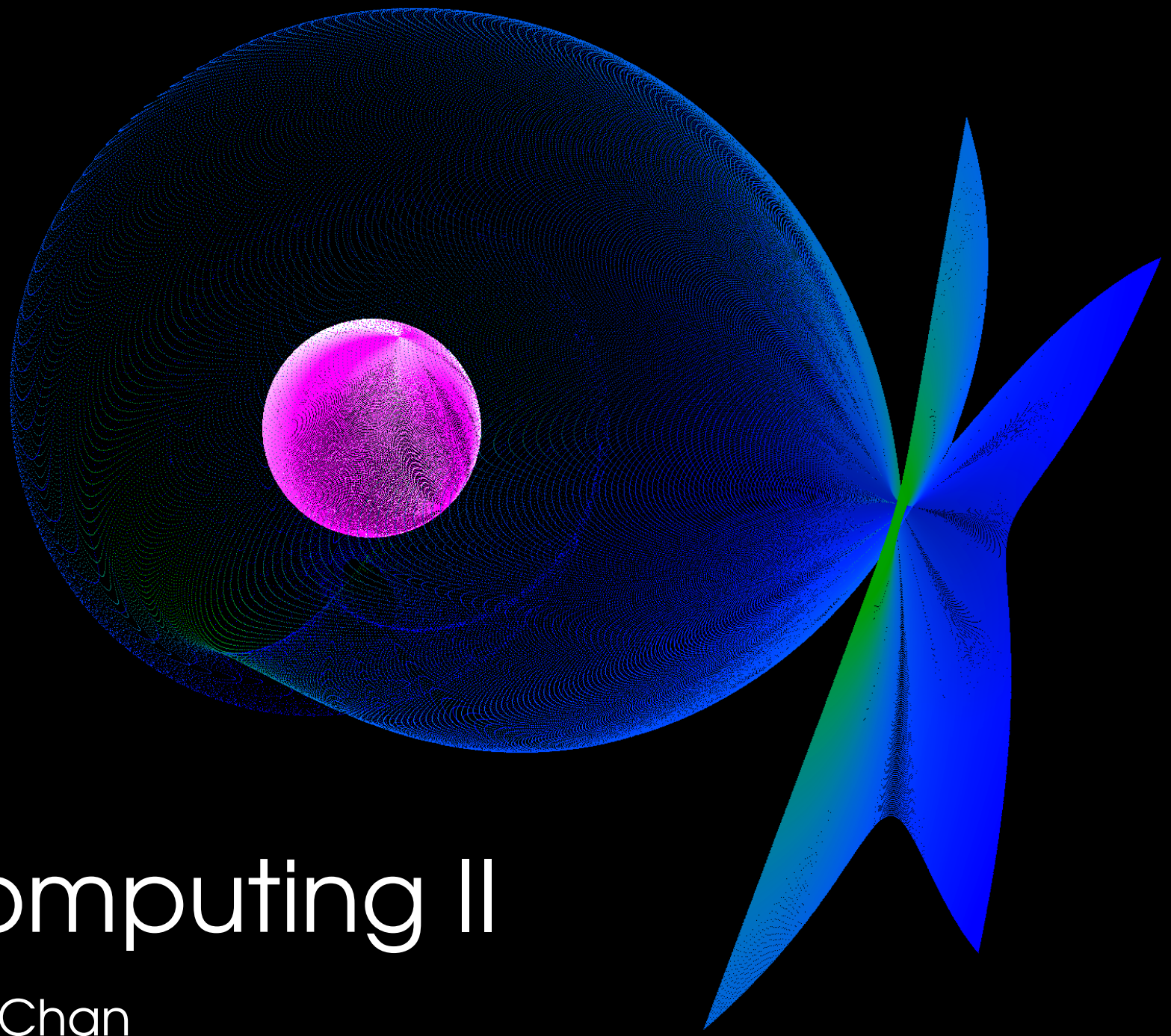
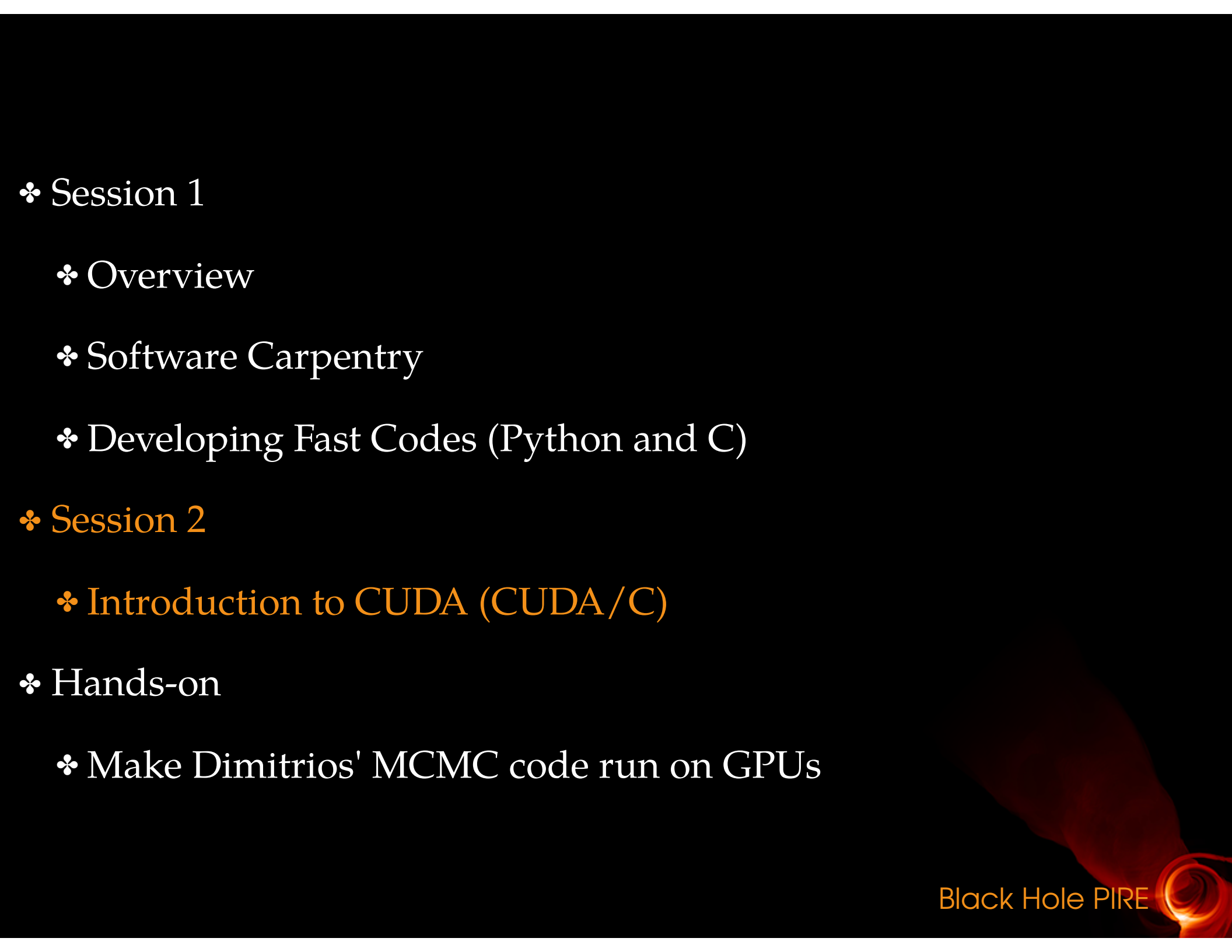


GPU Computing II

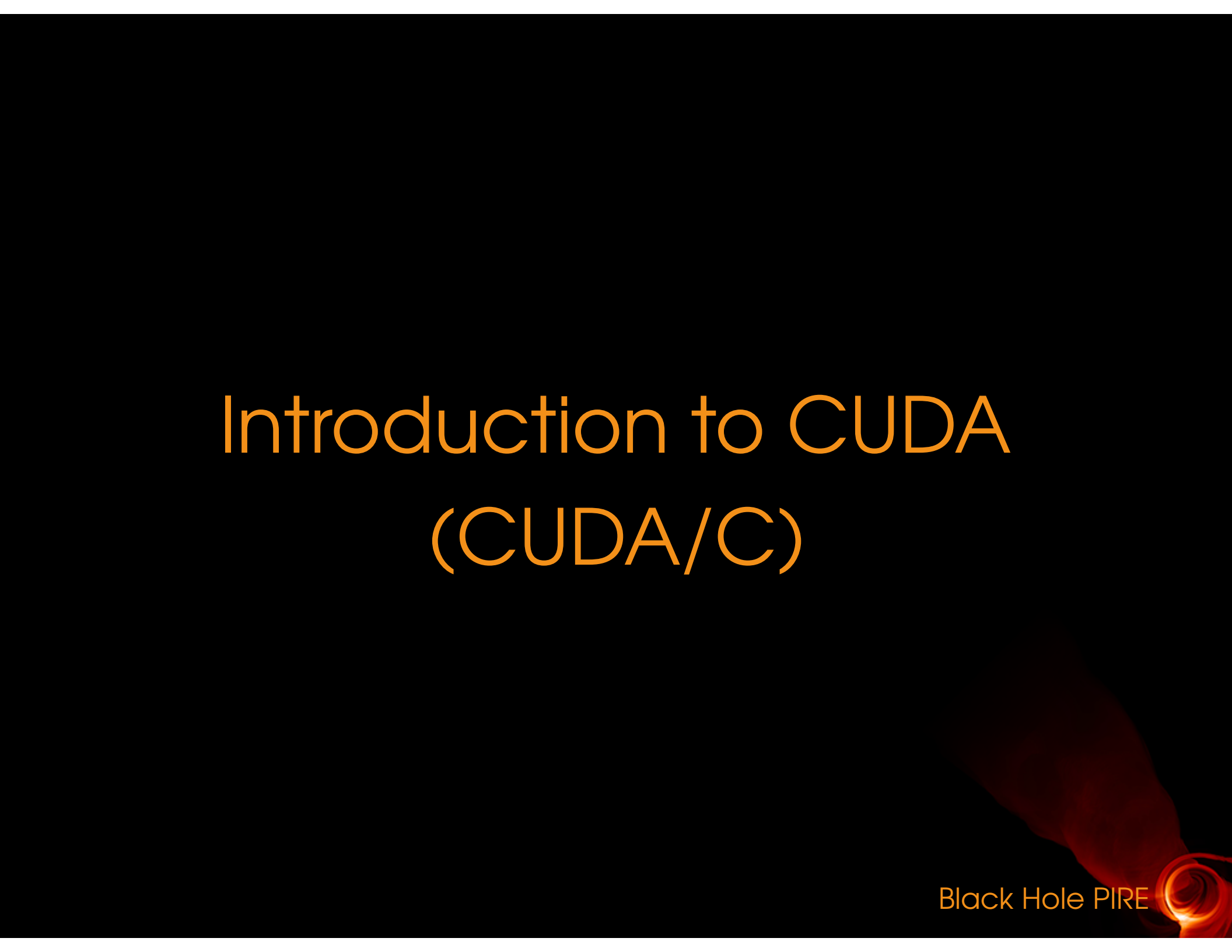
Chi-kwan "CK" Chan

Dec 5th, 2018, Biosphere2, PIRE Winter School

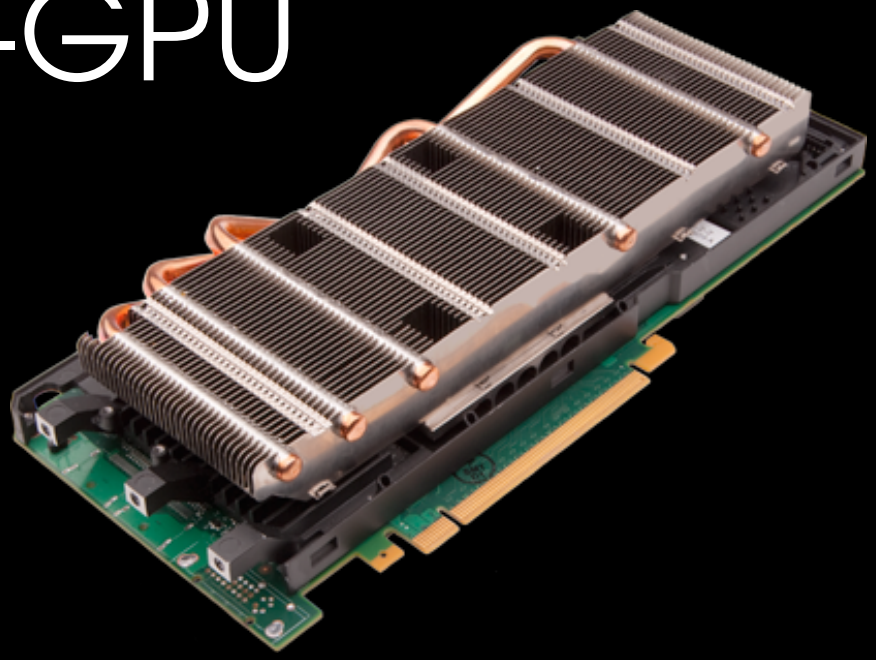
- Session 1
  - Overview
  - Software Carpentry
  - Developing Fast Codes (Python and C)
- Session 2
  - Introduction to CUDA (CUDA/C)
- Hands-on
  - Make Dimitrios' MCMC code run on GPUs



Introduction to CUDA (CUDA/C)

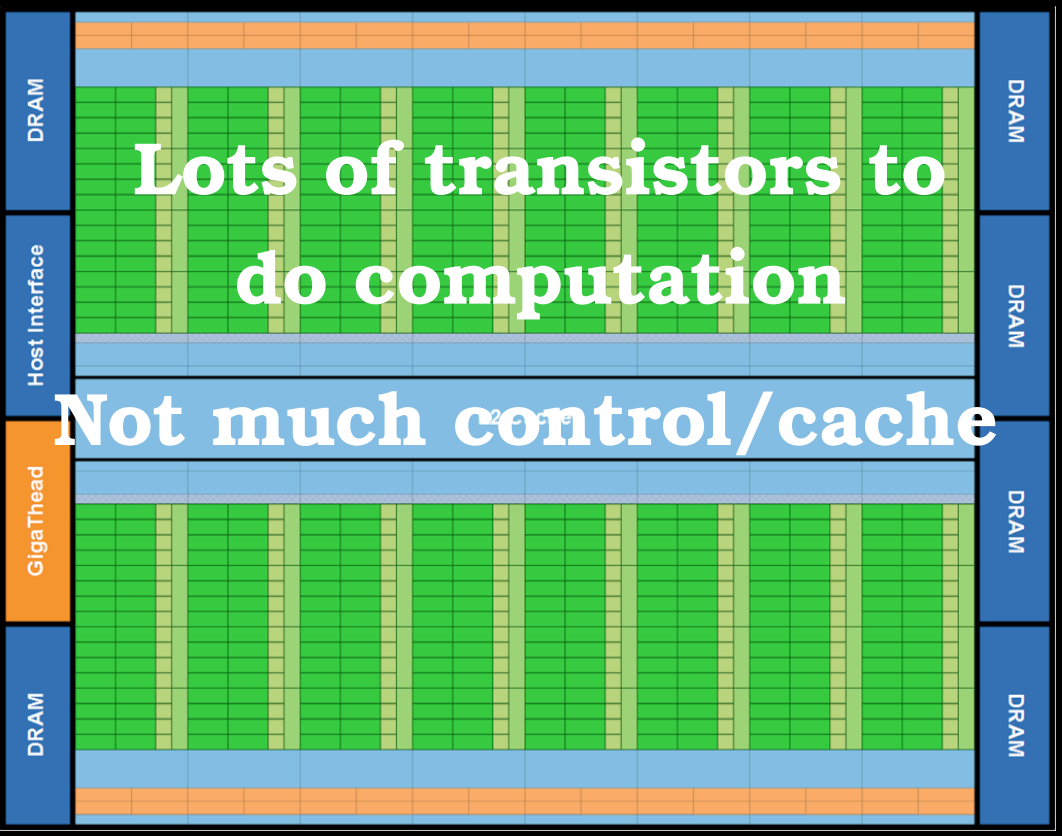


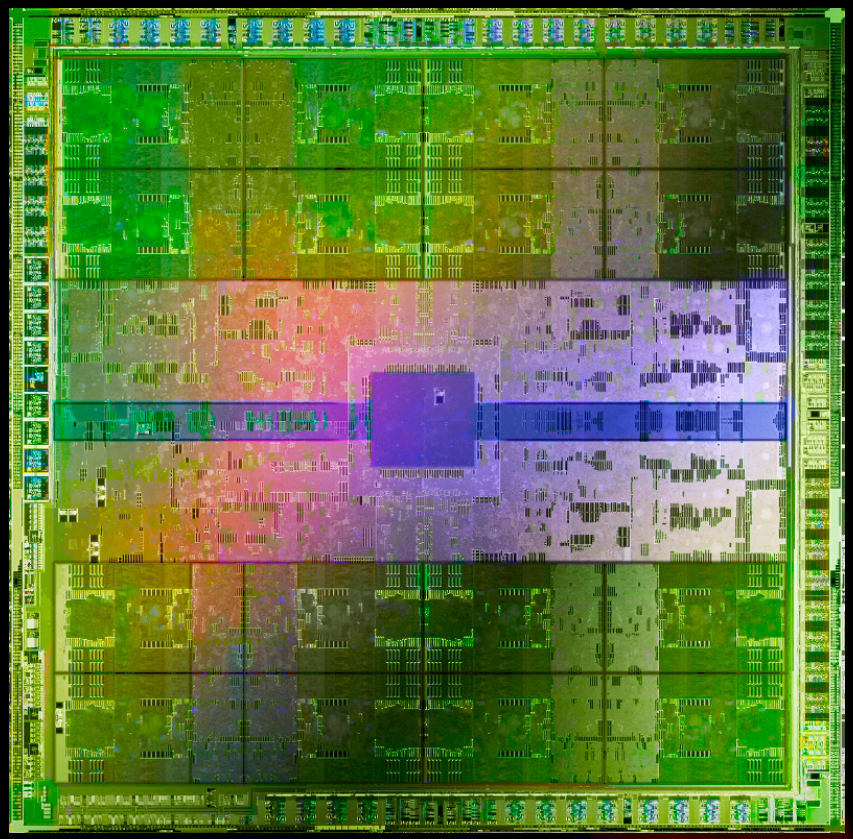
Introduction to GP-GPU



- General-Purpose computation on Graphics Processing Units
- Use massively parallel “stream” processors to accelerate computation
- Originally done by abusing/hacking the OpenGL API, later BrookGPU, Close To Metal, etc
- Currently done by CUDA C (nVidia and PGI) and OpenCL (open industrial standard)

Current Graphic Hardware





Introduction to CUDA

- Using Flynn's taxonomy: SIMD
- Data (stream) parallel
- Instead of writing for-loops such as

```
for(Z i = 0; i < n; ++i) job(i);
```

  we use the funny syntax

```
job<<<n>>>();
```

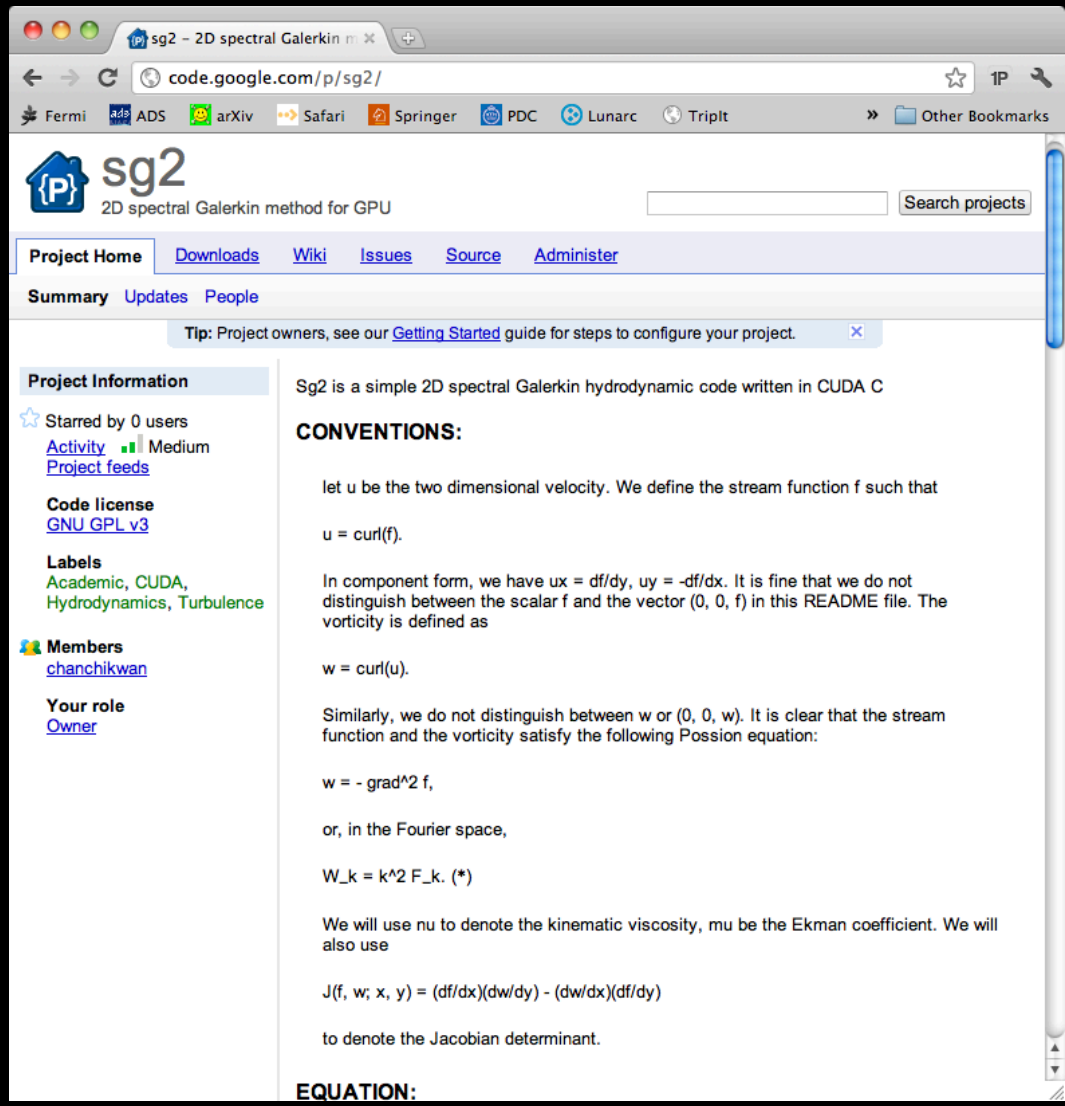
  to launch n jobs simultaneously, at least logically

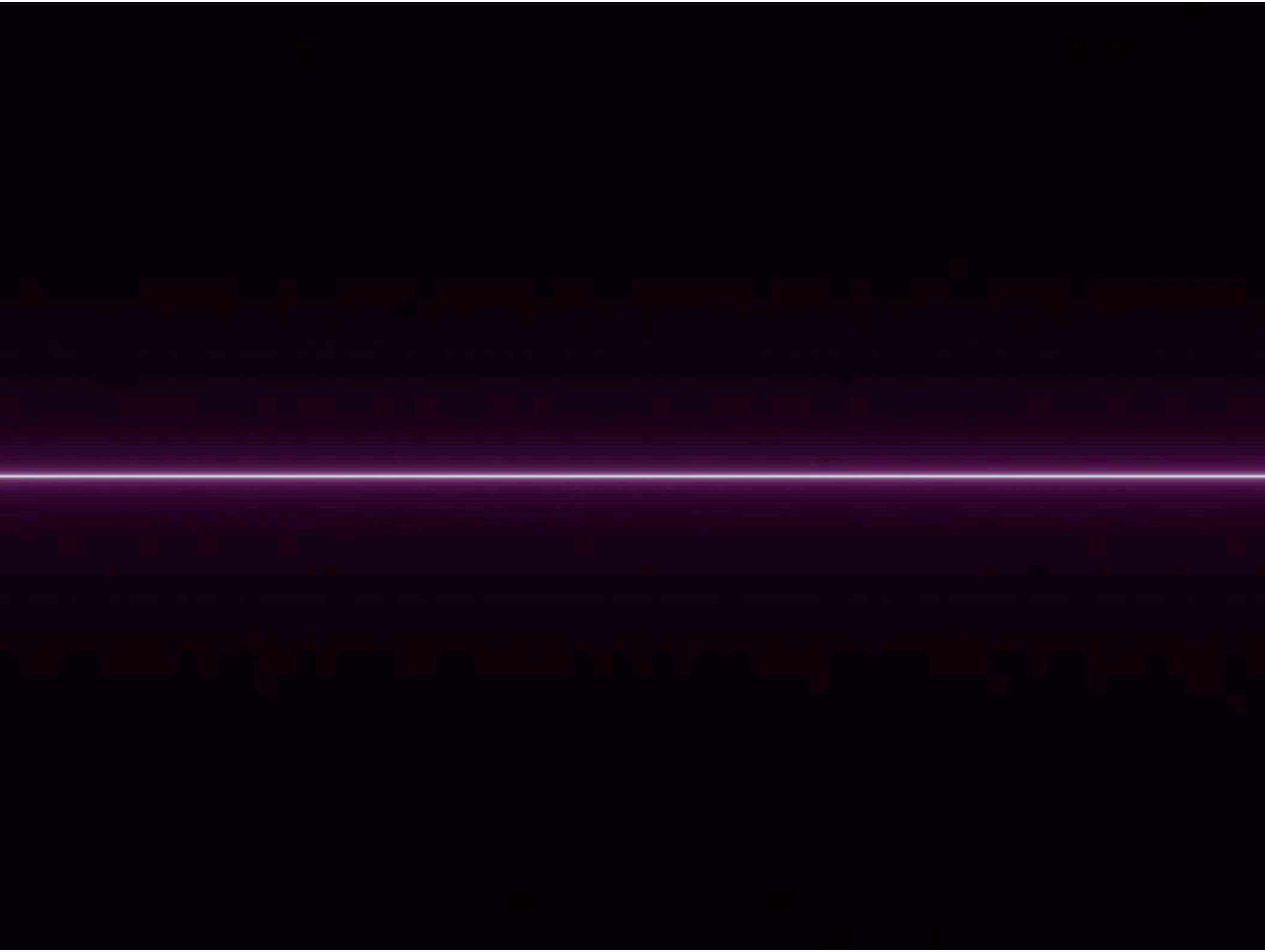
Example: Adding Vectors

```
__global__ void
kernel(R *c, const R *a, const R *b)
{
  int i = blockIdx.x * blockDim.x +
          threadIdx.x;
  c[i] = a[i] * b[i];
}

...
kernel<<<grid_sz, block_sz>>>(c, a, b);
```

Spectral Galerkin Method on GPUs

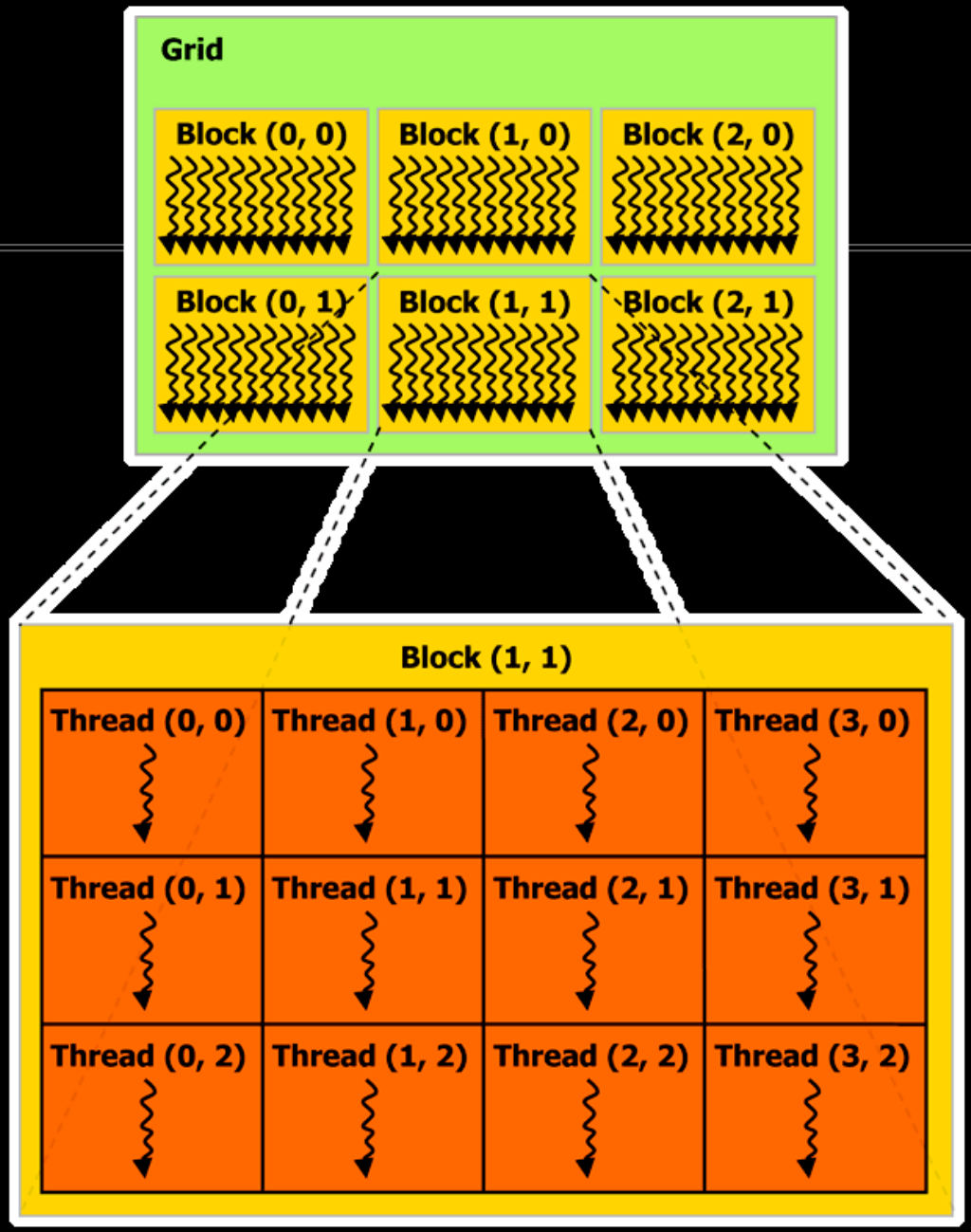
- Fourier spectral method with Galerkin truncation
- Implemented in CUDA C and runs on GPUs
- Use CUFFT library, home-made kernels are trivial
- Hosted on Google code: `sg2.googlecode.com`





Terminology

- Because of the hardware architecture…
- Threads (parallel jobs) are grouped into blocks
- Blocks are grouped into grids
- Shared memory is shared in blocks, but not in grids



Shared Memory Example: Transpose

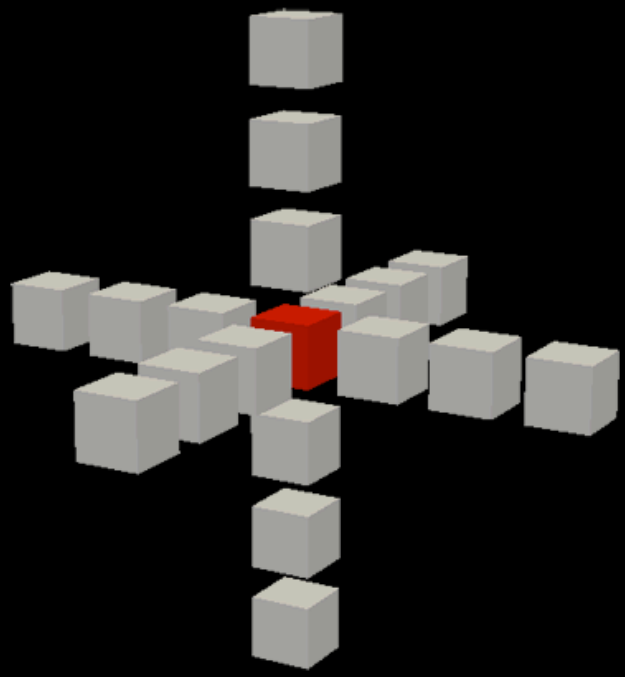
```
__global__ void
kernel(R *out, const R *in, Z n)
{

  int i = blockIdx.x * TILE + threadIdx.x;
  int j = blockIdx.y * TILE + threadIdx.y;




  out[j*n+i] = in[i*n+j];
}
```

Shared Memory Example: Transpose

```
__global__ void
kernel(R *out, const R *in, Z n)
{
  __shared__ R s[TILE][TILE];
  int i = blockIdx.x * TILE + threadIdx.x;
  int j = blockIdx.y * TILE + threadIdx.y;
  s[threadIdx.y][threadIdx.x] = in[j*n+i];
  __syncthreads();
  i = blockIdx.y * TILE + threadIdx.x;
  j = blockIdx.x * TILE + threadIdx.y;
  out[j*n+i] = s[threadIdx.x][threadIdx.y];
}
```

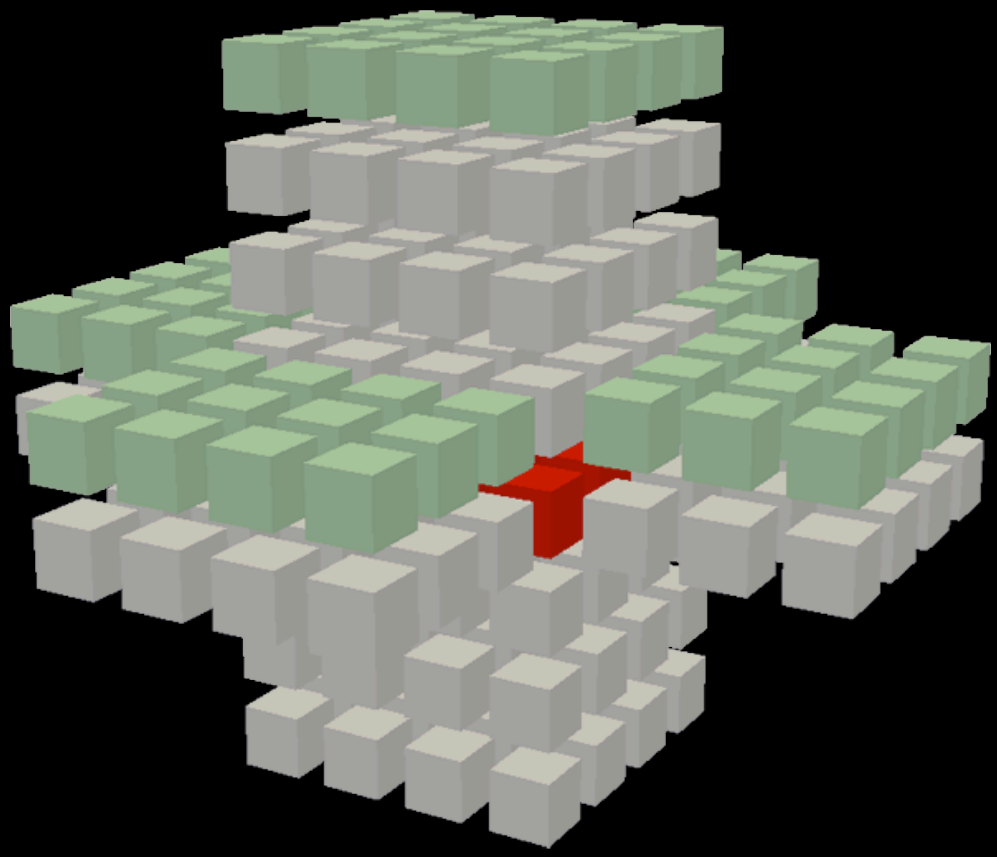
High-Order Finite Difference

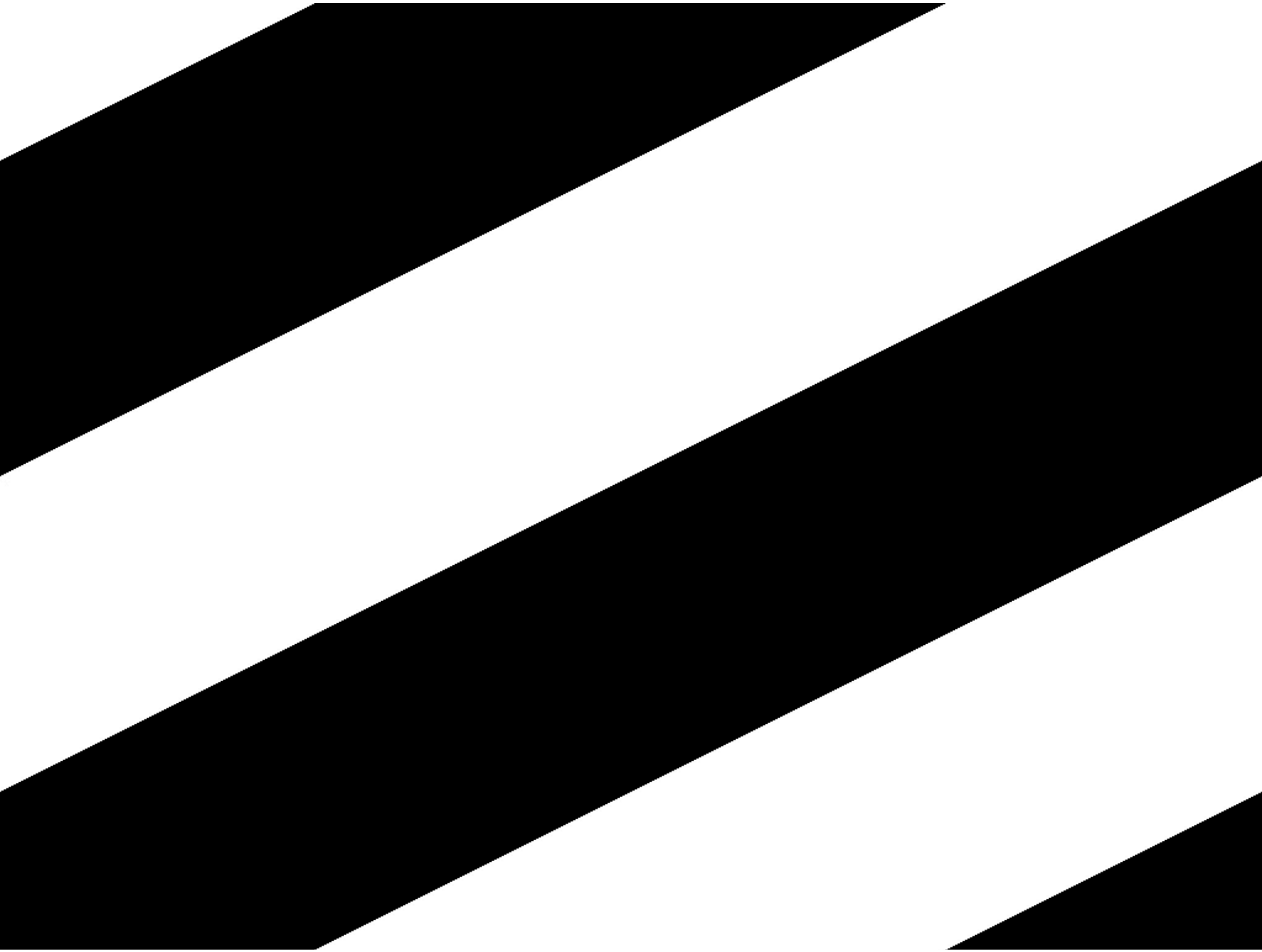
- 6th-order in space, 3rd-order in time (just like the `pencil` code)
- Simply “type” in equations, no Riemann solver or complications
- For derivatives along all directions (3 outputs), 19 input values, 30 FLOPS

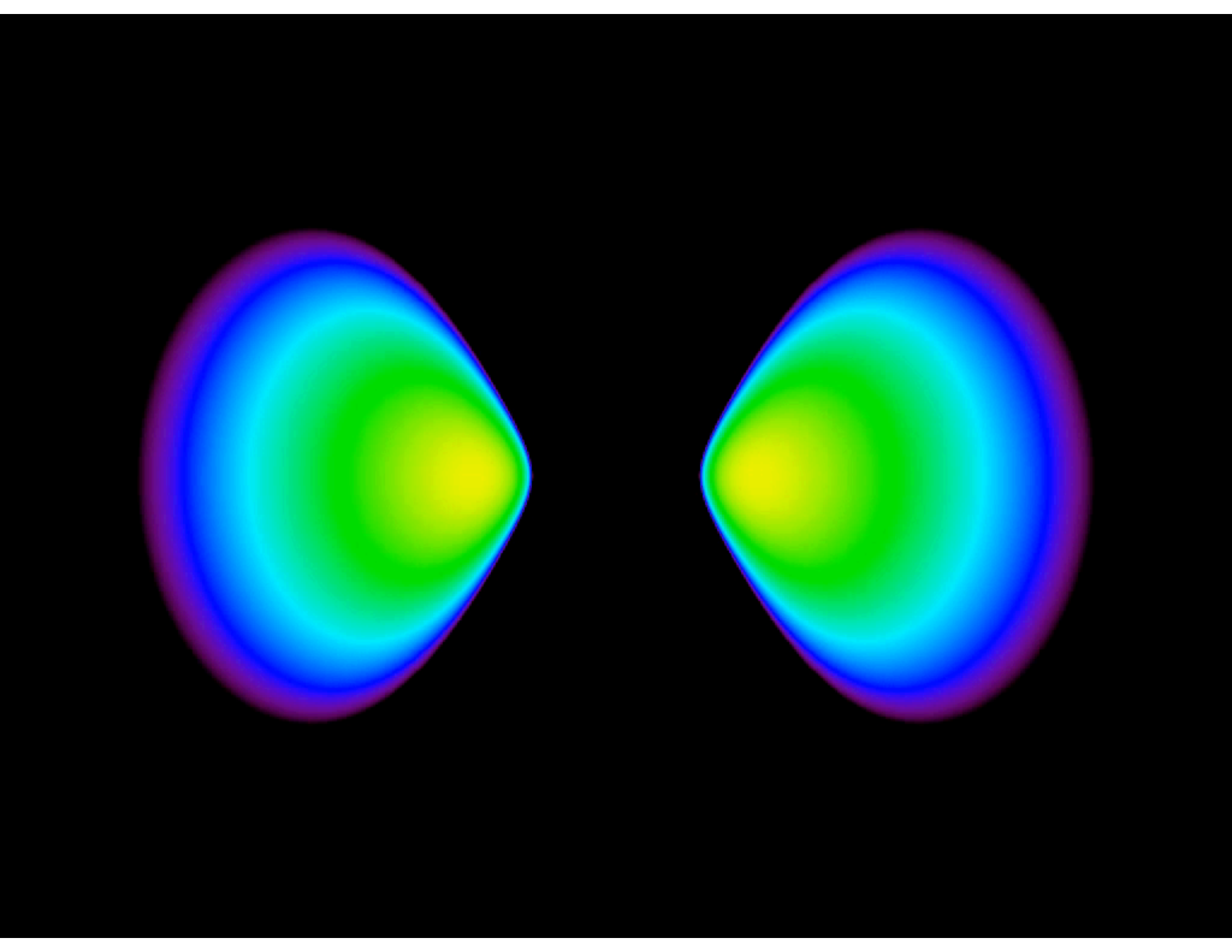


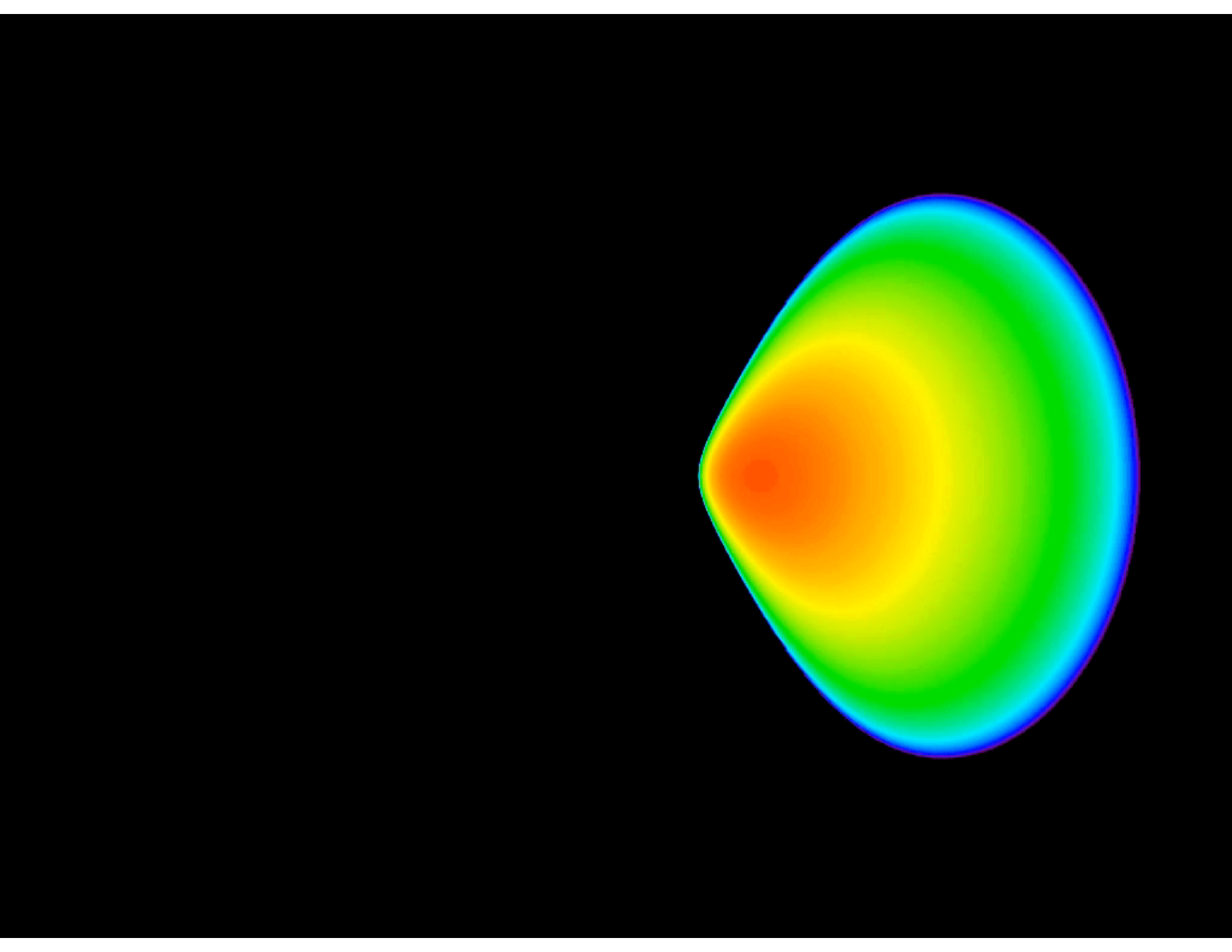
Rolling Cache

- At each time, compute a 2D tide (red cubes)
- To compute the next slide, we “roll” the kernel down along the grid
- For large tide, we load one cell per thread (purple)
- Forget the green cubes









Summary

- Jim Stone in 2007, "... my worry is that GPU will just go away like GRAPE and vector machines ..."
- After 10 years, GPUs seem to survive well, probably because of gaming market
- Tricks like the rolling cache method were proposed for different types of problem
- Still painful to program, but not worse than MPI
- Pick the right problems so GPU can solve them well