Advanced Topics in CUDA C

S. Sundar and M. Panchatcharam

August 9, 2014

Outline

Julia Set

- The following example will demonstrate code to draw slices of the Julia set
- Julia set is the boundary of certain class of functions over complex numbers
- A fun example of fractal
- Julia set evaluates a simple iterative equation for point in the complex plane
- A point is not in the set if the process of iterating the equation diverges for that point
- If a sequence of values produced by iterating the equation grows toward infinity, a point is considered outside the set
- If the values taken by the equation remain bounded, the point is in the set

Julia set

$$Z_{n+1} = Z_n^2 + C \tag{1}$$

It involves squaring the current value and adding a constant to get the next value of the equation

- Since it is more complicated program, we will split it into pieces
- Note: You have to use some of the header files, which is not discussed in detail here

Julia main function

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *ptr = bitmap.get_ptr();

    kernel( ptr );

    bitmap.display_and_exit();
}
```

The main function is so simple

Julia main function

```
void kernel( unsigned char *ptr ){
    for (int y=0; y<DIM; y++) {
        for (int x=0; x<DIM; x++) {
            int offset = x + y * DIM;

            int juliaValue = julia( x, y );
            ptr[offset*4 + 0] = 255 * juliaValue;
            ptr[offset*4 + 1] = 0;
            ptr[offset*4 + 2] = 0;
            ptr[offset*4 + 3] = 255;
        }
    }
}
```

Julia main function

```
int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

Julia main function

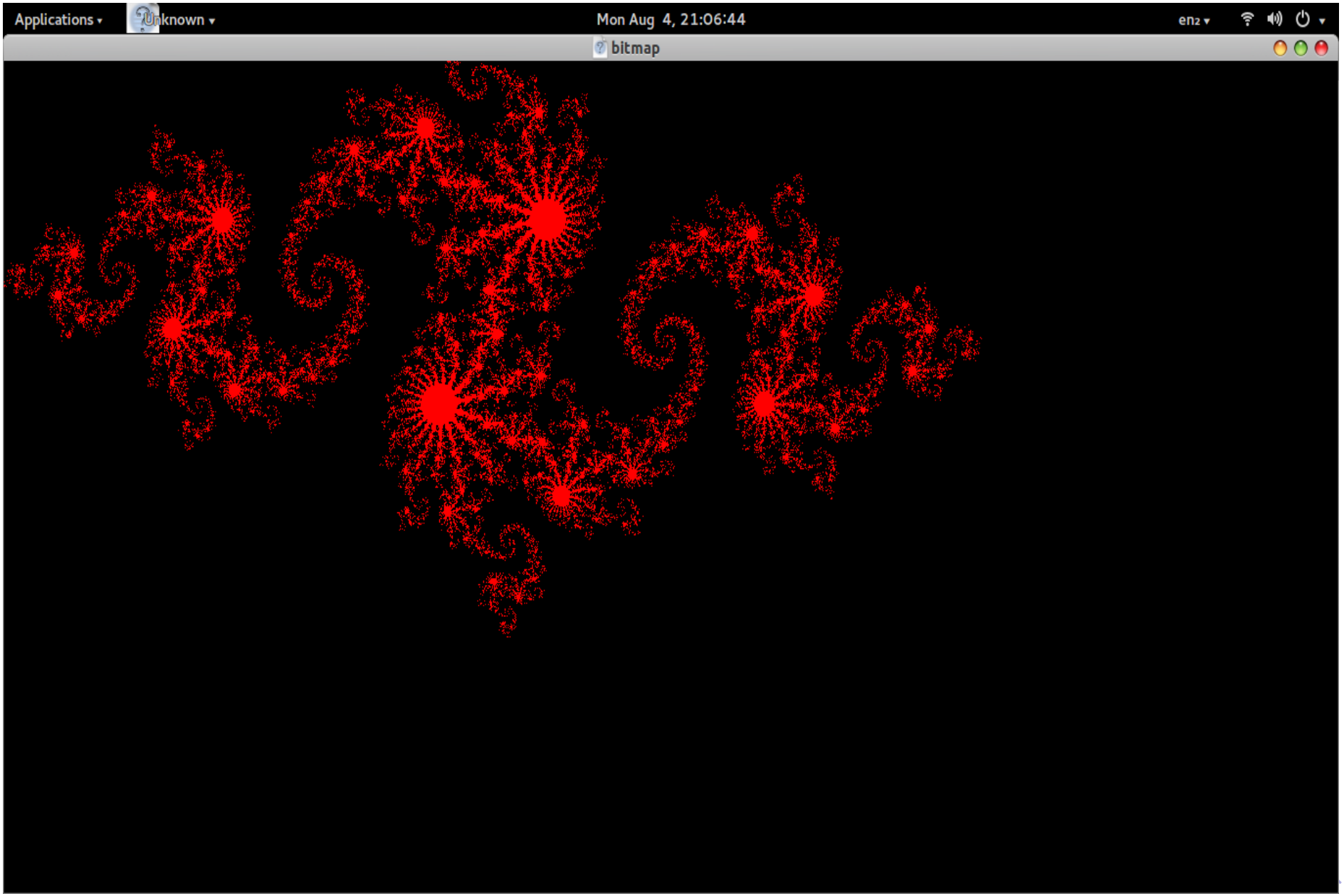
```
#include "book.h"
#include "cpu_bitmap.h"

#define DIM 1000

struct cuComplex {
    float   r;
    float   i;
    cuComplex( float a, float b ) : r(a), i(b)  {}
    float magnitude2( void ) { return r * r + i * i; }
    cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
```

- The kernel computation iterates through all points we care to render, calling julia()
- The `julia()` will return 1 if the point is in the set, 0 if it is not in the set
- The point color is 1 if `julia()` returns 1 and black if it returns 0
- These two colors are arbitrary and you can use any color

- This `julia()` is the main part of this example
- Translated our pixel coordinate to a coordinate in complex space
- To center the complex plane at the image center, we shift by DIM/2.
- To ensure that the image spans the range of -1.0 to 1.0, we scale the image coordinate by DIM/2
- Given an image point at (x,y), we get a point in complex space at ((DIM/2 – x)/(DIM/2),((DIM/2 – y)/(DIM/2))

- To zoom in or out, introduce a scale factor 1.5,
- But you should tweak this parameter to zoom in or out
- After obtaining the point in complex space, we then need to determine whether the point is in or out of the Julia Set
- Remember $Z_{n+1} = Z_n^2 + C$
- Since C is some arbitrary complex-valued constant, we have chosen -0.8 + 0.156i because it happens to yield an interesting picture
- Play with this constant if you want to see other versions of the Julia Set

Memory



- Since all the computations are being performed on complex numbers, we define a generic structure to store complex numbers
- The class represents complex numbers with two data elements: a single-precision real component r and a single-precision imaginary component i
- The class defines addition and multiplication operators that combine complex numbers as expected

Structure

Complex structure changed for GPU

```
#include "book.h"
#include "cpu_bitmap.h"

#define DIM 1000

struct cuComplex {
    float   r;
    float   i;
    __host__ __device__ cuComplex( float a, float b ) : r(a), i(b)
        {}
    __device__ float magnitude2( void ) {
        return r * r + i * i;
    }
    __device__ cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    __device__ cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
```

Julia

Changes julia function as device

```
__device__ int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

Julia Kernel

Same Kernel

```
__global__ void kernel( unsigned char *ptr ) {
    // map from blockIdx to pixel position
    int x = blockIdx.x;
    int y = blockIdx.y;
    int offset = x + y * gridDim.x;

    // now calculate the value at that position
    int juliaValue = julia( x, y );
    ptr[offset*4 + 0] = 255 * juliaValue;
    ptr[offset*4 + 1] = 0;
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}
```

Julia Main

```
// globals needed by the update routine
struct DataBlock {
    unsigned char    *dev_bitmap;
};
int main( void ) {
    DataBlock   data;
    CPUBitmap bitmap( DIM, DIM, &data );
    unsigned char     *dev_bitmap;

    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap, bitmap.
        image_size() ) );
    data.dev_bitmap = dev_bitmap;

    dim3    grid(DIM,DIM);
    kernel<<<grid,1>>>( dev_bitmap );

    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );
    HANDLE_ERROR( cudaFree( dev_bitmap ) );
    bitmap.display_and_exit();
}
```

- We then run our kernel() function exactly like in the CPU version, although now it is a ___ global___ function, run on the GPU.
- Similar to CPU, we pass kernel() the pointer we allocated in the previous line to store the results
- The only difference is that the memory resides on the GPU now, not on the host system.

Compilation

- W specify a two-dimensional grid of blocks in this line: `dim3 grid(DIM,DIM);`
- The type dim3 is not a standard C type,
- The type dim3 represents a three-dimensional tuple that will be used to specify the size of our launch

Compilation

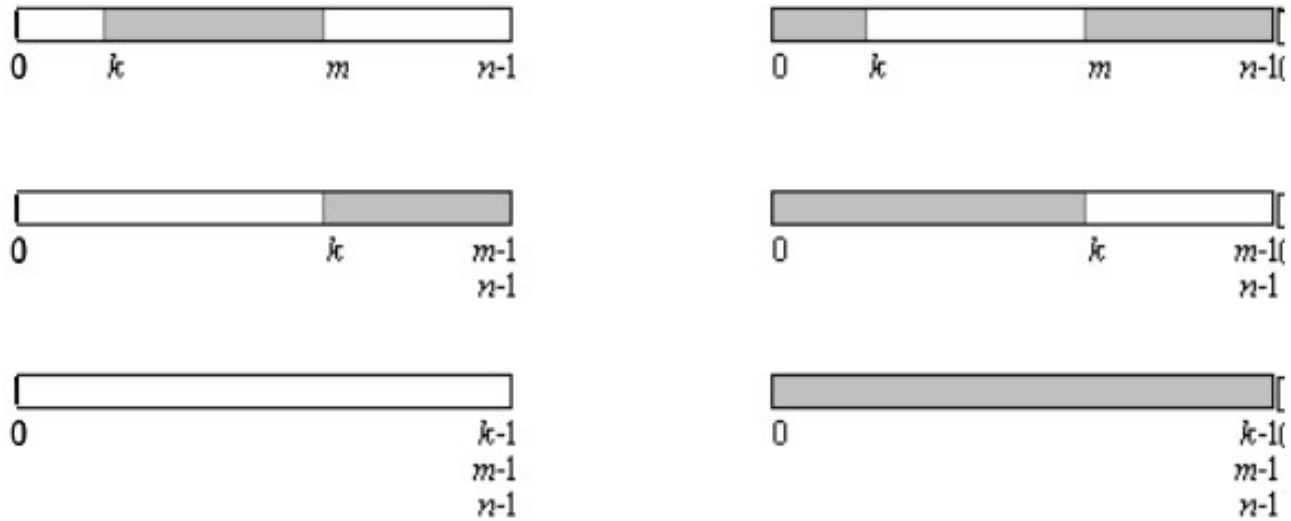
- To run this example, we require gl, glut, glut3 and glu files which is available in Ubuntu
- To get those files, install freeglut3-dev
- To install, type in terminal sudo apt-get install freeglut3-dev
- For compilation, nvcc -lglut -lGL -lGLU juliagpu.cu

specify architecture while compiling CUDA

- Sometimes it happens we need to specify the architecture
- For example, if your code contain `atomicAdd()` function then your SM architecture should be $>$ SM10
- Otherwise you get atomicAdd undefined identifier
- In this case, specify nvcc –arch=sm___ 30
- Or nvcc -gencode-arch=compute_30,code=sm_30,compute_30

Bitonic Sort

- **Binary sequence b_n-1 with at max two transitions at k and m.**



- **A sequence A = $a_0, a_1, \ldots, a_{n-1}$ is bitonic iff**

1. There is an index i, 0 < i < n, s.t.

$a_0 .. a_i$ **is increasing**

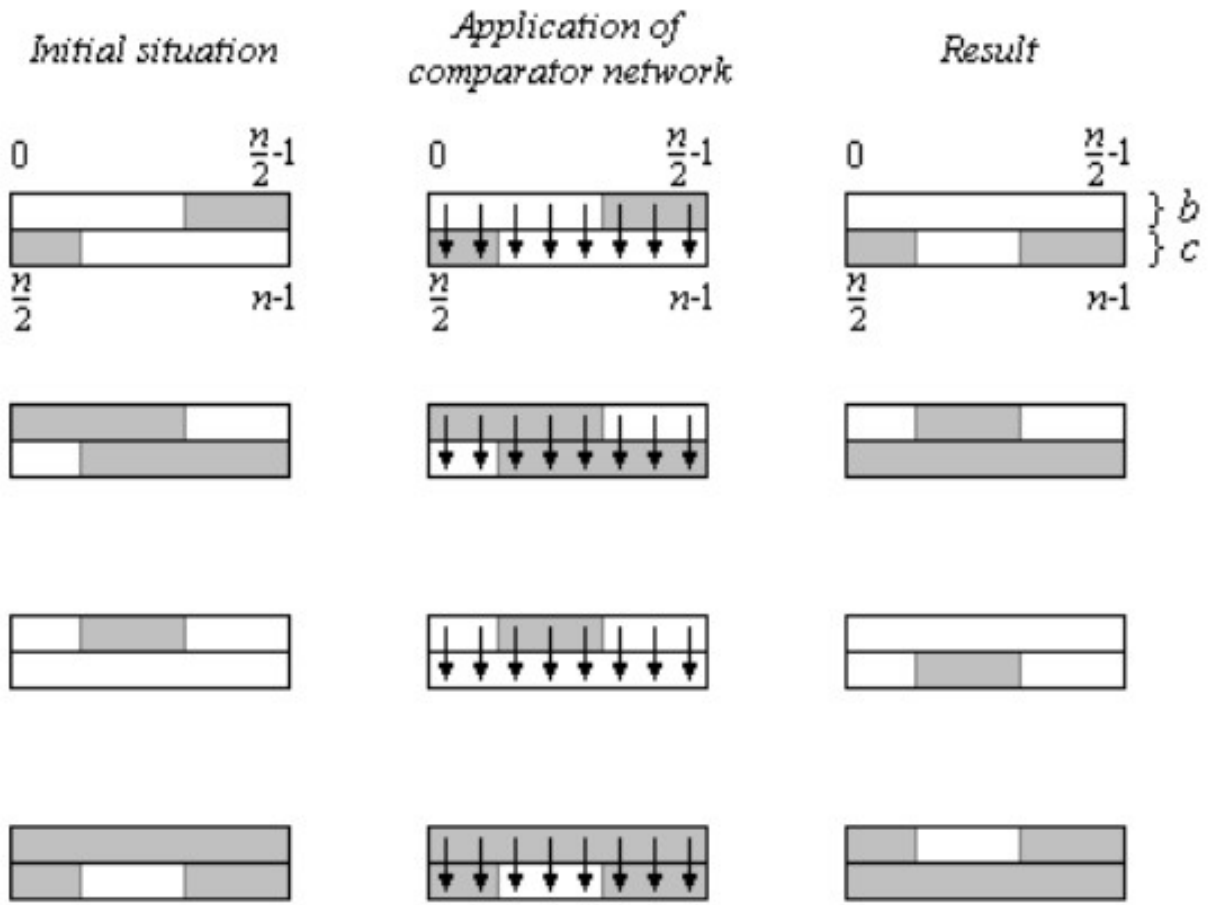
and

$a_i .. a_{n-1}$ **is decreasing**

or 2. There is a cyclic shift of A for which 1 holds.

Bitonic Sort

- **Given two bitonic sets b_n, c_n perform an element wise comparison storing the larger(smaller) value in c.**
- **Output is guaranteed to be bitonic again.**



Bitonic Sort

- **A bitonic split divides a bitonic sequence in two:**

$$\text{BitSplit(BS)} = \begin{cases} S1 = (\min(bs_0, bs_{n/2}),\ \min(bs_1, bs_{n/2+1}),\ \ldots,\ \min(bs_{n/2-1}, bs_{n-1})) \\ S2 = (\max(bs_0, bs_{n/2}),\ \max(bs_1, bs_{n/2+1}),\ \ldots,\ \max(bs_{n/2-1}, bs_{n-1})) \end{cases}$$

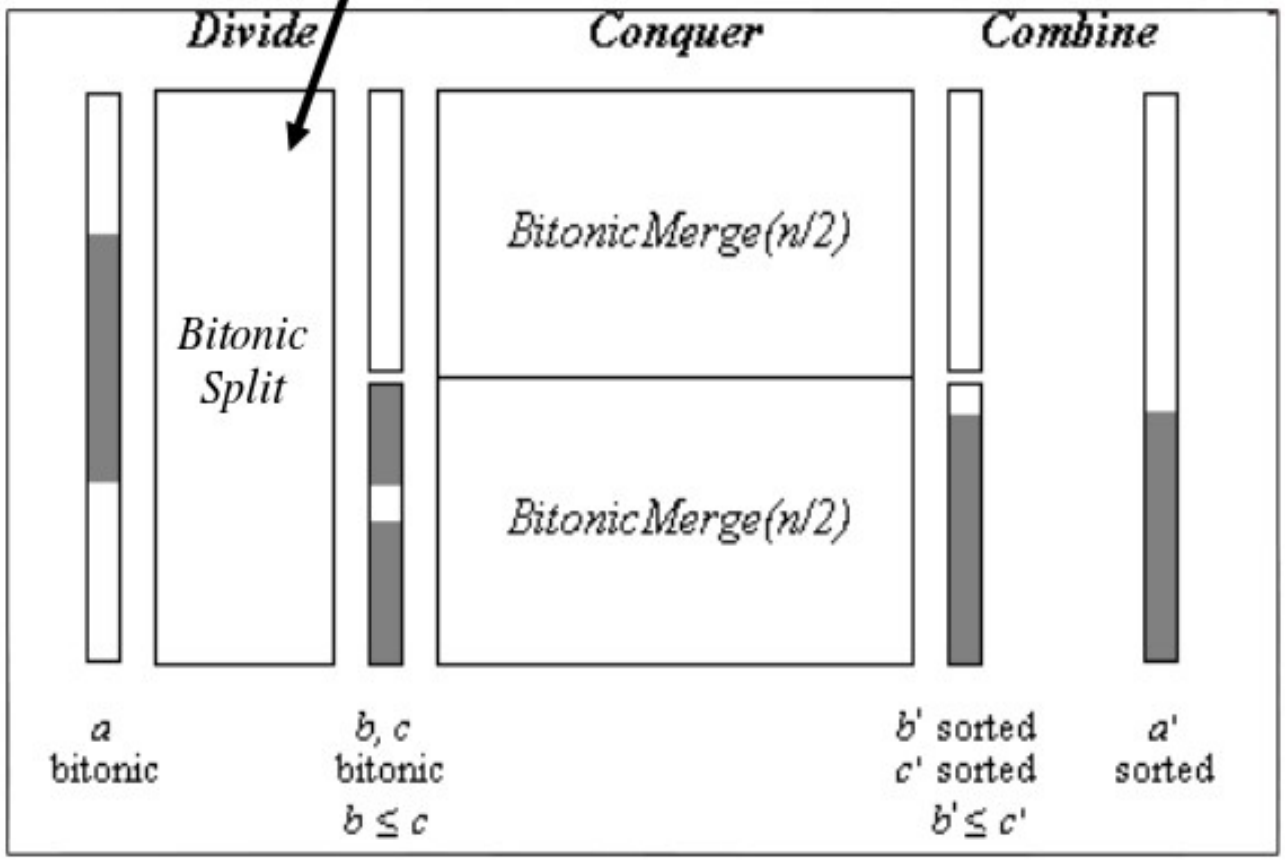
- **Theorem :**

  S1 and S2 are both bitonic

  S1 < S2

Bitonic Sort

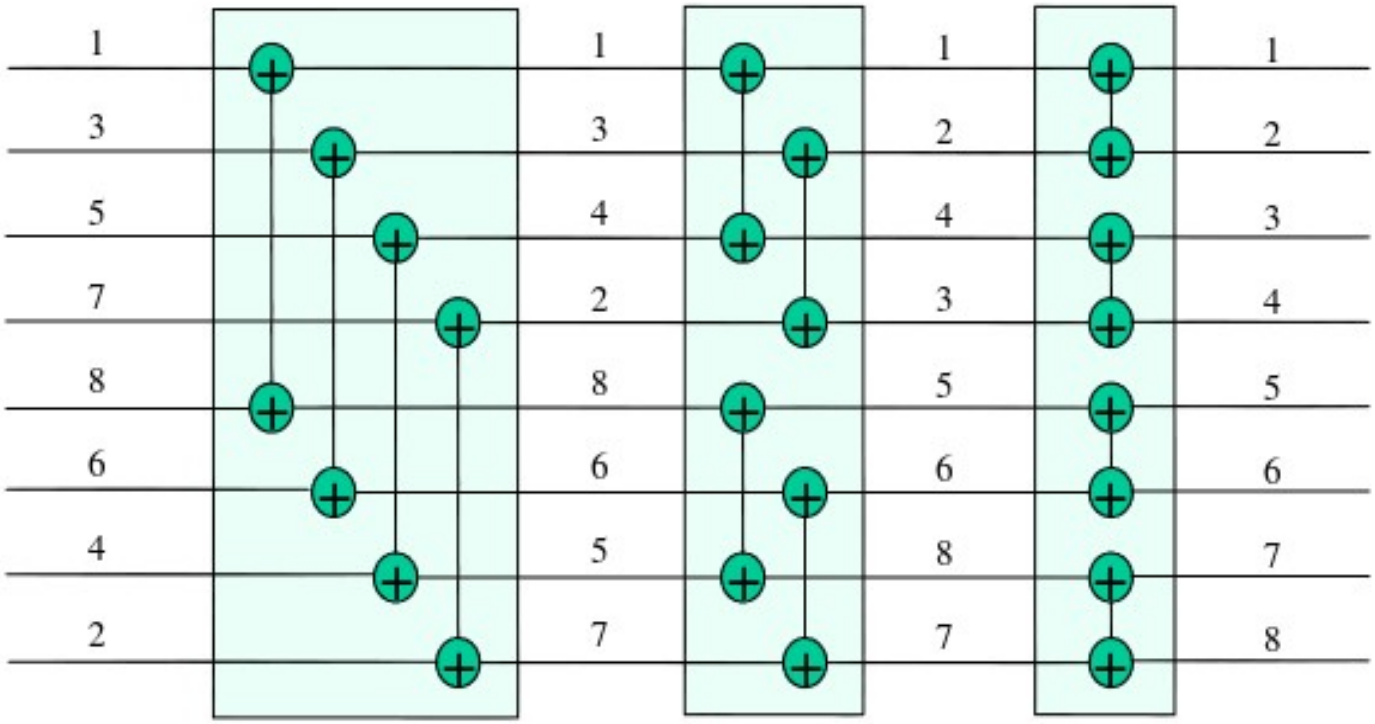
- **Given a bitonic vector, it will be sorted by applying a comparison network (divide) to its to halves.**
- **Then recurse.**



BitonicMerge(n)

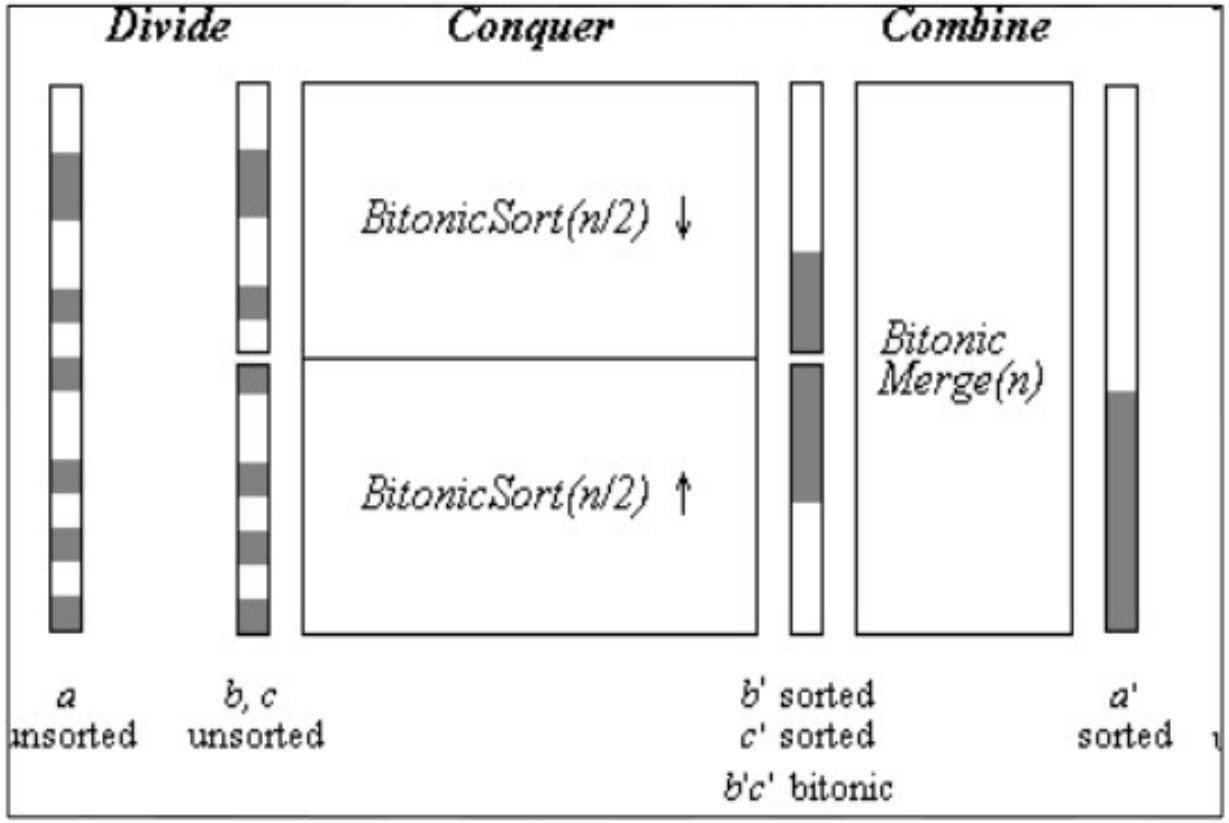
Bitonic Sort

- **Given: a Bitonic Sequence BS of size n = 2^m**
- **Sort BS using m (parallel) Bitonic Splits stages**



Bitonic Sort

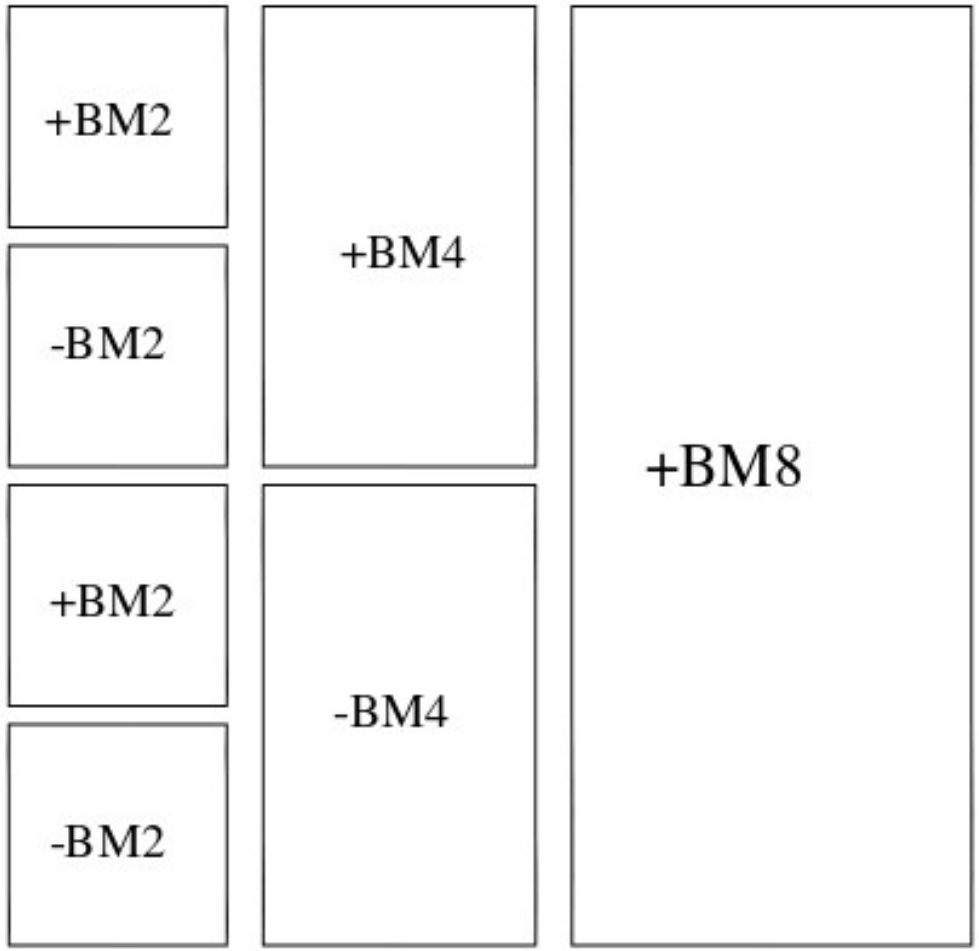
- **Split into two halves**
- **Sort ascending/descending**
- **Merge**



BitonicSort(n)

Bitonic Sort

- **Same number of comparison operations in each iteration**



Bitonic Sort- Summary

- ***Complexity O(n* log$(n)^2$)**
- **Fixed comparison networks**
  - they are all operating on the list in parallel
- **Simple to implement**

Code

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

/* Every thread gets exactly one value in the unsorted array. */
#define THREADS 512 // 2^9
#define BLOCKS 32768 // 2^15
#define NUM_VALS THREADS*BLOCKS

void print_elapsed(clock_t start, clock_t stop)
{
  double elapsed = ((double) (stop - start)) / CLOCKS_PER_SEC;
  printf("Elapsed time: %.3fs\n", elapsed);
}
```

Code

```
float random_float()
{
  return (float)rand()/(float)RAND_MAX;
}

void array_print(float *arr, int length)
{
  int i;
  for (i = 0; i < length; ++i) {
    printf("%1.3f ",  arr[i]);
  }
  printf("\n");
}
```

Code

```
void array_fill(float *arr, int length)
{
  srand(time(NULL));
  int i;
  for (i = 0; i < length; ++i) {
    arr[i] = random_float();
  }
}

__global__ void bitonic_sort_step(float *dev_values, int j, int k)
{
  unsigned int i, ixj; /* Sorting partners: i and ixj */
  i = threadIdx.x + blockDim.x * blockIdx.x;
  ixj = i^j;
```

Code

```
  /* The threads with the lowest ids sort the array. */
  if ((ixj)>i) {
    if ((i&k)==0) {
      /* Sort ascending */
      if (dev_values[i]>dev_values[ixj]) {
        /* exchange(i,ixj); */
        float temp = dev_values[i];
        dev_values[i] = dev_values[ixj];
        dev_values[ixj] = temp;
      }
    }
    if ((i&k)!=0) {
      /* Sort descending */
      if (dev_values[i]<dev_values[ixj]) {
        /* exchange(i,ixj); */
        float temp = dev_values[i];
        dev_values[i] = dev_values[ixj];
        dev_values[ixj] = temp;
      }
    }
  }
}
```

Code

```
/**
 * Inplace bitonic sort using CUDA.
 */
void bitonic_sort(float *values)
{
  float *dev_values;
  size_t size = NUM_VALS * sizeof(float);

  cudaMalloc((void**) &dev_values, size);
  cudaMemcpy(dev_values, values, size, cudaMemcpyHostToDevice);

  dim3 blocks(BLOCKS,1);    /* Number of blocks   */
  dim3 threads(THREADS,1);  /* Number of threads  */

  int j, k;
  /* Major step */
  for (k = 2; k <= NUM_VALS; k <<= 1) {
    /* Minor step */
    for (j=k>>1; j>0; j=j>>1) {
      bitonic_sort_step<<<blocks, threads>>>(dev_values, j, k);
    }
  }
```

Code

```
  cudaMemcpy(values, dev_values, size, cudaMemcpyDeviceToHost);
  cudaFree(dev_values);
}

int main(void)
{
  clock_t start, stop;

  float *values = (float*) malloc( NUM_VALS * sizeof(float));
  array_fill(values, NUM_VALS);

  start = clock();
  bitonic_sort(values); /* Inplace */
  stop = clock();

  print_elapsed(start, stop);
}
```

THANK YOU