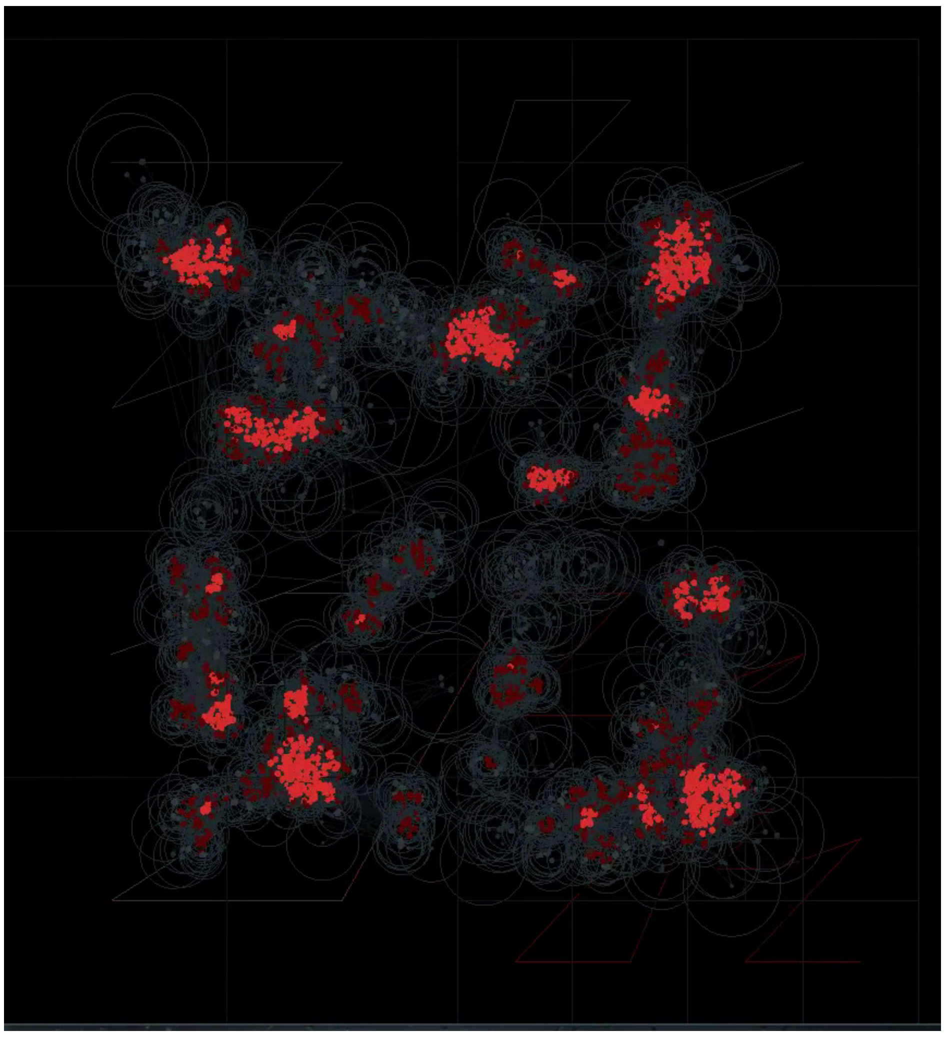
AstroCuda

github.com/emcf/AstroCuda

A description of the inner workings of a 2D C CUDA prototype of Smoothed Particle Hydrodynamics for astrophysical simulations on the GPU.

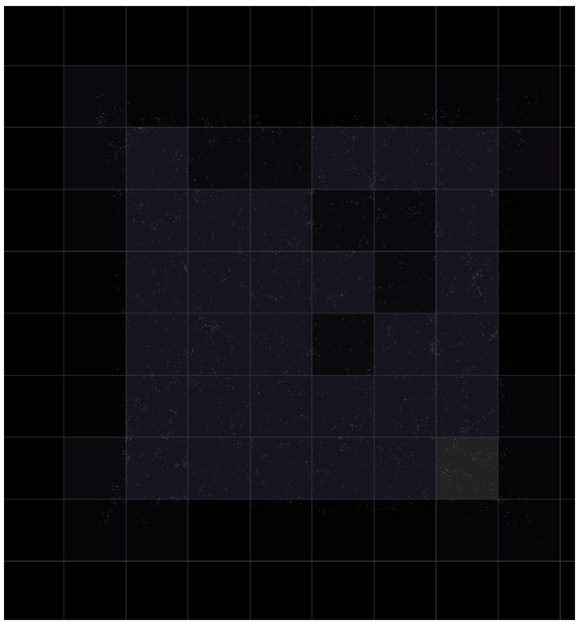
Emmett McFarlane



Department of Physics and Astronomy
McMaster University
Summer of 2018

1 Existing GPU SPH Implementations

GPUs are designed for operating on uniform 2D arrays of pixels, so naturally, using contiguous 1D, 2D, or 3D data within CUDA will allow for very fast computations (i.e., fast neighbour finding). Hérault et al. uses this method in GPUSPH for fluids with roughly uniform density.[1] Below is a 2D demo of a uniform N-body CUDA simulation I made on the order of 10^4 particles.



Astrophysical simulation can not be done like this, since density is too variable. Recursive space partitioning methods like K-d trees, octrees, etc. must be used. Since there seems to be no existing implementation of this using the GPU, the following is a report on how this can be done efficiently.

[1]Hérault, A., G. Bilotta, R.A. Dalrymple, SPH on GPU with CUDA, Journal of Hydraulic Research, 48 (Extra Issue), 74-79, 2010.

2 CPU Quadtree

Before any physical calculations can be made, particles must be recursively divided into quadrants in order to reduce computational complexity later on from $O(N^2)$ to $O(NN_{neibs})$. Quads are further divided into child quads until particles per quad $\leq$ threads per block in CUDA version

```
void quad::divide()
{
    foreach (child)
    {
        if (child.containedParticlesCount > MAX_PARTICLES_PER_BUCKET)
            child.divide();
    }
}
```

For each leaf node, a depth-first tree traversal is performed to determine if a quad is a neighbour based on the following:

$$distance \leq h_{cell} = 2 * max(\text{quad.width, quad.height}) + 10$$

```
void quad::neibSearchTraversal(int i)
{
    if (distance(quad, quadList[i]) < h_cell)
    {
        if (quadList[i] is leaf)
            neibLeafIndices.push_back(quadList[i].leafIdx);
        else
            foreach (child in quadList[i])
                neibSearchTraversal(child.index);
    }
}
```

3 CPU $\rightarrow$ GPU

non-leaf-node quads are removed. Recursive data types will not work for us on the GPU.

```
void quadtree::eliminateBranches()
{
    for (i = 0; i < quads; i+=0)
    {
        if (quad is leaf)
            i++;
        else
            quadList.erase(i);
    }
}
```

A new quad list is assembled using a more memory-conservative data type appropriate for the GPU.

```
struct deviceQuad
{
    float hCell;
    int neibBucketCount;
    int* d_neibBucketsIndices;
    int firstContainedParticleIdx;
    int containedParticleCount // Must be < MAX_PARTICLES_PER_BUCKET
};
```

Particles are also assembled into a list using a memory-conservative data type. Holding all of a particle's data in a single struct will be useful later on when we want to load particles into the GPU's cache memory.

```
struct deviceParticle
{
    float particleData[PARTICLE_DATA_LENGTH];
};
```

These deviceParticles are assembled into a list following the order of a Morton space-filling curve. The next page shows how the particleData is organized into a float array.

0	pos x
1	pos y
2	pos z
3	Idx
4	vel x
5	vel y
6	vel z
7	mass
8	smoothing length
9	density
10	pressure

The CPU → GPU data transfer is as follows,

```
void transfer(deviceQuad* d_deviceQuadList, deviceParticle* d_deviceParticleList)
{
    deviceParticle* h_deviceParticleList = new deviceParticle[N];
    deviceQuad* h_deviceQuadList = new deviceQuad[quadCount];

    int MortonCounter = 0;
    for (i = 0; i < quadList; i++)
    {
        h_deviceQuadList[i].neibSearchRadius = hCell;
        h_deviceQuadList[i].containedParticleCount = quadList[i].containedParticleCount;
        h_deviceQuadList[i].neibBucketCount = quadList[i].neibBucketCount;

        size_t mem = quadList[i].neibBucketCount * sizeof(int);
        cudaMalloc((void**) &(h_deviceQuadList[i].d_neibIndices, mem, cudaMemcpyHostToDevice);
        cudaMemcpy(h_deviceQuadList[i].d_neibIndices, quadList[i].neibLeafIndices, mem);

        for (j = 0; j < containedParticles; j++)
        {
            deviceParticle h_particle;
            h_particle.particleData[0] = particleSystem.pos[particleIdx].x;
            h_particle.particleData[1] = particleSystem.pos[particleIdx].y;
            ...
            ...
            h_deviceParticleList[MortonCounter++] = h_particle;
        }
    }

    size_t quadListMem = quadCount * sizeof(deviceQuad);
    cudaMemcpy(d_deviceQuadList, h_deviceQuadList, quadListMem, cudaMemcpyHostToDevice);

    size_t pListMem = N * sizeof(deviceParticle);
    cudaMemcpy(d_deviceParticleList, h_deviceParticleList, pListMem, cudaMemcpyHostToDevice);
};
```

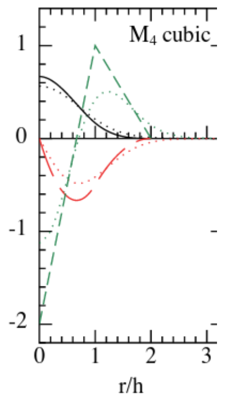
4 SPH Equations

Density is computed with Daniel Price's M_4 cubic spline function.[2]

$$p_i = \sum_j m_j W(|\vec{x_i} - \vec{x_j}|, h_i)$$

$$W(r,h) = \frac{1}{h^2} w(r,h)$$

$$w(r,h) = \sigma \begin{cases} \frac{1}{4}(2-\frac{r}{h})^3 - (1-\frac{r}{h})^3 & 0 \le \frac{r}{h} < 1 \\ \frac{1}{4}(2-\frac{r}{h})^3 & 1 \le \frac{r}{h} < 2 \\ 0 & \frac{r}{h} > 2 \end{cases}$$



However, before density can be computed this way, the following equation must be satisfied in order to find an appropriate value for h such that the mass within every smoothing kernel is roughly constant.

$$\zeta(h_i) = m_i(\frac{\eta}{h_i})^2 - \sum_j m_j W(\vec{x_i} - \vec{x_j}, h_i)$$

The Newton-Raphson method can be iterated to find h. That is,

$$h_{new} = h - \frac{\zeta(h)}{\frac{\partial \zeta}{\partial h}(h)}$$

where

$$\frac{\partial \zeta}{\partial h} = \frac{-2m_i\eta^2}{h^3} - \frac{\partial p}{\partial h}$$

$$\frac{\partial p}{\partial h} = \sum_j m_j \frac{\partial W}{\partial h}$$

$$\frac{\partial W}{\partial h} = \frac{-2}{h^3} w(r,h) + \frac{1}{h^2}\frac{\partial w}{\partial h}$$

$$\frac{\partial w}{\partial h} = \sigma \begin{cases} \frac{-9r^3+12hr^2}{4h^4} & 0 \le \frac{r}{h} < 1 \\ \frac{3r(2h-r)^2}{4h^4} & 1 \le \frac{r}{h} < 2 \\ 0 & \frac{r}{h} > 2 \end{cases}$$

[2]Price, Daniel J. "Smoothed Particle Hydrodynamics and Magnetohydrodynamics." Journal of Computational Physics, vol. 231, no. 3, 2012, pp. 759–794., doi:10.1016/j.jcp.2010.12.011.

Acceleration is calculated using

$$\frac{dv_i}{dt} = -\sum_i m_j(\frac{P_i + P_j}{p_i p_j})\nabla_i W_{ij}$$

where

$$P_i = (\gamma - 1)p_i u_i$$

and

$$\nabla_i W(\vec{r}, h_i) = \frac{\vec{r}}{{h_i}^4}\frac{1}{q}\frac{dw}{dq}$$

$$\frac{dw}{dq} = \sigma \begin{cases} \frac{3q(3q-4)}{4} & 0 \leq q < 1 \\ \frac{-3(q-2)^2}{4} & 1 \leq q < 2 \\ 0 & q > 2 \end{cases}$$

$$q = \frac{||\vec{r_{ij}}||}{h_i}$$

Solving these equations can be straightforward on the CPU, however we must be very conservative with memory access on the GPU. The calculations are split into two CUDA kernels; one for computing density, finding h, and computing $\nabla_i W_{ij}$, and one for computing acceleration and integrating positions. The W gradients are passed from kernel to kernel via global memory, in a float2 array:

$$\text{float2* } d\_gradW = \begin{vmatrix} \nabla W_{0_0} & \nabla W_{1_0} & \ldots & \nabla W_{N_0} \\ \nabla W_{0_1} & \nabla W_{1_2} & \ldots & \nabla W_{N_3} \\ \nabla W_{0_2} & \nabla W_{1_3} & \ldots & \nabla W_{N_{Nneibs}} \end{vmatrix}$$

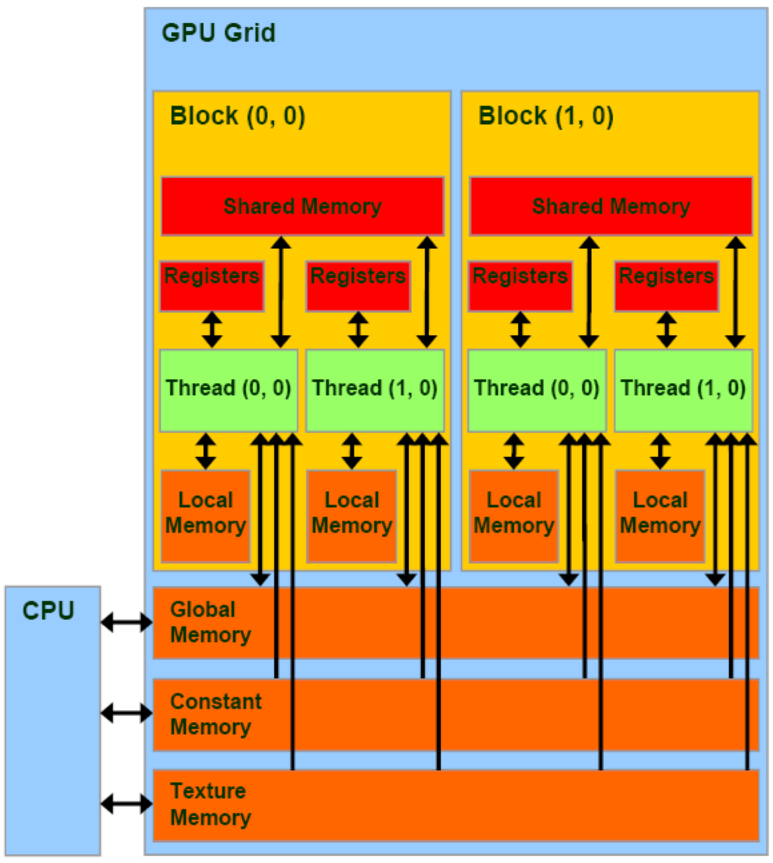
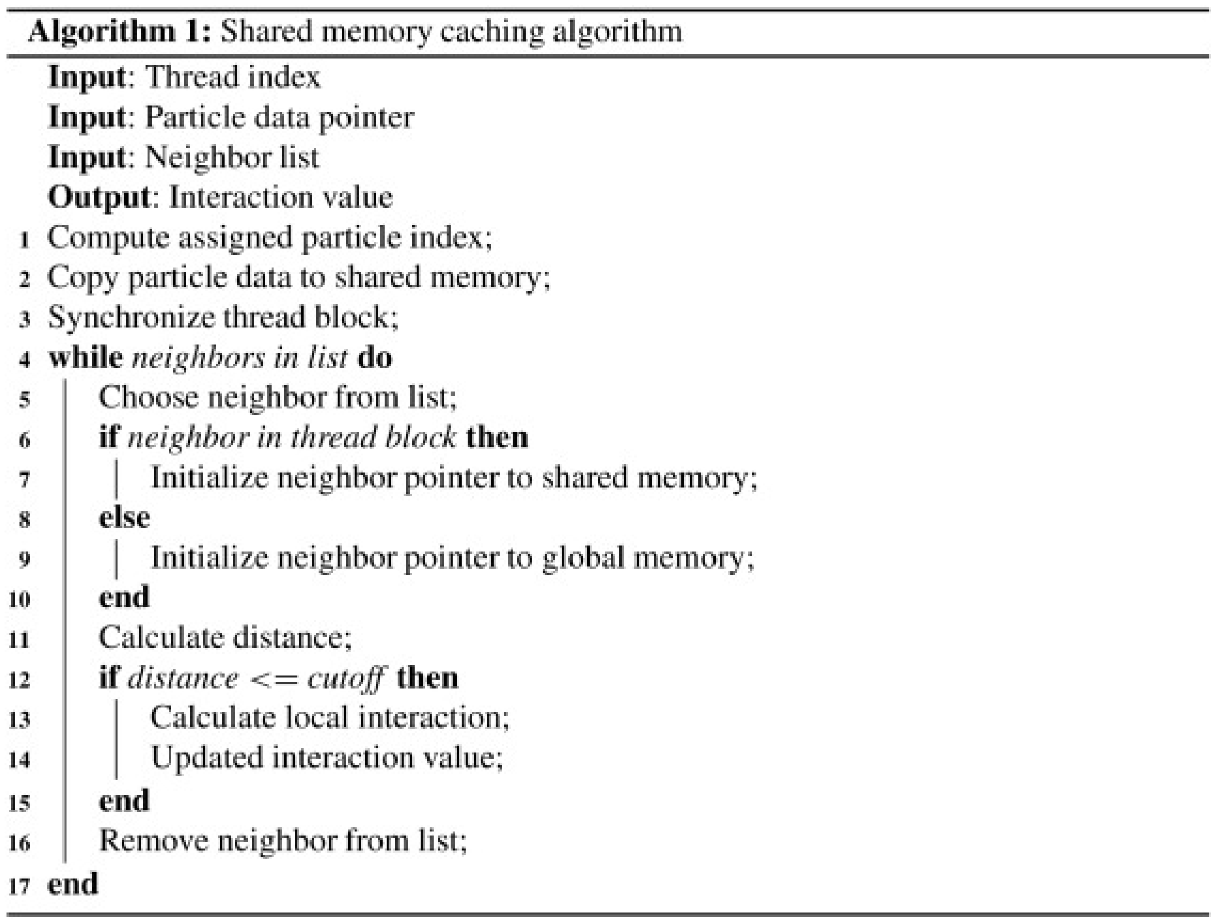


Figure 1: Visualization of CUDA's memory model from the NVidia CUDA C Programming Guide. Arrow length represents time to access. Funny enough, NVidia's CUDA model is so well paired to their hardware that many images simply overlay these diagrams on top of close-up images of each multiprocessor.

5 GPU Memory

- The GPU's **global memory** is large (about 12GB for an NVidia Tesla P100) but slow. It benefits from congruent memory access pattern (i.e., thread 1 accesses array[1], thread 2 accesses array[2], etc.)
- Similar to this is **texture memory**, however texture memory is read-only. It is slighty faster for accessing 2D array data (although this SPH simulation is 2D, there are actually no 2D arrays since data is flattened into a space-filling curve, thus texture memory is not of much use for us. in GPUSPH, they use a uniform grid to contain particles rather than a quadtree. A uniform cell grid is clearly well represented by 2D texture memory)
- Each thread in the newest CUDA version has 255 4-byte **registers**. This is the fastest memory type.
- **Local memory** represents global memory with specially designed data access that allows it to be very fast (100x faster than global memory) for a certain thread. Each thread can have local memory. We cannot control what goes into local memory. Once a thread runs out of register space, things go into local memory. Think of it as a "pre-planned" cache-miss. This is bad since local memory is relatively slow, so try to only let each thread use 255 ints/floats of data. In my current model, register use per thread is on the order of about 50 registers.
- Each block has its own 48KB cache called **shared memory** which can be just as fast as register memory if used correctly. The memory is divided into **banks**, which each contain 32 separate 4-byte sections called **words**. If different threads request different pieces of data (words) from the same bank, a **bank conflict** will occur, meaning the requests will be serialized. This is to be avoided. If all threads of a **warp** request the same word from the same bank, then a **broadcast** will occur. Likewise, if any number of threads in a block request the same word from the same bank, a **multicast** occurs. In both of these cases, a single read from shared memory is performed and sent to all threads requesting it. Winkler et al. designed a model (figure 2) using shared memory caching to efficiently access neighbour lists in SPH[3]. I have adjusted the model to also make use of multicasts. It should be noted that this model contains contains unavoidable bank conflicts.

[3]Winkler, Daniel, et al. "GpuSPHASE — A Shared Memory Caching Implementation for 2D SPH Using CUDA." Computer Physics Communications, vol. 213, 2017, pp. 165–180., doi:10.1016/j.cpc.2016.11.011.



Algorithm 1: Shared memory caching algorithm

```
Input: Thread index
Input: Particle data pointer
Input: Neighbor list
Output: Interaction value
Compute assigned particle index;
Copy particle data to shared memory;
Synchronize thread block;
while neighbors in list do
    Choose neighbor from list;
    if neighbor in thread block then
        Initialize neighbor pointer to shared memory;
    else
        Initialize neighbor pointer to global memory;
    end
    Calculate distance;
    if $distance <= cutoff$ then
        Calculate local interaction;
        Updated interaction value;
    end
    Remove neighbor from list;
end
```

Figure 2: Winkler et al. used this model in GPUSPHase to perform efficient GPU SPH computations. This is the general model I have used in the code below, however I have also included a Newton-Raphson scheme as well as an integration scheme. Note that the if-else branching is not actually branching, since logically following the design of this GPU SPH solver kernel will lead one to the conclusion that all threads within a block will always follow the same branch.

```
__global__ void SPHSolverKernel(deviceQuad* d_quadList, deviceParticle*
    d_deviceParticleList, float2* d_gradW)
{
    // Gather basic quad/particle data
    deviceQuad quad = d_quadList[blockIdx.x];
    int particleIdx = quad.firstContainedParticleIdx + threadIdx.x;
    if (threadIdx.x >= quad.containedParticleCount)
        return;

    // Place particle into shared mem cache, so other threads in this block can use it
    __shared__ deviceParticle containedParticles[MAX_PARTICLES_PER_BUCKET];
    deviceParticle thisParticle = d_deviceParticleList[particleIdx];
    containedParticles[threadIdx.x] = thisParticle;
    __syncthreads();

    // Declare registers
    int counter = 0;
    double mass = thisParticle.particleData[7];
    double h = thisParticle.particleData[8];
    double Z = 1, dZdh, p, dpdh;

    // Calculate density/acceleration at current smoothing length,
    // Adjust smoothing length using the Newton-Raphson method
    while ((fabs(Z) > THRESHOLD || h <= 0) && counter++ < 3)
    {
        // Reset SPH data
        p = 0;
        dpdh = 0;

        // Iterate through each neib bucket
        for (int i = 0; i < quad.neibBucketCount; i++)
        {
            deviceQuad neibQuad = d_quadList[quad.d_neibBucketsIndices[i]];
            bool sameQuad = neibQuad.firstContainedParticleIdx ==
                quad.firstContainedParticleIdx;
            // Iterate through each particle in neib bucket
            for (int j = 0; j < neibQuad.containedParticleCount; j++)
            {
                // Allows shared memory multicasts to occur
                __syncthreads();
                int neibIdx = neibQuad.firstContainedParticleIdx + j;
                // If the neib particle is within this quad, ensure shared mem cache is used
                deviceParticle neibParticle = sameQuad ? containedParticles[j] :
                    d_deviceParticleList[neibIdx];
                // Get i to j direction vector
                float neibMass = neibParticle.particleData[7];
                float rx = neibParticle.particleData[0] - thisParticle.particleData[0];
                float ry = neibParticle.particleData[1] - thisParticle.particleData[1];
                float r = sqrt(square(rx) + square(ry) + EPSILON);
                // Increment density and density derivative.
                dpdh += neibMass * dWdh(r, h);
                p += neibMass * W(r, h);
                // Find weight function gradient
                float q = r/h;
                float tempGradWx = rx * ((1.0f / quartic(h)) * (1.0f/q) * dwdq(q));
                float tempGradWy = ry * ((1.0f / quartic(h)) * (1.0f/q) * dwdq(q));
                d_gradW[particleIdx * N + neibIdx] = {tempGradWx, tempGradWy};
            }
        }

        // Newton-Raphson method calculations
        Z = mass * square(ETA / h) - p;
        dZdh = (-2.0f * mass * square(ETA) / cube(h)) - dpdh;
        h -= Z / dZdh;
    }

    // Assign SPH data
    d_deviceParticleList[particleIdx].particleData[8] = h;
    d_deviceParticleList[particleIdx].particleData[9] = p;
    d_deviceParticleList[particleIdx].particleData[10] = P(p);

    // Smoothing length of quad for the next iteration should = 2 * max smoothing length
    if (h > d_quadList[blockIdx.x].hCell)
        d_quadList[blockIdx.x].hCell = 2 * h;
}
```

The second iteration of SPH calculations also follows Winkler et al.'s pseudocode. It uses the previously calculated density and gradient data to calculate accelerations and integrate the velocities and positions of each particle.
Note the cache memory declaration:

```
__shared__ deviceParticle containedParticles[MAX_PARTICLES_PER_BUCKET];
```

The following inequality should be satisfied.

$$MAX\_PARTICLES\_PER\_BUCKET * sizeof(deviceParticle) \leq 49152 \text{ bytes}$$

$$MAX\_PARTICLES\_PER\_BUCKET * PARTICLE\_DATA\_LENGTH * sizeof(float) \leq 49152 \text{ bytes}$$

If there are too many particles per bucket, there will not be enough shared memory to cache all particles within a quad. Depending on context, one can find the maximum amount of particles per bucket that allows for a high amount of parallelism, but also a low amount of wasted particle interactions. For example, a neighbouring quad may contain 1000 particles, but only 9 of them could be within the smoothing length of particle i. This means that there will be 991 SPH calculations for particles that aren't even within the smoothing length of particle i. Note that branching, such as

```
if (particle j is within smoothing length)
   do calculation;
else
   continue;
```

seems like it could solve this problem, but in reality, it will deeply cut the performance of the GPU.

6 GPU $\to$ CPU

As shown near the end of this kernel, any global memory data modified by the GPU will be in d_deviceParticleList. This must be simply transferred back to the host:

```
cudaMemcpy(h_deviceParticleList, d_deviceParticleList, N*sizeof(deviceParticle),
    cudaMemcpyDeviceToHost);
cudaMemcpy(h_quadList, d_quadList, quadCount*sizeof(deviceQuad), cudaMemcpyDeviceToHost);
```

Any unneeded memory on the GPU can be freed using

```
cudaFree(variableName);
```

7 Notes

The remainder of the code for this prototype is either unimportant (such as the intricacies of the quadtree system) or trivial (such as the openGL code). Some noteworthy papers and sources that I have read are listed below, however it was not necessary for me to direclty cite them in-text.

SPH on GPU with CUDA[4]

Nearest Neighbour Searches on the GPU[5]

Don't be afraid of brute force on the GPU[6]

An Octree N-Body code that runs entirely on the GPU[7]

A talk on maximizing GPU memory efficiency. Note that most of these techniques are very specialized and will rarely see use outside of simple problems like matrix multiplication.[8]

[4]Hérault, Alexis. "SPH on GPU with CUDA." Journal of Hydraulic Research, vol. 48, no. extra, 2009, p. 000., doi:10.3826/jhr.2010.0005.

[5]Leite, Pedro, et al. "Nearest Neighbor Searches on the GPU." International Journal of Parallel Programming, vol. 40, no. 3, Dec. 2011, pp. 313–330., doi:10.1007/s10766-011-0184-3.

[6]Garcia, Vincent, et al. "Fast k Nearest Neighbor Search Using GPU." 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2008, doi:10.1109/cvprw.2008.4563100.

[7]Bédorf, Jeroen, et al. "A Sparse Octree Gravitational N-Body Code That Runs Entirely on the GPU Processor." Journal of Computational Physics, vol. 231, no. 7, 2012, pp. 2825–2839., doi:10.1016/j.jcp.2011.12.024.

[8]Google: Global Memory Usage and Strategy, GPU Computing Webinar 7/12/2011