Project Proposal

Assume we have a particle type that we want to update/simulate in a not a priori known way.

```c
struct Particle {
    float position[3];
    float velocity[3];
    float mass;
    //...
};
```

To control their behavior we have a set of rules that can affect them that are potentially dynamic in amount and appearance (the userdata pointers would contain any instance specific information, magnitudes of forces, centers of spheres...)

```c
struct Rule {
    void (*apply)(Particle *p, void *userdata);
    void *userdata;
};
void gravity(Particle *p, void *userdata);
void central_force(Particle *p, void *userdata);
void floor_collision(Particle *p, void *userdata);
void sphere_collision(Particle *p, void *userdata);
//...
```

the generic approach to apply these rules would then look something like:

```c
Particle *particles = malloc(sizeof(struct Particles)*N);
Rule rules[] = {
    {gravity, gravity_userdata(-9.81)},
    {central_force, center_userdata(x1,y1,z1)},
    {sphere_collision, sphere_userdata(x2,y2,z2)},
    {sphere_collision, sphere_userdata(x3,y3,z3)},
    ...
    {NULL, NULL}
};
apply(particles, rules, N);
```

where apply in the naive form would be implemented as

```c
void apply(Particle *particles, Rule *rules, int N) {
    for(int i = 0;i<N;++i) {
        for(int j = 0;rules[j].fun != NULL;++j) {
            rules[j].apply(particles + i, rules[j].userdata);
        }
    }
}
```

This is fairly inefficient due to a high amount of function calls etc. Also there are missed opportunities for optimization since the bulk operations over the particles are predestined for SPMD type parallelism while the one particle at a time implementation doesn't account for that. Precombining these on the other hand takes away the flexibility of changing the amount of colliders/force sources etc. at run time. Another aproach would be to implement the rules themselves as bulk operations and call then im sequence. This way we potentially end up reading and writing the whole particle array for every rule and wasting bandwidth and very likely end up memory bound.

The proposed solution approach is therefore to compile the rules array into an optimized function at run time. possible optimizations are:

- processing 4, 8 or more particles at a time by using SIMD instructions.
- common subexpression elimination between rules.

- combine load and store operations to lower bandwidth requirements
- interleaving of rule computations to improve ILP
- superoptimizing common rules combinations by online profiling all possible instruction orders (or other ”simple enough” transformations)

To achieve this rules would be specified as some simplified Vector SSA/proto assembly instead of functions. The `central_force` rule for example could look something like:

```
SSA_node central_force_program[] = {
    {LOAD,  VEC(0),     POSITION},
    {LOAD,  VEC(1),     CENTER},
    {SUB,   VEC(2),     VEC(1),     VEC(0)},    // diff = pos - center
    {DOT,   SCAL(0),    VEC(2),     VEC(2)},    // r2 = dot(diff, diff)
    {SQRT,  SCAL(1),    SCAL(0)},               // r = sqrt(r2)
    {MUL,   SCAL(2),    SCAL(1),    SCAL(0)},   // r3 = r*r2
    {DIV,   VEC(3),     VEC(2),     SCAL(2)},   // force = diff/r3
    {LOAD,  VEC(4),     POSITION},
    {ADD,   VEC(5),     VEC(4),     VEC(3)},    // velocity += force
    {STORE, VELOCITY,   VEC(3)}
};
```

The rule compiler can then concatenate all the update rules and apply optimization passes and finally generate code. This essentially equates to writing a optimizing backend to this fairly constrained problem. This seems doable to me since the supported instructions can be kept fairly minimal and at first branch free. I think an improvement over a generic dynamic implementation is certainly achievable and the high goal would be to match/overtake statically optimized versions of rule combinations which would be obtained by inlining test rules and having a regular compiler (or even ISPC) compile it as reference.