SUPERPIXELS - ECE1782 Project Report

Eugene Osovetsky, Felipe Sander Pereira Clark, Harika Gaggara

April 2020

1 Introduction

This project consisted of implementing and optimizing the Simple Linear Iterative Clustering (SLIC) Superpixels Algorithm in CUDA, whereby image pixels are grouped based on distance and color proximity into macro-blocks (not necessarily rectangular) called superpixels. The algorithm can be used either artistically or as a preprocessing step in computer vision algorithms. Fig. 1 shows an example. In the segmented image (Fig 1b) the superpixels clearly separate the butterfly, flower and background. With subsequent processing steps another algorithm could potentially isolate and label these subjects. This was especially attractive to Felipe Clark, as the company he works for is releasing a product with an imaging sensor soon and could benefit from an object detection algorithm.

Technically the algorithm is a variation of the K-Means algorithm. It is described in [1] and consists of:

- An initialization stage: converts image to LAB color space and initializes data structures;
- The core algorithm: assigns a "superpixel ID" to each pixel as described below;
- Post-processing: enforces continuity of superpixels and visual post-processing (e.g. drawing borders).

Many steps of the core algorithm can be described in terms of actions on each pixel. This makes it an "embarrassingly parallel" problem well-suited for GPUs.

2 Core Algorithm and Data Structures

As per the project proposal document, the focus was on optimizing the core algorithm, ignoring the pre/post processing steps and any memory transfer overhead. The core algorithm works with the following data structures:

pix: The image itself, indexed by **x** and **y** coordinates. A size of 4096×2048 pixels was chosen[1]. There are 3 channels per pixel (L/A/B color values, each fits in 1 byte).

spx: The superpixel grid, indexed by **i** and **j** coordinates. The size of each nominal superpixel was fixed to 128×128 pixels (as explained below), thus **spx** is 32×16 based on the standard image size defined above. Each superpixel must contain, at a minimum, data on its mean L/A/B colors channels and mean **x** and **y** coordinates.

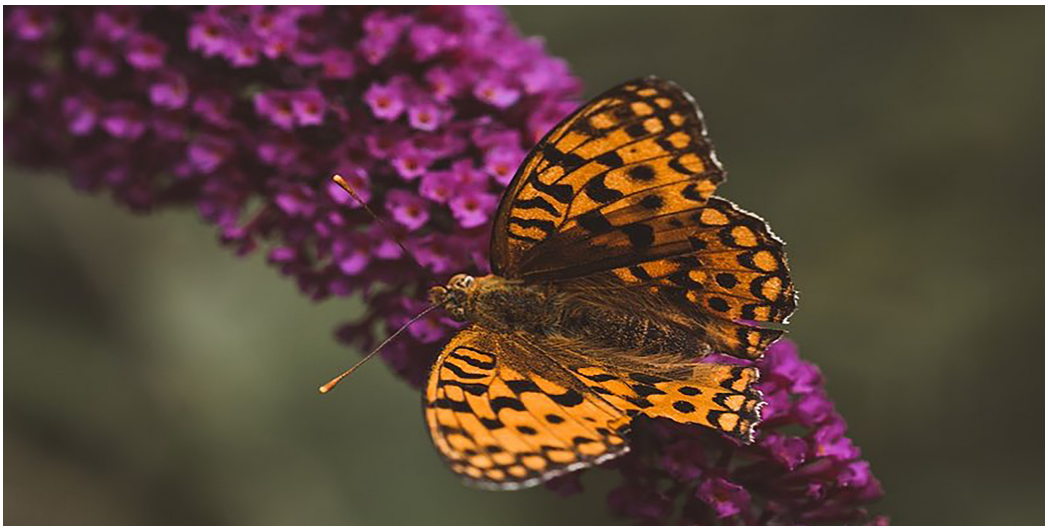
own: A 4096×2048 ownership grid, indexed by **x** and **y** coordinates, indicating for each pixel the **i** and **j** coordinates of the superpixel that "owns" it.

It is important for the discussion that follows to clarify the difference between pixel, superpixel and nominal superpixel, as well as the concept of "window size". If an image of 4096×2048 pixels is divided into a grid of 128×128 pixels squares, each resulting square is a nominal superpixel. For example, a pixel at $x = 1000$, $y = 500$ is in the nominal superpixel with $i = 1000/128 = 7$ and $j = 500/128 = 3$ (note the integer division).

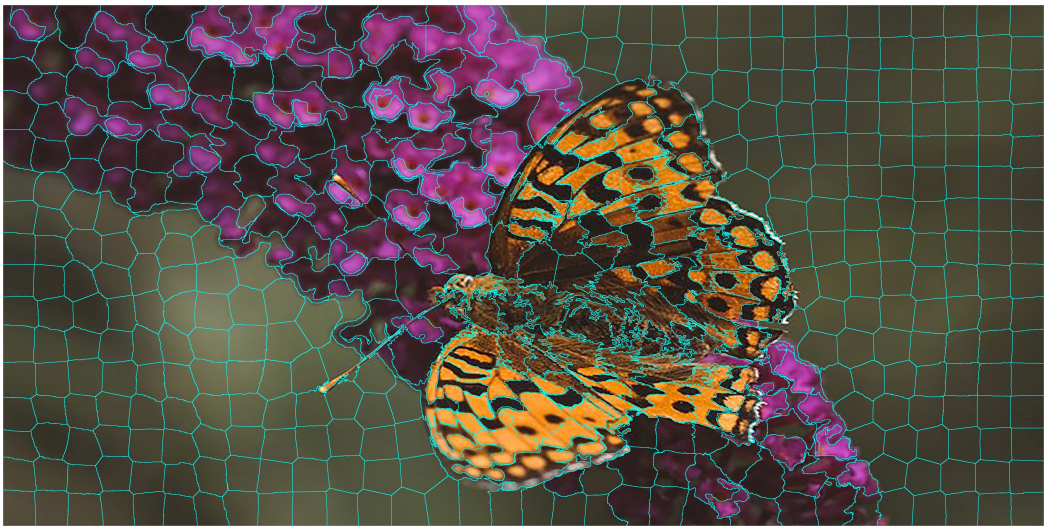
Despite nominal superpixels being initially square, the algorithm can of course produce superpixels of varied shape: each pixel may be "owned" by either its nominal superpixel, or by one of its neighbors in a certain window. Similar to the CPU implementation given in [4], this project uses a window size of 1, meaning that **i** and **j** can differ by at most 1 from the nominal superpixel. That is, a pixel at $x = 1000$, $y = 500$ is owned by some superpixel whose **i** is either 6, 7 or 8, and whose **j** is either 2, 3 or 4.

Initially, **own** is initialized based on nominal superpixels (i.e. just a square grid). The core algorithm then works as follows:

[1] This dimension was chosen because it matches approximately the 4K resolution, which is the current standard for commercial high resolution image.



(a) Original image



(b) Segmented image

(c) Artistically processed image (each superpixel colored with mean color)

Figure 1: Applications of SLIC

Figure 2: The Goldfish - 4096 × 2048 benchmark image.

1. For each superpixel, update **spx**: Compute the mean L/A/B color values and mean **x** and **y** coordinates of all the pixels it owns;
2. For each pixel, update **own**: Compute the superpixel that is "closest" to that pixel, according to a particular distance metric[2], and mark the pixel as "owned" by that superpixel. Only superpixels within the permitted window size are considered.

These are repeated multiple times (until convergence or for a fixed number of steps). This was the motivation for optimizing the core algorithm for this project: these steps run in a loop and thus likely have a significant effect on total system performance. For benchmarking 300 or 10 iterations were used[3] with Fig. 2.

3 Kernels

This work's GPU implementation has 4 kernels to accomplish the work of the 2 steps described in Section 2:

Step 1:

- **k_sum**: Compute, for each superpixel, the sum of all L/A/B/x/y values of the pixels it owns, and count the number of owned pixels. For this, L/A/B/**x**/**y**/**count** accumulators were added to the **spx** data structure;
- **k_averaging**: For each superpixel, divide accumulators by **count** to compute the mean L/A/B/**x**/**y** values;
- **k_reset**: Reset accumulators back to 0.

Step 2:

- **k_ownership**: Compute, for each pixel, its "closest" superpixel (as described in detail in Section 2), and make the pixel "owned" by that superpixel.

Of these, only, **k_sum** and **k_ownership** contribute non-trivially to the run time (as they run over a 4096 × 2048 pixel grid as opposed to the 32 × 16 superpixel grid, and they do non-trivial work), so optimization efforts were focused only on these 2 kernels for the purposes of this project.

[2]Euclidean distance in a 5-dimensional space (L/A/B color and x/y coordinates), with a scaling factor for the spatial squared distance vs. color squared distance. Computing the "closest" pixel does not require any evaluation of square roots.

[3]SLIC paper [1] suggests 10 steps. In this project 300 iterations were used for accurate average kernel timing, and 10 when detailed **nvprof** profiling was required, and for GPU/CPU comparisons. While it would be more accurate to measure on a wide range of images as well as multiple runs of 10 iterations, given that the algorithm run time has no strong dependence on the input data (other than perhaps for specially-constructed images), the accuracy should be sufficient.

4 Optimizations

4.1 Summing Kernel

The initial implementation was completely naive - 1 thread per pixel which did the following:

- Read, from **pix**, the pixel's L/A/B color values;
- Read, from **own**, the pixel's owning superpixel;
- Write to **spx**: Add to that superpixel's L/A/B/**x**/**y**/**count** accumulators (the current pixel's L/A/B values, **x**/**y** coordinates and 1 for the **count**).

Since many pixels can belong to the same superpixel, many threads can thus write to the same **spx**, so the additions were **atomicAdd()** calls, resulting in one atomic add per pixel. Average kernel run time with this approach (and naively chosen 32×32 block size) was 20.724ms.[4]

4.1.1 Optimization 1: Use shared memory

To optimize this kernel[5], the path laid out in lectures 4 and 5 of this course (optimizing the array sum problem) was roughly followed, but with important differences. Most notably:

- 2D grid instead of a 1D array;
- The presence of multiple heterogeneous "channels" to add (L/A/B/**x**/**y**/**count**);
- Most importantly, the notion of "ownership", whereby not all items but only a subset is summed.

The first optimization idea, then, was to perform the summation in shared memory and write it out (with an atomicAdd) only once at the end - roughly equivalent to "reduction #1" on slide 34, lecture 4. The concept of "ownership", however, presented a significant complication: It is not enough to just sum all the pixels; a separate sum per superpixel is needed. For example, taking the L color channel, the procedure involves the sum of all L values of all pixels belonging to superpixel at $\mathbf{i} = 0/\mathbf{j} = 0$, another sum for the superpixel at $\mathbf{i} = 0/\mathbf{j} = 1$, etc (and similarly for the other channels). This initially seemed intractable, given the limited amount of shared memory.

However, each pixel may only be owned by one of 9 superpixels (its nominal superpixel or its 8 neighbors, i.e. a 3×3 square on the **i**/**j** grid). Thus, if the algorithm guarantees that a given thread block will process pixels belonging to only one nominal superpixel, this makes the problem tractable: all there is to do is to compute 9 sums for each channel. For instance, if a given thread block handles only pixels nominally in $\mathbf{i} = 10/\mathbf{j} = 5$, then there is only the need to perform sums for $\mathbf{i} = 9..11/\mathbf{j} = 4..6$. (See Fig. 3)

Therefore, an important restriction was established: **each thread block only processes pixels belonging to one nominal superpixel**. Thus, no dimension of the thread block can exceed 128 (nominal superpixel size), and thread block dimensions must evenly divide the superpixel size (e.g., a thread block 48 high is not allowed). With this assumption in place, the algorithm's structure becomes :

- Allocate an array in shared memory of size [6][3][3][**Y**][**X**], where **Y** is **blockDim.y** and **X** is **blockDim.x**. This memory will contain the data to be summed, per pixel, for the 6 channels (L/A/B/**x**/**y**/**count**) and 3×3 neighbors. Indexing by **x** last ensures that a warp accesses consecutive addresses (at least in the case of same channel and neighbor) and thus reduces the chance of bank conflicts;
- Initialize this array to 0 (parallelize over the **y** and **x** indices);
- For each pixel (again, parallelized over **y** and **x** indices):

[4] All times are average kernel times as measured by nvprof. Average is being reported instead of minimum run times since there is some, albeit small, data dependence in the run time, and each iteration runs on slightly different data.

[5] Summing kernel optimizations can be seen in "opt_X.cu" files in the project's code repository at https://github.com/felipe-clark/gpu_slic - for example, opt_1.cu for optimization 1. Note that these files only contains kernel definitions - some optimizations also require changes to block sizes or global constants, as noted in the text. Note that the code has not been edited for clarity and is intended as supplementary material only. Note that variable, type, kernel and other function names may disagree between the code and this report - they have been edited for clarity in the report.

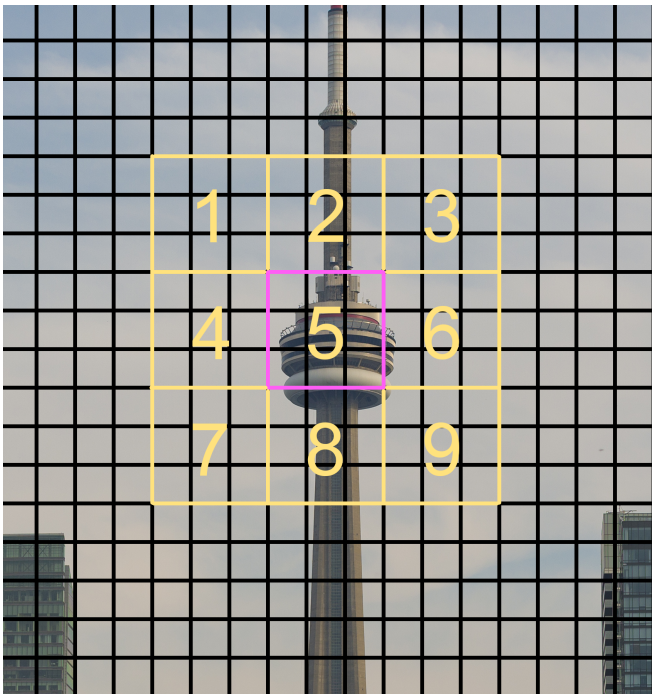


Figure 3: Neighboring superpixels for a block of pixels (pixel dimensions greatly exaggerated). Pixels in the pink nominal superpixel may only belong to one of 9 numbered nominal superpixels.

  - Determine, by reading the **own** data structure, which superpixel it belongs to. There are 9 possibilities, corresponding to neighbor indices [0][0] (above and left), [0][1] (above), [0][2] (above and right), [1][0] (left), etc., all the way to [2][2] (bottom and right);
  - Read the pixel from the **pix** data structure;
  - Write to shared memory at the location [**c**][**ny**][**nx**][**y**][**x**], where [**ny**][**nx**] are the neighbor indices described above. Channels **c** = 0/1/2 get the pixels L/A/B colors respectively, channel **c** = 3 gets the value 1 (pixel count), and channels **c** = 4/5 get the pixels **x** and **y** coordinates, respectively;

- After a __syncthreads(), treat each [**c**][**ny**][**nx**] chunk of the shared memory array as a separate summing problem. Sum over [**y**][**x**] of each such chunk;
- For a [**c**][**ny**][**nx**] chunk, determine the correct superpixel. For example, if the nominal superpixel is **i** = 10/**j** = 5, and the [**ny**][**nx**] coordinates are [2][0] (bottom left), then the actual superpixel is **i** = 9/**j** = 6;
- For each channel, atomically add the [**c**][**ny**][**nx**] sum computed above to the accumulators for the superpixel (**spx** data structure) at **i**/**j** coordinates identified above.

The summing was done in 2 steps: first horizontally over [**x**] (parallelizing over **y**), using the basic approach in the course slides, with the sums ending up at [**c**][**ny**][**nx**][**y**][0], and then vertically over [**y**], with the final sums ending up at [**c**][**ny**][**nx**][0][0].

For the initial block size, the choice was **32** × **4** (32 for coalesced access to **own** and **pix**, and 4 due to shared memory limitations)[6]. Thus, the kernel reached an atomic add per 128 pixels, instead of per pixel as before. This yielded a run time of 24.746ms, worse than the naive approach, but opened up new optimization possibilities.

4.1.2 Optimization 2: Use one summation instead of separate x/y summations

2-step summation was eliminated, instead treating the entire $[y][x]$ space as one big $32 \times 4 = 128$-value array. Run time of 24.916ms (worse than before, possibly noise) but simplified the code for further optimization.

[6]Maximum value of any sum is the sum for the **x** channel (L/A/B colors only go up to 255, but **x** coordinate can be up to 4095). With $32 \times 4 = 128$ pixels, maximum value is $4095 \times 128 = 524,160$, requiring a full **int** (4 bytes). Thus, shared memory size is (6 channels) × (3 × 3 neighbors) × (32 × 4 pixels) × (4 bytes) = 27,648 bytes. Doubling the block size to 32 × 8 would require 55,296 bytes, exceeding the available SMEM size.

4.1.3 Optimization 3: Reducing thread divergence

This optimization is analogous to "Reduction #2" from the course notes (Lecture 4, Slide 43), aiming to reduce thread divergence in the summation code. This did not yield improvement (run time of 27.410ms).

4.1.4 Optimization 4: Simple reduction of potential SMEM bank conflicts

Continuing with known techniques, the SMEM memory bank conflict elimination technique from Lecture 4 was implemented, Slide 48 ("Reduction #3"), yielding the first improvement - 20.725ms.

4.1.5 Optimization 5: Reusing threads for inner loops

The summation algorithm from the lecture slides throws away many threads at each iteration of its "step" loop. However, in SLIC multiple sums are computed by using an inner loop:

- Step = 64: Keep only 64 threads, each thread has an inner loop of $6 \times 3 \times 3 = 54$ (channels $\times$ neighbors);
- Step = 32: Keep only 32 threads, each thread has an inner loop of 54;
- ...
- Step = 1: Keep only 1 thread, which performs an inner loop of 54.

Clearly this is inefficient. Instead of throwing threads away, they were reused. Instead of a full inner loop:

```
const int numberOfSums = 6 * 3 * 3; // 54 sums (6 channels, 9 neighbors)
for (unsigned int step = arraySize/2; step > 0; step /= 2) {
    if (threadID < step) {
        for (int whichSum = 0; whichSum < numberOfSums; ++whichSum) {
            sum[whichSum][threadID] += sum[whichSum][threadID + step];
        }
    }
    __syncthreads();
}
```

the following was done: calculate the number of "thread groups" and reduce the inner loop accordingly. For example, when **step** is 64, there are 2 groups of 64 threads, and thus can cut the inner loop down to ceil(54/2) = 27. Similarly, at **step** = 32, there are 4 groups of 32 threads, so an inner loop of ceil(54/4) = 14.[7]

```
const int numberOfSums = 6 * 3 * 3; // 54 sums (6 channels, 9 neighbors)
for (unsigned int step = arraySize/2; step > 0; step /= 2) {
    int location = threadID % step;
    int groupID = threadID / step;
    int numGroups = arraySize / step;
    int maxLoopIndex = ceil(numberOfSums / numGroups);

    for (int index=0; index<maxLoopIndex; index++) {
        int whichSum = index * numGroups + groupID;
        if (whichSum >= numberOfSums) continue;
        sum[whichSum][location] += sum[whichSum][location + step];
    }
    __syncthreads();
}
```

This optimization yielded a very significant improvement, bringing the run time down to 7.379ms.

[7]The ceil function is not actually used in the code - everything is done through integer operations.

4.1.6 Optimization 6: More threads for inner loops

More threads for the [**c**][**ny**][**nx**] inner loops improved performance, so even more threads were added. I.e. still processed only a 32×4 block of pixels (keeping the shared memory requirements the same), but requesting a 32×8 (or larger) block of threads. During shared memory zero-initialization and then initialization from GMEM the additional threads are simply ignored[8]. During the summation stage the threads would be used as described in the previous optimization, to reduce the need for inner loops. This yielded the following results:

- 32×8 (2 threads per pixel): 7.266ms (slight improvement);
- 32×16 (4 threads per pixel): 7.867ms (worse);
- 32×32 (8 threads per pixel, threads per block limit): 10.886ms (much worse).

As can be seen, run time was improved to 7.266ms at the $2\times$ level. The slowdown at higher levels makes sense: benefits gained from parallelizing inner loops are offset by increased thread divergence elsewhere.

4.1.7 Optimization 6A: "short" instead of "int" for SMEM array

Continuing with the theme of increasing the number of threads, recall that the factor that initially limited the algorithm to a 32×4 block was the available shared memory size. Changing the SMEM array to an **unsigned short** instead of an **int** allowed for a 32×8 block. However, this required summing *offsets* of **x**/**y** coordinates from the nominal superpixel, instead of the full **x**/**y** coordinates, since otherwise the sums would not fit in 2 bytes[9]. Coordinate offset sums were adjusted to full sums adding writing to the GMEM **spx** accumulators.

Of course, even larger block sizes could be used, as long as the total pixels being summed stayed the same (see Section 4.1.6 above). The performance was measured for the following block sizes:

- 32×8 (1 thread per pixel): **5.122ms** (significant improvement);
- 32×16 (2 threads per pixel): **5.518ms**;
- 32×32 (4 threads per pixel, threads per block limit): **7.515ms**.

As can be seen above, very significant gains come from this optimization: run time of 5.122ms , but the previous optimization in Section 4.1.6 (more threads than pixels) no longer makes sense in this scenario.

4.1.8 Optimization 7: Parallelizing GMEM atomic adds loop

The last step of the kernel, $6 \times 3 \times 3 = 54$ atomic adds of the partial sums back to GMEM **spx** accumulators, was paralellized over the first 54 threads (with the remaining threads being idle). Run time improved to 4.843ms.

4.1.9 Optimization 8: Padding / memory access

The initial step (zero-initialization of the SMEM array) had no bank conflicts, since the last index of the array is **threadIdx.x**, and so a warp accesses different banks. There were worries that using **short** instead of **int** may introduce bank conflicts (two adjacent threads can access two adjacent **short**s in the same bank), but a simple experiment with **nvprof** showed this does not happen.

Similarly, SMEM initialization from GMEM should have been relatively safe from conflicts - although here, the use of **short** could have introduced issues, since two threads in a warp may write to different "neighbor coordinates" [**ny**][**nx**] [10]. This is addressed in a further optimization.

The next issue was ensuring coalesced reads from GMEM from the **pix** data structure during SMEM initialization. Pixels were read consecutively (a row of 32 per warp) so access should have been coalesced. The structure is 3 bytes long (L/A/B color values). The concerns were:

[8] They could be used for the inner loops during the SMEM zero-initialization, but doing so would require an extra __syncthreads call. They could also be used during SMEM initialization from GMEM to parallelize over the channels, but doing so would lead to highly divergent code since the channels are heterogenous - e.g. initializing the pixels color is very different from initializing its count or coordinate. Thus these ideas were not implemented at this stage.

[9] Coordinate offsets have a maximum value of 127 (nominal superpixel size), LAB colors have a larger maximum value of 255 so they become the limiting factor. $255 * (32 \times 8)$ fits in an **unsigned short**.

[10] E.g., thread 4 may see a "top left neighbor" pixel and write to [0][**0**][**0**][8][4] in the [**c**][**ny**][**nx**][**y**][**x**] array, and thread 5 may see a "bottom right neighbor" pixel and write to [0][2][2][8][5], with both accesses going to the same bank (since it is an array of **short** and the last index goes up to 32).

- Will reading L, then A, then B generate 3 separate memory accesses?
- Will the fact that the data structure is not an even 32 bytes cause a slowdown due to misaligned reads?

Here a crucial mistake was made when attempting to mitigate these worries. The initial code was roughly:

```
// d_pix_data is a pointer to the \textbf{pix} data structure,
// which is a 3-byte struct with three ‘‘char’’ members
// (l/a/b for the three color channels)

smem[0][ny][nx][y][x] = d_pix_data[pix_index].l;
smem[1][ny][nx][y][x] = d_pix_data[pix_index].a;
smem[2][ny][nx][y][x] = d_pix_data[pix_index].b;
```

Worried that it would generate 3 separate GMEM load instructions, the following code change was attempted:

```
pix pd = d_pix_data[pix_index]; // Copy entire struct locally
smem[0][ny][nx][y][x] = pd.l;
smem[1][ny][nx][y][x] = pd.a;
smem[2][ny][nx][y][x] = pd.b;
```

This did not change the run time, so it was assumed that the compiler was smart enough to issue just 1 load instruction for both code snippets. In fact, the opposite was true: it generated 3 load instructions in both cases. Adding an extra padding byte to **pix** to ensure aligned access did not change the run time either, so this change was reverted[11]. In hindsight, this should have been another clue to reexamine the previous assumption, but at the time this was not realized. Similarly, for the **own** data structure with 2 members, **i** and **j**, it was assumed (again, erroneously) that reading them would only require 1 load instruction.

Thus, from this point onward, the assumption was made that reads from both **pix** and **own** were perfectly coalesced and perfectly efficient[12], as reads were always of consecutive pixels in the **x** direction in multiples of 32, until the mistake was discovered in Section 4.1.19.

4.1.10 Optimization 9: SMEM Bank Conflicts Revisited

The problem of reducing bank conflicts was approached empirically using the **nvprof** tool, since it was difficult to discern a pattern in memory access. (Adjacent pixels can belong to different superpixels and so different [**ny**][**nx**] indices, making the pattern unpredictable; Secondly, the dimensions of the top SMEM array indices, [6][3][3], were not powers of two, making it more difficult to reason about access). Resizing the SMEM array (last [Y][X] indices) resulted in conflict counts in Table 1. Going from an [8][32] array to [8][35] reduced conflicts substantially and reduced the run time to 3.778ms.

4.1.11 Optimization 10A: Reading multiple GMEM pixels at once

An optimization similar to “Reductions 4/5/6” in the notes (slides 11-16, lecture 5) was attempted. Each thread initialized SMEM from multiple pixels in GMEM instead of just one. This was complicated by 3 factors:

1. Each pixel may belong to a different superpixel. Thus, it is not enough for a single thread to simply add the values from multiple GMEM pixels before writing a single value to SMEM. Instead, the correct superpixel (i.e. the correct SMEM location) must be deduced for each pixel, and a “+=” operator (i.e. both an SMEM read and write) must be used at each pixel after the first one;
2. With a 2D pixel array, there is the choice of reading additional pixels horizontally or vertically. Horizontally a warp already produces coalesced access[13], thus reading additional pixels vertically made more sense;

[11] This was done to save an extra processing step converting a 3-byte-per-pixel LAB image on the host to a 4-byte representation on the device. Even though this would not affect the metrics being optimized in this project, it would affect the run time of a real system.

[12] Or, at least, **pix** could be easily made perfectly efficient by adding a padding byte to the 3 existing L/A/B bytes, and **own** could be made perfectly efficient by making **i** and **j** into **short** instead of int - see Section 4.1.15. These changes ensure that **own** and **pix** are exactly 32 bytes per pixel.

[13] Or nearly coalesced, see Section 4.1.9.

	X=32	X=33	X=34	X=35	X=36	X=37
Y=8	15.962 7.328	2.537 1.278	2.328 1.034	0.767 0.351	6.162 2.727	0.767 0.643
Y=9	6.162 2.724	1.177 0.962	1.800 1.185	3.045 1.861	2.909 1.527	2.849 1.568
Y=10	15.961 7.324					
Y=11	6.162 2.723					

Table 1: Bank conflict results, in millions of conflicts (top number is loads, bottom number is stores). Empty cells not tested (as they were not promising)

3. With a 32×8 block, reading 2 pixels (or more) per thread, total number of pixels is at least $32 \times 8 \times 2 = 512$. The sum for the L/A/B color channels 512×256 no longer fits in a **short**. To address this, the optimization from Section 4.1.7 was reverted (changed **short** back to **int**, and changed to a 32×4 block).[14]

   The following results were obtained:

   - 32×4 block, read 2 pixels per thread: **3.091ms**;
   - 32×4 block, read 4 pixels per thread: **1.719ms**;
   - 32×4 block, read 8 pixels per thread: **1.161ms**;
   - 32×4 block, read 16 pixels per thread: **0.848ms**;
   - 32×4 block, read 32 pixels per thread: **0.798ms**.

This was the maximum possible value vertical block size (4) because the pixels per thread cannot exceed the nominal superpixel size (128) due to the constraint that all pixels in a block must belong to the same nominal superpixel. This optimization yielded excellent results, with a best run time of 0.798ms.

4.1.12 Optimization 10B: Even more GMEM reads (smaller vertical block size)

The limiting factor in Section 4.1.11 optimization was the vertical block size times the number of pixels per thread. The block size was reduced to 32×2 to go even further:

- 32×2 block, read 32 pixels per thread: **0.727ms**;
- 32×2 block, read 64 pixels per thread: **0.689ms**.

This improved the run time to 0.689ms.

4.1.13 Optimization 10C: Even smaller vertical block size

Taking the idea of Section 4.1.12 further, blocks were changed to 32×1 blocks:

- 32×1 block, read 64 pixels per thread: **0.681ms**;
- 32×1 block, read 128 pixels per thread: **0.650ms**.

Overall, this improved the run time to 0.650ms. Having the **y** dimension be 1 allowed the removal of some code that dealt with **y** calculations, which likely had a beneficial effect[15].

[14] These changes may have invalidated the analysis in optimization 9 (SMEM bank conflicts), but the X array size was kept at 35 (to be revised in the following sections). The **x**/**y** offset calculation code was kept for now.

[15] On the other hand, having only 32 threads per block meant that it was no longer possible to completely parallelize the 54 atomic adds (see Section 4.1.8) - this introduced a bit of extra complexity in optimization Section 4.1.13, which was later removed in Section 4.1.14.

4.1.14 Optimization 10D: Larger horizontal block size

After the optimization in Section 4.1.12, the block size became very small, and thus per-block overhead relatively large. So experiments with larger horizontal block sizes were attempted:

- 64×1 block (array of $64 + 3 = 67$ for bank conflict avoidance), 128 vert. pixels per thread: **0.652ms**;
- 128×1 block (array of $128 + 3 = 131$ for bank conflict avoidance), 128 vert. pixels per thread: **0.650ms**;
- 128×1 block (array of $128 + 3 = 131$ for bank conflict avoidance), 64 vert. pixels per thread: **0.695ms**.

There was no improvement, but 128×1 blocks and 128 vertical pixels per thread were kept as this likely had more room for further optimization. With these parameters, a thread block reads an entire nominal superpixel - this is the most it can read under the key assumption made previously.

4.1.15 Optimization 11: Optimizing the ownership data structure

Naively, the **own** data structure was initially designed with **int** fields for **i** and **j**. A warp would then need to read $32 \times 2 = 64$ **int** values, resulting in 2 GMEM reads. Since **i** and **j** are very small, an alternative would be:

- **short**: Run time of **0.563ms**;
- **char**: Run time of **0.528ms**.

Overall run time reduction to $\boxed{0.528\text{ms}}$[16].

4.1.16 Optimization 12: SMEM bank conflicts revisited again

SMEM bank conflicts were revisited, with the following results (thousands of conflicts, load/store):

- Array size $\mathbf{X} = 128$: 222/110;
- Array size $\mathbf{X} = 129$: 528/484;
- Array size $\mathbf{X} = 130$: 514/480;
- Array size $\mathbf{X} = 131$: 505/476;
- Array size $\mathbf{X} = 132$: 491/470.

Changing the size of the last array index to 128 (as opposed to 131 used previously) approximately halved the number of bank conflicts, but did not result in an appreciable drop in run time - measured at $\boxed{0.526\text{ms}}$. Based on this, it was determined that further attempts to reduce bank conflicts were likely not worth the effort.

4.1.17 Optimization 13: Eliminating atomic addition

At this point the algorithm had a one-to-one mapping between a block and a nominal superpixel. Therefore, only 9 blocks (3×3) may update values for a given superpixel. Instead of atomically adding to one counter (per channel), 9 counters (per channel) can be allocated in **spx**. It is then possible do a simple write (instead of an atomic add, which involves both a read and a write, as well as synchronization overhead). This increases the run time of the **k_averaging** and **k_reset** kernels (need to sum/reset 9 accumulators) but the impact is negligible.

Surprisingly this optimization did not translate into a measurable difference in run time.

[16] Actually, these reductions likely resulted not from reducing 2 accesses to 1, but from increasing the coalesced nature of each of 2 separate memory accesses - first with all threads in a warp reading **i**, and then all threads in a warp reading **j**. However, this was not know at the time - see optimizations 4.1.9 and 4.1.19 for a full discussion.

4.1.18 Optimization 14: Micro-optimizations

The last two optimizations had nearly no effect - this raised the suspicion that the limits of the hardware were being reached. Before measuring this to confirm, some "micro-optimizations" were attempted:

- Unrolling loops with #pragma unroll;
- Getting rid of multiplication/division where possible, replacing with addition (in loops) or bit shifts;
- Getting rid of the offset math for **x**/**y** summation, no longer needed after Section 4.1.11;
- Eliminating unneeded conditions.

Only beneficial (or at least neutral) changes were kept. Run time reduced to 0.512ms.

4.1.19 Optimization 15: Optimality check and truly coalesced access

At this stage the kernel was profiled and performance was measured against theoretical limits.

Using the **inst_integer nvprof** metric, an average of 191.8 Mln. instructions was reported during the 0.512ms run time (374.6 Bln instr./s). For a baseline, a test kernel (with looped integer operations) was run and achieved 1.9 trillion instructions per second. Thus, the summing kernel is only utilizing about $\frac{1}{5}$ of the available computational resources - not nearly at the limit - so the performance was not compute-bound.

Turning the focus to memory bandwidth, computing the required sums involves the following GMEM access:

- Reading **own** data structure, per pixel: 2 bytes;
- Reading **pix** data structure, per pixel: 3 bytes;
- Writing **spx** data structure, per 128×128 nominal superpixel: 54 sums of 4 bytes each, so (54×4)/(128×128) = 0.0132 bytes per pixel (nearly negligible).

This results in $(2 + 3 + 0.0132)$ bytes of bandwidth required per pixel, which for 4096×2048 pixels is 42.1 million bytes or 40MB. This gives a bandwidth of (40MB / 0.512ms) = **78.3GB/s**, just over a third of the optimal 224 GB/s value. However, **nvprof** with the **gld_throughput** (loads) metric reports 200.4GB/s, as well as 0.6 GB/s with **gst_throughput** (stores), for a total throughput of **201 GB/s**, much closer to optimal.

This presented a paradox: Why is the reported bandwidth so much higher than the calculated one? Upon further research, it was discovered that **gld_throughput** reports the bytes actually read from memory, whether useful or not, which can exceed the actually useful / requested bytes in case of non-coalesced memory access. It was also discovered that the **gld_efficiency**[17] metric in **nvprof** which reports the ratio of useful bytes to loaded bytes (100% when all access is coalesced). This metric was around 35%, which was very surprising given the previous assumption of perfectly-coalesced access (see discussion in Section 4.1.9).

To debug this, the first step was to add back the padding byte to the **pix** data structure, and switching to the 2nd code snippet in optimization Section 4.1.9:

```
pix pd = d_pix_data[pix_index]; // Copy entire struct locally
smem[0][ny][nx][y][x] = pd.l;
smem[1][ny][nx][y][x] = pd.a;
smem[2][ny][nx][y][x] = pd.b;
```

Neither of these helped, so print statements were added for **threadIdx.x** and the pointer value of **d_pix_data** at **pix_index**[18], which showed perfectly coalesced access in the printout - but not in **nvprof**.

Finally, what pinpointed the root cause of the discrepancy was passing the **-ptx** flag to the **nvcc** compiler and directly examining the Assembly-like PTX code running on the GPU. To facilitate the analysis the kernel was greatly simplified, keeping only the **pix** reads, and the truth was unveiled: 3 separate GMEM load instructions per pixel, even with the code snippet above (which seemingly creates a local copy of the **pix** structure).

To get around this compiler limitation, the entire struct was read as an int and then cast back to the struct:

[17] From here on only loads will be discussed, as stores are negligible. There is a gst_efficiency metric for stores.
[18] In C++: **&(d_pix_data[pix_index])**.

```
int pdata = *((int*)(d_pix_data + pix_index));
pix pd = *((pix*)(&pdata));
smem[0][ny][nx][y][x] = pd.l;
smem[1][ny][nx][y][x] = pd.a;
smem[2][ny][nx][y][x] = pd.b;
```

This was done for both **pix** and **own** data structures (after adding a padding byte to **pix** and changing **i** and **j** in **own** to be **short**, undoing some of the optimization from Section 4.1.15).

This, finally, resulted in just one GMEM load per pixel, for each of the **pix** and **own** reads. The profiler reported 100% load efficiency (perfectly coalesced) at 124.5GB/s of memory bandwidth - consistent with loading 8 bytes/pixel, 4 for **pix** and 4 for **own**, i.e $8 \times 4096 \times 2048 = 64$MB in $\sim 0.5ms$, or just under 128GB/s).

However, the run time actually increased to **0.529ms**.

The theory now was that just like in the lecture slides, when doing summation with perfectly coalesced reads from GMEM, reading too many values from GMEM per thread may actually hurt performance. The next idea, then, was to reduce the number of pixels per thread, while simultaneously increasing block size (increasing the efficiency of the SMEM summation algorithm). That is, instead of:

- Read into 128×1 SMEM array, each array slot is the sum of 128 values from GMEM;

  The new goal is to:

- Read into 128×2 SMEM array, each array slot is the sum of 64 values from GMEM.

This reduced the number of GMEM reads per thread, and gave the extremely efficient SMEM summation algorithm a chance to run on 256 values instead of 128 [19]. Note that a [6][3][3][2][128] array of **int** no longer fits in shared memory, so the code was reverted back to using **unsigned short** and using **x**/**y** offsets instead of full coordinates, adding some computational overhead.

The above optimization yielded a run time of **0.434ms** (memory bandwidth of 147 GB/s), indicating a move in the right direction. However, this result could not be used. Recall that with the settings above, the kernel sums over an entire nominal superpixel (block size 128×2, each thread reads 64 vertical pixels, so reading 128×128 pixels in total). Color values can be up to a byte, and $128 \times 128 \times 256$ does not fit in an **unsigned short**.

Thus, this optimization direction was blocked, and no further progress was possible.

4.1.20 Summing Kernel: Final thoughts

The run time of GPU algorithms are typically compute-bound or memory-bound. In this kernel's case, only about one fifth of the compute capability is used, and one half of the memory bandwidth. Here, the limitation is of another nature: the size of shared memory.

With the chosen approach, only $65,536/256 = 256$ pixels can be summed at a time if results are stored in an **unsigned short**. The optimization results in the previous Sections have clearly shown that taking so few pixels at a time is extremely inefficient.

The next largest data type, **int**, is 4 bytes long, and with 48kB of shared memory it is only possible to sum $49,152/(4 \times 6 \times 3 \times 3) = 227$ "items" at a time, with 6 layers and 3×3 superpixel neighbors. Rounding down to the nearest power of 2, gives 128 "items" that one can sum. As the optimizations above have shown, the most efficient thing to do is have each "item" represent the sum of 128 vertical pixels, so that the entire set of 128 items represents 128×128 pixels, the maximum that can be handled in a block (given the key assumption that each block only handles one nominal superpixel).

Thus, performance is bound by the size of shared memory per block at a run time of **0.512ms** to process the entire image (achieving memory bandwidth of 128 GB/s). At a higher compute capability with higher shared memory size, better run time could be achieved (at least as good as 0.434ms as indicated in Section 4.1.19). It is also possible that radically different approaches could achieve better run time (for example, finding a way to somehow relax the constraint of one nominal superpixel per block, or splitting into 2 kernels - one to prepare

[19] Recall that the total pixels added per block cannot exceed 128×128, the nominal superpixel size. Also, if one would like to keep the benefit of the optimization in Section 4.1.17, the total pixels added must exactly match 128×128.

a $6 \times 3 \times 3 \times 4096 \times 2048$ data structure and then simple sum reductions on each of the $6 \times 3 \times 3$ channels, attempting to trade memory for run time). However, these were not explored as part of this project.

4.2 Ownership Kernel

The initial ownership kernel was also naively implemented, with 1 thread per pixel doing the following:

- Read, from **pix**, the pixel's L/A/B color values
- Read the surrounding 9 nominal superpixels' L/A/B and x/y values from **spx**;
- Perform 9 of the 5-dimensional Euclidian distance calculation with each superpixel as the reference;
- Store in **own** the **i/j** coordinates of the superpixel that yields the minimum distance.

Average kernel run time with this approach (and naively chosen 32×32 block size) was 2.622ms.

For the optimization of this kernel a lot of the learning from optimizing the Summing Kernel was transferred. For this reason, the following sections are not as detailed.

4.2.1 Optimization 1: Use shared memory

After making the same key assumption as in the Summing Kernel (that a thread block will never handle more than one nominal superpixel), it was observed that every pixel is "tested" against the exact same nine superpixels. Therefore, as the first optimization, **spx** data was pre-fetched into shared memory instead of loading from global memory for each pixel. This concept is analogous to the one illustrated earlier in Fig. 3, where all the pixels in the pink block have the same neighboring superpixels. This approach yielded a run time of 1.504ms.

4.2.2 Optimization 2: Read pix data as a group

Similar to the Summing Kernel, the initial approach (reading L/A/B colors separately) generated 3 GMEM load instructions, and by using the "casting trick" (see two code snippets in Section 4.1.19) 1 load was achieved, decreasing the run time to 1.497ms.

4.2.3 Optimization 3: Write ownership data as a group

Applying the same concept as in Section 4.2.2, but this time for global memory write instead of read, the ownership data (**i/j** coordinates of the minimum-distance superpixel) can be written to GMEM with just 1 store instruction as follows, achieving a run time of 1.437ms[20]:

```
int mins = min_i << 0 | min_j <<  8;
*(int*)(d_own_data  + pix_index) = mins;
```

4.2.4 Optimization 4: Read 16 pixels per thread

Up to this point blocks of 32×32 threads were being used. However, as noted in Section 4.2.1, an entire nominal superpixel (128×128 pixels) has the exact same nine neighboring superpixels. This means that 16 blocks were accessing **spx** to read the very same data.

To circumvent this redundancy, the blocks were reshaped to 128×8 (the maximum possible threads per block), with each thread processing 16 pixels vertically. Thus, all pixels in a nominal superpixel were processed in a single thread block, eliminating redundant memory accesses. This concept is illustrated by Fig. 4.

This optimization unfortunately made the kernel too big, requiring more registers per block than the GTX980 had to offer. For this reason, one variable had to be converted into volatile to reduce register usage[21]. This lead to a run time of 1.138ms.

[20] Some micro-optimization was also performed at this stage (removing unnecessary conditional code creating thread divergence).

[21] The compiler seemed to unroll the loop over 9 neighbors, and used separate registers for each loop iteration. Marking variables as volatile, as suggested at StackOverflow, reduced register usage.

Figure 4: Using 128×8 blocks to avoid redundant memory access. One thread block (one nominal superpixel) shown. Each Thread Group is a different **threadIdx.y**

4.2.5 Optimization 5: Avoid volatile variables

Minor code changes reduced register usage, avoiding volatile variables and resulting in 1.124ms run time.

4.2.6 Optimization 6: Reduce the number of thread groups

Further exploring variations of the idea developed on Section 4.2.4, the number of thread groups to cover the 128×128 blocks was changed to 128×2, with each thread in a group of 128 threads processing 64 pixels vertically.

This improved the run time to 1.035ms, probably due to the reduction in the overhead of launching blocks.

4.2.7 Optimization 7: Read superpixel data as a group

The overhead of reading **spx** was reduced using the idea of Section 4.2.2, with just 2 GMEM reads per superpixel:

```
  // This reads all 64 bits of the spx data structure
  // in two instructions instead of 5 (one per field).
  int spx_index = j * spx_width + i;
int a = *((int*)(d_spx_data + spx_index)); // L/A/B color bytes and padding
int b = *((int*)(d_spx_data + spx_index) + 1); // x/y coordinates as 2-byte shorts
```

With this change the run time was reduced to 0.964ms.

4.2.8 Optimization 8: Further reduce the number of thread groups

This is a variation of Section 4.2.6. Instead of 128×2 blocks with each thread processing 64 pixels vertically, 128×1 blocks were used, with threads processing 128 pixels vertically. This further reduced the run time to 0.931ms.

4.2.9 Optimization 9: Tweak the horizontal block size

Up to this point it was assumed that the greater the horizontal block size the better. However, after some experimentation it was found that reducing from 128×1 to 64×1 led to better results, 0.910ms, and reducing

further to 32×1 led to 0.905ms. In each case, each thread was still processing 128 pixels vertically. Further horizontal size reduction did not make sense due to coalesced access.

4.2.10 Optimization 10: Floating-Point Math

Further attempts to optimize the performance of this kernel were made. The most notable ones were:

- Trying other combinations of block size and pixels processed at a time;
- Loading all pixels up-front (into SMEM or registers) instead of loading and processing them one at a time;
- Replacing floating point math by integer math;
- Substituting multiplications by bit-shifts.

However, none of these experiments were successful.

It is very likely that no more improvements were possible because the kernel was driving the GPU close to its peak arithmetic performance. According to **nvprof** reports, this kernel reached fully-coalesced global memory stores and loads, and zero shared memory bank conflicts. Most importantly, it shows 1.51 trillion integer operations (as well as about 0.16 trillion floating-point operations[22]) per second in comparison to the GPU's maximum of 1.9 trillion integer operations per second (determined empirically as described in Section 4.1.19).

Given the above, some of the integer math was switched to floating-point math instead (the entire distance calculation for the L/A/B color channels), so that both the integer arithmetic units and the floating-point units on the GPU would split the load. This significantly reduced the run time to 0.716ms. Further conversion to floating point math (e.g. the **x** coordinate channel) did not improve results.

4.2.11 Ownership Kernel: Final Thoughts

The kernel, as discussed above, is very likely bounded by arithmetic performance. The final optimization resulted in 656 Billion integer and 1054 Billion floating point operations per second, for a total of 1.71 trillion ops/sec. It is unclear how the GPU hardware handles simultaneous load on both the integer and floating-point units, but it appears that the 0.716ms value is truly close to the limit, as further tweaking (e.g. increasing block size) did not improve performance.

Each pixel requires 9 distance measurements, each of which involves 5 subtractions (one per L/A/B/x/y channel), 5 multiplications (to square the distance components), 4 addition for the 5 squared components, and a comparison with the minimum distance so far. Thus, 9*(5+5+5) = 135 operations for 4096*2048 pixels, or 1,132 Billion operations. Using the 0.716ms run time, this is 1.58 Trillion operations/second, matching the 1.71 number above very well (some addition operations are likely the result of loop overhead and **y** coordinate math). Memory bandwidth is only at 55 GB/s for loads and 49 GB/s for stores, about half of what is possible. Thus, the conclusion that the kernel is truly arithmetic-limited and cannot improve further.

5 Comparison with the sequential CPU implementation

The CPU implementation of the SLIC algorithm used for comparison is available in [4], and is the implementation of [2] made available by the École Polytechnique Fédérale de Lausanne. This algorithm is the reference SLIC algorithm as designed by its original authors.

The time needed for an Intel Core i7 4930k @ 3.60GHz CPU to run the code respective to the kernels optimized in this work (excluding initialization and connectivity enforcement) for the 4096×2048 pixels Goldfish image was **4562ms**. The GPU version processes the same image in just **71.8ms** that is, 64× faster.

At this point it is natural to ask "has this work reached the maximum possible optimization for this algorithm?" As a point of comparison, [3] introduces the fastest GPU SLIC implementation known to date, and it achieved an 83× speedup when compared to the CPU version.

The fact that the implementation in this work was not as fast does not mean the kernels optimized here have not achieved their maximum performance. Recall from Section 2 that only the core kernels were optimized, and

[22] A floating-point constant to combine color distance with spatial distance in the Euclidian-like 5D distance metric was used.

that host to device and device to host transfers were not optimized. Taking into consideration that from the total 71.8ms a significant portion of 32.7ms is taken just for the memory transfers, the kernels (including those that were not optimized) run in just 39.1ms. Therefore, if this project had optimized all kernels and memory transfers, the overall performance gain would be lower-bounded by $4562/71.8 = 64\times$ and upper-bounded by more than $4562/39.1 = 117\times$.

Therefore, it is fair to say that the optimizations achieved here are comparable to the state of the art gSLIC. Based on **nvprof** results and the above analysis of known GPU limitations, this project's implementation is driving the hardware close to its limit. Another point to consider is that [3] does not disclose which CPU was used in their tests, although they mention the GPU: a GTX TITAN X, which is more powerful than the GTX980 used by the authors of this work. Just to exemplify how these factors can have a significant impact on the measured results, running the same comparison against an Intel Core i7 6600U @ 2.6GHz CPU resulted in a run time of **5023ms**, which would make the presented GPU version $70\times$ faster, and on an Intel Core i5 7200U @ 2.5GHz CPU the speedup is further enlarged to $90\times$.

As a final point to consider, an interesting application of SLIC is to use the pixel clusters to segment the image and track moving subjects. While the CPU implementation would only be able to handle 0.22 4K frames per second, the presented GPU optimized code is able to process 13.92 frames per second at the same resolution if all frames are transferred to the GPU and back to the host individually, and up to 25.55 frames per second if the frames are transferred in larger batches (thus making the memory transfers overhead comparatively smaller). This performance is very suitable for offline processing, and even for some real time applications.

6 Key Learnings

With this project the authors have learned how easy it is to think that one is making progress and further optimizing a kernel when, in actuality, marginal progress is being made while the real bottle neck is not addressed. This learning comes from the experience with the summing kernel, where several iterations were made where the authors thought they were achieving great optimizations, but in fact the key issue of the lack of coalesced access was not caught. Similarly, many memory-oriented optimizations of the Ownership Kernel were attempted, without knowing that the run time is actually bound by arithmetic performance.

Another important learning is that data structures may make the code easier to read, but they must be designed with a strategy in mind to efficiently access their data members. In this project the authors have used data types that were compatible with the problem being solved, but that could also be conveniently aggregated by pointer type cast conversions.

The authors have also learned that while one thread per processed "element" makes kernels easier to understand, this is not necessarily what leads to best performance. In both kernels in this project having threads processing many pixels proved to be a good optimization strategy.

References

[1] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Susstrunk. *SLIC Superpixels*. EPFL, 2010.

[2] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Susstrunk. *SLIC Superpixels Compared to State-of-the-Art Superpixel Methods*. IEEE transactions on pattern analysis and machine intelligence, vol. 34, 2012.

[3] Carl Yuheng Ren, Victor Adrian Prisacariu, and Ian D Reid *gSLICr: SLIC superpixels at over 250Hz*. ArXiv , 2015.

[4] https://www.epfl.ch/labs/ivrl/research/slic-superpixels/