Exploring Dynamic Parallelism in CUDA C with Mandelbrot Sets

Joseph Zhong
josephz@cs.washington.edu
University of Washington

June 1, 2017

1 Introduction

Mandelbrot Sets are a well studied mathematical concept. Mathematically, the set of complex numbers is formally defined as the set of complex value c for which the quadratic $z_{n+1} = z_n^2 + c$ remains bounded for a large number of iterations n where $z_0 = 0$. We can formally define the mandelbrot set M as the following for a radius R:

$$M = \{c \in \mathcal{C} : \exists R \forall n : |z_n| < R\}$$

Visually, we can create an image of the Mandelbrot set by plotting the complex points c, assigning membership of the Mandelbrot set M. By treating the real and imaginary parts of c as our a and bi coordinates in the complex plane. In theory, this visualization of the Mandelbrot Set will generate infinitely recursing factals at increasing magnification levels throughout the shape of the set.

In the context of graphically representing Mandelbrot sets in through computer generated graphics, we can color pixels representing points based on the behavior the algorithm used to determine whether a point belongs in the set. A common strategy to determine whether a point belongs within the Mandelbrot Set is to limit the number of iterations, and assign set membership once the iteration limit has been reached.

Notice that with this strategy, we can notice that complex numbers of real and imaginary parts greater than 2 cannot be a part of the set, and thus a common cheap and easily implemented strategy is simply to check the complex number's parts for exceeding this threshold. While there exist methods which can detect set membership quicker such as using Euclidean distance from the origin, for simplicity we present this strategy to compute a point's membership.

With this strategy, we can then utilize a color mapping function using the iteration limit to assign pixel color values for each point. One simple algorithm which utilizes this strategy is known as the Escape Time Algorithm, which can compute the per-pixel values of the points within the Mandelbrot Set.

In this report we compare the performance of the Escape Time Algorithm when implemented naively to calculate the per-pixel value, against an algorithm, which takes advantage of the fact that Mandelbrot sets are connected. Thus, by only calculating the values of the borders, we can determine whether the inner shape can be filled in if the border values are equal. Thus, we can assign compute power with finer grain detail, assigning fewer threads to portions of the problem, which either complete earlier or uniformly assign pixel value to a larger area, while leaving more compute resources for smaller areas of the problem which require more time to complete for a smaller area of pixels.

This implementation strategy is also known as Dynamic Parallelism. Parallel computation is exploited as our algorithm discovers more parallelizable work during runtime, rather than statically dispatching a stringent fixed number of parallel compute resources at compile time. In CUDA C specifically, dynamic parallelism was introduced in CUDA 5.0 with the Kepler architecture in 2012, and is available to devices of compute capacity of 3.5 or greater.

We will see that for the initial and conlonical image of a Mandelbrot set that we can achieve significant speed up in terms of pixel values computed per second for increasing resolution and iteration limits. We also explore pathological cases in which dynamic parallelism cannot provide much improvement, or where the modified border-tracing algorithm introduces additional overhead against the Escape Time algorithm. Code will be made available at https://github.com/joseph-zhong/MandelbrotSets

2 The Escape Time Algorithm

The Escape Time Algorithm conducts a repeating calculation for each c in our plot with a fixed iteration limit. The color assigned for each point is determined by a mapping from the number of iterations needed to determine membership. The core idea is to quantize our plot into representable pixels by coordinates x and y, and for each pixel, approximate the likelihood that it belongs in the Mandelbrot set. Realize that by definition, that points within the set in theory will never "escape", and thus representation accuracy is a function of how high the maximum iterations cap is set, along with the resolution granularity of the image to subdivide the plot into. We present the Escape Time Algorithm below.

Algorithm: Escape Time Algorithm Formulation
x: Input real value coordinate.
y: Input imaginary value coordinate.
k: Input maximum iterations cap.

set iterations count to 0.
set z value to (x, yi)
while count is less than k or z^2 is less than 4 {
  $z \leftarrow z^2 + c$
  increment count
}
return count

Recall that the square of a complex number is defined as the following.

$$z^2 = (x^2 - y^2) + i(2xy)$$

With the output of the escape time algorithm, we then can easily map to a value [0,255] to be used as a pixel intensity value. In particular, this value will be used to color the corresponding pixel value at (x, y). In languages which do not support complex typing, one must manually compute the real and imaginary operations separately. In our implementation we define a `complexNum` struct which acts as our complex number. We also define the corresponding complex number operations. We present our per-pixel kernel implementation below. Note that we abstract the Escape Time Algorithm such that it may be used on the host as well.

```
__global__ void cudaNaiveMandelbrotSetsKernel(
    int *d_output, int width, int height,
    int maxIterations, const float radius,
    complexNum cMin, complexNum cMax) {

  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  if (x >= width || y >= height) return;

  int value = calculatePixelValue(width, height,
    maxIterations, cMin, cMax,
    x, y, radius);
  d_output[y * width + x] = value;
}

__host__ __device__ int calculatePixelValue(
    int width, int height, int maxIterations,
    complexNum cMin, complexNum cMax,
    int x, int y,
    const float radius) {
  // Plot bounds.
  complexNum diff = cMax - cMin;

  // Determine pixel position.
  float xPos = (float) x / width * diff.a;
  float yPos = (float) y / height * diff.bi;

  // Initialize c and z.
  complexNum c = cMin + complexNum(xPos, yPos);
  complexNum z = c;

  int iterations = 0;
  while (iterations < maxIterations
    && absSquared(z) < radius) {
    z = z * z + c;
    iterations++;
  }
  return iterations;
}
```

As modeled from the pseudocode, the iterations is then utilized later in assigning colors to each pixel subdividing our plot space. The `width` and `height` are parameters specifying the resolution size of our target image, while `cMin` and `cMax` represent the points in our complex plane which bound the plot area we wish to work in. Notice that we derive our pixel locations by mapping the complex plane into the domain of the `height` and `width`, and scale the points with the relative positions of `x` and `y`. With this implementation, we can easily launch a kernel which assigns a thread to each pixel, and simply calculate the per-pixel value using the Escape Time Algorithm. We present our per-pixel kernel launch below.

```
__host__ void cudaNaiveMandelbrotSets(
    int height, int width,
    int maxIterations, const float radius,
    const complexNum cMin, const complexNum cMax,
    const char *filename) {
  // Host input setup: image.
  const int OUTPUT_SIZE =
    sizeof(int) * height * width;
  int *h_output = (int*) malloc(OUTPUT_SIZE);

  // Device output setup: image.
  int *d_output;
  cudaCheck(cudaMalloc(&d_output, OUTPUT_SIZE));

  // Kernel Size.
  dim3 gridSize(ceil(width / TILE_WIDTH),
                ceil(height / TILE_WIDTH), 1);
  dim3 blockSize(TILE_WIDTH, TILE_WIDTH, 1);

  // Begin timer.
  clock_t start = clock();

  // Launch Kernel.
  cudaNaiveMandelbrotSetsKernel
      <<<gridSize, blockSize>>>(
      d_output, width, height, maxIterations,
      radius, cMin, cMax);

  // Synchronize across threads once completed.
  cudaCheck(cudaThreadSynchronize());

  // Stop timer.
  endClock(start);

  if (filename != NULL) {
    // Copy output.
    cudaCheck(cudaMemcpy(h_output, d_output,
              OUTPUT_SIZE,
              cudaMemcpyDeviceToHost));

    // Write to output.
    saveImage(filename, h_output, width, height,
              maxIterations);
  }

  // Free output.
  cudaFree(d_output);
  free(h_output);
}
```

While this kernel formulation strategy greatly takes advantage of parallelizable work, assigning each available thread to compute the membership of each data-independent pixel, the fact that the Mandelbrot set is connected is overlooked. Within the mandelbrot set, there are large plots of area which the neighboring pixels could all be immediately assigned membership, saving significant compute time and device resources. In particular, here the algorithm spends the most resources computing the points within the mandelbrot set for large maximum iteration caps. However, when we utilize the fact that the Mandelbrot set is connected, we can greatly reduce compute resources required to assign membership, and instead, allow additional resources to compute membership along the fractal boundary, where iterations will be still high, but of a unique recursively repeating pattern of fine grained detail. Let us introduce the Border-Tracing Algorithm and modify our above implementation to utilize dynamic parallelism to achieve finer granularity computation for the above motivated scenario.

3 The Border-Tracing Algorithm

The Border-Tracing Algorithm deviates from the original per-pixel kernel method by instead, assigning patches of the plot to each thread, where the goal of each parallel thread determines the membership of the entire patch if possible. The algorithm takes advantage of the property that Mandelbrot Sets are connected. Thus, in theory, if an arbitrary closed polygon has borders of uniform values, then we can determine that the pixels within also share the value. In practice, this algorithm works under the assumption of fine enough grain resolution to capture the subtle fractal pattern at the fringe of the Mandelbrot set. Additionally, rather than computing the borders of arbitrary polygons, rectangles are used instead as the rectangle borders are easier to represent. Additionally, with rectangles, we can recursively repeat the algorithm to operate on sub-rectangles. Thus, the core idea is to utilize the Escape Time Algorithm only on the border of our selected rectangle, filling the internal pixels with the uniform border's value if possible. Otherwise we split the rectangle and recurse. One additional optimization to realize is that at some cutoff, our sub-rectangles will eventually become small enough such that the overhead involved with checking the border values and continuously recursing will outweigh simply running our Escape Time Algorithm kernel. We present the Border-Tracing Algorithm below.

Algorithm: Border-Tracing Algorithm Formulation
x: Input real value coordinate of our rectangle.
y: Input imaginary value coordinate of our rectangle.
w: Input width of our rectangle.
h: Input height of our rectangle.
threshold: Input cut off threshold for small rectangles.

border = pixels$[(x, y){:}(x + w, y)]$
  + pixels$[(x + w, y){:}(x + w, y + h)]$
  + pixels$[(x, y){:}(x, y + h)]$
  + pixels$[(x, y + h){:}(x + w, y + h)]$
for every pixel along the border:
  Compute the value with the Escape Time Algorithm.
if every pixel value along the border is the same:
  Fill every pixel within the border.
else if $w * h < threshold$:
  Calculate every pixel within the border.
else:
  Split our rectangle into sub-rectangles and recurse.

With this implementation, implementation complexity has increased over the per-pixel Escape Time Algorithm, and thus has produced the opportunity for finer-grain implementation optimization. Notice that through the introduction of Dynamic Parallelism in CUDA, we can implement the `Fill every pixel...` block as a dynamically parallel kernel. Through Dynamic Parallelism, we can reuse our previous Escape Time Algorithm kernel for the `Calculate every pixel within...` block. Recall that the in our previous implementation using the per-pixel Escape Time Algorithm kernel, that every pixel was assumed to be entirely independent. Notice that this pseudocode takes into account computational intensity as a patch area over our plot. Ideally we should only be computing independent per-pixel values near the fringe of the Mandelbrot set fractal, which in general would be at miniscule scale relative to the overall image size and thus should receive per-pixel granular computation. We present our implementation of the kernels below.

Below is the Border-Tracing Kernel.

```
__global__ void cudaDPMandelbrotSetsKernel(
    int height, int width, int maxIterations,
    complexNum cMin, complexNum cMax,
    int x0, int y0, int size, int depth,
    const float radius, int *d_output) {
  x0 += size * blockIdx.x;
  y0 += size * blockIdx.y;

  int borderVal = calculateBorder(width, height,
                  maxIterations, cMin, cMax,
                  x0, y0, size, radius);

  if(threadIdx.x == 0 && threadIdx.y == 0) {
    if (borderVal != -1) {
      dim3 fillBlockSize(BLOCK_SIZE,
         DIVIDE_FACTOR);
      dim3 fillGridSize(divup(size, BLOCK_SIZE),
         divup(size, DIVIDE_FACTOR));
      fillKernel<<<fillGridSize, fillBlockSize>>>(
  width, x0, y0, size, borderVal, d_output);
    }
    else if (depth + 1 < MAX_DEPTH
  && size / DIVIDE_FACTOR > MIN_SIZE) {
      dim3 recurseGridSize(DIVIDE_FACTOR,
         DIVIDE_FACTOR);
      dim3 recurseBlockSize(blockDim.x,
        blockDim.y);
      cudaDPMandelbrotSetsKernel
        <<<recurseGridSize, recurseBlockSize>>>(
          height, width, maxIterations,
          cMin, cMax, x0, y0,
          size / DIVIDE_FACTOR,
          depth + 1, radius, d_output);
    }
    else {
      dim3 pixelGridSize(divup(size, BLOCK_SIZE),
         divup(size, DIVIDE_FACTOR));
      dim3 pixelBlockSize(BLOCK_SIZE,
        DIVIDE_FACTOR);
      pixelKernel
  <<<pixelGridSize, pixelBlockSize>>>(
    width, height, maxIterations,
    cMin, cMax, x0, y0, size, radius, d_output);
    }
  }
}
```

Below is the `calcaulteBorder` implementation.

```
__device__ int calculateBorder(int width, int height, int maxIterations,
    complexNum cMin, complexNum cMax, int x0, int y0, int size, const float radius) {
  int tIdx = threadIdx.y * blockDim.x + threadIdx.x;
  int blockSize = blockDim.x * blockDim.y;
  int value = maxIterations + 1;
  for (int pixel = tIdx; pixel < size; pixel += blockSize) {
    for (int boundary = 0; boundary < 4; boundary++) {
      int x = boundary \% 2 != 0 ? x0 + pixel : (boundary == 0 ? x0 + size - 1 : x0);
      int y = boundary \% 2 == 0 ? y0 + pixel : (boundary == 1 ? y0 + size - 1 : y0);
      value = commonValue(value, calculatePixelValue(width, height, maxIterations, cMin, cMax, x, y, radius), max
    }
  }

  __shared__ int s_output[BLOCK_SIZE * DIVIDE_FACTOR];
  int numThreads = min(size, BLOCK_SIZE * DIVIDE_FACTOR);
  if (tIdx < numThreads) {
    s_output[tIdx] = value;
  }
  __syncthreads();

  // while (numThreads > 1) {
  for (; numThreads > 1; numThreads /= 2) {
    if (tIdx < numThreads / 2) {
      s_output[tIdx] = commonValue(s_output[tIdx], s_out
    }
    __syncthreads();
  }
  return s_output[0];
}
```

Below are the `fillKernel` and `pixelKernel` implementations. Notice their stark similarity to the original per-pixel implementation.

```
__global__ void pixelKernel(int width, int height,
    int maxIterations, complexNum cMin,
    complexNum cMax, int x0, int y0, int size,
    const float radius, int *d_output) {
  int x = threadIdx.x + blockDim.x * blockIdx.x;
  int y = threadIdx.y + blockDim.y * blockIdx.y;

  if (x < size && y < size) {
    x += x0;
    y += y0;
    d_output[y * width + x] = calculatePixelValue(
      width, height, maxIterations, cMin, cMax, x,
      y, radius);
  }
}

__global__ void fillKernel(int width, int x0,
    int y0, int size, int value, int *d_output) {
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;

  if (x < size && y < size) {
    x += x0;
    y += y0;
    d_output[y * width + x] = value;
  }
}
```

One major difference however, is to realize that the above two kernels were launched **from the device** with parallelizable work discovered at runtime, rather than being prepared and run once after compile time! This is the core of Dynamic Parallelism: the capacity to **allocate device memory and parallel tasks from children of the host**.

4 Results

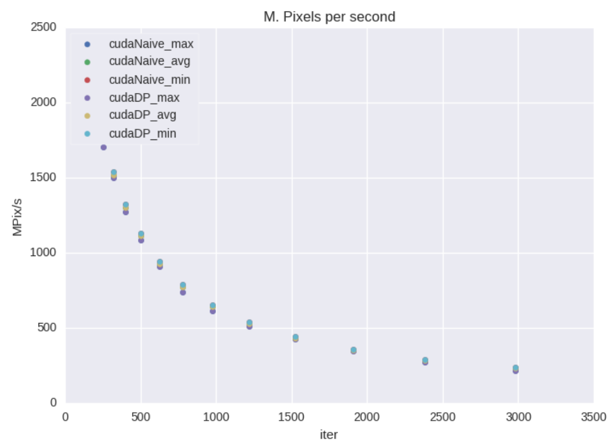

Figure 1: Megapixels per second, varying iteration cap.

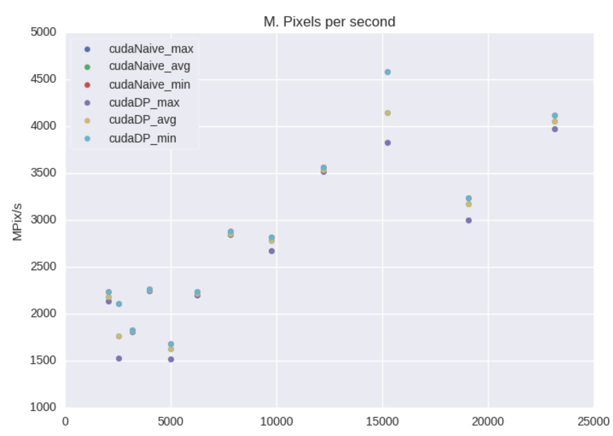

Figure 2: Megapixels per second, varying image resolution.

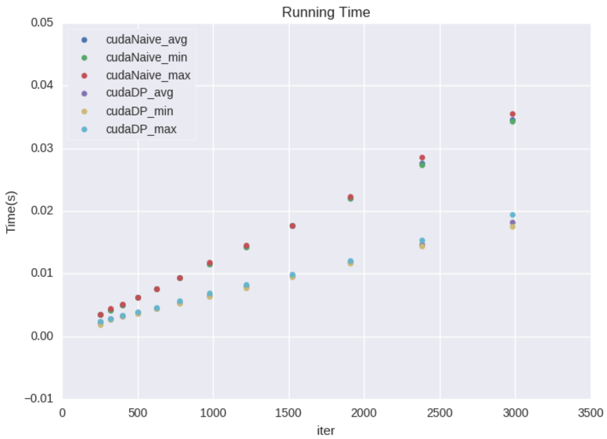

Figure 3: Kernel running time, varying iteration cap.

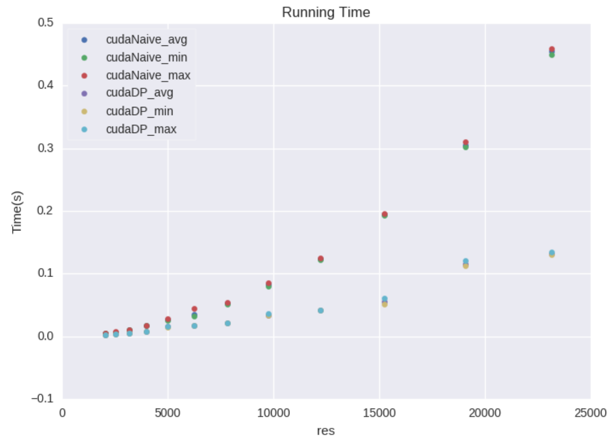

Figure 4: Kernel running time, varying image resolution.

5 Conclusion