Downsampling using max pooling

Gagarine Yaikhom

***Abstract*—In this note we implement the downsampling of a two-dimensional matrix using max pooling.**

I. Matrix data structure

We define a matrix data structure to contain three key information. First, the number of rows and colums in the matrix, and a pointer to the memory allocated to store the matrix elements.

⟨*Common types*⟩≡

```
typedef struct {
    float *m;    /* matrix elements */
    uint32_t nr; /* number of rows */
    uint32_t nc; /* number of columns */
} matrix_2d;
```

II. Sliding window data structure

For downsampling, we specify the sliding window using two pairs of parameters. The first pair gives the width and height of the window, and the second pair gives the stride of the window along the row and column.

⟨*Common types*⟩+≡

```
typedef struct {
    uint16_t nr, nc; /* number of rows and columns */
    uint16_t rs, cs; /* row and column strides */
} window_2d;
```

III. Max pooling

The `maxpool` function take as input the original matrix `orig` that is to be downsampled, the specification `window` of the sliding window to be used for the downsampling, and the number of matrix elements to process in a single batch. Based on this, the function returns two matrices: `result` stores the downsampled maximum values, and `indices` stores the corresponding index of the maximium value in the original matrix. For each maximum element in `result`, the corresponding index i stored in `indices` is calculated as $i = r * n + c$, where r and c are respectively the row and column of the maximum element, and n is the number of columns in the original matrix.

To maximise cache utilisation, we process a batch of row elements from the original matrix. For each element in a batch, we then process all of the rows in the downsampled `result` matrix that it affects. The size of a batch is calculated based on the cacheline available on the processor, and is supplied to the function as `batch`.

Since we will be accumulating the maximum value in `result` matrix as we process each batch, we need to ensure that at the beginning, all of elements in the `result` matrix are initialised to the minimum possible values. This is supplied as `min_value`. For instance, we if are downsampling the results of ReLU activation, we will set `min_value` to zero. On the other hand, if we are downsampling following a $\tanh()$ activation, the `min_value` will be set to -1.

⟨*Functions*⟩≡

```
void maxpool(
    matrix_2d *result, matrix_2d *indices,
    const matrix_2d *orig, const window_2d *windows,
    uint16_t batch, float min_value)
{
    ⟨Initialise downsampled matrix⟩;
    uint16_t rs = windows->rs; /* row stride */
    uint16_t cs = windows->cs; /* column stride */
    uint16_t wr = windows->nr; /* window rows */
    uint16_t wc = windows->nc; /* window colums */
    for (uint32_t r = 0; r < orig->nr; ++r) {
        for (uint32_t c = 0; c < orig->nc; c += batch) {
            ⟨Process rows affected by batch elements⟩;
        }
    }
}
```

We must initialise the `result` matrix before processing the original matrix. This is required as each cell in the `result` matrix is used as an accumulator that stores the current running maximum for a row of the sliding window.

⟨*Initialise downsampled matrix*⟩≡
```
for (uint32_t i = 0; i < result->nr * result->nc; ++i) {
    result->m[i] = min_value;
}
```

For each element in a batch, we must process all of the windows that includes the element. Note that as we slide the window by the specified row and column strides, certain elements will fall inside the window while some are not. Hence, for a given element in the original matrix, depending on the sliding window specification, it will be part of several windows. Since each of these windows corresponds to an element in the downsampled matrix `result`, we must process all of the cells in `result` affected by an element in the batch.

To further maximise cache utilisation when accessing the `result` matrix, we try to process all affected columns in a row of `result` before we process other rows. Furthermore, since the elements of a batch are memory local, we process a row in `result` against all elements of a batch, before moving to other rows. To do this, we first determine the top-most window that contains elements of the current batch. In this window, the elements will occupy the bottom cells of the sliding window.

We first calculate the correct position of the sliding window, and place the `top` of the window at the correct row. For each window we process all of the columns affected by the elements in the batch that belong to that window. We then slide the window vertically down by the specified row stride `rs`, until the window can no longer include elements in the batch, which happens when the bottom of the sliding window goes beyond the bottom of the original matrix.

We determine if the elements of a batch lie inside a sliding window by checking if the row `r` lies inside the `top` and `bottom` of the sliding window.

⟨*Process rows affected by batch elements*⟩≡
```
⟨Calculate start row for the sliding window⟩;
for (uint32_t top = ridx * rs; top <= r; top += rs) {
    uint32_t bottom = top + wr - 1;
    if (bottom >= orig->nr) {
        break;
    }
    if (top <= r && bottom >= r) {
        ⟨Process affected columns of elements in batch⟩;
    }
    ++ridx;
}
```

The variable `row_start` gives the row index so that the elements are at the bottom of the window, which is calculated relative to the current row of the batch `r`. We must ensure that we do not go beyond the boundary of the matrix. From this, we calculate the index of the affected row in `result`, which is stored in the variable `ridx`. We will use `ridx` to place the sliding window correctly.

⟨*Calculate start row for the sliding window*⟩≡
```
int32_t row_start = r - wr + 1;
if (row_start < 0) {
    row_start = 0;
}
uint32_t ridx = (uint32_t) floor(row_start / rs);
```

For each row covered by a sliding window, we process all of the elements in the current batch. We first ensure that all elements of the batch belong to the original matrix. This is necessary because we store the matrix in row-major form, and if the batch size does not divide the number of the columns in the original matrix evenly, we will have batched that are not full. We check this by first determining the real column index of the element in the batch `k` using the current batch start index `c`. All of these indices match the indices of the elements in the original matrix. Hence, if `k` is not less than the number of the columns in the original matrix, we know that the batch is not full.

For each element in the batch, we then process all of the columns at row `ridx` in `result` that are affected by the element.

⟨*Process affected columns of elements in batch*⟩≡
```
for (uint16_t j = 0; j < batch; ++j) {
    uint32_t k = c + j;
    if (k >= orig->nc) {
        break;
    }
    ⟨Process affected columns for the row⟩;
}
```

First we determing the value `v` and index `vidx` of the batch element. We then calculate the starting column `left` of the sliding window. While the sliding window contains `v`, we check if the value is greater than the value in `result`. If yes, we replace the cell in `result` with `v` and updated the index in `indices` with `vidx`.

To check that the sliding window contains `v`, we ensure that the `right` of the sliding window does not extend beyong the right boundary of the original matrix. We then check if the column index `k` of the element lies inside the window.

⟨*Process affected columns for the row*⟩≡

```
⟨Get batch element to process and index to store⟩;
⟨Calculate start column for the sliding window⟩;
for (uint32_t left = cidx * cs; left <= k; left += cs) {
    uint32_t right = left + wc - 1;
    if (right >= orig->nc) {
        break;
    }
    if (left <= k && right >= k) {
        ⟨Check an update maximum value and index⟩;
    }
    ++cidx;
}
```

The index `vidx` is the value stored in `indices` matrix when the value `v` in the original matrix `orig` is greater than the current value of the cell in `result`.

⟨*Get batch element to process and index to store*⟩≡

```
uint32_t vidx = r * orig->nc + k;
float v = orig->m[vidx];
```

Similar to how we determine the top of the sliding window, we use a similar method to determine the leftmost sliding window for the given row that contains the element of the batch. In this leftmost sliding window, the current batch element should occupy the rightmost cells of the window. The variable `cidx` stores the index in `result` that corresponds to the sliding window starting at the leftmost sliding window.

⟨*Calculate start column for the sliding window*⟩≡

```
int32_t col_start = k - wc + 1;
if (col_start < 0) {
    col_start = 0;
}
uint32_t cidx = (uint16_t) floor(col_start / cs);
```

We get the current maximum from `result` using the `ridx` and `cidx`, and update both `result` and `indices` if the batch element is greater than the current value.

⟨*Check an update maximum value and index*⟩≡

```
uint32_t q = ridx * result->nc + cidx;
if (v > result->m[q]) {
    result->m[q] = v;
    indices->m[q] = vidx;
}
```

Create a matrix by allocating the memory for the matrix elements. This function assumes that the number or rows and columns are already specified inside the object pointed to by the supplied pointer `m` in accordance to the matrix data-structure specified in `matrix_2d`.

⟨*Functions*⟩+≡

```
int create_matrix(matrix_2d *m)
{
    if (!m) {
        return -1;
    }
    m->m = (float *) calloc(
        m->nr * m->nc, sizeof(float));
    if (m->m == NULL) {
        return -2;
    }
    return 0;
}
```

Destroys a matrix by freeing up the memory allocated for the matrix elements.

⟨*Functions*⟩+≡

```
void destroy_matrix(matrix_2d *m)
{
    if (m && m->m) {
        free(m->m);
        m->m = NULL;
    }
}
```

⟨*Test functions*⟩≡

```
void print_mat(matrix_2d *m, const char *title)
{
    printf("\n%s:\n", title);
    for (size_t i = 0; i < m->nr; ++i) {
        for (size_t j = 0; j < m->nc; ++j) {
            printf("%7.2f ", m->m[i * m->nc + j]);
        }
        printf("\n");
    }
}
```

The cachline size is 64 bytes on the Intel Core i5 processor. Hence, we set the batch size `batch` to accommodate 16 floating-point values (64 bytes / 4 bytes-per-float). The correct cacheline size should be detected or supplied.

⟨*Test functions*⟩+≡

```
void test_maxpooling()
{
    matrix_2d orig = {.nr = 10, .nc = 10};
    matrix_2d result = {.nr = 5, .nc = 5};
    matrix_2d indices = {.nr = 5, .nc = 5};
    window_2d window = {.nr = 2, .nc = 2,
                        .rs = 2, .cs = 2};
    uint16_t batch = 16;
    float min_value = 0.0f;

    int r = create_matrix(&orig);
    if (r) {
        goto mem_alloc_fail;
    }
    r = create_matrix(&result);
    if (r) {
        goto mem_alloc_fail;
    }
    r = create_matrix(&indices);
    if (r) {
        goto mem_alloc_fail;
    }
    size_t n = orig.nr * orig.nc;
    uint32_t i = 0;
    while (i < n) {
        orig.m[i] = i + 1;
        ++i;
    }
    maxpool(&result, &indices, &orig,
            &window, batch, min_value);
    print_mat(&orig, "Original matrix");
    print_mat(&result, "Downsampled matrix");
    print_mat(&indices, "Max indices");
    goto done;

mem_alloc_fail:
    fprintf(stderr, "Failed memory allocation\n");

done:
    destroy_matrix(&indices);
    destroy_matrix(&result);
    destroy_matrix(&orig);
}
```

⟨*Test functions*⟩+≡

```
void test_downsampling()
{
    test_maxpooling();
}
```

IV. Main program

Main program for testing the functions.

⟨*Main*⟩≡

```
int main(void) {
    test_downsampling();
    return 0;
}
```

⟨*Standard libraries*⟩≡

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
```

⟨*algorithm.c*⟩≡

⟨*Standard libraries*⟩
⟨*Common types*⟩
⟨*Functions*⟩
⟨*Test functions*⟩
⟨*Main*⟩