

# Speeding Up MWE Detection and Minimization Through CUDA Parallelism and Access-Pattern-Aware DP Optimization

Robert Joachim Olivera Encinas  
De La Salle University  
robert\_joachim\_encinas@dlsu.edu.ph

Lanz Kendall Yong Lim  
De La Salle University  
lanz\_kendall\_lim@dlsu.edu.ph

Tyler Justin Hernandez Tan  
De La Salle University  
tyler\_justin\_tan@dlsu.edu.ph

## ABSTRACT

Multiword expressions (MWEs), such as idioms and fixed phrases, are essential for accurate linguistic analysis but remain relatively underexplored in computational research, particularly in the context of parallelization. In this light, this work applies CUDA-based GPU parallelization to accelerate two core stages of MWE processing: dictionary-based lookup table construction and dynamic-programming (DP) segmentation minimization. By parallelizing substring lookup and restructuring DP computation to improve memory-access contiguity, our GPU implementations achieve substantial speedups over sequential CPU baselines. Experiments on short, long, and mixed sentence sets show speedups of up to  $37\times$  for lookup and up to  $7.3\times$  for DP minimization. The results highlight the impact of reducing CPU-GPU transfer overhead and designing memory-friendly access patterns, demonstrating the effectiveness of GPU acceleration for linguistic sequence-processing tasks.

## KEYWORDS

multiword expressions, natural language processing, CUDA, GPU parallelization, computational linguistics

## 1 INTRODUCTION

Multiword expressions (MWEs), such as idioms, phrasal verbs, and fixed phrases (e.g., “looked up” or “take off”), are essential components of natural language. Unlike individual words, MWEs convey meanings that cannot always be deduced from their constituent words. Accurate identification and segmentation of MWEs is crucial for a wide range of natural language processing (NLP) tasks, including machine translation, information retrieval, and semantic analysis.

However, traditional word-by-word processing methods fail to capture these multiword units, often resulting in suboptimal sentence representations. Existing research on MWE segmentation is limited, particularly in the context of parallelized computation. Current approaches rely on sequential algorithms with  $O(n^2)$  runtime complexity, which becomes inefficient for long sentences or large corpora.

As such, this project addresses these challenges by automatically segmenting sentences into meaningful MWEs while minimizing the total number of MWEs per sentence.

## 2 PROBLEM DESCRIPTION

Let  $X = \{W_1, W_2, \dots, W_x\}$  be a dataset of input sentences. Each sentence  $W$  is a sequence of words represented as:

$$W = (w_1, w_2, \dots, w_n)$$

The goal is to generate an output sentence  $V$  for each input sentence  $W$  by grouping consecutive words into multiword expressions (MWEs). For example, the words  $w_1$  and  $w_2$  may be combined into a single MWE:

$$\{w_1, w_2\}$$

The final output sentence should contain the **fewest possible MWEs** while preserving the meaning of the original sentence. Thus, each output sentence takes a form such as:

$$V = (\{w_1, w_2\}, \dots, w_n)$$

Across the entire dataset, the resulting outputs form:

$$Y = \{V_1, V_2, V_3, \dots, V_x\}.$$

### 2.1 Example

For a sentence like “she looked up and down the hallway”, multiple segmentation candidates are possible. The correct final segmentation is the one containing the **minimum number of valid MWEs**, as shown in Table 1.

Candidate Output	# of MWEs
⟨she⟩⟨looked⟩⟨up⟩⟨and⟩⟨down⟩⟨the⟩⟨hallway⟩	7
⟨she⟩⟨looked up⟩⟨and⟩⟨down⟩⟨the⟩⟨hallway⟩	6
⟨she⟩⟨looked⟩⟨up and down⟩⟨the⟩⟨hallway⟩	5
⟨she⟩⟨looked up and down⟩⟨the⟩⟨hallway⟩	4

Table 1: Example of sentence segmentation into MWEs. The final segmentation minimizes the number of MWEs.

The final segmentation in the table is the optimal segmentation, as it achieves the minimum number of MWEs while preserving the original meaning of the sentence.

### 2.2 Valid MWEs

To determine whether a sequence of words forms a meaningful unit, the system uses a dictionary of English multiword expressions. This dictionary is used to build a lookup function that takes any sequence of words and returns a Boolean value indicating whether it is a valid MWE.

## 3 RELATED WORK

Extensive research exists on multiword expressions (MWEs), but most studies focus on **MWE extraction or detection**—identifying valid MWEs from large text corpora [1, 2, 4].

In contrast, this project focuses on **MWE segmentation**, where the task is to divide a sentence into meaningful MWE segments given a sentence and a predefined list of valid MWEs [5].

Current literature shows that research on MWE segmentation is limited, and **none have explored parallelizing internal processing steps**, such as lookup table generation or dynamic programming (DP) table computations using CUDA.

Our proposed implementation differs by **introducing parallelization strategies** for both dictionary lookup and DP segmentation, providing faster execution while retaining semantic coherence. This approach contributes novelty and potential value to the fields of computational linguistics and computer science.

## 4 ALGORITHM OVERVIEW

The algorithm begins by taking a single sentence as input. From this sentence, it generates all possible multiword expression (MWE) candidates by computing every sequential word combination within the sentence.

If a sentence  $W$  consists of  $n$  words, then the total number of candidate phrases can be expressed as:

$$\sum_{i=1}^n i = 1 + 2 + \dots + n = \frac{n(n+1)}{2}.$$

**Example:** For the sentence “She looked up and down the hallway,” the set of possible sequential word combinations includes:

```
["she", "she looked", "she looked up",
 "looked", "looked up", ...]
```

Once all candidate phrases are enumerated, each phrase is checked against a dictionary of valid MWEs. The dictionary serves as a lookup table that indicates whether a sequence of words forms a recognized multiword expression. For example, the phrase “looked up and down” is a valid MWE and is retained in the filtered list.

After obtaining the list of valid MWEs, the algorithm identifies the **optimal segmentation** of the sentence — the segmentation that minimizes the total number of MWEs while preserving grammatical and semantic coherence.

**Example:** For the sentence “She looked up and down the hallway,” the optimal segmentation is:

```
["she"], ["looked up and down"], ["the"], ["hallway"]
```

This reduces the total count from seven individual words to four MWEs.

This optimization is implemented using a **dynamic programming (DP) approach**. In this setup:

- The DP table’s rows and columns represent word indices in the sentence.
- Each cell stores the minimum number of MWEs required to segment that word span.
- The table is filled iteratively based on substructure dependencies, enabling efficient computation of the global minimum segmentation.

Formally, let  $l(x, y)$  be a Boolean function that returns TRUE if the span from word  $x$  to  $y$  (inclusive) forms a valid MWE. Let  $M(x, y)$  denote the minimum number of MWEs needed to cover

the subsentence from word  $x$  to word  $y$ . The following formula computes  $M(x, y)$ :

$$M(x, y) = \begin{cases} 1, & \text{if } x = y \text{ or } l(x, y) = \text{TRUE}, \\ \min_{x \leq m < y} (M(x, m) + M(m + 1, y)), & \text{otherwise.} \end{cases}$$

For the sentence “she looked up and down the hallway,” and assuming the only MWEs present are:

- “looked up”
- “up and down”
- “down the hallway”
- “looked up and down”

the following DP table can be constructed (as seen in Table 2):

	she	looked	up	and	down	the	hallway
<b>she</b>	1	2	2	3	2	3	4
<b>looked</b>		1	1	2	1	2	3
<b>up</b>			1	2	1	2	3
<b>and</b>				1	2	3	2
<b>down</b>					1	2	1
<b>the</b>						1	2
<b>hallway</b>							1

Table 2: Dynamic programming table for MWE segmentation. Each cell represents the minimum number of MWEs for the corresponding span.

## 5 PARALLELIZATION STRATEGIES

Several stages of the MWE segmentation process can benefit from parallelization. This project primarily utilizes CUDA for GPU-based parallel processing, focusing on three main opportunities:

### 5.1 Sentence-Level Parallelization

Multiple sentences can be processed concurrently, with each CUDA thread handling an entire sentence. Each thread constructs its own lookup table and executes the segmentation algorithm independently.

**Limitation:** This approach requires substantial GPU memory since each thread maintains its own dictionary and dynamic programming table.

### 5.2 Dictionary Lookup Parallelization

The generation and validation of MWE candidates against the dictionary can be massively parallelized. Each GPU thread can verify one or more candidate phrases concurrently, significantly reducing the time needed to build the lookup table.

### 5.3 Dynamic Programming (DP) Parallelization

The DP segmentation algorithm can be parallelized by exploiting the independence of cells along the same diagonal of the DP matrix.

- Each cell depends only on values from previous diagonals.
- All cells on a single diagonal can be computed concurrently, with one thread per cell.
- This reduces the effective computation time from  $O(n^2)$  to roughly  $O(n)$ , aside from minor synchronization overheads.

## 5.4 Proposed Implementation

For this project, we implement **dictionary lookup parallelization** and **DP parallelization** together.

- Sentence-level parallelization is avoided due to excessive memory requirements and complexity.
- Combining dictionary and cell-level parallelization provides efficient speedup while keeping memory usage manageable and minimizing synchronization overhead in CUDA kernels.

## 6 PARALLELIZED MWE DETECTION AND MINIMIZATION

The goal of our project is to **speed up the detection of multi-word expressions (MWEs)** and **accelerate the algorithm for minimizing the number of MWEs in a sentence**. The program consists of two main stages: lookup table creation and MWE minimization.

### 6.1 Lookup Table Creation

The first stage requires a predefined list of MWEs, stored as a **hashmap**. For each input sentence, all possible contiguous substrings are considered as potential MWEs. Each substring is checked against the dictionary, and valid matches are marked for later use in the minimization stage.

Traditionally, this process is performed **sequentially**, checking one substring at a time. By **parallelizing** this step, multiple candidate substrings can be evaluated concurrently, as their computations are independent, allowing for significant speedup.

Two parallelized versions were developed for lookup table creation:

#### (1) Version 1

- Launches a single GPU kernel per sentence with all necessary data, including the sentence and the entire MWE dictionary.
- The MWE dictionary is sent for every sentence, resulting in **redundant host-to-device (CPU-to-GPU) memory transfers**.

#### (2) Version 2

- Similar to Version 1, but the MWE dictionary is sent **only once** and persists in GPU memory.
- This reduces redundant memory transfers, cutting a large portion of host-to-device overhead and improving overall performance.

### 6.2 MWE Minimization Using Dynamic Programming

The MWE minimization algorithm uses a **dynamic programming (DP) approach**:

- Smaller combinations of the sentence are solved first, progressively moving to larger spans.
- The DP table's rows and columns represent word indices in the sentence.
- Each cell stores the **minimum number of MWEs** required to segment that span.

- The table is filled iteratively based on substructure dependencies, enabling efficient computation of the global minimum segmentation.

Four parallelized versions of the MWE minimization algorithm were developed:

#### (1) Version 1

- Diagonal-based parallelization: all cells along a diagonal of the DP table are computed concurrently.
- Launches a new GPU kernel for each diagonal, introducing significant kernel launch overhead.

#### (2) Version 2

- Launches a single kernel per sentence instead of per diagonal, reducing kernel launch overhead.
- Cells are still computed along diagonals, which may lead to **non-coalesced memory access**, as threads may not access contiguous memory locations.

#### (3) Versions 3 and 4

- Restructure the DP table so that rows and columns represent **start word index** and **span length** of the subsentence.
- Allows DP cells to be computed in **row-major or column-major order**, improving memory coalescence.
- This reorganization can improve cache utilization and overall efficiency.

## 7 EVALUATION METHODOLOGY

To evaluate the baseline sequential and CUDA-parallelized implementations, we conducted experiments using real data. Two main data sources were required: the MWE dictionary for lookup table creation and a set of test sentences containing MWEs.

### 7.1 MWE Dictionary

To obtain a reliable source of MWEs, we used **WordNet** [3], a lexical database of English words that organizes them into sets of synonyms (synsets) and captures semantic relationships between them. WordNet is widely used in computational linguistics for tasks such as word sense disambiguation and semantic analysis.

After extracting MWEs from WordNet, we performed preprocessing:

- For nouns, we generated plural forms (e.g., “light bulb” → “light bulbs”).
- For verbs, we generated all common inflections:
  - Base form (e.g., “lighten up”)
  - Past tense (e.g., “lightened up”)
  - Present participle (e.g., “lightening up”)
  - Past participle (e.g., “lightened up”)
  - Third-person singular present (e.g., “lightens up”)

The preprocessed MWEs were exported as a text file `mwe.txt`, which was imported by our program as the MWE dictionary.

### 7.2 Test Sentences

Test sentences were collected from two sources:

- (1) **Kaggle Dataset:** (“Random English Sentences” by NikiTricky)
- (2) **Created Sentences:** Constructed manually to include MWEs.

A total of **100 sentences** were selected, divided as follows:

- **Short Sentences:** 50 sentences with 10–20 words.
- **Long Sentences:** 50 sentences with more than 20 words.

These sentences were used consistently across all implementations to evaluate execution speed. Evaluation scenarios included:

- (1) Using only the 50 short sentences.
- (2) Using only the 50 long sentences.
- (3) Using all 100 sentences (short + long).

## 8 PERFORMANCE RESULTS

We evaluated all CPU and GPU implementations using 100 sentences (50 short, 50 long). For GPU implementations, the reported times include kernel execution as well as host-to-device (HtoD) and device-to-host (DtoH) memory transfer overheads. Each reported value is the average over 10 runs.

### 8.1 Lookup Table Creation

Dataset	CPU	GPU v1	GPU v2
50 Short	15.96	69.44	1.42
	1x	0.23x	11.26x
50 Long	45.01	77.96	1.49
	1x	0.58x	30x
100 (Both)	63.45	126.80	1.70
	1x	0.50x	37.34x

Table 3: Average execution times (first row, in milliseconds) and speedups relative to CPU (second row) for MWE lookup table creation. GPU v2 achieves a substantial speedup over both CPU and GPU v1.

### 8.2 MWE Minimization (Dynamic Programming)

Dataset	CPU DP	GPU v1	GPU v2	GPU v3	GPU v4
50 Short	2.14	1.88	0.64	0.75	0.66
	1x	1.14x	3.36x	2.84x	3.21x
50 Long	7.23	3.20	0.99	1.25	1.08
	1x	2.26x	7.30x	5.78x	6.71x
100 (Both)	8.72	5.06	1.63	1.98	1.71
	1x	1.72x	5.34x	4.39x	5.09x

Table 4: Average execution times (in milliseconds, first row) and speedups relative to CPU (second row) for DP-based MWE minimization. GPU v2 provides the highest speedup across all datasets.

### 8.3 Discussion of Results

8.3.1 *Lookup Implementation.* For the lookup function, the CPU kernel initially outperformed the GPU kernel (GPU Lookup v1).

This is not because the CPU is inherently faster; rather, each time we launch the GPU kernel for a new sentence, the entire dictionary lookup must be transferred from the host (CPU) to the device (GPU).

Analysis shows that most of the GPU execution time is **spent in this HtoD (host-to-device) memory transfer**. These heavy transfers caused GPU Lookup v1 to sometimes run only half as fast as the CPU (0.58x for long sentences and 0.50x for the merged set) or even a quarter as fast (0.23x for short sentences).

To address this, we created a **second version of the GPU kernel (GPU Lookup v2)** that keeps the dictionary in GPU memory across multiple sentence kernels, transferring it only once. With this optimization, the speedup compared to the CPU kernel improves dramatically: for short sentences, GPU Lookup v2 achieves approximately **11.3x** speedup; for long sentences, around **30x**; and for the merged 100-sentence set, about **37.34x**.

Overall, this exercise in optimizing overhead only serves to exemplify the importance of carefully examining the overhead involved in GPU implementations. Without breaking down where time is actually spent, performance comparisons between CPU and GPU kernels can be misleading. More importantly, analyzing overhead such as memory transfers provides actionable insights that guide how GPU implementations can be improved moving forward.

8.3.2 *MWE-Minimization Dynamic Programming.* For the **MWE minimization algorithm**, we hypothesized that shorter sentences would have a smaller speedup factor compared to the CPU version. This is because the DP table for short sentences has smaller diagonals, resulting in fewer cells processed in parallel.

Our results support this hypothesis:

- **Short sentences:** 3.36x speedup using GPU DP v2 over CPU DP
- **Long sentences:** 7.30x speedup using GPU DP v2 over CPU DP
- **Merged set:** 5.34x speedup using GPU DP v2 over CPU DP

Across all test sets, the **GPU v2 implementation consistently outperforms GPU v3 and v4**, showing that non-coalesced memory is not a significant issue for the diagonal-based DP implementation. GPU v2 represents the best-performing GPU optimization and is used for speedup comparisons. The consistent performance trend is: **v2 > v4 > v3**.

After deeper analysis, this trend is related to memory non-coalescence at a more subtle level. Across versions v2, v3, and v4, the main difference is the *order in which the DP algorithm fills the table*, i.e., the outer loop traversal pattern. However, each DP cell computation heavily depends on an **inner loop**, which executes far more times than the outer loop. Thus, the inner loop's memory-access pattern has a stronger effect on overall performance.

Memory-access patterns of the inner loop:

- (1) **v2:** accesses DP cells in **row-major (fully contiguous)** and **column-major (semi-contiguous)** order
- (2) **v3:** accesses DP cells **diagonally (non-contiguous)** and **column-major (semi-contiguous)**
- (3) **v4:** accesses DP cells **diagonally (non-contiguous)** and **row-major (fully contiguous)**

As a visualization, Tables 5–7 demonstrate how the inner-loop memory-access patterns differ across GPU DP implementations.

**Table 5: GPU DP v2, solving for the final cell labeled X. Dependency cells are labeled O, unused cells as -. Access pattern: row-major + column-major.**

	0	1	2	3
0	O	O	O	X
1	-	-	O	
2		-	O	
3			O	

**Table 6: GPU DP v3, solving for the final cell labeled X. Dependency cells are labeled O, unused cells as -. Access pattern: diagonal + column-major.**

	0	1	2	3
0	O	-	-	O
1	O	-	O	
2	O	O		
3	X			

**Table 7: GPU DP v4, solving for the final cell labeled X. Dependency cells are labeled O, unused cells as -. Access pattern: row-major + diagonal.**

	0	1	2	3
0	O	O	O	X
1	-	-	O	
2	-	O		
3	O			

From this analysis, it is clear that **versions whose inner loops access memory more contiguously perform better**. Row-major access is most efficient, column-major is moderately efficient, and diagonal access is the worst due to irregular memory strides. This explains why v2 performs best: the outer loop traverses diagonally, but the inner loop achieves fully contiguous, cache-friendly memory access.

This highlights the importance of **carefully considering memory-access patterns and contiguity** when designing GPU algorithms in CUDA.

## 9 CONCLUSION

This work has demonstrated the application of **CUDA parallelization** to the problem of segmenting sentences into **MWEs**. By presenting various **implementations** and **iterations**, we provided insights into **minimizing overhead**, particularly in the context of **CPU-GPU communication**, and highlighted the importance of designing algorithms that take advantage of **coalesced memory access**. Furthermore, given that our CUDA implementations achieved significant **speedups** compared to traditional **sequential**

**implementations**, this project exemplifies the power of **parallelization** and the substantial **performance gains** that can be achieved through careful and thoughtful **GPU optimization**.

## 10 USE OF AI ASSISTANCE

For transparency, ChatGPT was used to assist in **grammar refinement**, **wording improvements**, and **formatting** throughout this report. Prompts were primarily of the form: “[Paragraph]. Please fix the wording of this section.” All **analysis**, **insights**, **interpretations**, and **technical discussions** originate solely from the authors and reflect their own understanding and reasoning.

All AI-assisted outputs were **carefully reviewed and verified by the authors** to ensure that they were **factually correct** and free of hallucinations.

## REFERENCES

- [1] Mathieu Constant, Gülsen Eryiğit, Johanna Monti, Lonneke van der Plas, Carlos Ramisch, Michael Rosner, and Amalia Todorascu. 2017. Multiword Expression Processing: A Survey. *Computational Linguistics* 43, 4 (12 2017), 837–892. arXiv:[https://direct.mit.edu/coli/article-pdf/43/4/837/1808392/coli\\_a0302.pdf](https://direct.mit.edu/coli/article-pdf/43/4/837/1808392/coli_a0302.pdf) doi:10.1162/COLI\_a\_00302
- [2] Kamil Kanclerz and Maciej Piasecki. 2022. Deep Neural Representations for Multiword Expressions Detection. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics: Student Research Workshop*, Samuel Louvan, Andrea Madotto, and Brielen Madureira (Eds.). Association for Computational Linguistics, Dublin, Ireland, 444–453. doi:10.18653/v1/2022.acl-srw.36
- [3] George A. Miller. 1994. WordNet: A Lexical Database for English. In *Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*. <https://aclanthology.org/H94-1111/>
- [4] Nathan Schneider, Emily Danchik, Chris Dyer, and Noah A. Smith. 2014. Discriminative Lexical Semantic Segmentation with Gaps: Running the MWE Gamut. *Transactions of the Association for Computational Linguistics* 2 (04 2014), 193–206. arXiv:[https://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl\\_a0176/1566816/tacl\\_a0176.pdf](https://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a0176/1566816/tacl_a0176.pdf) doi:10.1162/tacl\_a\_00176
- [5] Jake Williams. 2017. Boundary-based MWE segmentation with text partitioning. In *Proceedings of the 3rd Workshop on Noisy User-generated Text*, Leon Derczynski, Wei Xu, Alan Ritter, and Tim Baldwin (Eds.). Association for Computational Linguistics, Copenhagen, Denmark, 1–10. doi:10.18653/v1/W17-4401