# Simplified Word Sense Performance Improvement Using Spatial Locality

**Ahmed Siddiqui**
ahmednsiddiqui@uvic.ca

**Jordan Kirchner**
jordankirchner@uvic.ca

## ABSTRACT

The simplified Lesk algorithm is a simplified version of Michael Lesk's algorithm for word sense disambiguation devised in 1986 [1]. Given a sentence and a word in that sentence, the algorithm will simply give the best definition (sense) of that word from a provided dictionary by computing the overlap of words between the words in a sense and the surrounding words in the provided sentence. The sense with the maximum overlap is considered to be the best sense.

In this paper, we introduce an optimized version of our implemented simplified Lesk's algorithm. These optimizations were done through the hot-and-cold data usage strategy on our local dictionary and optimizing the temporal locality of our data when computing the overlap of sense definitions and surrounding words in a sentence. The hot-and-cold data usage optimization proved to be the most effective way of optimizing the algorithm as splitting up the dictionary to smaller files categorized by the the first and second letter of the word increased our benchmark speeds by about 31x the speed! Attempts to optimize the algorithm by optimizing temporal locality ended up being ineffective and, in fact, seemed to slow down the runtime by 0.42%. This was most probably due to the additional overhead of using more loops.

## 1. INTRODUCTION

### 1.1 Word Sense Disambiguation

The problem our team needed to solve is called Word Sense Disambiguation. Word Sense Disambiguation is the problem of, given a sentence and a word, find the best definition (sense) of that word in the provided sentence. It is described as follows:

*Algorithm:* Word Sense Disambiguation
*Input:* A sentence and a word in that sentence who's sense the user wants.
*Output:* The best sense from our local dictionary

A word can have multiple senses. The correct sense can usually be easily figured out by people by thinking about the context that the word lies in, which is what the Simplified Lesk's algorithm attempts to do.



### 1.2 Importance of Optimization

Optimizing this algorithm is important as it can run pretty slow due to its heavy memory usage. A simple, small sentence and a word in that sentence took 2.36 seconds. It is pretty clear this speed would be considered unacceptable in most applications. Especially in those circumstances where larger input cases may be used, and speed is of the essence e.g. websites.

### 1.3 Motivation

The reason our team chose this problem to solve was because we are fascinated by the concept of NLP (Natural Language Processing) and were hoping to learn more about the topic by working on this problem. Furthermore, this seemed like a very promising candidate for optimization purposes.

## 2. ALGORITHM DESCRIPTION

Simplified Lesk algorithm works by first querying its local dictionary to get all the senses of the word who's definition we want to find. In our baseline implementation, we had the local dictionary stored as a JSON file. It then goes through all the sense definitions, and for each one, and it computes the overlap of words (intersection) between two sets: the set of words in the sentence (excluding the word we're trying to define) and the set of words in the current sense. The way the algorithm computes overlap is it starts with the first element of set A and then checks for equality with every in set B. Then it goes to the second element of set A and checks for equality, and so on. It does this for all senses, and the sense with the maximum overlap is considered the best sense, and it is outputted as the answer.

## 3. BREAKING THE MEMORY WALL

It is well known that the main bottleneck within the execution of many algorithms nowadays is not dictated solely by the speed of the CPU, but by the rate of which the CPU gains access to the data it needs to work with. This section is dedicated to explaining how we were able to garner a runtime speedup of 31x through the concept of hot and cold data and our attempt at utilizing tiling to further improve execution times.

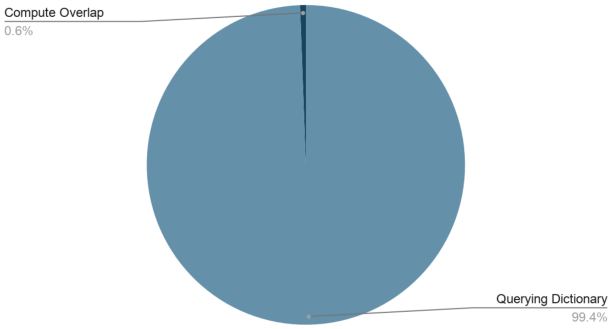



**Figure 1**. Benchmarking each step of the algorithm showed querying the dictionary and computing overlap took up most of the runtime of the algorithm

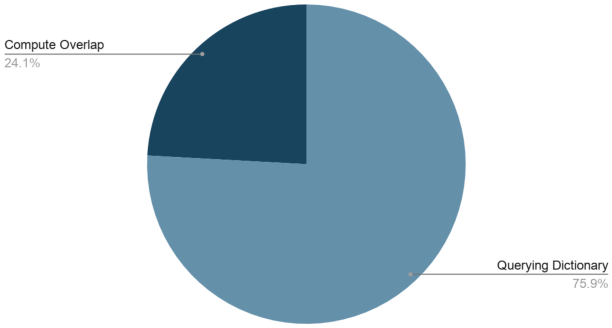



**Figure 2**. Benchmarking each step of the optimized algorithm showed querying the dictionary ended up taking less time than the unoptimized code

### 3.1 Datasets

In order to bolster the claim that our chosen datasets were not cherry-picked for our own evaluations, an arbitrary dictionary from Websters English Dictionary [2] was selected to test upon the word sense-disambiguation algorithm.

### 3.2 Querying the Local Dictionary

In the way of optimizing the algorithm, benchmarking each logical step of the algorithm seemed like a good place to start to see where there is room for improvement. These results of these benchmarks can be found in Figure 1. In our experiments, we found that the step of querying the local dictionary to get all senses of a word took up 95% of the runtime of the algorithm! It was quite apparent that loading in a JSON file as big as our dictionary was slowing down the algorithm quite a bit. As it stood, the program was loading the entire dictionary into its main memory and that process of moving the JSON file from the hard disk to its Random Access Memory was taking a significantly long time in the context of the runtime.

The way our team addressed this issue is by using the hot-and-cold approach of optimizing data access. This means that the program should only concern itself with data that is important to it and it will use frequently (hot data), and it should be concerning itself with data that it will most likely not be using (cold data). This concept seemed very applicable since loading up the entire dictionary is loading up a lot of data that the program is guaranteed not to need.

What we did was split up the dictionary into multiple little files that were named after the first + second letters of that word that is in that file. For example, the word "stock" would be placed in a file called "st.json" and so would other words that start with "st". Where before the program was loading in the entire dictionary to query, it could now just find out the first two letters of the word that it wants to find the definition for and then just query a smaller file.

### 3.3 Computing Overlap

Now that we had fixed the original bottleneck which was querying the dictionary, we had to find the next bottleneck we could optimize. Through our previous experimentation, we knew the next part of the code that would probably be the bottleneck is computing the overlap, so we conducted experiments to confirm this. Figure 2 on this page shows that addressing the dictionary problem reduced its impact on performance by quite a bit, but the next thing to improve was computing the overlap.

One potential problem we found with the current way we were calculating the overlap was we ended up resetting the cache line quite a bit and weren't optimizing the data that was already in the cache line. This can be better illustrated by looking at figures Z and 3. When we chose one element to compare from Set A to the rest of the elements in Set B, and then went to the next element in Set A to start comparing again, we would lose a lot of the information already loaded in the cache lines and would end up having to reset a lot of the data. This is because as we load a new cache line of information in set B in between sets, we lose the previous cache line and would need to reset it.

This could easily be optimized by maximizing the usage of every cache line of information we load in. For example, when comparing elements in set B, instead of going through every element in set B and moving in between cache lines for every element in set A, we stay inside cache lines and optimize its use. This idea is illustrated in figures 4 and 5.

Note that this is not in fact a change to the complexity of the algorithm as the same amount of comparisons are still being made, namely every sense in A against every context in B, so if there were to be a speedup resulting from this alteration it would have to result from a performance improvement in cache hits, not an efficiency improvement in algorithm complexity. This observation is demonstrated in the pseudocode we applied to the process of calculating the overlap, as seen in the naive traversal in Figure 2 and the tiling traversal in Figure 3.

```
1.  For each sense a in set A
2.      For each context b in set B
3.          if a == b:
4.              overlap++
```

**Figure 3**. Pseudocode of our implementation to calculate the overlap between contexts and senses before applying tiling.

```
1.  For each tile t_a in set A
2.      For each tile t_b in set B
3.          For each sense a in tile t_a
4.              For each context b in in tile t_b
5.                  if a == b
6.                      overlap++
```

**Figure 4**. Pseudocode of our implementation to calculate the overlap between contexts and senses after tiling.

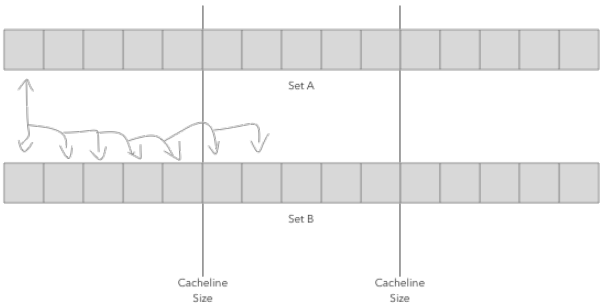



**Figure 5**. Typically, the algorithm, for the first element in set A will compare for equality to every single element in Set B.

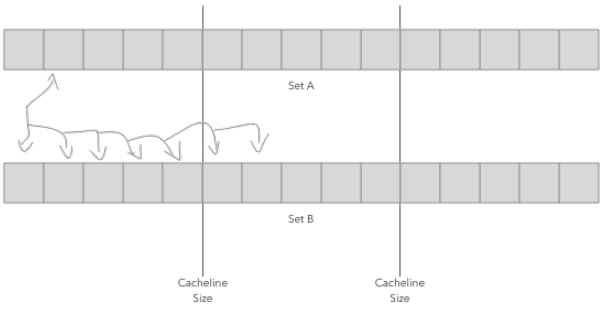



**Figure 6**. Comparing to every single element in set B loses data stored in the first cache line since the second cache line will have replaced the cache line loaded in previously, and so we will have to load that in again.

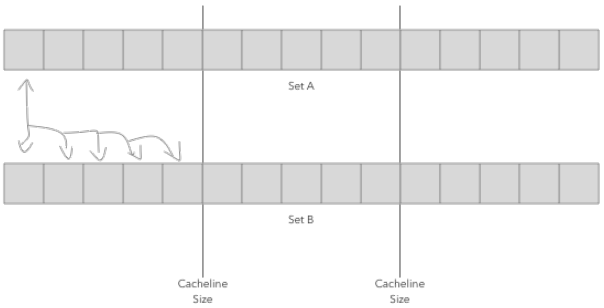



**Figure 7**. The process of tiling with a tile size of 5. At this point, the end of the Set B tile has been reached.

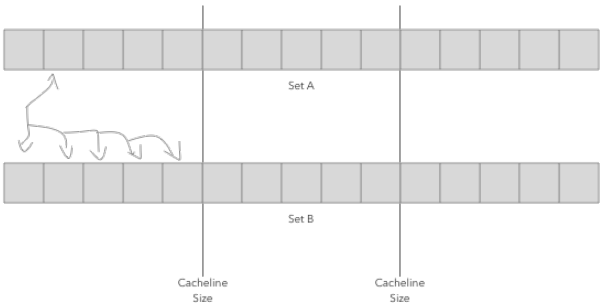



**Figure 8**. The process of tiling with a tile size of 5. Since the first five values of Set B are already loaded into the cache, instead of going to the next tile in set B we go to the next item in Set A, which trades off 1 cache miss for a subsequent 5 hits in Set B.

However, after we swapped from our naive method of iterating over each element in A and B, we noticed in our testing that despite in theory we should be receiving more cache hits due to this procedure, our overall runtime went up by 0.42%. This can be attributed to the fact that although we were getting a speed up do to successive cache hits and not frequently cycling out values that were to be used shortly after, there was as slowdown due to branch prediction failing quite more often. The reason being is that in order to traverse the data in such a way that cache hits are maximized, two additional for loops were inserted into our overlap traversal. This would be especially significant in the innermost loop as it is frequently (the length of a tile) being predicted to jump back to the beginning of the loop, when in fact it has to retreat to one of the outer loops. Because of this, our final optimized version does not include this alteration.

### 3.4 Experimental Results

We conducted a few experiments to see how the runtime changes when each optimization is added to the baseline code and the varying runtimes can be seen in Table 1. As discussed in the previous section, splitting up the dictionary showed the highest effects of optimization by improving the runtime by 31x, while the tiling optimization slowed down the runtime by 0.42%.

| Average Runtime with Varying Optimizations (ms) | | | |
|---|---|---|---|
| Baseline | Tiling | Splitting & Tiling | Splitting |
| 2359 | 2369 | 77 | 75 |

**Table 1**. Experimenting runtimes with different optimizations

These experiments were averaged our over the course of 100 iterations each – testing on the words stock - leaving little doubt that these are accurate values for the averages. Although the runtimes for different words changed, one thing that remained the same was significant improvements of the splitting optimization as opposed to the base version, showing that our program is markedly improved not just for one case, but for seemingly all.

Table 2 shows our results when using perf to examine the resource stalling when running the optimized and unoptimized version of our code. It provides supporting evidence to support our explanation that our optimizations addressed the bottleneck which was the querying of the dictionary. This table demonstrates the memory-boundedness of the unoptimized code.

| Resource Stalling Stats | |
|---|---|
| Unoptimized | Optimized |
| 1,319,068,337 | 21,957,019 |

**Table 2**. Experimenting runtimes with different optimizations

## 4. SPLITTING THE WORKLOAD

Once data was reaching the CPUs fast enough such that there were no major lag times between when the CPU finished working on its current task to its next task, our goal shifted from increasing the speed by which the processor received its data, to distributing the workload across multiple cores. This was achieved through coarse grained parallelism and the use of Single Instruction Multiple Data (SIMD).

### 4.1 Quality of Baseline

For each sense of the word we are trying to find the sense of, for each word that sense contains, there must be one comparison of that word to each word in the sentence in order to actually compute the overlap. As we have explicitly mentioned that this was accomplished in previous sections, it is clear to see that our algorithm – in terms of asymptotic complexity – is as efficient as it can be without sacrificing correctness.

In our initial report, we demonstrated that we had over a 30x speedup over our naive algorithm when using the word "stock" to check for its intended sense within a sentence. Although its impossible to declare that this is the most efficient that we can get, it is safe to say that our run time is not inefficient.

We have also attempted to make adjustments to our algorithm by implementing tiling, but despite witnessing a reduction in all of cache L1, L2, L3, and branch prediction misses with that strategy, the run-time was still slower with it than without it. There must be additional overhead with the inclusion of two extra for loops necessary for tiling that we are unaware of, but as of now we are uncertain as to where that overhead is coming from. The bottom line however is that excluding bit hacks – which we are intending to consider for our next report – there are not many feasible ways to break the memory wall even further than we already have, thus we have concluded that now is a great time to shift our attention to introducing parallel processes.

### 4.2 Parallel Algorithm Description

With having a baseline implementation that was as efficient as possible, the next task was to parallelize the algorithm. This was done by leveraging the OpenMP library to achieve coarse-grained task parallelism. Algorithm 1 describes the algorithm prior to our parallelization efforts.

**Algorithm 1:** Non-parallelized wsd algorithm

```
1  best_overlap = 0;
2  best_sense = "";
3  for curr_sense : all senses of the input word do
4      overlap = compute_overlap(curr_sense,
         surrounding_words);
5      if overlap > best_overlap then
6          best_overlap = overlap
7          best_sense = curr_sense
8      else
9  end
10 return best_sense;
```

What we specifically tried to achieve was to split up the work done for every different sense in computing the overlap between the words in a sense and the surrounding words in the input sentence. By doing so, we were creating different threads to do the work of computing the overlap for each individual sense.

Where we ran into problems of data race was with the $best\_overlap$ and $best\_sense$ variables. Two threads could modify these variables at the same time, and thus we would get incorrect results. How we dealt with this was we created an array of overlaps, the size of which was the number of threads we created, which is equal to the number of senses found for the input word we want to find the definition for.

Afterwards, we compare the overlap computed by each thread and pick the the maximum overlap within the array and pick the appropriate best sense. The resulting, parallelized algorithm can be found in Algorithm 2 below.

Furthermore, the $compute\_overlap$ function is also parallelized using SIMD. The idea of this is simple. Instead of going through Vector A individually and comparing all the other elements from Vector B with the selected

**Algorithm 2:** Parallelized wsd algorithm

```
1  overlaps = [] best_overlap = 0;
2  best_sense = "";
3  // Below For loop is parallelized using OpenMP
   for curr_sense : all senses of the input word do
4  |  overlaps[i] = compute_overlap(curr_sense,
   |    surrounding_words);
5  end
6  for curr_overlap : overlaps do
7  |  if curr_overlap > best_overlap then
8  |  |  best_overlap = curr_overlap;
9  |  |  best_sense = all_senses[index of
   |  |    curr_overlap];
10 |  else
11 end
12 return best_sense;
```

element from Vector A for equality, we use a 128-bit register to load in 4 elements from Vector A and step through each element of Vector B and grab 4 elements to check for equality. The way this is done is further expanded upon later in section 4.6.

### 4.3 Experiment Setup

To test whether our parallel implementation was seeing significant improvements over our previous version, we compared run times between both of them across three words on their respective sentences. These three words – namely "faucet", "stock", and "set" – were chosen to test a wide range of senses for a given word. Faucet merely has one noun sense, stock has slightly over 10 senses (some of which are nouns while others are not), while the behemoth set has a whopping 430 senses.

It should be noted that although coarse grained parallelism should work well in theory with a word like set, which has a plethora of senses, for those such a faucet that have very few (one in this case) senses there should only be an increase in overhead as only one thread can be run in total. Although we allowed the case of *compute_overlap* being ran on a one sense word like faucet to showcase that coarse grained parallelism might be inadequate in these minimal sense cases, we note here that an improvement to this strategy would be to just return the one sense as soon as it is verified that is the only sense, as no other sense can have a higher overlap value if there are no other senses to begin with. There is also another problem whereby if there are two senses being run across two cores, but one is significantly shorter than the other, there will be a large period of idle time in the core that handles the short sense while the other core handles the large sense.

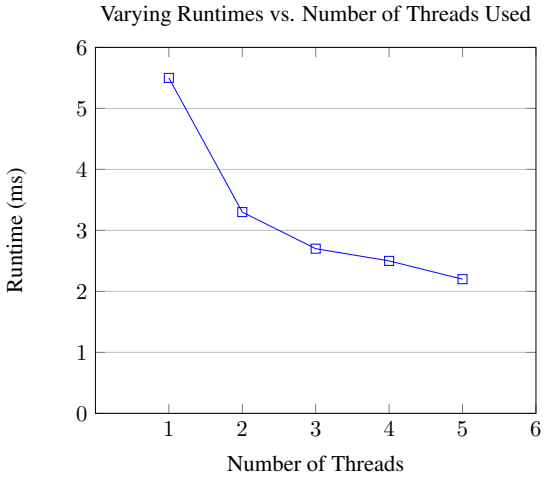
To counter these inefficient cases, for our next report we plan on swapping over to fine grained parallelism so that we can compare each word in each sense against each word in the sentence simultaneously. This should allow for a relevant speedup to occur even with a low number of senses (since we could still run each word within a sense in parallel) and should reduce the amount of idle time within each core (every core would be responsible for relatively equal work).

### 4.4 Parallel Scalability

The following table shows the results of runtimes for different input words with parallelization and no parallelization. It can be seen that words like "Set" end up with relatively bigger improvements for runtime when parallelized than a word like "Stock". This makes sense when you recall that the word "Set" has 430 senses while the word stock has slightly over 10 senses. Therefore the benefits of parallelization end up being more beneficial for words with more senses. While the word faucet in fact seems to slow down with parallelization. This can be explained because faucet has only one sense, and so the overhead that creating a thread causes ends slowing down the runtime compared to its non-parallelized counterpart.

| Runtimes with different input words | | |
|---|---|---|
| Input Word | No Parallelization | Parallelization (2 threads) |
| Set | 13.9 | 3.3 |
| Stock | 7.2 | 1.6 |
| Faucet | 0.4 | 1.3 |

What further illustrates the success of our parallelization efforts is the graph below showing the varying runtimes when the number of threads are increased. In this example, the input word was "set".




### 4.5 Evidence of work efficiency

To validate that our changes in our algorithm accommodate the concept of work efficiency, we noted both the asymptotic changes and the observed work efficiency. Our first revelation was that the *compute_overlap* function (where our focus for parallelization lay) was still doing the same amount of comparisons as the serialized version, namely every word from the sense needed to be compared to every word of its respective sentence.

Our main concern in work efficiency, however, was to check and see whether the overhead generated by transforming the serialized version in the parallelized form would have a significant impact. What little needed

to be changed was to transform the overlap integer into an array and convert the instructions for normal assignments/operations into their respective SIMD variants. Because of this little change in the design, it was not surprising that our work efficiency ratio was exceptionally low, clocking in at 1.02 across 10 executions per setup.

| Work Efficiency of *compute_overlap* (ms) | | |
|---|---|---|
| Parallel | Serialized | Work Efficiency Ratio |
| 4.53555 | 4.43529 | 1.02261 |

**Table 3**. Comparing runtimes of *compute_overlap* with a parallel implementation on one core to its serialized version.

We chose to observe cache L1, L2, L3, and branch misses as a secondary metric to verify that our observed work efficiency was as good as the runtime suggested. Although there was almost a double in the amount of L1 and L3 cache misses, L2 only saw a slight increase, while branch prediction misses went down minimally. As none of these parameters inflated to an extended amount, this also suggests that our work efficiency was kept intact from *compute_overlap*s transformation from serialized to parallelized.

| Work Efficiency of *compute_overlap* (misses) | | | | |
|---|---|---|---|---|
| | L1 | L2 | L3 | Branch |
| Parallel | 23728 | 3855 | 2597 | 34083 |
| Serialized | 13418 | 3118 | 1631 | 38106 |

**Table 4**. Comparing cache and branch prediction misses of *compute_overlap* with a parallel implementation on one core to its serialized version.

### 4.6 Overall analysis and evidence to support design decisions

In order to SIMDize the *compute_overlap* function, we needed to make a fair bit of changes to how the algorithm was currently working to check for equality between the words for a specific sense and the set of surrounding words. The way this is done is by ensuring that every word we deal with has no punctuation and is lowercased, then we hash the word into an integer. This design decision gives us the great benefit of being able to compare integers instead of having to compare strings. This was the first step in order to get SIMD working.

The next step to get SIMD working was to use SIMD intrinsics in order to compare each element with Vector A with every other element from Vector B. This was not as simple as simply loading in 4 integers from Vector A and comparing 4 integers from Vector B because the SIMD function that was provided would only compare one index from Vector A with the element in the same index position in Vector B. Although in order to accomplish the *compute_overlap* functionality, we would need to also compare the same element from Vector A with other elements from Vector B that were not in the same index.

The way we worked around this problem is by first padding both vectors. We padded vector A by ensuring we add 1's to the end of the vector until it becomes a size that is a multiple of 4. Then we padded Vector B and ensured that the Vector will have three 2's at the front and back of the vector.

Then we simply loaded in 4 integers at a time from Vector A, and for those 4 integers, we stepped incrementally through Vector B, and index compare Vector A and Vector B until the end of Vector B, and then we move onto the next 4 integers in Vector A.

In order to see why this algorithm works, it is helpful to look at Figure 8 (where NP represents the hashed string and 1 and 2 represent the paddings) and work through the algorithm. We would first start off by taking off the first 4 elements off of Vector A, then we step incrementally through Vector B, and comparing the loaded in 4 integers from Vector A with each 4 integers step by step of Vector B. If you work through it, you'll see that every NP in vector A will eventually compare with every other NP in Vector B.

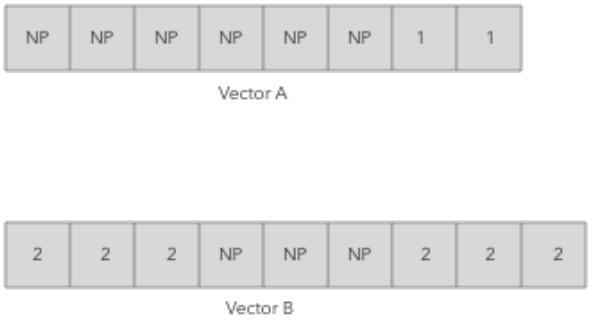



**Figure 9**. Example of how the Vectors SIMD will deal with. NP represents the hashed strings. 1 and 2 represent the padding numbers.

Because of the different padding numbers we used for each vector and because the hash function we are using can not equate strings to 1 or 2, the paddings do not interfere with the results that we come up with for the overlaps.

The Table 2 below shows the results between our baseline serial, optimized version of the *compute_overlap* function and the optimized version with SIMD.

| Average Runtime of *compute_overlap* (ms) | |
|---|---|
| Non-SIMD | SIMD |
| 13.9 | 5.5 |

**Table 5**. Experimenting runtimes with different optimizations.

One last part to touch upon, is the overall cache and branch prediction misses between the versions of our algorithm we have designed so far. One claim we made earlier

was that we assumed that the tiling slowdown was due to branch prediction misses failing more frequently, however this upcoming table clears up that assumption yet raises further questions.

| Overall Cache and Branch Prediction Misses | | | | |
|---|---|---|---|---|
| | L1 | L2 | L3 | Branch |
| OV2 | 90703 | 39497 | 18705 | 116493 |
| OV1 | 196639 | 58292 | 41040 | 172537 |
| UV1 | 5939109 | 1693812 | 1042007 | 9538238 |
| UV1 Tiled | 4585271 | 1562141 | 773946 | 9369812 |

**Table 6**. Comparing L1, L2, L3, and branch prediction misses through the various iterations (Optimized vs unoptimized, version 1 vs version 2) of our project.

Across the board with tiling in place, there appears to be a reduction in all cache and branch prediction misses. As to why tiling is still marginally slower despite seeing improvements in all these areas can once again be surmised to some overhead caused by creating the extra two for loops needed for implementation, but as to what exactly is causing the slowdown could be a subject of investigation for our next report.

### 4.7 Experimental Results

In conclusion to all the other results discussed in this section of the report, the following table shows our final runtime improvements to the $compute\_overlap$ functionality of the algorithm as compared to the serial implementation of the algorithm.

| Average Runtime with Varying Optimizations (ms) | | | |
|---|---|---|---|
| Baseline | SIMD | OpenMP | SIMD & OpenMP |
| 14.3 | 14.2 | 5.5 | 2.15 |

**Table 7**. Experimenting runtimes with different optimizations

The final result of our parallelization effort resulted in an improved runtime by 6.61x for the $compute\_overlap$ functionality.

## 5. MAKING USE OF THE GPU

After successfully demonstrating the utility of SIMD and multi-core parallelism within our Word Sense Disambiguation algorithm by achieving significant speedups over our initial optimized version, our next problem to tackle was optimizing the algorithm using parallelization, this time with the GPU. This was achieved through CUDA, our GeForce GTX 1660 Ti GPU, and the Single Instruction Multiple Threads (SIMT) model.

### 5.1 Discussion of Computational Models

There are many computational models to consider when implementing an algorithm which do not affect the efficiency (how much work needs to be done given an input) of it, but rather the performance (how quickly the work that needs to be done is done). An easy way to picture the difference between efficiency and performance is to imagine needing to arrive at some destination; efficiency would be akin to finding the quickest path to the destination, while performance would be akin to gauging how quickly the mode of transportation is travelling on a particular path. Note that although some paths may be shorter than others, if the only vessel that can operate on that terrain is slow (say a canoe across a lake), then sometimes a longer path with a faster vehicle (say a car around a lake) may be a faster alternative.

With the differences between these two concepts in mind, the question of what computational model to choose for an algorithm becomes a bit easier to answer. There is the traditional sequential model, where instructions are dealt with one at a time in subsequent order. This model is appropriate when the algorithm in question has components that are highly dependent on resources that have a chance to be modified while being used, causing data race conditions that cannot be easily resolved in many other computational paradigms. Although one solution is to lock out the critical section of the code with a mutex, this will result in massive slowdowns to a parallelized algorithm if the locks are opened and closed frequently, completely negating (or more) the gains resulting from the introduction of parallel processes.

When there are no data race concerns or when the critical section of a code will be accessed infrequently, there may be the potential for parallelization of an algorithm: namely multi-core, SIMD, and GPGPU options. Multi-core is independently running the same processes on different sets of data. One thing to note here, is that each core can be working at different points in the same piece of code, so at one instance one core might be executing the middle while another might be running at the beginning of that same segment concurrently. The Single instruction, multiple data (SIMD) model is – as the name implies – one operation applied to multiple data points. Not only will there be less operations needed to be loaded in order to perform work on the data, but instead of being forced to load a certain number of elements one by one, a single instruction can load multiple elements into a SIMD register, thus saving plenty of I/Os.

Finally we arrive at the computation model that this section covers in great detail, which is the GPGPU model. Here Single Instruction Multiple Thread (SIMT) commands are used frequently which can be considered similar to a combination of multi-core and SIMD in one neat package. The gist of how the General-Purpose Graphics Processing Unit (GPGPU) model works is similar to multi-core; many operations run in parallel on different threads, with the stark difference being that every thread needs to be executing the exact same line of code at any given time. It is intuitive to see why this would cause a massive speedup, as for every line of code being read, numerous data points can be worked on, as opposed to a multi-core option which will need to read the same operations over and over again

when operating on each core (due to the fact that they may be operating on different parts of the code). There is one caveat to using SIMT which is that it suffers greatly if there is a need for branching within the code, since each thread is step-locked with one another. If one thread needs to access a portion of code that few others need at a time, the other threads will have to waste time being idle until that segment finishes.

### 5.2 GPGPU Algorithm Description

The challenge of converting the optimized, sequential algorithm into a parallelized SIMT algorithm was in figuring out how to expose the parallelism in our algorithm. We took inspiration from our multi-core parallelization efforts, and focused on the $compute\_overlap$ function. There were a few ways to parallelize this function. The approach we took was to have a coarse-grained parallelization akin to our OpenMP parallelization where we parallelized over the different senses of our input word. The algorithm can be seen below in Algorithm 3.

**Algorithm 3:** Parallelized compute_overlap

```
1 for curr_sense : all senses IN PARALLEL do
2     for sense_token : tokens_of_curr_sense do
3         for context_token :
            tokens_of_context_words do
4             sense_overlap += (sense_token ==
                context_token)
5         end
6     end
7 end
```

To accomplish this we employed a SIMT computational model, by which every thread would be using the same line of code at a given time to compare between a word in a sense to a word in a sentence. The most troublesome aspect of actualizing this was converting a 2-dimensional array of the sense token for every sense into a 1-dimensional array of words for all senses to be fed into the GPU using CUDA. This was implemented by assuming that no sense definition would exceed 300 words (i.e. the nth sense would start at the n * 300th index) which allowed us to easily calculate the offsets for each individual thread to use during the $compute\_overlap$ comparison. As explained previously, these comparisons were independent of each other and relied on no external resources, so race conditions were avoided entirely through this process.If the number of words in a sense was less than 300, then the additional space was padded with a special token to indicate to the inner for loops that their work is done. The data structure can be seen in Figure 10 below.

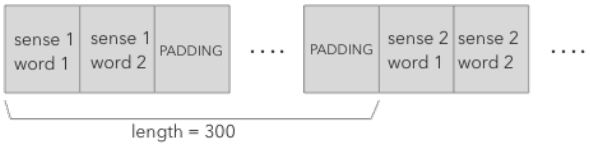



**Figure 10**. The one-dimensional senses data structure for the $compute\_overlap$ CUDA kernel

### 5.3 Evidence of Correctness

In order to demonstrate that our GPGPU implementation did not affect the correctness of our algorithm by introducing errors through SIMT indexing complications, we have ran both our sequential GPGPU algorithms on three words which there were numerous senses within our chosen words, namely "break", "run", and "set." The reason for choosing words with high sense counts is that the odds of randomly picking a correct sense given a particular sentence becomes worse with each additional sense there is to choose from, so if there are matches between the procedural implementation and the GPGPU implementation outputs for these haphazardly chosen words – each with a plethora of senses – then we can be fairly certain that our implementation would be correct given any other input.

Table. 8 verifies that the outputs of the sequential version and SIMT version of $compute\_overlap$ match when ran with these high sense words and their corresponding sentences to be tested with. Additionally, we have provided cases for entirely randomly chosen words (which may or may not have lots of senses) to leverage our claim that these words were not cherry picked.

| Sequential vs GPU Output | | |
|---|---|---|
| Word | Test Sentence | Match? |
| break | That is when we saw the break of dawn | YES |
| run | That team gave us a run for our money | YES |
| set | It was a great day of tennis. Game, set, match | YES |
| evacuate | We were asked to evacuate when the fire alarm started going off for our safety | YES |
| hoax | Covid is a hoax and a lie geared towards controlling the naive public | YES |
| language | That's a really fascinating language you're speaking | YES |
| nice | That lady was awful helpful. Really nice of her to guide us. | YES |
| secret | I'm going to let you in on a secret, but you have to hide it from mother. | YES |

**Table 8**. Verifying our GPU implementation of $compute\_overlap$ has the same output as our sequential algorithm.

### 5.4 Raw GPU Performance

Although it is nice to have some theory as to why a GPU computational model should be much superior to both a sequential and SIMD model, what is even better is to present some stats to back that claim up. As shown in Table 9., we observed that our GPU implementation of $compute\_overlap$ runs at a whopping 34x faster speed than our sequential version, which is not too surprising given how it was stated in theory to be the perfect fit for our algorithm. However what might be a little more unexpected was that it was roughly 5 times faster than our SIMD & OpenMP implementation, which of course was further aided by switching from coarse grained to fine grained parallelism on its own right.

| Runtimes on a GeForce GTX 1660 Ti | | | | |
|---|---|---|---|---|
| Word | Senses | Seq | GPGPU | Scale |
| set | 48 | 15.36 | 0.447433 | 34x |
| stock | 28 | 2.34812 | 0.127743 | 18x |
| faucet | 1 | 0.181334 | 0.04545 | 3x |

**Table 9**. Comparing runtimes between our optimized sequential $compute\_overlap$ to our optimized GPU $compute\_overlap$. Scale indicates the runtime multiplier.

It is not enough for our program to merely see a sufficient increase in speed by itself, however, as there are more metrics to consider such as the ratio of active warps to potential warps (occupancy) and transmission rate of the memory. Table 10 highlights both our reasonable occupancy rate and memory throughput, which were both obtained by manipulating NVIDIA's Nsight tool to monitor the GPU activity of one of UVic's ECS354 lab machines while executing the final version of our code.

| GPU Utilization | |
|---|---|
| Occupancy (%) | Memory Throughput (MB/s) |
| 93.65 | 276.64 |

**Table 10**. Showcasing the GPU utilization of our algorithms third version by means of occupancy and throughput.

Our numbers for work efficiency might be desirable, but they also might be a tad too ideal – or even well beyond ideal. Table 11 presents our recorded number of instructions for our sequential algorithms $compute\_overlap$ function for both our first optimized version and our GPU version. Initially we were comparing strings against each other which resulted in a higher number of executions needed than necessary, but we swapped over to utilizing integer comparisons, which the GPU instruction count is compared to under the ratio header. As the sequential algorithm is supposed to have more or less an ideal number of executed instructions relative to the GPU implementation, we are left with some confusing results that suggest that somehow our latter algorithm is more work efficient than our baseline, as the same amount of work to achieve our result should be the same albeit with more overhead to cover the cost of managing all of the GPUs threads.

| Work Efficiency (# of Instructions) | | | |
|---|---|---|---|
| Seq (str) | Seq (int) | GPU (int) | Ratio |
| 10622428 | 6348262 | 1982046 | 0.31222 |

**Table 11**. Listing the number of instructions required for varying versions of our $compute\_overlap$ function. Ratio indicates our skeptical work efficiency ratio calculation derived from calculating the number of instruction used in our sequential versus GPU program.

After conducting numerous experiments confirming that the number of instructions for both are indeed verifably correct, the explanation that we formulated is that using different compilers for the sequential algorithm and the GPU parallelized algorithm ($gcc$ and $nvcc$ respectively) resulted in the binary files being optimized in different ways and thus resulting in very different number of instructions.

### 5.5 Reproducibility

For those wishing to run our code solely on the CPU, please run our optimized_wsd_v1.cpp following the instructions included in the README.md file located at our code repository (the link to which can be found under section 7 of this paper). Likewise for those wishing to run our code which accelerates the $compute\_overlap$ function by use of the GPU, please run our optimized_wsd_v3.cpp.

### 5.6 Overall analysis and evidence to support design decisions

The majority of the work done in transforming our serialized implementation to a SIMT friendly variant was more a matter of transferring the data to the GPU and back correctly than actually converting data structures to a unique format. Our original two dimensional array that once held a list of sequences of words comprising senses was reformatted into a one dimensional array to allow for easy indexing with CUDA code, but aside from that change there really were not any major overhauls to data structures used.

Of course, the same cannot be said entirely for the algorithm – as a change from sequential to SIMT took place – however the core idea remained untouched, which was that each word in each sense needed to be compared to each word in the example sentence exactly once. As nothing but the inclusion of multiple threads operating during each line of code read within compute overlap was altered, we are confident that the improvements gleaned were a result of the design choices discussed at length within this section.

## 6. CONCLUSION

Through our application of spatial locality (splitting up our dictionary in hot and cold data), we were able to get a running time reduction of around 95%. Although in the end we did not receive much reward for our efforts in applying

temporal locality (iterating through senses and contexts of words by tiles), we are glad to have checked in order to eliminate any doubt as to whether we could have improved the algorithm any further than we did with the knowledge we had up until that point.

We also successfully parallelized our project with the inclusion SIMD as well as using OpenMP to parallelize the $compute\_overlap$ functionality resulting in an improvement of 6.61x improvement in the said functions runtime.

Last but not least, we tapped into the power of the General-Purpose Graphics Processing Unit to further reduce the execution time of our program. This was achieved by applying the concept of SIMT alongside the CUDA programming language to generate a speedup of 34 times faster over the first version of our $compute\_overlap$ function. From the start of our journey to the end, we managed to bring down the runtime of both the overall runtime and the pivotal $compute\_overlap$ method to fractions of what they originally were: at least when our go to testing word "set" was operated on. This goes to show that there is more to programming optimally than just taking asymptotic complexity into account, and reminds us as we were lectured at the beginning of our foray into CSC 485C – not only does the efficiency of an algorithm need to be considered, but also its performance on a specific piece of computer architecture.

## 7. CODE REPOSITORY

`https://github.com/AhmedNSidd/wsd-485c`

## 8. REFERENCES


[1] Lesk's algorithm
`https://en.wikipedia.org/wiki/Lesk_algorithm`

[2] The local dictionary used in our implementation of Lesk's algorithm,
`https://github.com/matthewreagan/WebstersEnglishDictionary/blob/master/dictionary.json`