Job scheduler
просто, насколько можно
но не проще

Арсений “Zeux” Капулкин
CREAT Studios
a.kapoulkine@creatstudios.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*

Параллельный мир

- Много ядер общего назначения
  - PS3: 6 SPU
  - XBox360: 3 PowerPC ядра
- Разные подходы
  - Параллельность по потокам (thread-parallel)
  - Параллельность по задачам (task-parallel)
  - Параллельность по данным (data-parallel)

Термины

- Job
  - Код, производящий преобразование над данными
- Batch
  - Ссылка на job + данные
- Scheduler
  - Система, выполняющая batch-и

Job

- Функция, обрабатывающая данные
  - Влияние ограничено входными данными
    - Нет доступа к глобальным переменным
- Выполняется много раз на кадр
  - Возможно, одновременно
  - Возможно, на одних и тех же данных
- Нет preemption
  - Требует дополнительной памяти

Scheduler v1.0

- Максимально простой код
  - Достаточно удобный
  - Быстрый
- Добавление батчей на выполнение
- Ожидание выполнения
  - Синхронизация
  - Два уровня: батч и группа батчей
    - 32 группы батчей, uint32 счетчик на группу

Scheduler v1.0

- Глобальная очередь батчей
  - Фиксированный размер очереди
  - Lock-free
  - Операции push и pop
    - Операции копируют данные
      - Устраняет ручной менеджмент батчей
    - Операций empty, size нет
      - Не имеют смысла в многопоточной среде
  - Применима не только в schedulerax!

Lock-free очередь

- Очередь (FIFO) – важный примитив
- Много опубликованных реализаций
  - 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 очередь – 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 очередь – 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 очередь – 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 очередь – 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.            }
```

- Бывают ли lock-free алгоритмы без багов?
  - Бывают.
  - А если кода – 4 экрана?
    - В этом коде была найдена утечка памяти.

Batch

```
struct Batch
{
   /* заголовок: 16 байт */
   ...

   /* данные для job-а: 112 байт */
   char user_data[112];
};
```

Queue

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

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

Queue: принцип работы

- Кольцевой буфер
  - get – первый еще не обработанный батч
  - put – первый еще не записанный батч
  - get != put – в буфере есть батчи
- get/put переполняются
  - Индексы батчей по модулю QUEUE_SIZE
    - QUEUE_SIZE – степень двойки
  - 2^{32} * 3000 тактов = 1.2 часа

Queue: принцип работы

- У каждого батча есть два индекса
  - Глобальный – uint32_t, уникальный
  - Локальный – от 0 до QUEUE_SIZE - 1
- Операции начинаются с выбора батча
  - 1 атомарная инструкция
- Race conditions – в пределах 1 батча
  - ... или между батчами с совпадающим локальным индексом

Queue: добавление батча

```
// атомарный аналог index = q.put++
uint32_t index = atomic_increment(&q.put);

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

Queue: удаление батча

```
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

```
// атомарный аналог index = q.put++
uint32_t index = atomic_increment(&q.put);

// index указывает на еще не заполненный
// элемент; прочитанный batch может быть
// целиком или частично устаревшим

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

Race condition №1 – решение

```
// queue_push
batch->user_data = user_data;
memory_barrier();
batch->ready = true;

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

Race condition №2

```
// атомарный аналог index = q.put++
uint32_t index = atomic_increment(&q.put);

// index указывает на уже заполненный, но
// еще не обработанный элемент; batch
// будет целиком или частично утерян

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

Race condition №2 – решение?

```
// 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();
```

- Много потоков ждут одного батча
  - Один и тот же локальный индекс
  - Случается при переполнении очереди
- Race при освобождении батча
  - Много потоков модифицируют один батч

Race condition №3

- Требуется различать потоки, ждущие освобождения батча
  - Нужен уникальный идентификатор...
  - У нас есть глобальный индекс батча!
- Записать батч может поток с наименьшим глобальным индексом
  - Остальные ждут, пока этот батч не освободится

Batch

```
struct Batch
{
   /* заголовок: 12 байт + 4 байта padding */
   uint32_t index;
   struct Job job;

   /* данные для job-а: 112 байт */
   char user_data[112];
};
```

Queue: добавление батча

```
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: удаление батча

```
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; // освобождение
```

Queue: статус батча

- batch->index == index
  - Батч с глобальным номером index добавлен в очередь, но еще не удален
- (index - batch->index) >= QUEUE_SIZE
  - Батч с глобальным номером != index добавлен в очередь – batch_busy
- В противном случае
  - Батч с глобальным номером index удален

Queue: статус батча

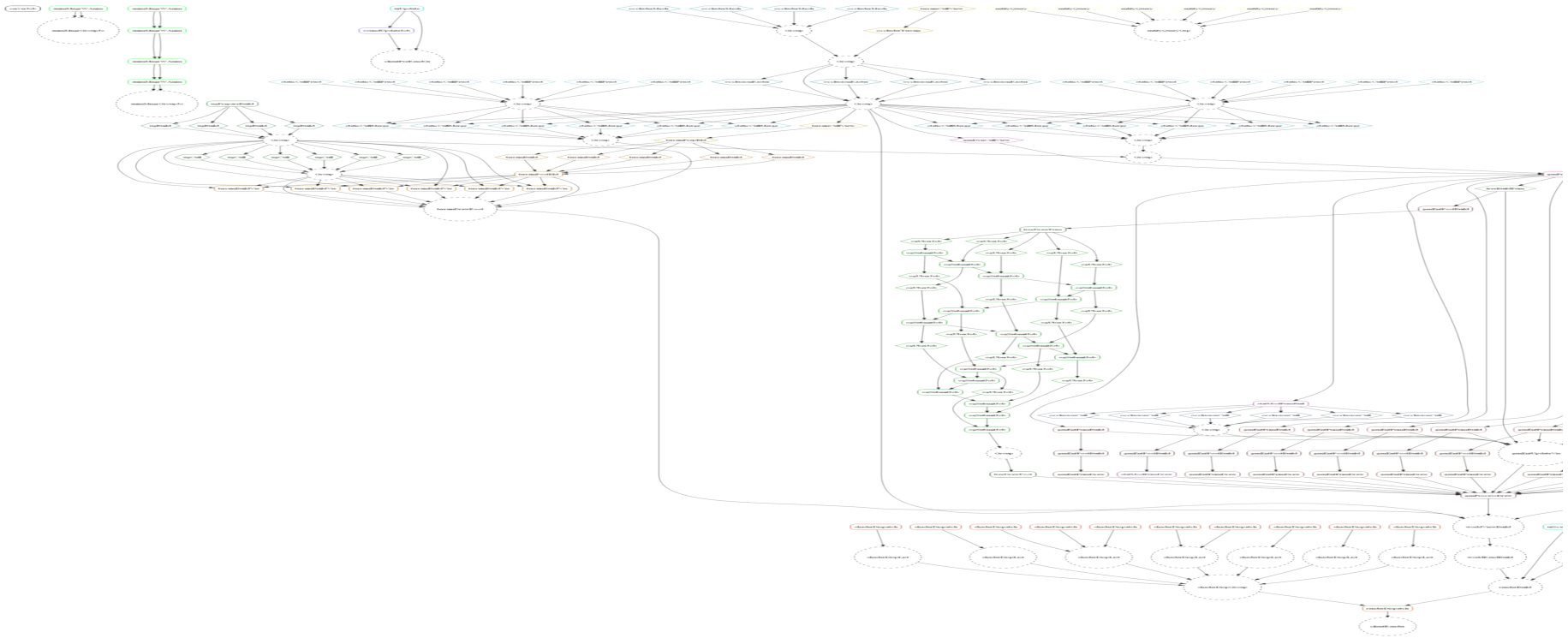
- batch->index == index
  - Батч с глобальным номером index добавлен в очередь, но еще не удален
- Отложим освобождение батча до конца его выполнения
  - Бесплатный wait_for_batch!

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

Scheduler v1.0 - результаты

- Интерфейс для task-parallel задач
  - uint32_t push_batch
  - wait_for_batch
  - wait_for_group
- 1 атомарная операция на push/pop
  - +1 атомарная операция на group counter
- Простая реализация

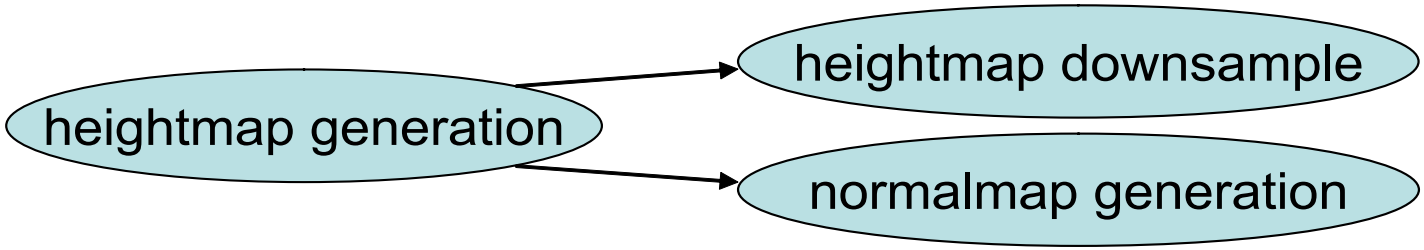
Граф зависимости батчей



(EA DICE Frostbite)

Граф зависимости батчей

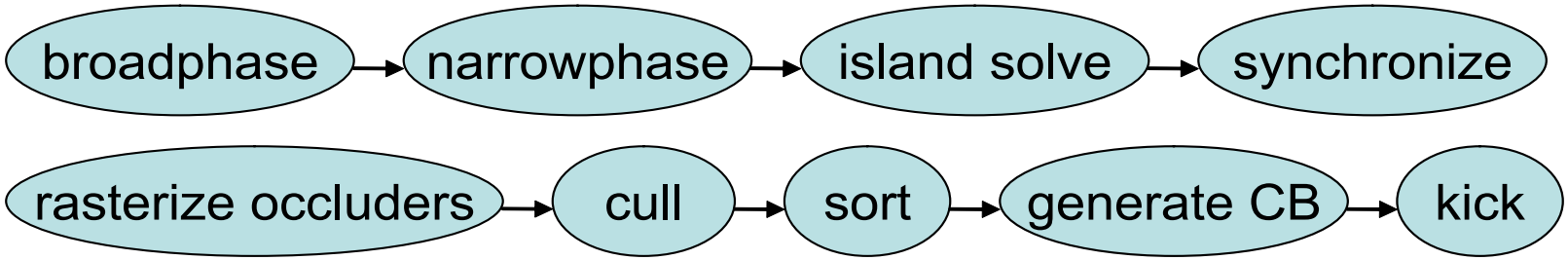
- Динамическое построение графа
  - Job запускает несколько новых батчей



  - Эффективнее, чем добавлять все батчи и расставлять зависимости
    - Working set scheduler-а минимален

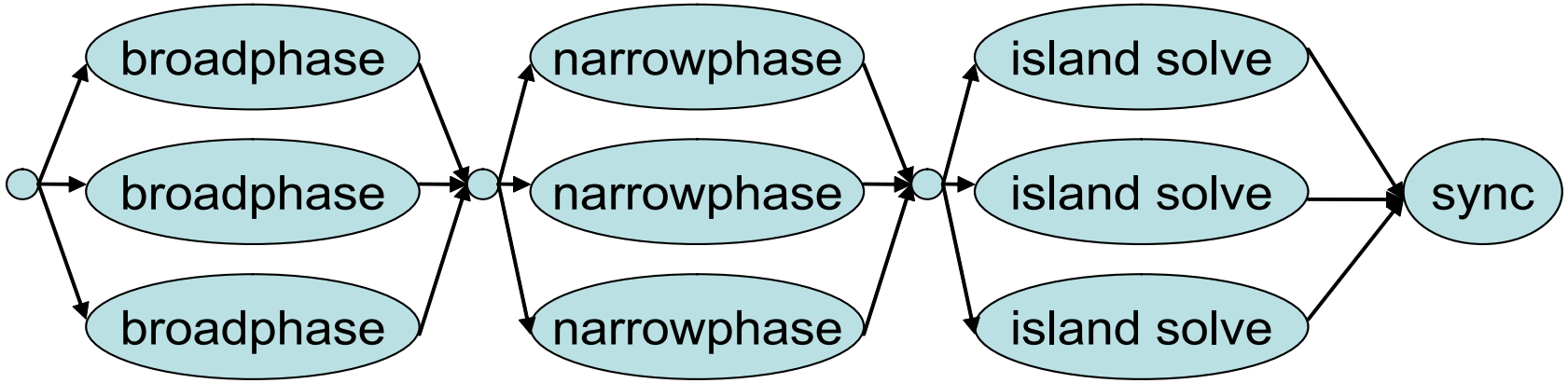
Граф зависимости батчей

- Параллелизм по данным
  - Частый паттерн №1: набор стадий
    - Возможно несколько одновременно работающих цепочек
      - Несколько проходов рендера



Граф зависимости батчей

- Параллелизм по данным
  - Частый паттерн №2: data-parallel стадии
    - Гораздо лучшая масштабируемость



Scheduler v2.0

- Data-parallel jobs
  - Система должна остаться простой
  - Ищем вдохновение в GPGPU
- CUDA / DirectCompute / OpenCL
  - Запуск batch-а с одним набором параметров на N ядрах
  - Каждому ядру приходят данные и индекс в блоке
    - blockIdx, blockDim

Batch block

- В batch добавляются параметры блока
  – index – индекс батча в блоке [0..count)
  – count – общее число батчей в блоке
- Пример: parallel for

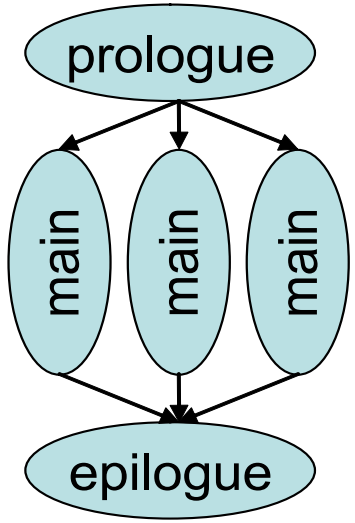
```
start = size / block.count * block.index
end = size / block.count * (block.index + 1)
for (i = start; i < end; ++i)
    // обработка элемента номер i
```

Batch block - синхронизация

- Как синхронизировать job-ы в блоке?
  - GPU: __syncthreads
    - Требует preemption
    - Требует жестких критериев schedul-инга
  - Идея!
    - Пролог и эпилог

Пролог и эпилог

- Пролог
  - Выполняется один раз, до первого main
- Эпилог
  - Выполняется один раз, после последнего main
- Набор данных один и тот же
  - Для всех трех функций



Batch block - примеры

- Динамический parallel for
  - element = atomic_increment(&g_counter)
  - Block index используется для менеджмента scratch памяти
- Data-parallel for
  - Пролог: подготовка данных (если нужно)
  - main: обработка данных
  - Эпилог: запуск batch-а следующей стадии

Batch

```
struct Batch {
   /* заголовок: 16 байт */
   uint32_t index;
   struct Block block;
   struct Job job;

   /* данные для job-а: 112 байт */
   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: удаление батча

```
uint32_t index, block, new_index, new_block;
do {
    index, block = q.get, 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->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: статус батча

- Батч остается занят, пока все job-ы не отработали
  - Включая пролог и эпилог
- Можно использовать тело батча для синхронизации между job-ами!
  - volatile uint16_t semaphore;
  - Batch занимает cache line на PS3/XBox360
    - 128b размер и выравнивание

Обработка пролога

- Если пролог есть:
  - Если batch.index == 0, увеличить семафор
  - В противном случае ждать, пока семафор не станет 1
- Наличие пролога означает ожидание
  - Применяется редко
    - Обычно лучше вынести пролог в отдельный батч

Обработка эпилога

- Увеличить семафор на 1
- Если семафор был равен block.count, то это последний батч
  - В случае, если пролога нет, семафор будет равен block.count – 1
  - Запустить эпилог, если нужно
  - Освободить батч

Дополнительный overhead

- Батч с block.count == 1
  - Дополнительные действия не требуются
    - Проверка count перед обновлением семафора
- Батч с block.count > 1 без пролога
  - +1 атомарная инструкция на запуск
- Батч с block.count > 1 с прологом
  - +2 атомарных инструкции на запуск первого батча

Scheduler v2.0 - результаты

- Интерфейс для data-parallel задач
  - block.index и block.count
  - Синхронизация (пролог и эпилог)
- Все еще низкий overhead
  - В случае не data parallel jobs – как и в v1.0
- Все еще простой код

- Многопоточный код бывает простым
  - Как код высокого уровня... (parallel_for)
  - … так и код низкого уровня (MPMC FIFO)
- От job scheduler-а нужно немного
  - Простой интерфейс
  - Низкий overhead

- Будьте проще, и к вам потянутся такты.

?

Арсений “Zeux” Капулкин
CREAT Studios
a.kapoulkine@creatstudios.com