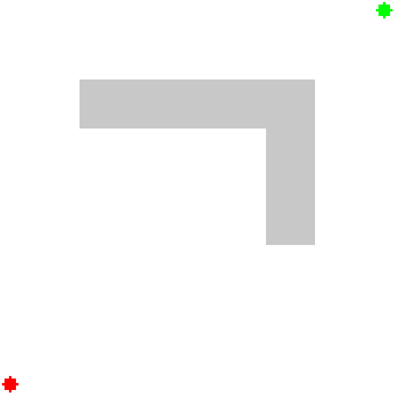
A* GPGPU Implementations for Path-Finding

Florian Klemme

July 26, 2017

Overview

The A* search algorithm



- **A*** is a *best-first search* algorithm
- Chooses minimal node $f(n) = g(n) + h(n)$
- Needs *priority queue* to select next node
- Needs look-up data structure for visited nodes

Multi-agent A* vs. parallel GA*

Problem: It's hard to parallelize the priority queue!

Multi-agent A* [1]

- Per-agent local priority queue
- Node and edge information can be shared. Open and closed list has to be stored for each agent.
- **Local memory is a scarce resource!** I've seen 16, 32 and 48 kilobyte.

Parallel GA* [2]

- Multiple independent priority queues
- Expand multiple nodes at the same time
- The challenge is **handling of duplicate nodes!**
- Again, bound by memory

Priority queue: Dispatch memory access

```
typedef struct {
    __local  uint_float *localMem;
    const    size_t      localSize;
    __global uint_float *globalExt;
             size_t      size;
} OpenList;

uint_float _read_heap(OpenList *open, size_t index) {
    return index < open->localSize ?
        open->localMem[index] :
        open->globalExt[index - open->localSize];
}

void _write_heap(OpenList *open, size_t     index,
                                 uint_float value) {
    if (index < open->localSize)
        open->localMem[index] = value;
    else
        open->globalExt[index - open->localSize] = value;
}
```

Priority queue as binary heap

We need the common...

- push
- top & pop

But also...

- find
- update

```
void push(OpenList *open, uint value, float cost) {
    _push_impl(open, &open->size, value, cost);
}

void update(OpenList *open, size_t index, uint  value,
                                          float cost) {
    _push_impl(open, &index, value, cost);
}
```

Binary heap: Push

```
void _push_impl(OpenList *open, size_t *size,
                uint      value, float   cost) {
    size_t index = (*size)++;

    while (index > 0) {
        size_t parent = (index - 1) / 2;

        uint_float pValue = _read_heap(open, parent);
        if (cost < pValue.second) {
            _write_heap(open, index, pValue);
            index = parent;
        } else
            break;
    }

    _write_heap(open, index, (uint_float){value, cost});
}
```

Binary heap: Pop

```
void pop(OpenList *open) {
    uint_float value = _read_heap(open, --(open->size));
    size_t     index = 0;

    while (index < open->size / 2) {
        size_t     child  = index * 2 + 1;
        uint_float cValue = _read_heap(open, child);
        if (child + 1 < open->size) {
            uint_float c1Value = _read_heap(open,
                                            child + 1);
            if (c1Value.second < cValue.second) {
                ++child;
                cValue = c1Value;
            }
        }

        if (cValue.second < value.second) {
            _write_heap(open, index, cValue);
            index = child;
        } else break;
    }
```

Binary heap: Find

This could certainly be improved! Now it's basically a simple breadth-first search.

```
uint find(OpenList *open, uint value) {
    for (uint index = 0; index < open->size; ++index) {
        uint_float iValue = _read_heap(open, index);
        if (iValue.first == value)
            return index;
    }

    return open->size;
}
```

Alternative: Except duplicate entries and ignore them later on. Would cost extra memory!

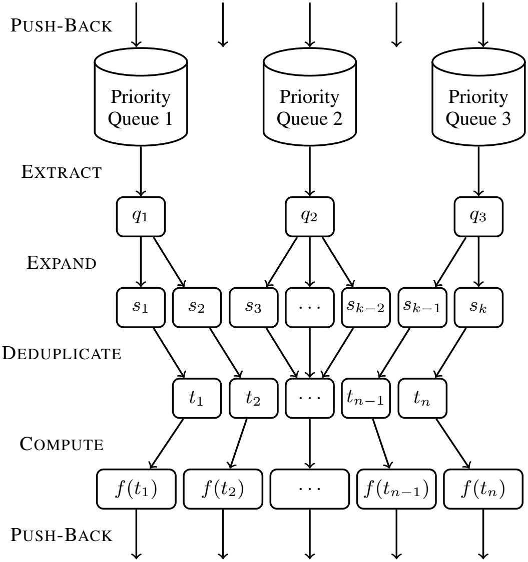
Multi-agent A* kernel

```
push(&open, source, 0.0f);
while (open.size > 0) {
    const uint current = top(&open); pop(&open);
    /* ... */
    const uint2 range = adjacencyMap[current];
    for (uint edge = range.x; edge != range.y; ++edge) {
        /* ... */
        const uint nbIndex = find(&open, nbNode);

        if (nbIndex < open.size &&
            nbInfo.totalCost <= nbTotalCost) continue;
        /* ... */

        if (nbIndex < open.size)
            update(&open, nbIndex, nbNode, nbTotalCost +
                                           nbHeuristic);
        else
            push(&open, nbNode, nbTotalCost + nbHeuristic);
    }
}
```

Data flow in parallel GA*



Run kernels, in order:

- clearSList
- extractAndExpand
- clearTList
- duplicateDetection
- exclusive_scan
- compactTList
- computeAndPushBack

Image from [2].

GA* Pseudocode

Algorithm 1 GA*: Parallel A* search on a GPU

```
procedure GA*(s, t, k)
                                  ▷ find the shortest path from s to t with k queues
    Let {Q_i}_{i=1}^k be the priority queues of the open list
    Let H be the closed list
    PUSH(Q_1, s)
    m ← nil                       ▷ m stores the best target state
    while Q is not empty do
        Let S be an empty list
        for i ← 1 to k in parallel do
            if Q_i is empty then
                continue
            end if
            q_i ← EXTRACT(Q_i)
            if q_i.node = t then
                if m = nil or f(q_i) < f(m) then
                    m ← q_i
                end if
                continue
            end if
            S ← S + EXPAND(q_i)
        end for
        if m ≠ nil and f(m) ≤ min_{q∈Q} f(q) then
            return the path generated from m
        end if
        T ← S
        for s' ∈ S in parallel do
            if s'.node ∈ H and H[s'.node].g < s'.g then
                remove s' from T
            end if
        end for
        for t' ∈ T in parallel do
            t'.f ← f(t')
            Push t' to one of priority queues
            H[t'.node] ← t'
        end for
    end while
end procedure
```

Run kernels, in order:

- clearSList
- extractAndExpand
- clearTList
- duplicateDetection
- exclusive_scan
- compactTList
- computeAndPushBack

Pseudocode from [2].

Use of synchronization mechanisms: return code

```
__kernel void extractAndExpand(/* ... */
                               __global uint *returnCode) {
    /* ... */
    if (openSize == 0) {
        atomic_min(returnCode, 2);
        return; // failure: no path found!
    }

    /* ... */
    if (current == destination) {
        atomic_min(returnCode, 0);
        return; // success: path found!
    }

    /* ... */
    atomic_min(returnCode, 1); // still running...
}
```

Use of synchronization mechanisms: inserting elements

```
__kernel void duplicateDetection(/* ... */) {
    /* ... */
    if (nodeInfo.closed == 1 &&
        nodeInfo.totalCost < current.totalCost)
        return; // better candidate already in open list

    const uint hash = current.node % hashTableSize;
    const uint old = atomic_xchg(hashTable + hash,
                                 current.node);
    if (old == current.node)
        return; // node has already been added

    __global Info *tlist =
        tlistChunks + GID.x * slistChunkSize;
    const uint index = atomic_inc(tlistSizes + GID.x);
    tlist[index] = current;
}
```

Updating information: Transaction approach

```
#pragma OPENCL EXTENSION cl_khr_int64_base_atomics : enable

typedef struct {
    uint  closed;      uint node;
    float totalCost;   uint predecessor;
} Info;

// Update totalCost and pred. as one 64 bit transaction.
ulong *curCostPred = (ulong *) &current.totalCost;
__global ulong *infoCostPred =
    (__global ulong *) &info[current.node].totalCost;
ulong oldCostPred = atom_xchg(infoCostPred, *curCostPred);
float *oldCost = (float *) &oldCostPred;

// assert: current.totalCost > 0.0f
while (*oldCost != 0.0f && *oldCost < current.totalCost) {
    // The old entry was better. Swap back!
    *curCostPred = oldCostPred;
    oldCostPred = atom_xchg(infoCostPred, *curCostPred);
}
```

Lessons learned

- OpenCL is hard to make portable and reliable
  - Compiler crashes
  - Wrong work sizes (even through API)
- Performance depends on actual hardware
- CPU A* is hard to beat, random obstacles have a huge impact

Multi-agent A*

- Work items $>$ local memory
- Make use of __*private* memory?

Parallel GA*

- Needs lots of memory, highly dependent on random obstacles
- Still lots of room for improvement!

Demo configuration on my secondary computer

```
OpenCL device: GeForce GT 520
 ----- CPU reference run...
CPU time for 2500 runs: 0.545225 seconds
 ----- GPU A* run...
GPU time for 2500 runs:
 - Upload time: 0.0184138 seconds
 - Kernel runtime: 1.05191 seconds
 - Download time: 0.00187089 seconds
 ----- CPU reference run...
CPU time for graph (500, 500): 0.655936 seconds
 ----- GPU GA* run...
GPU time for graph (500, 500):
 - Upload time: 0.00613427 seconds
 - Kernel runtimes:
   - CompactTList: 0.0401437 seconds
   - ComputeAndPushBack: 0.129521 seconds
   - DuplicateDetection: 0.102979 seconds
   - ExtractAndExpand: 0.358814 seconds
   - compute::exclusive_scan: 0.505029 seconds
 - Download time: 0.000844134 seconds
GPU GA*: Gold test failed!
 - Path length CPU: 580, GPU: 581
 - Path cost CPU: 728.528, GPU: 742.369
```

Boost Compute

Pure Template library, brings RAII, type-safety, common interface

```
// get the default compute device
compute::device gpu = compute::system::default_device();

// create a compute context and command queue
compute::context ctx(gpu);
compute::command_queue queue(ctx, gpu);

// generate random numbers on the host
std::vector<float> host_vector(1000000);
std::generate(host_vector.begin(), host_vector.end(), rand);

// create vector on the device
compute::vector<float> device_vector(1000000, ctx);

// copy data to the device
compute::copy(
    host_vector.begin(), host_vector.end(), device_vector.begin(), queue
);

// sort data on the device
compute::sort(
    device_vector.begin(), device_vector.end(), queue
);

// copy data back to the host
compute::copy(
    device_vector.begin(), device_vector.end(), host_vector.begin(), queue
);
```

https://github.com/boostorg/compute

References

[1] Andre Silva, Fernando Rocha, Artur Santos, Geber Ramalho, and Veronica Teichrieb. Gpu pathfinding optimization. In *Games and Digital Entertainment (SBGAMES), 2011 Brazilian Symposium on*, pages 158–163. IEEE, 2011.

[2] Yichao Zhou and Jianyang Zeng. Massively parallel a* search on a gpu. In *AAAI*, pages 1248–1255, 2015.

In case you want to check it out:
`https://github.com/Kruecke/ocl-astar`