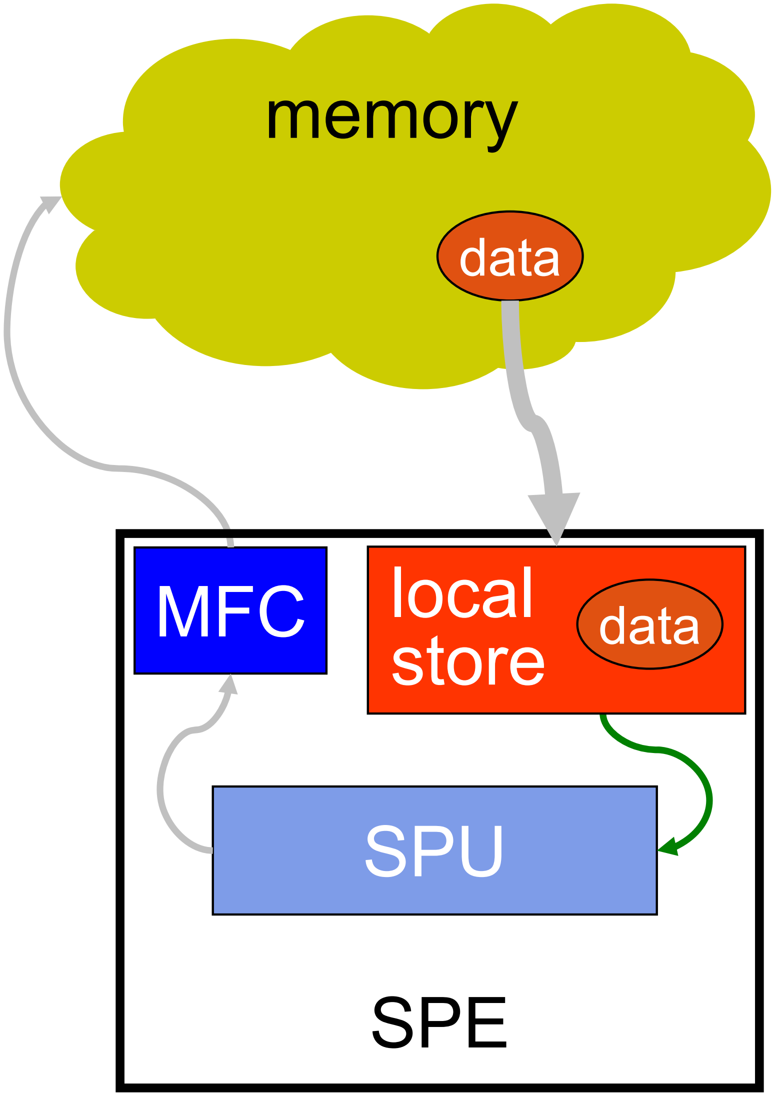
1. SPU initiates MFC request for data
2. MFC requests data from memory
3. Data is **copied** to local store



Data In and Out of the SPE Local Store

- SPU needs data

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

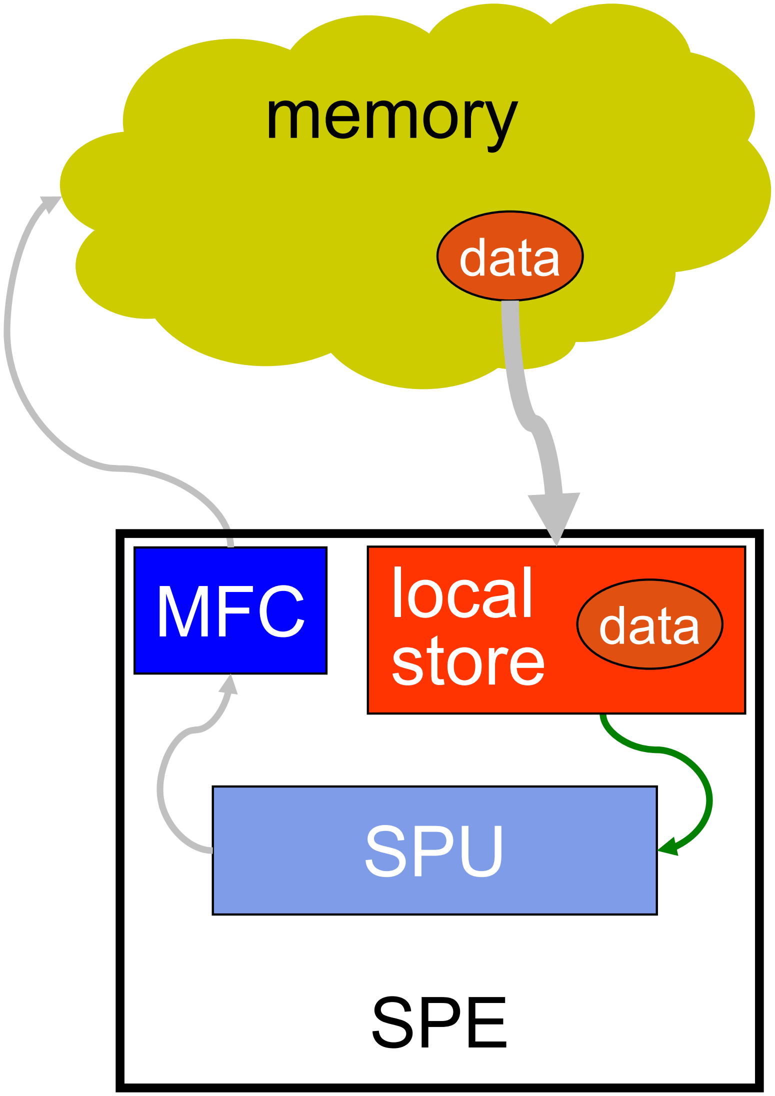


Data In and Out of the SPE Local Store

- SPU needs data

1. SPU initiates MFC request for data
2. MFC 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



MFC DMAs and SPEs

- 1 Memory Flow Controller (MFC) per SPE
- High bandwidth – 16B/cycle

- Each MFC can service up to 24 outstanding DMA commands
  - 16 transfers initiated by SPU
  - 8 additional transfers initiated by PPU
  - PPU initiates transfers by accessing MFC through MMIO registers

- DMA transfers initiated using special channel instructions

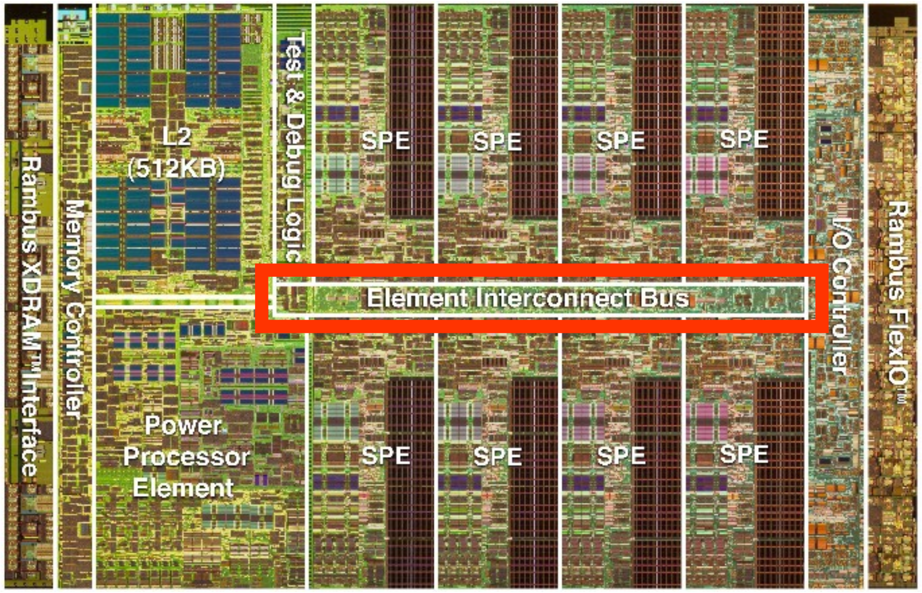
MFC DMAs and SPEs

- DMA transfers data between virtual address space and local store
  - SPE uses PPE address translation machinery
  - Each SPE local store is mapped in virtual address space
    - Allows direct local store to local store transfers
    - Completely on chip, very fast

- Once DMA commands are issued, MFC processes them independently
  - SPU continues executing/accessing local store
  - Communication-computation concurrency/multibuffering essential for performance

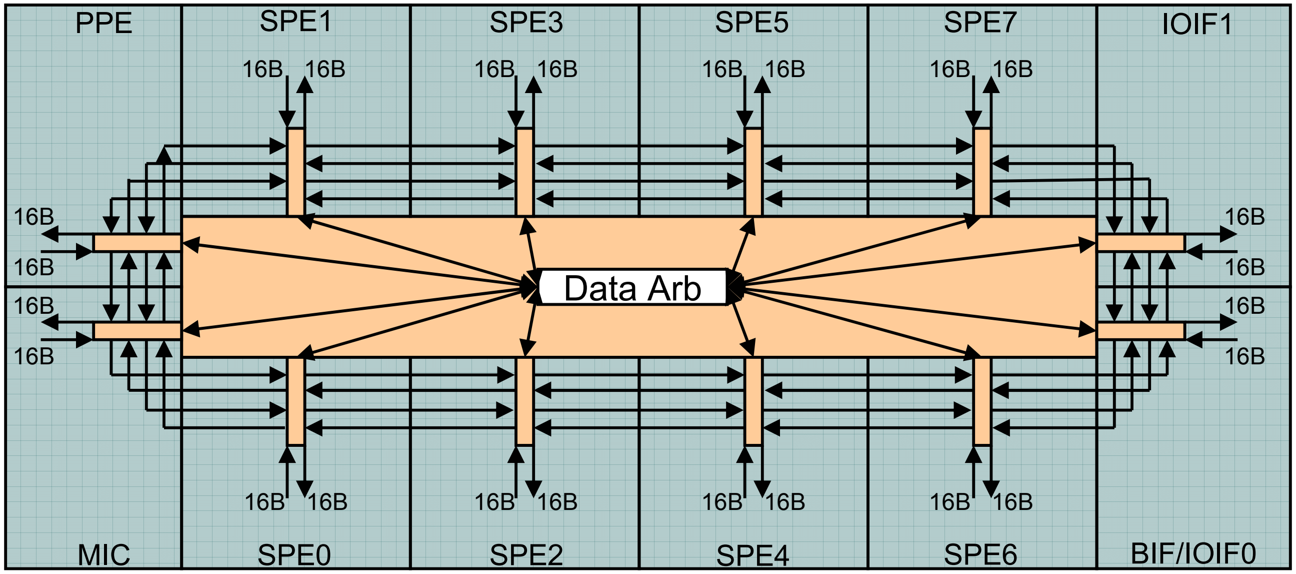
Element Interconnect Bus

- EIB data ring for internal communication
- Four 16B data rings, supporting multiple transfers
  - 2 clockwise and 2 counter-clockwise
- 96B/cycle peak bandwidth
- Over 100 outstanding requests



EIB Data Topology

- Physically overlaps all processor elements
- Central arbiter supports up to 3 concurrent transfers per ring
  - 2 stage, dual round robin arbiter
- Each port supports concurrent 16B in and 16B out data path
  - Ring topology is transparent to element data interface



Internal Bandwidth Capability

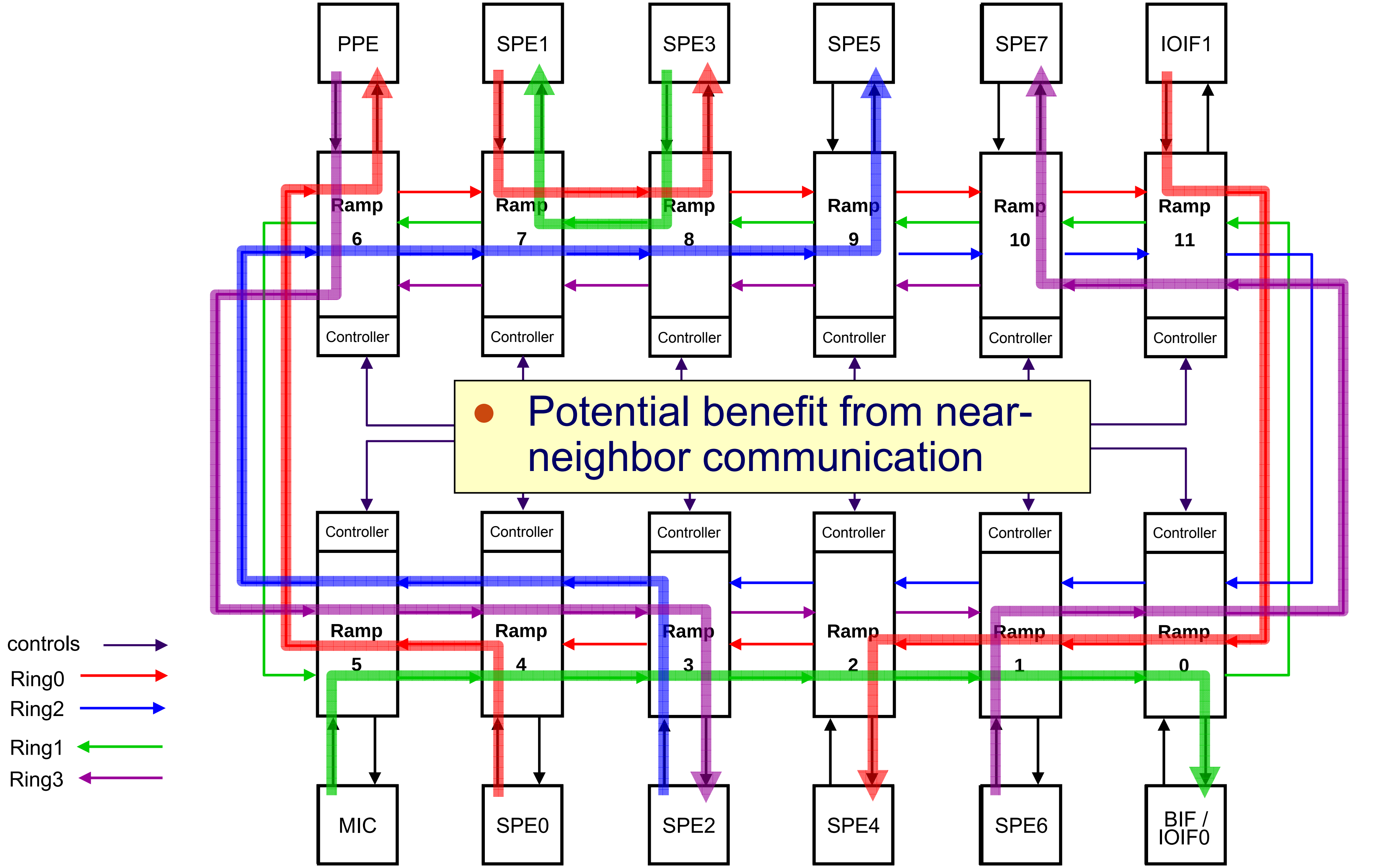
- Each EIB Bus data port supports 25.6GBytes/sec* in each direction

- The EIB Command Bus streams commands fast enough to support 102.4 GB/sec for coherent commands, and 204.8 GB/sec for non-coherent commands.

- The EIB data rings can sustain 204.8GB/sec for certain workloads, with transient rates as high as 307.2GB/sec between bus units

* Assuming a 3.2GHz core frequency – internal bandwidth scales with core frequency

Example of 8 Concurrent Transactions



- Potential benefit from near-neighbor communication

Programming Cell

The good and the hard

What Makes Cell Diff*?

- Multiple programs in one
  - PPU and SPU programs cooperate to carry out computation

- Local store
  - Something new to worry about, but potential for high performance

- Short vector parallelism (SIMD)
  - Intra-core parallelism in addition to parallelism across cores

SPU Programs

- SPU programs are designed and written to work together but are compiled independently
- Separate compiler and toolchain (ppu-gcc and spu-gcc)
- Produces small ELF image for each program that can be embedded in PPU program
  - Contains own data, code sections
  - On startup, C runtime (CRT) initializes and provides malloc
  - printf/mmap/other I/O functions are implemented by calling on the PPU to service the request

A Simple Cell Program

PPU (hello.c)

```
#include <stdio.h>
#include <libspe.h>

extern spe_program_handle_t hello_spu;

int main() {
  speid_t id[8];

  // Create 8 SPU threads
  for (int i = 0; i < 8; i++) {
    id[i] = spe_create_thread(0,
                              &hello_spu,
                              NULL,
                              NULL,
                              -1,
                              0);
  }

  // Wait for all threads to exit
  for (int i = 0; i < 8; i++) {
    spe_wait(id[i], NULL, 0);
  }

  return 0;
}
```

SPU (hello_spu.c)

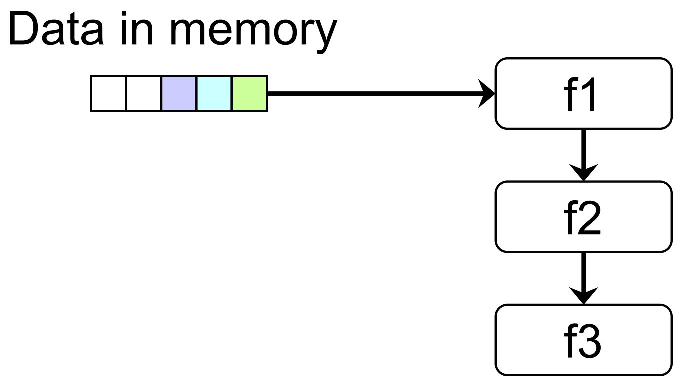
```
#include <stdio.h>

int
main(unsigned long long speid,
     unsigned long long argp,
     unsigned long long envp)
{
  printf("Hello world! (0x%x)\n", (unsigned int)speid);
  return 0;
}
```

SPE Threads

- Not the same as “normal” threads
- SPE does not have protection, 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
- Early SDKs did 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
  - Improvements to this model in newer SDKs

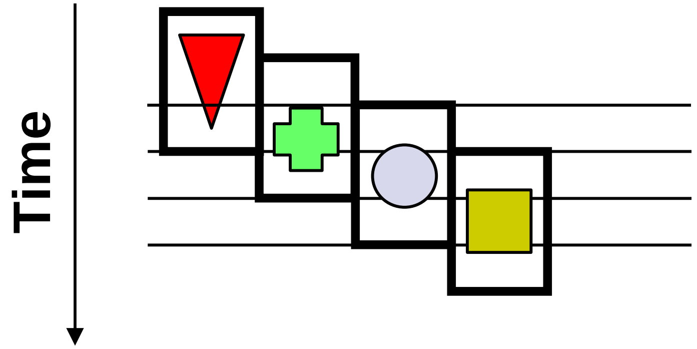
Mapping Computation to SPEs

- Example: single-threaded program performs computation in three stages on data: f3 (f2 (f1 (…))

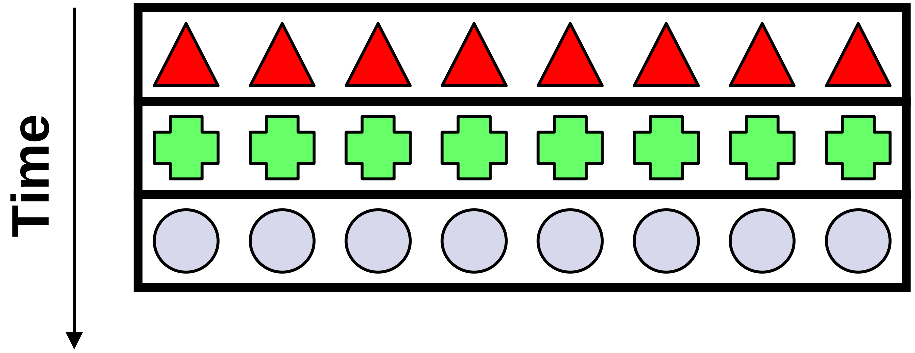


- Several possible parallel mappings to SPEs

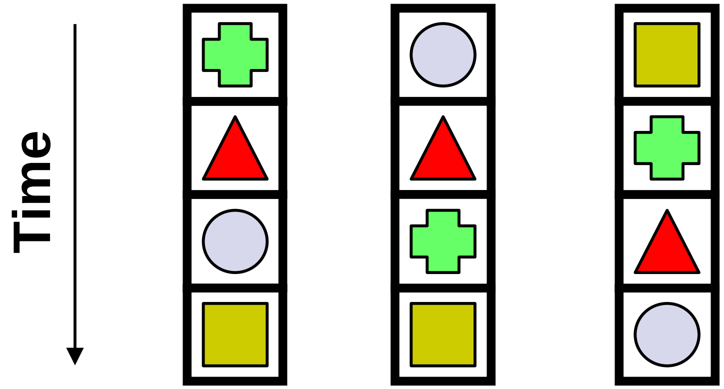
Types of Parallelism



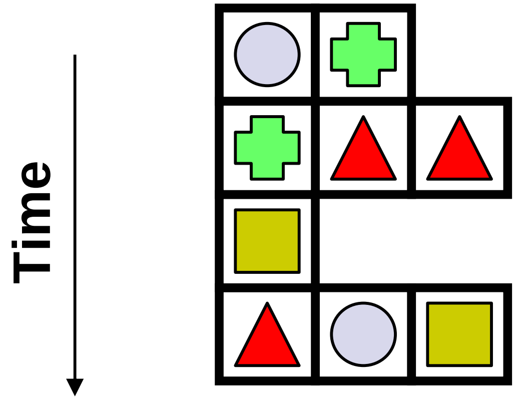
Pipeline Parallelism



Data-Level Parallelism (DLP)



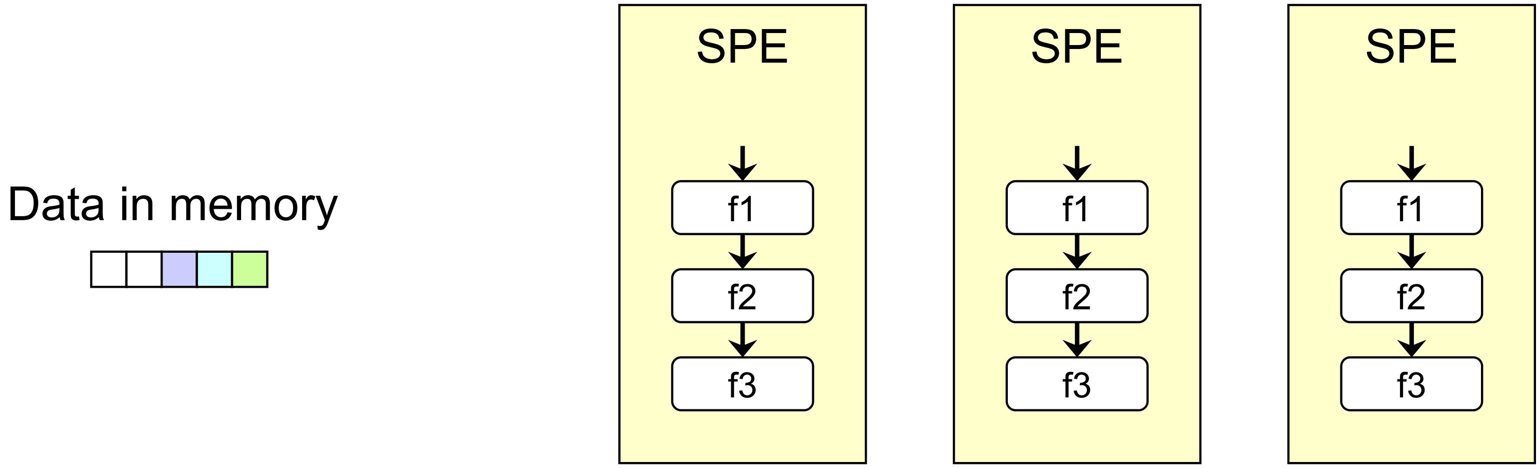
Thread-Level Parallelism (TLP)



Instruction-Level Parallelism (ILP)

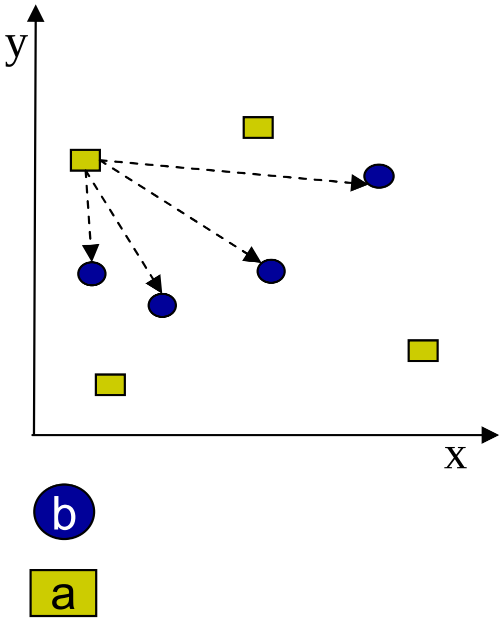
Mapping Computation to SPEs

- Coarse-Grained Data Parallelism
  - Each SPE contains all computation stages
  - Split up data and send to different SPEs



Example Data Parallelization on Cell

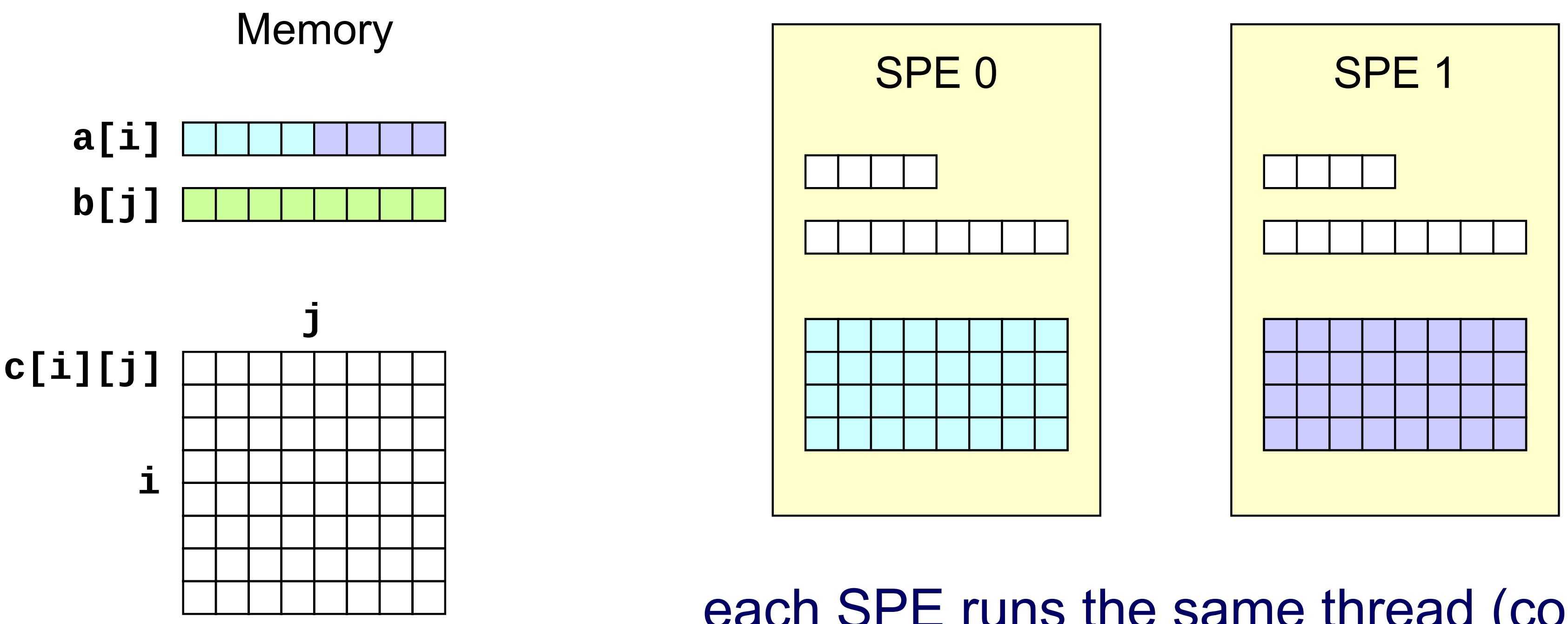
- Calculate distance from each point in a[...] to each point in b[...] and store result in c[...][...]



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

- How to divide the work between 2 SPEs?

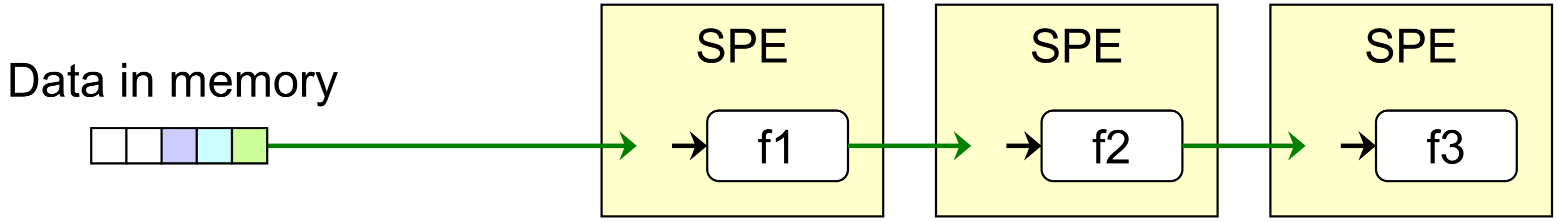
Example Data Parallelization on Cell



each SPE runs the same thread (code)

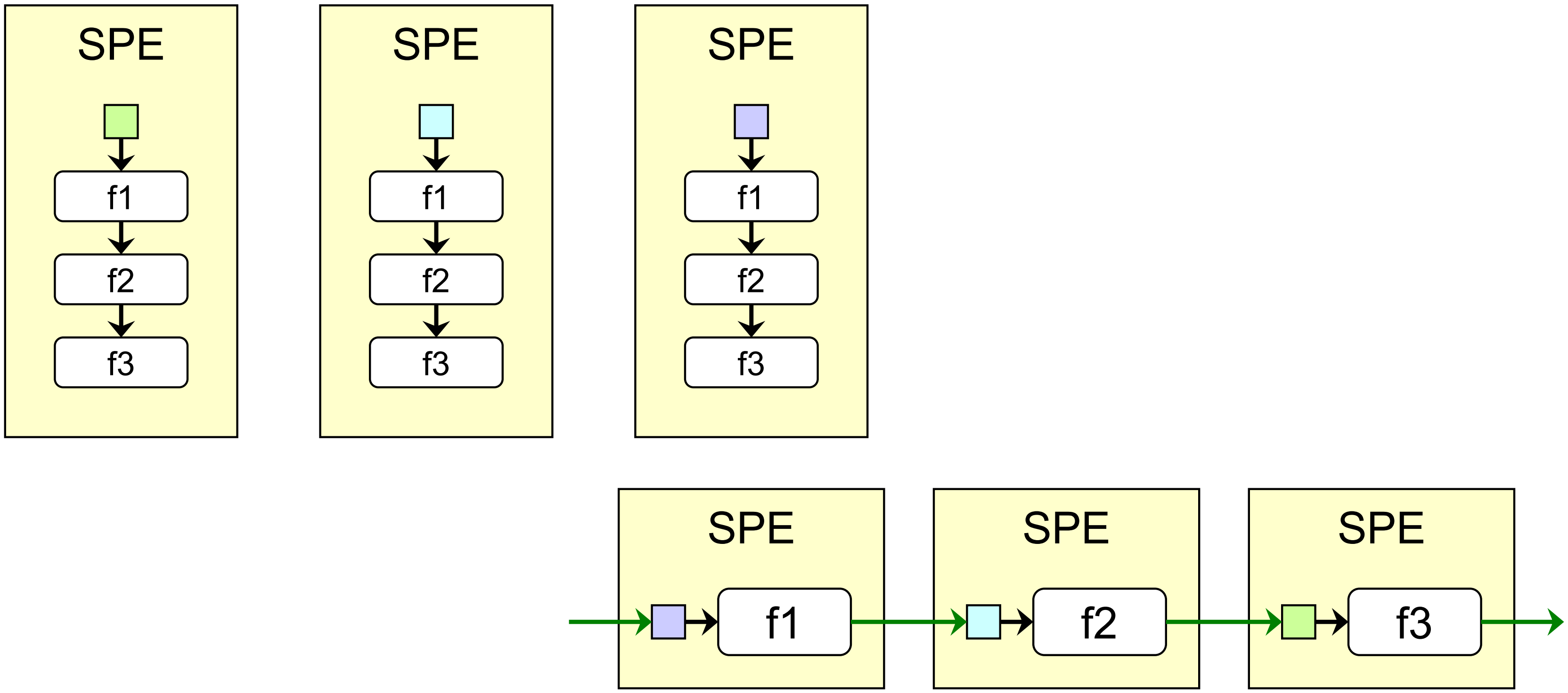
Mapping Computation to SPEs

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



Mapping Computation to SPEs

- Mixed or other approaches are possible, depends on problem
  - Pipeline parallelism when stateful computation is bottleneck
  - Or when locality is important
  - Data parallelism across most of the cores for simplicity



Increasing Performance with Parallelism

What's all the fuss about?

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

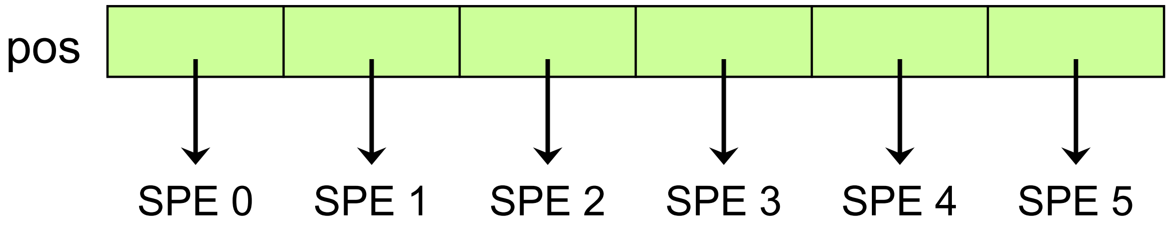
- For each step in simulation
  - Calculate acceleration of all objects
    - For each pair of objects, calculate the force between them and update accelerations accordingly
  - Update positions and velocities
- Slow: *n* = 3072 → 1500ms

```
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];
  }
}
```

```
VEC3D acc[NUM_BODIES] = 0;

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

Cell-ification: using SPEs for acceleration

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



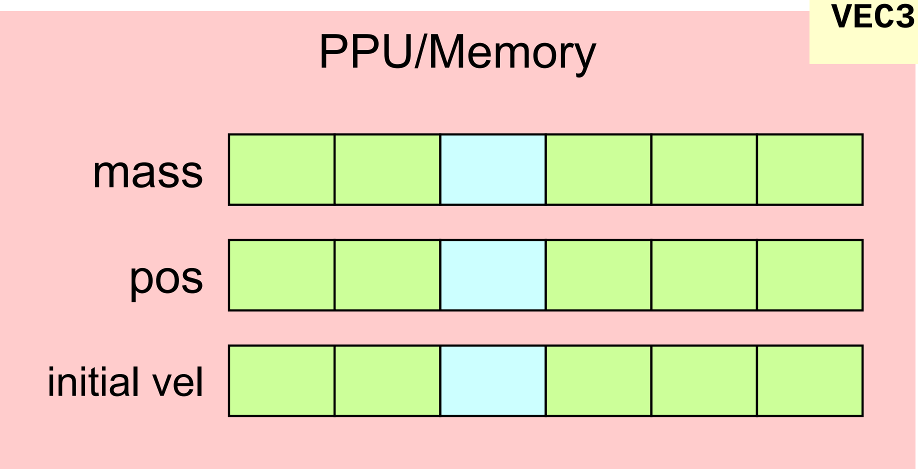
- Each SPE is responsible for calculating the motion of one section of objects
  - SPE still needs to know mass, position of all objects to compute accelerations
  - SPE only needs to know and update velocity of the objects it is responsible for
- Everything fits in local store
  - Positions for 3072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

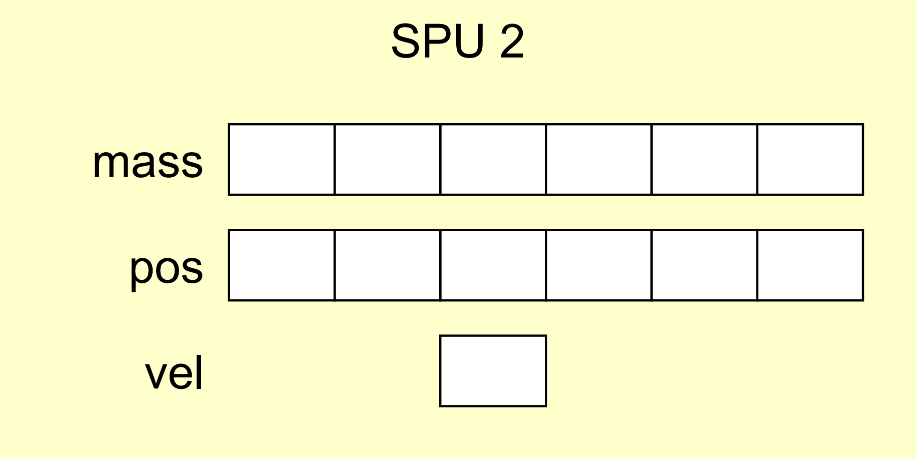
Cell-ification: using SPEs for acceleration

- 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];
```





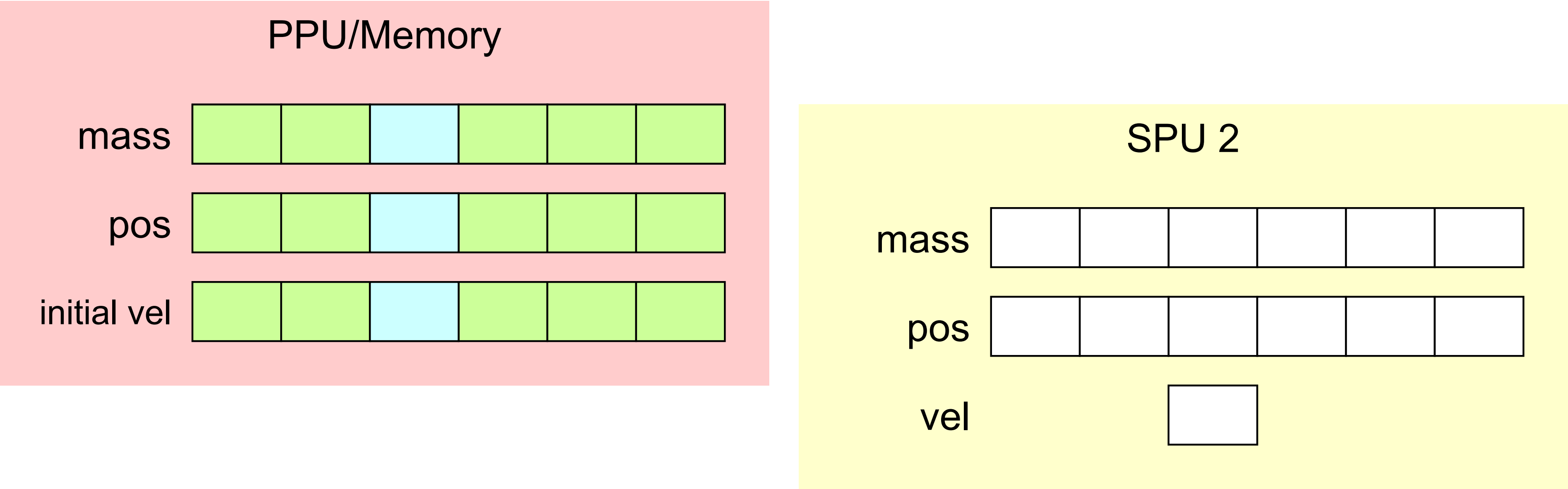
Cell-ification: using SPEs for acceleration

- SPU copies in mass of all objects

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

- SPU copies in initial position, velocity of its objects

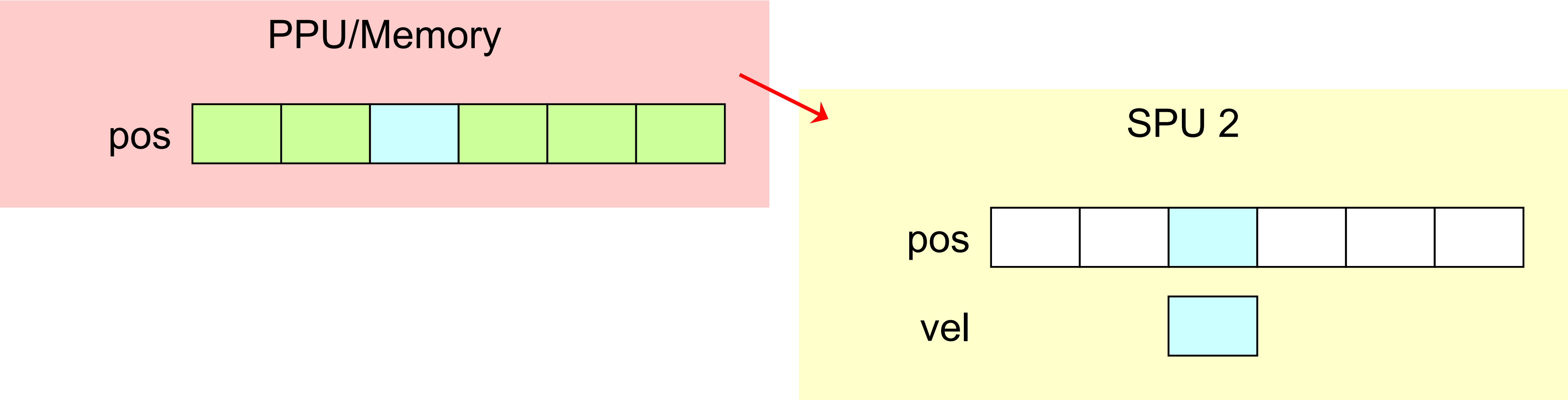
```
mfc_get(own_pos, cb.pos_addr + id * sizeof(pos[0]), sizeof(pos[0]), ...);
mfc_get(own_vel, cb.vel_addr + id * sizeof(own_vel), sizeof(own_vel), ...);
```



Cell-ification: using SPEs for acceleration

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

```
spu_read_in_mbox();
```



PPU/Memory

pos

SPU 2

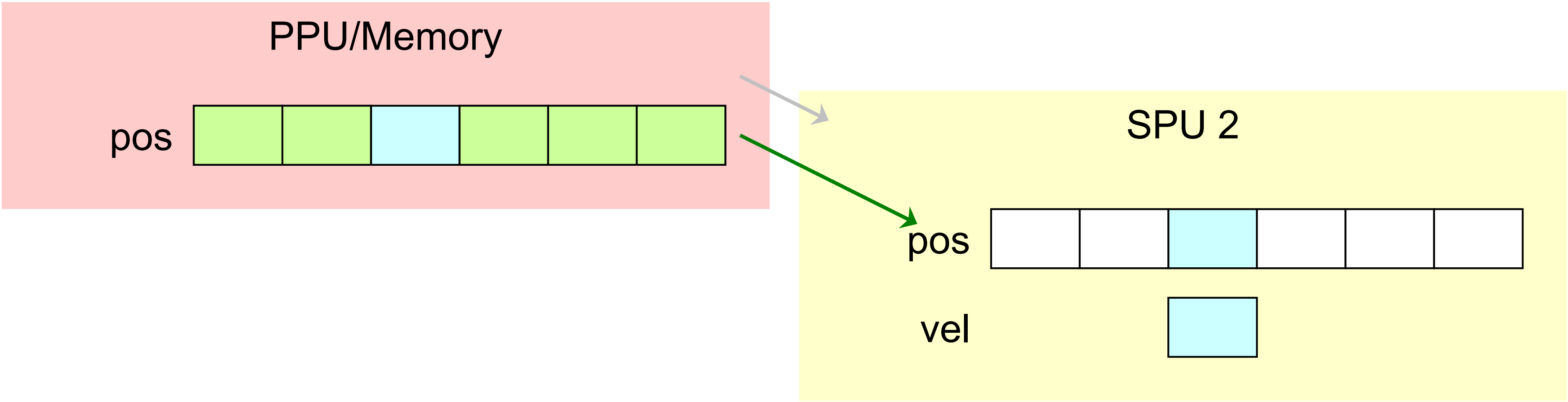
pos

vel

Cell-ification: using SPEs for acceleration

- SPU copies in updated positions of other objects

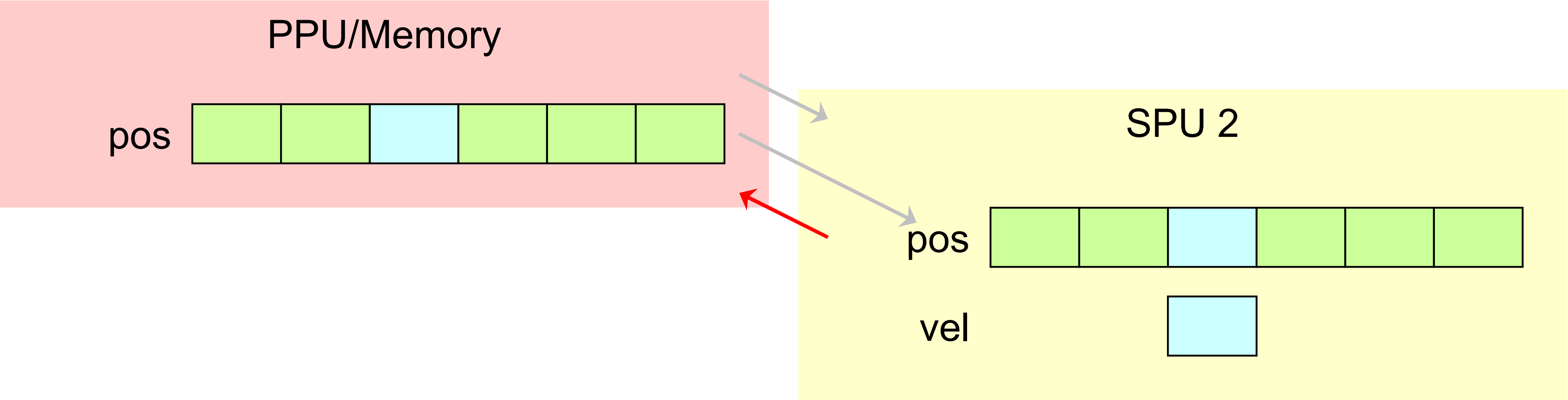
```
if (id != 0) {
  mfc_get(pos, cb.pos_addr + id * sizeof(pos[0]), id * sizeof(pos[0]), ...);
};
if (id != 5) {
  mfc_get(pos[id + 1], cb.pos_addr + (id + 1) * sizeof(pos[0]),
          (5 – id) * sizeof(pos[0]), ...);
}
```



Cell-ification: using SPEs for acceleration

- 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

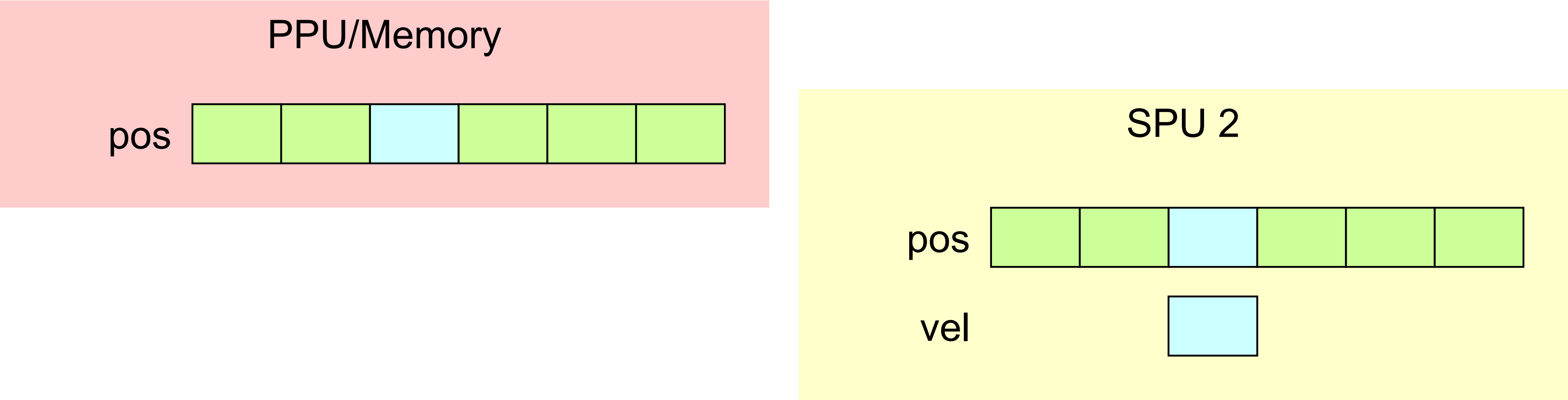
```
spu_write_out_mbox(0);
```



Cell-ification: using SPEs for acceleration

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

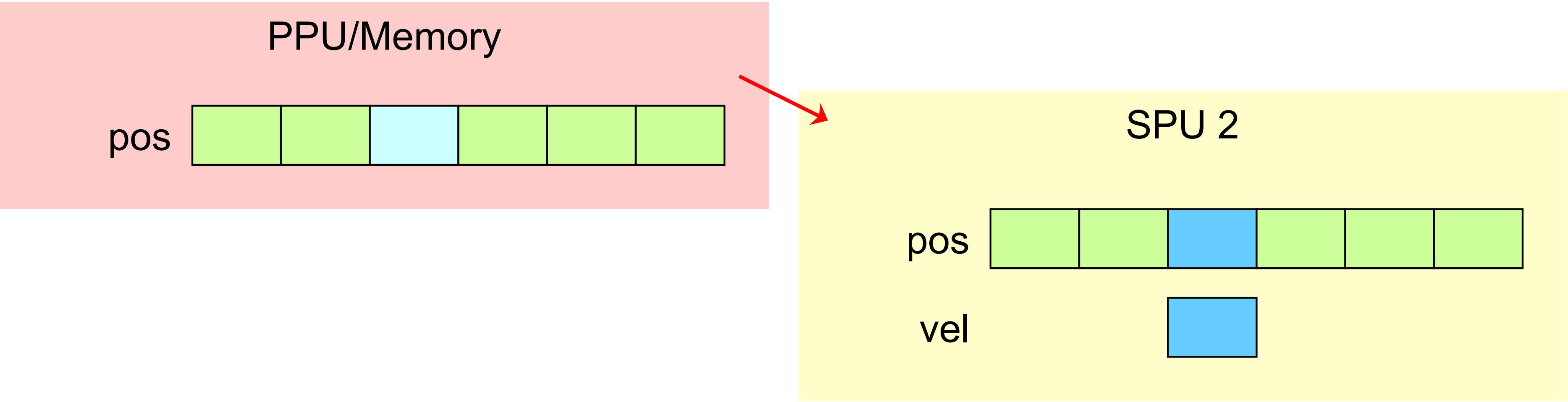
```
// Process interactions between this SPU's objects
process_own();
// Process interactions with other objects
for (int i = 0; i < 6; i++) {
  if (i != id) {
    process_other(pos[i], mass[i]);
  }
}
```



Cell-ification: using SPEs for acceleration

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

```
spu_read_in_mbox();
```



PPU/Memory

pos

SPU 2

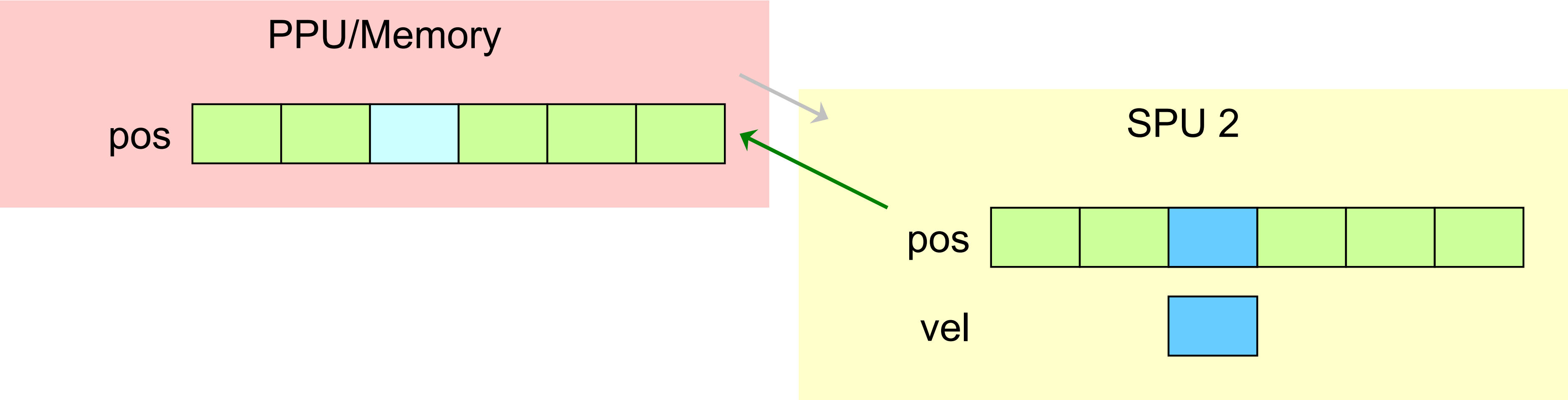
pos

vel

Cell-ification: using SPEs for acceleration

- SPU writes back updated positions to PPU

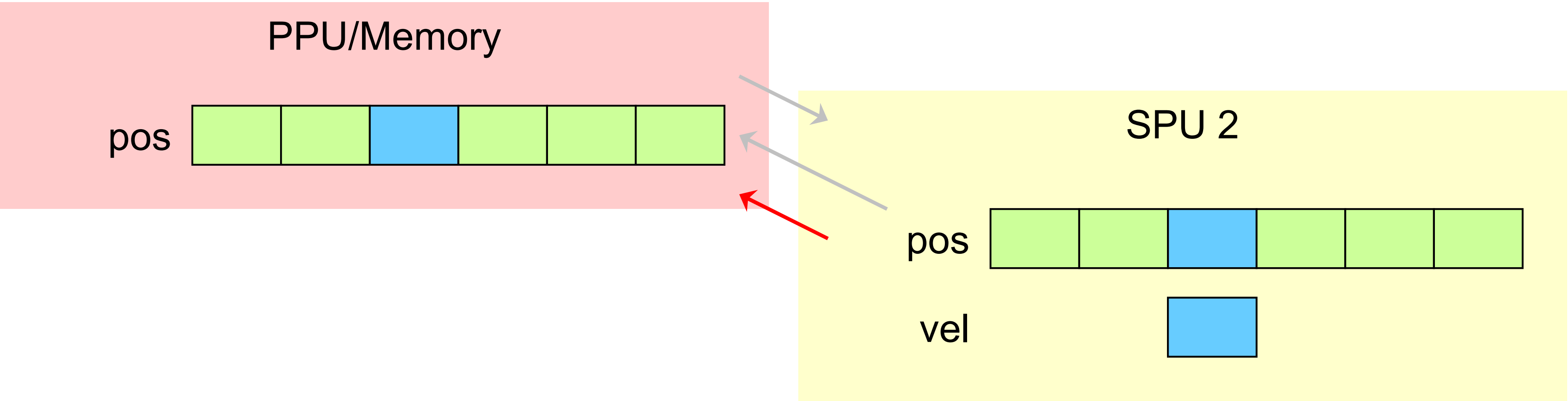
```
mfc_put(own_pos, cb.pos_addr + id * sizeof(pos[0]), sizeof(pos[0]), ...);
```



Cell-ification: using SPEs for acceleration

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

```
spu_write_out_mbox(0);
```



Coordination with Mailboxes and Signals

- Facility for SPU to exchange small messages with PPU/other SPUs
  - E.g. memory address, “data ready” message
- From perspective of SPU
  - 1 inbound mailbox (4-entry FIFO) – send messages to this SPU
  - 1 outbound mailbox (1-entry) – send messages from this SPU
  - 1 outbound mailbox (1-entry) – interrupts PPU to send messages from SPU
  - 2 signal notification registers – send messages to this SPU
  - 32 bits
- SPU accesses its own mailboxes/signals by reading/writing to channels with special instructions
  - Read from inbound mailbox, signals
  - Write to outbound mailboxes
  - Accesses will stall if empty/full
- SPU/PPU accesses another SPU mailboxes/signals through MMIO registers

Orchestration and Coordination

- Lots of signals sent back and force
  - I’m ready
  - I’m done
  - What’s my work?
  - Where’s my data?
  - …

- Couple this with architecture issues
  - Cell alignment constraints

- And a lot can go wrong

Cell Debugging Tools

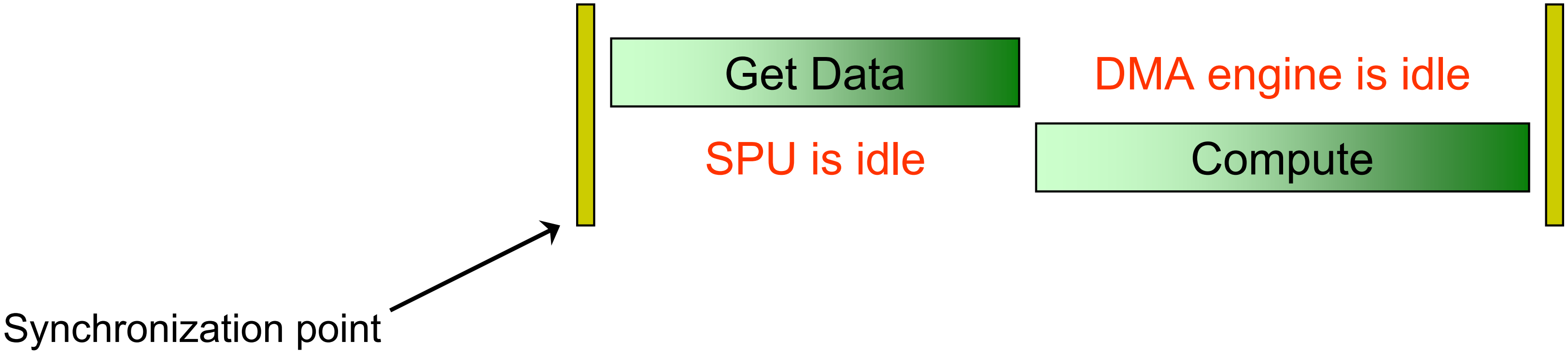
- GNU gdb source level debugger
  - Supports PPE and SPE multithreading
  - Interaction between PPE and SPE threads
  - Standalone SPE debugging
  - Or attach to SPE threads

- Existing methodologies for debugging are not well suited for multicores

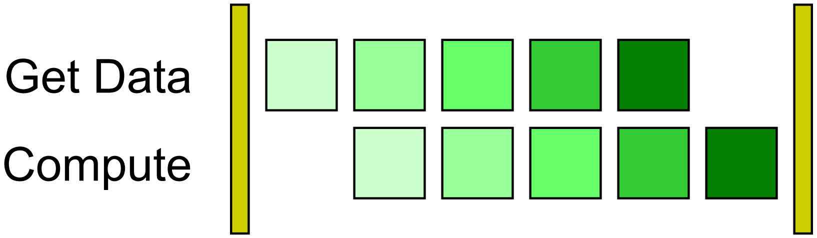
Overlapping Communication and Computation

Overlapping DMA and Computation

- Simple approach:

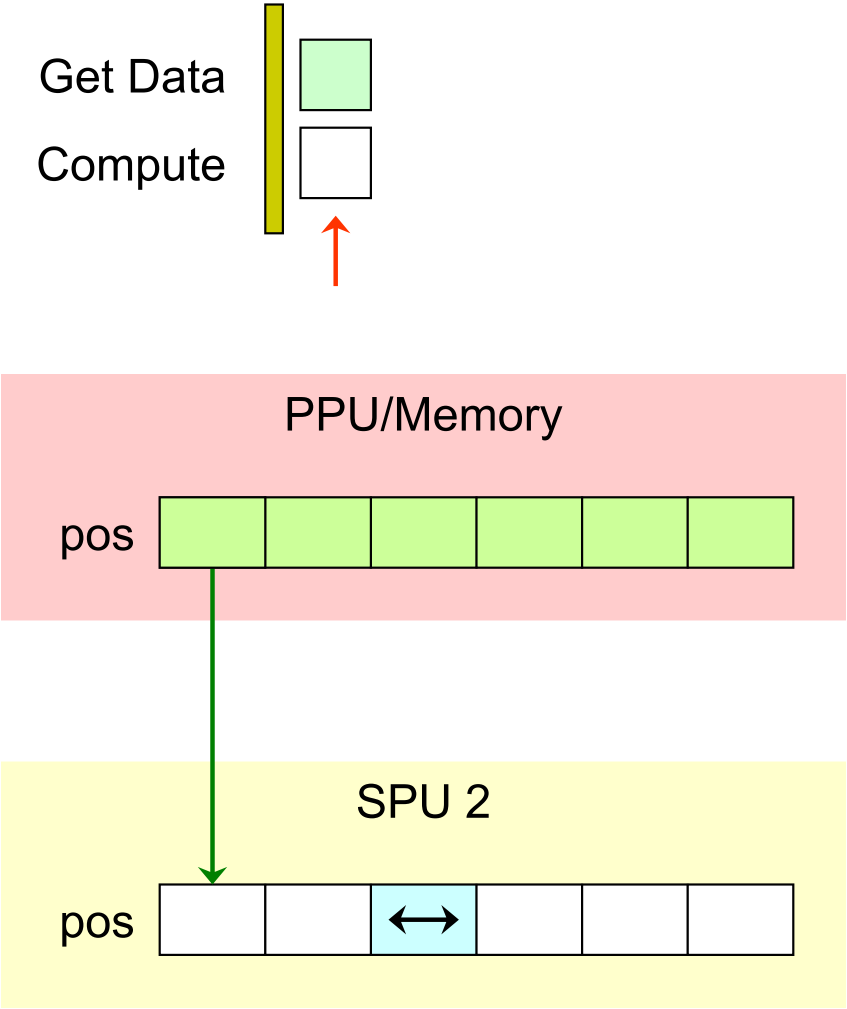


- Pipelining can achieve communication-computation concurrency
  - Start DMA for next piece of data while processing current piece



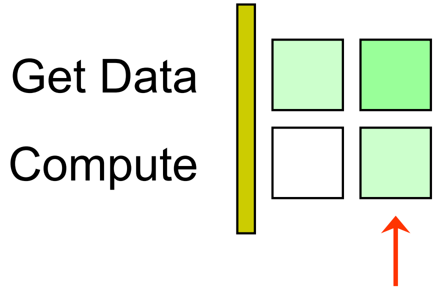
Overlapping DMA and Computation

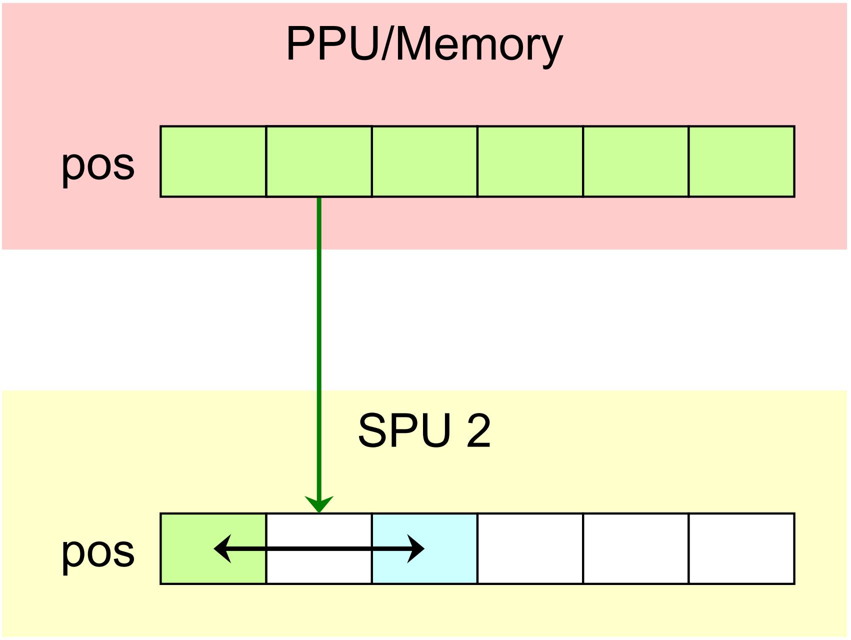
```
// pos[i] stores positions of objects SPU i is
// responsible for
VEC3D pos[6][SPU_BODIES];


// 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

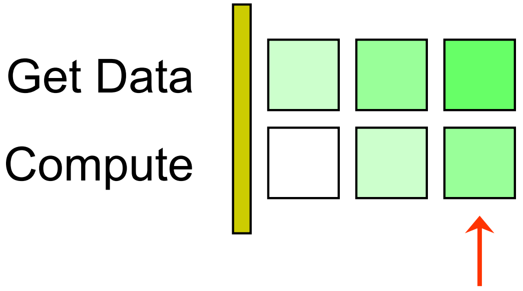
```
while (!done) {
  // Start transfer for next section of positions
  mfc_get(pos[next_i],
          cb.pos_addr + next_i * sizeof(pos[0]),
          sizeof(pos[0]),
          tag,
          ...);

  // 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;
}
```

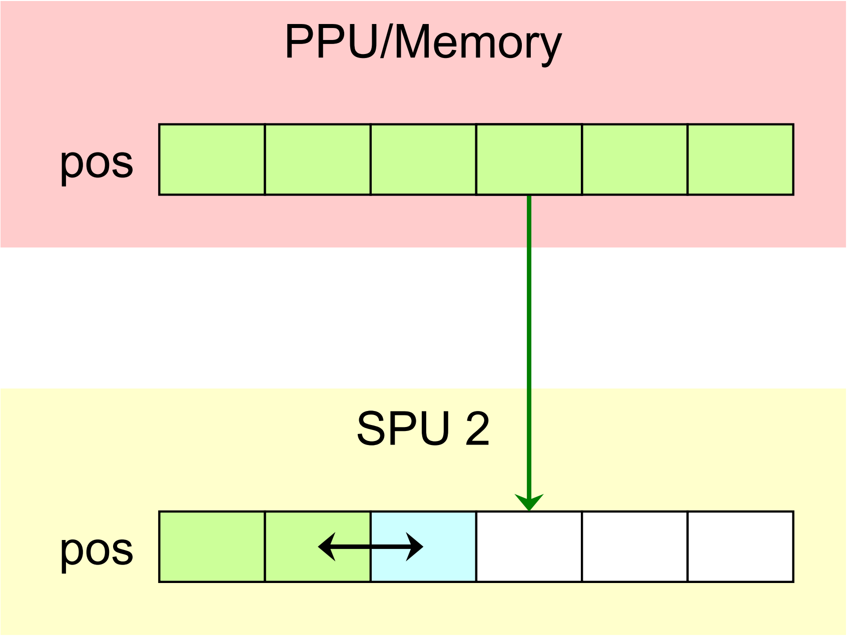




Overlapping DMA and Computation

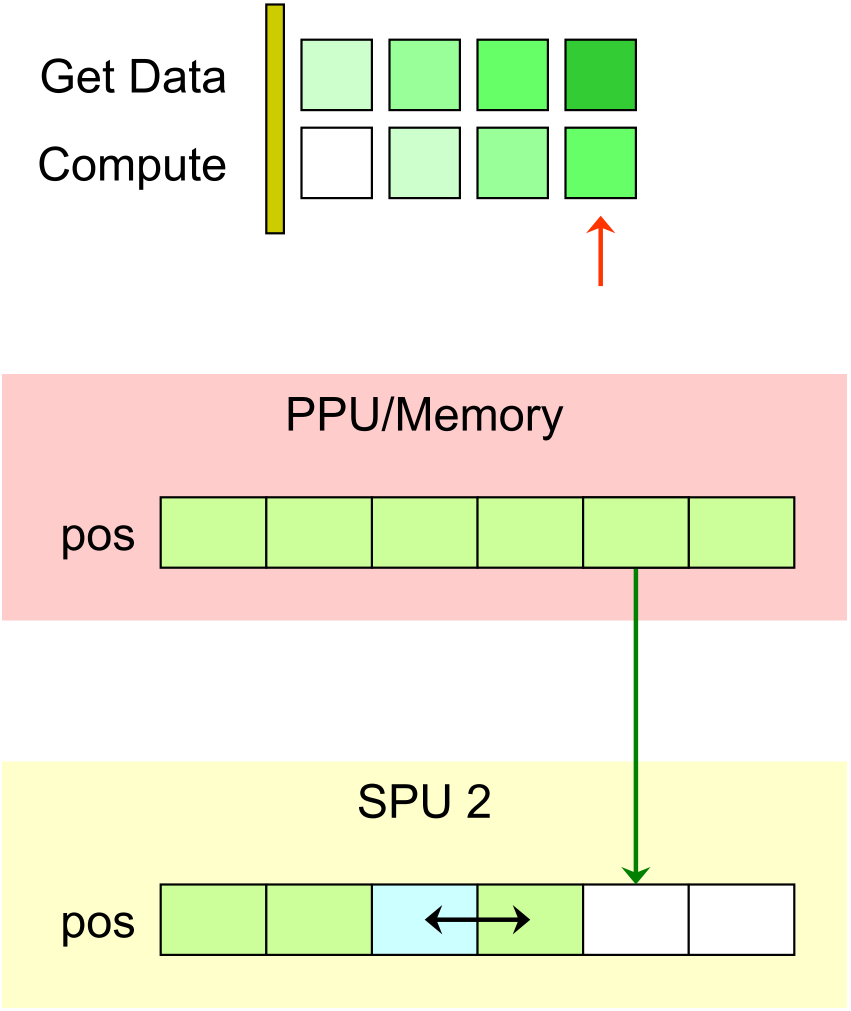
```
while (!done) {
  // Start transfer for next section of positions
  mfc_get(pos[next_i],
          cb.pos_addr + next_i * sizeof(pos[0]),
          sizeof(pos[0]),
          tag,
          ...);

  // 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;
}
```





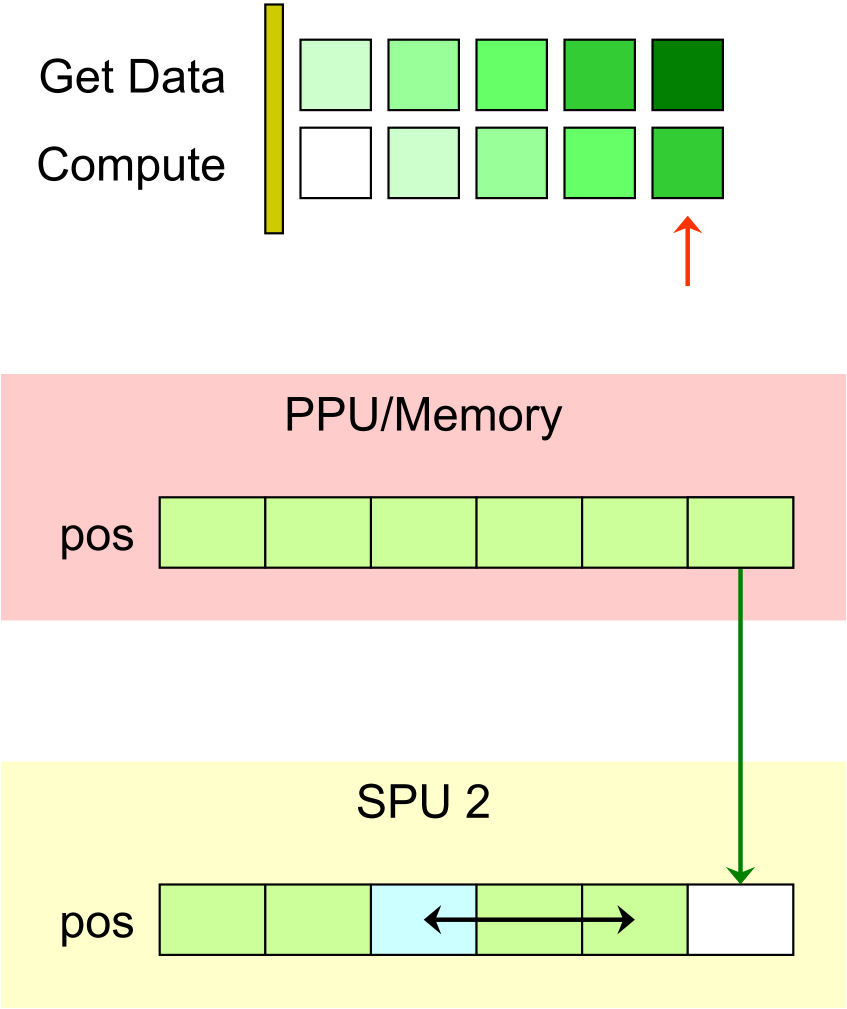
Overlapping DMA and Computation

```
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;
}
```



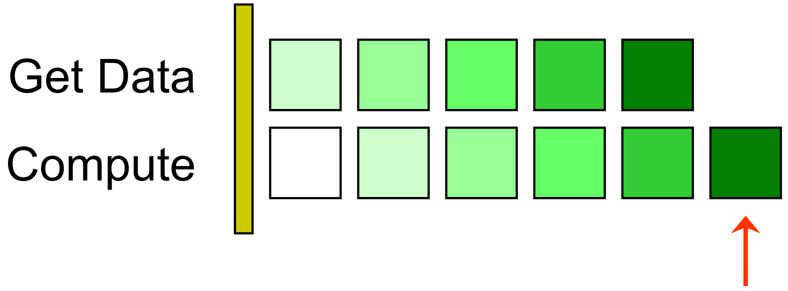
Overlapping DMA and Computation

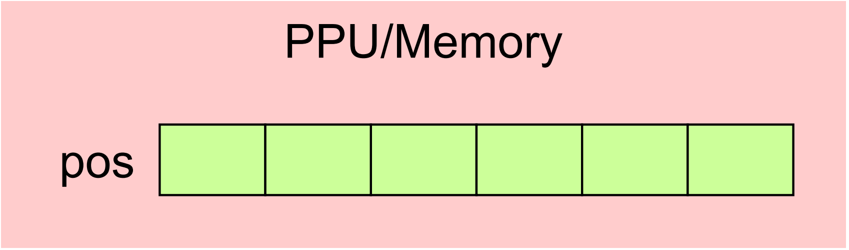
```
while (!done) {
  // Start transfer for next section of positions
  mfc_get(pos[next_i],
          cb.pos_addr + next_i * sizeof(pos[0]),
          sizeof(pos[0]),
          tag,
          ...);

  // 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;
}
```

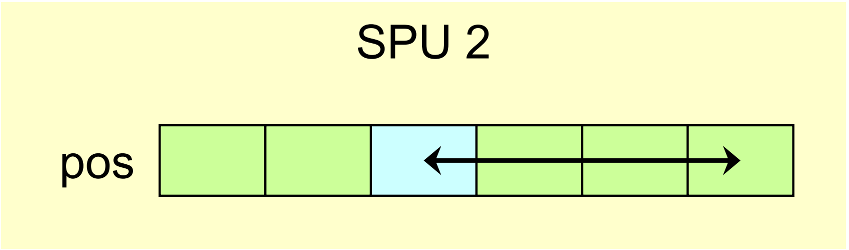


Overlapping DMA and Computation

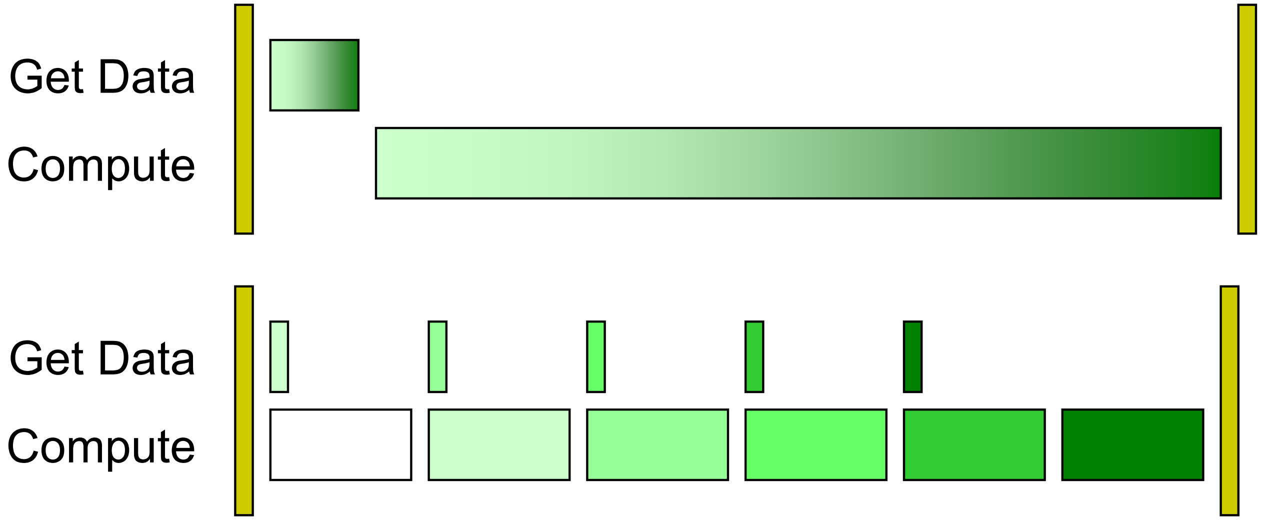
```
// Wait for last section of positions to finish
// transferring
tag ^= 1;
mfc_write_tag_mask(1 << tag);
mfc_read_tag_status_all();

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

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







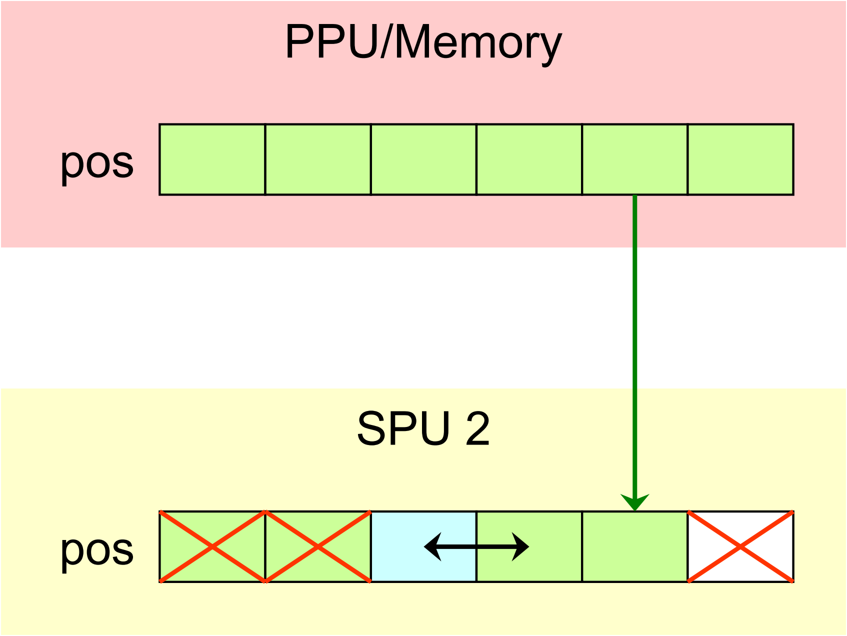
Overlapping DMA and Computation



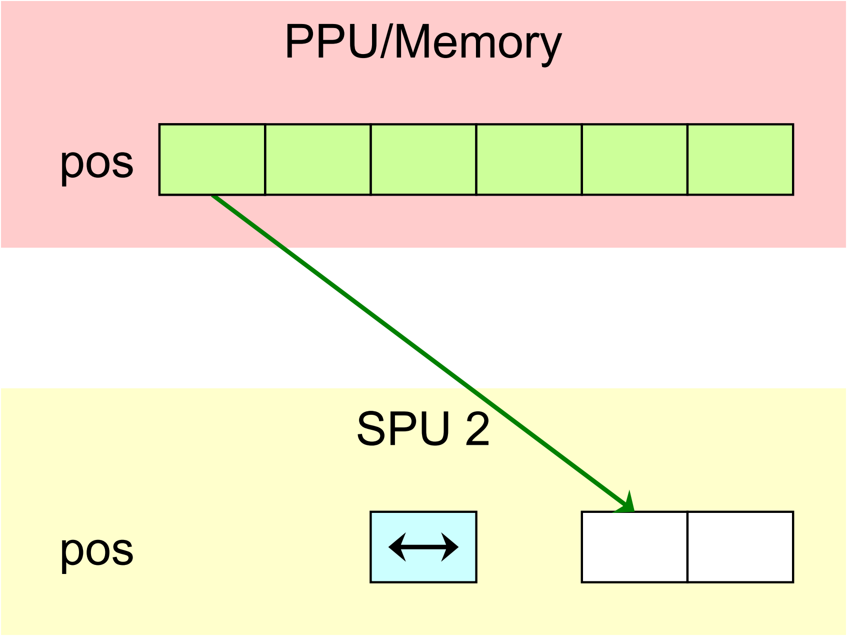
- Pipelining can improve performance by a lot, or not by much
  - Depends on program: communication to computation ratio
  - Can avoid optimizing parts that don’t greatly affect performance

Double-buffering

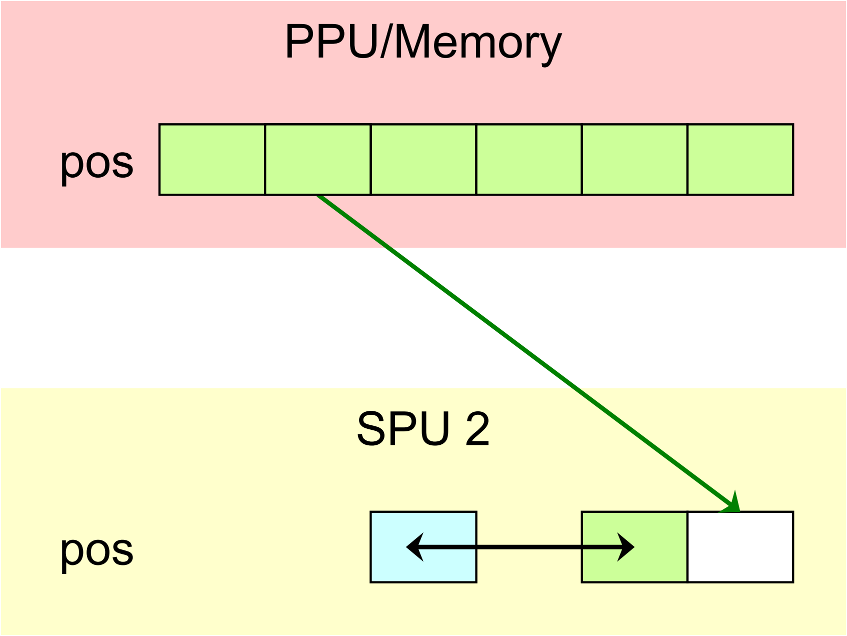
- LS is finite
- Avoid wasting local store space
- Keep 2 buffers
  - Start data transfer into one
  - Process data in other
  - Swap buffers for next transfer



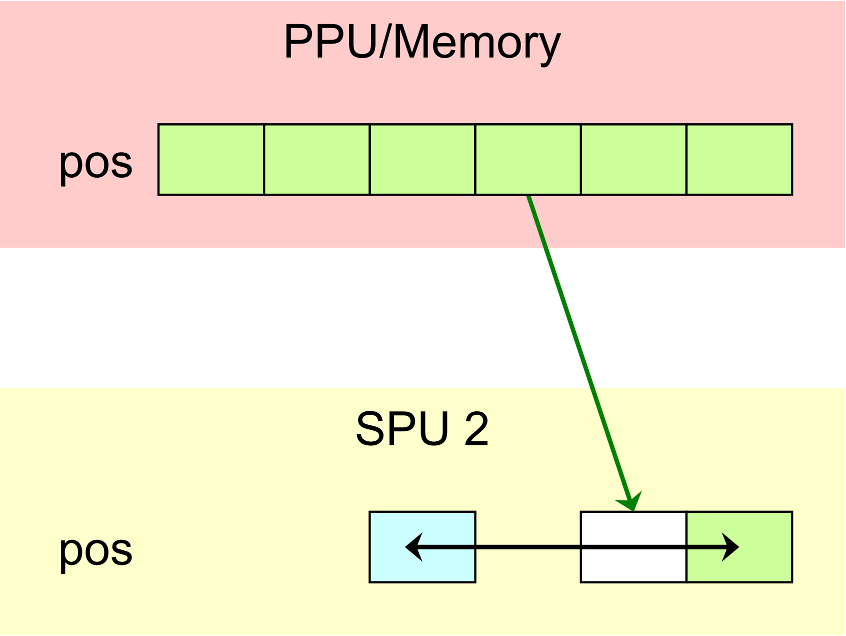
Double-buffering



Double-buffering



Double-buffering



Intra-Core Parallelism

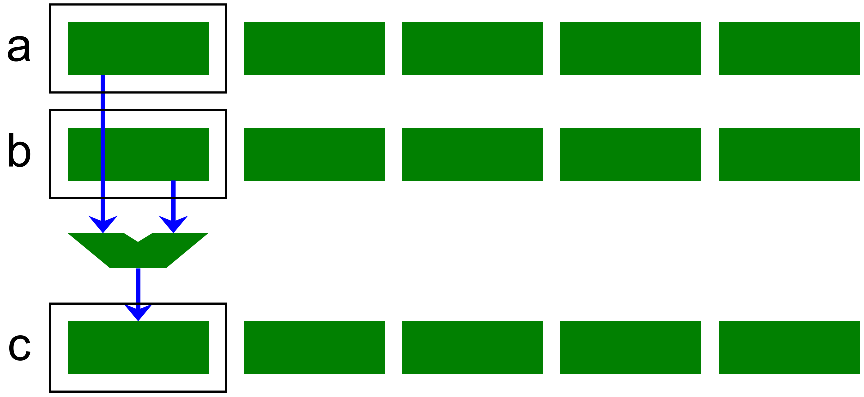
SIMD Programming on Cell

SIMD

- Many compute-bound applications perform the same computations on a lot of data
  - Dependence between iterations is rare
  - Opportunities for data parallelization

Scalar code

```
for (int i = 0; i < n; i++) {
  c[i] = a[i] + b[i]
}
```



SIMD

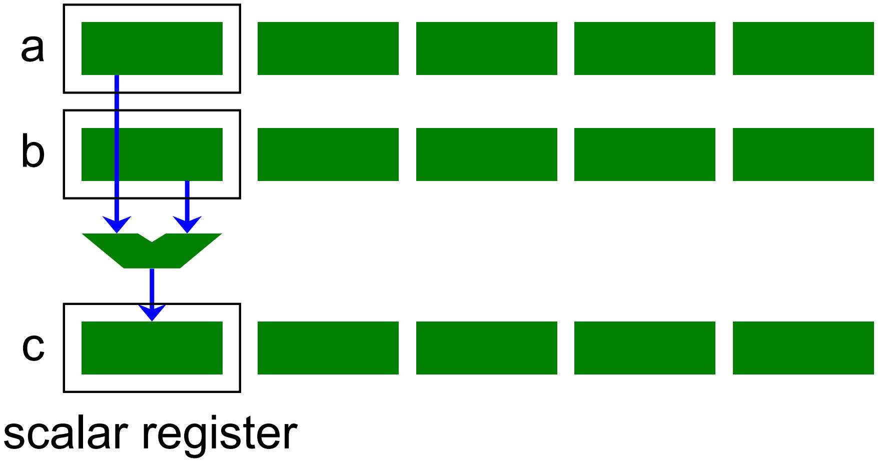
- Single Instruction, Multiple Data
- SIMD registers hold short vectors
- Instruction operates on all elements in SIMD register at once

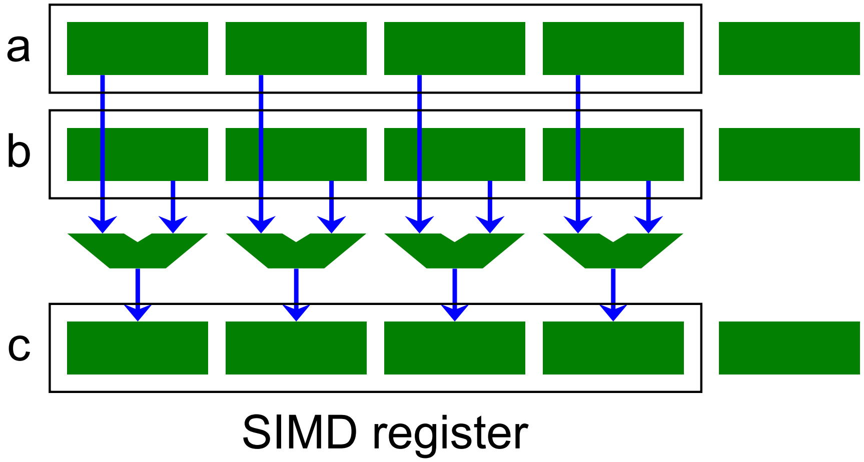
Scalar code

```
for (int i = 0; i < n; i++) {
  c[i] = a[i] + b[i]
}
```

Vector code

```
for (int i = 0; i < n; i += 4) {
  c[i:i+3] = a[i:i+3] + b[i:i+3]
}
```

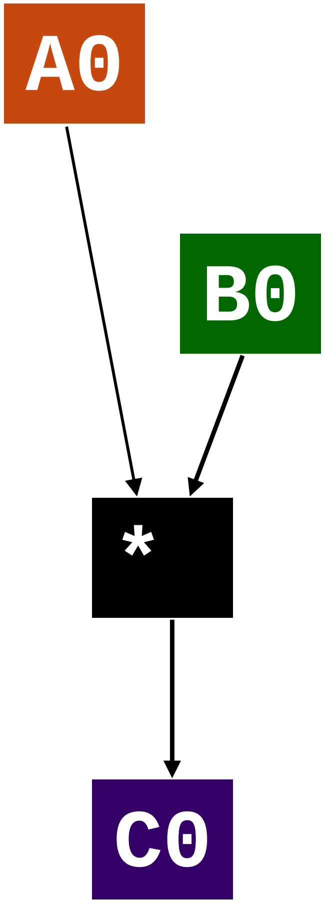




SIMD

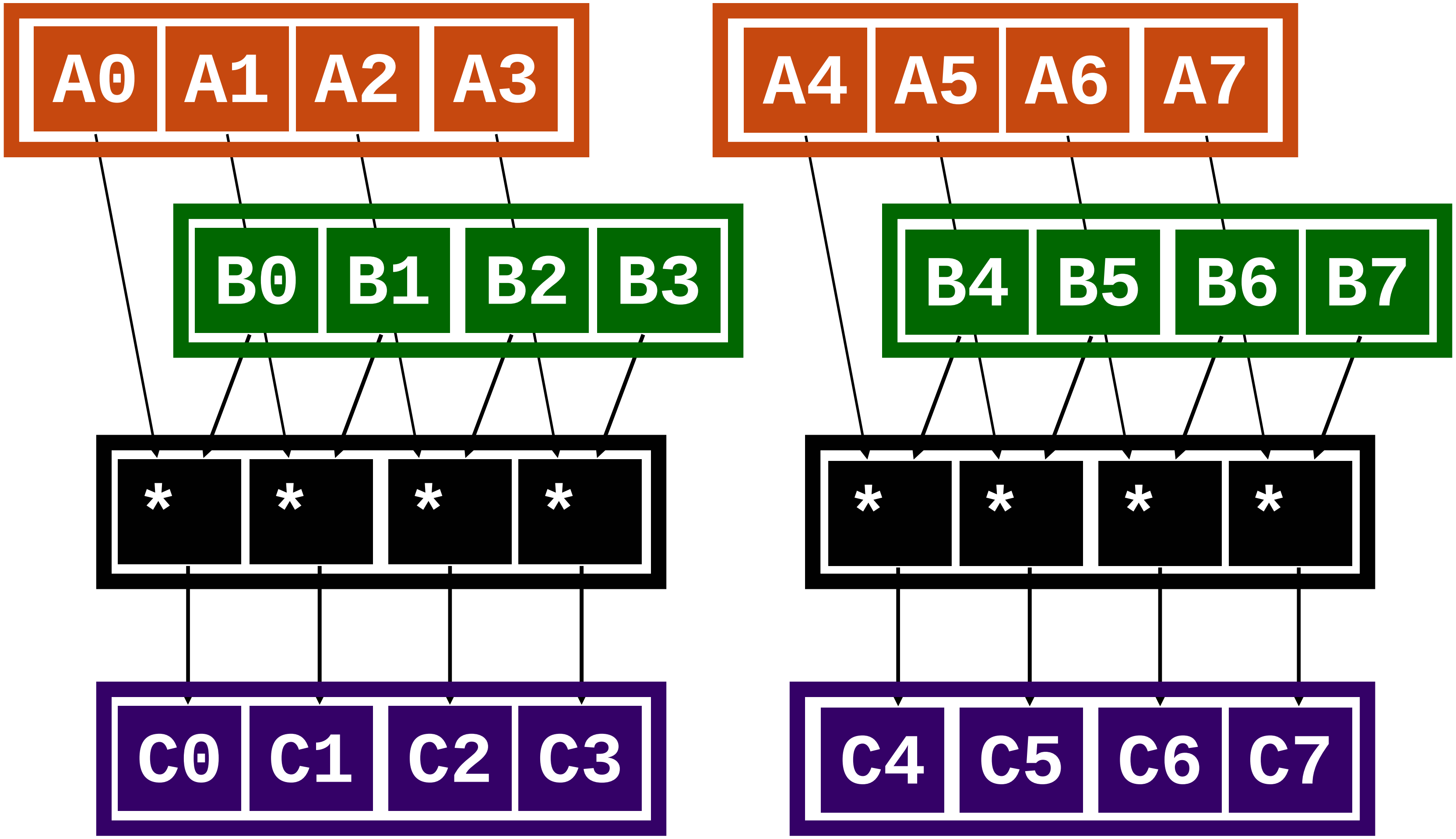
- Can offer high performance
  - Single-precision multiply-add instruction: 8 flops per cycle per SPE
- Scalar code works fine but only uses 1 element in vector
- SPU loads/stores on quad-word (qword) granularity only
  - Can be an issue if the SPU and other processors (via DMA) try to update different variables in the same qword
- For scalar code, compiler generates additional instructions to rotate scalar elements to the same slot and update a single element in a qword
- SIMDizing code is important
  - Auto SIMDization (compiler optimization)
  - Intrinsics (manual optimization)

Example: Scalar Operation



```
C[0] = A[0] * B[0]
```

Example: SIMD Vector Operation



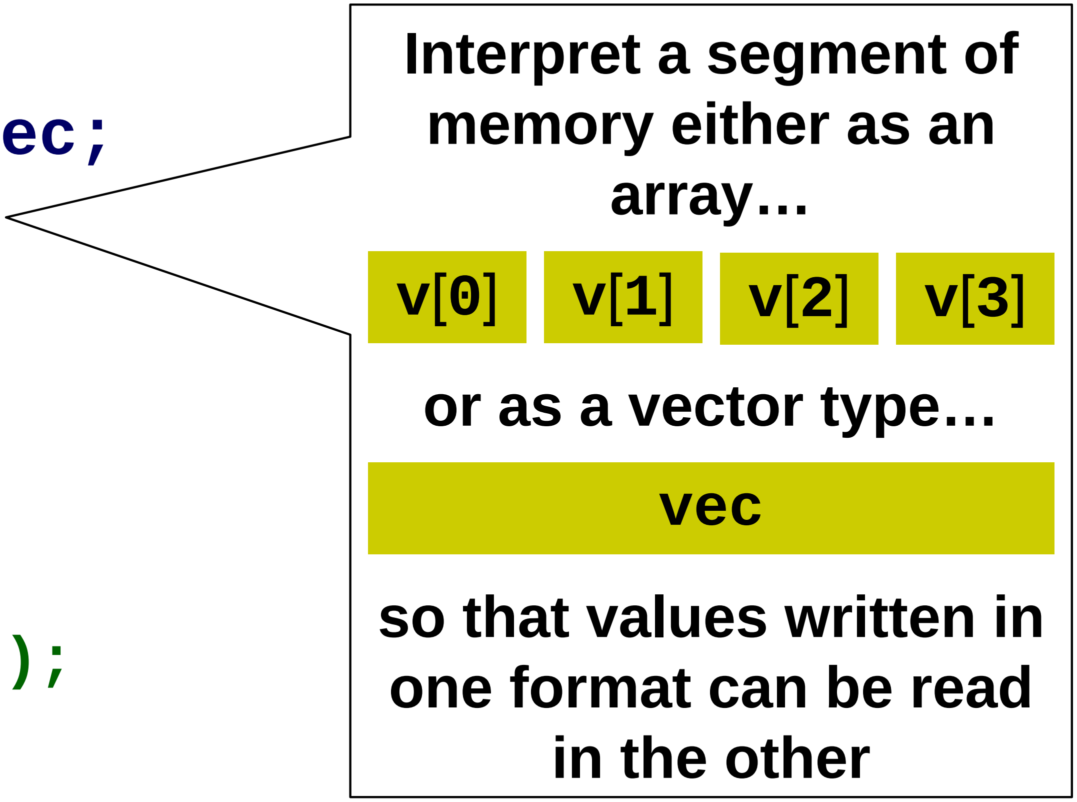
```
for(i = 0; i < N/4; ++i)
  C[i] = vector_mul(A[i],B[i]);
```

Hardware Support for Data Parallelism

- Registers are 128-bits
- Can pack vectors of different data types into registers
- Operations consume and produce vector registers
  - Special assembly instructions
  - Access via C/C++ language extensions (intrinsics)

Accessing Vector Elements

-
  ```
  typedef union {
     int v[4];
     vector signed int vec;
  } intVec;
  ```



Interpret a segment of memory either as an array…

or as a vector type…

so that values written in one format can be read in the other

- Unpack scalars from vector:

  -
    ```
    intVec a;
    a.vec = …;
    … = a.v[2];
    ```

  -
    ```
    … = spu_extract(va, 2);
    ```

- Pack scalars into vector:

  -
    ```
    a.v[0] = …; a.v[1] = …;
    a.v[2] = …; a.v[3] = …;
    … = a.vec;
    ```

Review: *3D Gravitational Simulator*

- *n* objects, each with mass, initial position, initial velocity

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

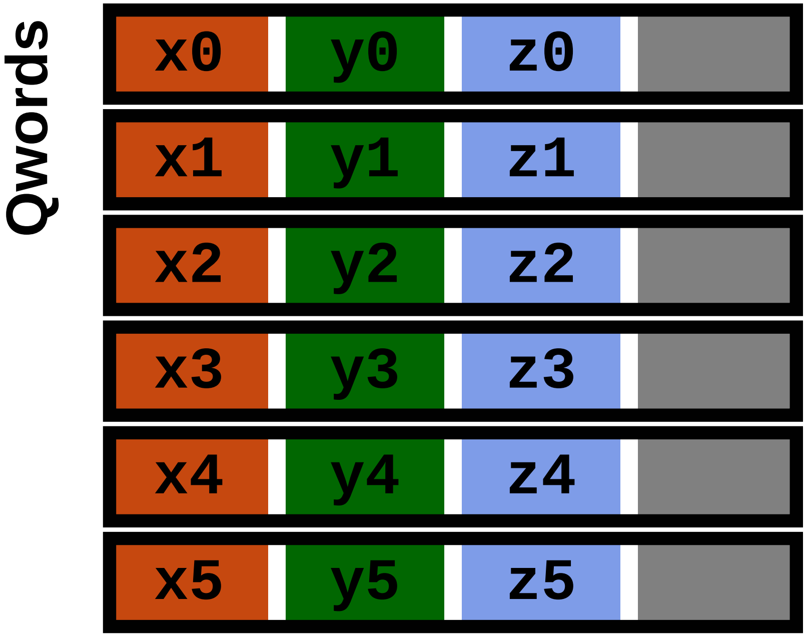
```
typedef struct _VEC3D {
  float x, y, z;
} VEC3D;
```

- Simulate motion using Euler integration
  - Calculate the force of each object on every other
  - Calculate net force on and acceleration of each object
  - Update position

```
VEC3D d;
// Calculate displacement from i to j
d.x = pos[j].x - pos[i].x;
d.y = pos[j].y - pos[i].y;
d.z = pos[j].z - pos[i].z;
```

Re-engineering for SIMD

- One approach to SIMD: array of structs
  - Pad each (x, y, z) vector to fill a qword
  - Components (x, y, z) correspond to first three words of vector float
  - Qwords for different vectors stored consecutively



x0	y0	z0	
x1	y1	z1	
x2	y2	z2	
x3	y3	z3	
x4	y4	z4	
x5	y5	z5	

```
typedef union _VEC3D {
  struct {float x, y, z;};
  vector float vec;
} QWORD_ALIGNED VEC3D;
```

Re-engineering for SIMD

- Now we can replace component-wise addition, subtraction, and multiplication with SIMD instructions

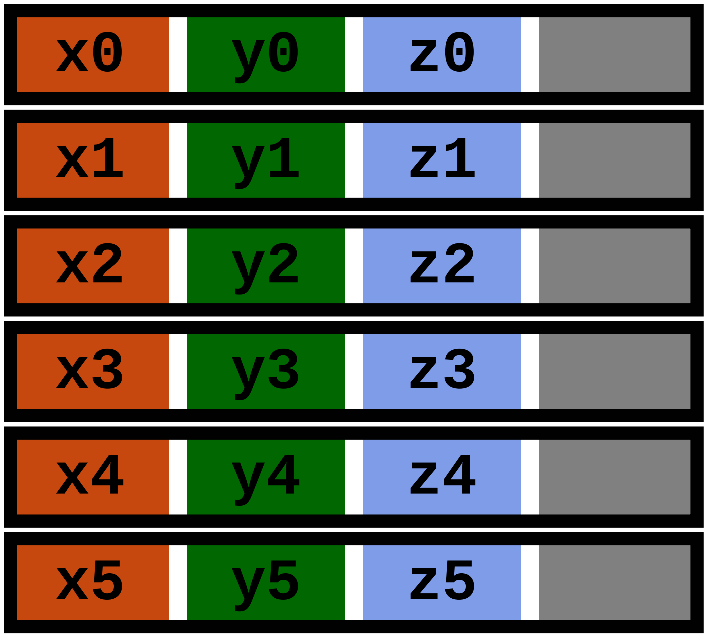
```
VEC3D d;
// Calculate displacement from i to j
d.x = pos[j].x - pos[i].x;
d.y = pos[j].y - pos[i].y;
d.z = pos[j].z - pos[i].z;
```

```
vector float d;
// Calculate displacement from i to j
d = spu_sub(pos[j].vec, pos[i].vec);
```

SIMD Design Considerations

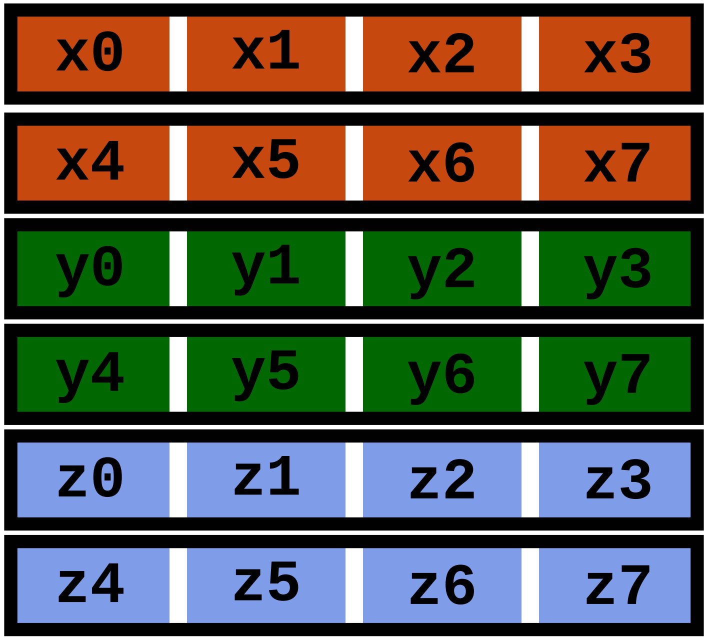
- Data layout: array of structs (AOS) vs. struct of arrays (SOA)
  - SOA layout is alternative data organization to lay out the same fields consecutively
  - Can apply different algorithms on new data layout

array of structs



x0	y0	z0	
x1	y1	z1	
x2	y2	z2	
x3	y3	z3	
x4	y4	z4	
x5	y5	z5	

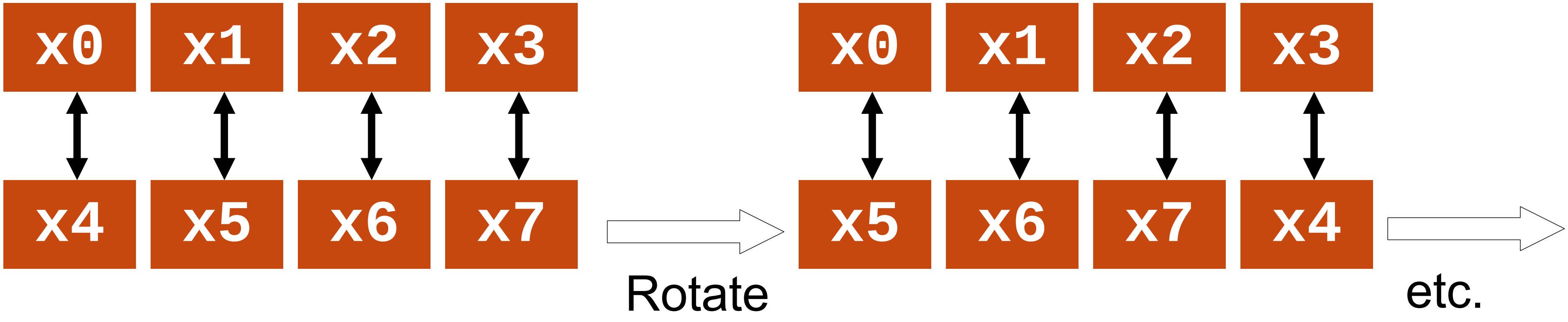
struct of arrays



x0	x1	x2	x3
x4	x5	x6	x7
y0	y1	y2	y3
y4	y5	y6	y7
z0	z1	z2	z3
z4	z5	z6	z7

Struct of Array Layout

- Need 12 qwords to store state for 8 objects
  - **x**, **y**, **z** position and velocity components
  - No padding component needed in SOA
- For each component, do four pair-interactions at once with SIMD instructions
  - Rotate qword 3 more times to get all 16 pair-interactions between two qwords



Performance Summary for Example

- Baseline native code was sequential and scalar
  - Scalar (PPU): 1510 ms
- Parallelized code with double buffering for SPUs
  - Scalar (6 SPUs): 420 ms
- Applied SIMD optimizations
  - SIMD array of structs: 300 ms
- Redesigned algorithm to better suite SIMD parallelism
  - SIMD struct of arrays: 80 ms

- Overall speedup compared to native sequential execution
  - Expected: ~ 24x (6 SPUs ∗ 4 way SIMD)
  - Achieved: 18x*

* Note comparison is PPU to 6 SPUs

Programming the Cell

- Guide to programming PS3/Cell: google “PS3 programming”
  - http://cag.csail.mit.edu/ps3
  - MIT short course on parallel programming using the PS3/Cell as the student project platform
  - Provides detailed examples with walk through
    - Lectures, recitations, and labs
  - Student projects and source code
  - Lots of recipes (installing Linux, SDK, Cell API mini-reference)
  - Links to additional documentation

Cell Programming Summary

- Programming multicore architectures: “parallelize or perish”
- Orchestrating parallelism is hard
  - Data management
  - Code placement
  - Scheduling
  - Hiding communication latency
- Lots of opportunities for impact
  - Scheduling ideas
  - Dynamic load balancing
  - Static scheduling
  - Intra-core performance still matters
- Cell offers a unique platform to explore and evaluate lots of ideas, PS3s make it easily accessible

Tutorial Agenda

- Brief overview of Cell performance characteristics

- Programming Cell
  - Cell components
  - Application walk through
  - Inter-core parallelism: structuring computation and communication
  - Orchestration: synchronization mechanisms
  - SIMD for single thread performance: it still matters

- Opportunities for research and innovation, and education
  - Programming Language
  - Parallelizing Compiler
  - Abstract Streaming Layer

Acknowledgements

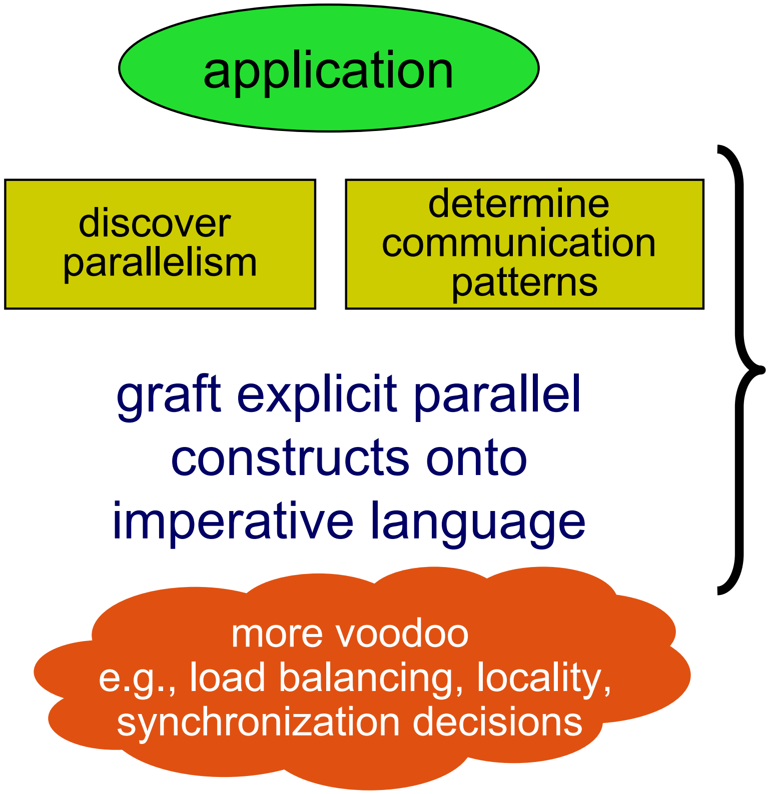
StreamIt group at MIT, especially

- Prof. Saman Amarasinghe
- Michael Gordon (PhD)
- Bill Thies (PhD)
- Qiuyuan Jimmy Li (MEng)
- David Zhang (MEng)

CCCP group at UMICH, especially

- Prof. Scott Mahlke
- Manjunath Kudlur (PhD)

Programming For Parallelism



application

discover parallelism

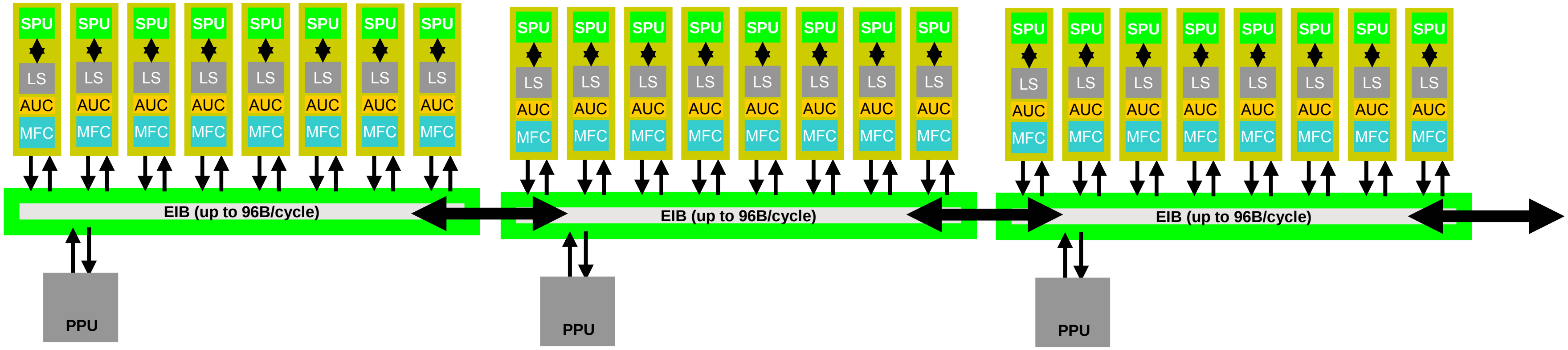
determine communication patterns

graft explicit parallel constructs onto imperative language

more voodoo
e.g., load balancing, locality, synchronization decisions

- Huge burden on programmer
  - Introducing parallelism
  - Correctness of parallelism
  - Optimizing parallelism

- Is implementation composable or malleable?



Explicit Parallelism

- Programmer controls details of parallelism!
- Granularity decisions:
  - If too small, lots of synchronization and thread creation
  - If too large, bad locality
- Load balancing decisions
  - Create balanced parallel sections (not data-parallel)
- Locality decisions
  - Sharing and communication structure
- Synchronization decisions
  - barriers, atomicity, critical sections, order, flushing
- For mass adoption, we need a better paradigm:
  - Where the parallelism is natural
  - Exposes the necessary information to the compiler

Common Machine Language

- Represent common properties of architectures
    - Necessary for performance
- Abstract away differences in architectures
    - Necessary for portability
- Cannot be too complex
    - Must keep in mind the typical programmer
- C and Fortran were the common machine languages for uniprocessors

Common Machine Languages

Uniprocessors:

Common Properties
Single flow of control
Single memory image

Differences:
Register File
ISA
Functional Units

von-Neumann languages represent the common properties and abstract away the differences

Multicores:

Common Properties
Multiple flows of control
Multiple local memories

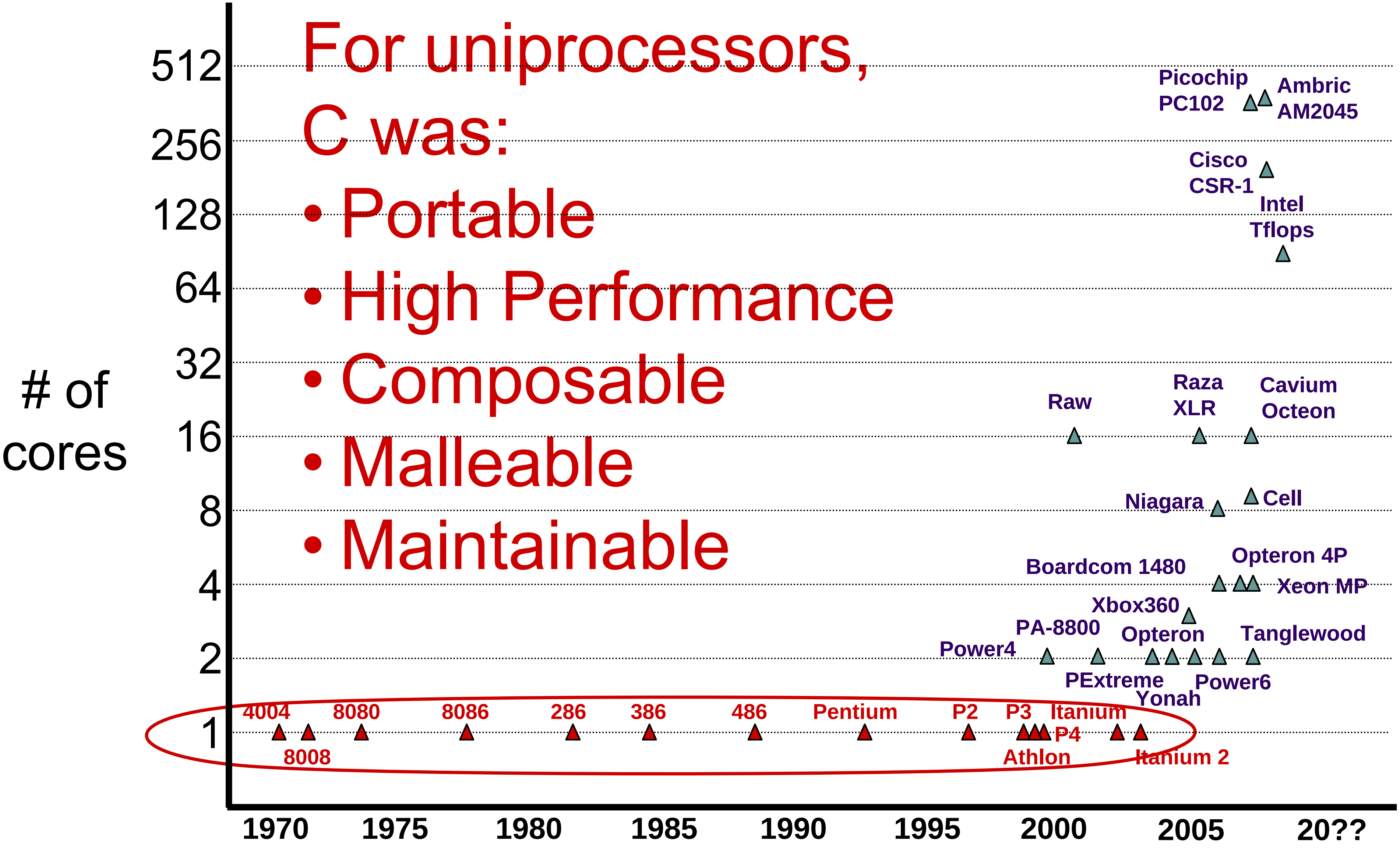
Differences:
Number and capabilities of cores
Communication Model
Synchronization Model

Need common machine language(s) for multicores

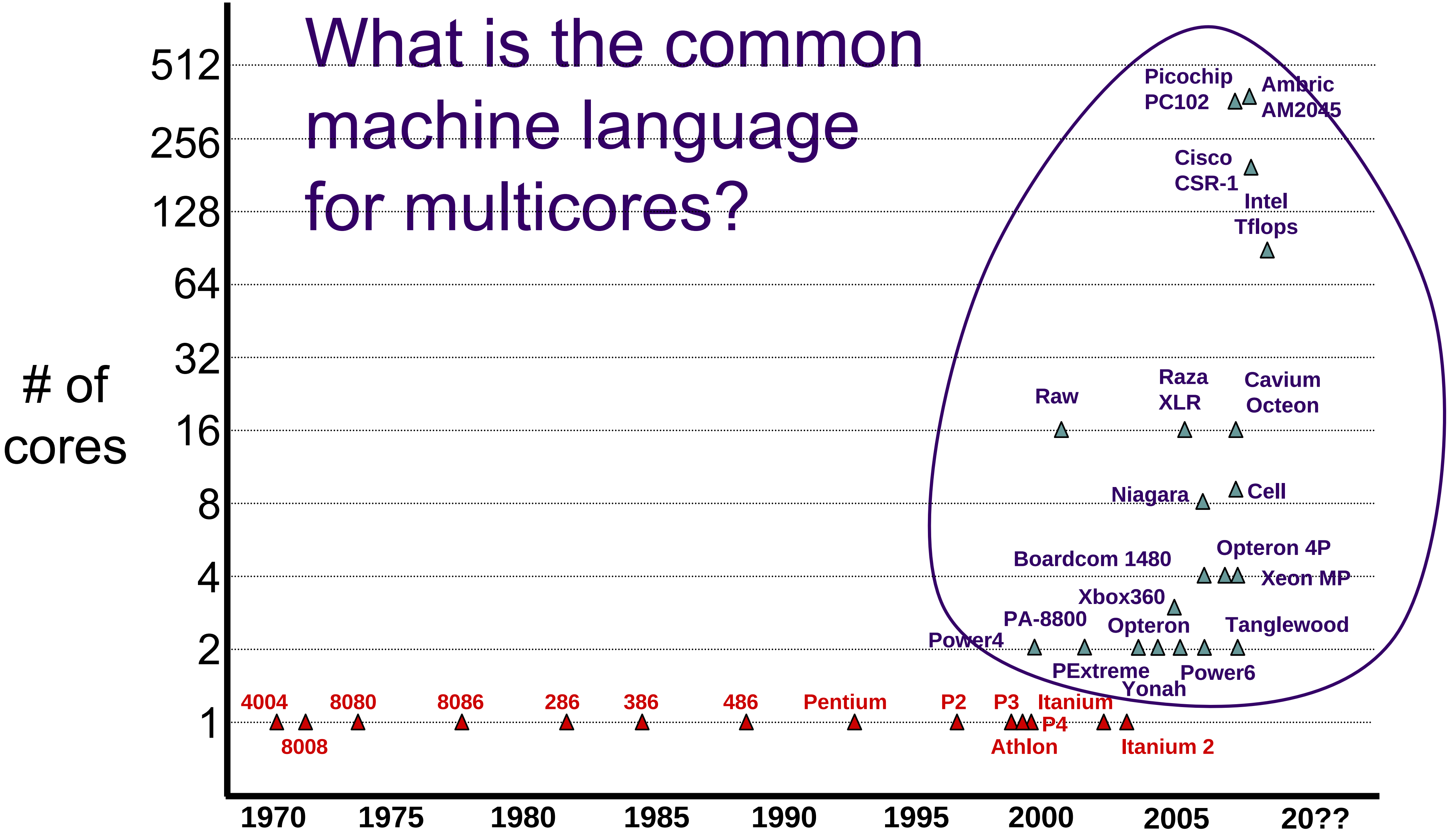
Why a New Language?

For uniprocessors, C was:

- Portable
- High Performance
- Composable
- Malleable
- Maintainable



Why a New Language?



What is the common machine language for multicores?

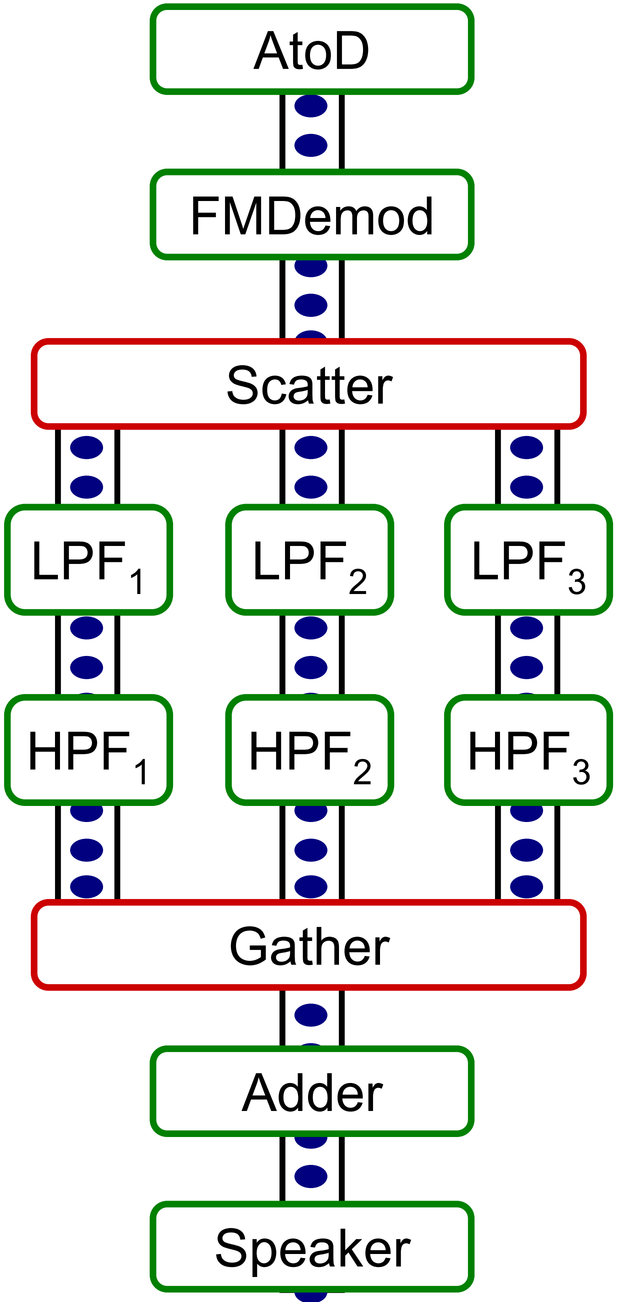
Unburden the Programmer

- Move hard decisions to compiler!
  - Granularity
  - Load Balancing
  - Locality
  - Synchronization

- Hard to do in traditional languages: can a novel language help?

Streaming as a Common Machine Language

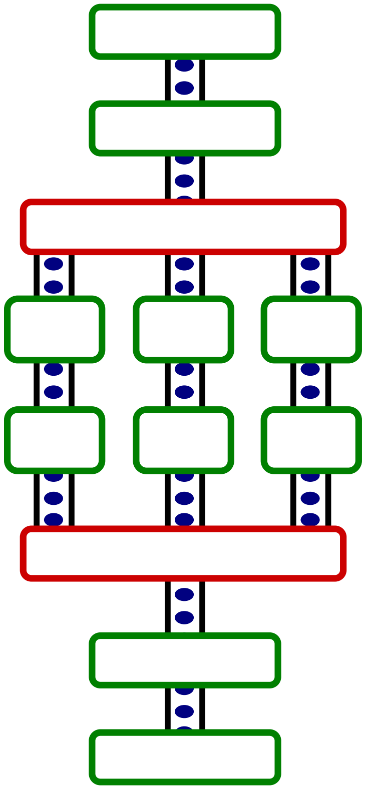
- For programs based on streams of data
  - Audio, video, DSP, networking, security (cryptography), etc.
  - Examples: HDTV editing, radar tracking, microphone arrays, cell phone base stations, graphics
- Several attractive properties
  - Regular and repeating computation
  - Independent filters with explicit communication
  - Task, data, and pipeline parallelism
- Benefits:
  - Naturally parallel
  - Expose dependencies to compiler
  - Enable powerful transformations



Streaming Models of Computation

- Many different ways to represent streaming
  - Do senders/receivers block?
  - How much buffering is allowed on channels?
  - Is computation deterministic?
  - Can you avoid deadlock?

- Three common models:
  1. Kahn Process Networks
  2. Synchronous Dataflow
  3. Communicating Sequential Processes



Streaming Models of Computation

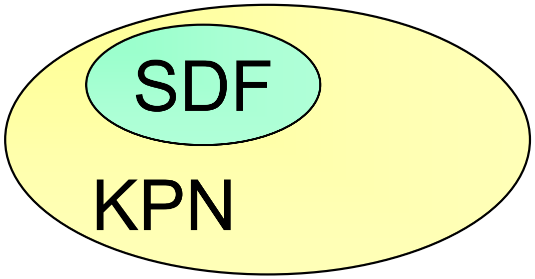
	Communication Pattern	Buffering	Notes

Streaming Models of Computation

	Communication Pattern	Buffering	Notes
Kahn process networks (KPN)	Data-dependent, but deterministic	Conceptually unbounded	- UNIX pipes - Ambric (startup)

Streaming Models of Computation

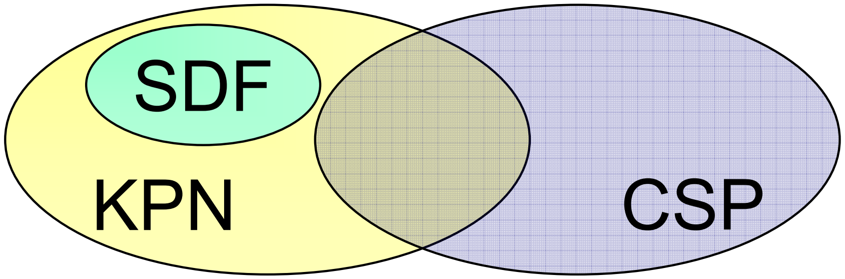
	Communication Pattern	**Buffering**	**Notes**
Kahn process networks (KPN)	Data-dependent, but deterministic	Conceptually unbounded	- UNIX pipes - Ambric (startup)
Synchronous dataflow (SDF)	Static	Fixed by compiler	- Static scheduling - Deadlock freedom



space of program behaviors

Streaming Models of Computation

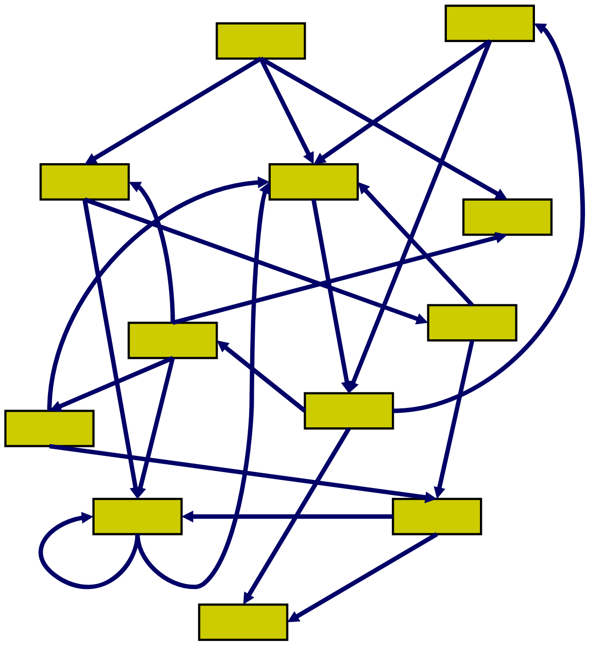
	Communication Pattern	Buffering	Notes
Kahn process networks (KPN)	Data-dependent, but deterministic	Conceptually unbounded	- UNIX pipes - Ambric (startup)
Synchronous dataflow (SDF)	Static	Fixed by compiler	- Static scheduling - Deadlock freedom
Communicating Sequential Processes (CSP)	Data-dependent, allows non-determinism	None (Rendesvouz)	- Rich synchronization primitives - Occam language



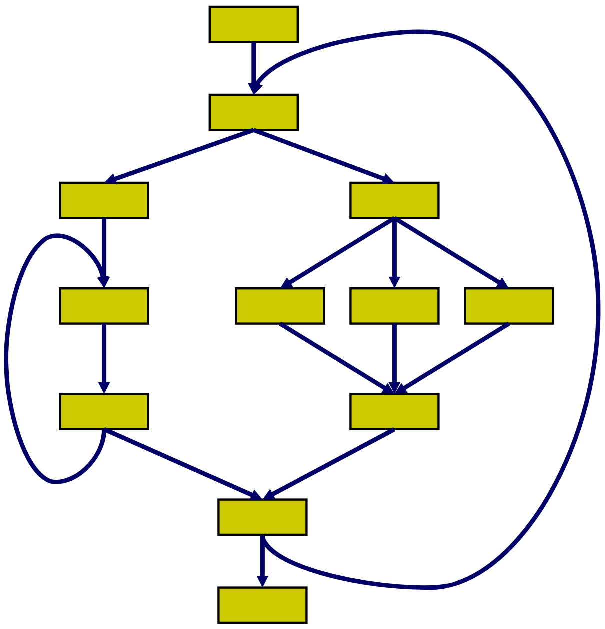
space of program behaviors

Representing Streams

- Conventional wisdom: streams are graphs
  - Graphs have no simple textual representation
  - Graphs are difficult to analyze and optimize
- Insight: stream programs have structure



unstructured



structured

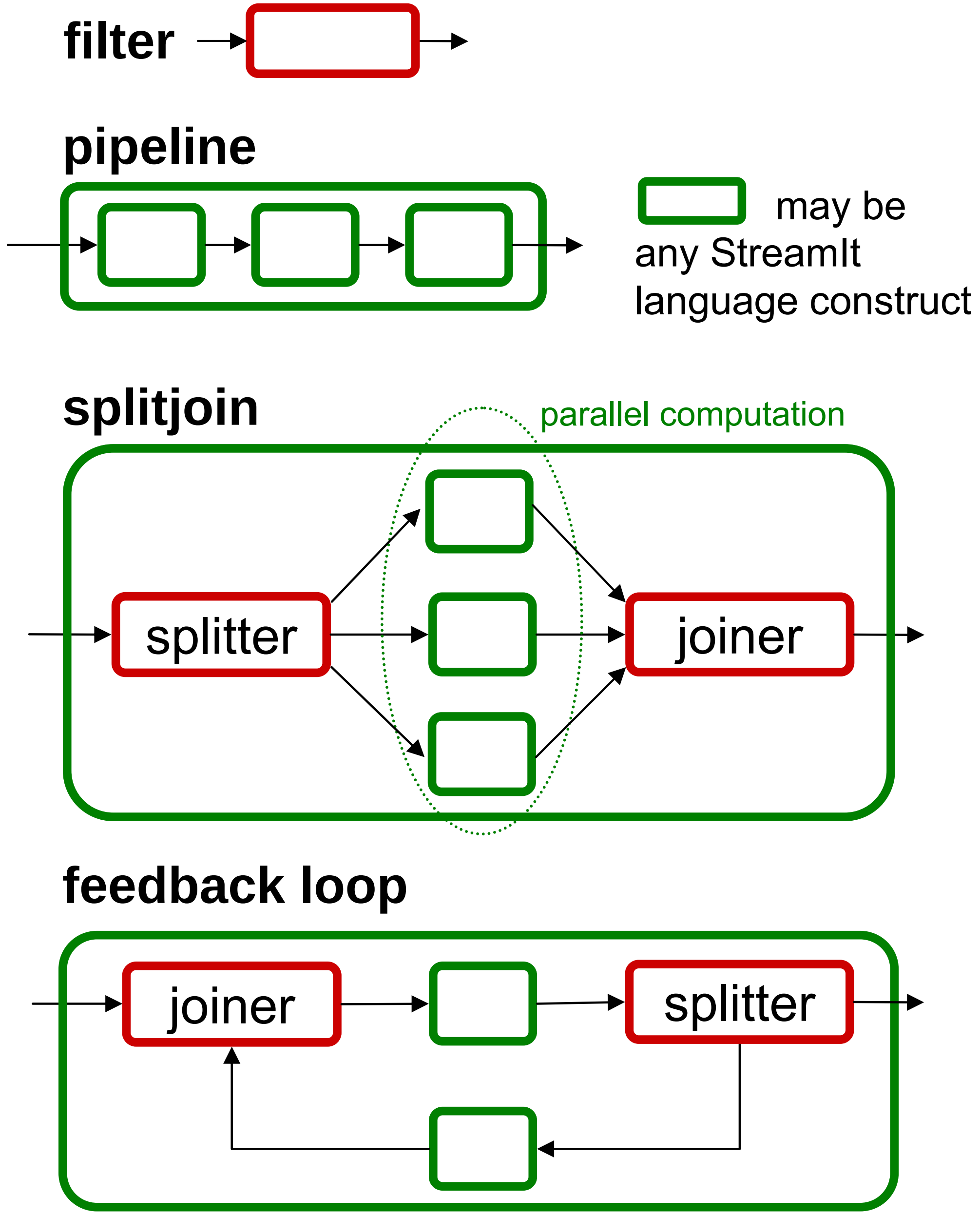
Streaming and Multicore Related Work

- CellSs (http://www.bsc.es/cellsuperscalar)
- Corepy (http://www.corepy.org/)
- Mercury Multicore Plus SDK (http://www.mc.com/ps3/)
- Rapidmind (http://www.rapidmind.net/)
- Sequoia (http://sequoia.stanford.edu/)

The StreamIt Language

- A high-level, architecture-independent language for streaming applications
  - Improves programmer productivity (vs. Java, C)
  - Offers scalable performance on multicores
- Based on synchronous dataflow, with dynamic extensions
  - Compiler or dynamic scheduler can determine execution order
  - Many aggressive optimizations possible

Structured Streams in StreamIt



filter

pipeline

may be any StreamIt language construct

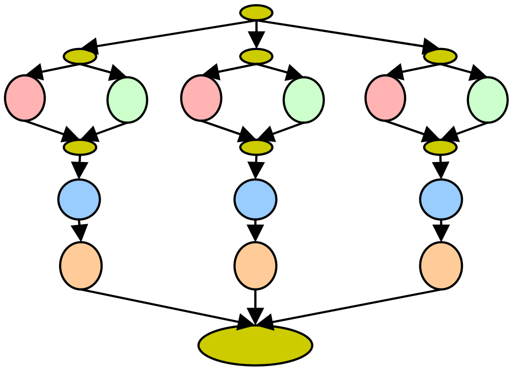
splitjoin

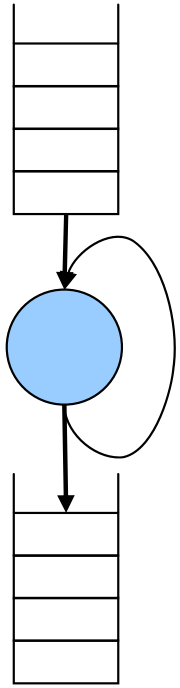
parallel computation

feedback loop

- Each structure is single-input, single-output
- Hierarchical and composable

StreamIt Execution Model

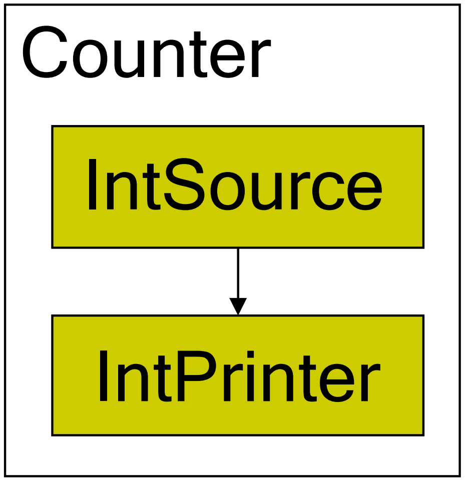




- Nodes **push** and **pop** data to FIFOs
- Constant number of items every time
- Static schedule possible
- Nodes can have local state

Example: A Simple Counter

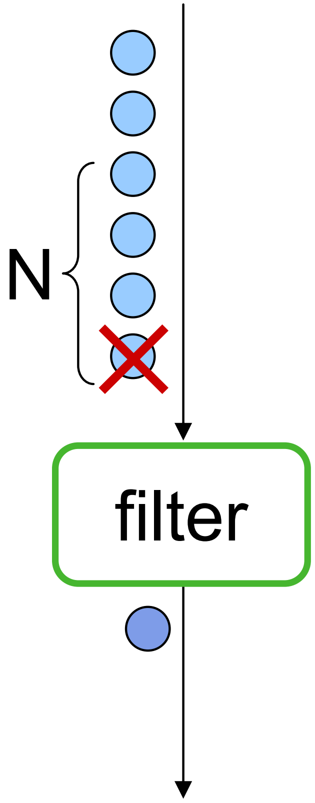
```
void->void pipeline Counter() {
    add IntSource();
    add IntPrinter();
}
void->int filter IntSource() {
    int x;
    init { x = 0; }
    work push 1 { push (x++); }
}
int->void filter IntPrinter() {
    work pop 1 { print(pop()); }
}
```



```
% strc Counter.str -o counter
% ./counter -i 4
0
1
2
3
```

Filter Example: Low Pass Filter

```
float->float filter LowPassFilter (int N, float freq) {
    float[N] weights;

    init {
        weights = calcWeights(freq);
    }

    work peek N push 1 pop 1 {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```



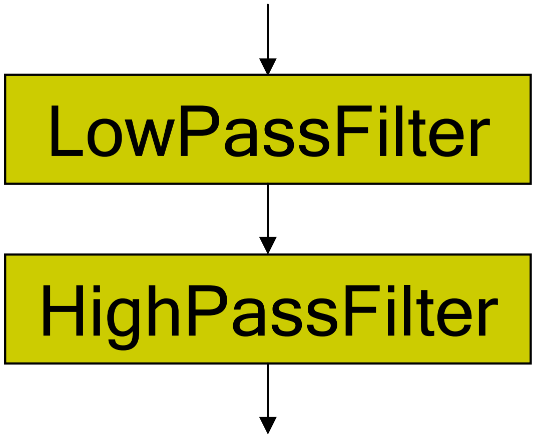
Low Pass Filter in C

```
void FIR(
  int* src,
  int* dest,
  int* srcIndex,
  int* destIndex,
  int srcBufferSize,
  int destBufferSize,
  int N) {

  float result = 0.0;
  for (int i = 0; i < N; i++) {
    result += weights[i] * src[(*srcIndex + i) % srcBufferSize];
  }
  dest[*destIndex] = result;
  *srcIndex = (*srcIndex + 1) % srcBufferSize;
  *destIndex = (*destIndex + 1) % destBufferSize;
}
```

- FIR functionality obscured by buffer management details
- Programmer must commit to a particular buffer implementation strategy

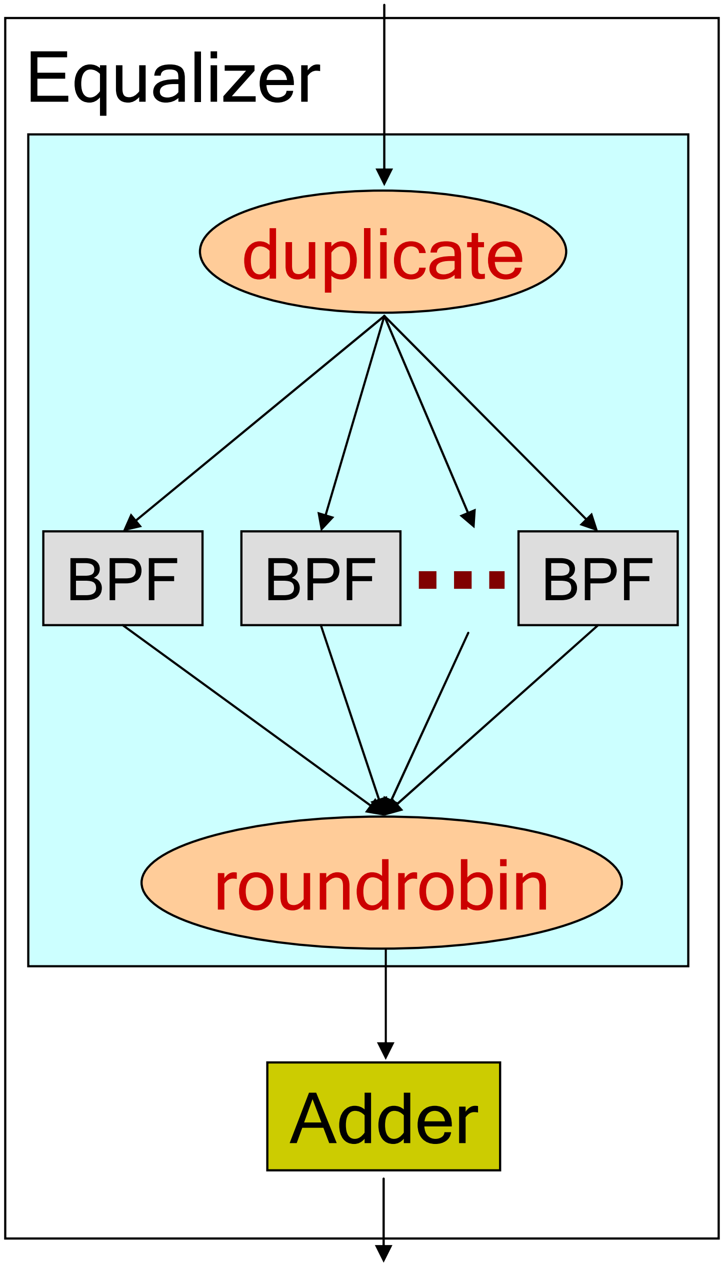
Pipeline Example: Band Pass Filter

```
float→float pipeline BandPassFilter (int N,
                                     float low,
                                     float high) {
    add LowPassFilter(N, low);
    add HighPassFilter(N, high);
}
```



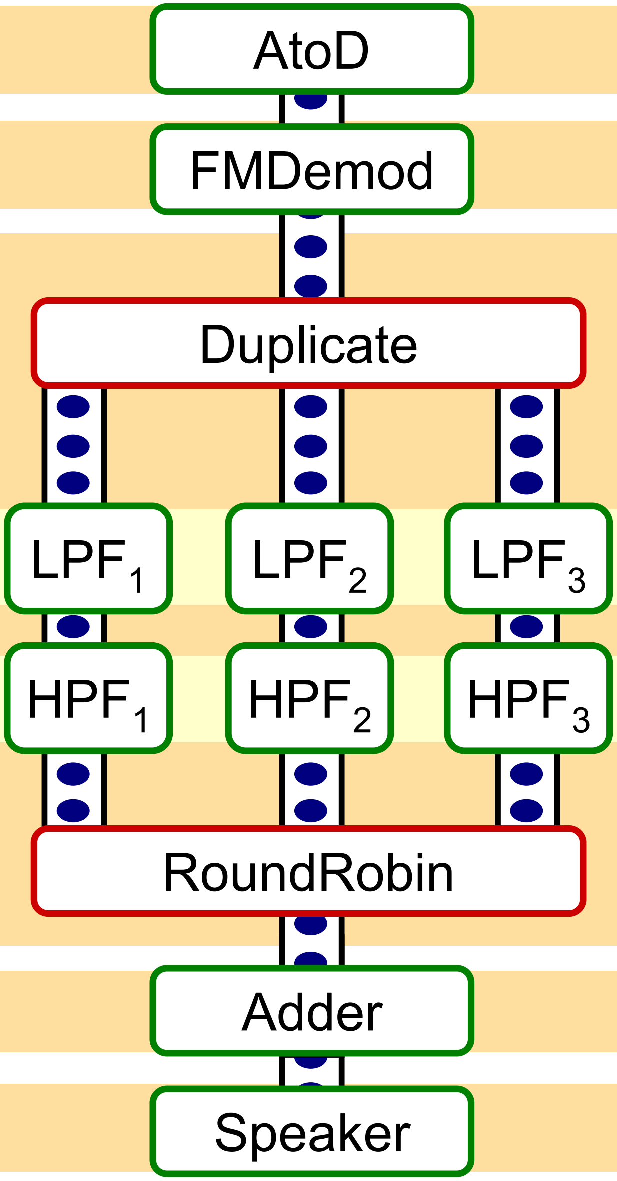
SplitJoin Example: Equalizer

```
float→float pipeline Equalizer (int N,
                                float lo,
                                float hi) {

   add splitjoin {
      split duplicate;
      for (int i=0; i<N; i++)
         add BandPassFilter(64, lo + i*(hi - lo)/N);
      join roundrobin(1);
   }
   add Adder(N);
}
```



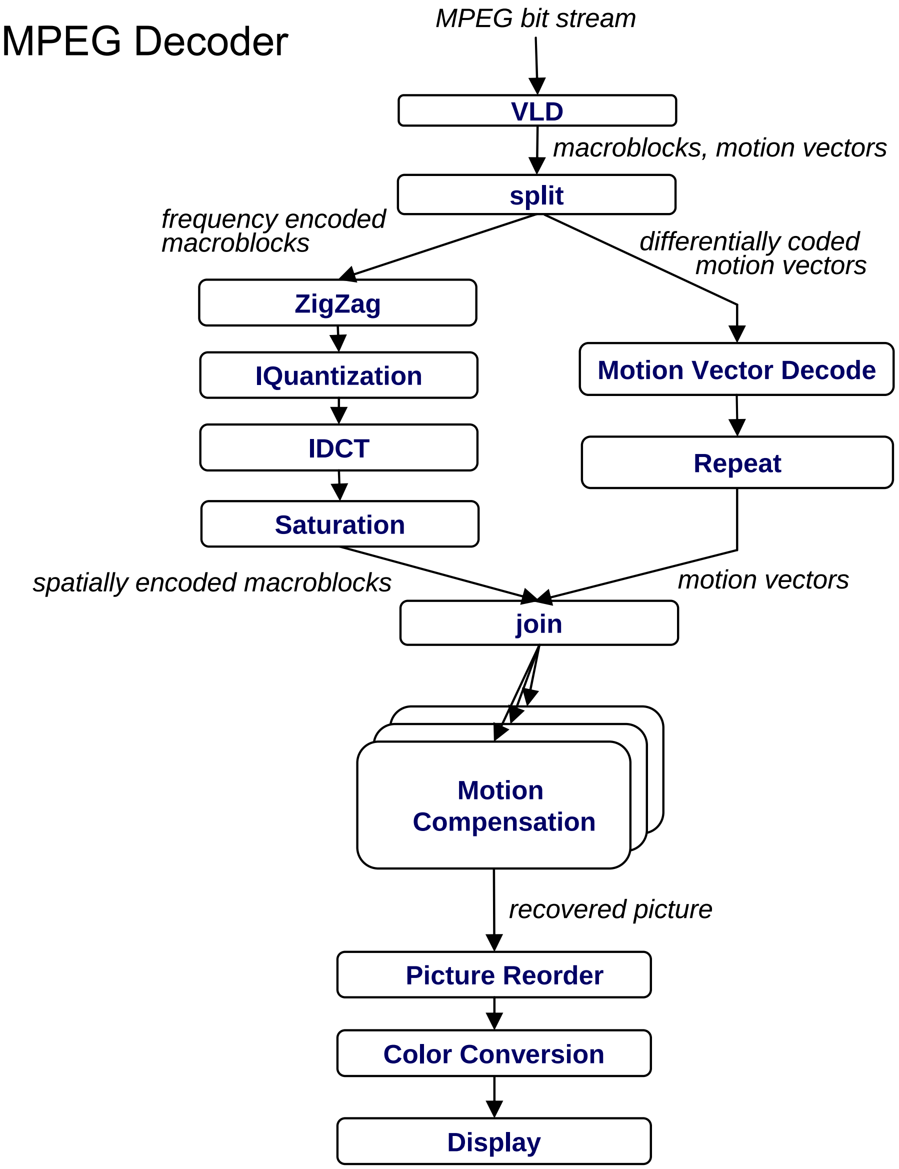
Building Larger Programs: FMRadio

```
void->void pipeline FMRadio(int N, float lo, float hi) {
  add AtoD();

  add FMDemod();

  add splitjoin {
   split duplicate;
   for (int i=0; i<N; i++) {
      add pipeline {
         add LowPassFilter(lo + i*(hi - lo)/N);

         add HighPassFilter(lo + i*(hi - lo)/N);
      }
   }
   join roundrobin();
  }
  add Adder();

  add Speaker();
}
```



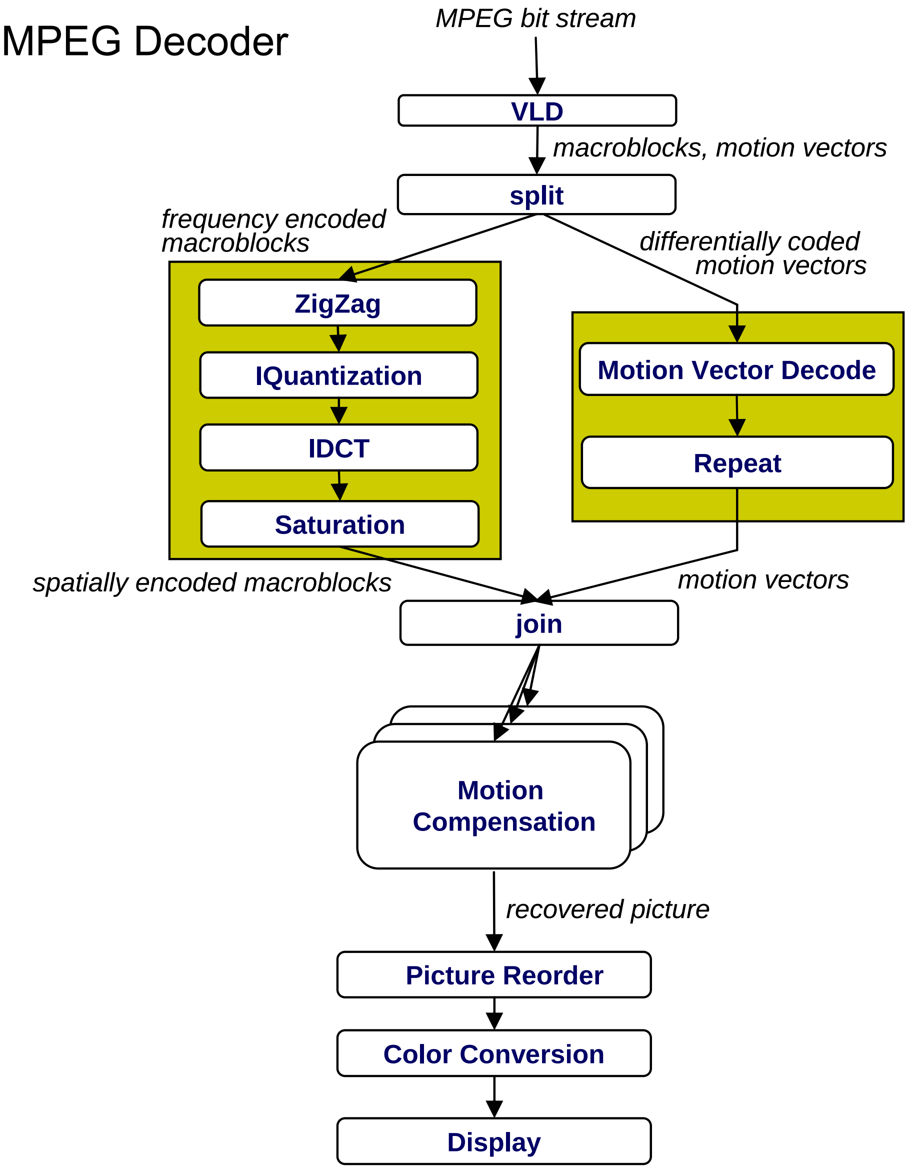
Where’s the Concurrency?

MPEG Decoder



Where's the Concurrency?

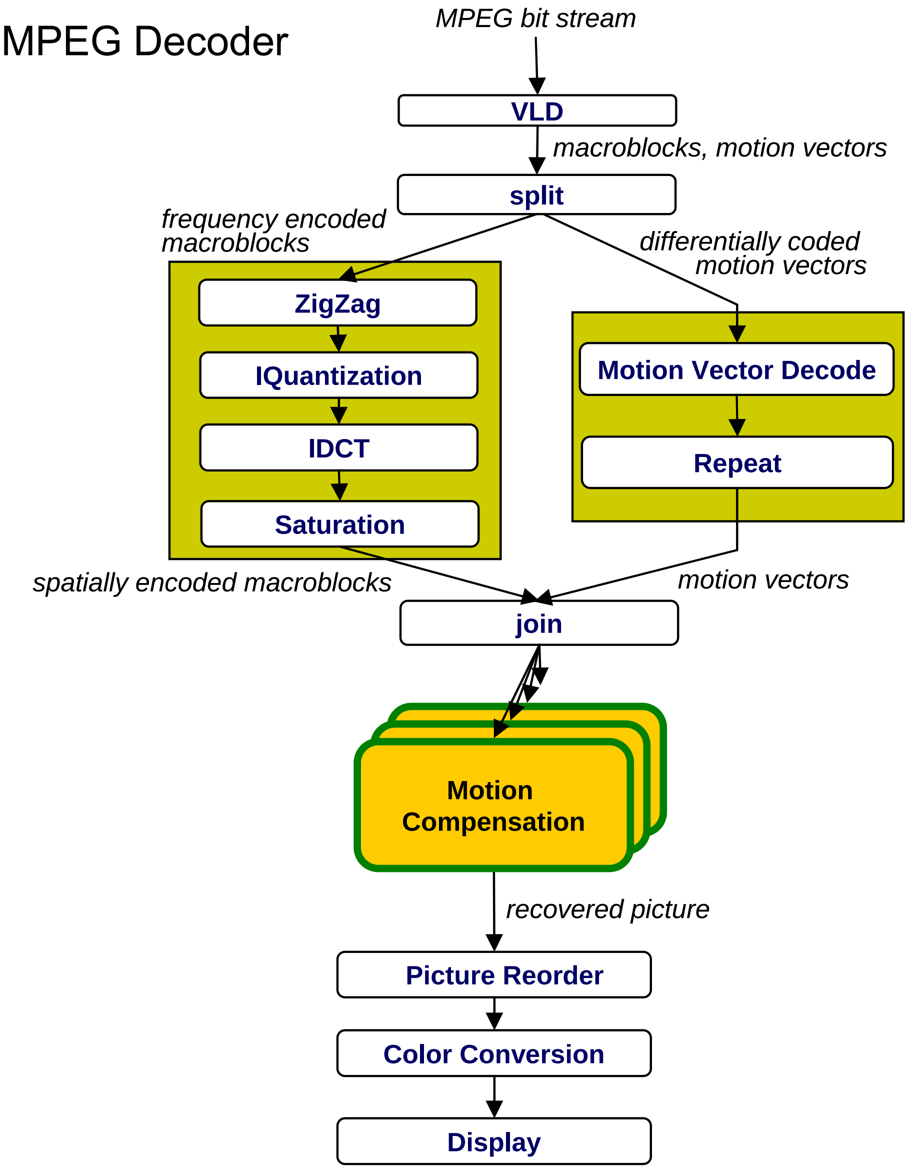
MPEG Decoder

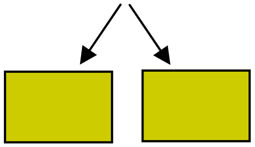


- Task decomposition
  - Independent coarse-grained computation
  - Inherent to algorithm
- Sequence of statements (instructions) that operate together as a group
  - Corresponds to some logical part of program
  - Usually follows from the way programmer thinks about a problem

Where's the Concurrency?

MPEG Decoder

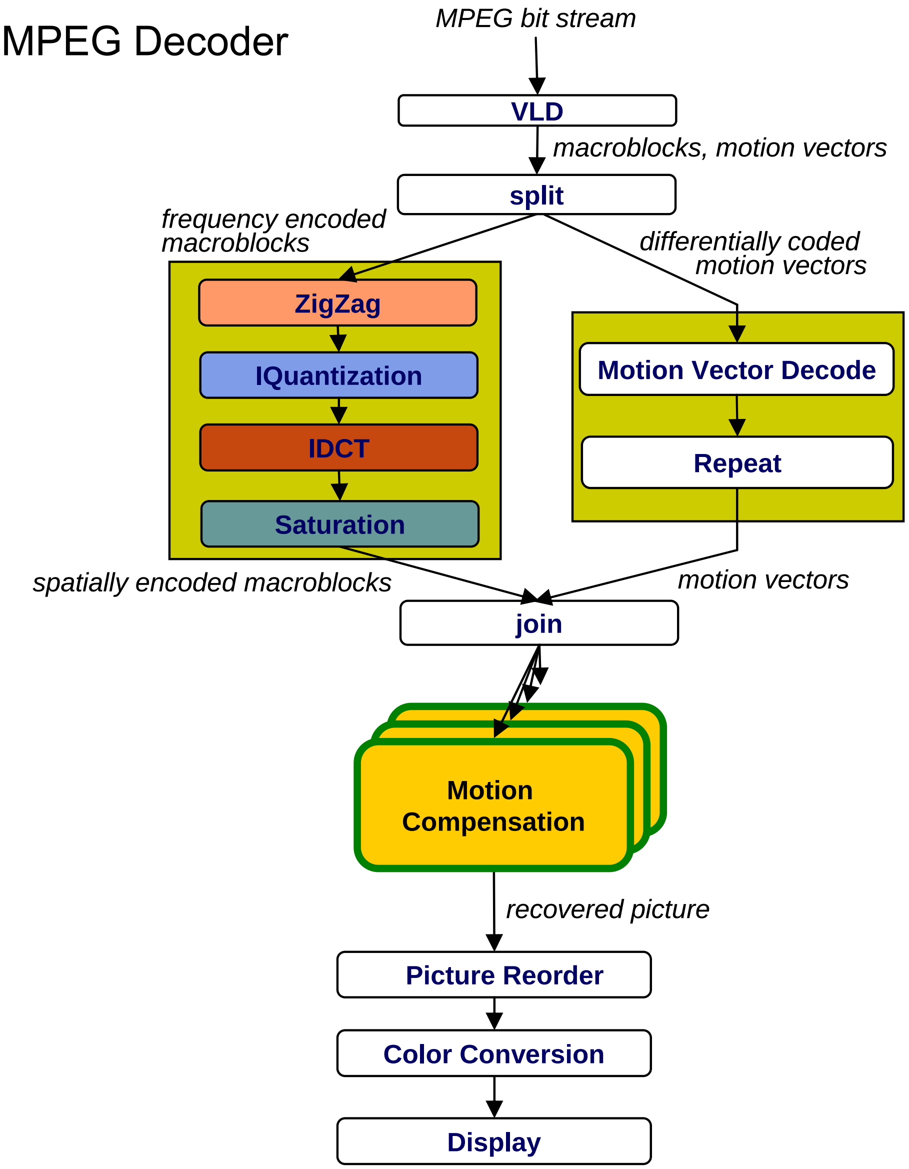




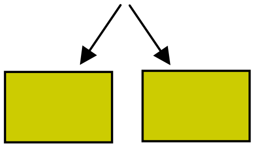
- Task decomposition
  - Parallelism in the application
- Data decomposition
  - Same computation is applied to small data chunks derived from large data set

Where's the Concurrency?

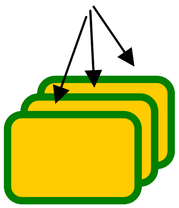
MPEG Decoder



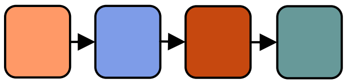
- Task decomposition
  - Parallelism in the application



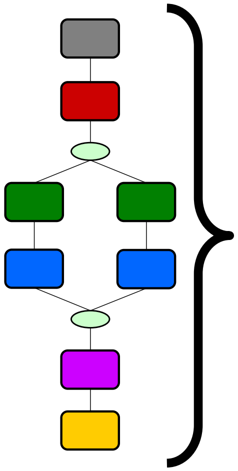
- Data decomposition
  - Same computation many data



- Pipeline decomposition
  - Data assembly lines
  - Producer-consumer chains



Productive (Stream) Programming For Parallelism using StreamIt



application

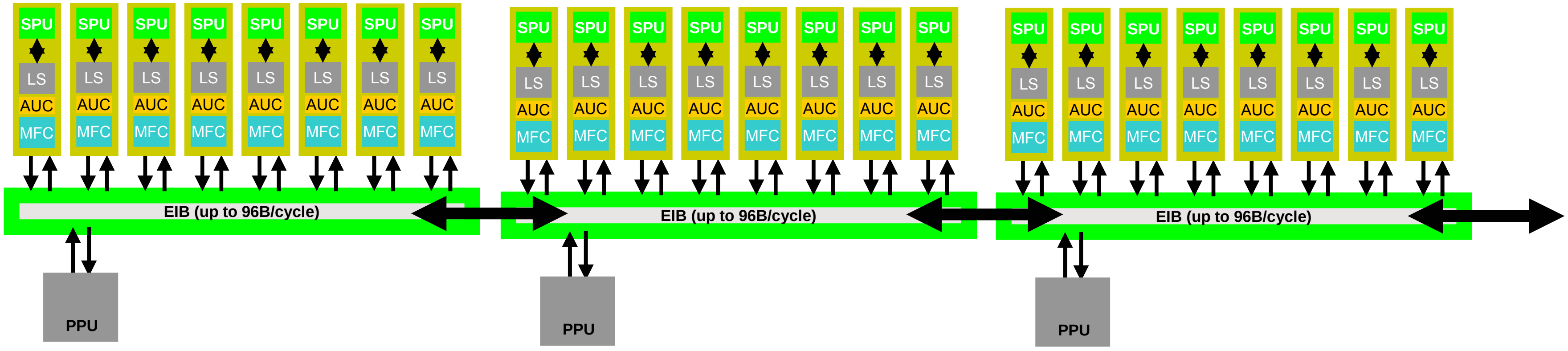
- Application is naturally parallel, exposes concurrency, dependencies, and communication patterns

StreamIt Compiler

StreamIt Dynamic Scheduler

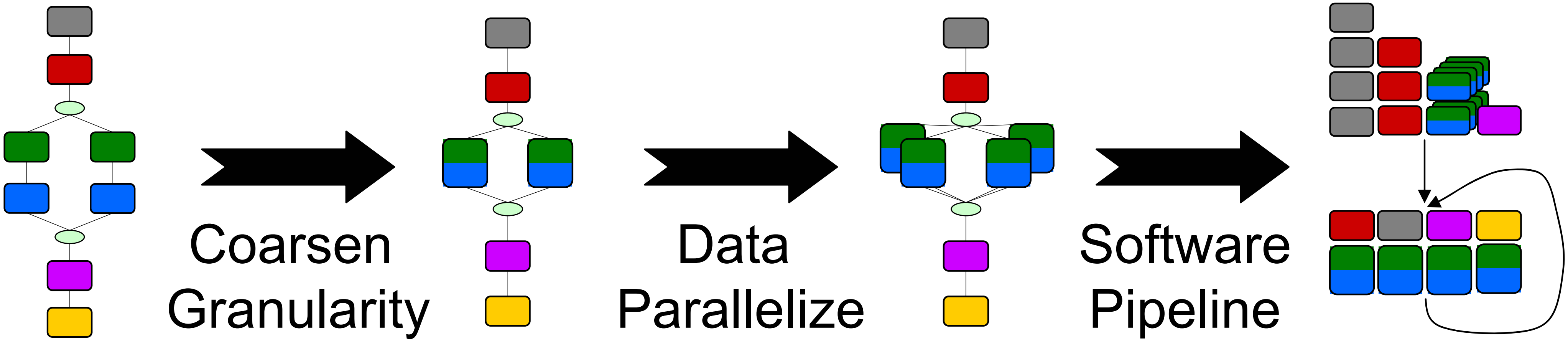
MSL is not StreamIt specific

Multicore Streaming Layer (Collection of Cores e.g., SPEs)



The StreamIt Compiler

[Gordon et al. ASPLOS ‘06]



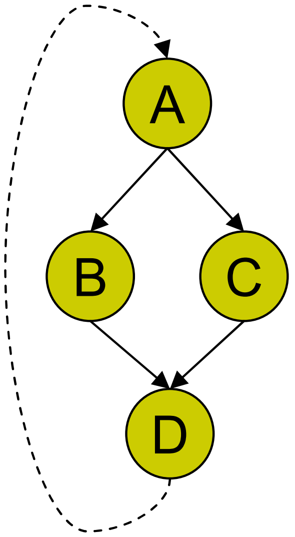
Coarsen Granularity

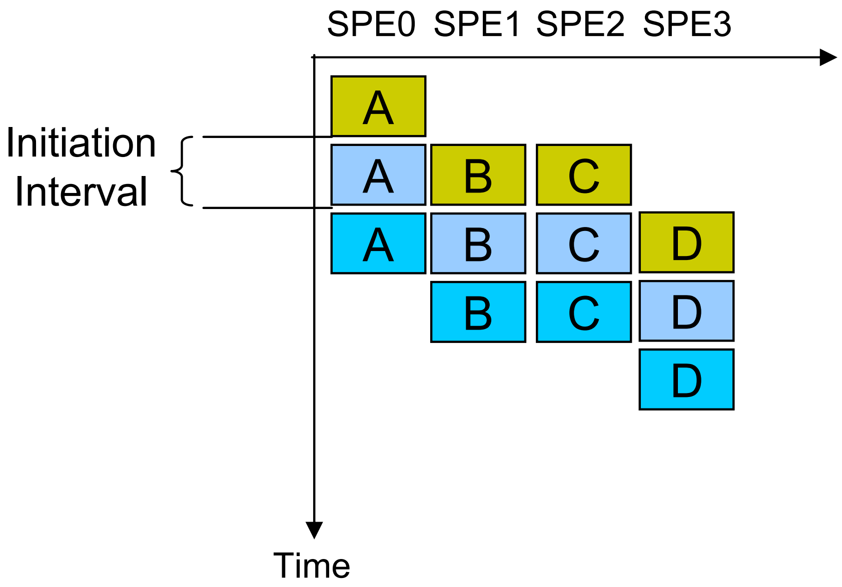
Data Parallelize

Software Pipeline

1. Coarsen: Fuse stateless sections of the graph
2. Data Parallelize: parallelize stateless filters
3. Software Pipeline: parallelize stateful filters

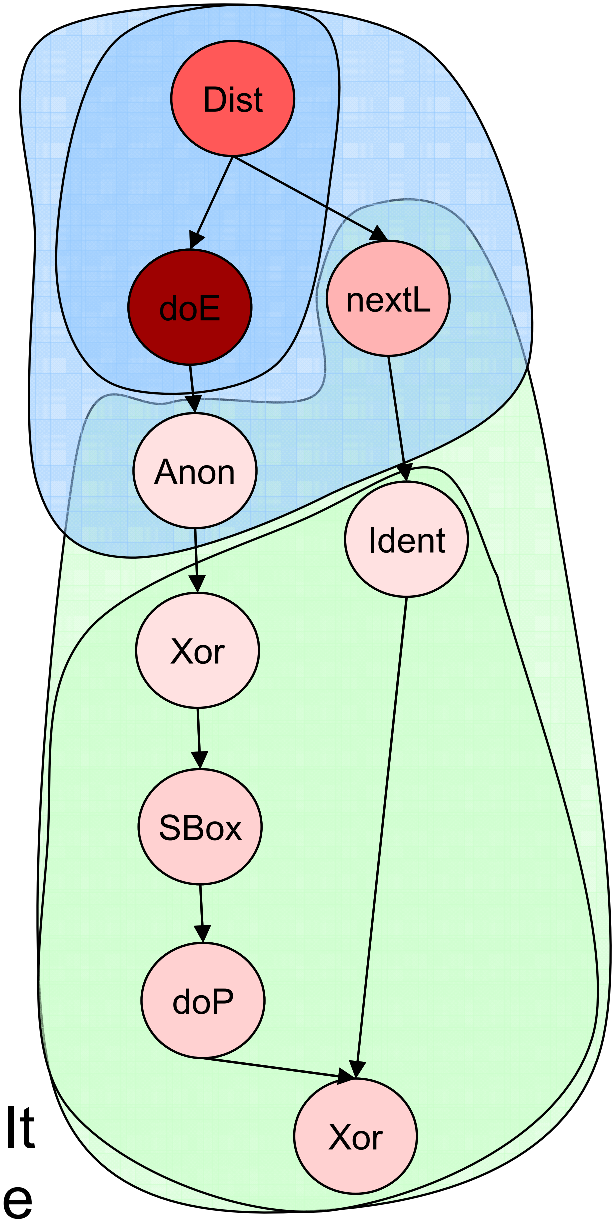
Coarse Grained Software Pipelining



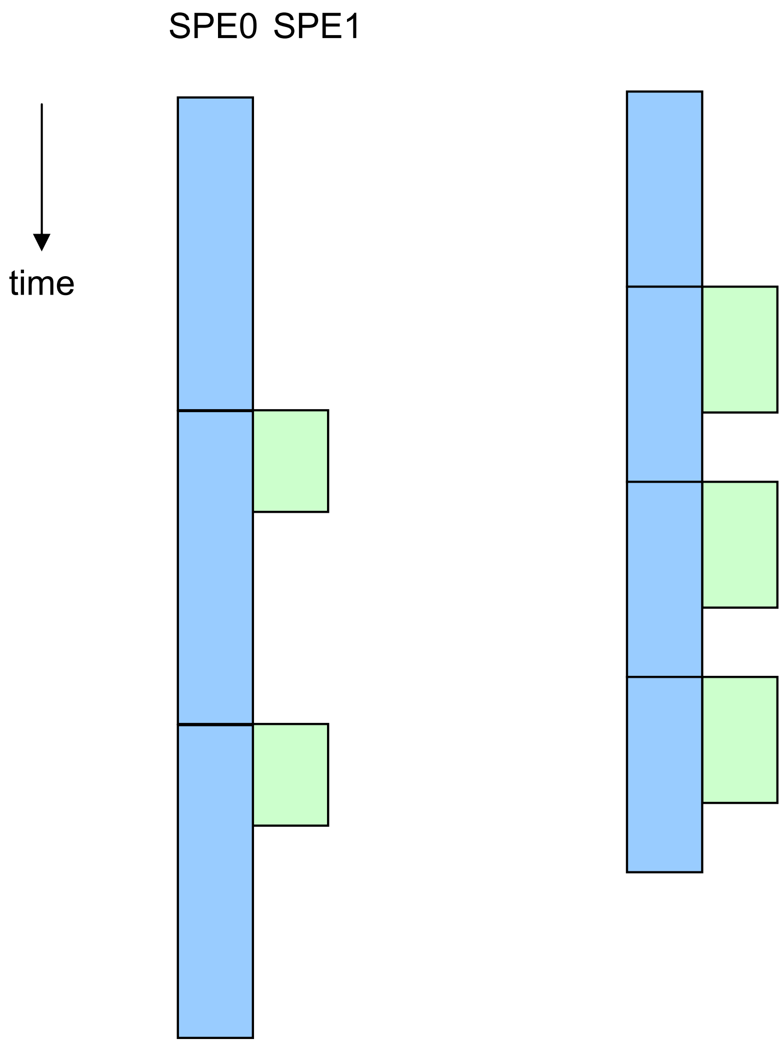


- Good work estimation enables static load balancing…
- … Leads to better utilization and throughput
- Pipelining hides communication latency

Impact of Load Balance



DES StreamIt pipeline

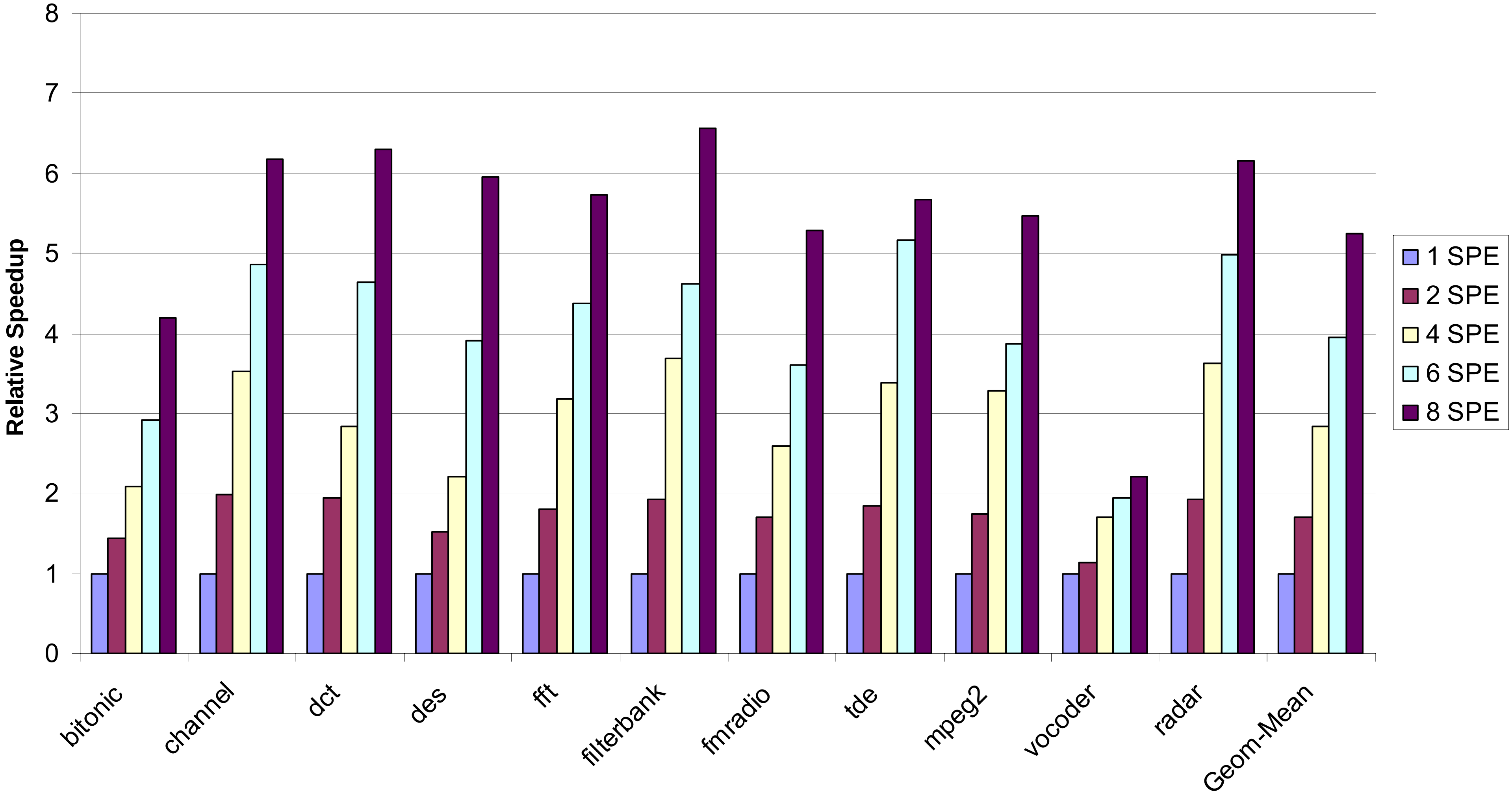


Speedup ~ 1.2

Speedup ~ 1.7

Software Pipelining On Cell

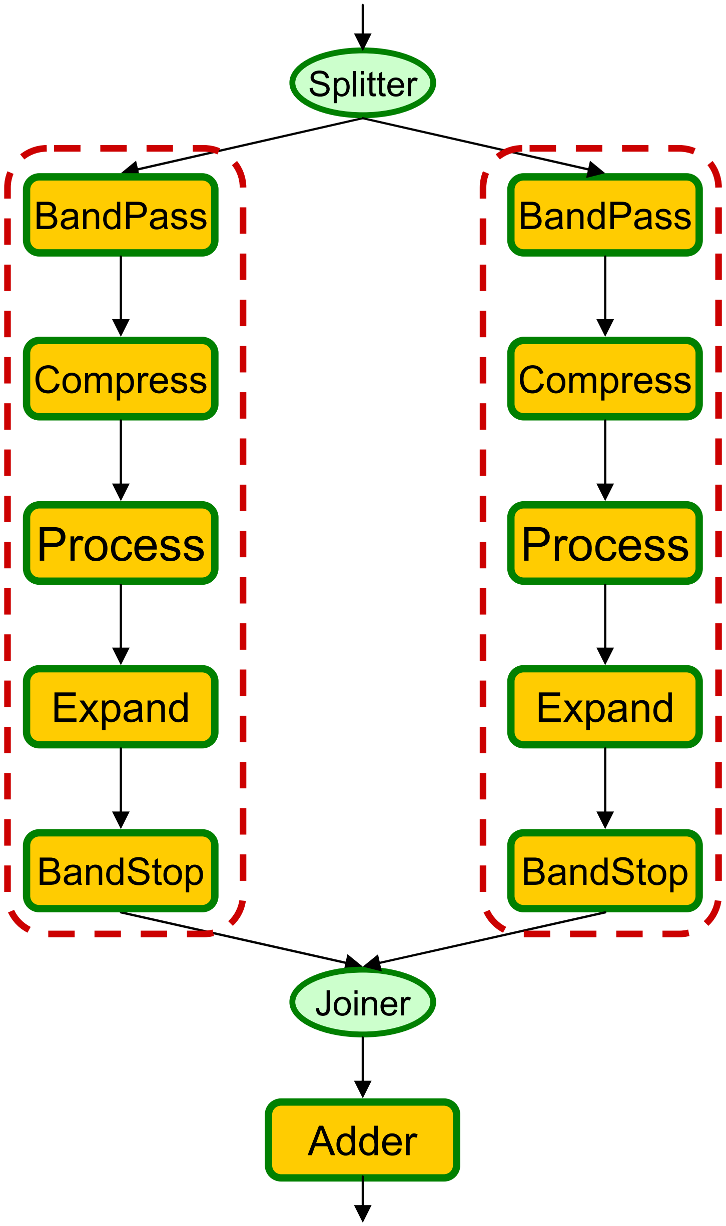
[Kudlur et al. UMich '07]



Automatic Parallelization Beyond Cell

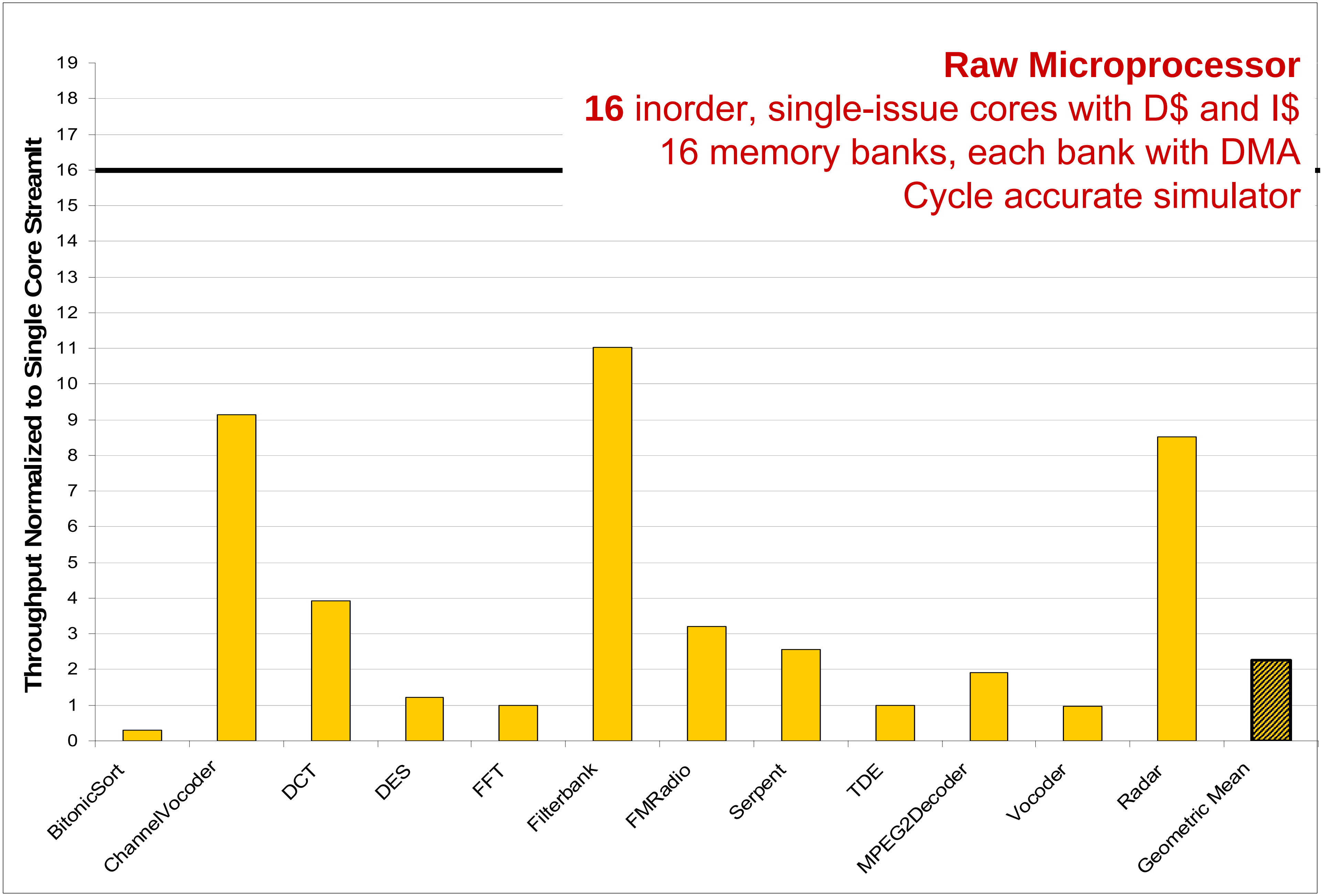
[Gordon et al. ASPLOS ‘06]

Task Parallelism



- Inherent task parallelism between two processing pipelines
- Task Parallel Model:
  - Only parallelize explicit task parallelism
  - Fork/join parallelism
- Execute this on a 2 core machine ~2x speedup over single core
- What about 4, 16, 1024, … cores?

Task Parallelism



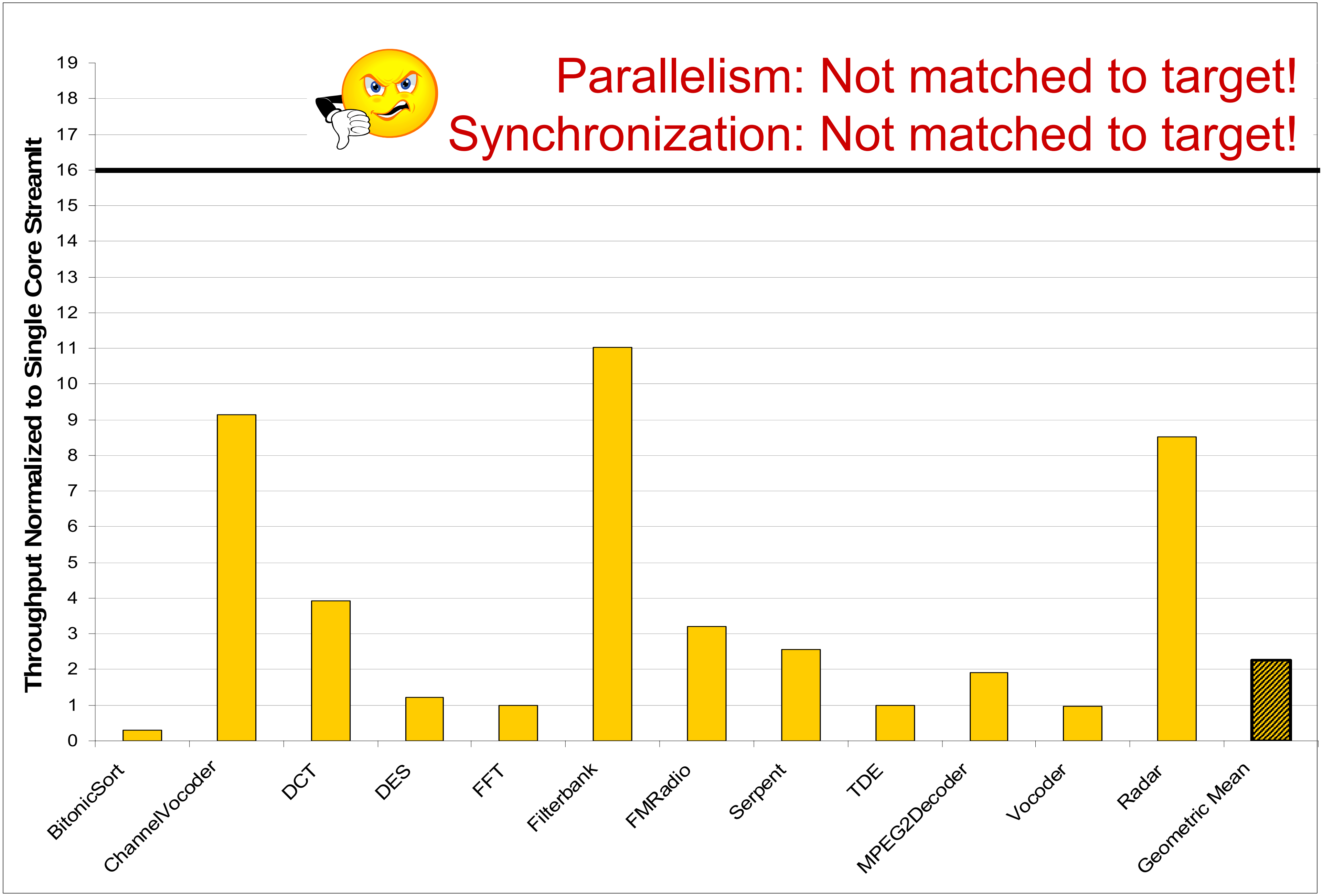
Raw Microprocessor

16 inorder, single-issue cores with D$ and I$

16 memory banks, each bank with DMA

Cycle accurate simulator

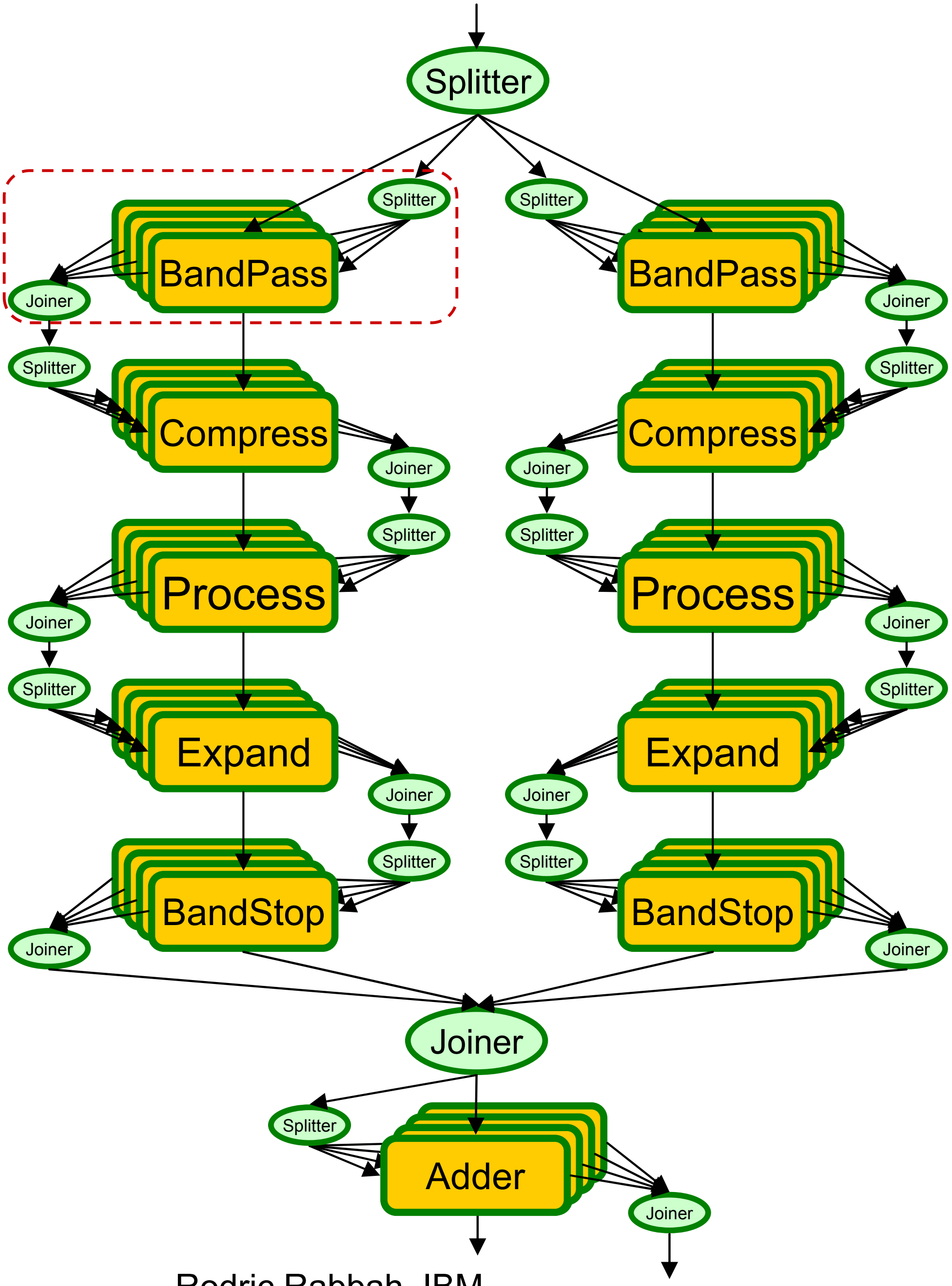
Task Parallelism



Parallelism: Not matched to target!

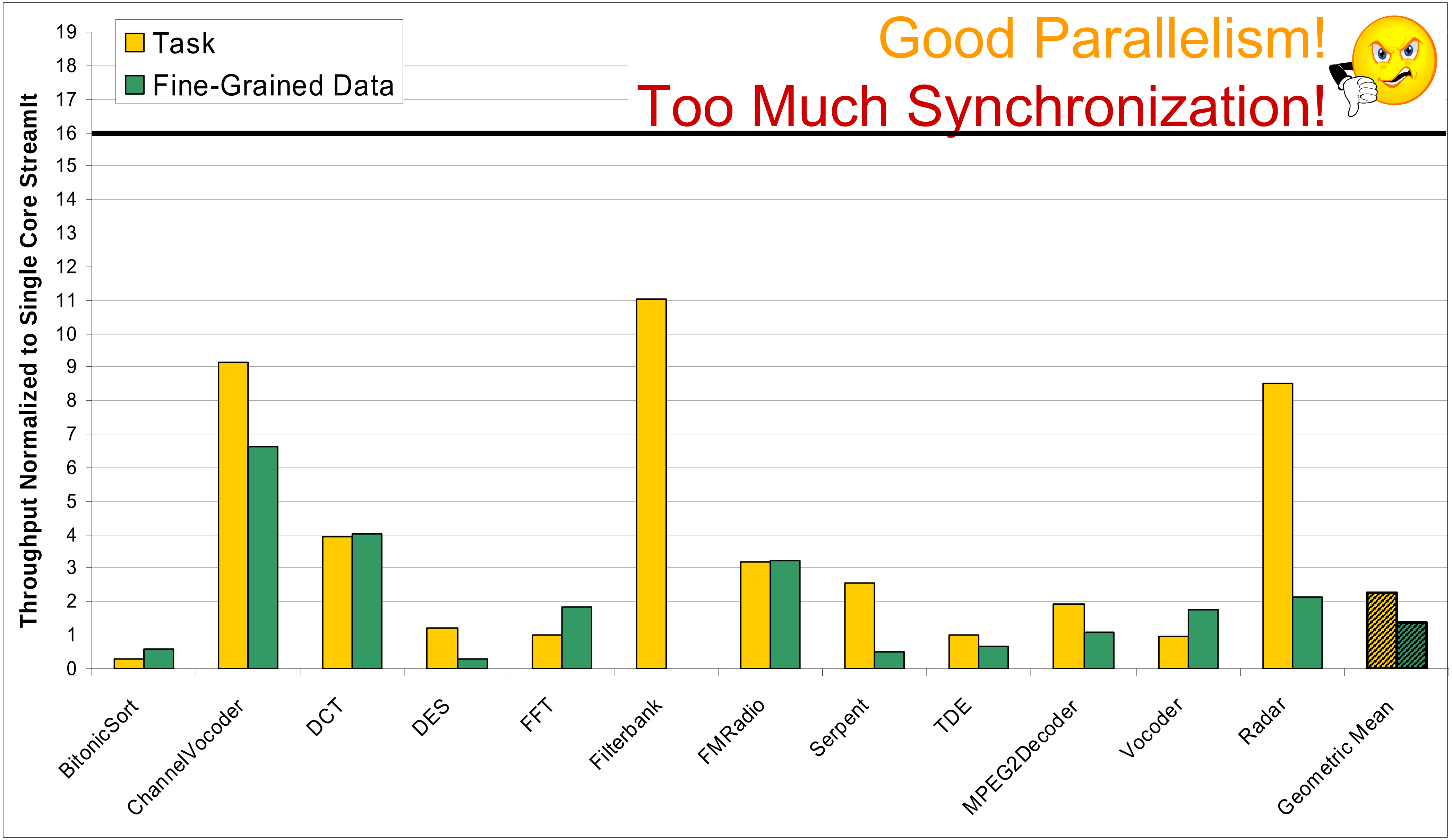
Synchronization: Not matched to target!

Data Parallelism



- Each of the filters in the example are stateless
- Fine-grained Data Parallel Model:
  - *Fiss* each stateless filter *N* ways (*N* is number of cores)
  - Remove scatter/gather if possible
- We can introduce data parallelism
  - Example: 4 cores
- Each fission group occupies entire machine

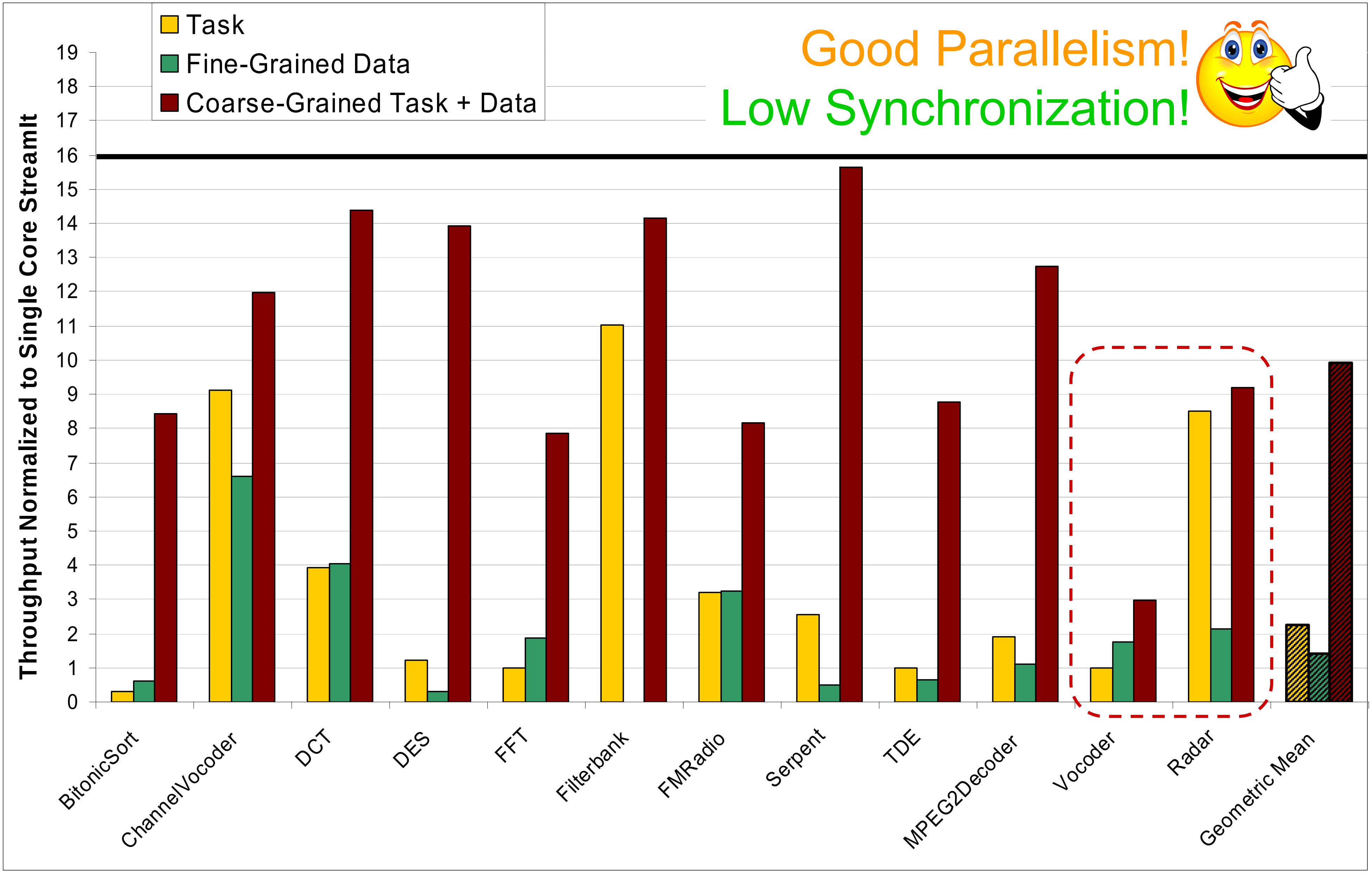
Fine-Grained Data Parallelism



Good Parallelism!

Too Much Synchronization!

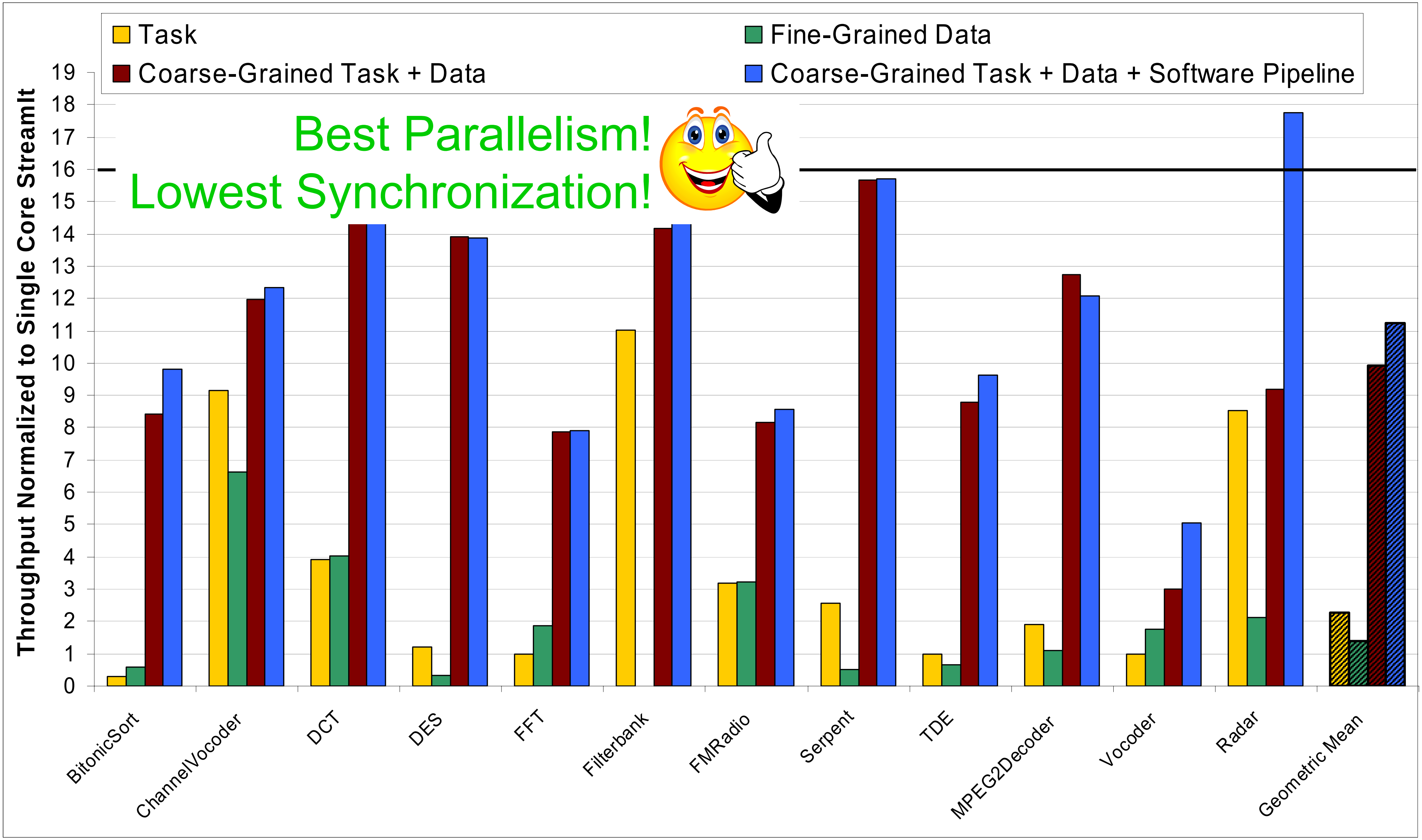
Coarse-Grained Data Parallelism



Good Parallelism!

Low Synchronization!

Coarse-Grained
Task + Data + Software Pipelining

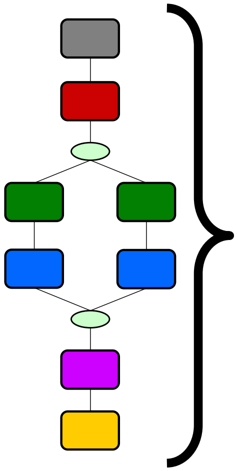


Best Parallelism!

Lowest Synchronization!

Virtualizing Multicore Architecture

Productive (Stream) Programming For Parallelism using StreamIt



application

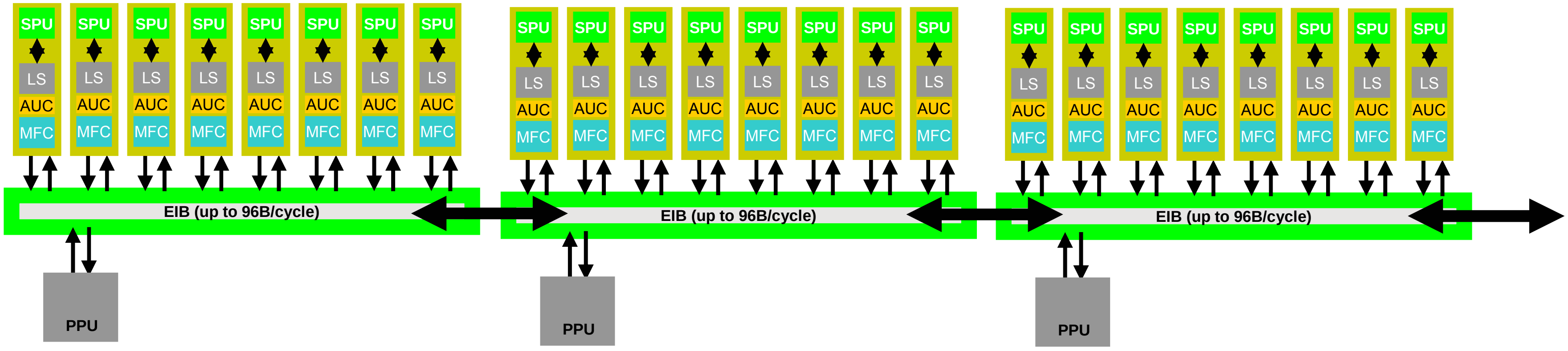
- Application is naturally parallel, exposes concurrency, dependencies, and communication patterns

StreamIt Compiler

StreamIt Dynamic Scheduler

MSL is not StreamIt specific

Multicore Streaming Layer (Collection of Cores e.g., SPEs)



Multicore Steaming Layer

[Zhang. MIT MEng ‘07]

- Computation viewed as a collection of
  - Filters: encapsulate computation and state
  - Buffers: input and output attached to filters
- Provides a instruction set
  - Filter commands (load, unload)
  - Buffer commands (allocate, attach)
  - Data transfer commands (indirectly translate to synchronization)

Multicore Steaming Layer

[Zhang. MIT MEng '07]

- Abstracts away details of explicit data communication
  - Far easier to implement scheduling patterns on top of MSL, compared to for example Cell API

- Facilitate mapping of computation to a multicore
  - Compiler (or programmer) focuses on optimizations and graph refinement
  - Compiler uses communication patterns that are suitable for the application
  - Details of the actual communication mechanism may differ, and are hidden from the compiler

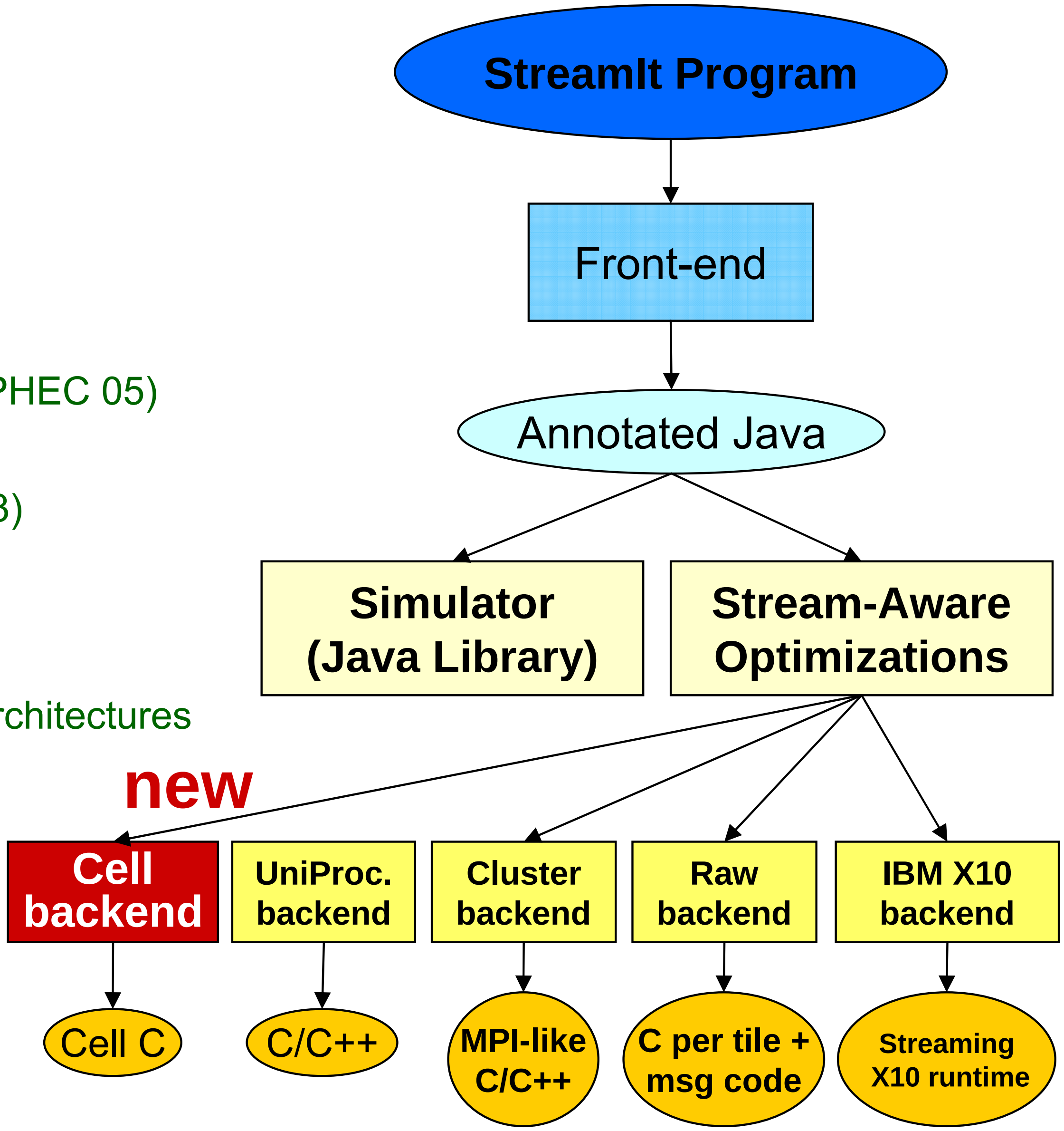
Multicore Steaming Layer

- StreamIt compiler easily maps filters and buffers for MSL
  - Benefit of a practical and lightweight dynamic scheduler
  - Between 1-9% of runtime overhead
    [Zhang. MIT MEng '07]

- Compiler can also implement static schedule directly in MSL instruction set
  - Evaluate both static and dynamic scheduling algorithms
    [Zhang, Li, et al. '07]

The StreamIt Project

- **Applications**
  - DES and Serpent [PLDI 05]
  - MPEG-2 [IPDPS 06]
  - SAR, DSP benchmarks, JPEG, …
- **Programmability**
  - StreamIt Language (CC 02)
  - Teleport Messaging (PPOPP 05)
  - Programming Environment in Eclipse (P-PHEC 05)
- **Domain Specific Optimizations**
  - Linear Analysis and Optimization (PLDI 03)
  - Optimizations for bit streaming (PLDI 05)
  - Linear State Space Analysis (CASES 05)
- **Architecture Specific Optimizations**
  - Compiling for Communication-Exposed Architectures (ASPLOS 02)
  - Phased Scheduling (LCTES 03)
  - Cache Aware Optimization (LCTES 05)
  - Load-Balanced Rendering (Graphics Hardware 05)
  - Exploiting Coarse-Grained Parallelism in Stream Programs (ASPLOS 06)
- http://cag.csail.mit.edu/streamit/cell



Special Notice – Trademarks

Revised July 23, 2006