NVIDIA CUDA

Architecture and Programming Model

John Rugis

New Zealand eScience Infrastructure
University of Auckland, New Zealand

j.rugis@auckland.ac.nz

July 2, 2014

Motivation

Why use GPU's?

Motivation

Why use GPU's?

Speedup!

Motivation

Why use GPU's?

Speedup!

2x, 10x, 100x

Motivation

Why use GPU's?

Speedup!

2x, 10x, 100x

- Applications that use GPU's (i.e. MatLab, Gromacs)

Motivation

Why use GPU's?

Speedup!

2x, 10x, 100x

- Applications that use GPU's (i.e. MatLab, Gromacs)
- Graphics engines (i.e. SceniX, OptiX)

Motivation

Why use GPU's?

Speedup!

2x, 10x, 100x

- Applications that use GPU's (i.e. MatLab, Gromacs)
- Graphics engines (i.e. SceniX, OptiX)
- Libraries implemented on GPU's (i.e. cuBLAS, cuFFT)

Motivation

Why use GPU's?

Speedup!

2x, 10x, 100x

- Applications that use GPU's (i.e. MatLab, Gromacs)
- Graphics engines (i.e. SceniX, OptiX)
- Libraries implemented on GPU's (i.e. cuBLAS, cuFFT)
- Compilers that off-load to GPU's (i.e. PGI OpenACC)

Motivation

Why use GPU's?

Speedup!

2x, 10x, 100x

- Applications that use GPU's (i.e. MatLab, Gromacs)
- Graphics engines (i.e. SceniX, OptiX)
- Libraries implemented on GPU's (i.e. cuBLAS, cuFFT)
- Compilers that off-load to GPU's (i.e. PGI OpenACC)
- Device code (i.e. OpenCL, CUDA)

GPU Programming

NVIDIA CUDA

C++ object oriented programming

GPU Programming

NVIDIA CUDA

C++ object oriented programming

Scientific applications

- numerical
- combinatoric
- simulation

GPU Programming

NVIDIA CUDA

C++ object oriented programming

Scientific applications

- numerical
- combinatoric
- simulation

Target: UoA Pan cluster

- interactive development
- batch submission

Software tool chain

ssh to Pan cluster GPU build node:

- `make`
- `gcc`
- `nvcc`

Software tool chain

ssh to Pan cluster GPU build node:

- `make`
- `gcc`
- `nvcc`

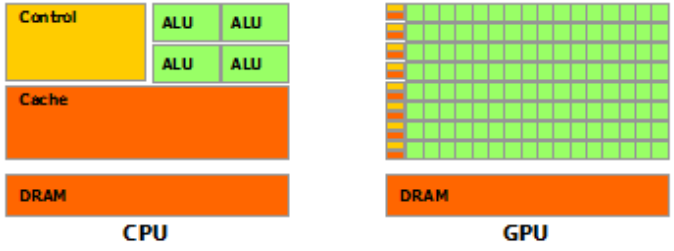
```
ssh upi@login-01.uoa.nesi.org.nz
ssh build2
module load gcc/4.7.3
module load CUDA/6.0
```

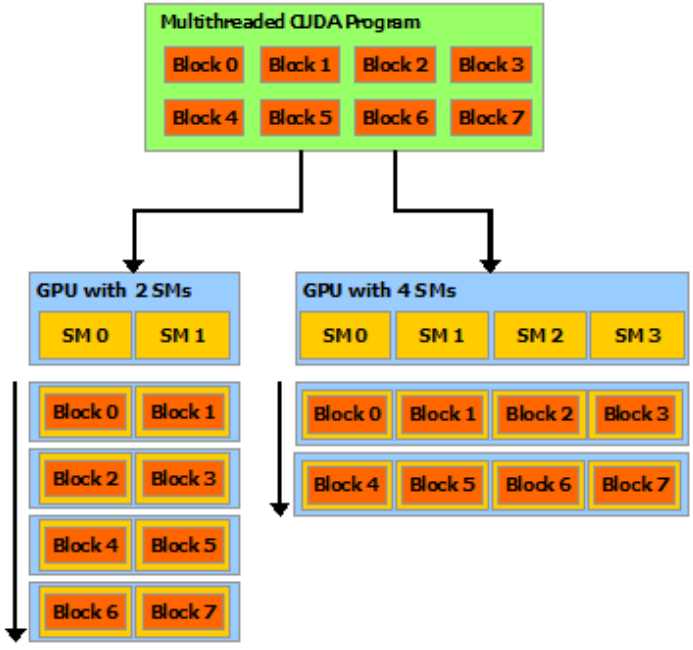
Code C0001: CUDA device query

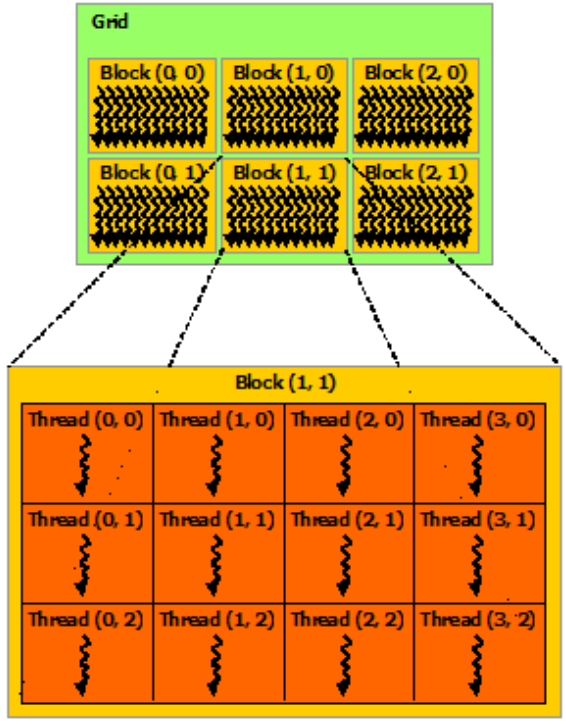
file: C0001.cpp

```
int main(int argc, char* argv[])
{
  int driver, runtime, deviceCount;
  cudaError_t error_id;

  error_id = cudaDeviceReset();
  if (error_id != cudaSuccess) {
    cout << "cudaDeviceReset returned " << error_id << ": "
         << cudaGetErrorString(error_id) << endl;
    exit(1);
  }
  cudaDriverGetVersion(&driver);
  cudaRuntimeGetVersion(&runtime);
  cudaGetDeviceCount(&deviceCount);
  cout << "              CUDA driver: " << driver/1000 << "." << driver%100 << endl;
  cout << "           CUDA runtime: " << runtime/1000 << "." << runtime%100 << endl;
  cout << "             GPU devices: " << deviceCount << endl << endl;

  return 0;
}
```

NVIDIA GPU architecture diagrams

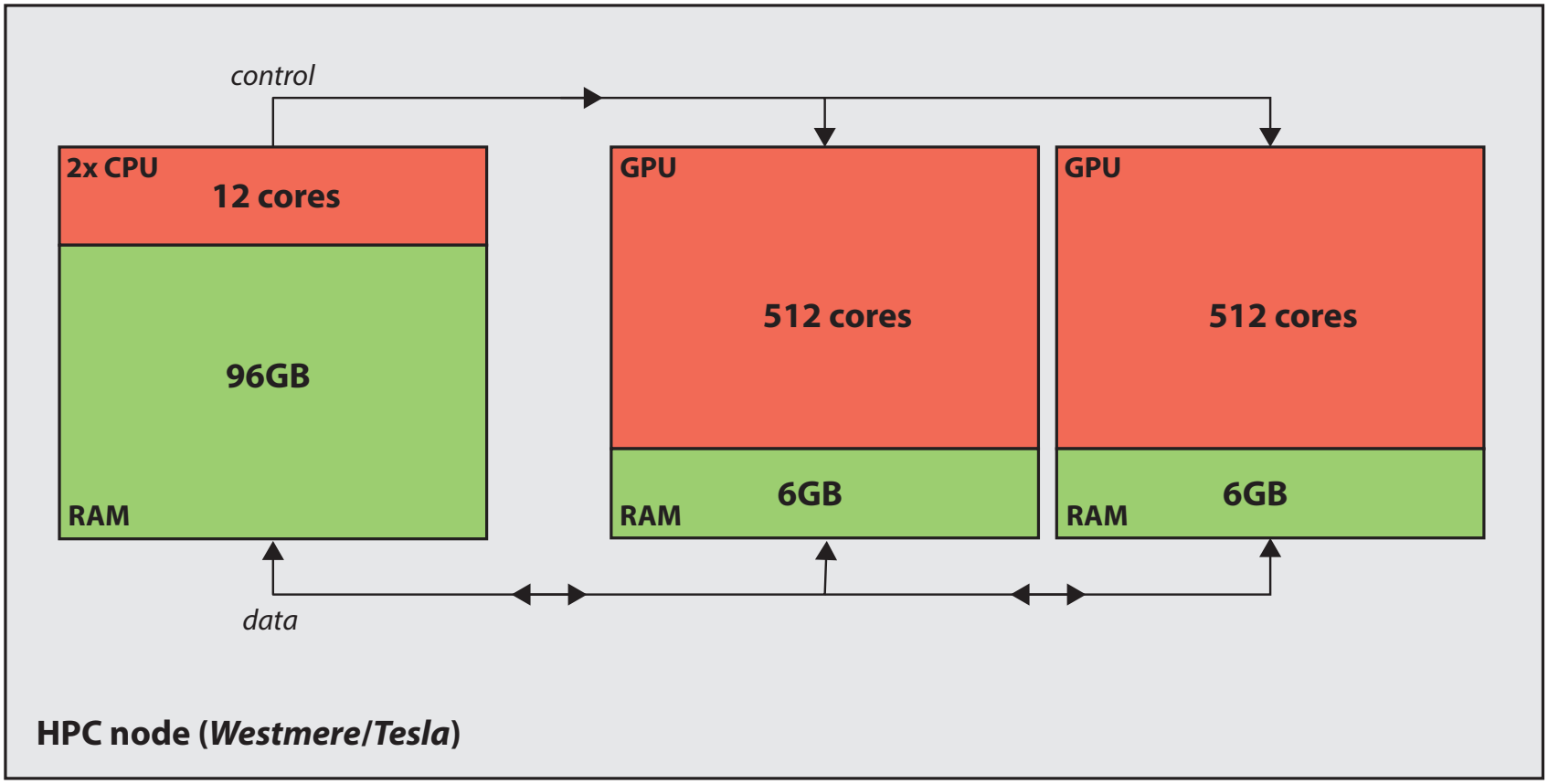






A largely hardware-centric view.

CPU / GPU node architecture



A programming-centric view.

Using GPU devices

GPU programming outline:

- allocate host and device memory
- copy data to device
- calculate
- copy data from device
- de-allocate host and device memory

Using GPU devices

GPU programming outline:

- allocate host and device memory
- copy data to device
- calculate
- copy data from device
- de-allocate host and device memory

Performance is constrained by the speed of data copying!

Code C0006: Vector add

file: C0006.cpp

```
// setup calculation(s)
float *A, *B, *C1, *C2;
C1 = new float[n];
cudaHostAlloc(&A, n * sizeof(float), cudaHostAllocDefault);
cudaHostAlloc(&B, n * sizeof(float), cudaHostAllocDefault);
cudaHostAlloc(&C2, n * sizeof(float), cudaHostAllocDefault);

srand (static_cast<unsigned>(time(0))); // random floats in range 0.0 to 1.0
for(size_t i = 0; i < n; i++) {
  A[i] = static_cast <float> (rand()) / static_cast <float> (RAND_MAX);
  B[i] = static_cast <float> (rand()) / static_cast <float> (RAND_MAX);
}

// perform calculation(s)
cout << endl << "Starting calculations..." << endl;
CVecAdd *vecAdd;
CGpuVecAdd *gvecAdd;
vecAdd = new CVecAdd();
gvecAdd = new CGpuVecAdd(0, t); // use CUDA device 0
```

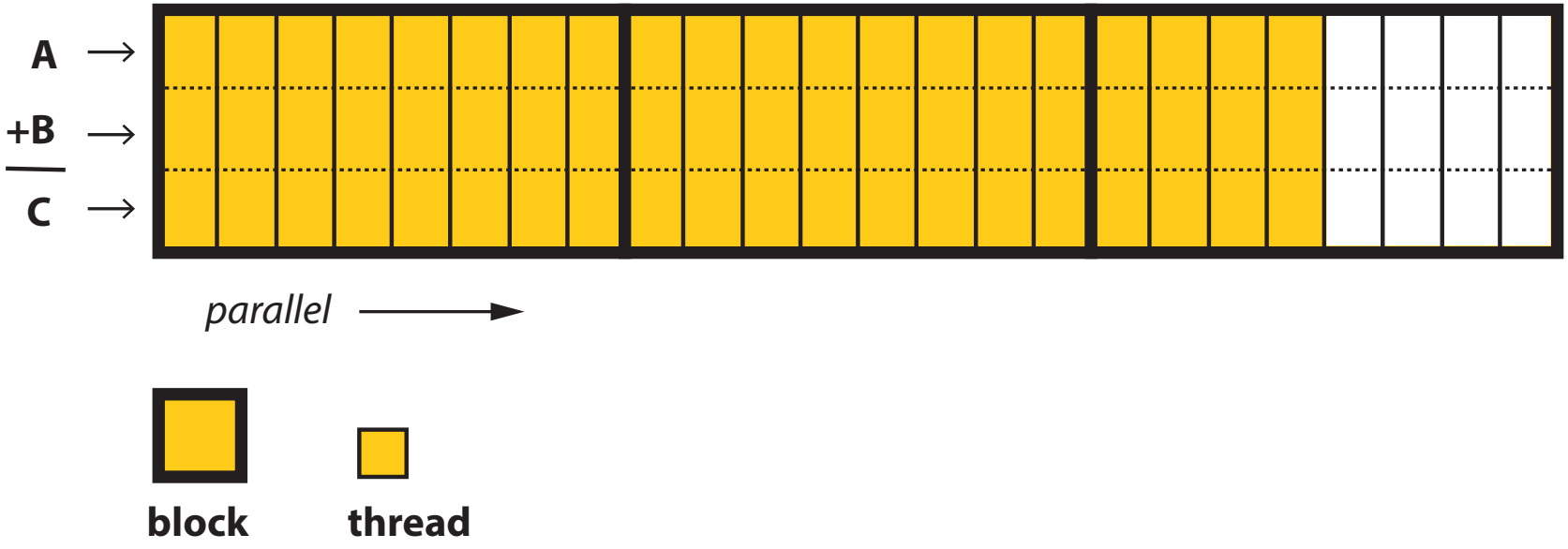
Code C0006: Vector add

file: gpuvecadd.cu

```
// allocate device memory
float *d_A, *d_B, *d_C;
cudaCheck(cudaMalloc(&d_A, size), "cudaMalloc");
cudaCheck(cudaMalloc(&d_B, size), "cudaMalloc");
cudaCheck(cudaMalloc(&d_C, size), "cudaMalloc");

// copy from host memory to device memory
cudaMemcpy(d_A, A, size, cudaMemcpyDefault);
cudaMemcpy(d_B, B, size, cudaMemcpyDefault);

// invoke kernel
int blocks = (N + threads - 1) / threads;
kVecAdd <<<blocks, threads>>>(d_A, d_B, d_C, N);
cudaCheck(cudaGetLastError(), "kVecAdd");

// copy result from device memory to host memory
cudaMemcpy(C, d_C, size, cudaMemcpyDefault);

// free device memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Parallelization count: *block(s)* each containing *thread(s)*

Code C0006: Computation blocks



Vector add

All threads run in parallel.

Code C0006: GPU vector add kernel

file: gpuvecadd.cu

```
////////////////////////////////////////////////////////////////////////////////
// CUDA device kernel(s)
////////////////////////////////////////////////////////////////////////////////
__global__ void kVecAdd(const float *A, const float *B, float *C, size_t N)
{
  size_t i = blockDim.x * blockIdx.x + threadIdx.x;
  if(i < N) {
    C[i] = A[i]+ B[i];
  }
}
```

CUDA programming

It's ok, even desirable, to oversubscribe GPU usage.

The GPU's internal scheduler queues kernels and does its best to maximize *occupancy*.

Code C0200: N-body simulation

An n-body problem is one in which every simulated object interacts with every other simulated object.

Code C0200: N-body simulation

An n-body problem is one in which every simulated object interacts with every other simulated object.

In general, n-body problems cannot be solved analytically.

Code C0200: N-body simulation

An n-body problem is one in which every simulated object interacts with every other simulated object.

In general, n-body problems cannot be solved analytically.

An astro-physics example:

Multiple solar system bodies interacting with each other through mutual gravitational attraction.

Code C0200: N-body simulation

An n-body problem is one in which every simulated object interacts with every other simulated object.

In general, n-body problems cannot be solved analytically.

An astro-physics example:

Multiple solar system bodies interacting with each other through mutual gravitational attraction.

1) Start with *governing physics equations*.

Code C0200: N-body simulation

An n-body problem is one in which every simulated object interacts with every other simulated object.

In general, n-body problems cannot be solved analytically.

An astro-physics example:

Multiple solar system bodies interacting with each other through mutual gravitational attraction.

1) Start with *governing physics equations*.

2) *Discretize* the governing equations for computer implementation.

Code C0200: N-body simulation

Governing equations:

$$\mathbf{f}_{ij} = \left(G \frac{m_i m_j}{||\mathbf{r}_{ij}||^2} \right) \left(\frac{\mathbf{r}_{ij}}{||\mathbf{r}_{ij}||} \right) \tag{1}$$

$$\mathbf{F}_i = \sum_{1 \leq j \leq N} \mathbf{f}_{ij} \quad , \; j \neq i \tag{2}$$

$$\mathbf{F}_i = m_i \mathbf{a}_i \tag{3}$$

$$\mathbf{p}_i = \iint \mathbf{a}_i \, \mathrm{d}t \tag{4}$$

Code C0200: N-body class

file: nbody.h

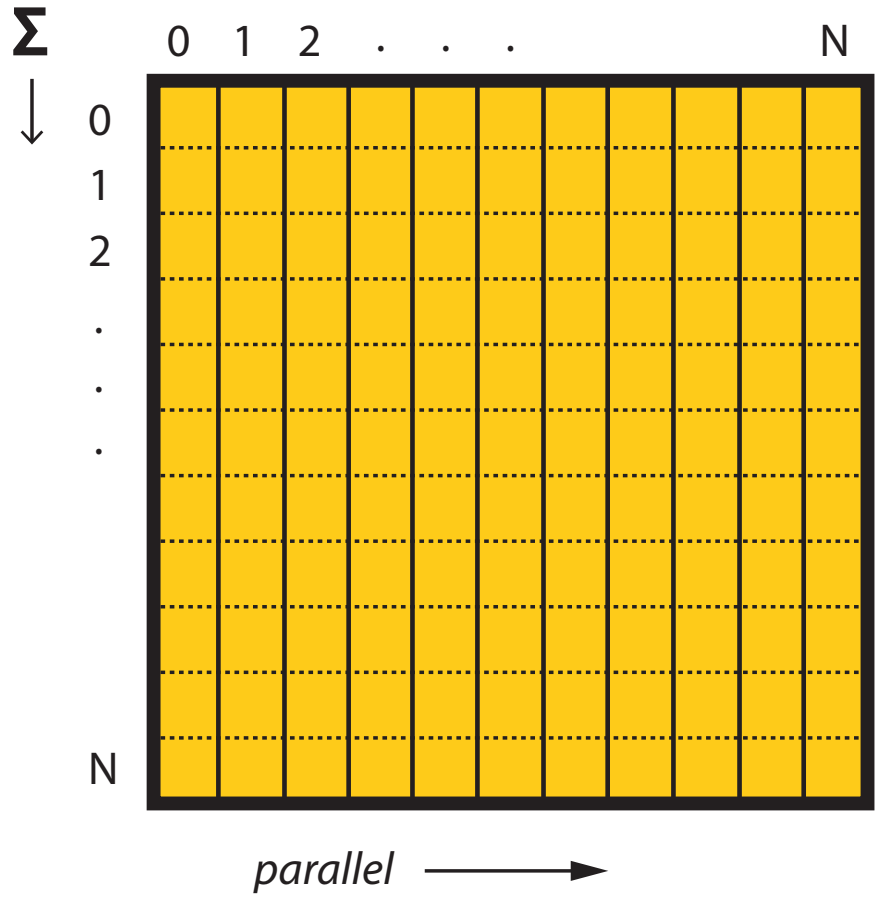
```
// n-body elements
enum NBodyElementType
{
  MG,           // mass (kg) x gravitational constant
  PX, PY, PZ,     // position (m)
  VX, VY, VZ,     // velocity (m/s)
  AX, AY, AZ,     // acceleration (m/s^2)
  NUM_ELEMENTS
};
```

Parallel Computing

Model *decomposition* for parallel computing:

Design an implementation that makes use of multiple independent, preferably duplicate, computation blocks.

Code C0200: Computation blocks



N-body

Code C0200: CUDA kernel

file: nbody_device.cuh

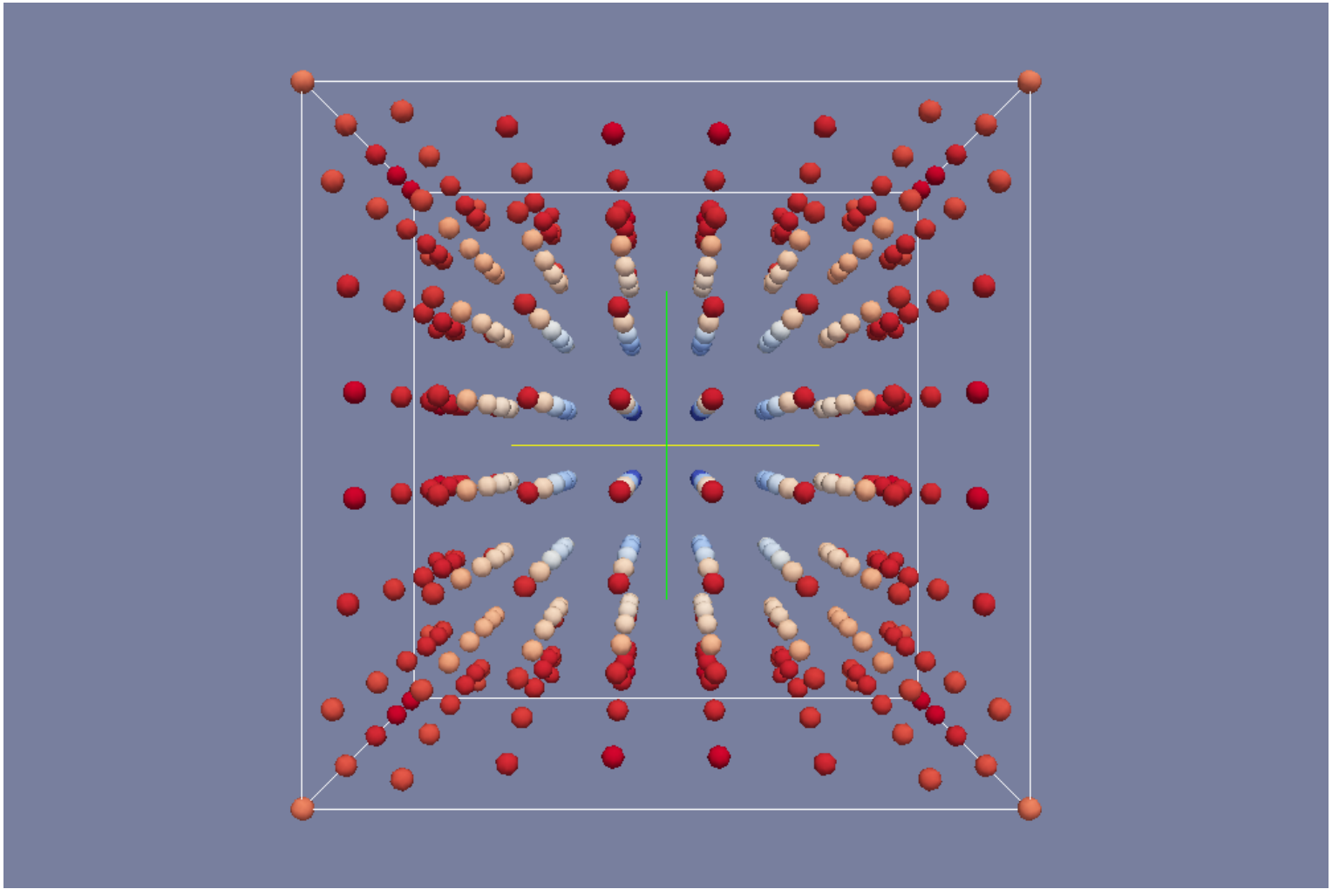
```
__global__ void update_acceleration_kernel(
  F_TYPE *axG, F_TYPE *ayG, F_TYPE *azG,
  const size_t number, const F_TYPE *mgG,
  const F_TYPE *pxG, const F_TYPE *pyG, const F_TYPE *pzG)
{
  const size_t n = blockIdx.x * blockDim.x + threadIdx.x;
  F_TYPE rx, ry, rz, s;
  F_TYPE distSqr, distSixth;

  for(size_t i = 0; i < number; i++) {
    rx = pxG[i] - pxG[n];
    ry = pyG[i] - pyG[n];
    rz = pzG[i] - pzG[n];
    distSqr = rx * rx + ry * ry + rz * rz + EPS2;
    distSixth = distSqr * distSqr * distSqr;
    s = mgG[i] / sqrt(distSixth);
    // total acceleration = old acceleration + new acceleration
    axG[n] += rx * s;
    ayG[n] += ry * s;
    azG[n] += rz * s;
  }
}
```

CUDA programming

If the device memory size is large enough to hold all of the data associated with a simulation, *then updated data does not necessarily need to be copied out to the host after every time step*.

Code C0200: Visualization of output data

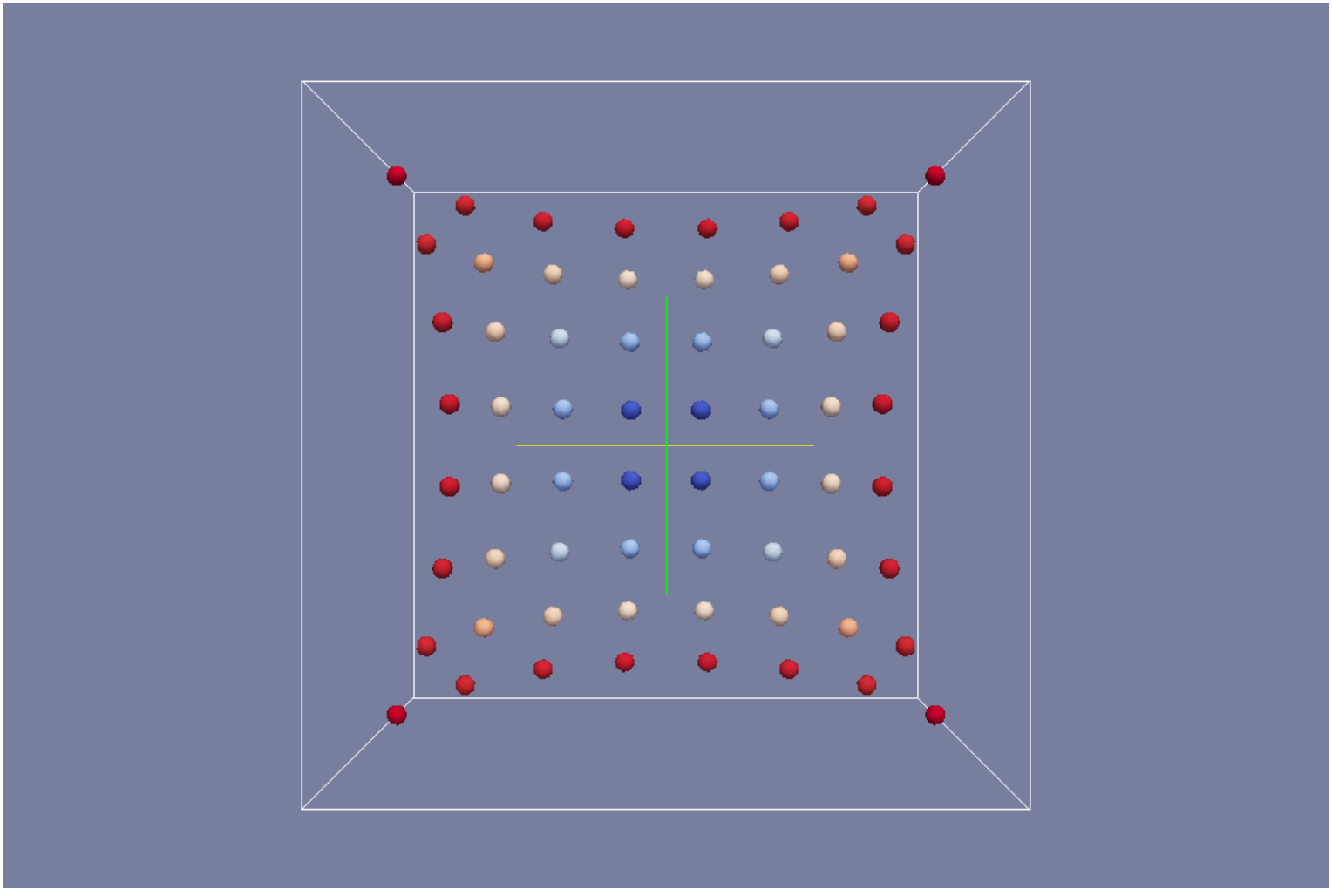
Output VTK format for visualisation in ParaView.



Results after 150 time steps.

Code C0200: Visualization of output data

Single layer data slice.



Results after 150 time steps.

Code C0100: Electromagnetic field simulation

This is another example of a time stepping simulation.

An electromagnetic field is a combination of two vector fields: one electric (symbol **E**) and one magnetic (symbol **H**). Electromagnetic fields propagate in free-space at the speed of light.

The electromagnetic field equations consist of a pair of coupled space and time varying wave equations.

Code C0100: Electromagnetic field simulation

Governing equations:

$$\varepsilon \frac{\partial \mathbf{E}}{\partial t} = (\nabla \times \mathbf{H}) - \sigma \mathbf{E} \tag{1}$$

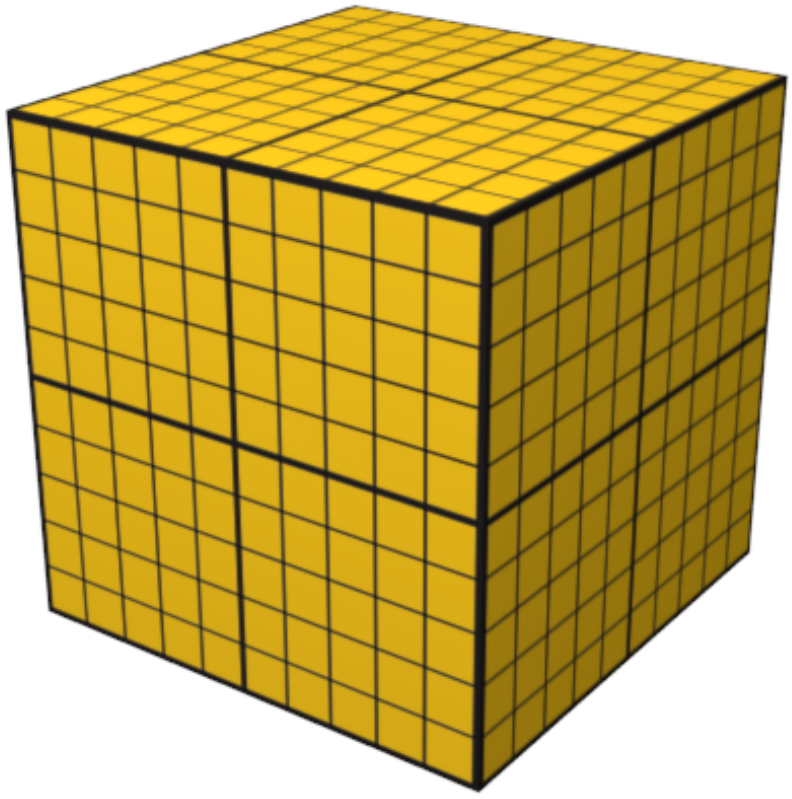
$$\mu \frac{\partial \mathbf{H}}{\partial t} = -(\nabla \times \mathbf{E}) - \sigma_m \mathbf{H} \tag{2}$$

Code C0100: EH space class

file: spaceEH3d.h

```
// Yee-cell elements
enum YeeElementType
{
  EX, EY, EZ, HX, HY, HZ,                 // field values
  CEXE, CEYE, CEZE, CHXH, CHYH, CHZH, // material values
  CHXE, CHYE, CHZE, CEXH, CEYH, CEZH
};
```

Code C0100: Computation blocks

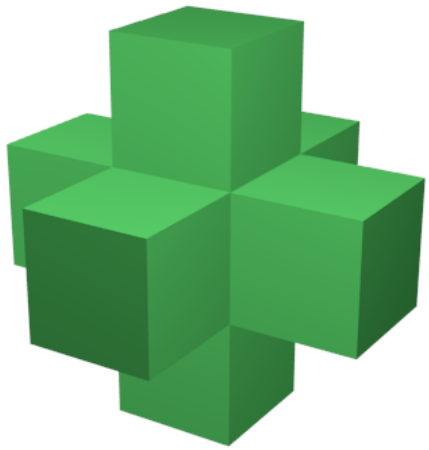


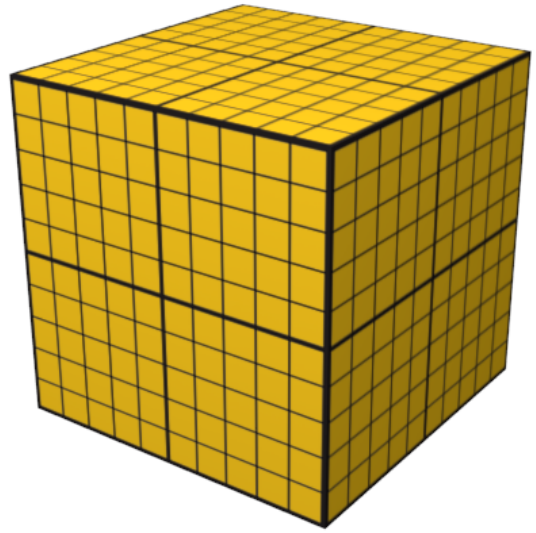
3D EH-space

block

thread

Code C0100: Computation blocks





3D EH-space

block thread

Each thread kernel needs values from six adjacent cells.

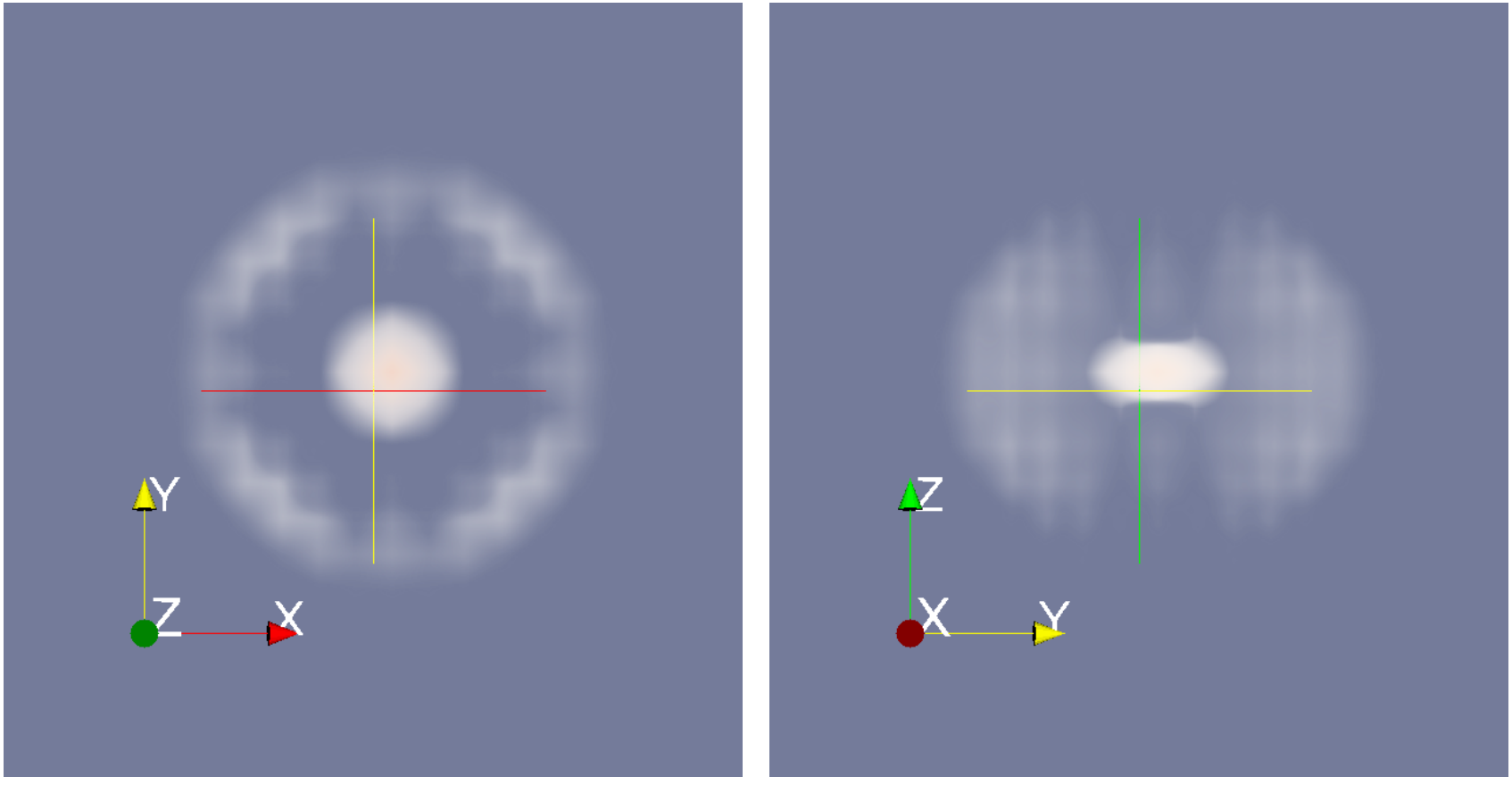
Code C0100: CUDA kernel

file: spaceEH3d_device.cuh

```
__global__ void update_e_Kernel(
  F_TYPE *exG, F_TYPE *eyG, F_TYPE *ezG,
  const F_TYPE *hxG, const F_TYPE *hyG, const F_TYPE *hzG,
  const F_TYPE *cexeG, const F_TYPE *ceyeG, const F_TYPE *cezeG,
  const F_TYPE *cexhG, const F_TYPE *ceyhG, const F_TYPE *cezhG,
  const size_t sX, const size_t sXY)
{
  // skip outer boundry
  if( (blockIdx.x == 0 and threadIdx.x == 0) or
      (blockIdx.y == 0 and threadIdx.y == 0) or
      (blockIdx.z == 0 and threadIdx.z == 0) or
      (blockIdx.x == (gridDim.x - 1) and threadIdx.x == (blockDim.x - 1)) or
      (blockIdx.y == (gridDim.y - 1) and threadIdx.y == (blockDim.y - 1)) or
      (blockIdx.z == (gridDim.z - 1) and threadIdx.z == (blockDim.z - 1))) return;

  const size_t n =
    (((blockIdx.z * blockDim.z) + threadIdx.z) * gridDim.x * blockDim.x * gridDim.y * blockDim.y)
  + (((blockIdx.y * blockDim.y) + threadIdx.y) * gridDim.x * blockDim.x)
  +   (blockIdx.x * blockDim.x) + threadIdx.x;

  exG[n] = cexeG[n] * exG[n]
    + cexhG[n] * ((hzG[n] - hzG[n - sX]) - (hyG[n] - hyG[n - sXY]));
  eyG[n] = ceyeG[n] * eyG[n]
    + ceyhG[n] * ((hxG[n] - hxG[n - sXY]) - (hzG[n] - hzG[n - 1]));
  ezG[n] = cezeG[n] * ezG[n]
    + cezhG[n] * ((hyG[n] - hyG[n - 1]) - (hxG[n] - hxG[n - sX]));
}
```

Code C0100: Visualization of output data

Volumetric visualization in ParaView.



Results after 10 time steps.

Questions and discussion?