Julia 2D

```
__global__ void kernel(byte* buffer, const int side, const float cx, const float
cy, const int iters)
{
    int offset = threadIdx.x + blockDim.x * blockIdx.x;
    if (offset >= side * side)
        return;
    int x = offset % side;
    int y = offset / side;

    // Compute point at this position
    int halfSide = side >> 1;
    float jx = 2.0f * (float)(x - halfSide) / halfSide;
    float jy = 2.0f * (float)(y - halfSide) / halfSide;
    cuComplex c(cx, cy);
    cuComplex z(jx, jy);

    // Iterating
    int i;
    for (i = 0; i < iters; ++i)
    {
        z = z * z + c;
        if (z.sqrMagnitude() > 4.0f)
            break;
    }
    float k = (float)i / iters;

    // Setting point color
    buffer[offset] = (byte)(k * 255);
}
```

Mandelbrot 2D

```
__global__ void kernel(byte* buffer, const int side, const int iters)
{
    int offset = threadIdx.x + blockDim.x * blockIdx.x;
    if (offset >= side * side)
        return;
    int x = offset % side;
    int y = offset / side;

    // Compute point at this position
    int halfSide = side >> 1;
    float jx = 2.0f * (float)(x - halfSide) / halfSide;
    float jy = 2.0f * (float)(y - halfSide) / halfSide;
    jx -= 0.5f;
    cuComplex c(jx, jy);
    cuComplex z(jx, jy);

    // Iterating
    int i;
```

```
    for (i = 0; i < iters; ++i)
    {
        z = z * z + c;
        if (z.sqrMagnitude() > 4.0f)
            break;
    }
    float k = (float)i / iters;

    // Setting point color
    buffer[offset] = (byte)(k * 255);
}
```

Complex

```
struct cuComplex
{
    float r, i;

    __device__ cuComplex(float a, float b) : r(a), i(b) {}

    __device__ float sqrMagnitude(void)
    {
        return r * r + i * i;
    }

    __device__ float magnitude(void)
    {
        return sqrtf(r * r + i * i);
    }

    __device__ cuComplex operator*(const cuComplex& a)
    {
        return { r * a.r - i * a.i, i * a.r + r * a.i };
    }

    __device__ cuComplex operator+(const cuComplex& a)
    {
        return { r + a.r, i + a.i };
    }
};
```

Mandelbulb 3D

```
__device__ int side1;
__device__ int side2;
__device__ int side3;

__global__ void initVars(const int side)
{
    side1 = side;
    side2 = side * side;
    side3 = side * side * side;
}

__global__ void kernel(
```

```
    byte* buffer,
    const float n,
    const int maxIter,
    const float bailout,
    const float sqrBailout,
    int* counterPoints)
{
    int offset = threadIdx.x + blockDim.x * blockIdx.x;
    if (offset >= side3)
        return;
    int z = offset / side2;
    offset -= z * side2;
    int y = offset / side1;
    int x = offset % side1;
    offset += z * side2;

    // Compute point at this position
    int halfSide = side1 >> 1;
    float fx = bailout * (float)(x - halfSide) / halfSide;
    float fy = bailout * (float)(y - halfSide) / halfSide;
    float fz = bailout * (float)(z - halfSide) / halfSide;
    Hypercomplex hc(fx, fy, fz);
    Hypercomplex hz(fx, fy, fz);

    // Iterating
    bool belongs;
    if (hc.sqrRadius() > sqrBailout)
        belongs = false;
    else
    {
        for (int i = 0; i < maxIter; ++i)
            hz = (hz ^ n) + hc;
        belongs = hz.sqrRadius() <= sqrBailout;
    }

    if (belongs)
    {
        buffer[offset] = (byte)(hc.sqrRadius() / sqrBailout * 255);
        atomicAdd(counterPoints, 1);
    }
    else
        buffer[offset] = 0;
}
```

Hypercomplex

```
struct Hypercomplex
{
    float x, y, z;

    __device__ Hypercomplex() : x(0.0f), y(0.0f), z(0.0f) {}

    __device__ Hypercomplex(float a, float b, float c) : x(a), y(b), z(c) {}

    __device__ float radius()
    {
```

```
        return sqrtf(x * x + y * y + z * z);
    }

    __device__ float sqrRadius()
    {
        return x * x + y * y + z * z;
    }

    __device__ float phi()
    {
        return atan2f(y, x);
    }

    __device__ float theta()
    {
        return atan2f(sqrtf(x * x + y * y), z);
    }

    __device__ Hypercomplex operator+(const Hypercomplex& a)
    {
        return Hypercomplex{ x + a.x, y + a.y, z + a.z };
    }

    __device__ Hypercomplex operator*(const float a)
    {
        return Hypercomplex{ x * a, y * a, z * a };
    }

    __device__ Hypercomplex operator^(const float n)
    {
        float rn = powf(radius(), n);
        float ph = phi();
        float th = theta();
        return Hypercomplex(
            sinf(n * th) * cosf(n * ph),
            sinf(n * th) * sinf(n * ph),
            cosf(n * th)
        ) * rn;
    }
};
```

Quaternion Julia 4D

```
__device__ int side1;
__device__ int side2;
__device__ int side3;

__global__ void initVars(const int side)
{
    side1 = side;
    side2 = side * side;
    side3 = side * side * side;
}

__global__ void kernel(
    byte* buffer,
```

```
    const float q1,
    const float q2,
    const float q3,
    const float q4,
    const int maxIter,
    const float bailout,
    const float sqrBailout,
    int* counterPoints)
{
    int offset = threadIdx.x + blockDim.x * blockIdx.x;
    if (offset >= side3)
        return;
    int z = offset / side2;
    offset -= z * side2;
    int y = offset / side1;
    int x = offset % side1;
    offset += z * side2;

    // Compute point at this position
    int halfSide = side1 >> 1;
    float fr = bailout * (float)(x - halfSide) / halfSide;
    float fa = bailout * (float)(y - halfSide) / halfSide;
    float fb = bailout * (float)(z - halfSide) / halfSide;
    float fc = q4;
    Quaternion qc(q1, q2, q3, q4);
    Quaternion qv(fr, fa, fb, fc);

    // Iterating
    bool belongs;
    if (qv.sqrRadius() > sqrBailout)
        belongs = false;
    else
    {
        for (int i = 0; i < maxIter; ++i)
            qv = qv.sqr() + qc;
        belongs = qv.sqrRadius() <= sqrBailout;
    }

    if (belongs)
    {
        buffer[offset] = (byte)((fr * fr + fa * fa + fb * fb) / (sqrBailout - fc
* fc) * 255);
        atomicAdd(counterPoints, 1);
    }
    else
        buffer[offset] = 0;
}
```

```
QFractal::QFractal(float r, float a, float b, float c, QFractal::ParamToHide h,
int maxIter)
{
    switch (h)
    {
    case QFractal::R:
        this->q1 = a;
        this->q2 = b;
        this->q3 = c;
```

```
        this->q4 = r;
        break;
    case QFractal::A:
        this->q1 = r;
        this->q2 = b;
        this->q3 = c;
        this->q4 = a;
        break;
    case QFractal::B:
        this->q1 = r;
        this->q2 = a;
        this->q3 = c;
        this->q4 = b;
        break;
    case QFractal::C:
        this->q1 = r;
        this->q2 = a;
        this->q3 = b;
        this->q4 = c;
        break;
    }
    this->maxIter = maxIter;
    this->bailout = 2.0f;
    this->sqrBailout = 4.0f;
}
```

Quaternion

```
struct Quaternion
{
    float r, a, b, c;

    __device__ Quaternion() : r(0.0f), a(0.0f), b(0.0f), c(0.0f) {}

    __device__ Quaternion(float _r, float _a, float _b, float _c) : r(_r),
a(_a), b(_b), c(_c) {}

    __device__ float radius()
    {
        return sqrtf(r * r + a * a + b * b + c * c);
    }

    __device__ float sqrRadius()
    {
        return r * r + a * a + b * b + c * c;
    }

    __device__ Quaternion operator+(const Quaternion& q)
    {
        return Quaternion{ r + q.r, a + q.a, b + q.b, c + q.c };
    }

    __device__ Quaternion sqr()
    {
        return Quaternion{
            r * r - a * a - b * b - c * c,
```

```
            2.0 * r * a,
            2.0 * r * b,
            2.0 * r * c
        };
    }
};
```