# Job scheduler

## as simple as possible

but not simpler

Arseny “Zeux” Kapoulkine
CREAT Studios
arseny.kapoulkine@gmail.com

# Free lunch is over!

- The biggest sea change in software development since the OO revolution is knocking at the door, and its name is Concurrency.

- Herb Sutter

# Parallel universe

- A lot of general-purpose cores
  - PS3: 6 SPU
  - XBox 360: 3 PowerPC cores
- Different approaches
  - Thread-parallel programming
  - Task-parallel programming
  - Data-parallel programming

# Glossary

- Job
  - Code that performs a data transformation
- Batch
  - Job reference + data
- Scheduler
  - A system that executes batches

# Job

- A function that transforms data
  - Side effects limited to the input data
    - No global variable access
- Can run many times per frame
  - Sometimes simultaneously
  - Sometimes on the same data
- No preemption
  - Left out since it requires extra memory

# Scheduler v1.0

- Simplest code possible
  - Reasonably convenient
  - Fast
- Add batches to execute
- Wait for execution to stop
  - Two levels of synchronization
    - Single batch
    - Group of batches (32 groups, uint32 counter per group)

# Scheduler v1.0

- Global batch queue
  - Fixed queue size
  - Lock-free
  - Push and pop copy batch data
    - No memory management for batch structures
  - No empty/size operations
    - Does not make sense in a multi-threaded queue
  - Applicable beyond schedulers!

# Lock-free queue

- Queue (FIFO) is a fundamental container
- A lot of published lock-free implementations
  - MPMC – Multi-Producer, Multi-Consumer
    - Implementing Lock-Free Queues [94] – RACE!
    - Correction of a Memory Management Method for Lock-Free Data Structures [95]
    - Optimised Lock-Free FIFO Queue [01] – RACE!
    - Optimised Lock-Free FIFO Queue [03]
    - Optimized Lock-Free FIFO Queue continued [05]

# Lock-free queue – 1/3

```
1.  template <class T>
2.  class TLockFreeQueue
3.  {
4.      struct TListNode
5.      {
6.          TListNode *volatile Next;
7.          T Data;
8.      };
9.
10.     struct TRootNode
11.     {
12.         TListNode *volatile PushQueue;
13.         TListNode *volatile PopQueue;
14.         TListNode *volatile ToDelete;
15.         TRootNode *volatile NextFree;
16.
17.         TRootNode() : PushQueue(0), PopQueue(0),
    ToDelete(0), NextFree(0) {}
18.     };
19.
20.     static void EraseList(TListNode *n)
21.     {
22.         while (n) {
23.             TListNode *keepNext = n->Next;
24.             delete n;
25.             n = keepNext;
26.         }
27.     }
28.
29.     TRootNode *volatile JobQueue;
30.     volatile long FreememCounter;
31.     TRootNode *volatile FreePtr;
32.
33.     void TryToFreeAsyncMemory()
34.     {
35.         TRootNode *current = FreePtr;
36.         if (current == 0)
37.             return;
38.         if (atomic_add(FreememCounter, 0) == 1) {
39.             // we are the last thread, try to cleanup
40.             if (cas(&FreePtr, (TRootNode*)0, current)) {
41.                 // free list
42.                 while (current) {
43.                     TRootNode *p = current->NextFree;
44.                     EraseList(current->ToDelete);
45.                     delete current;
46.                     current = p;
47.                 }
48.             }
49.         }
50.     }
51.     void AsyncRef()
52.     {
53.         atomic_add(FreememCounter, 1);
54.     }
55.     void AsyncUnref()
56.     {
57.         TryToFreeAsyncMemory();
58.         atomic_add(FreememCounter, -1);
59.     }
60.     void AsyncDel(TRootNode *toDelete, TListNode *lst)
61.     {
62.         toDelete->ToDelete = lst;
63.         for(;;) {
64.             toDelete->NextFree = FreePtr;
65.             if (cas(&FreePtr, toDelete, toDelete-
    >NextFree))
66.                 break;
67.         }
68.     }
69.     void AsyncUnref(TRootNode *toDelete, TListNode *lst)
70.     {
71.         TryToFreeAsyncMemory();
72.         if (atomic_add(FreememCounter, -1) == 0) {
73.             // no other operations in progress, can safely
    reclaim memory
74.             EraseList(lst);
75.             delete toDelete;
76.         } else {
77.             // Dequeue()s in progress, put node to free
    list
78.             AsyncDel(toDelete, lst);
79.         }
80.     }
81.
82.
83.     struct TListInvertor
84.     {
85.         TListNode *Copy;
86.         TListNode *Tail;
87.         TListNode *PrevFirst;
88.
89.         TListInvertor() : Copy(0), Tail(0), PrevFirst(0) {}
90.         ~TListInvertor()
91.         {
92.             if (Copy)
93.                 Copy = Copy;
94.             EraseList(Copy);
95.         }
96.         void CopyWasUsed()
97.         {
98.             Copy = 0;
99.             Tail = 0;
100.                PrevFirst = 0;
101.            }
102.            void DoCopy(TListNode *ptr)
```

# Lock-free queue – 2/3

```
103.                {
104.                    TListNode *newFirst = ptr;
105.                    TListNode *newCopy = 0;
106.                    TListNode *newTail = 0;
107.                    while (ptr) {
108.                        if (ptr == PrevFirst) {
109.                            // short cut, we have copied
    this part already
110.                            Tail->Next = newCopy;
111.                            newCopy = Copy;
112.                            Copy = 0; // do not destroy prev
    try
113.                            if (!newTail)
114.                                newTail = Tail; // tried to
    invert same list
115.                            break;
116.                        }
117.                        TListNode *newElem = new TListNode;
118.                        newElem->Data = ptr->Data;
119.                        newElem->Next = newCopy;
120.                        newCopy = newElem;
121.                        ptr = ptr->Next;
122.                        if (!newTail)
123.                            newTail = newElem;
124.                    }
125.                    EraseList(Copy); // copy was useless
126.                    Copy = newCopy;
127.                    PrevFirst = newFirst;
128.                    Tail = newTail;
129.                }
130.            };
131.
132.            TLockFreeQueue(const TLockFreeQueue&) {}
133.            void operator=(const TLockFreeQueue&) {}
134.        public:
135.            TLockFreeQueue() : JobQueue(new TRootNode),
    FreememCounter(0), FreePtr(0) {}
136.            ~TLockFreeQueue()
137.            {
138.                AsyncRef();
139.                AsyncUnref();
140.                EraseList(JobQueue->PushQueue);
141.                EraseList(JobQueue->PopQueue);
142.                delete JobQueue;
143.            }
144.            void Enqueue(const T &data)
145.            {
146.                TListNode *newNode = new TListNode;
147.                newNode->Data = data;
148.                TRootNode *newRoot = new TRootNode;
149.                AsyncRef();
150.                newRoot->PushQueue = newNode;
151.                for(;;) {
152.                    TRootNode *curRoot = JobQueue;
153.                    newRoot->PushQueue = newNode;
154.                    newNode->Next = curRoot->PushQueue;
155.                    newRoot->PopQueue = curRoot->PopQueue;
156.                    if (cas(&JobQueue, newRoot, curRoot)) {
157.                        AsyncUnref(curRoot, 0);
158.                        break;
159.                    }
160.                }
161.            }
162.            bool Dequeue(T *data)
163.            {
164.                TRootNode *newRoot = 0;
165.                TListInvertor listInvertor;
166.                AsyncRef();
167.                for (;;) {
168.                    TRootNode *curRoot = JobQueue;
169.                    TListNode *tail = curRoot->PopQueue;
170.                    if (tail) {
171.                        // has elems to pop
172.                        if (!newRoot)
173.                            newRoot = new TRootNode;
174.
175.                        newRoot->PushQueue = curRoot-
    >PushQueue;
176.                        newRoot->PopQueue = tail->Next;
177.                        ASSERT(curRoot->PopQueue == tail);
178.                        if (cas(&JobQueue, newRoot,
    curRoot)) {
179.                            *data = tail->Data;
180.                            tail->Next = 0;
181.                            AsyncUnref(curRoot, tail);
182.                            return true;
183.                        }
184.                        continue;
185.                    }
186.                    if (curRoot->PushQueue == 0) {
187.                        delete newRoot;
188.                        AsyncUnref();
189.                        return false; // no elems to pop
190.                    }
191.
192.                    if (!newRoot)
193.                        newRoot = new TRootNode;
194.                    newRoot->PushQueue = 0;
195.                    listInvertor.DoCopy(curRoot->PushQueue);
196.                    newRoot->PopQueue = listInvertor.Copy;
197.                    ASSERT(curRoot->PopQueue == 0);
198.                    if (cas(&JobQueue, newRoot, curRoot)) {
199.                        newRoot = 0;
200.                        listInvertor.CopyWasUsed();
201.                        AsyncDel(curRoot, curRoot-
    >PushQueue);
```

# Lock-free queue – 3/3

```
202.                    } else {
203.                        newRoot->PopQueue = 0;
204.                    }
205.                }
206.            }
207.            bool IsEmpty()
208.            {
209.                AsyncRef();
210.                TRootNode *curRoot = JobQueue;
211.                bool res = curRoot->PushQueue == 0 &&
    curRoot->PopQueue == 0;
212.                AsyncUnref();
213.                return res;
214.            }
```

# Lock-free queue – 3/3

```
202.                    } else {
203.                        newRoot->PopQueue = 0;
204.                    }
205.                }
206.            }
207.            bool IsEmpty()
208.            {
209.                AsyncRef();
210.                TRootNode *curRoot = JobQueue;
211.                bool res = curRoot->PushQueue == 0 &&
    curRoot->PopQueue == 0;
212.                AsyncUnref();
213.                return res;
214.            }
```

- Are there lock-free and bug-free algorithms?
  - Yes.
  - How about algorithms with 4 pages of code?
    - The code above has a memory leak

# Batch

```
struct Batch {
    /* header: 16 bytes */
    ...
    /* job data: 112 bytes */
    char user_data[112];
};
```

# Queue

```
struct Queue {
    Batch batches[QUEUE_SIZE];

    volatile uint32_t get;
    volatile uint32_t put;
};
```

# Queue internals

- Circular buffer
  - get – first batch that has not been processed
  - put – first batch that has not been queued
  - get != put **iff** queue is not empty
- get/put can overflow
  - Batch indices are specified modulo QUEUE_SIZE
    - QUEUE_SIZE is a power of two
  - $2^{32}$ * 3000 cycles = 1.2 hours

# Queue internals

- Every batch has two indices
  - Global – uint32_t, reasonably unique
  - Local – from 0 to QUEUE_SIZE-1
- All operations start by selecting the global index
  - 1 atomic instruction
- All race conditions are limited to the same batch
  - … or to two batches with the same local index

# Queue: adding a batch

```
// atomically: index = q.put++
uint32_t index = atomic_increment(&q.put);

batch = &q.batches[index % QUEUE_SIZE];
batch->user_data = user_data;
```

# Queue: removing a batch

```
uint32_t index;
do {
    index = q.get;
    if (index == q.put) return QUEUE_EMPTY;
} while (!atomic_cas(&q.get, index, index + 1));

*result = q.batches[index % QUEUE_SIZE];
```

# Race condition #1

```
uint32_t index = atomic_increment(&q.put);

// index points to a batch that has not been fully
// written; data read below is partially stale
batch = &q.batches[index % QUEUE_SIZE];
batch->user_data = user_data;
```

# Race condition #1 - solution

```
// queue_push
batch->user_data = user_data;
memory_barrier();
batch->ready = true;
// queue_pop
while (batch->ready == false) yield();
*result = *batch;
```

# Race condition #2

```
uint32_t index = atomic_increment(&q.put);

// index points to a batch that has been written
// previously but has not been processed;
// old batch data is overwritten (lost)
batch = &q.batches[index % QUEUE_SIZE];
batch->user_data = user_data;
```

# Race condition #2 – solution?

```
// queue_push
while (batch->ready == true) yield();
// queue_pop
while (batch->ready == false) yield();
*result = *batch;
batch->ready = false;
```

# Race condition #3

```
// queue_push
while (batch->ready == true) yield();
```

- Multiple threads wait on the same batch
  - Same local index
  - Can happen if queue overflows
- Race when freeing the batch
  - Multiple threads change the same batch

# Race condition #3

- We need to differentiate the waiting threads
  - Need a globally unique batch identifier...
  - … wait, we have the global batch index!
- The thread with smallest batch index can write
  - All other threads wait until this batch becomes free

# Batch

```
struct Batch {
    /* header: 12 bytes + 4 bytes of padding */
    uint32_t index;
    struct Job job;
    /* job data: 112 bytes */
    char user_data[112];
};
```

# Queue: adding a batch

```
uint32_t index = atomic_increment(&q.put);
batch = &q.batches[index % QUEUE_SIZE];
while (batch_busy(index, batch->index)) yield();
batch->job = job;
batch->user_data = user_data;
memory_barrier();
batch->index = index;
```

# Queue: removing a batch

```
uint32_t index;
do {
    index = q.get;
    if (index == q.put) return QUEUE_EMPTY;
    batch = &q.batches[index % QUEUE_SIZE];
    if (batch->index != index) { yield(); continue; }
} while (!atomic_cas(&q.get, index, index + 1));
*result = *batch;
batch->index = index + 1; // free the batch slot
```

# Queue: batch status

- batch->index == index
  - Batch with global index *index* has been added to the queue but has not been removed
- (index – batch->index) >= QUEUE_SIZE
  - Batch with global index other than *index* has been added to the queue – batch_busy condition!
- Otherwise
  - Batch with global index *index* has been removed

# Queue: batch status

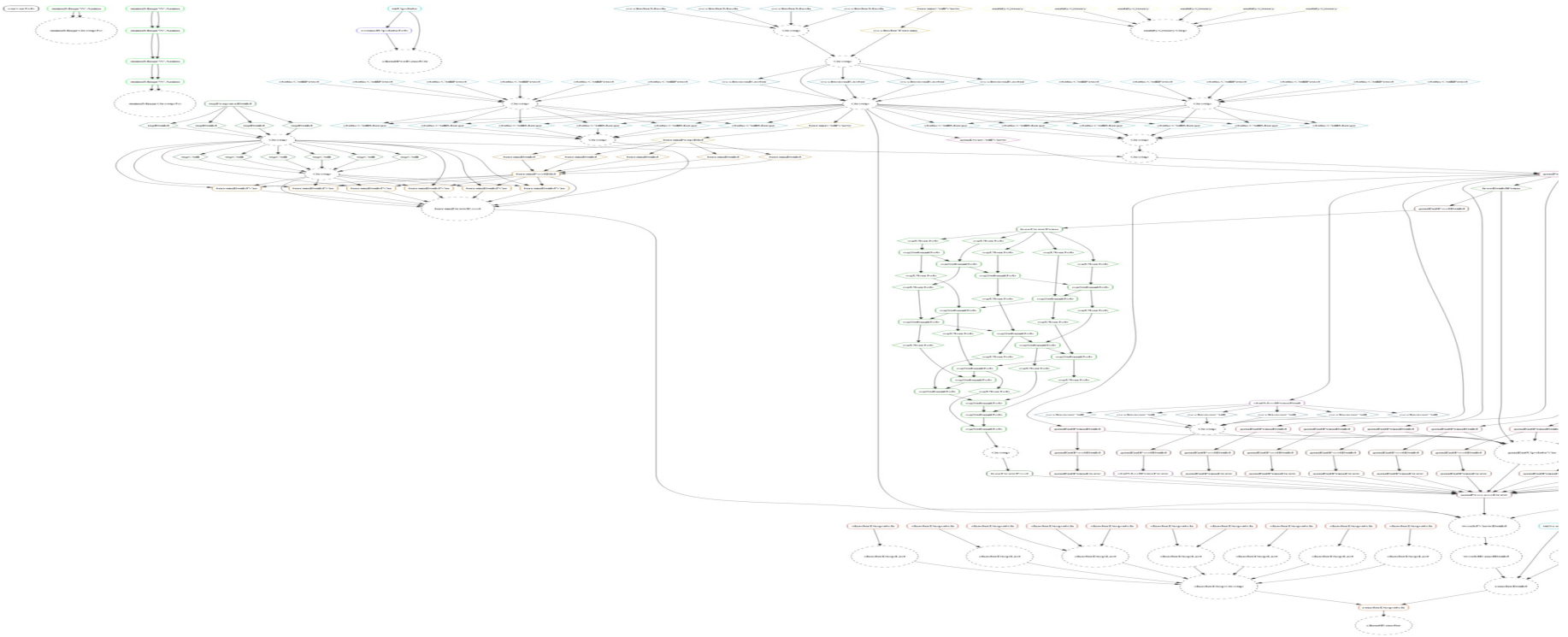
- batch->index == index
  - Batch with global index *index* has been added to the queue but has not been removed
- Let's delay freeing the batch slot until it has been executed
  - Hey, we just got wait_for_batch for free!

```
batch = &q.batches[index % QUEUE_SIZE];
while (batch->index == index) yield();
```

# Scheduler v1.0 – results

- API for task-parallel processing
  - uint32_t push_batch(...);
  - void wait_for_batch(uint32_t index);
  - void wait_for_group(uint32_t group_index);
- 1 atomic operation for push/pop
  - +1 atomic operation for group batch counter
- Simple implementation

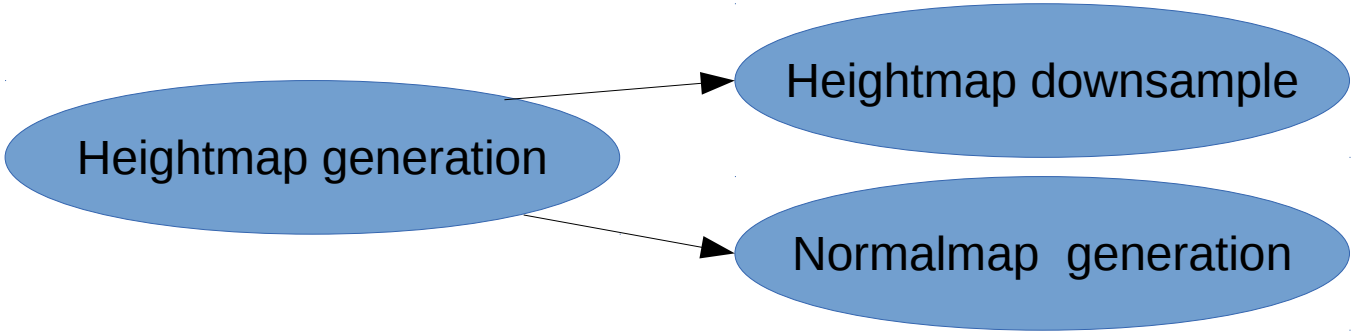
# Batch dependency graph



(EA DICE Frostbite)

# Batch dependency graph

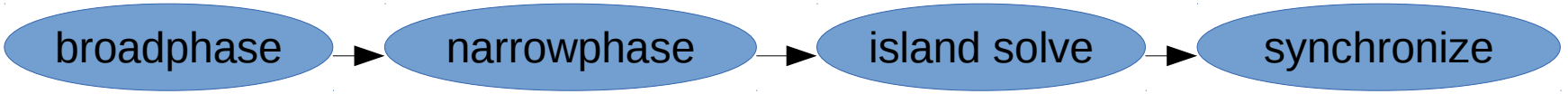
- Graph can be built dynamically
  - Job queues several batches



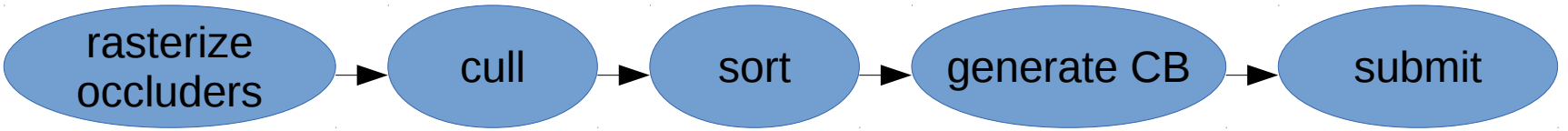

- More efficient than building graph up-front
  - Minimizes working set for scheduler
  - However, scheduler has less freedom

# Data processing patterns

- Common pattern #1: sequence of stages
  - Can have multiple parallel sequences
    - i.e. multiple render queues
- Physics

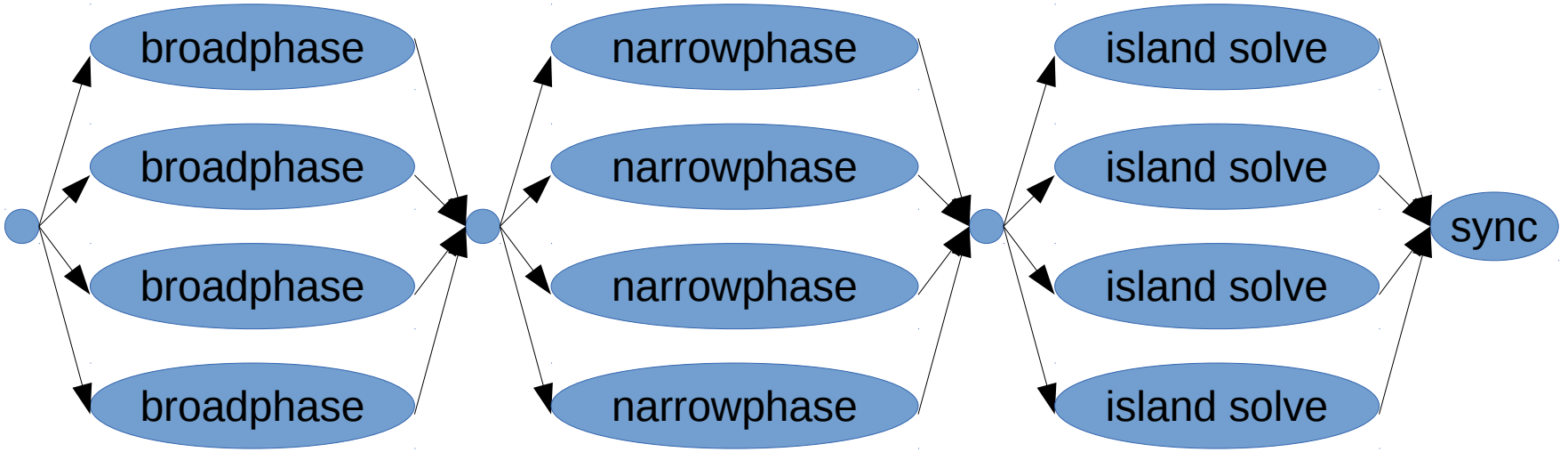



- Render



# Data processing patterns

- Common pattern #2 – data-parallel stages
  - Good scaling given enough data



# Scheduler v2.0

- Data-parallel jobs
  - System has to stay simple
  - Drawing inspiration from GPGPU
- CUDA / DirectCompute / OpenCL
  - Run the same batch with same input arguments on multiple cores
  - Each core knows the invocation index and size
    - blockIdx, blockDim

# Batch block

- Block parameters are added to Batch
  - index – batch index in block; [0..count-1]
  - count – total number of batches in the block
- Example: parallel for

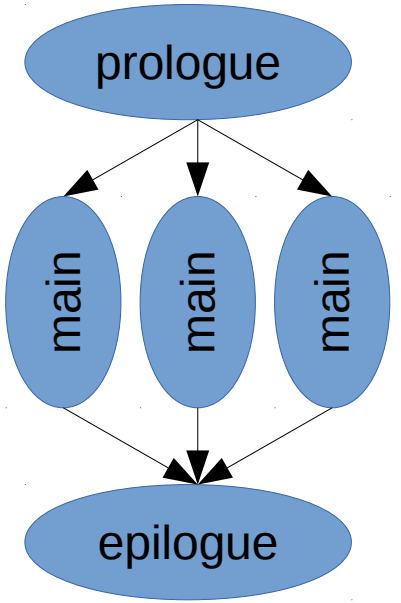
```
start = size / block.count * block.index;
end = size / block.count * (block.index + 1);
for (i = start; i < end; ++i)
    // process element i
```

# Batch block - synchronization

- How do we sync jobs processing one block?
- GPU: __syncthreads
  - Requires preemption
  - Requires specific scheduling constraints
- New concept!
  - Prologue/epilogue for the block

# Prologue and epilogue

- Prologue
  - New job entrypoint
  - Executes once before first main()
- Epilogue
  - New job entrypoint
  - Executes once after last main()
- Input data is the same for all entrypoints



# Batch block - examples

- Parallel for with dynamic load balancing
  - element = atomic_increment(&g_counter);
  - Can use block index to manage scratch memory
- Stage sequence with data-parallel stages
  - Prologue: prepare data (if necessary)
  - Main: process data
  - Epilogue: queue batch for next stage

# Batch

```
struct Batch {
    uint32_t index;
    struct Block block;
    struct Job job;
    /* job data: 112 bytes */
    char user_data[112];
};
```

# Block

```
struct Block {
    volatile uint16_t semaphore;
    uint16_t count;
};
```

# Queue

```
struct Queue {
    Batch batches[QUEUE_SIZE];

    volatile uint32_t get;
    volatile uint32_t get_block;
    volatile uint32_t put;
};
```

# Queue: removing a batch

```
uint32_t index, block, new_index, new_block;
do {
    index = q.get; block = q.get_block;
    if (index == q.put) return QUEUE_EMPTY;
    batch = &q.batches[index % QUEUE_SIZE];
    if (batch->index != index) { yield(); continue; }
    bool last_batch = (block + 1 == batch->block.count);
    new_index = last_batch ? index + 1 : index;
    new_block = last_batch ? 0 : block + 1;
} while (!atomic_cas2(&q.get, index, new_index,
                      &q.get_block, block, new_block));
```

# Queue: batch status

- Batch slot is in use until all jobs finished
  - Including prologue/epilogue
- We can use batch slot for job synchronization!
  - volatile uint16_t semaphore;
  - Batch occupies a full cache line on PS3/Xbox 360
    - 128b size and alignment
    - Can use SPU atomic cache unit

# Prologue execution

- If prologue exists:
  - If batch.index == 0, increment semaphore
  - Otherwise wait until semaphore reaches 1
- If prologue exists we need to wait
  - Limits parallelism
  - Often best to move prologue code to another batch
  - However this is not a problem if prologue is small

# Epilogue execution

- Increment semaphore
- If semaphore was equal to block.count, this is the last batch in the block
  - (compare with block.count-1 if there is no prologue)
  - Execute epilogue if necessary
  - Free the batch slot

    batch->index = index + 1;

# Prologue/epilogue overhead

- For batches with block.count == 1
  - No additional overhead
  - Check block.count before working with semaphore
- Batch with block.count > 1 without prologue
  - +1 atomic operation for executing batch
- Batch with block.count > 1 with prologue
  - +2 atomic operations for executing first batch

# Scheduler v2.0 – results

- API for data-parallel processing
  - block.index and block.count
  - Synchronization (prologue/epilogue)
- Scheduling overhead is still low
  - Same as v1.0 if block.count == 1
- Scheduler code is still simple

- Multithreaded code can be simple
  - Both high level... (parallel for)
  - … and low level (lock-free MPMC FIFO)
- Job scheduler can be simple
  - Simple API
  - Simple code
  - Low overhead

- Keep It Simple, Stupid

?

Arseny “Zeux” Kapoulkine
CREAT Studios
arseny.kapoulkine@gmail.com