player 1

highscore 2500

player 2

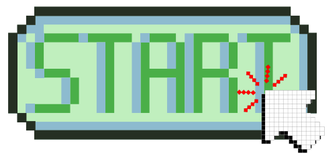
Term frequency – Inverse document frequency

ชัชวาล เมืองใหม่

6310400959

ณภัทร ดลภาวิจิต

6310400967



Information retrieval

การ**หา**เอกสารจากกองข้องมูล
ที่ตอบโจทย์ความต้องการ

The democracy in Thailand

Search

Advanced search: Sort by relevance

Search in: (Article)

The page "The democracy in Thailand" does not exist. You can create a draft and submit it for review, but consider checking the search results below to see whether the topic is already covered.

Thailand

Thailand's neighbours. Apart from a brief period of parliamentary **democracy in the** mid-1970s, **Thailand** has periodically alternated between **democracy** and...

209 KB (19,758 words) - 14:30, 2 April 2023

Democracy Monument

The Democracy Monument (**Thai**: อนุสาวรีย์ประชาธิปไตย, romanized: Anusawari Prachathipatai) is a public monument **in the** city center of Bangkok, capital...

12 KB (1,444 words) - 03:41, 15 October 2022

Red Shirts (**Thailand**)

the extreme inequality and of the fundamentally weak **democracy in Thailand**, typified by **Thailand**'s primate city problem. Red Shirts group dynamics center...

6 KB (683 words) - 05:28, 15 November 2022

New **Democracy** Party (**Thailand**)

New **Democracy** Party (**Thai**: พรรคประชาธิปไตยใหม่, Phak Prachathipataimai) is a political party **in Thailand** that founded on 21 April 2011. Suratin Pichan...

3 KB (62 words) - 05:37, 22 February 2022

Feminism **in Thailand**

through a medium of social movement activist groups within **Thailand**'s illiberal **democracy**. **The Thai** State claims to function as a civil society with an intersectionality...

17 KB (2,107 words) - 22:42, 1 December 2022

Term frequency

counts the number of times each term occurs in each document; the number of times a term occurs in a document is called its **term frequency**. If documents are of differing lengths, adjustments may be necessary. The Hans Peter Luhn (1957) method is the first form of term weighting, in which

> the weight of a term in a document is proportional to its frequency.

With this method, query term frequency can serve to distinguish among relevant texts.

Word important

Some word may or may not important to the document

Inverse document frequency

To adjust the weight of terms in a document set, a factor called "inverse document frequency" is used. This factor reduces the weight of frequently occurring terms and amplifies the weight of rarely occurring terms. In 1972, Karen Spärck Jones developed a statistical approach to measure term specificity called Inverse Document Frequency (IDF), which is now a fundamental component of term weighting. This approach considers

a term's specificity to be inversely proportional to how frequently it appears across documents.

Term frequency

Term frequency, tf(t,d), is the relative frequency of term t within document d,

count of a term in a document

total number of terms in document *d*

$$\mathrm{tf}(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

Inverse document frequency

It is the logarithmically scaled inverse fraction of the documents that contain the term t.

number of documents

$$\mathrm{idf}_t = \log \frac{N}{df_t} = \log \frac{N}{|\{d \in D : t \in d\}|}$$

number of documents in which the term appears

it measures the rarity of the term in the set of documents

Back to Agenda Page

parallel TF-IDF

Pre processing

Corpus

```
['this is a a sample',
 'this is another another example example example']
```

Unique terms

```
['sample', 'is', 'a', 'another', 'this', 'example']
```

Term ids

```
   0       1     2       3         4         5
```

Corpus

```
[[4, 1, 2, 2, 0],
 [4, 1, 3, 3, 5, 5, 5]]
```

Load corpus

Input file

```
4 1 2 2 0
4 1 3 3 5 5 5
```

Padding

Corpus array

```
{4, 1, 2, 2, 0, -1, -1, 4, 1, 3, 3, 5, 5, 5}
```

```
corpusSize = 2; seqLen = 7;
```

Find vocabulary size

sequential

```c
int find_vocab_size(int *corpus, size_t corpus_size, size_t max_len)
{
  if (corpus == NULL)
  {
    fprintf(stderr, "corpus is null\n");
    return -1;
  }

  int max = corpus[0];
  for (size_t i = 0; i < corpus_size; i++)
  {
    for (size_t j = 0; j < max_len; j++)
    {
      if (corpus[i * max_len + j] > max)
      {
        max = corpus[i * max_len + j];
      }
      if (corpus[i * max_len + j + 1] == -1)
      {
        break;
      }
    }
  }
  return max + 1;
}
```

O(corpus_size * max_len)

reduce max

parallel

```c
__global__ void find_vocab_size_kernel(int *corpus, int *out_arr, size_t corpus_size, size_t max_len)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  __shared__ int sdata[256];
  if (idx < corpus_size * max_len)
  {
    // load data into shared memory
    sdata[threadIdx.x] = corpus[idx];
    __syncthreads();

    // sequential addressing reduction
    for (int stride = blockDim.x / 2; stride > 0; stride /= 2)
    {
      if (threadIdx.x < stride)
      {
        // compare and replace
        int lv = sdata[threadIdx.x];
        int rv = sdata[threadIdx.x + stride];
        if (lv < rv)
        {
          sdata[threadIdx.x] = rv;
        }
        else
        {
          sdata[threadIdx.x] = lv;
        }
      }
      __syncthreads();
    }
  }

  // write result for this block to global memory
  if (threadIdx.x == 0)
  {
    out_arr[blockIdx.x] = sdata[0];
  }
}
```

O(log stride)

Count term occurrence

sequential

```
void frequency_count(int *corpus,
                     int corpus_size,
                     int max_len,
                     int **count_arr)
{
  for (size_t i = 0; i < corpus_size; i++)
  {
    for (size_t j = 0; j < max_len; j++)
    {
      if (corpus[i * max_len + j] != -1)
      {
        count_arr[i][corpus[i * max_len + j]]++;
      }
    }
  }
}
```

O(corpus_size * max_len)

parallel

```
__global__ void frequency_count_kernel(int *corpus, int *count_arr,
                                       int seq_max_len, int vocab_size,
                                       int corpus_size)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < corpus_size * seq_max_len)
  {

    __shared__ int shcorpus[256];

    shcorpus[threadIdx.x] = corpus[idx];
    __syncthreads();

    if(shcorpus[threadIdx.x] != -1){
      // count_arr[shcorpus[threadIdx.x]] += 1;
      atomicAdd(&count_arr[shcorpus[threadIdx.x] +
               (vocab_size * (threadIdx.x / seq_max_len))], 1);
    }

    __syncthreads(); // Add a synchronization call here



  }
}
```

O(corpus_size)

term frequency

sequential

```
void term_frequency(int **term_counts, int corpus_size, int vocab_size, double **out_arr)
{
  for (int i = 0; i < corpus_size; i++)
  {
    int wc_sum = 0;
    for (size_t wj = 0; wj < vocab_size; wj++)
    {
      wc_sum += term_counts[i][wj];
    }

    for (size_t j = 0; j < vocab_size; j++)
    {
      out_arr[i][j] = (double)term_counts[i][j] / wc_sum;
    }
  }
}
```

O(corpus_size * vocab_size)

parallel

```
__global__ void term_frequency_kernel(int *term_counts, int corpus_size,
                                      int vocab_size, double *out_arr)
{
  int doc = blockIdx.x;
  int wrd = threadIdx.x;
  __shared__ int scounts[256];

  if (wrd < vocab_size)
  {
    scounts[wrd] = term_counts[doc * vocab_size + wrd];
    __syncthreads();

    for (int stride = 1; stride < vocab_size; stride *= 2)
    {
      int index = 2 * stride * wrd;
      if (index < vocab_size)
      {
        scounts[index] += scounts[index + stride];
      }
      __syncthreads();
    }

    out_arr[doc * vocab_size + wrd] = (double)term_counts[doc * vocab_size + wrd]
                                      / scounts[0];

  }
}
```

each block contains one document

reduce sum

O(log vocab_size)

Invert doc frequency

sequential

```
void invert_document_frequency(int **term_counts, int corpus_size,
                               int vocab_size, int smooth_idf,
                               double *out_arr)
{
  for (size_t wi = 0; wi < vocab_size; wi++)
  {
    int w_dc = 0 + smooth_idf;
    for (size_t di = 0; di < corpus_size; di++)
    {
      if (term_counts[di][wi])
        w_dc++;
    }

    out_arr[wi] = log((double)corpus_size / w_dc) + 1;
  }
}
```

O(vocab_size * corpus_size)

reduce sum

parallel

```
__global__ void invert_document_frequency_kernel(int *term_counts, int corpus_size,
                                                 int vocab_size, int smooth_idf,
                                                 double *out_arr)
{
  int wrd = blockIdx.x;
  int doc = threadIdx.x;
  __shared__ int scounts[256];

  if (wrd < vocab_size)
  {
    scounts[doc] = term_counts[doc * vocab_size + wrd] ? 1 : 0;
    __syncthreads();

    for (int stride = (corpus_size + 1) / 2; stride > 0; stride >>= 1)
    {
      if (doc < stride)
      {

        scounts[doc] += scounts[doc + stride];
      }
      __syncthreads();
    }
  }
```

each block contains one term in all doc

O(log corpus_size)

TF-IDF

sequential

```
void tfidf(double** tf_arr, double *idf_arr,
           double **out_arr, int corpus_size,
           int vocab_size)
{
  for (int di = 0; di < corpus_size; di++)
  {
    for (size_t wi = 0; wi < vocab_size; wi++)
    {
      out_arr[di][wi] = tf_arr[di][wi] * idf_arr[wi];
    }
  }
}
```

O(corpus_size * vocab_size)

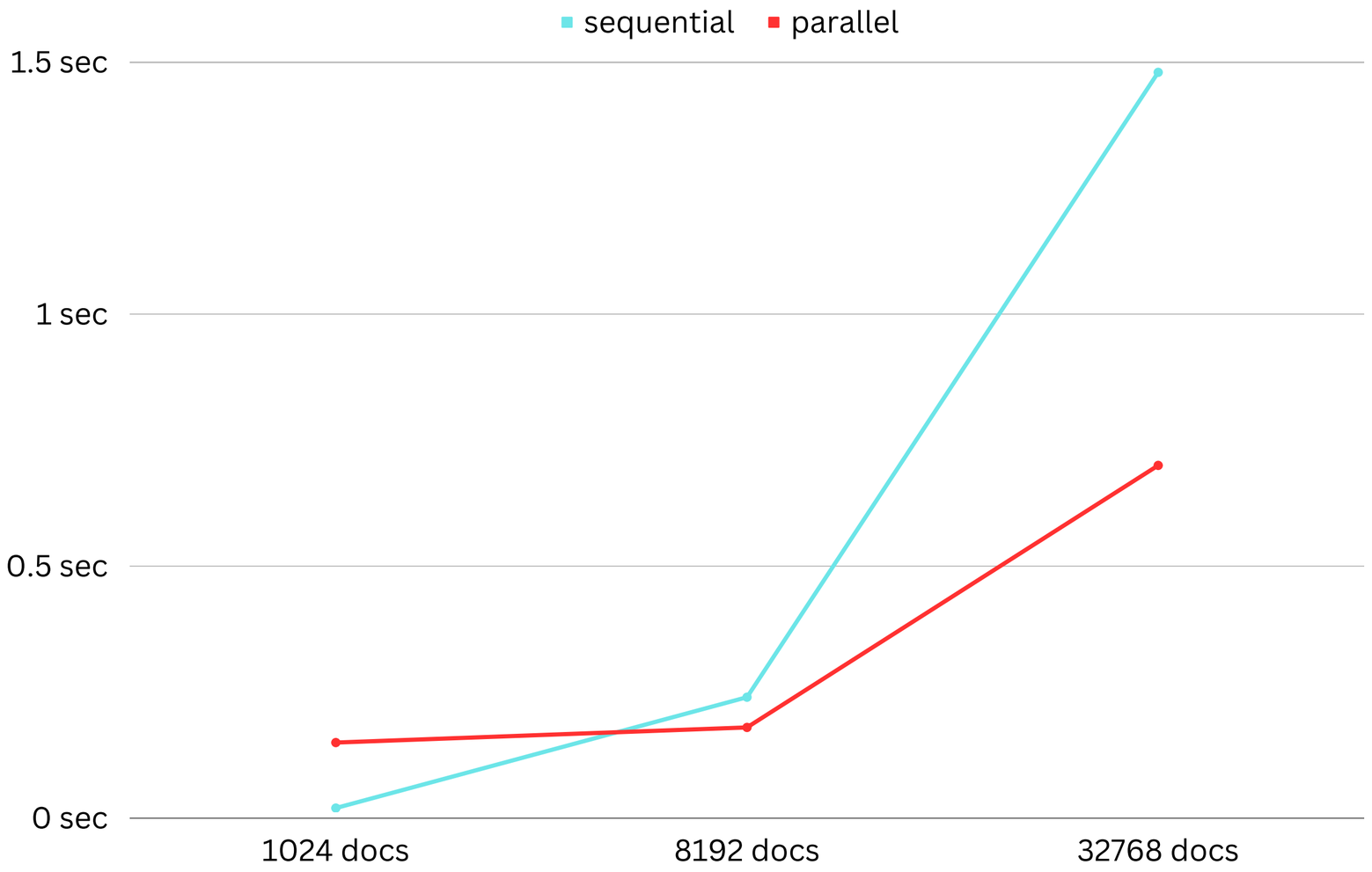
parallel

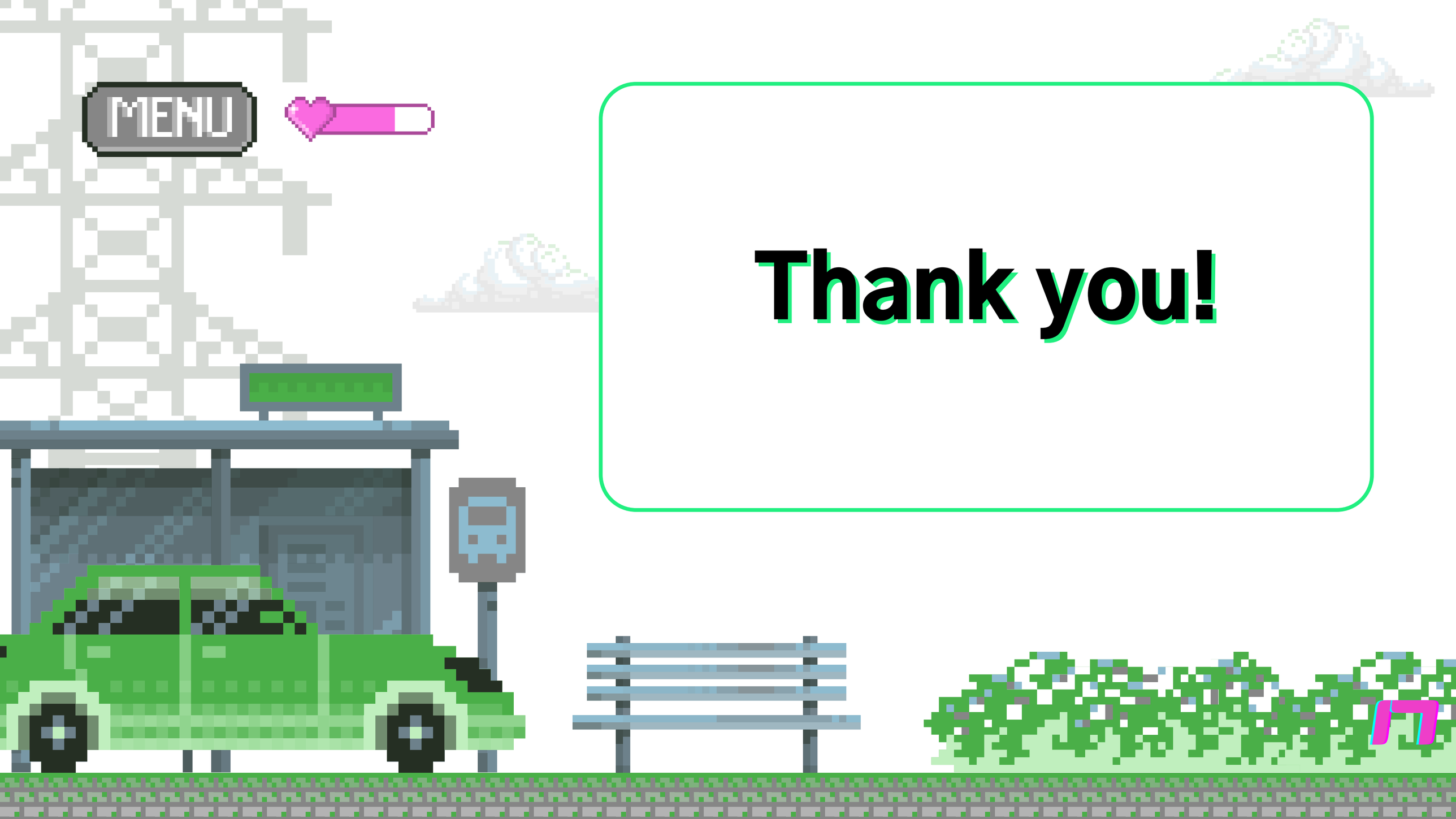
```
__global__ void tfidf_kernel(const double *tf_arr,
                             const double *idf_arr,
                             double *out_arr,
                             int corpus_size,
                             int vocab_size)
{
  int idx = blockDim.x * blockIdx.x + threadIdx.x;
  if (idx < corpus_size * vocab_size)
  {
    int wi = idx % vocab_size;

    out_arr[idx] = tf_arr[idx] * idf_arr[wi];
  }
}
```

O(corpus_size)

Compute time





Thank you!