# Parallel binary image skeletonization with CUDA

Sahand Kashani-Akhavan [201034]

June 15, 2017

## Contents

**1 Problem description** **2**

1.1 "Hot zone" detection . . . . . . . . . . 2

1.2 Complexity analysis . . . . . . . . . . 6

1.3 Theoretical maximum speedup . . . . . . . . . . 6

1.4 Reference machine . . . . . . . . . . 7

**2 Parallelization solutions** **7**

2.1 GPU 1 . . . . . . . . . . 8

2.1.1 Idea . . . . . . . . . . 8

2.1.2 Analysis . . . . . . . . . . 8

2.2 GPU 2 . . . . . . . . . . 9

2.2.1 Idea . . . . . . . . . . 9

2.2.2 Analysis . . . . . . . . . . 10

2.3 GPU 3 . . . . . . . . . . 10

2.3.1 Idea . . . . . . . . . . 10

2.3.2 Analysis . . . . . . . . . . 11

2.4 GPU 4 . . . . . . . . . . 11

2.4.1 Idea . . . . . . . . . . 11

2.4.2 Analysis . . . . . . . . . . 12

**3 Detailed Analysis** **13**

**4 Conclusion** **14**

## 1 Problem description

The problem to solve consists of the skeletonization of binary images. A binary pixel is colored white if its value is 0, and black if its value is 1. Given an $M \times N$ binary image, the skeletonization algorithm performs a thinning operation iteratively until the image no longer changes.

For each pixel $P1$, the thinning algorithm decides whether it becomes white by considering information about its 8 neighbor pixels $P2$, $P3$, $\ldots P9$. Let $TR(P1)$ be the number of white to black transitions in the neighborhood of $P1$. Let $NZ(P1)$ be the number of black neighbors of $P1$. Then, $P1$ becomes white if *all* the following conditions are true:

$$\begin{cases} 2 \leq NZ(P1) \leq 6 \\ TR(P1) = 1 \\ (P2 \cdot P4 \cdot P8 = 0) \vee (TR(P2) \neq 1) \\ (P2 \cdot P4 \cdot P6 = 0) \vee (TR(P4) \neq 1) \end{cases}$$

Note that the value at $P1$ after one iteration of the thinning algorithm depends on $TR(P2)$ and $TR(P4)$. This causes $P1$ to depend on the values of its 15 surrounding pixels. Figure 1 shows this dependency.

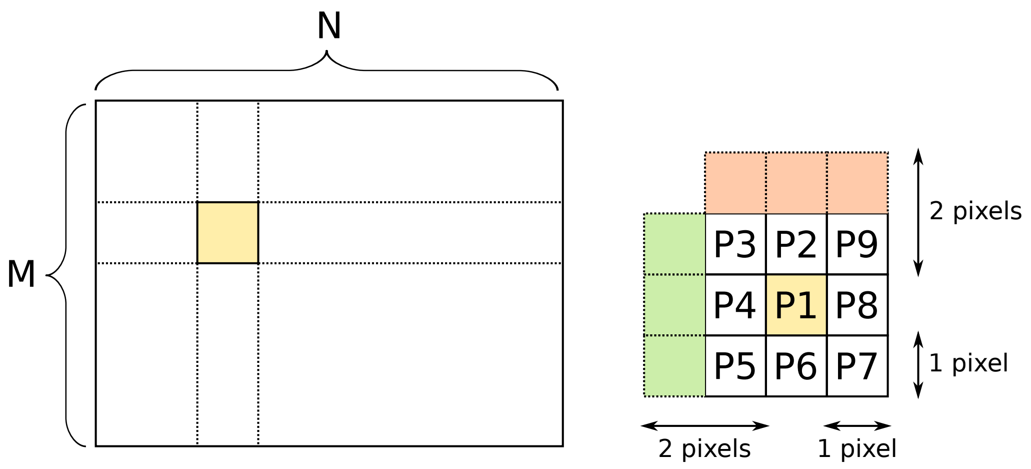


Figure 1: Problem description

### 1.1 "Hot zone" detection

I am going to use CUDA to accelerate this appliction. However, in order find what part to accelerate, I first need to know where the "hot zones" of the code are located. To study the application, I have written a straightforward single-threaded CPU implementation of the algorithm.

The code is supplied in Listing 1 for reference. To keep the code brief, most input checking code and NULL pointer verifications are not shown here, and I only focus on the main functions of the implementation.

```c
// Performs an image skeletonization algorithm on the input Bitmap, and stores
// the result in the output Bitmap.
int skeletonize(Bitmap** src_bitmap, Bitmap** dst_bitmap)
{
  int iterations = 0;

  do
  {
    memcpy((*dst_bitmap)->data,
           (*src_bitmap)->data,
           (*src_bitmap)->width * (*src_bitmap)->height * sizeof(uint8_t));

    skeletonize_pass((*src_bitmap)->data,
                     (*dst_bitmap)->data,
                     (*src_bitmap)->width, (*src_bitmap)->height);

    swap_bitmaps((void**) src_bitmap, (void**) dst_bitmap);

    iterations++;
  }
  while (!are_identical_bitmaps(*src_bitmap, *dst_bitmap));

  return iterations;
}

// Performs 1 iteration of the thinning algorithm.
void skeletonize_pass(uint8_t* src, uint8_t* dst, int width, int height)
{
  for (int row = 0; row < height; row++)
  {
    for (int col = 0; col < width; col++)
    {
      // Optimization for CPU algorithm: You don't need to do any of these
      // computations if the pixel is already BINARY_WHITE
      if (src[row * width + col] == BINARY_BLACK)
      {
        uint8_t NZ = black_neighbors_around(src, row, col, width, height);
        uint8_t TR_P1 = wb_transitions_around(src, row, col, width, height);
        uint8_t TR_P2 = wb_transitions_around(src, row-1, col, width, height);
        uint8_t TR_P4 = wb_transitions_around(src, row, col-1, width, height);
        uint8_t P2 = P2_f(src, row, col, width, height);
        uint8_t P4 = P4_f(src, row, col, width, height);
        uint8_t P6 = P6_f(src, row, col, width, height);
        uint8_t P8 = P8_f(src, row, col, width, height);

        uint8_t thinning_cond_1 = ((2 <= NZ) && (NZ <= 6));
        uint8_t thinning_cond_2 = (TR_P1 == 1);
        uint8_t thinning_cond_3 = (((P2 && P4 && P8) == 0) || (TR_P2 != 1));
        uint8_t thinning_cond_4 = (((P2 && P4 && P6) == 0) || (TR_P4 != 1));

        uint8_t thinning_cond_ok = thinning_cond_1 && thinning_cond_2 &&
                                   thinning_cond_3 && thinning_cond_4;

        if (thinning_cond_ok)
        {
          dst[row * width + col] = BINARY_WHITE;
        }
      }
    }
  }
}

// Computes the number of black neighbors around a pixel.
uint8_t black_neighbors_around(uint8_t* data, int row, int col, int width,
                               int height)
{
  uint8_t count = 0;

  count += (P2_f(data, row, col, width, height) == BINARY_BLACK);
  count += (P3_f(data, row, col, width, height) == BINARY_BLACK);
  count += (P4_f(data, row, col, width, height) == BINARY_BLACK);
  count += (P5_f(data, row, col, width, height) == BINARY_BLACK);
  count += (P6_f(data, row, col, width, height) == BINARY_BLACK);
  count += (P7_f(data, row, col, width, height) == BINARY_BLACK);
  count += (P8_f(data, row, col, width, height) == BINARY_BLACK);
  count += (P9_f(data, row, col, width, height) == BINARY_BLACK);

  return count;
}

// Computes the number of white to black transitions around a pixel.
uint8_t wb_transitions_around(uint8_t* data, int row, int col, int width,
                              int height)
{
  uint8_t count = 0;

  count += ((P2_f(data, row, col, width, height) == BINARY_WHITE) &&
            (P3_f(data, row, col, width, height) == BINARY_BLACK));
  count += ((P3_f(data, row, col, width, height) == BINARY_WHITE) &&
            (P4_f(data, row, col, width, height) == BINARY_BLACK));
  count += ((P4_f(data, row, col, width, height) == BINARY_WHITE) &&
            (P5_f(data, row, col, width, height) == BINARY_BLACK));
  count += ((P5_f(data, row, col, width, height) == BINARY_WHITE) &&
            (P6_f(data, row, col, width, height) == BINARY_BLACK));
  count += ((P6_f(data, row, col, width, height) == BINARY_WHITE) &&
            (P7_f(data, row, col, width, height) == BINARY_BLACK));
  count += ((P7_f(data, row, col, width, height) == BINARY_WHITE) &&
            (P8_f(data, row, col, width, height) == BINARY_BLACK));
  count += ((P8_f(data, row, col, width, height) == BINARY_WHITE) &&
            (P9_f(data, row, col, width, height) == BINARY_BLACK));
  count += ((P9_f(data, row, col, width, height) == BINARY_WHITE) &&
            (P2_f(data, row, col, width, height) == BINARY_BLACK));

  return count;
}

uint8_t P2_f(uint8_t* data, int row, int col, int width, int height)
{
  return is_outside_image(row-1, col, width, height)
                                              ? BINARY_WHITE
                                              : data[(row-1) * width + col];
}

uint8_t P3_f(uint8_t* data, int row, int col, int width, int height)
{
  return is_outside_image(row-1, col-1, width, height)
                                              ? BINARY_WHITE
                                              : data[(row-1) * width + (col-1)];
}

uint8_t P4_f(uint8_t* data, int row, int col, int width, int height)
{
  return is_outside_image(row, col-1, width, height)
                                              ? BINARY_WHITE
                                              : data[row * width + (col-1)];
}

uint8_t P5_f(uint8_t* data, int row, int col, int width, int height)
{
  return is_outside_image(row+1, col-1, width, height)
                                              ? BINARY_WHITE
                                              : data[(row+1) * width + (col-1)];
}

uint8_t P6_f(uint8_t* data, int row, int col, int width, int height)
{
  return is_outside_image(row+1, col, width, height)
                                              ? BINARY_WHITE
                                              : data[(row+1) * width + col];
}
```

```

uint8_t P7_f(uint8_t* data, int row, int col, int width, int height)
{
  return is_outside_image(row+1, col+1, width, height)
                                              ? BINARY_WHITE
                                              : data[(row+1) * width + (col+1)];
}

uint8_t P8_f(uint8_t* data, int row, int col, int width, int height)
{
  return is_outside_image(row, col+1, width, height)
                                              ? BINARY_WHITE
                                              : data[row * width + (col+1)];
}

uint8_t P9_f(uint8_t* data, int row, int col, int width, int height)
{
  return is_outside_image(row-1, col+1, width, height)
                                              ? BINARY_WHITE
                                              : data[(row-1) * width + (col+1)];
}

uint8_t is_outside_image(int row, int col, int width, int height)
{
  return (row < 0) || (row > (height-1)) || (col < 0) || (col > (width-1));
}
```

Listing 1: CPU version

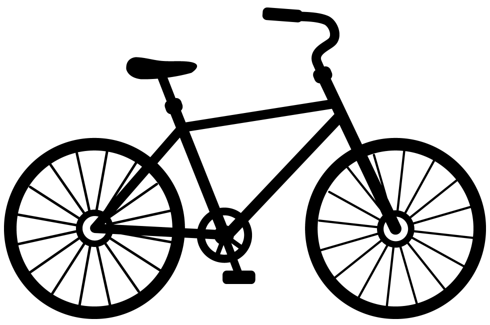


Figure 2: *Bicycle* : $6409 \times 4139$

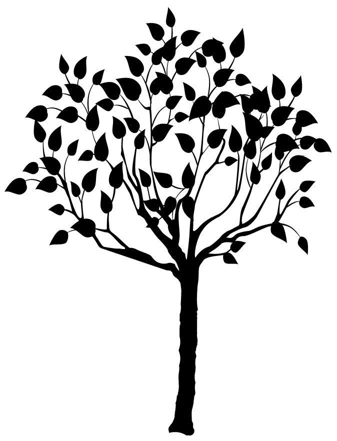


Figure 3: $Tree_6$ : $3507 \times 4582$

| | *Bicycle* | | $Tree_6$ | |
|---|---|---|---|---|
| Function | Incl. % | Excl. % | Incl. % | Excl. % |
| `main` | 100.0 | 0.00 | 100.0 | 0.00 |
| `skeletonize` | 98.62 | 0.00 | 97.91 | 0.00 |
| `skeletonize_pass` | 96.31 | 56.82 | 94.97 | 61.04 |
| `wb_transitions_around` | 39.49 | 39.49 | 33.93 | 33.93 |
| `are_identical_bitmaps` | 2.31 | 2.31 | 2.94 | 2.94 |

Table 1: CPU profiling data

Table 1 shows some profiling results, as collected by the *Callgrind* profiler when the algorithm is run on the images shown in Figures 2 and 3. Note that optimized code was profiled, so some functions have been automatically inlined by the compiler, and don't show up in the results.

The two columns show the *inclusive* and *exclusive* time spent in some functions.

- Inclusive time measures the total time spent in a function, *including* time spent executing subfunctions.
- Exclusive time measures the total time spent in a function, *excluding* time spent executing subfunctions.

The results show that more than 97% of the total CPU execution time is spent in `skeletonize`. This function is the "hot zone" of the application, and therefore is the function I will try to parallelize.

## 1.2 Complexity analysis

To calculate the complexity of the application, I need to know the complexity of all functions. Table 2 shows the complexity of all functions.

| Function | Complexity | Explanation |
|---|---|---|
| `black_neighbors_around` | $\mathcal{O}(1)$ | Straight-line arithmetic |
| `is_outside_image` | $\mathcal{O}(1)$ | Straight-line arithmetic |
| `P[1-9]_f` | $\mathcal{O}(1)$ | Straight-line arithmetic |
| `swap_bitmaps` | $\mathcal{O}(1)$ | Pointer swap |
| `wb_transitions_around` | $\mathcal{O}(1)$ | Straight-line arithmetic |
| `are_identical_bitmaps` | $\mathcal{O}(M \cdot N)$ | Iterates over all pixels |
| `skeletonize_pass` | $\mathcal{O}(M \cdot N)$ | Iterates over all pixels |
| `skeletonize` | $\geq \mathcal{O}(M \cdot N)$ | Iterates until image stops changing |

Table 2: Function complexities

The main issue for determining the application's complexity lies in the fact that `skeletonize`'s ending condition depends on the input image. The algorithm is data-dependant, and it isn't possible to give an accurate estimate of the complexity.

However, I have experimentally observed that the number of iterations is proportional to

$$\frac{1}{2} \times \max(w_b, h_b),$$

where $w_b$ and $h_b$ represent the width and height of the "thickest" black zone in the image, respectively.

Therefore, the overall algorithms's complexity is $\mathcal{O}(M \cdot N \cdot \max(w_b, h_b))$.

## 1.3 Theoretical maximum speedup

Amdahl's law says that if $P$ is the proportion of a program that can be made parallel, and $(1-P)$ is the proportion that cannot be parallelized, then the maximum speedup that can be obtained by using $N$ processors is

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

The part of my application that I want to parallelize is `skeletonize`, which takes roughly 97% of the total execution time. By substituting this value in the equation, I get

$$S(N) = \frac{1}{(1-0.97) + \frac{0.97}{N}}$$

To find the theoretical maximum speedup, I can assume that the problem is "infinitely" parallelizable by setting $N$ to $\infty$.

$$S(N) = \frac{1}{(1-0.97)} = 33.33$$

Therefore, the theoretical maximum speedup that can be achieved is around 33.33.

### 1.4 Reference machine

The actual speedup obtained depends on the reference machine measurements will be performed on. I was granted access to one of LACALs servers for this project. Its specifications are listed in Table 3.

| Device | Specifications | Cores | Speed |
|---|---|---|---|
| CPU | Intel Core i7-2600K | 4 | 3.4 GHz |
| GPU | Nvidia GeForce GTX 580 | 512 | 772 MHz |
| Bus | PCIe 2.0 $\times$ 16 | - | 8 GB/s |

Table 3: Reference machine specifications

All CPU & GPU implementations were compiled with `-O3` to take advantage of any compile-time optimizations. The project was built with the tools contained in CUDA SDK v5.0.

## 2 Parallelization solutions

As stated earlier, I will parallelize the `skeletonize` function, which calculates a new image by iterating over all its pixels and applying a thinning operation. The only parallelizable parts of this function are `skeletonize_pass`, and `are_identical_bitmaps`, as all the other ones are straight-line arithmetic, and are $\mathcal{O}(1)$.

In order to map the problem to the GPU, I decided to assign each pixel of the image to one thread which will be responsible for computing the next value of that single pixel.

Note that the `while` loop of the calling `skeletonize` function cannot be part of a CUDA kernel. The reason is that the image is going to be split into multiple blocks for the computation, and that CUDA does not currently support synchronizing threads across multiple blocks.

If we suppose the `while` loop was inside the CUDA kernel, then it is possible for threads of one block to already be on their *second* pixel computation, whereas other blocks haven't finished their *first* pixel computation. At the moment, there are only two ways to synchronize threads across multiple blocks:

1. Atomically use a variable in device global memory to check if every thread has finished one pass of their thinning opreation. This is very slow, as many threads will be accessing the same global memory location.
2. Terminate the kernel and launch it again. This is easier to do and is faster. The CUDA runtime guarantees that all blocks have terminated upon kernel termination. Additionally, multiple posts on Nvidia forums claim that a kernel invocation is lightweight for the host. I will use this method.

Note that all CUDA binaries produced in this project take 2 arguments as input:

1. Horizonatal block dimension
2. Vertical block dimension

The grid dimensions are computed from the provided block dimensions above in order to cover the entire image.

CUDA devices have multiple different memories, each with different characteristics. To obtain a good speedup, one must select the appropriate memory for their application. I will present 4 parallel solutions, each *generally* faster than the previous by leveraging the different CUDA memories.

## 2.1 GPU 1

### 2.1.1 Idea

This first parallel implementation consists of simply transforming `skeletonize_pass` into a CUDA kernel. This is straight-forward and can be implemented by only using the CUDA device's global memory.

### 2.1.2 Analysis

This implementation suffers from a high amount of transfer time between the host and the device. Indeed, at every iteration, both the `src` and `dst` bitmaps are transferred back to the host.

Figure 4 shows the timing diagram of $GPU_1$ during 1 iteration of the `while` loop in the `skeletonize` function (taken from the Nvidia Visual Profiler) when the application was run on a 8000 $\times$ 8000 *full* black image. Table 4 summarizes the results.

A 100 % black image was chosen in order to have high amount of arithmetic done in both the CPU and GPU versions.

The gaps between the different events shown on the timing diagram are due to the CUDA API overhead.

As one can see, around 41 % of the total time spent in each iteration is used to transfer data to the host. This issue brings me to my next implementation.

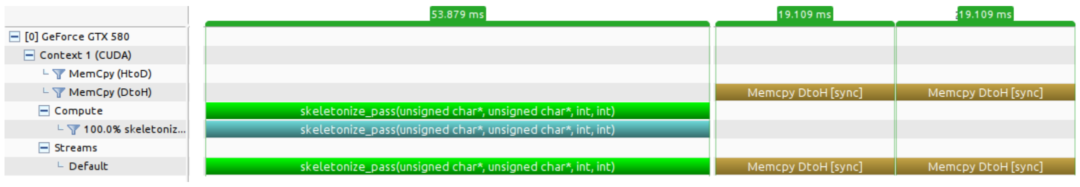


Figure 4: $GPU_1$ timing diagram during 1 iteration ($8000 \times 8000$ full black image).

| Function | Time spent (%) | Time spent (ms) |
|---|---|---|
| `skeletonize_pass` | 57.95 | 53.879 |
| `cudaMemcpyDeviceToHost` | 41.10 | 38.218 |
| `Total` | 99.04 | 92.983 |

Table 4: $GPU_1$ profiling data during 1 iteration ($8000 \times 8000$ full black image).

## 2.2 GPU 2

### 2.2.1 Idea

This second solution is identical to the one presented in 2.1, but it addresses the huge transfer time issue between the host and the device.

The reason data was being sent back to the host was for it to decide whether the 2 images have changed during the iteration. We can avoid sending the data to the host by making the GPU compute the images' equality.

This can be accomplished by performing a *reduction* algorithm on the GPU to compute a single flag from the content of both images. This flag would indicate whether the images have changed during the iteration. All that would need to be done at the end of an iteration would be to transfer this flag back to the host, in order for it to decide whether to relaunch the `skeletonize_pass` kernel.

The `skeletonize_pass` kernel remains unchanged, and only code for performing the reduction is added. To do this, 2 new kernels are added:

- `pixel_equality`, which tests the equality of each element in the `src` and `dst` bitmaps, and stores this information in another array of the same size as the images, `equ`. Concretely, it calculates `equ[i] = (src[i] == dst[i])`.
- `and_reduction`, which reads the `equ` array and performs a binary AND reduction over it.

Note that the reduction code uses a small chunk of shared memory to perform the algorithm before writing the result back to global memory. Also, depending on the size of the image, the reduction may not be finished in 1 iteration, so multiple successive reductions are necessary to obtain the final flag. This could be an issue, as the CUDA API overhead can add up, however, each reduction reduces the size of the output by `(blockDim.x * blockDim.y)`, so few passes are necessary (for example, with a $16 \times 16$ block size, the image gets reduced 256 times each iteration).

### 2.2.2 Analysis

Figure 5 shows the timing diagram of $GPU_2$ during 1 iteration of the `while` loop. Table 5 summarizes the results.

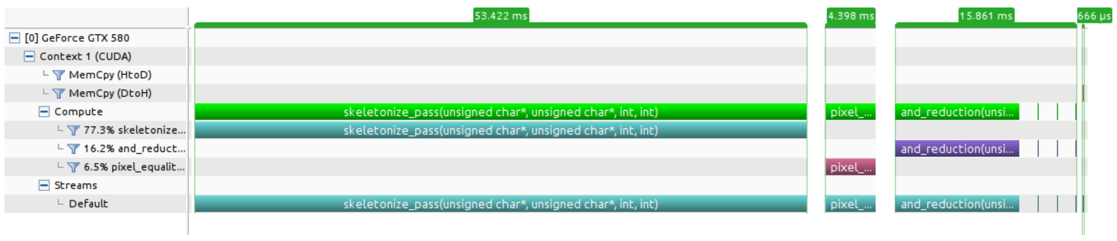


Figure 5: $GPU_2$ timing diagram during 1 iteration ($8000 \times 8000$ full black image).

| Function | Time spent (%) | Time spent (ms) |
|---|---|---|
| `skeletonize_pass` | 68.77 | 53.422 |
| `pixel_equality` | 5.66 | 4.398 |
| `and_reduction` | 20.42 | 15.861 |
| `cudaMemcpyDeviceToHost` | 0.21 | 0.161 |
| `Total` | 95.06 | 77.682 |

Table 5: $GPU_2$ profiling data during 1 iteration ($8000 \times 8000$ full black image).

This version solves the memory copy issue, as transfers between then host and the device only account for 0.2 % of the total time spent during an iteration.

However, this version still suffers from one issue: global memory contention. Each pixel in `dst` requires the values of 15 pixels in `src` in order to be computed. Therefore, each thread in a block is going to somehow load these 15 values from global memory. Additionally, these pixel valus are not merely read a single time, but are accessed very often. Indeed, neighboring threads will also load most of the same values, resulting in even more global memory accesses. This CUDA kernel is very likely to suffer from latency issues due to contention in global memory.

## 2.3 GPU 3

### 2.3.1 Idea

The idea here is to load the pixels that lie in a block's tile to its *shared* memory. All threads in the block will heavily use the contents of the shared memory to perform their computation.

The reduction kernel already loads data into shared memory, so the issue is in `skeletonize_pass`. In the `and_reduction` kernel, every unique data element is read once. As such, reductions don't have high arithmetic intensity.

However, each thread in `skeletonize_pass` reads the value of 15 neighboring pixels multiple times. There is a good opportunity to avoid many global memory reads by using shared memory. In this version, the host allocates `(blockDim.x + 3) * (blockDim.y + 3)` bytes of shared memory for each tile in `skeletonize_pass`. The `+3` is due to the surrounding pixel values that are also needed for this tile. Each thread in the "center" of the tile is responsible for loading 1 pixel value into shared memory, but the threads on the boundaries load 3, 4, 6, or 9 pixels depending on their position (corner, border, ...).

### 2.3.2 Analysis

Figure 6 shows the timing diagram of $GPU_3$ during 1 iteration of the `while` loop. Table 6 summarizes the results.

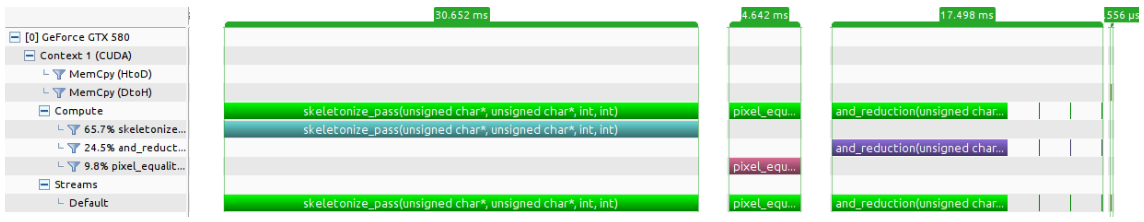


Figure 6: $GPU_3$ timing diagram during 1 iteration ($8000 \times 8000$ full black image).

| Function | Time spent (%) | Time spent (ms) |
|---|---|---|
| `skeletonize_pass` | 53.37 | 30.652 |
| `pixel_equality` | 8.08 | 4.642 |
| `and_reduction` | 30.47 | 17.498 |
| `cudaMemcpyDeviceToHost` | 0.31 | 0.179 |
| `Total` | 92.23 | 57.435 |

Table 6: $GPU_3$ profiling data during 1 iteration ($8000 \times 8000$ full black image).

Compared to $GPU_2$, this version reduces the time spent in `skeletonize_pass` from 53.422 ms to 30.652 ms, which is a 42.6 % improvement. It is not as I expected though. I was expecting a reduction of around 93 % (15 *times* less loads from global memory), but it seems that the arithmetic is not enough to hide so much global memory latency. In future work, one should give more work to each thread to hide more of the global memory access latency.

There is not much more that can be done in this version to speed up `skeletonize_pass`. However, by looking at the time spent in % in Table 6, we can see that `and_reduction` is starting to take a more important percentage of the total execution time (30.47 % vs 20.42 % in $GPU_2$).

This brings me to my final implementation.

## 2.4 GPU 4

### 2.4.1 Idea

The state of the code until now is that `skeletonize_pass` calculates the future value of each pixel, `pixel_equality` computes an array `equ[i] = (src[i] == dst[i])`, and `and_reduction` reduces `equ` to a single value.

The issue is that `pixel_equality` and `and_reduction` are kernels, so they are forced to load data from global memory, and cannot use any shared memory loaded by `skeletonize_pass`.

However, `skeletonize_pass` has everything the first iteration of `and_reduction` needs, namely

- `src` in shared memory
- Future value of `dst` in a register

If we could somehow do the first iteration of the reduction with these shared memory values, we could avoid having to compute the `pixel_equality` kernel completely, and would get rid of the big first chunk of `and_reduction`, which is the biggest part. The subsequent iterations of `and_reduction` are small, and are only affected by the CUDA API overhead.

In order to do this, the host must allocate more shared memory on the device. Now, it allocates `(((blockDim.x + 3) * (blockDim.y + 3)) + (blockDim.x * blockDim.y))` bytes of shared memory, which is enough to hold the values of `src`, and of `equ` for this tile.

At the end of `skeletonize_pass`, `equ` is set from the values of `src` and `dst` stored in shared memory. Then, I perform the first iteration of the reduction on the *shared* value of `equ`. This operation lengthens `skeletonize_pass` compared to $GPU_3$, but it avoids more global memory loads.

However, we are not done yet. We are only using `equ` at the end of `skeletonize_pass`, but we could also use it at the beginning to mimic an optimization done on the CPU algorithm. The CPU's greatest optimization is that it has the ability to skip any pixels that are white, so it can rapidly get to the areas of the image where work is to be done. On the GPU, after `src` is loaded into shared memory, we can use `equ` to check if the tile only contains white pixels. If it is the case, then nothing needs to be done, and we can skip the skeletonization part of the tile entirely.

This optimization is also included in $GPU_4$, and has the advantage that the number of global memory stores decreases after each iteration, making each one slightly faster than the previous (more on this in section 3).

### 2.4.2 Analysis

Figure 7 shows the timing diagram of $GPU_4$ during 1 iteration of the `while` loop. Table 7 summarizes the results.

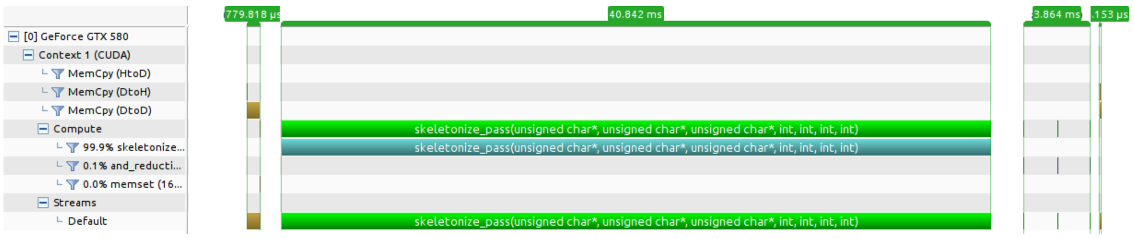

[0] GeForce GTX 580
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
MemCpy (DtoD)
Compute
99.9% skeletonize...
0.1% and_reducti...
0.0% memset (16...
Streams
Default
779.818 μs
40.842 ms
3.864 ms
153 μs
skeletonize_pass(unsigned char*, unsigned char*, unsigned char*, int, int, int, int)
skeletonize_pass(unsigned char*, unsigned char*, unsigned char*, int, int, int, int)
skeletonize_pass(unsigned char*, unsigned char*, unsigned char*, int, int, int, int)

Figure 7: $GPU_4$ timing diagram during 1 iteration ($8000 \times 8000$ full black image).

As we can see, the total time spent in `and_reduction` is greatly reduced, and the overall execution time of an iteration has decreased by roughly 14.24 % (depends on the image).

| Function | Time spent (%) | Time spent (ms) |
|---|---|---|
| `cudaMemcpyDeviceToDevice` | 1.58 | 0.780 |
| `cudaMemset` | 0.22 | 0.106 |
| `skeletonize_pass` | 82.92 | 40.842 |
| `and_reduction` | 7.84 | 3.864 |
| `cudaMemcpyDeviceToHost` | 0.22 | 0.106 |
| `Total` | 92.78 | 49.255 |

Table 7: $GPU_4$ profiling data during 1 iteration (8000 × 8000 full black image).

## 3 Detailed Analysis

I will give an analysis for the solution given in 2.4.

Figure 8a shows that the duration of each iteration of `skeletonize_pass` reduces by around 30 % due to the reduced global memory operations.

Figure 8b shows that the number of global stores decreases until 0. However, the number of global loads stays constant, as the image must be read at each iteration.

Figure 8d shows that once the shared memory is loaded, its use decreases over time as the image becomes more and more white. This is normal, as the skeletonization algorithm is not run on a white block of pixels.

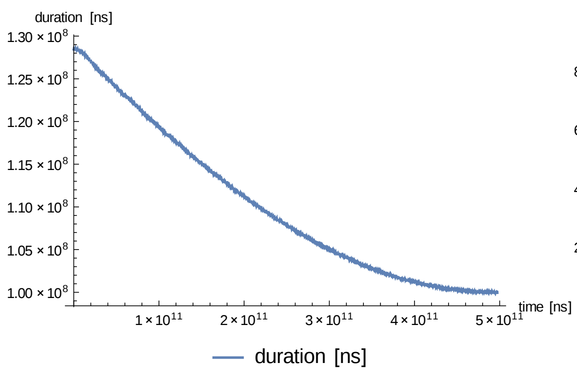
Figure 9 shows how the execution time varies when different block sizes are used. Due to how I implemented the algorithm, it isn't possible to specify grid sizes, so it isn't possible to artificially limit the number of cores used in the application. The only thing that can be done is vary the number of threads that are contained in a block. We can see that the sweet spot for the execution time is when 128 threads are used per block.

Finally, Figure 10 summarizes the speedups obtained across all implementations over a wide range of images. It is interesting to see that the speedup is indeed correlated with the width and height of the "thickest" black zone of the image. The images are available in the CD that came with the report.

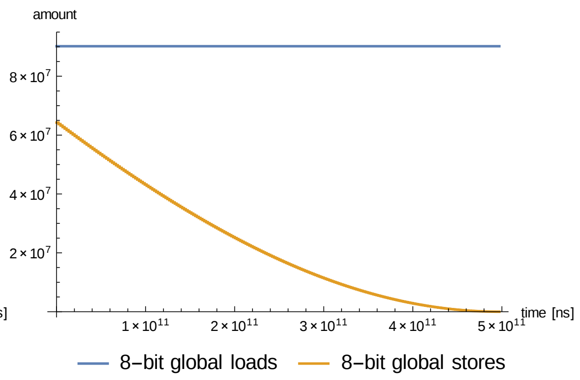
The closest we get to the maximum theoretical speedup is around 29 times, which is close to the 33.3 times we predicted, but only when an image has many black pixels to give an equivalent "sense" of work to both the CPU and the GPU implementations.

We can put this in contrast by also analyzing the speedup obtained if the serial algorithm didn't skip any white pixels. If I profile the serial code with such an algorithm, I see that more than 99.2 % of the total execution time is spent in `skeletonize`, which can theoretically give a maximum speedup of 125.

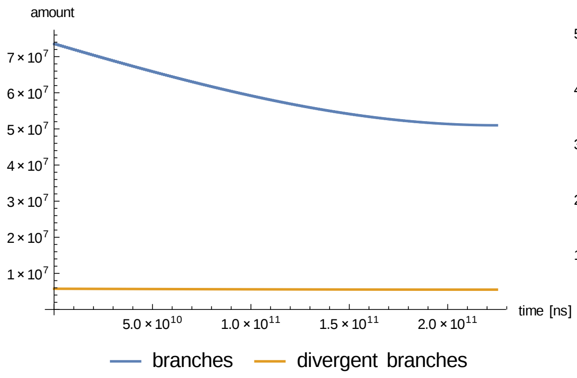
Figure 11 shows some results with an unoptimized serial algorithm, and we can clearly see that the speedup is much closer to the theoretical result, as the problem almost scales perfectly with the size of the image.



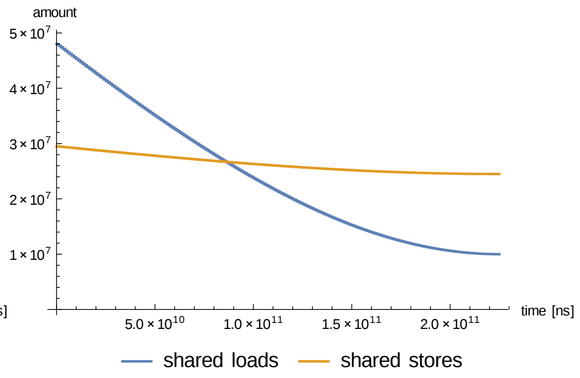
(a) Duration of 1 iteration over time.



(b) Number of global loads/stores over time.



(c) Number of normal/divergent branches over time.



(d) Number of shared loads/stores over time.

Figure 8: Evolution of each iteration of skeletonize_pass kernel over time.

## 4 Conclusion

In section 1.3, we calculated that the maximum speedup possible is 33.3. This computation was based upon a skeletonization algorithm where each pixel is *treated* irrespective of its color. However, in practice, some optimizations can be done to speed up the computation, the most prominent of which is skipping any white pixels in the serial version of the code. This optimization can distort the real speedup obtained, and it was easily visible in the benchmarks of Figure 10. As we saw in Figure 11, the theoretical speedups are much easier to achieve when the serial code and the parallel code truly follow the exact same algorithm.

However, algorithms are made to be fast, so it is cheating to compare a parallel implementation against an intentionally slow serial one. Indeed, parallel implementations have to be tested against their fastest serial equivalent to measure true speedups.

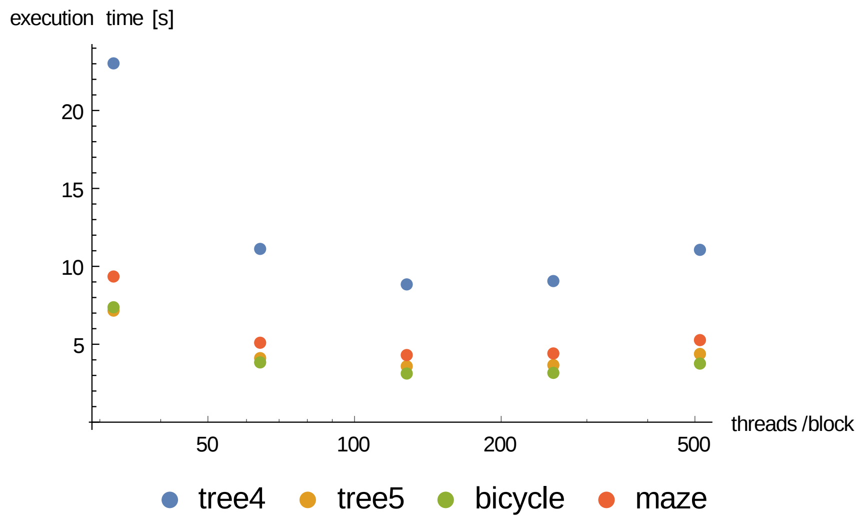


Figure 9: $GPU_4$ execution time with various block sizes.

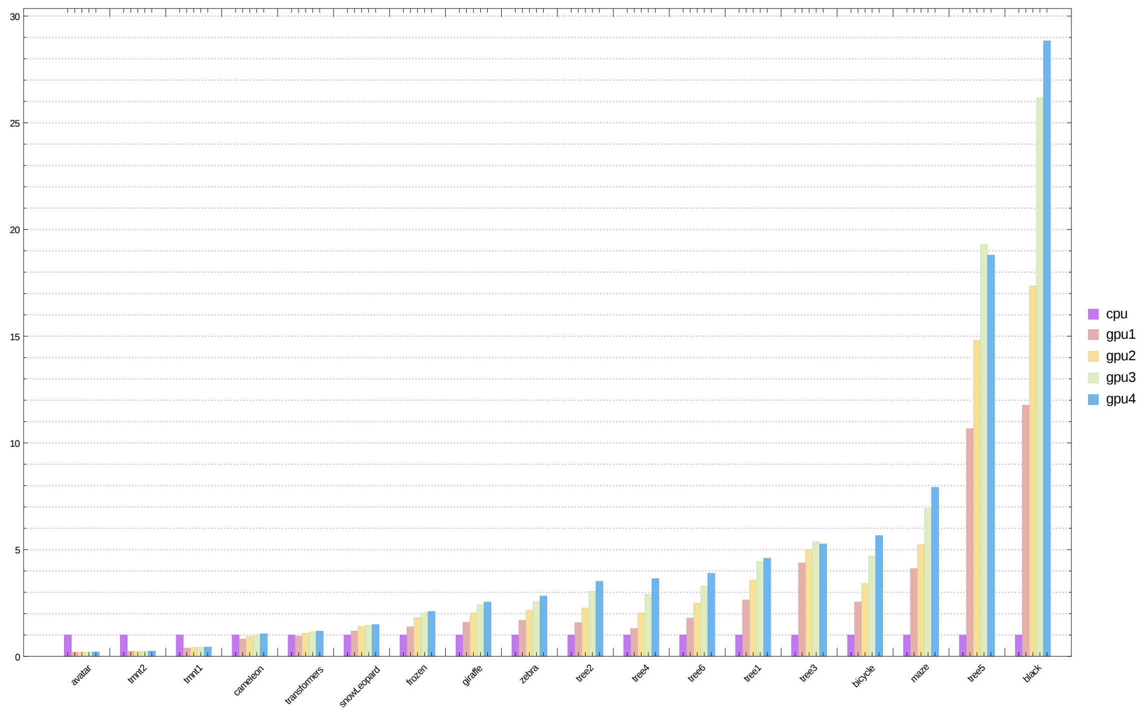


Figure 10: Speedup benchmarks with various images, from slowest to fastest.

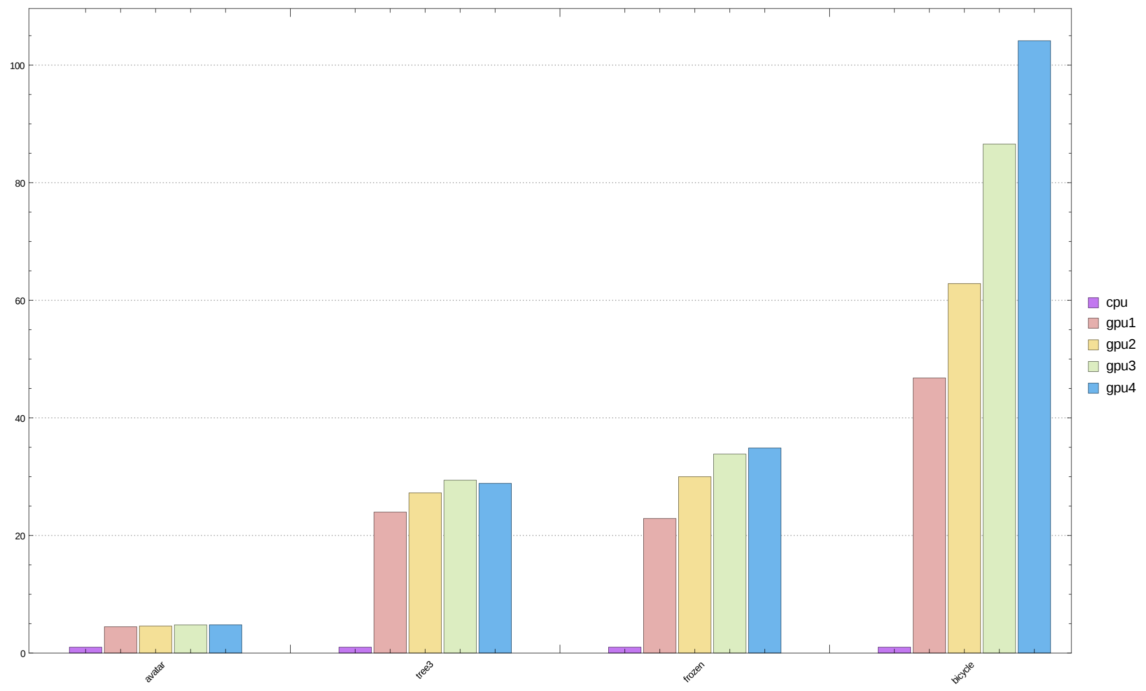


Figure 11: Speedup benchmarks with various images, from slowest to fastest (no skipping white pixels in serial code).