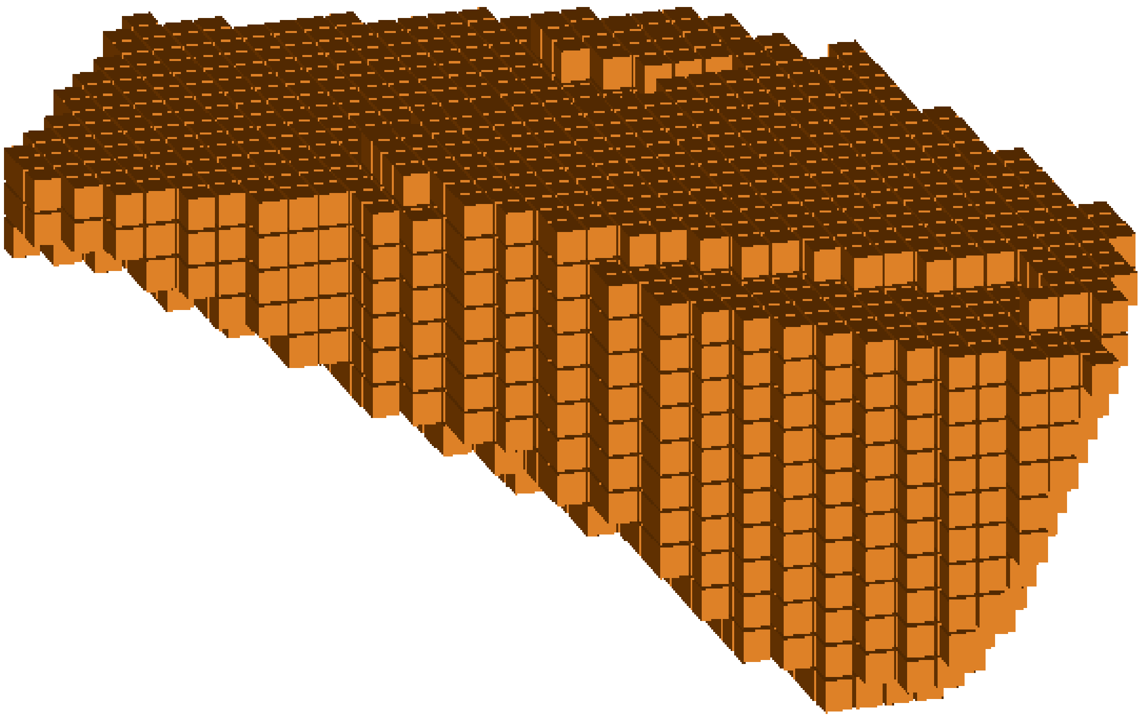
MiningMath Associates

Dense Cholesky CUDA implementation

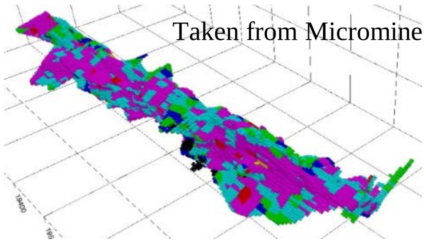
SimSched

Fabrício Ceolin e Alex Teixeira

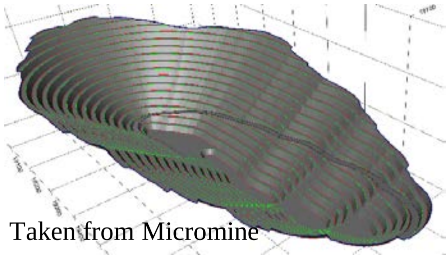
Block Model



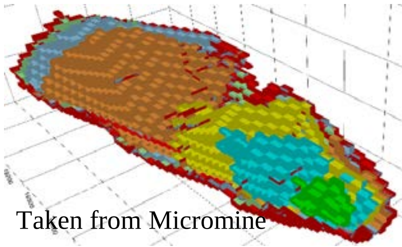
The current state of the mine planning technology

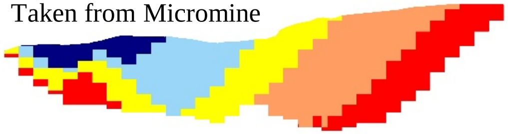

Taken from Micromine

Pre-assigned destinations
Single block values


Taken from Micromine

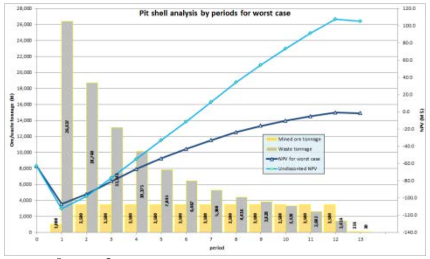
Lerchs-Grossman
Max. Undisc. Cash Flow


Taken from Micromine

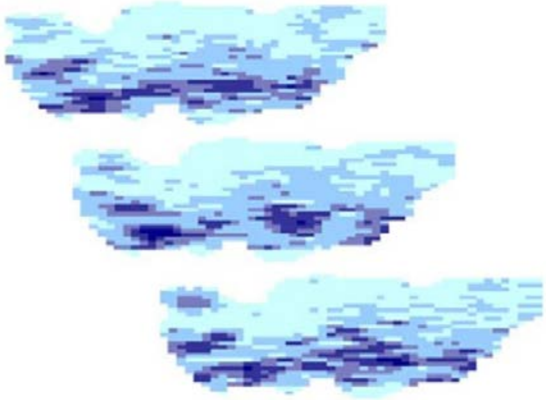
Taken from Micromine


Period-by-period scheduling with blending

Stockpiling and Cutoff policy

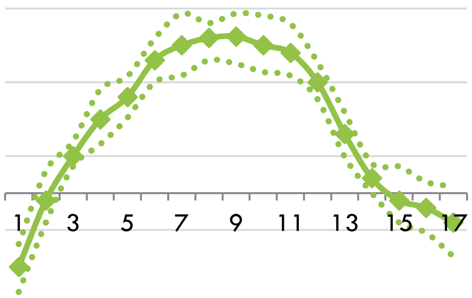


Taken from Micromine



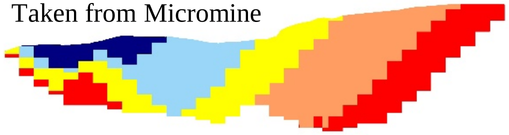
Leite and Dimitrakopoulos (2009)

Block values for each model and possible destination



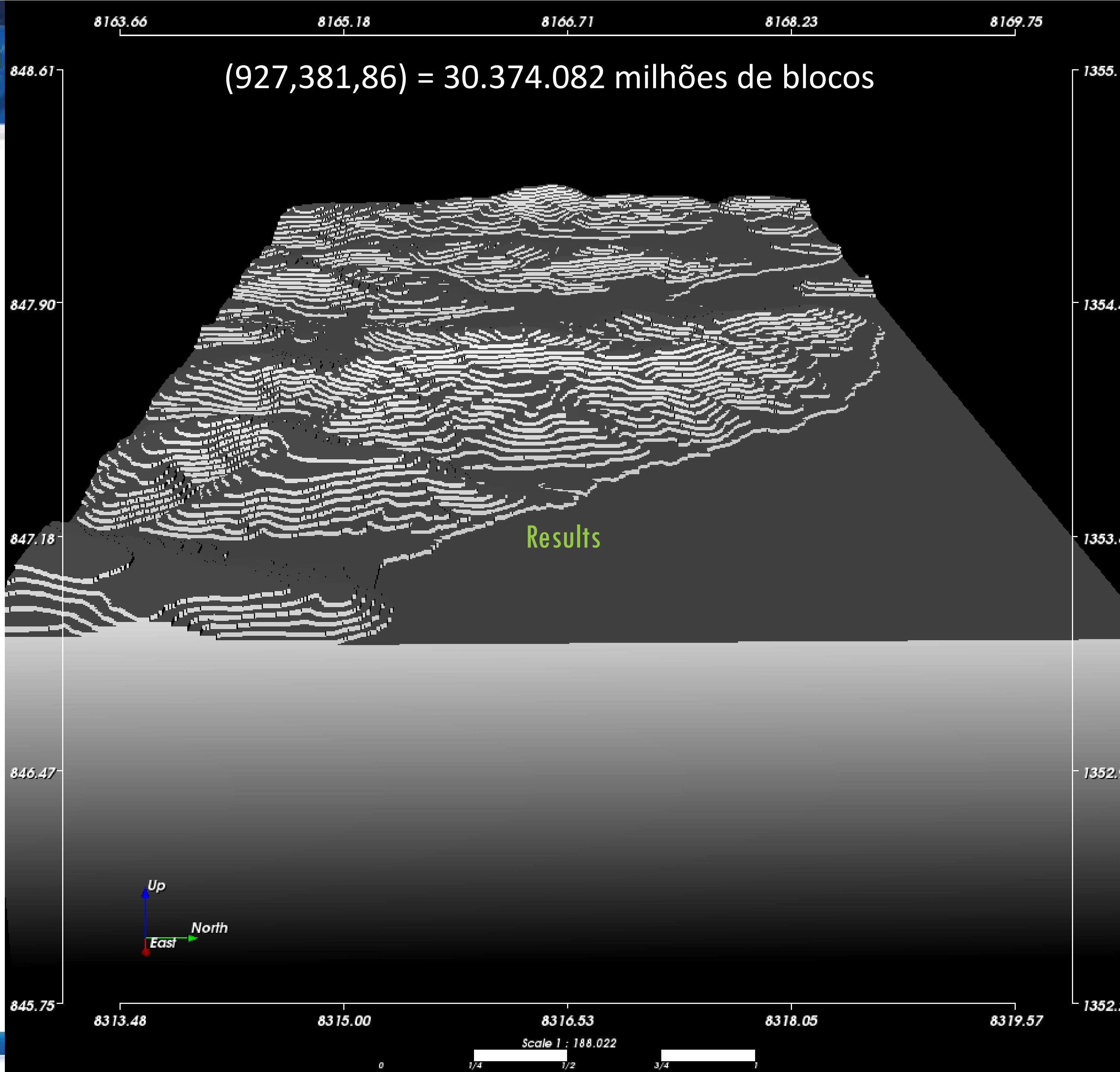
Indicators with risk profiles

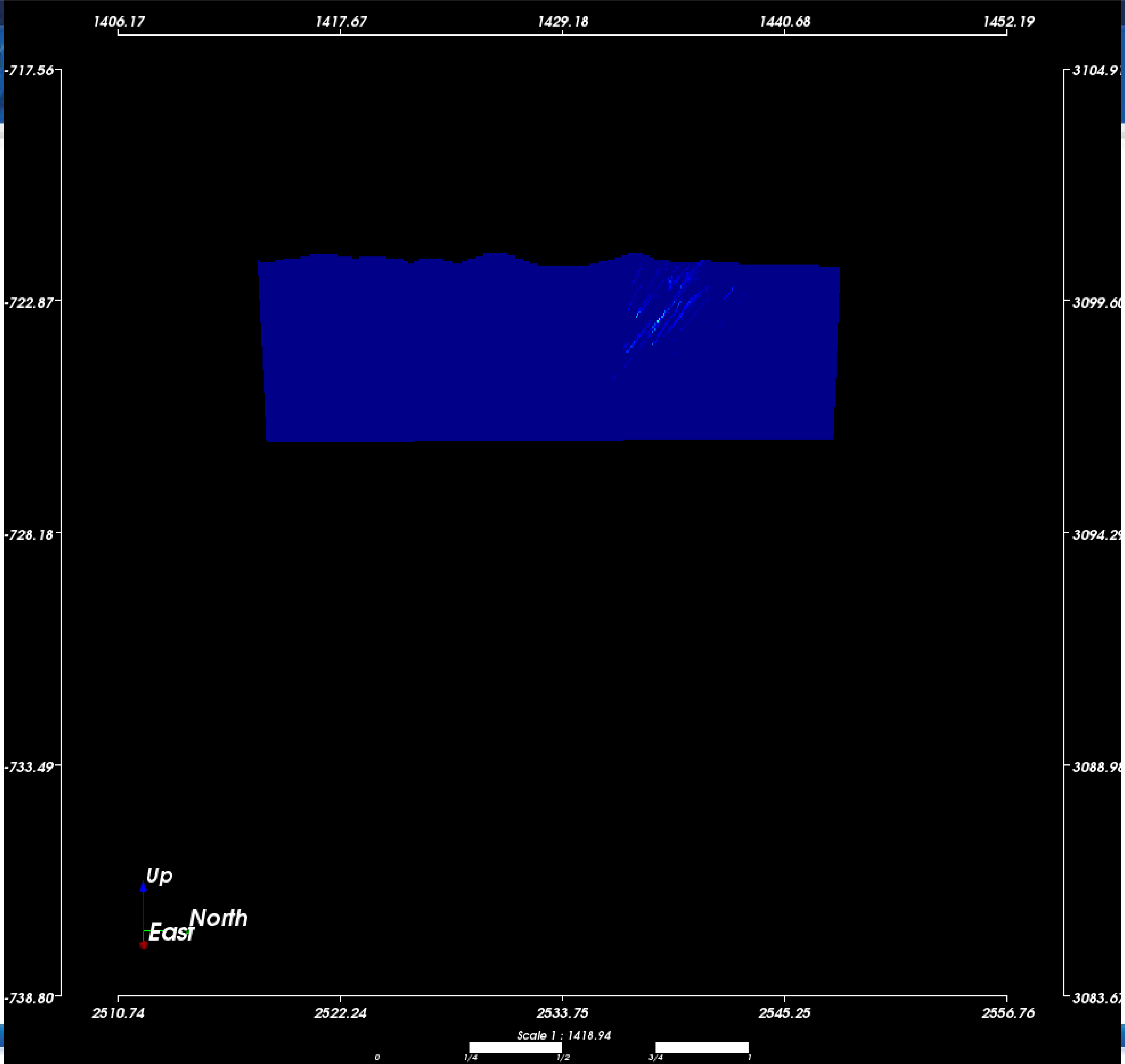
Taken from Micromine

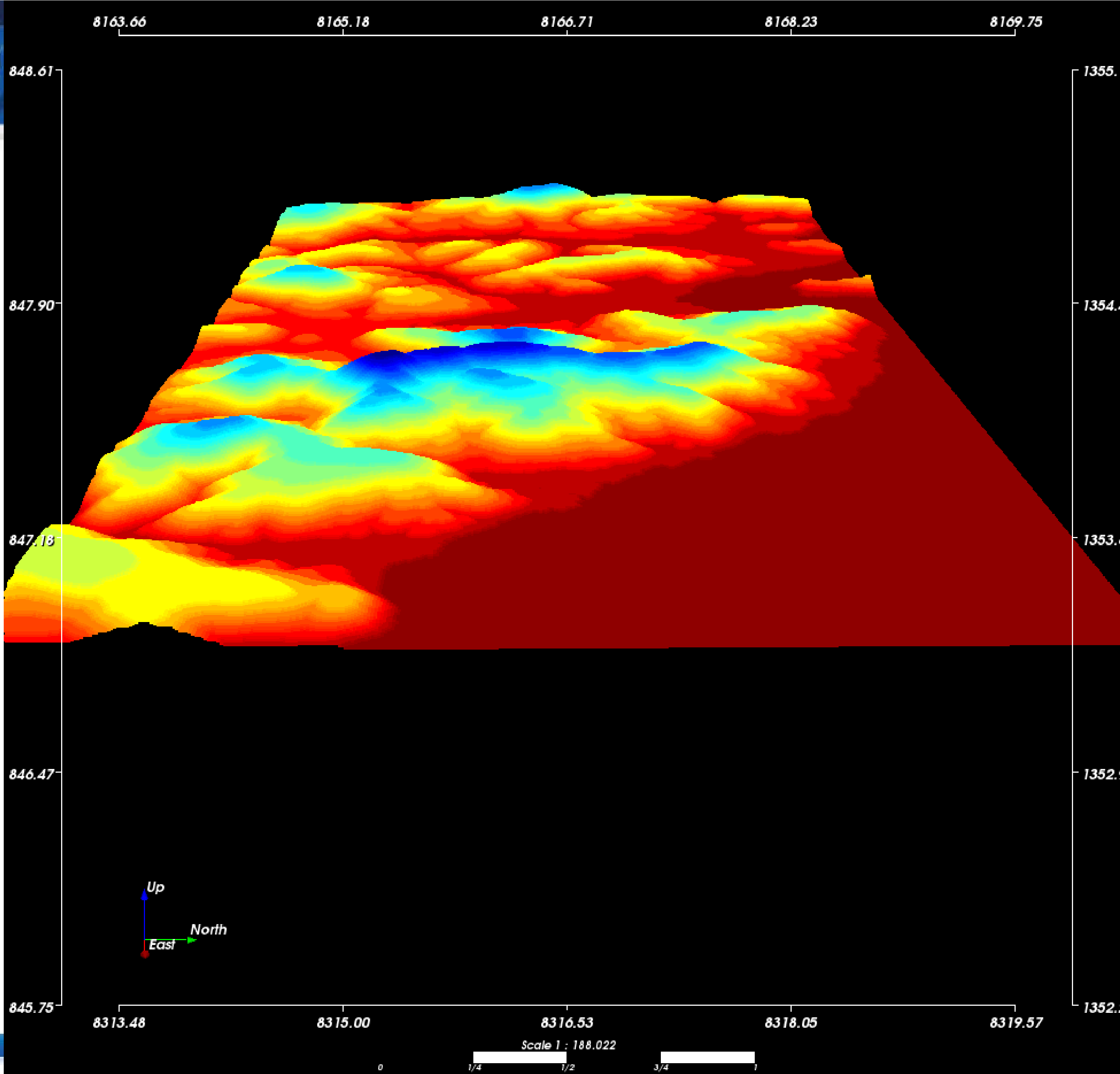


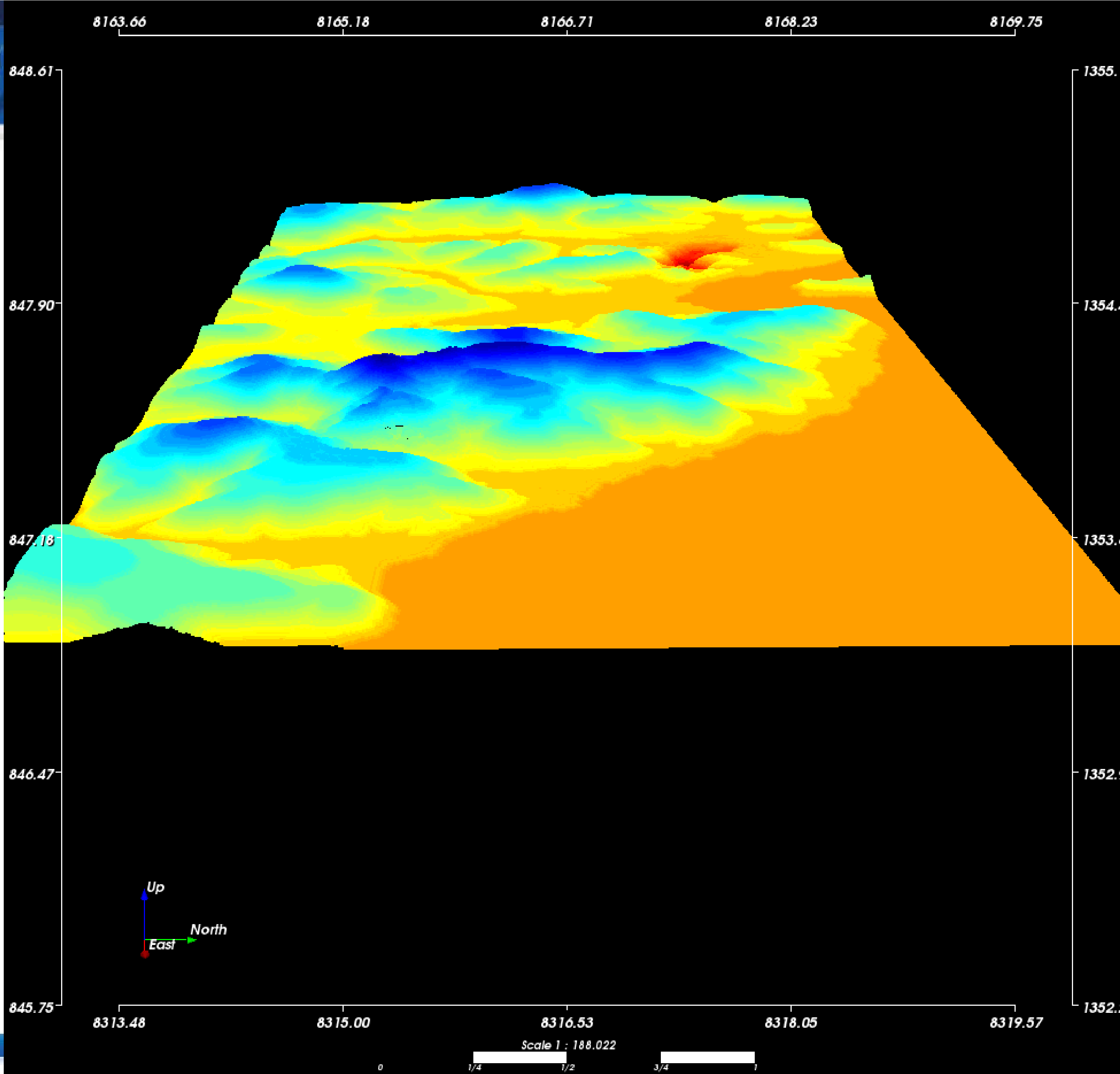
Multi-period scheduling under uncertainty with blending and stockpiling

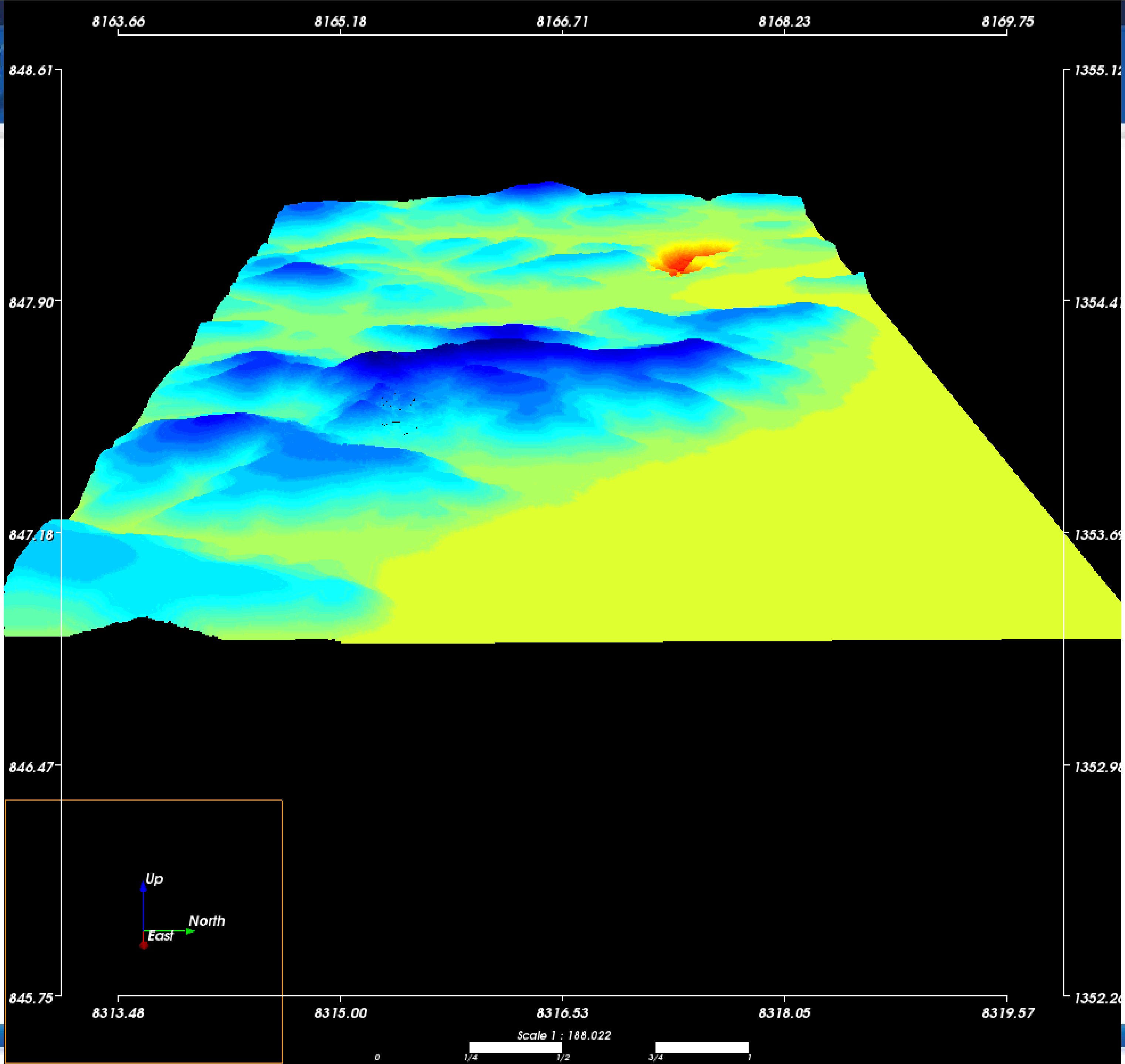
Results

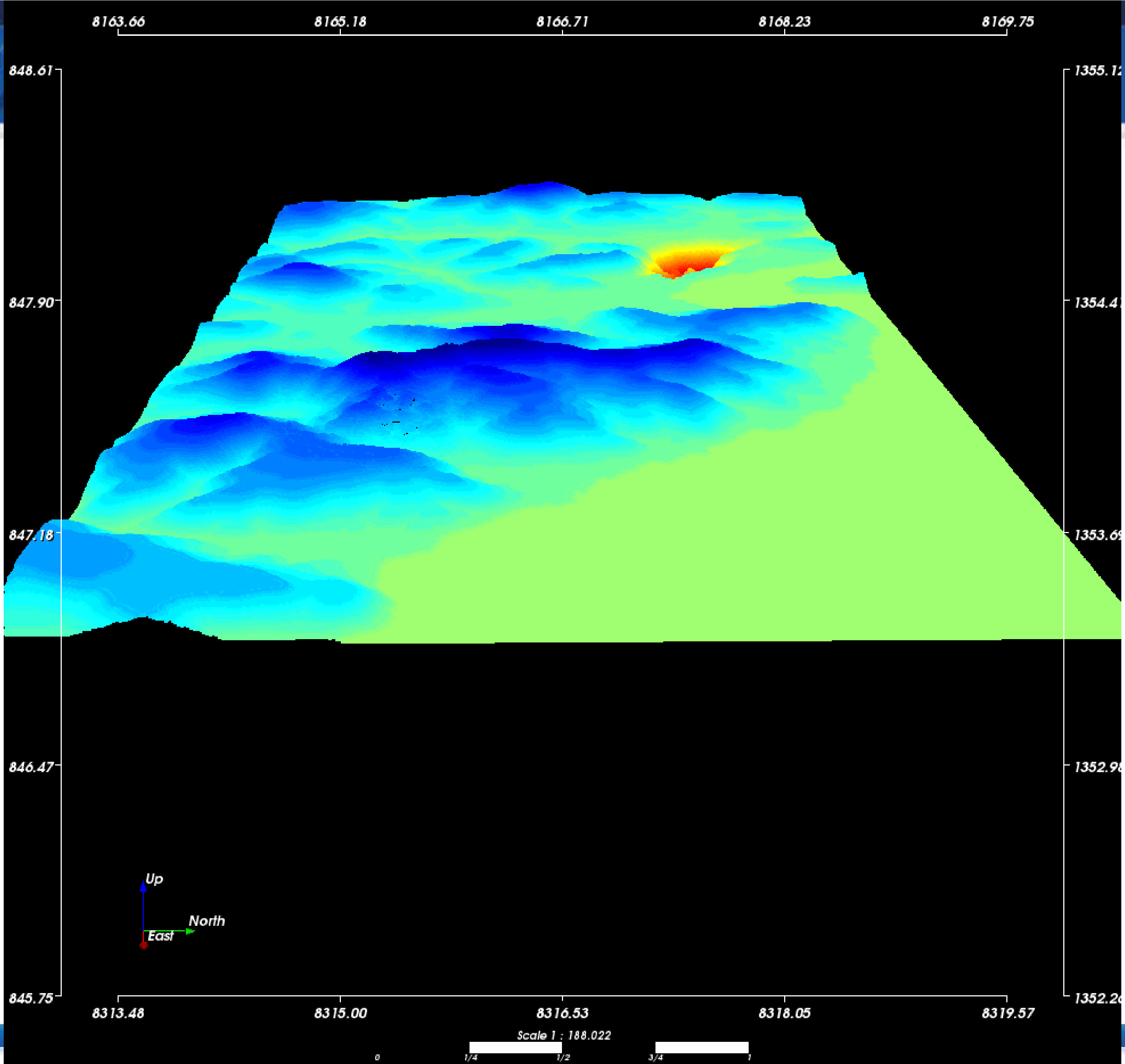

(927,381,86) = 30.374.082 milhões de blocos
8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
Up
North
East
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022

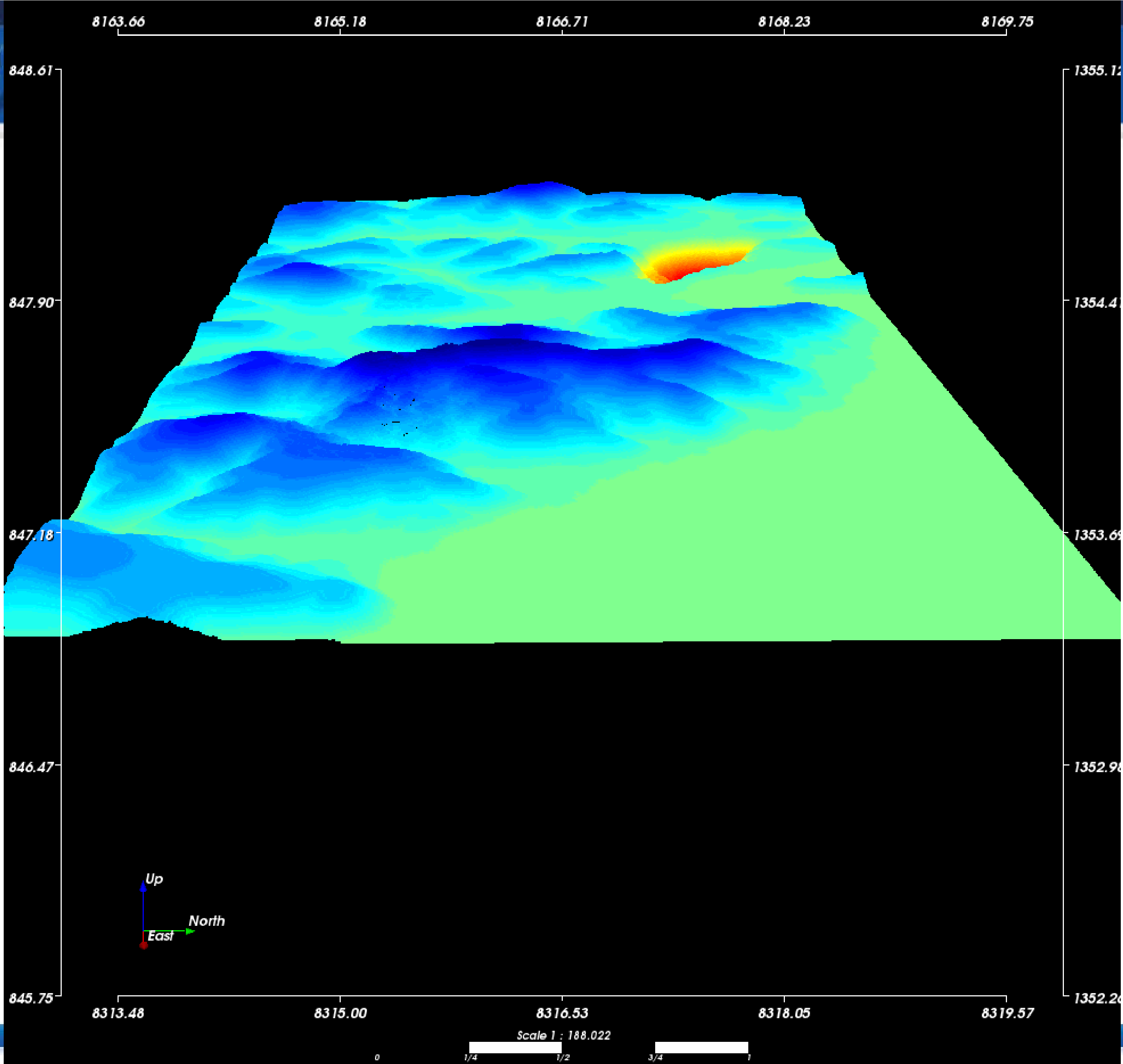

1406.17
1417.67
1429.18
1440.68
1452.19
-717.56
-722.87
-728.18
-733.49
-738.80
3104.97
3099.66
3094.29
3088.98
3083.67
Up
North
East
2510.74
2522.24
2533.75
2545.25
2556.76
Scale 1 : 1418.94
0
1/4
1/2
3/4
1

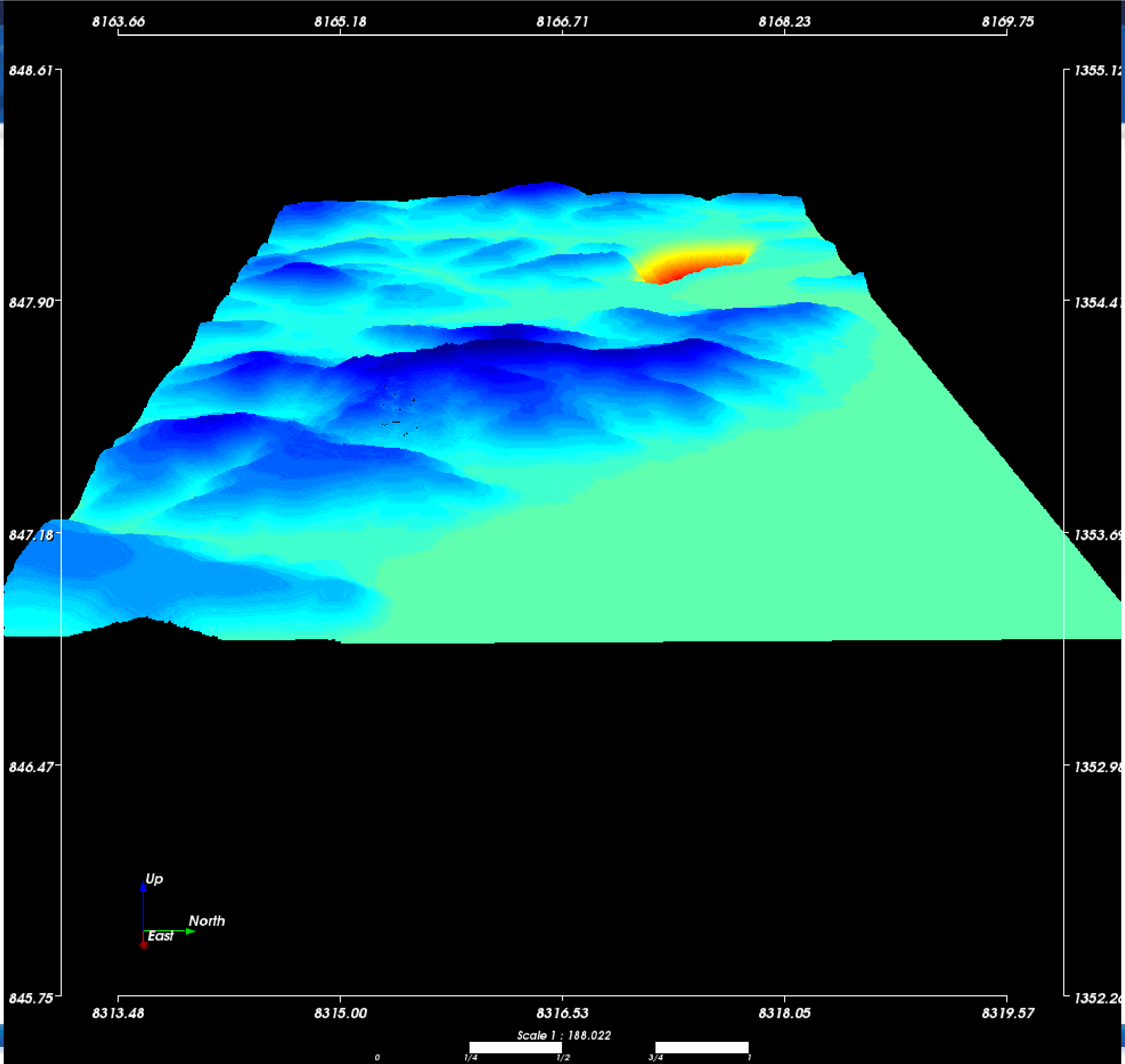

8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
0
1/4
1/2
3/4
1
Up
North
East

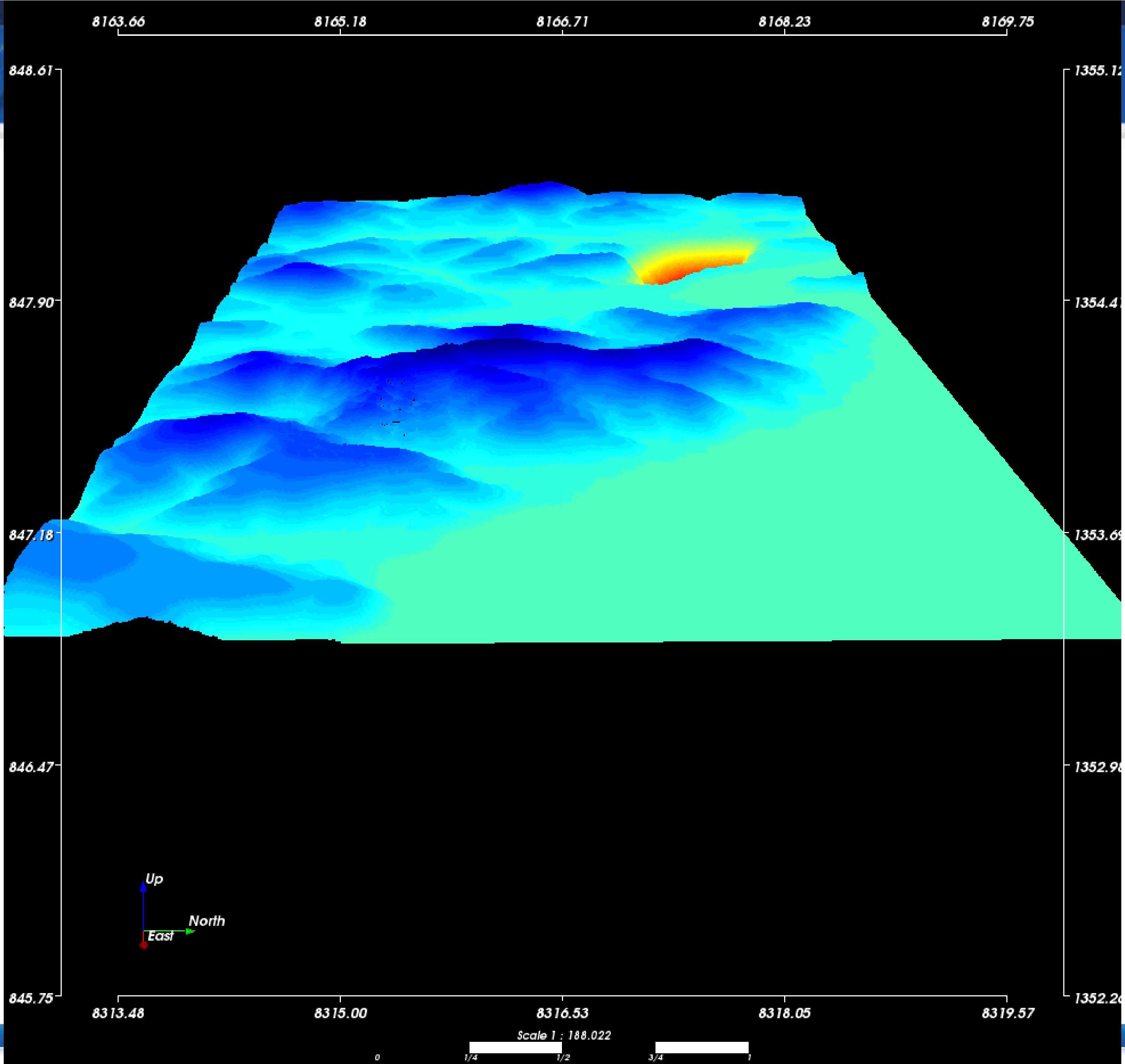

8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.1
1354.4
1353.6
1352.9
1352.2
Up
North
East
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
0
1/4
1/2
3/4
1

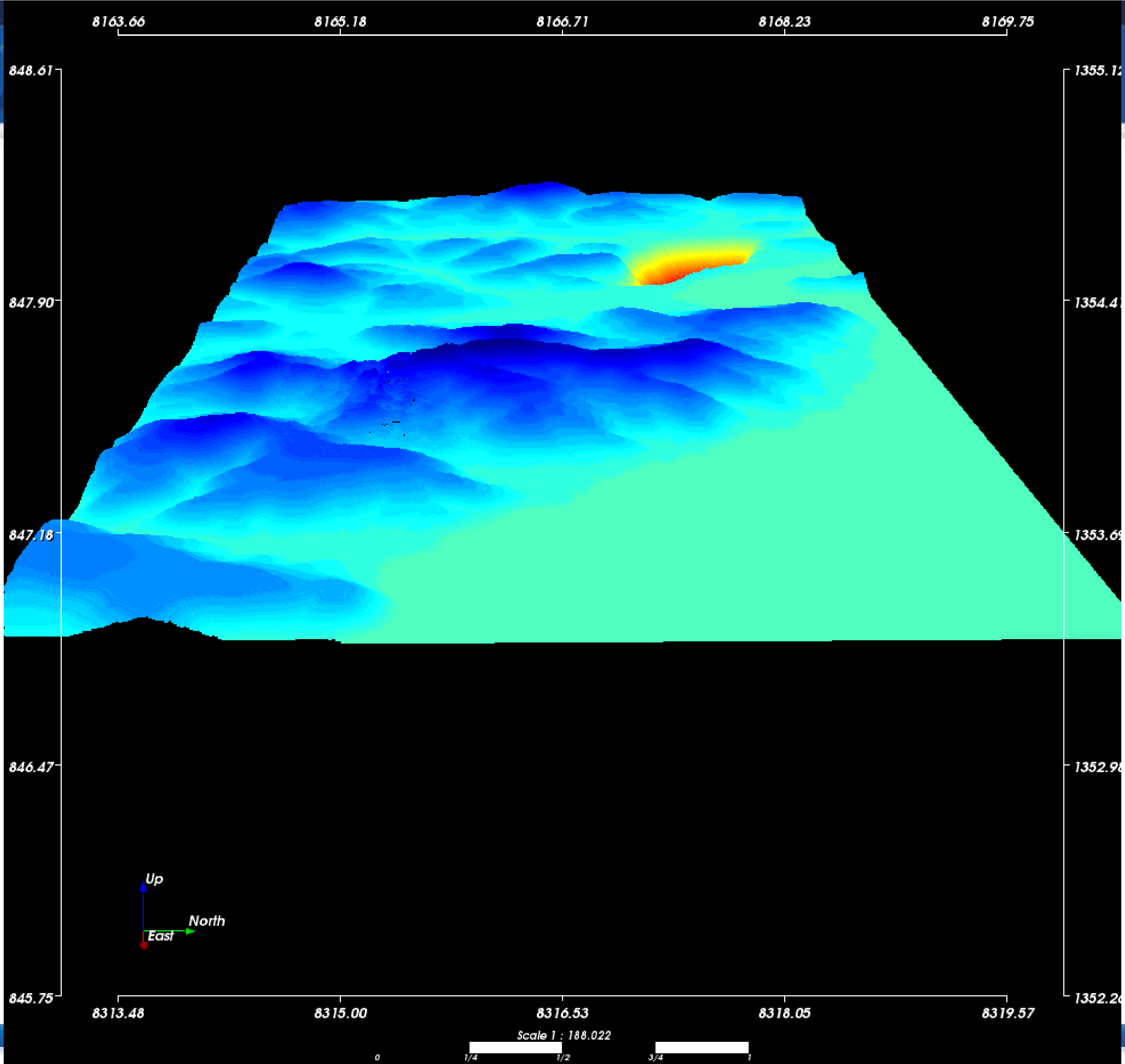

8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.12
1354.41
1353.69
1352.98
1352.26
Up
North
East
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
0
1/4
1/2
3/4
1


8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
Up
North
East
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022

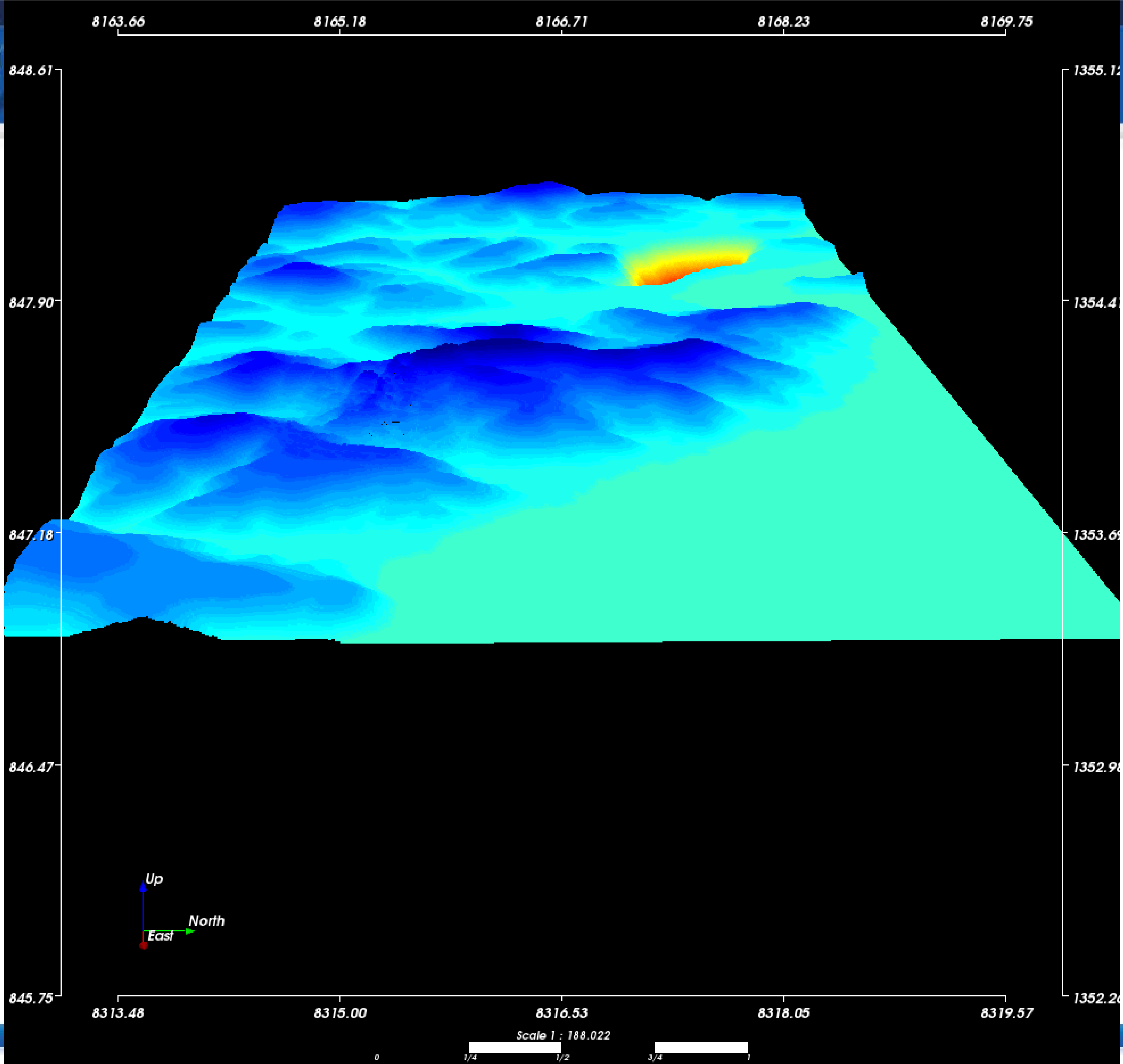

8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
Up
North
East
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
0
1/4
1/2
3/4
1


8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.12
1354.41
1353.69
1352.98
1352.26
Up
North
East
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
0
1/4
1/2
3/4
1

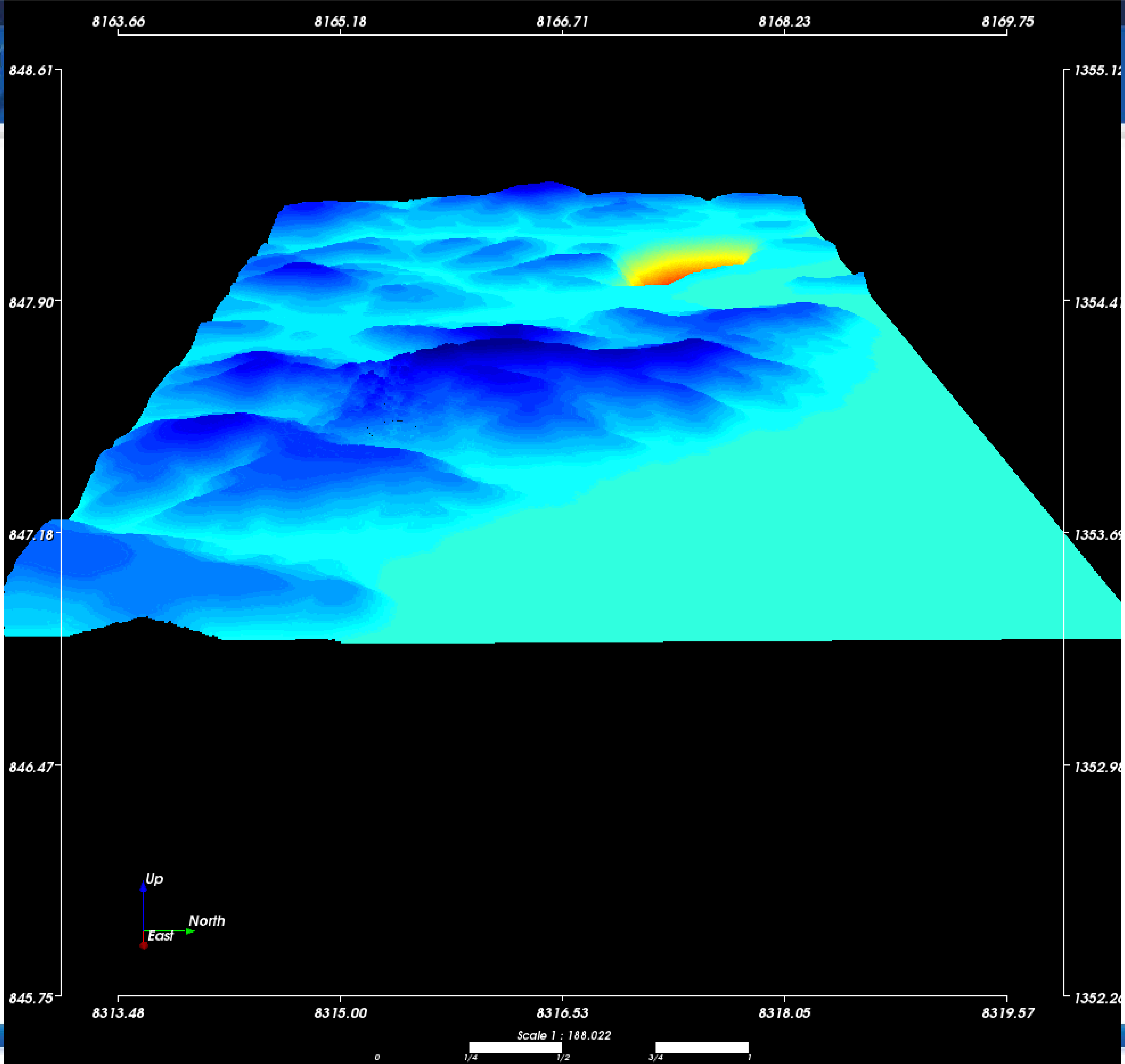

8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
8313.48
8315.00
8316.53
8318.05
8319.57
Up
North
East
Scale 1 : 188.022
0
1/4
1/2
3/4
1



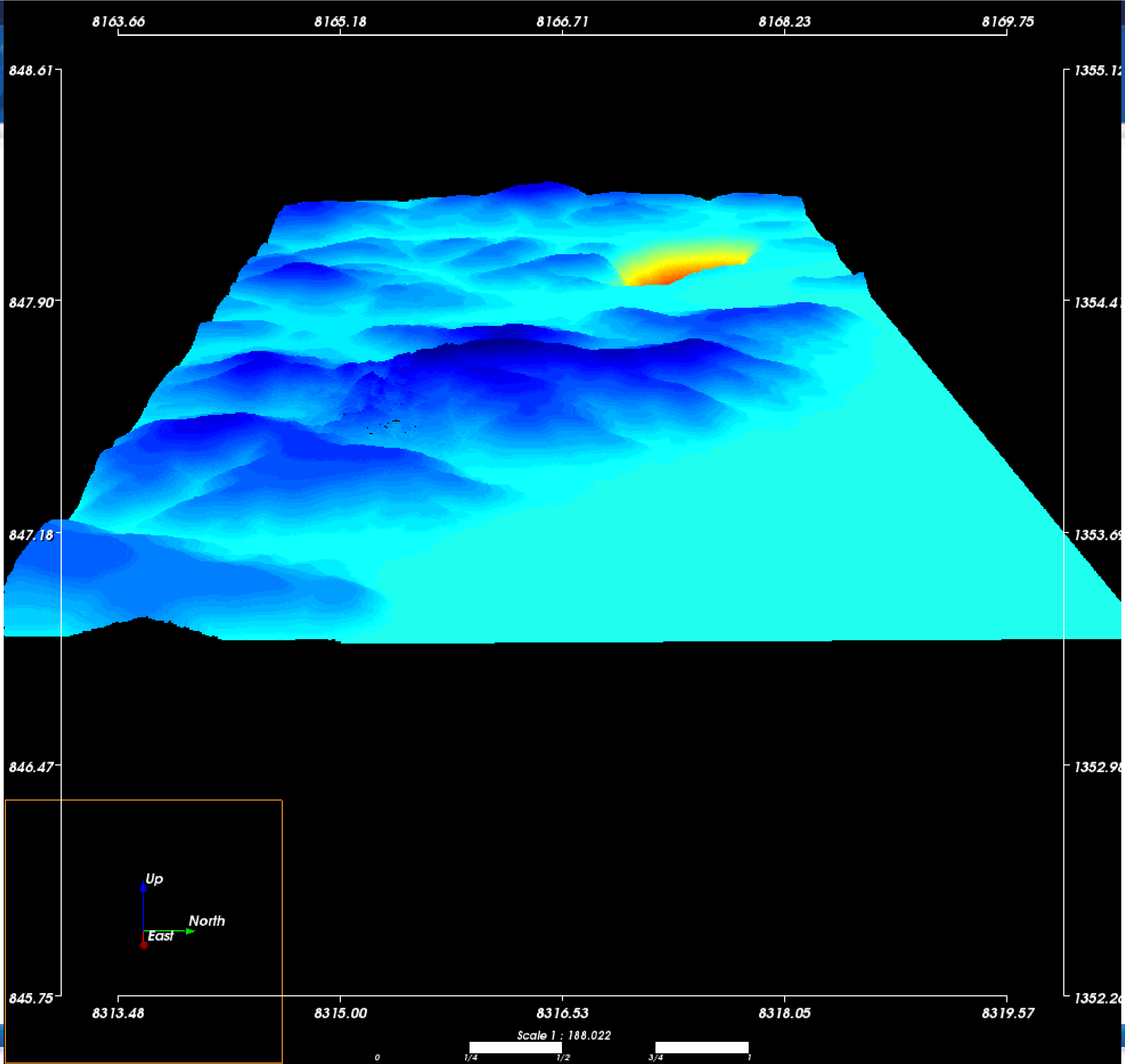
8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.12
1354.41
1353.69
1352.98
1352.26
8313.48
8315.00
8316.53
8318.05
8319.57
Up
North
East
Scale 1 : 188.022
0
1/4
1/2
3/4
1



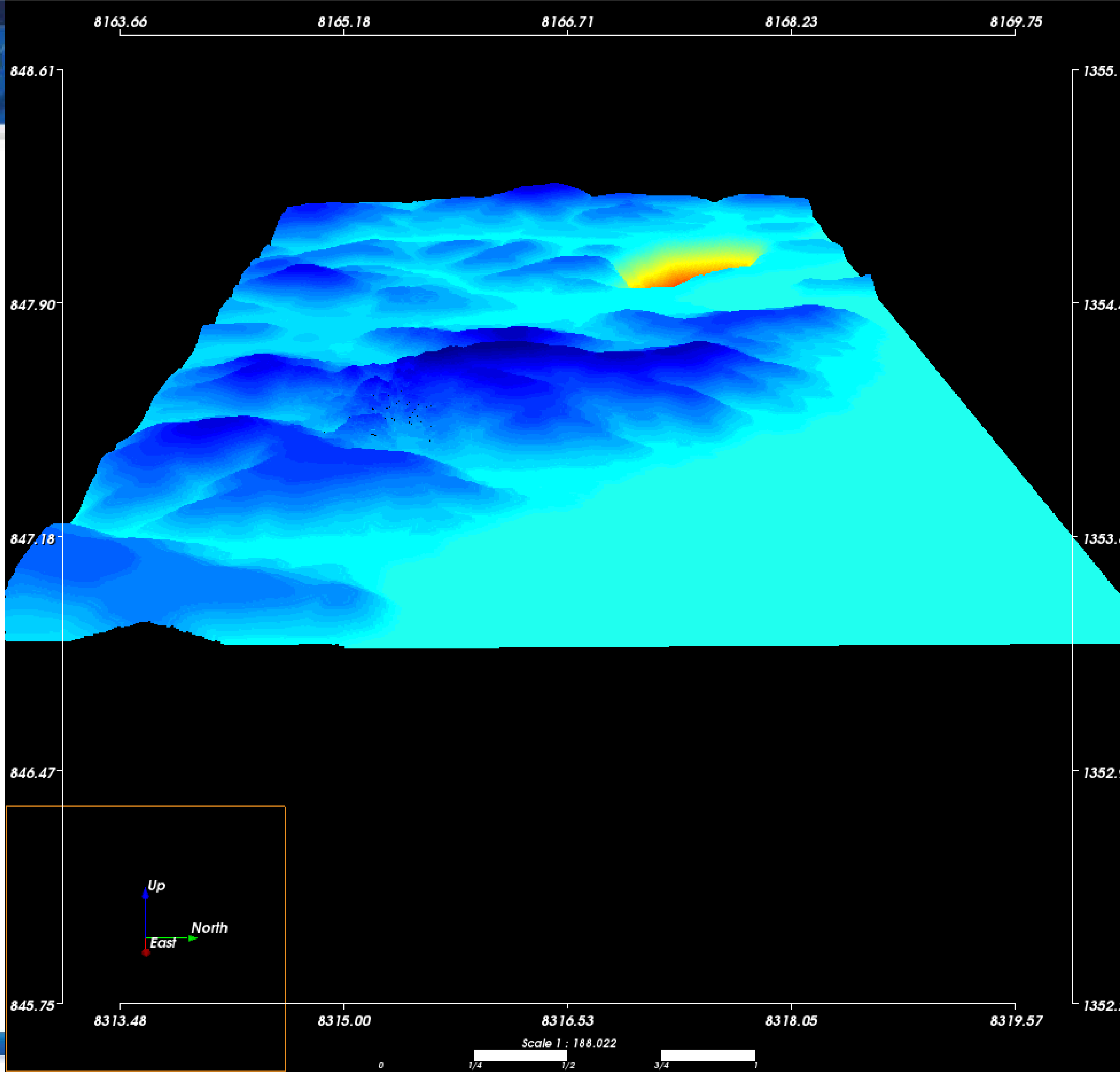
8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
8313.48
8315.00
8316.53
8318.05
8319.57
Up
North
East
Scale 1 : 188.022
0
1/4
1/2
3/4
1

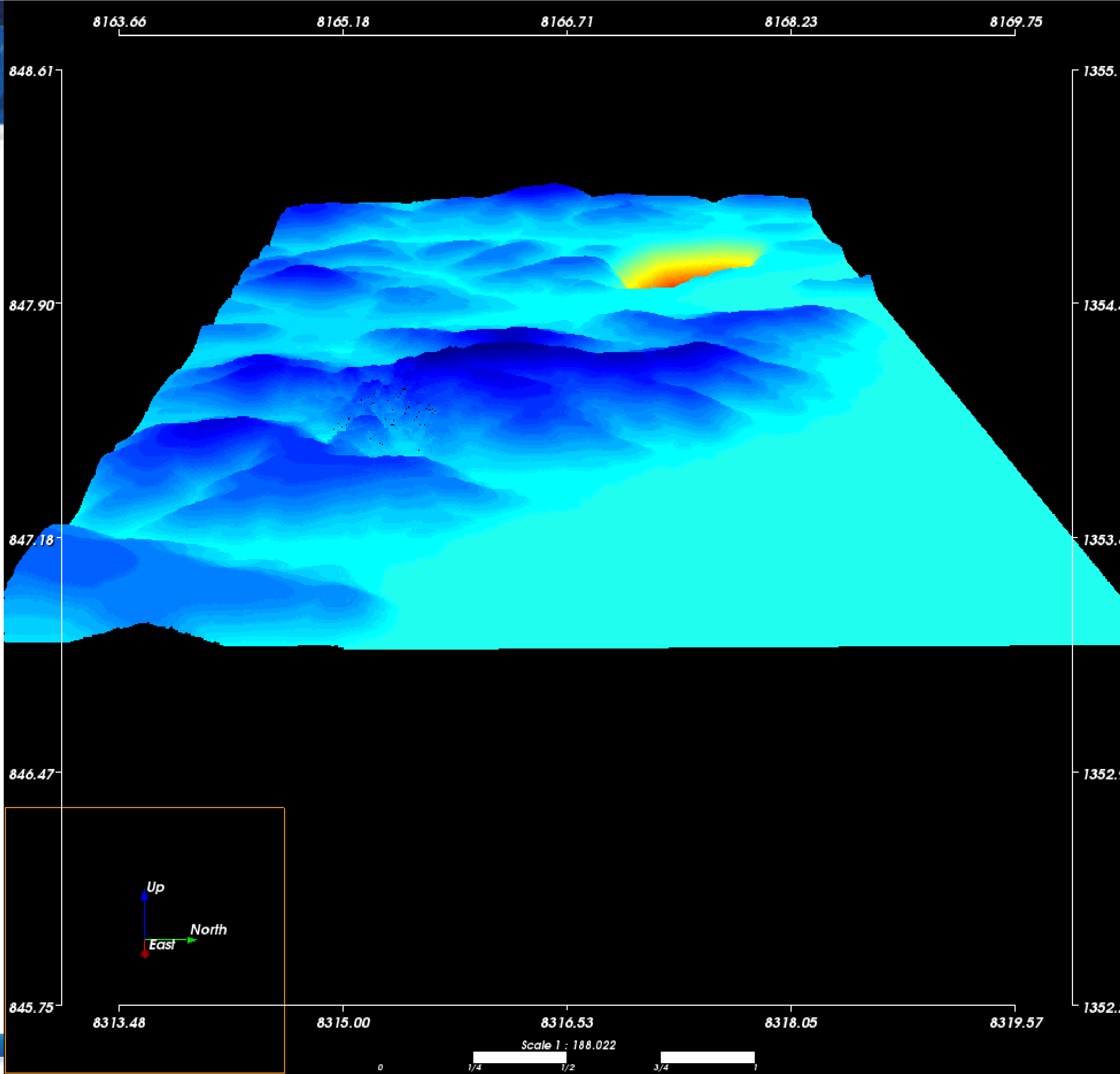


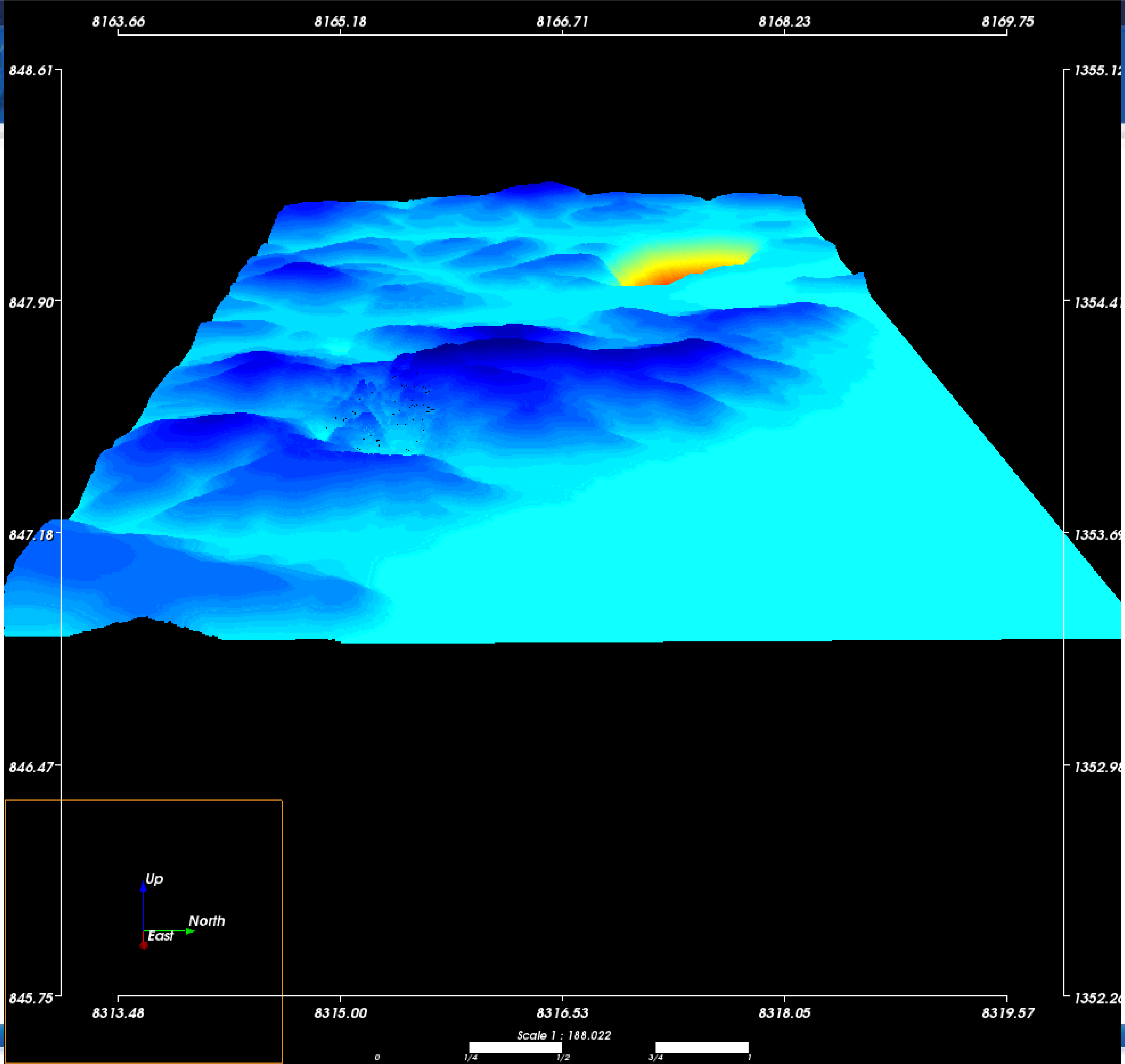
8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.12
1354.41
1353.69
1352.98
1352.26
Up
North
East
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
0
1/4
1/2
3/4
1

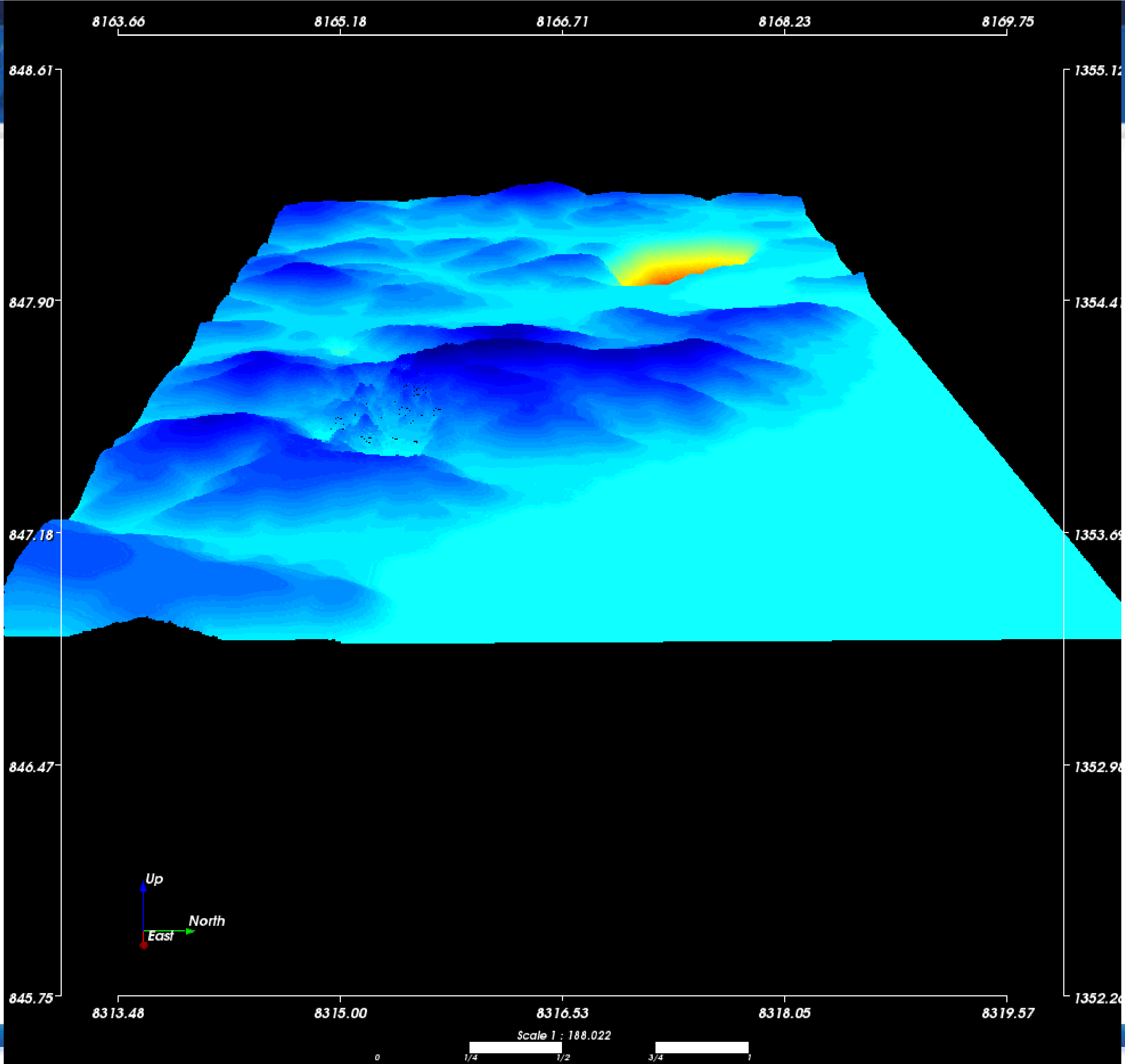


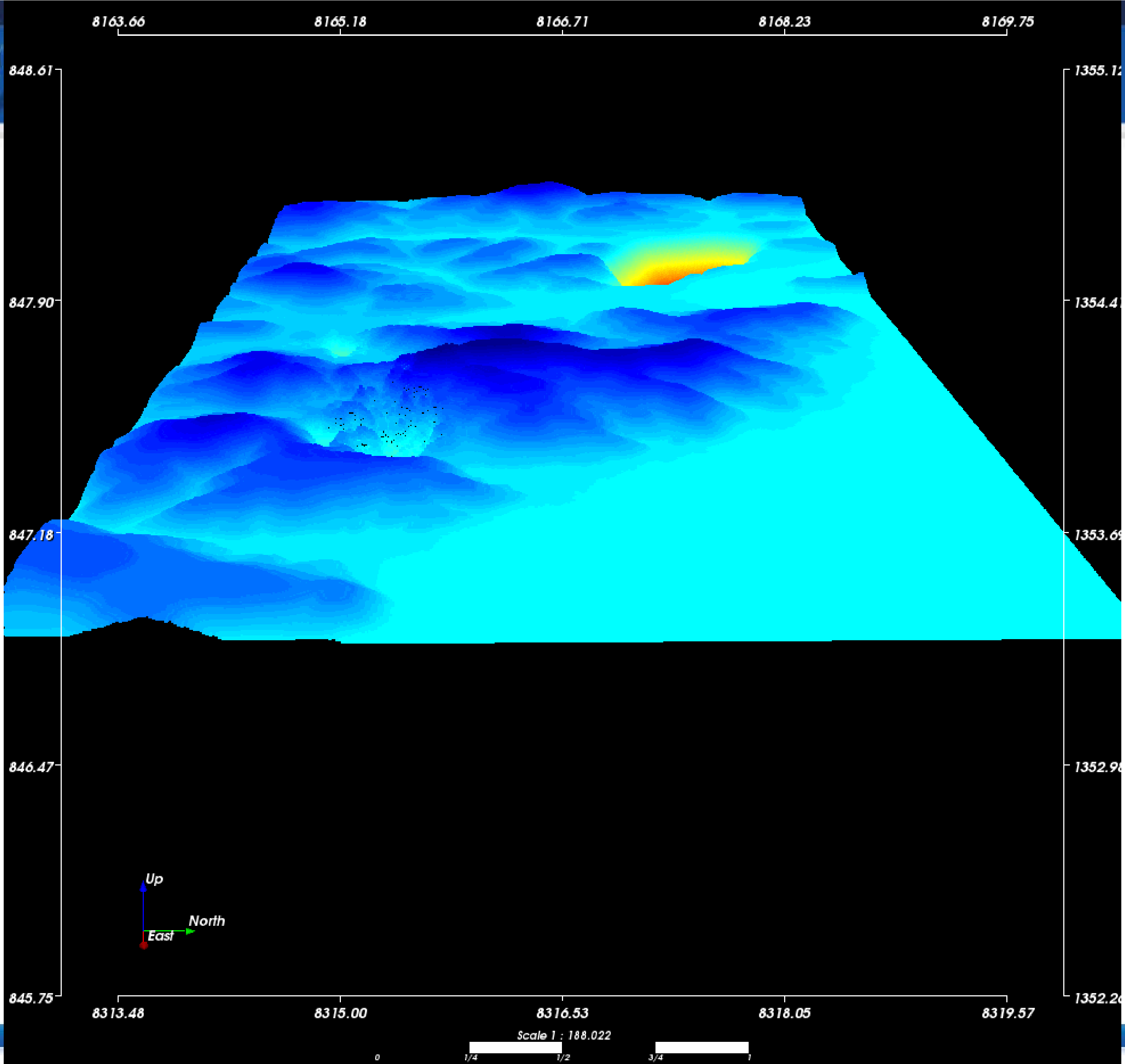
8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
8313.48
8315.00
8316.53
8318.05
8319.57
Up
North
East
Scale 1 : 188.022


8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.1
1354.4
1353.6
1352.9
1352.2
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
0
1/4
1/2
3/4
1
Up
North
East

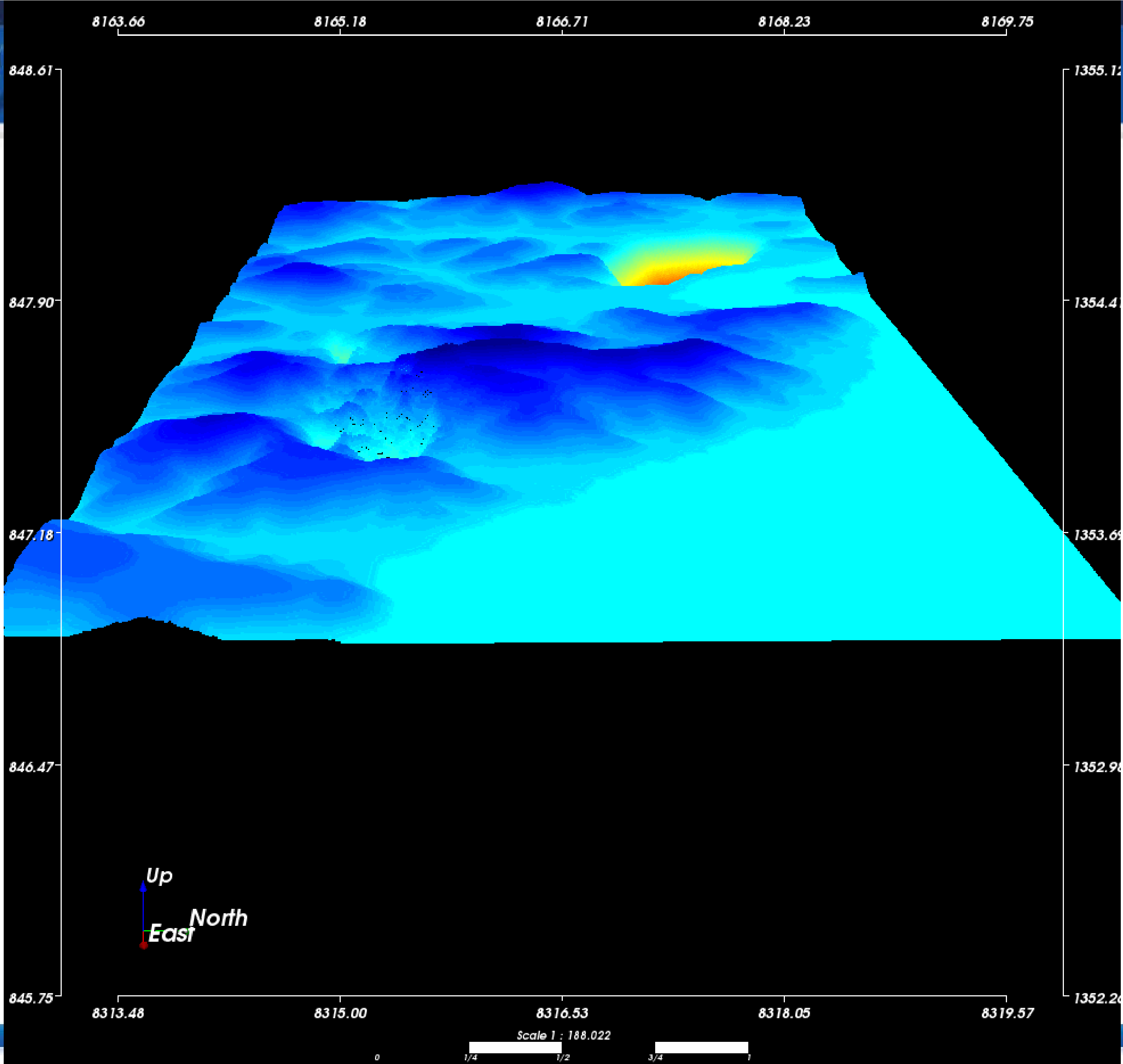

8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.12
1354.41
1353.69
1352.98
1352.26
8313.48
8315.00
8316.53
8318.05
8319.57
Up
North
East
Scale 1 : 188.022
0
1/4
1/2
3/4
1


8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.12
1354.41
1353.69
1352.98
1352.26
Up
North
East
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
0
1/4
1/2
3/4
1

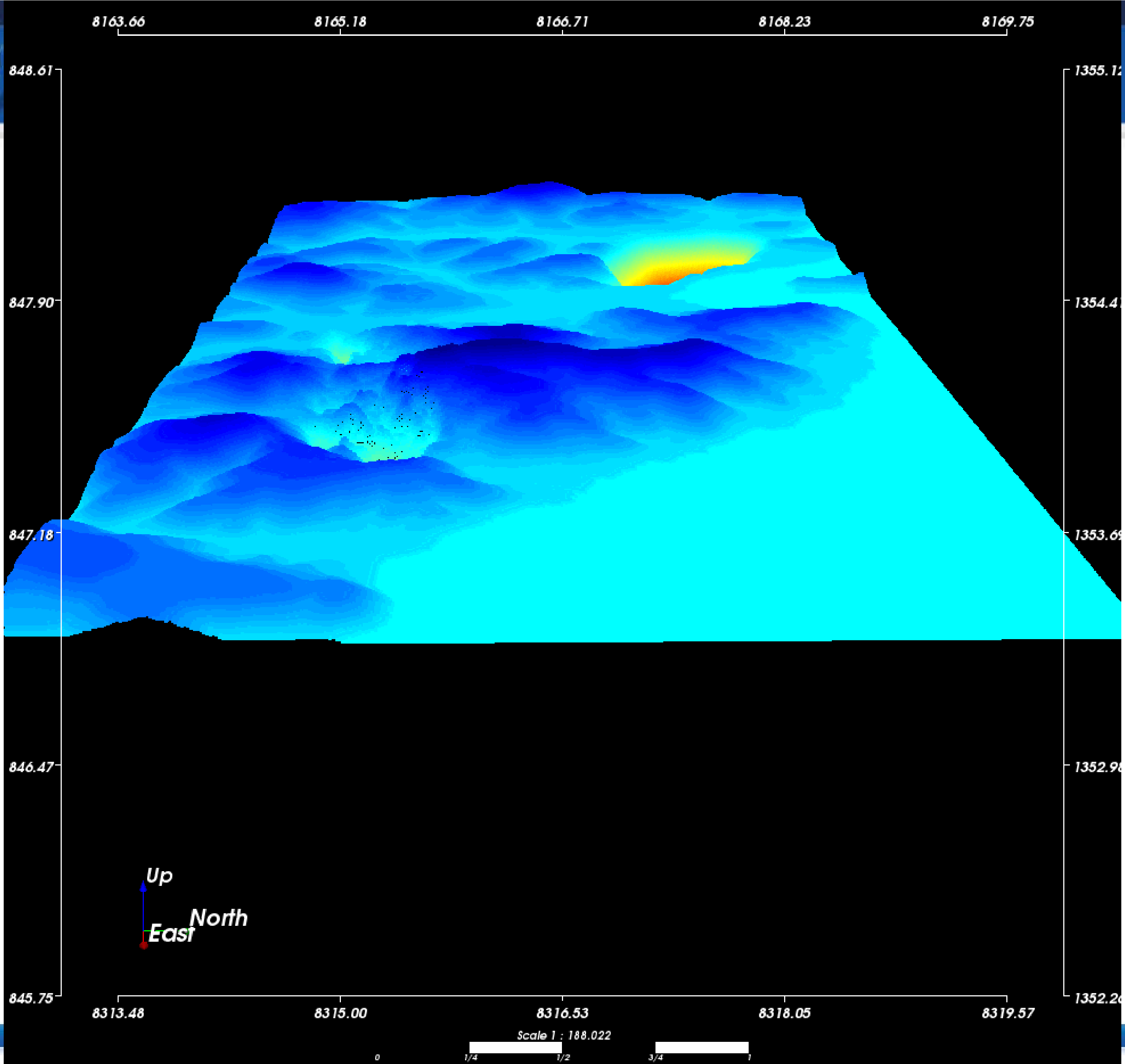

8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.12
1354.41
1353.69
1352.98
1352.26
Up
North
East
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
0
1/4
1/2
3/4
1

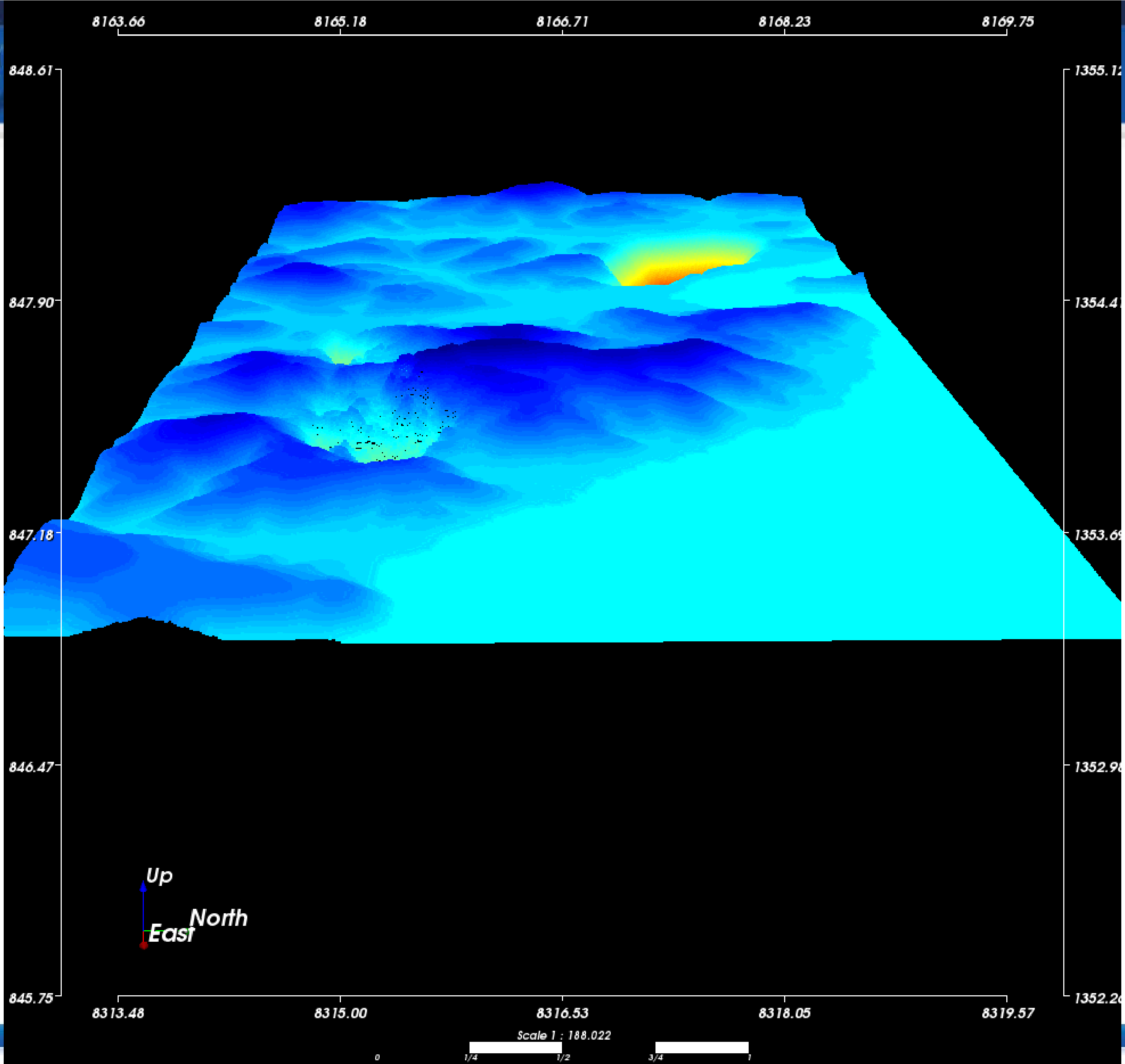


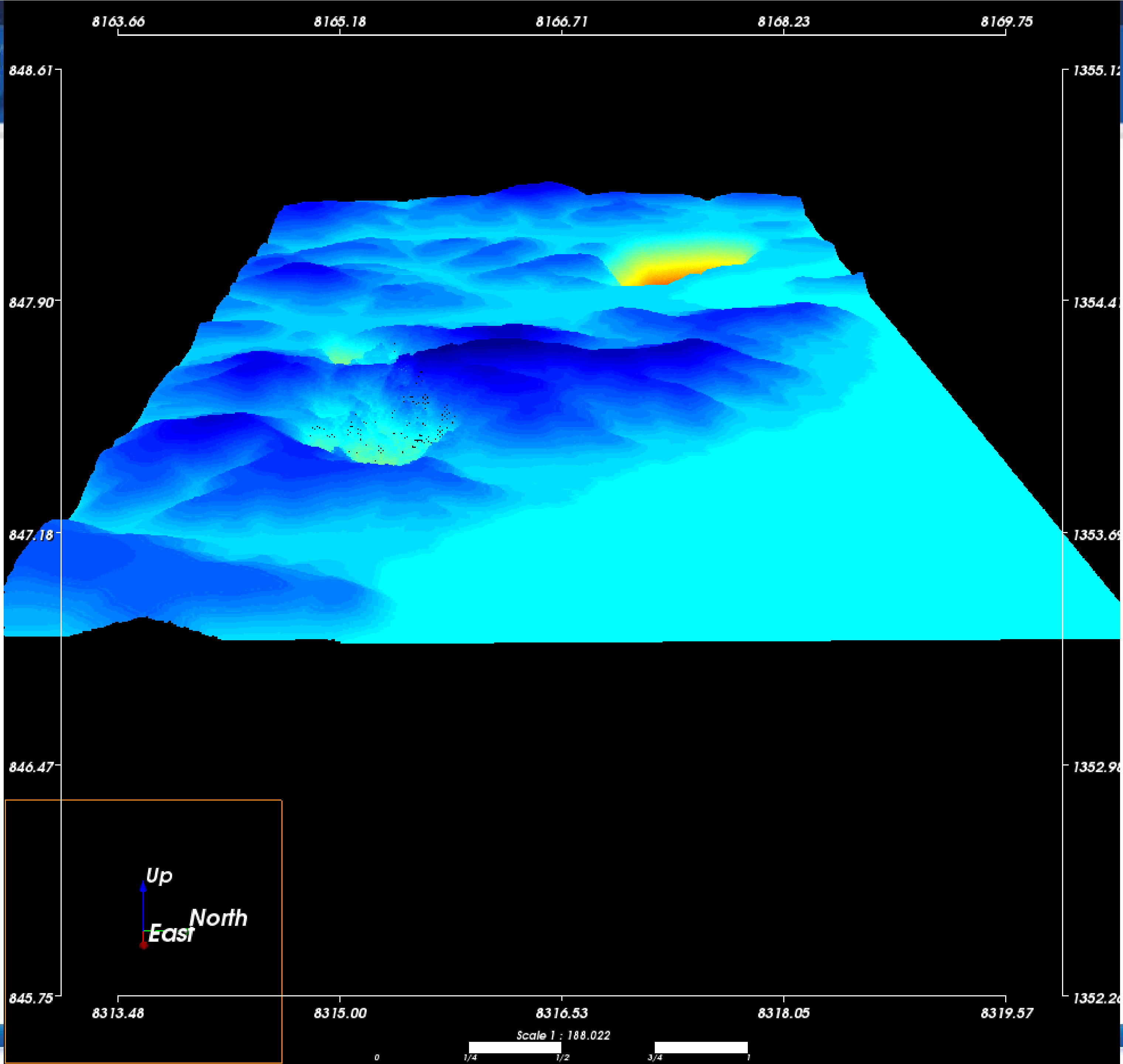
8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
Up
North
East
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
0
1/4
1/2
3/4
1

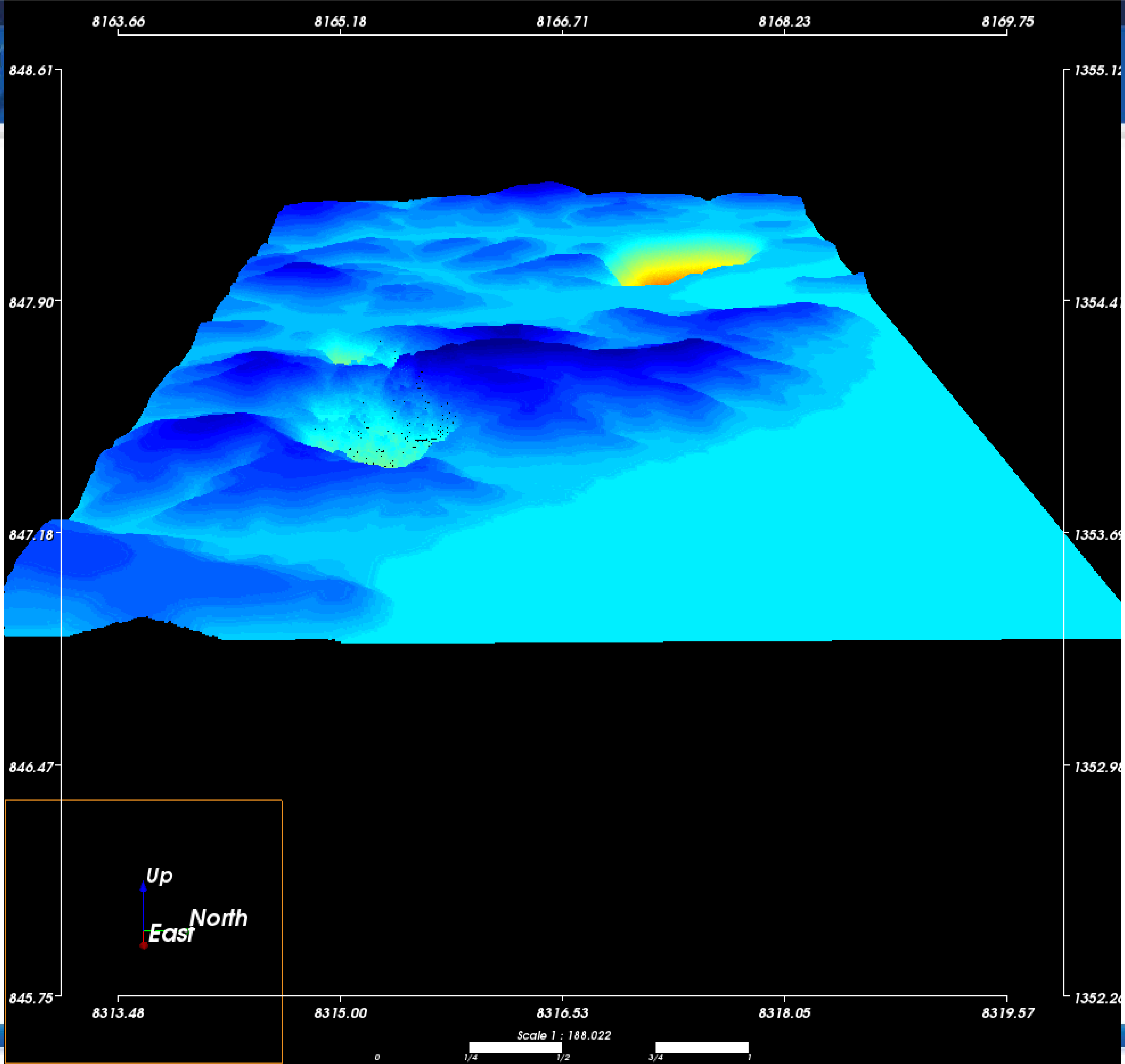


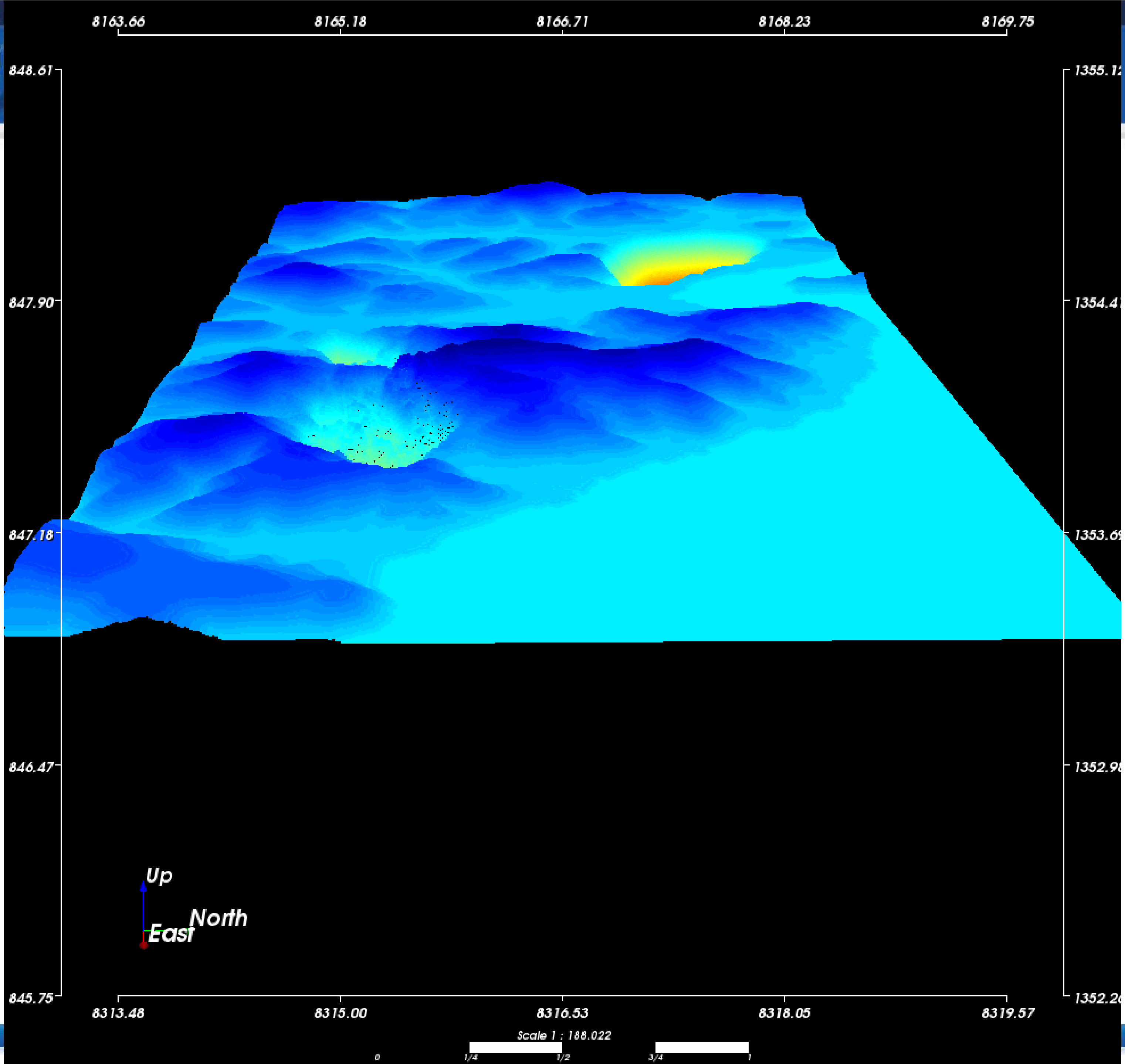
8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.12
1354.41
1353.69
1352.98
1352.26
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
Up
North
East


8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.12
1354.41
1353.69
1352.98
1352.26
Up
North
East
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
0
1/4
1/2
3/4
1

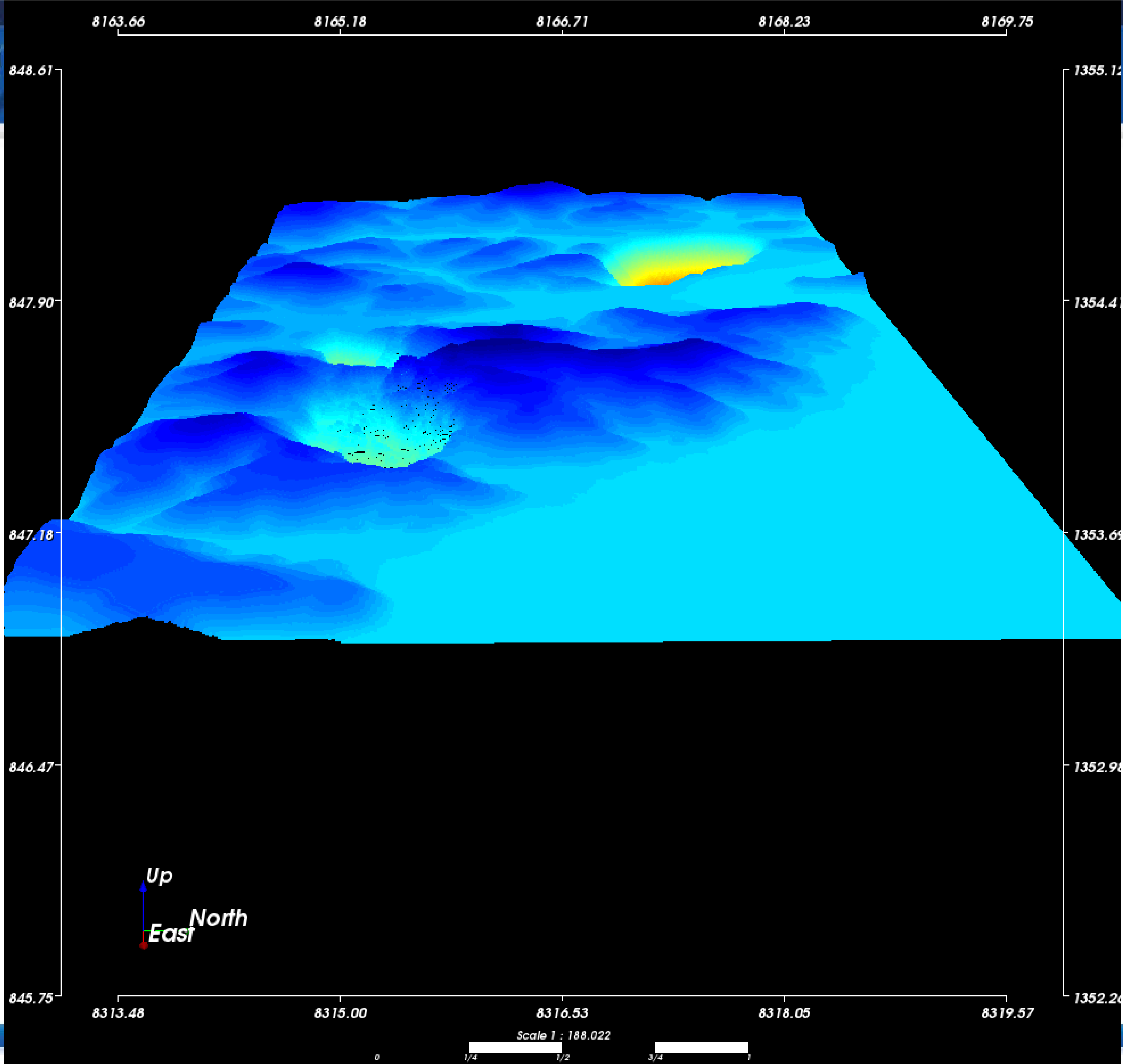

8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.12
1354.41
1353.69
1352.98
1352.26
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
Up
North
East


8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.12
1354.41
1353.69
1352.98
1352.26
8313.48
8315.00
8316.53
8318.05
8319.57
Up
North
East
Scale 1 : 188.022
0
1/4
1/2
3/4
1

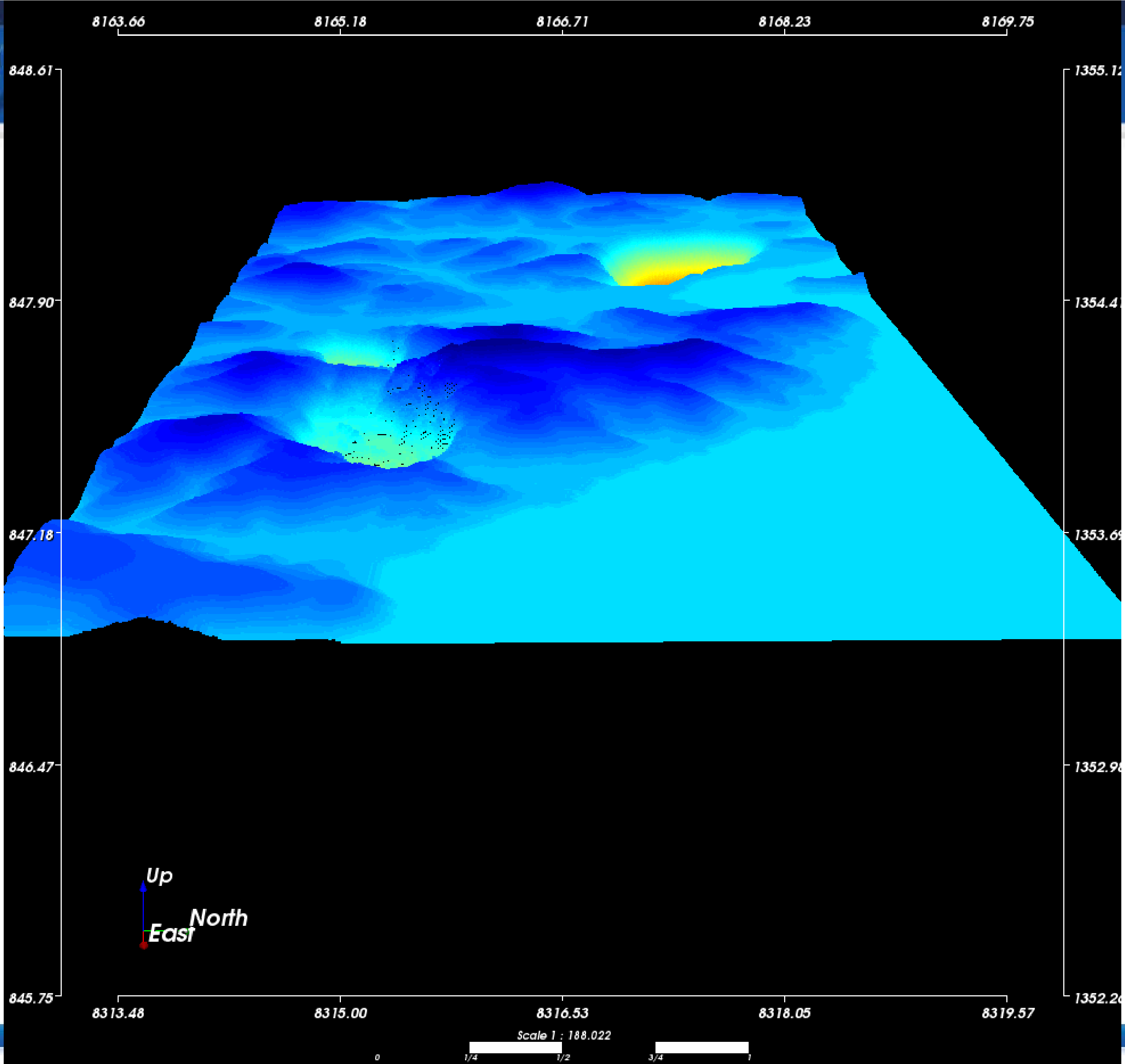

8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.12
1354.41
1353.69
1352.98
1352.26
Up
North
East
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
0
1/4
1/2
3/4
1

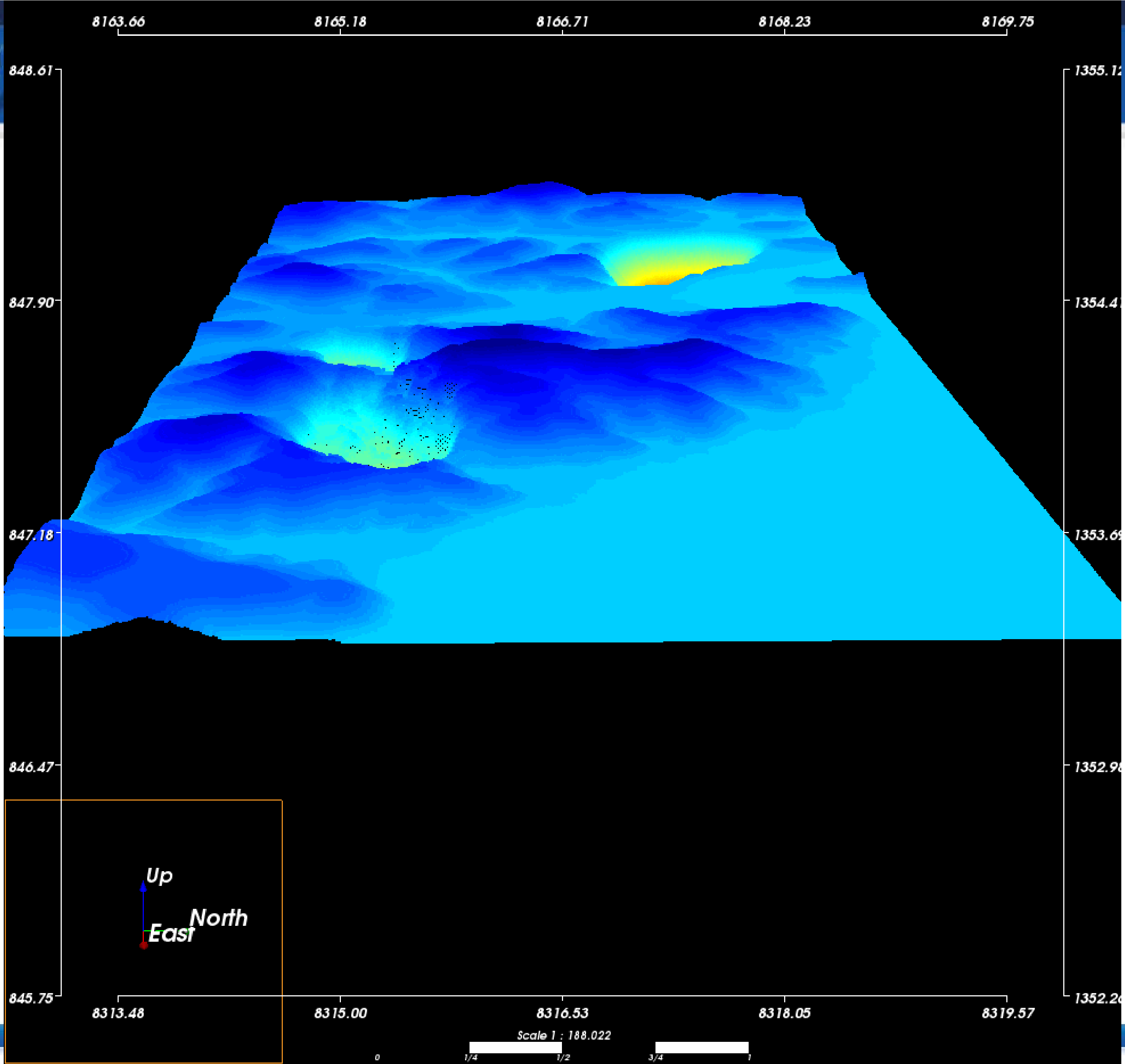


8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.12
1354.41
1353.69
1352.98
1352.26
8313.48
8315.00
8316.53
8318.05
8319.57
Up
North
East
Scale 1 : 188.022
0
1/4
1/2
3/4
1

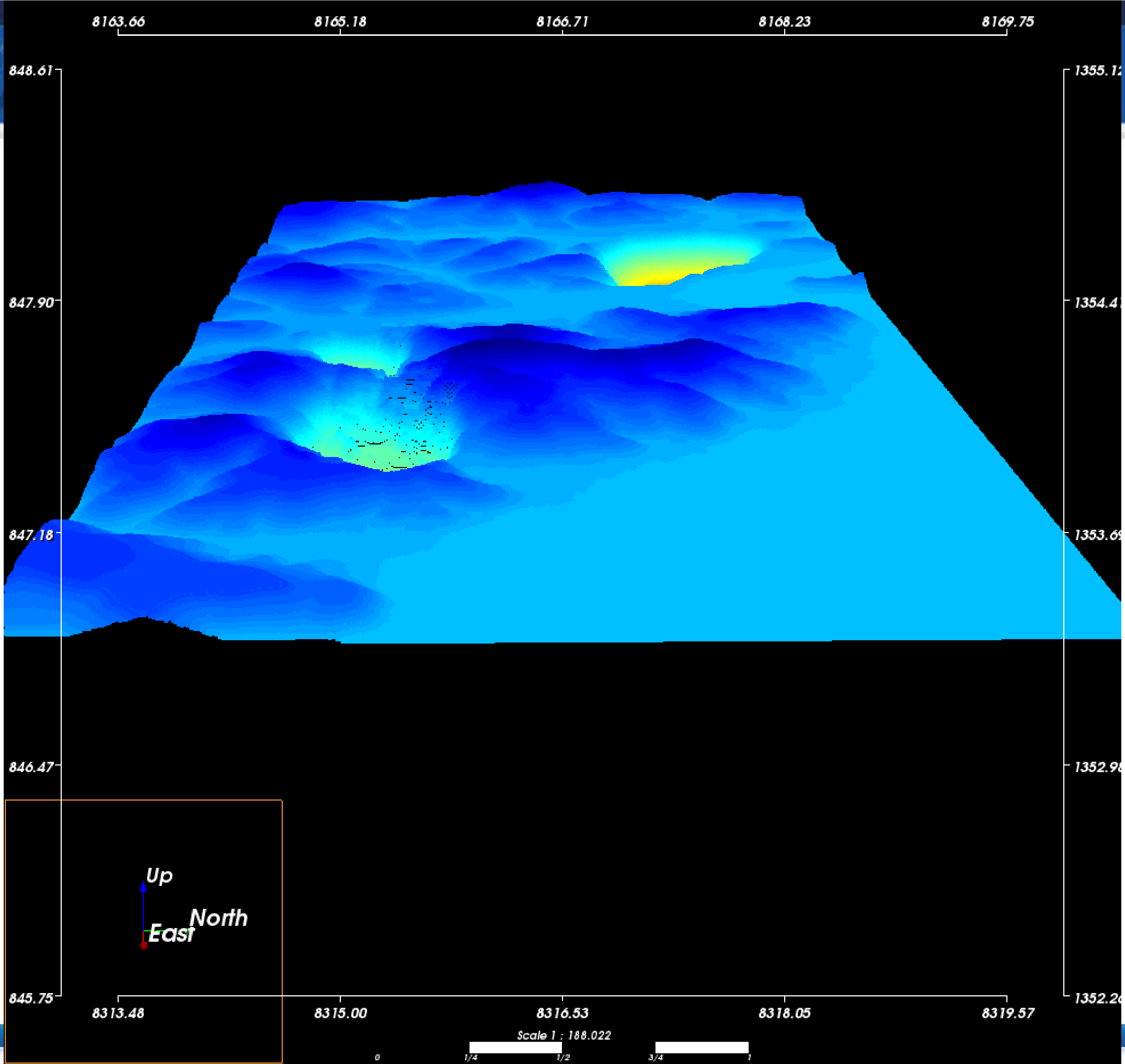


8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.12
1354.41
1353.69
1352.98
1352.26
Up
North
East
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
0
1/4
1/2
3/4
1

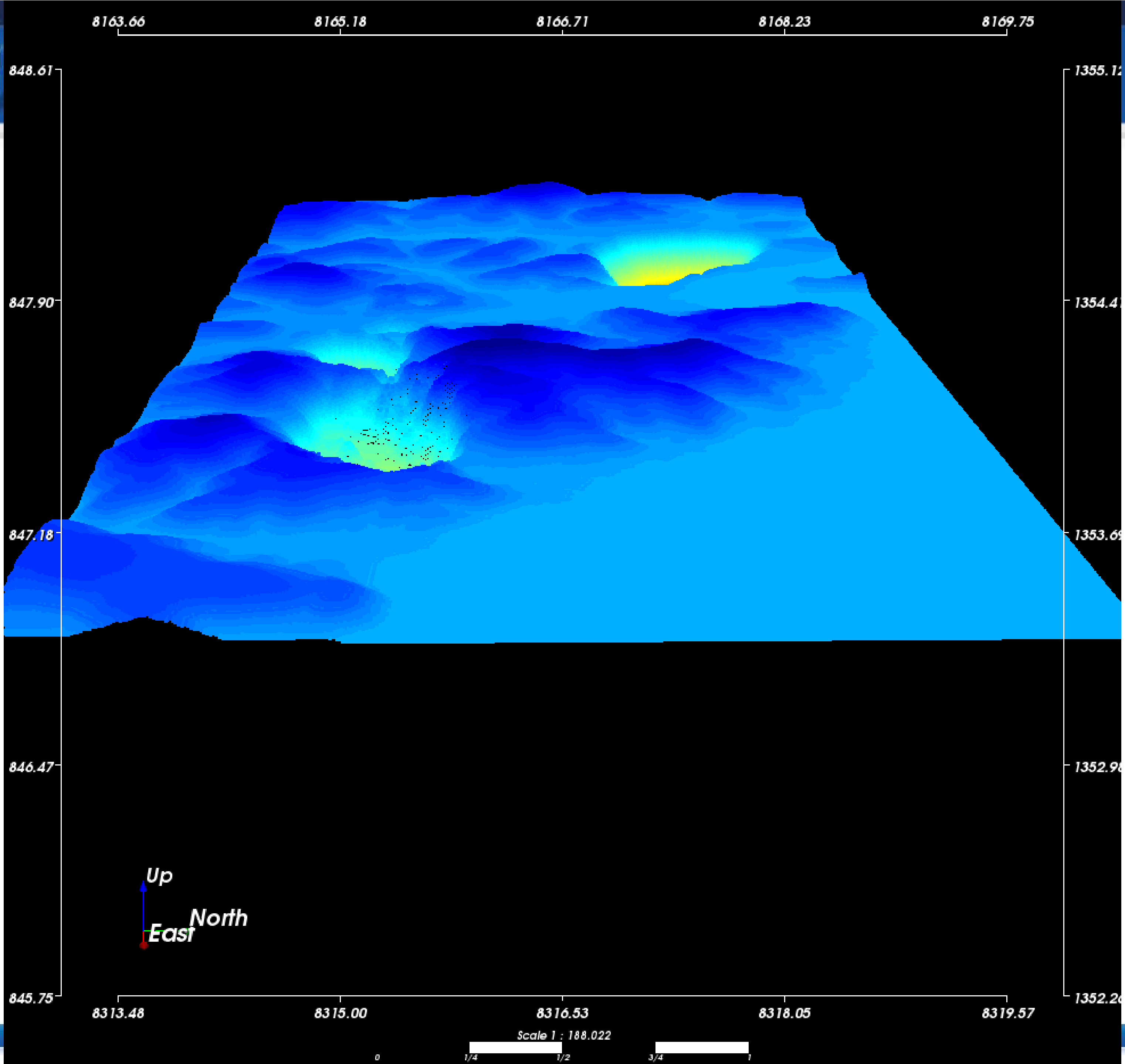

8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
Up
North
East
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
0
1/4
1/2
3/4
1

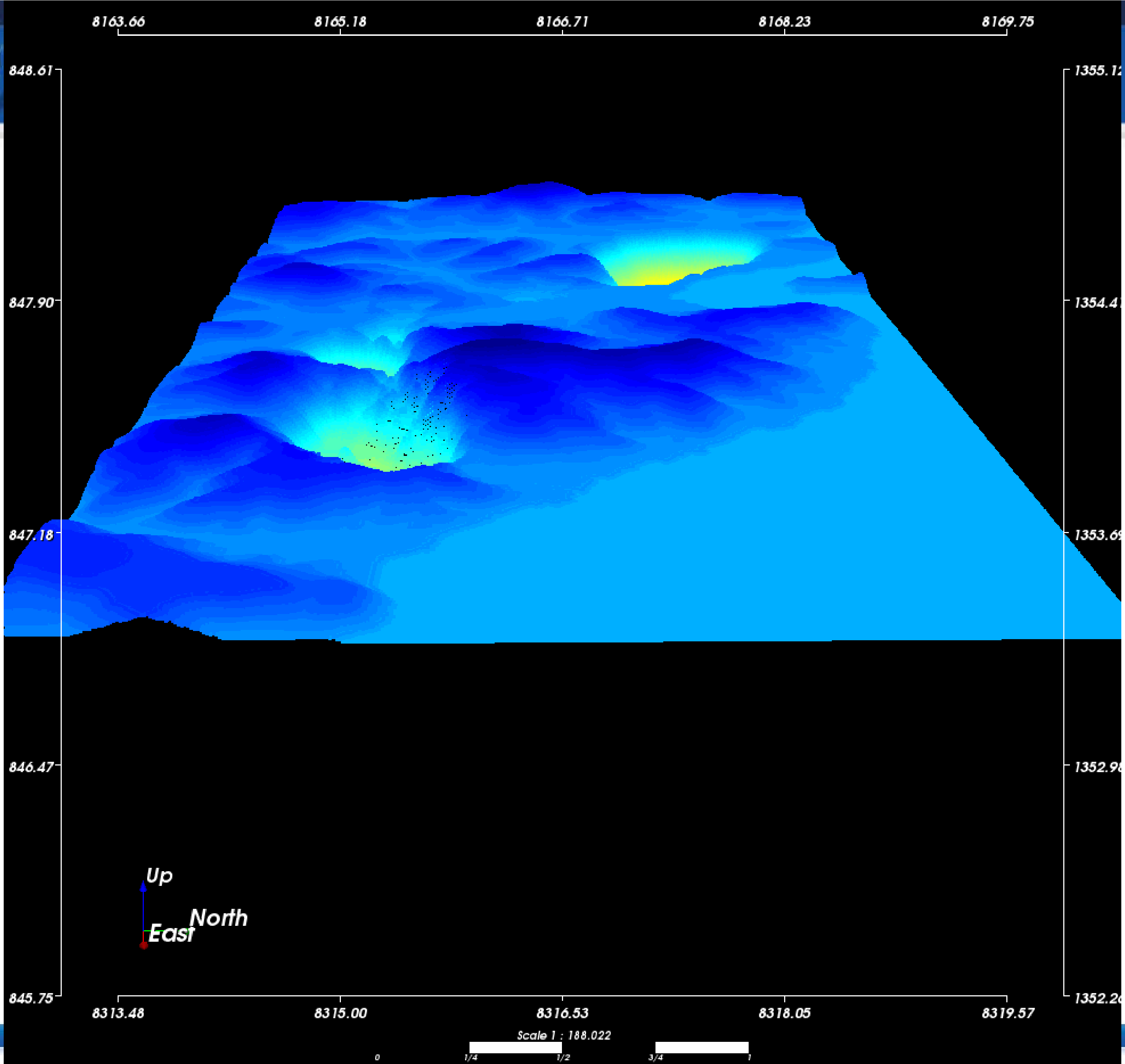


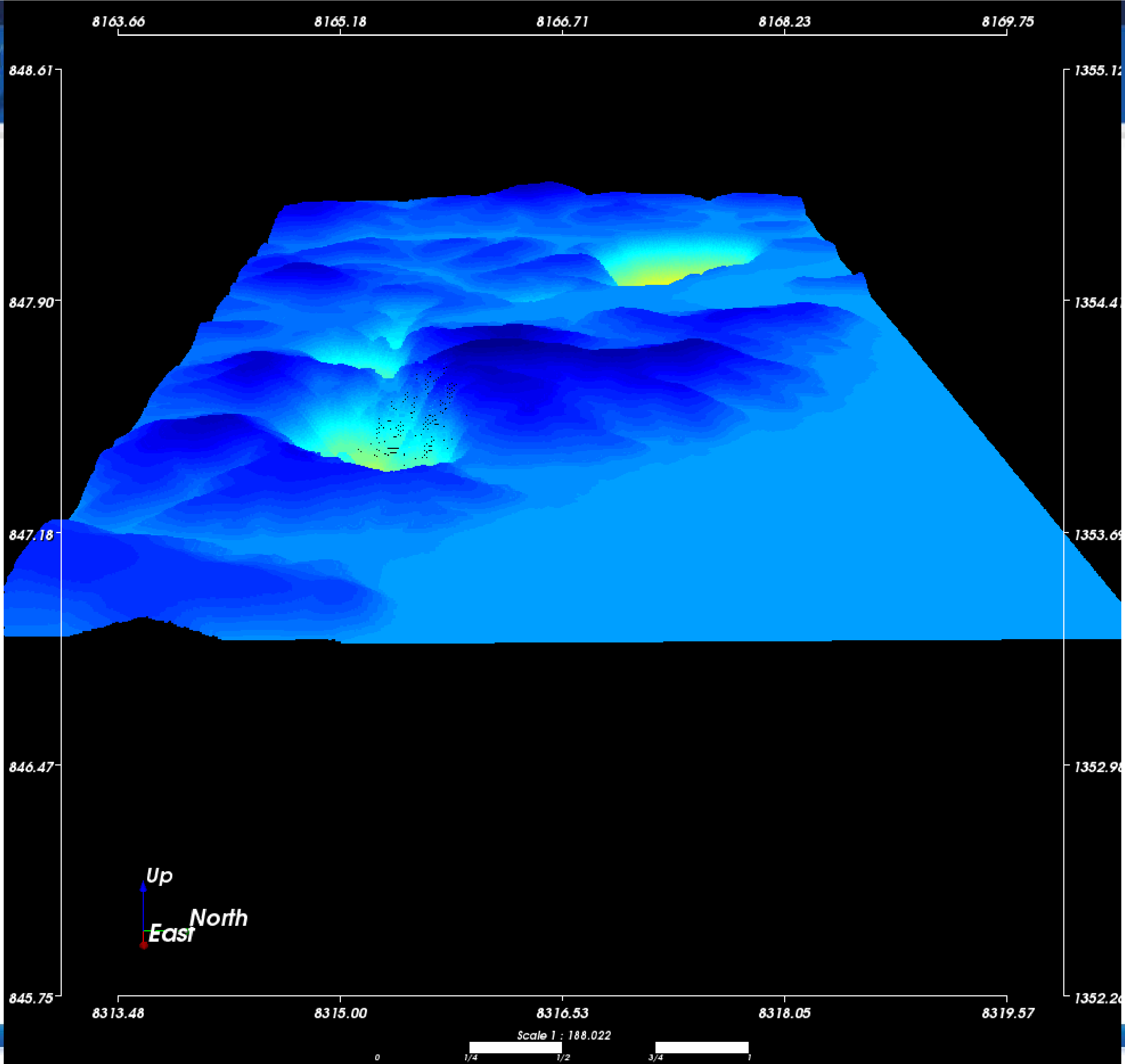
8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.12
1354.41
1353.69
1352.98
1352.26
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
0
1/4
1/2
3/4
1
Up
North
East



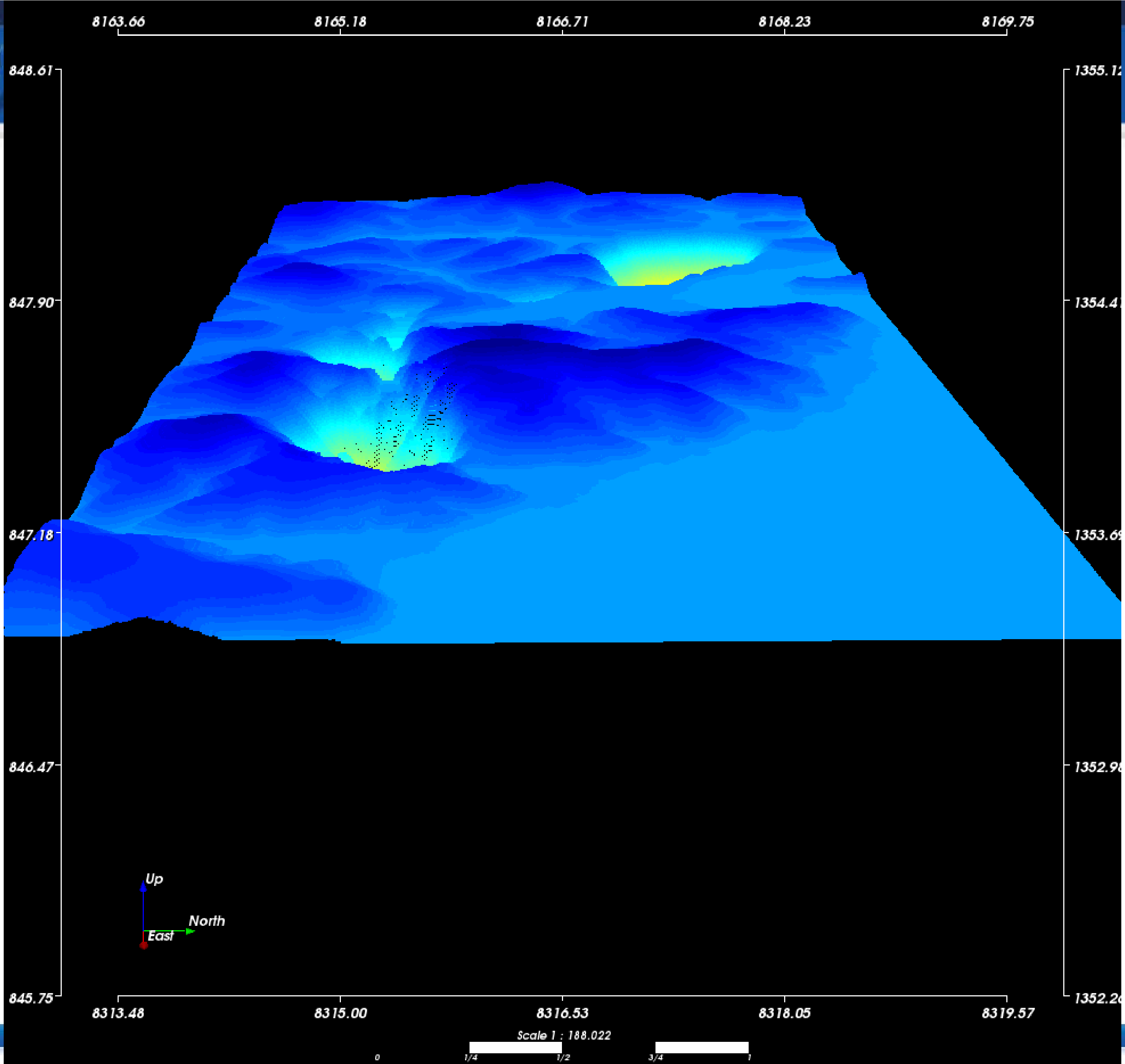
8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
8313.48
8315.00
8316.53
8318.05
8319.57
Up
North
East
Scale 1 : 188.022
0
1/4
1/2
3/4
1


8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
Up
North
East
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022

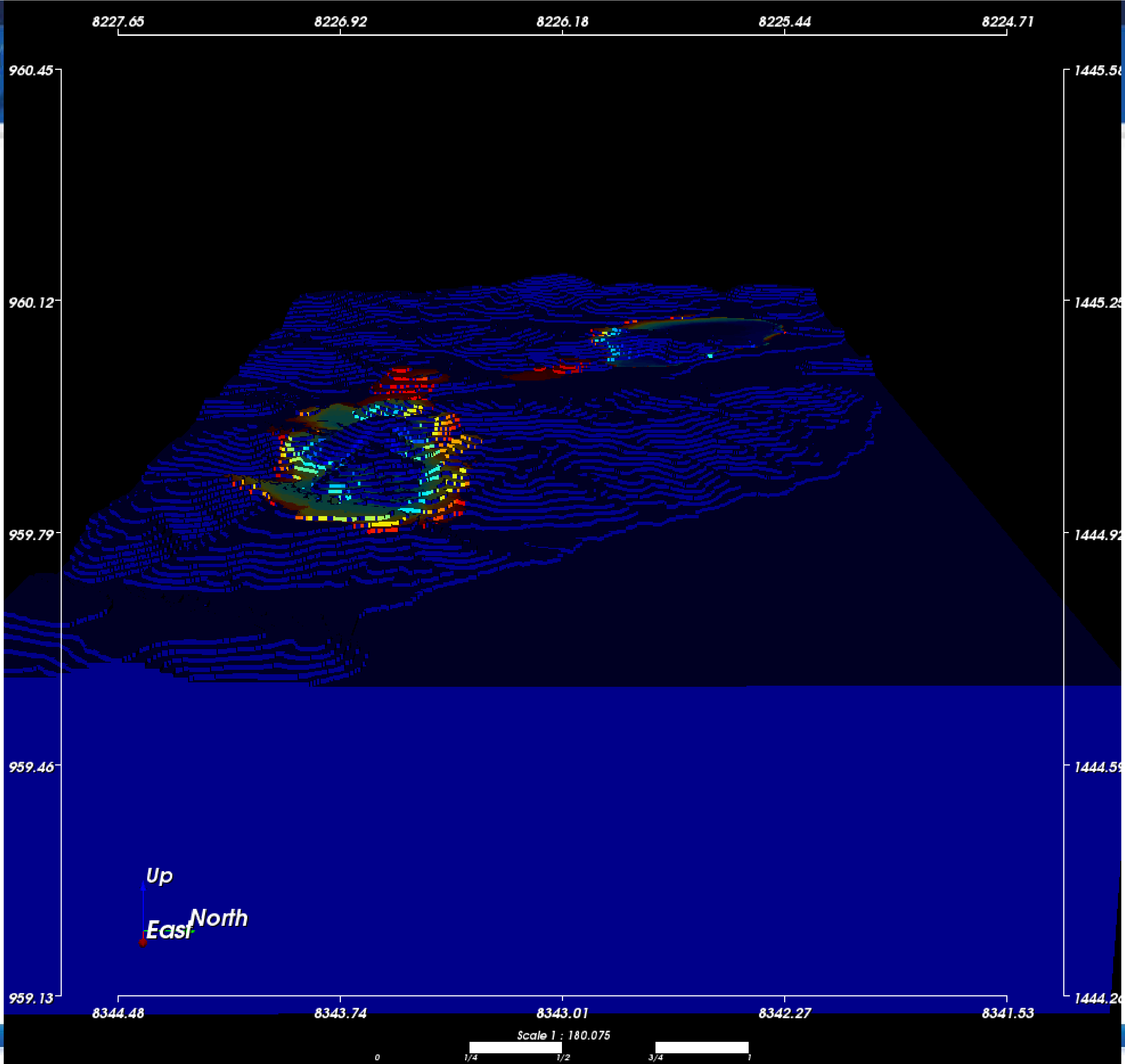

8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.12
1354.41
1353.69
1352.98
1352.26
Up
North
East
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
0
1/4
1/2
3/4
1



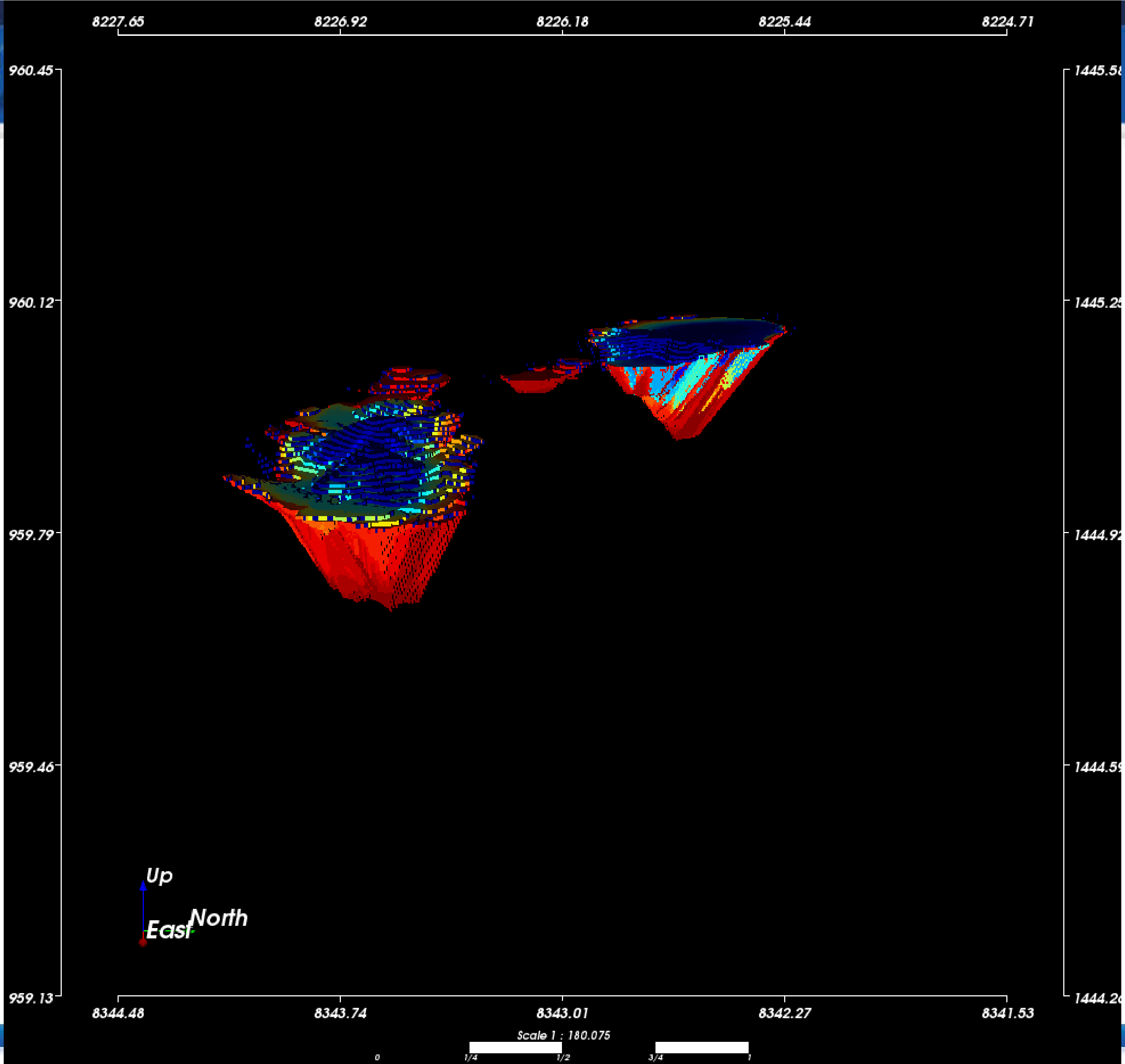
8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
Up
North
East
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022
0
1/4
1/2
3/4
1



8163.66
8165.18
8166.71
8168.23
8169.75
848.61
847.90
847.18
846.47
845.75
1355.12
1354.41
1353.69
1352.98
1352.26
Up
North
East
8313.48
8315.00
8316.53
8318.05
8319.57
Scale 1 : 188.022



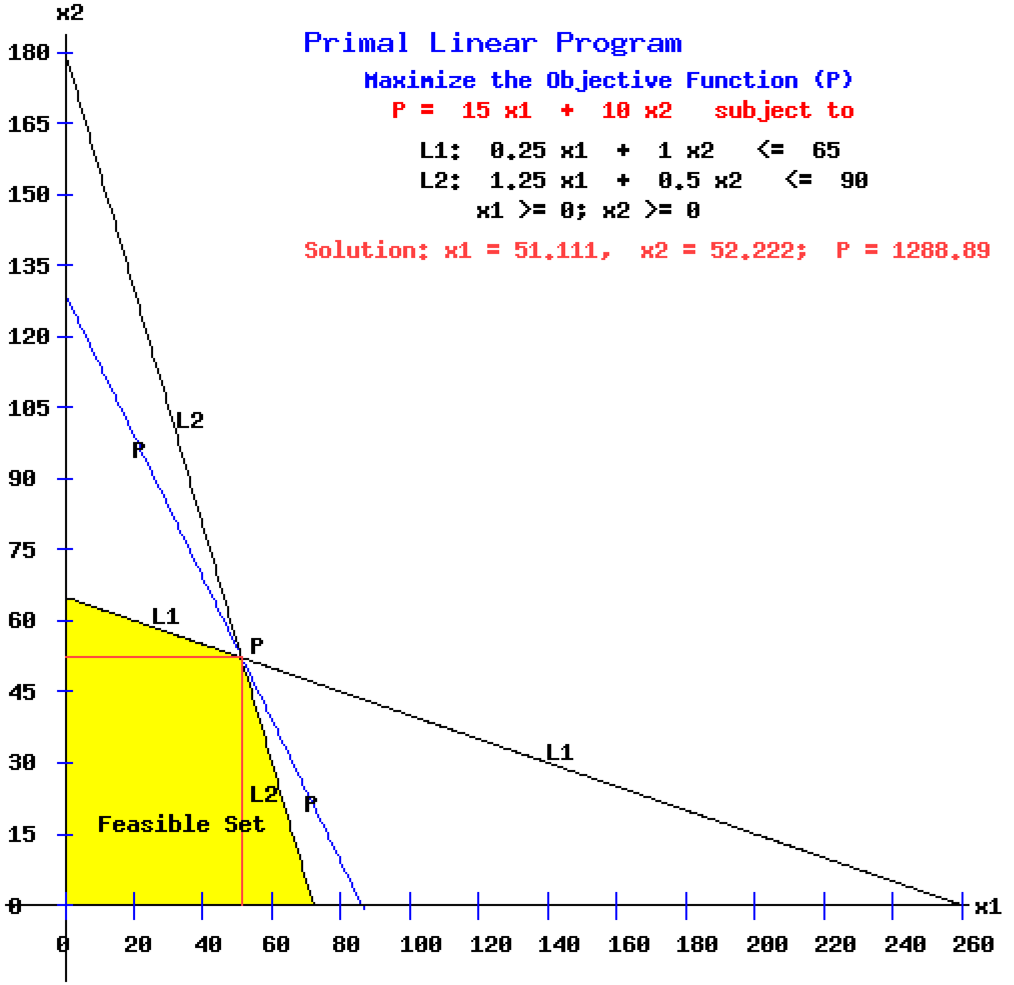
8227.65
8226.92
8226.18
8225.44
8224.71
960.45
960.12
959.79
959.46
959.13
1445.58
1445.25
1444.92
1444.59
1444.26
Up
East
North
8344.48
8343.74
8343.01
8342.27
8341.53
Scale 1 : 180.075
0
1/4
1/2
3/4
1



8227.65
8226.92
8226.18
8225.44
8224.71
960.45
960.12
959.79
959.46
959.13
Up
East
North
8344.48
8343.74
8343.01
8342.27
8341.53
Scale 1 : 180.075

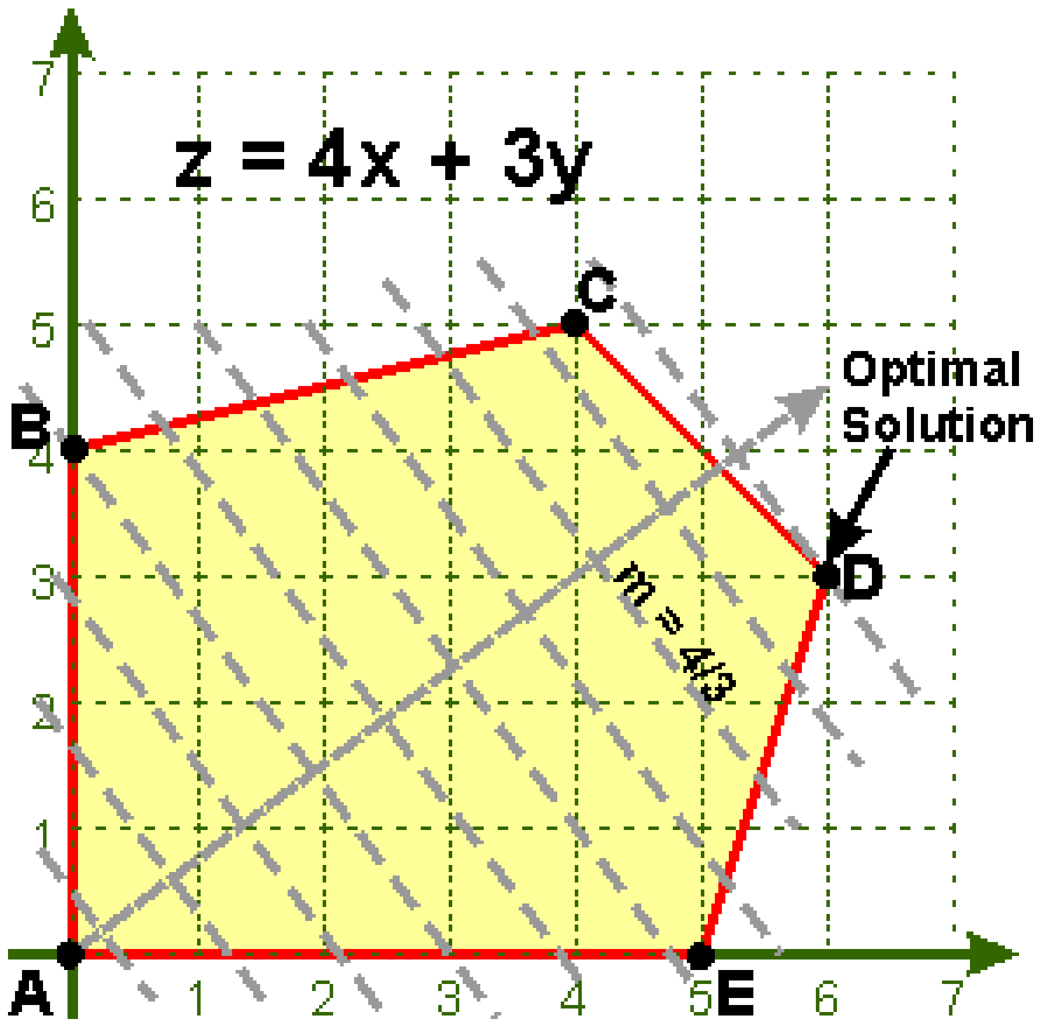
$$\begin{aligned} \textbf{max.} \quad & c^T x \\ \textbf{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

Simplex



x2
180
165
150
135
120
105
90
75
60
45
30
15
0
Primal Linear Program
Maximize the Objective Function (P)
P = 15 x1 + 10 x2 subject to
L1: 0.25 x1 + 1 x2 <= 65
L2: 1.25 x1 + 0.5 x2 <= 90
x1 >= 0; x2 >= 0
Solution: x1 = 51.111, x2 = 52.222; P = 1288.89
L2
P
L1
P
L1
L2
P
Feasible Set
x1
0
20
40
60
80
100
120
140
160
180
200
220
240
260

Interior Point



LINPACK AND MATRIX

Organized around matrix decompositions

LU	Cholesky
QR	singular value

Cholesky

- Small block model ~ 66k blocks
  - Sparse Cholesky 23000 x 23000 for one iteration
    - 529M elements
    - 207k non zero elements
  - Dense CPU single solution
    - Over 30 min
  - Sparse CPU single solution (not state of the art)
    - Few seconds
- CHOLMOD (suitesparse from NVIDIA)
  - 3x speedup over CPU multithread implementation

Cholesky

$$A = \begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{21} & a_{22} & a_{32} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

$$= \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{pmatrix} \equiv LL^T$$

$$= \begin{pmatrix} l_{11}^2 & l_{21}l_{11} & l_{31}l_{11} \\ l_{21}l_{11} & l_{21}^2 + l_{22}^2 & l_{31}l_{21} + l_{32}l_{22} \\ l_{31}l_{11} & l_{31}l_{21} + l_{32}l_{22} & l_{31}^2 + l_{32}^2 + l_{33}^2 \end{pmatrix}$$

$$l_{kk} = \sqrt{a_{kk} - \sum_{j=1}^{k-1} l_{kj}^2}$$

$$l_{ik} = \frac{1}{l_{kk}} \left(a_{ik} - \sum_{j=1}^{k-1} l_{ij} l_{kj} \right)$$

Cholesky

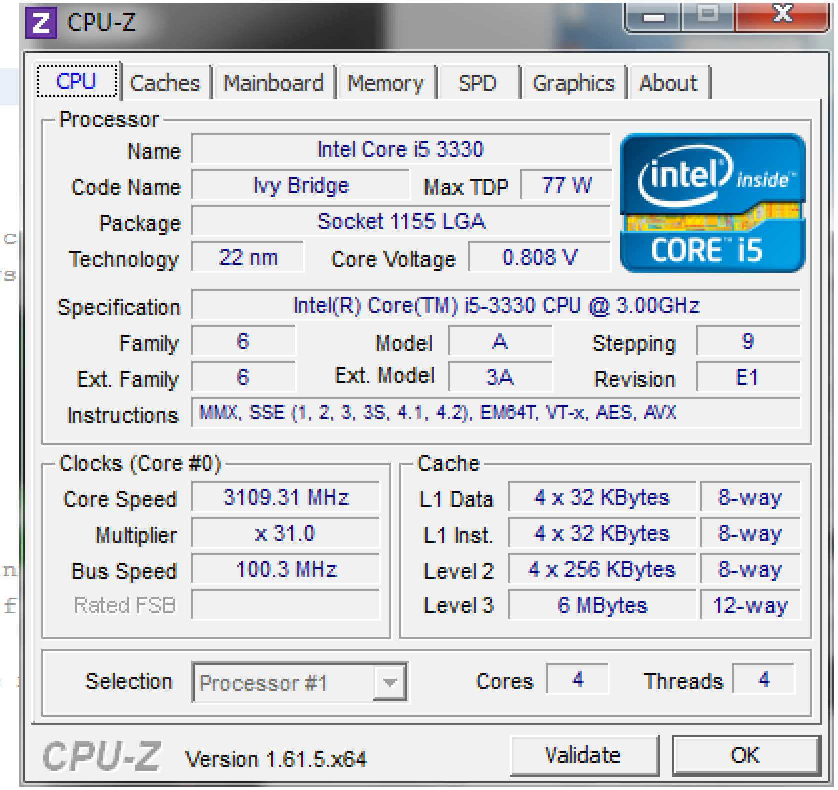
```
procedure CHOLESKY(A)
int i, j, k;
for k := 0 to n − 1 do
    A[k, k] := √A[k, k];    /* Obtain the square root of the diagonal element. */
    for j := k + 1 to n − 1 do
        A[k, j] := A[k, j]/A[k, k];    /* The division step. */
    end for
    for i := k + 1 to n − 1 do
        for j := i to n − 1 do
            A[i, j] := A[i, j] - A[k, i] × A[k, j];    /* The elimination step. */
        end for
    end for
end for
```

Single Thread

```
int chol_gold(const Matrix A, Matrix U) {
    unsigned int i, j, k;
    unsigned int size = A.num_rows * A.num_columns;

    // Copy the contents of the A matrix into the working matrix U
    for (i = 0; i < size; i++)
        U.elements[i] = A.elements[i];

    // Perform the Cholesky decomposition in place on the U matrix
    for (k = 0; k < U.num_rows; k++) {
        // Take the square root of the diagonal element
        U.elements[k * U.num_rows + k] = sqrt(U.elements[k * U.num_rows + k]);
        if (U.elements[k * U.num_rows + k] <= 0) {
            printf("Cholesky decomposition failed. \n");
            return 0;
        }

        // Division step
        for (j = (k + 1); j < U.num_rows; j++)
            U.elements[k * U.num_rows + j] /= U.elements[k * U.num_rows + k]; // Division step

        // Elimination step
        for (i = (k + 1); i < U.num_rows; i++)
            for (j = i; j < U.num_rows; j++)
                U.elements[i * U.num_rows + j] -= U.elements[k * U.num_rows + i] * U.elements[k * U.num_rows + j];
    }

    // As the final step, zero out the lower triangular portion of U
    for (i = 0; i < U.num_rows; i++)
        for (j = 0; j < i; j++)
            U.elements[i * U.num_rows + j] = 0.0;
    return 1;
}
```

pthread

```c
#ifndef _MATRIX_H_
#define _MATRIX_H_

// Thread block size
#define MATRIX_SIZE 2048
#define NUM_PTHREADS 4
#define NUM_OMPTHREADS  NUM_PTHREADS

// Matrix dimensions
#define NUM_COLUMNS MATRIX_SIZE // Number of c
#define NUM_ROWS MATRIX_SIZE // Number of rows

// Matrix Structure declaration
typedef struct {
        //width of the matrix represented
    unsigned int num_columns;
        //height of the matrix represented
    unsigned int num_rows;
        //number of elements between the begin
        // rows in the memory layout (useful f
    unsigned int pitch;
        //Pointer to the first element of the
    float* elements;
} Matrix;


#endif // _MATRIX_H_
```



pthread

```
// Perform the Cholesky decomposition in place on the U matrix
for(k = 0; k < U.num_rows; k++)
{
        //Only one thread does squre root and division
        if(id==0)
        {
                // Take the square root of the diagonal element
                U.elements[k * U.num_rows + k] = sqrt(U.elements[k * U.num_rows + k]);
                if(U.elements[k * U.num_rows + k] <= 0){
                                printf("Cholesky decomposition failed. \n");
                                return 0;
                }

                // Division step
                for(j = (k + 1); j < U.num_rows; j++)
                {
                        U.elements[k * U.num_rows + j] /= U.elements[k * U.num_rows + k]; // Division step
                }
        }

        //Sync threads!!!!!
        sync_pthreads(barrier, id);

        //For this k iteration, split up i
        //Size of i range originally
        int isize = U.num_rows - (k + 1);
        int items_per_thread, items_last_thread;
        range_splitter(isize, NUM_PTHREADS, &items_per_thread, &items_last_thread);
        //Divy up work
        //Elim work
        int elimi_start, elimi_end;
        int offset = (k + 1); //To account for not starting at i=0 each time
```

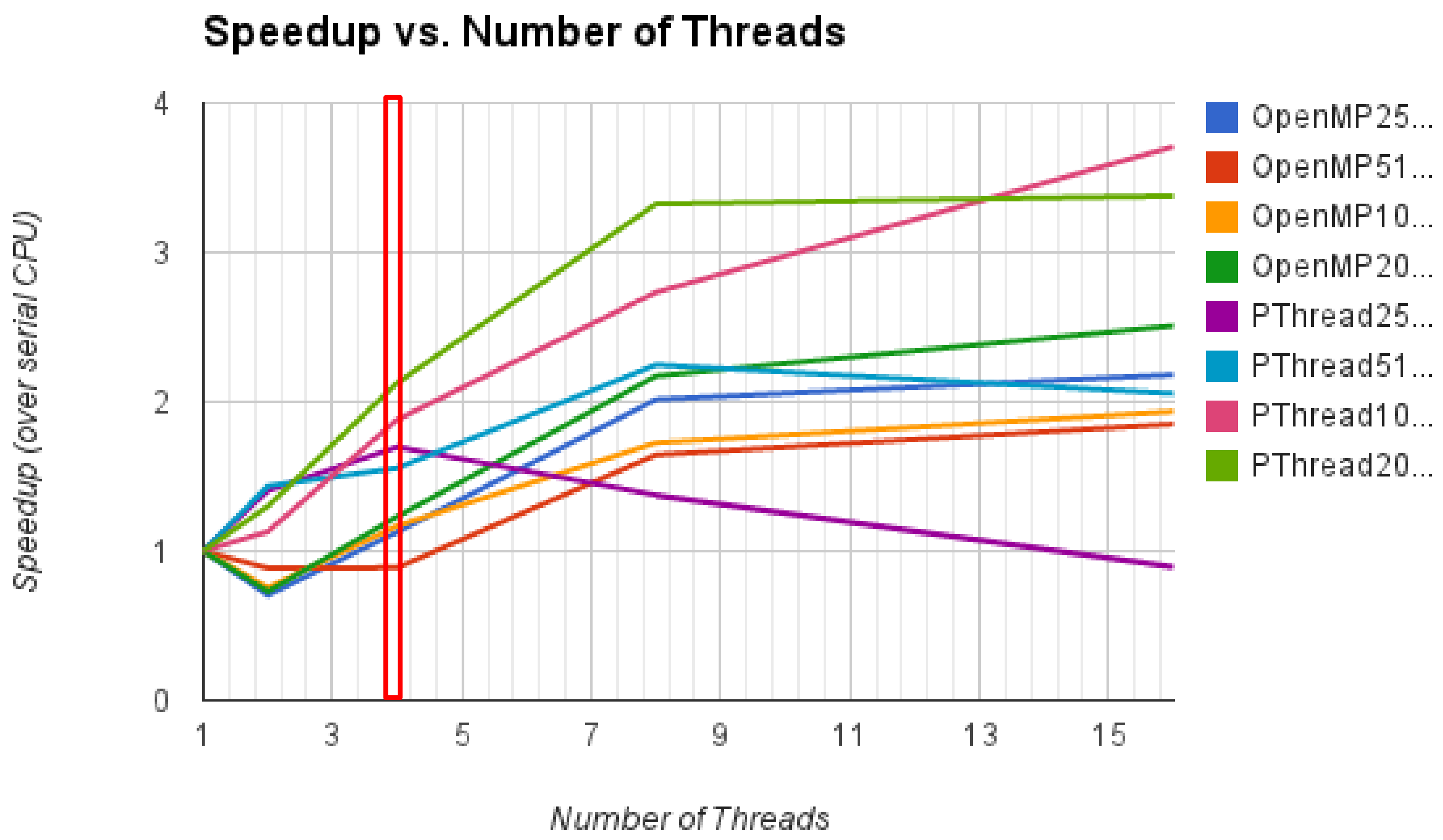
pthread

```
// Perform the Cholesky decomposition in place on the U matrix
for(k = 0; k < U.num_rows; k++)
{
        //Only one thread does squre root and division
        if(id==0)
        {
                // Take the square root of the diagonal element
                U.elements[k * U.num_rows + k] = sqrt(U.elements[k * U.num_rows + k]);
                if(U.elements[k * U.num_rows + k] <= 0){
                                printf("Cholesky decomposition failed. \n");
                                return 0;
                }

                // Division step
                for(j = (k + 1); j < U.num_rows; j++)
                {
                        U.elements[k * U.num_rows + j] /= U.elements[k * U.num_rows + k]; // Division step
                }
        }

        //Sync threads!!!!!
        sync_pthreads(barrier, id);

        //For this k iteration, split up i
        //Size of i range originally
        int isize = U.num_rows - (k + 1);
        int items_per_thread, items_last_thread;
        range_splitter(isize, NUM_PTHREADS, &items_per_thread, &items_last_thread);
        //Divy up work
        //Elim work
        int elimi_start, elimi_end;
        int offset = (k + 1); //To account for not starting at i=0 each time
```

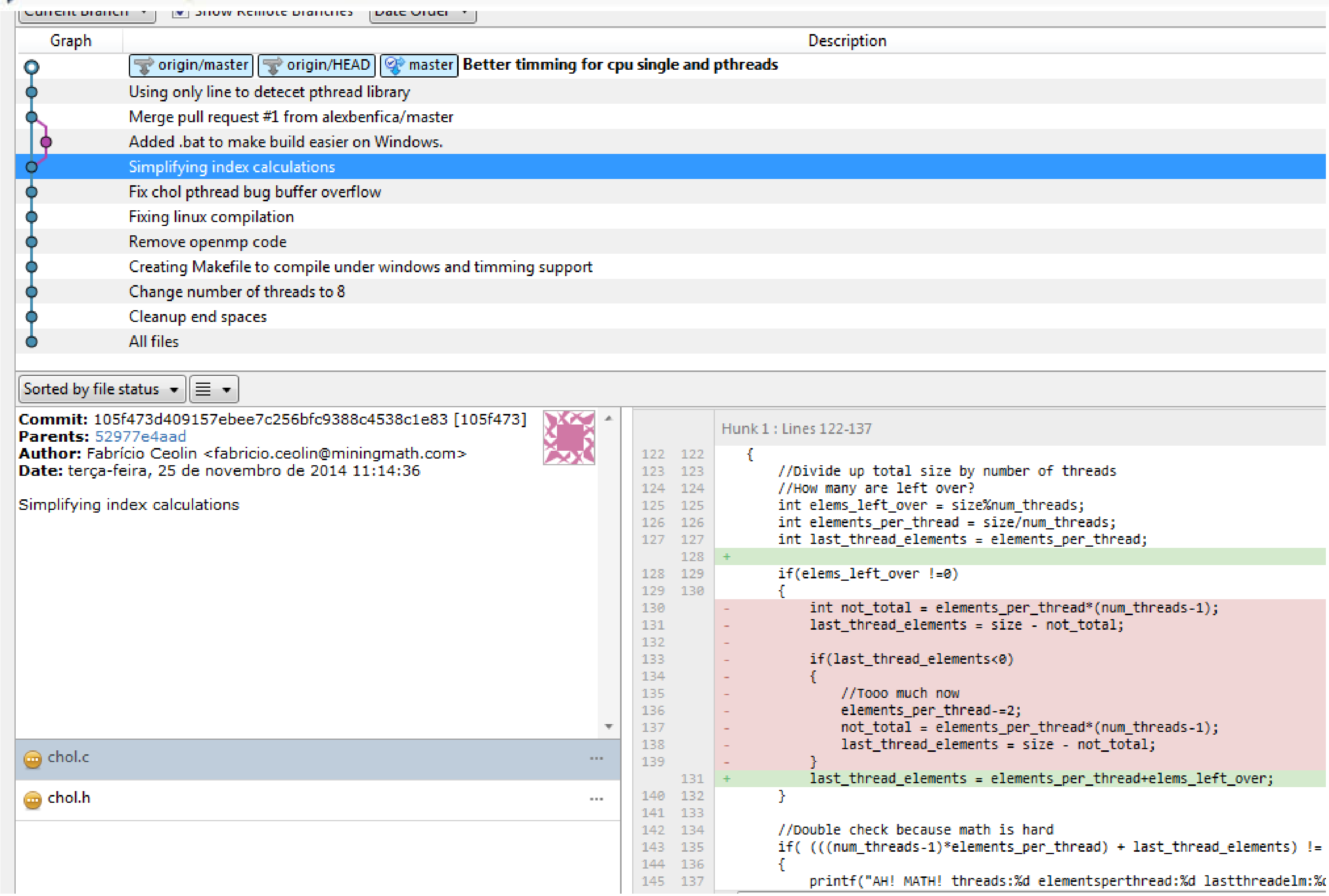
pthread

```c
// Perform the Cholesky decomposition in place on the U matrix
for(k = 0; k < U.num_rows; k++)
{
    //Only one thread does squre root and division
    if(id==0)
    {
        // Take the square root of the diagonal element
        U.elements[k * U.num_rows + k] = sqrt(U.elements[k * U.num_rows + k]);
        if(U.elements[k * U.num_rows + k] <= 0){
                printf("Cholesky decomposition failed. \n");
                return 0;
        }

        // Division step
        for(j = (k + 1); j < U.num_rows; j++)
        {
            U.elements[k * U.num_rows + j] /= U.elements[k * U.num_rows + k]; // Division step
        }
    }

    //Sync threads!!!!!
    sync_pthreads(barrier, id);

    //For this k iteration, split up i
    //Size of i range originally
    int isize = U.num_rows - (k + 1);
    int items_per_thread, items_last_thread;
    range_splitter(isize, NUM_PTHREADS, &items_per_thread, &items_last_thread);
    //Divy up work
    //Elim work
    int elimi_start, elimi_end;
    int offset = (k + 1); //To account for not starting at i=0 each time
```

Pthread: before optimizations



Pthread: our optimizations



Pthread: our optimizations

```
D:\Dropbox\Devel\Win\CudaProject\Drexel-ECEC622-Midterm1\build>CholThread.exe
Creating a 2048 x 2048 matrix with random numbers between [-.5, .5]...done.
Generating the symmetric matrix...done.
Generating the positive definite matrix...done.
Performing Cholesky decomposition on the CPU using the single-threaded versio

        Run time:    9.4130001068 s.
Cholesky decomposition on the CPU was successful.

Performing Cholesky decomposition on the CPU using the PTHREAD version.
        Run time:    3.0450000763 s.
Speedup from single to phreads = 3.091297 x.
Double checking for correctness by recovering the original matrix.
```

CUDA device query: Geforce GTS 450

```
 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTS 450"
  CUDA Driver Version / Runtime Version          6.5 / 6.5
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 1024 MBytes (1073741824 bytes)
  ( 4) Multiprocessors, ( 48) CUDA Cores/MP:     192 CUDA Cores
  GPU Clock rate:                                1620 MHz (1.62 GHz)
  Memory Clock rate:                             1804 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 262144 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65535), 3D=(2048, 2048, 2048)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (65535, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 6.5, CUDA Runtime Version = 6.5, NumDevs = 1, Device0 = GeForce GTS 450
Result = PASS
```

CUDA device query: Geforce GTS 450

```
 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTS 450"
  CUDA Driver Version / Runtime Version          6.5 / 6.5
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 1024 MBytes (1073741824 bytes)
  ( 4) Multiprocessors, ( 48) CUDA Cores/MP:     192 CUDA Cores
  GPU Clock rate:                                1620 MHz (1.62 GHz)
  Memory Clock rate:                             1804 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 262144 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65535), 3D=(2048, 2048, 2048)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (65535, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 6.5, CUDA Runtime Version = 6.5, NumDevs = 1, Device0 = GeForce GTS 450
Result = PASS
```

CUDA sqrt

```
int threads_per_block_sqrt = 512;
int blocks_sqrt = MATRIX_SIZE / threads_per_block_sqrt;
dim3 thread_block(threads_per_block_sqrt, 1, 1);
dim3 grid(blocks_sqrt, 1);
chol_kernel_cudaUFMG_sqrt <<<grid, thread_block>>>(gpu_u.elements);
```

CUDA sqrt kernel

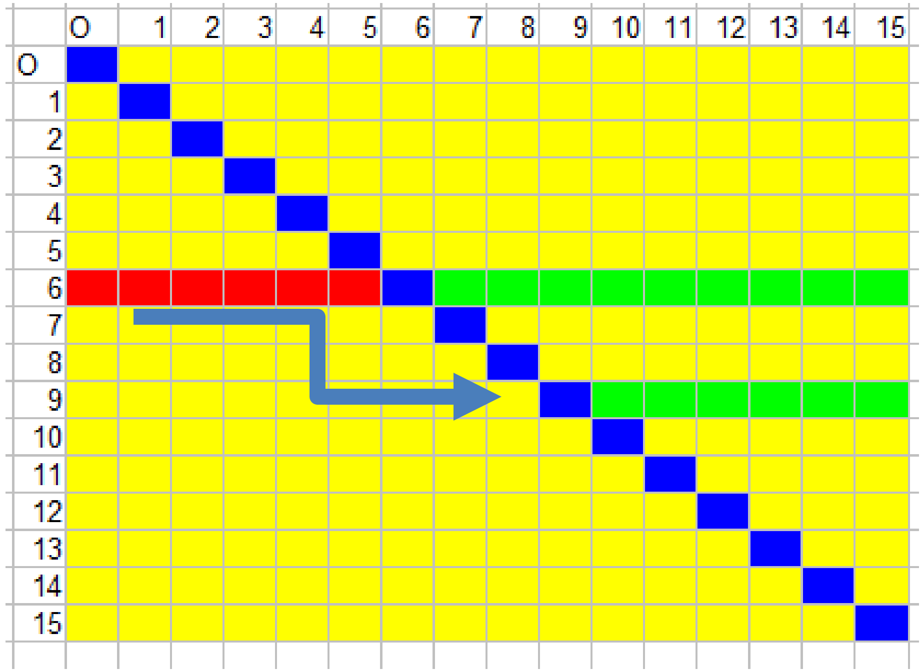
```
__global__ void chol_kernel_cudaUFMG_sqrt(float * U) {
    // Get a thread identifier
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int tx_diag = tx * MATRIX_SIZE + tx;
    U[tx_diag] = sqrt(U[tx_diag]);
}
```

CUDA div

```
int block_x_div = 16;
int block_y_div = 16;
int thread_x_div = 4;
int thread_y_div = 4;
dim3 grid_div(block_x_div, block_y_div, 1);
dim3 thread_block_div(thread_x_div, thread_y_div, 1);
int elements_per_thread_div = ((MATRIX_SIZE * MATRIX_SIZE) / 2) /  (thread_x_div * thread_y_div * block_x_div * block_y_div);
chol_kernel_cudaUFMG_division <<<grid_div, thread_block_div >>>(gpu_u.elements, elements_per_thread_div);
```

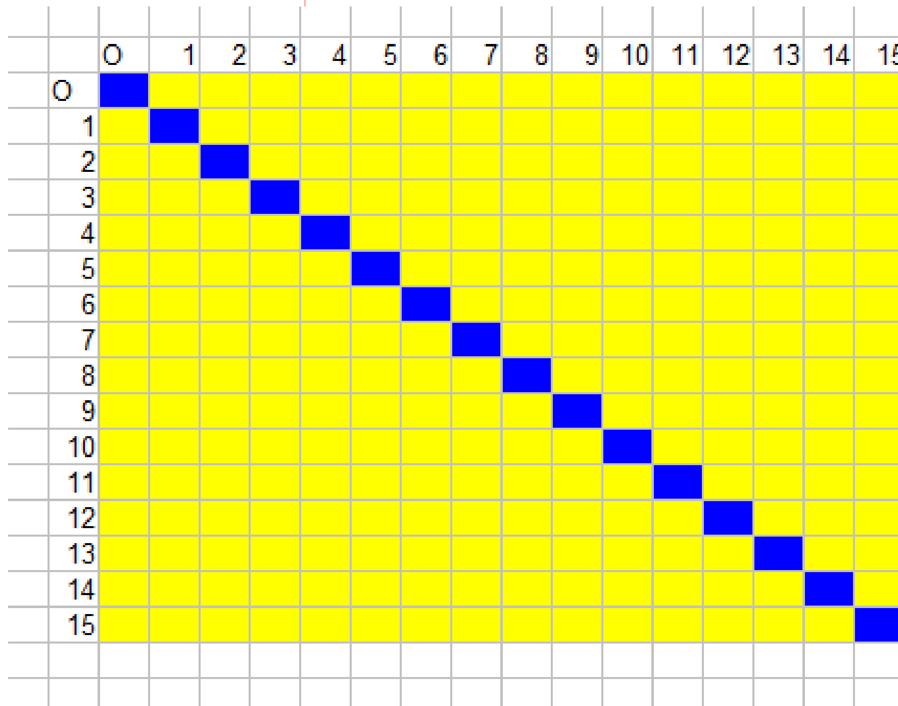
CUDA div kernel

```
__global__ void chol_kernel_cudaUFMG_division(float * U, int elem_per_thr) {
    // Get a thread identifier
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int ty = blockIdx.y * blockDim.y + threadIdx.y;

    int tn = ty * blockDim.x * gridDim.x + tx;

    for(unsigned i=0;i<elem_per_thr;i++){
        int iel = tn * elem_per_thr + i;
        int xval = iel % MATRIX_SIZE;
        int yval = iel / MATRIX_SIZE;

        if(xval == yval){
            continue;
        }

        // if on the lower diagonal...
        if(yval > xval){
            xval = MATRIX_SIZE - xval - 1;
            yval = MATRIX_SIZE - yval - 1;
        }

        int iU = xval + yval * MATRIX_SIZE;
        int iDiag = yval + yval * MATRIX_SIZE;

        U[iU] /= U[iDiag];
    }
}
```



CUDA elimination

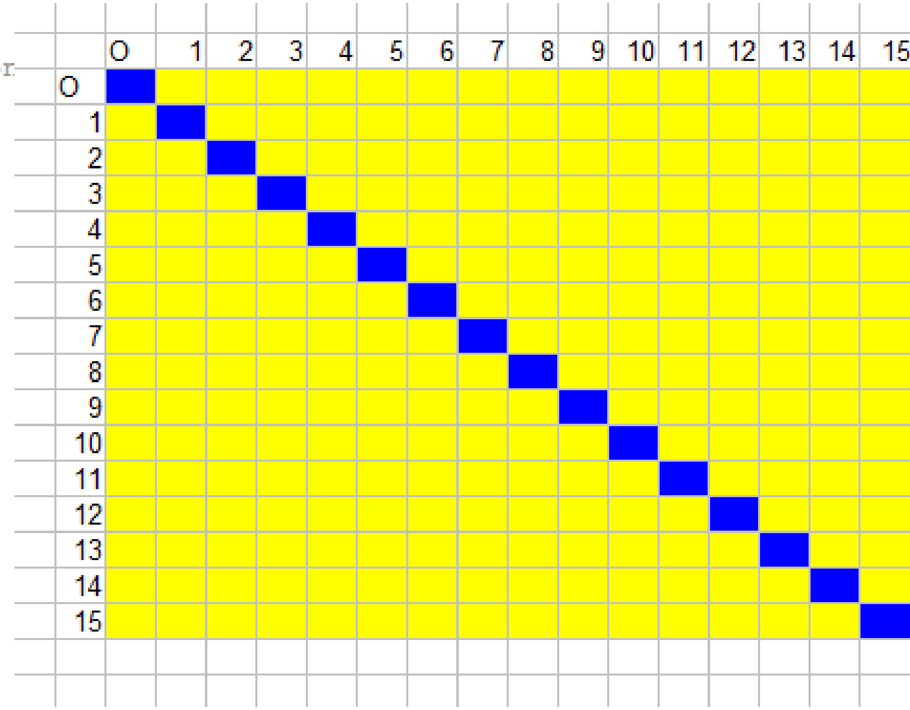
```
int block_y_eli = 1;
//Each thread within a block will take some j iterations
int thread_x_eli = 256;
int thread_y_eli = 1;

//Each kernel call will be one iteration of out K loop
for (int k = 0; k < MATRIX_SIZE; k++) {

    //Want threads to stride across memory
    //i is outer loop
    //j is inner loop
    //so threads should split the j loop
    //Each thread block will take an i iteration

    // i=k+1;i<MATRIX_SIZE
    int isize = MATRIX_SIZE - (k + 1);
    if(isize==0){
        isize++;
    }
    int block_x_eli = isize;

    //Set up the execution grid on the GPU
    dim3 thread_block(thread_x_eli, 1, 1);
    dim3 grid(block_x_eli, 1);

    //Call kernel with for this K iteration
    chol_kernel_cudaUFMG_elimination <<<grid, thread_block>>>(gpu_u.elements, k);
}
```

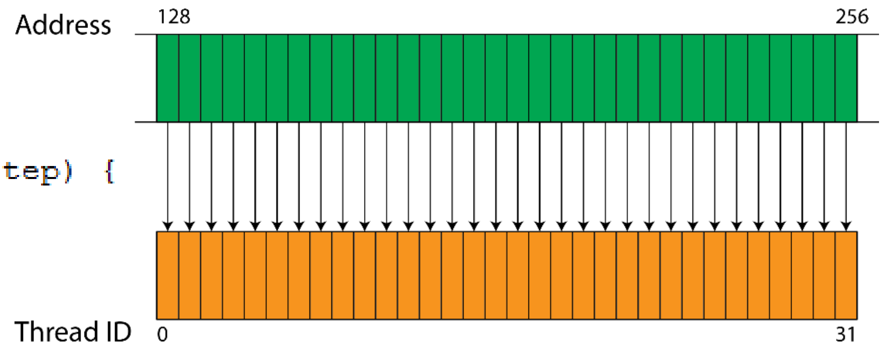



```
chol_kernel_cudaUFMG_zero <<<grid_div, thread_block_div>>>(gpu_u.elements, elements_per_thread_div);
```

CUDA elimination kernel

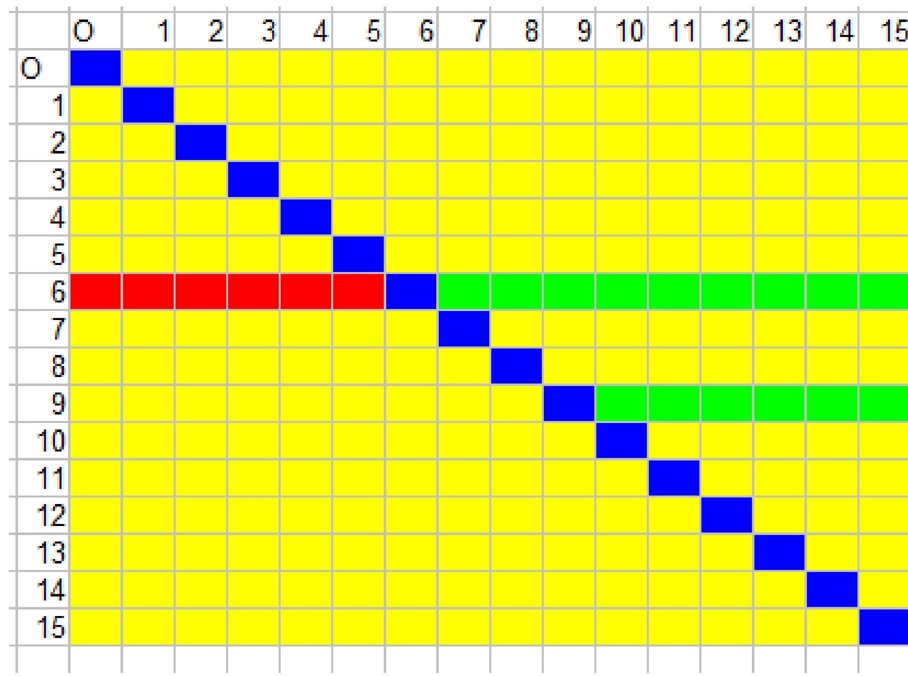
```
__global__ void chol_kernel_cudaUFMG_elimination(float * U, int k) {

    //This call acts as a single K iteration
    //Each block does a single i iteration
    int i = (k+1) + blockIdx.x;

    //Each thread does some part of j
    //Stide in units of 'stride'
    //Thread 0 does 0, 16, 32
    //Thread 1 does 1, 17, 33
    //..etc.
    int jstart = i + threadIdx.x;
    int jstep = blockDim.x;

    // Pre-calculate indexes outside loop
    int kM = k * MATRIX_SIZE;
    int iM = i * MATRIX_SIZE;
    int ki = kM + i;

    //Do work for this i iteration
    for (int j=jstart; j<MATRIX_SIZE; j+=jstep) {
        U[iM + j] -= U[ki] * U[kM + j];
    }
}
```





CUDA zero kernel

```
__global__ void chol_kernel_cudaUFMG_zero(float * U, int elem_per_thr) {
    // Get a thread identifier
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int ty = blockIdx.y * blockDim.y + threadIdx.y;

    int tn = ty * blockDim.x * gridDim.x + tx;

    for(unsigned i=0;i<elem_per_thr;i++){
        int iel = tn * elem_per_thr + i;
        int xval = iel % MATRIX_SIZE;
        int yval = iel / MATRIX_SIZE;

        if(xval == yval){
            continue;
        }

        // if on the upper diagonal...
        if(yval < xval){
            xval = MATRIX_SIZE - xval - 1;
            yval = MATRIX_SIZE - yval - 1;
        }
        int iU = xval + yval * MATRIX_SIZE;
        U[iU] = 0;
    }

}
```



SPEEDUP Results

TYPE	SINGLE	PTHREAD	OUR PTHREAD	CUDA	OUR CUDA
SINGLE	1.00				
PTHREAD	2.20	1.00			
OUR PTHREAD	3.10	**1.41**	1.00		
CUDA	24.00	10.91	7.74	1.00	
OUR CUDA	33.00	15.00	10.65	**1.38**	1.00

Conclusion

- Improvements
  - It is possible to speedup using our algorithm with CUDA
  - No room for shared memory in our implementation
    - (We’ve tried other methods, without success)

Conclusion

- Future work
  - Pthread
    - Split sqrt and division over threads
  - Study Sparse Matrix algorithm
    - suitesparse with metis
    - Test CUDA and CPU multithread

Questions?

Fabrício Ceolin e Alex Teixeira