Viola-Jones Face Detection on the GPU (Part 1...)

Nathaniel Burgdorfer

Overview

- Overall Algorithm for Viola-Jones Face Detection
- Optimizing Integral Image Computation
- Feature Building
- Optimizing AdaBoosting
- Results

Viola-Jones Algorithm

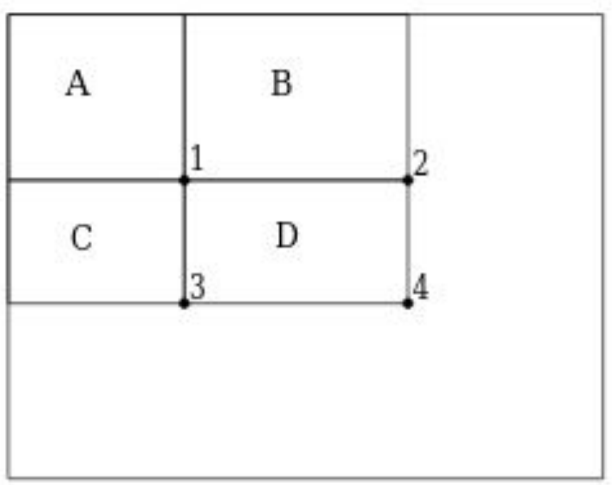
Viola-Jones Face Detection - Overview

<u>Goal:</u> Build a set of classifiers that are applied over images for the purpose of object detection

- Integral Images
- Learning Algorithm (basically AdaBoosting)
- Cascading Classification

Integral Images

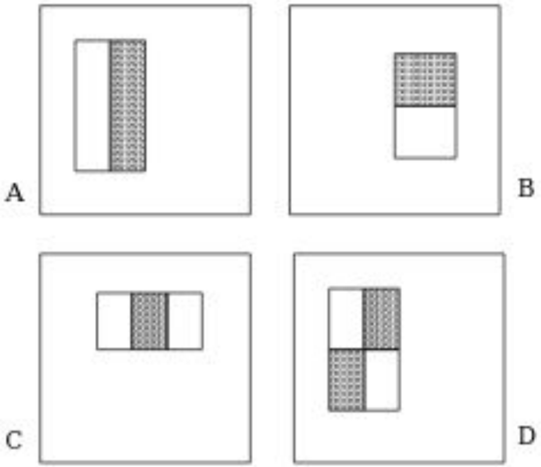
- An intermediate representation of an images
- Each pixel in the integral image is the inclusive sum of all pixels above and to the left



Sum of D: 4 - (2 + 3) + 1

Haar Features

- Rectangular filters
- Applied to an image
- Returns a response value for each feature



Implementation

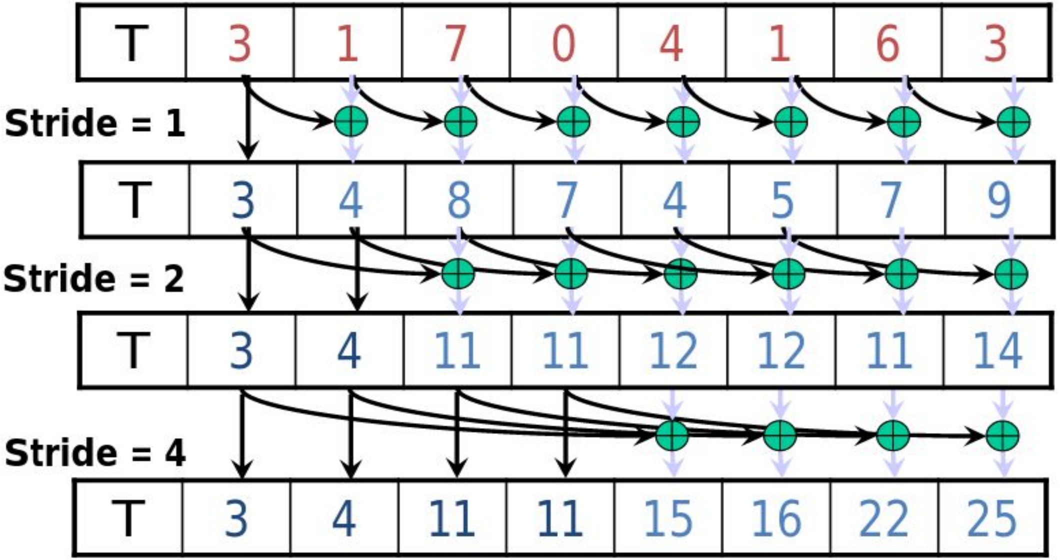
Algorithm Pipeline

Computing Integral Images

```
struct Mat* compute_integrals(struct Mat *images, int total_samples) {
    int rows;
    int cols;
    struct Mat *integral_images;

    rows = images[0].rows;
    cols = images[0].cols;

    float row_sum[rows][cols];

    integral_images = (struct Mat*) malloc(sizeof(struct Mat) * total_samples);

    for (int i=0; i<total_samples; ++i) {
        integral_images[i].values = (float*) malloc(sizeof(float) * rows*cols);
        integral_images[i].rows = images[i].rows;
        integral_images[i].cols = images[i].cols;

        for (int r=0; r<rows; ++r){
            for (int c=0; c<cols; ++c) {
                row_sum[r][c] = images[i].values[(r*rows)+c];

                if (c>0) {
                    row_sum[r][c] += row_sum[r][c-1];
                }

                integral_images[i].values[(r*rows)+c] = row_sum[r][c];

                if (r>0) {
                    integral_images[i].values[(r*rows)+c] += integral_images[i].values[((r-1)*rows)+c];
                }
            }
        }
    }

    return integral_images;
}
```

- Loop through all images
  - Set image params
  - Loop through all pixels
    - Set initial row-sum value
    - Add previous column value
    - Set initial integral images value to current row-sum
    - Add previous row's row-sum

A Kogge-Stone Parallel Scan Algorithm



1. Load input from global memory to shared memory.
2. Iterate log(n) times, stride from 1 to ceil(n/2.0). Threads *stride* to *n-1 active:* add pairs of elements that are s*tride* elements apart.
3. Write output from shared memory to device memory

Iteration #3
Stride = 4

Initial Implementation

```
__global__ void parallel_scan_pitched(float *images, int rows, int cols, size_t pitch) {

    float temp_val = 0.0;
    int offset = 0;
    int max_stride = ceil(blockDim.x/2.0);

    // get image for current block
    float *img = (float*) ((char*) images + blockIdx.x*pitch);

    // build image integral per block (576 threads) via Kogge-Stone Parallel Scan Algo (w/o double buffering)
    //#pragma unroll 1
    for (int stride=1; stride<=max_stride; stride*=2) {
        __syncthreads();
        offset = threadIdx.x - stride;
        if (offset >= 0) {
            temp_val = img[(threadIdx.y*rows)+threadIdx.x] + img[(threadIdx.y*rows)+offset];
        }

        __syncthreads();
        if (offset >= 0) {
            img[(threadIdx.y*rows)+threadIdx.x] = temp_val;
        }
    }
}
```

- compute the max stride
- grab the correct image for the current block
- loop through the rows of the images
    - compute offset
    - store the addition
    - set the img value to the temp value

*(pitched data used for coalescing)

Utilizing Shared Memory

```
__global__ void parallel_scan_shared_mem_sb_pitched(float *images, int rows, size_t pitch) {

    // create shared memory array
    __shared__ float temp[576];

    float temp_val = 0.0;
    int offset = 0;
    int max_stride = ceil(blockDim.x/2.0);

    // get image for current block
    float *img = (float*) ((char*) images + blockIdx.x*pitch);

    // each thread pulls one pixel into shared
    temp[(threadIdx.y*rows)+threadIdx.x] = img[(threadIdx.y*rows)+threadIdx.x];

    // build image integral per block (576 threads) via Kogge-Stone Parallel Scan Algo (w/o double buffering)
    for (int stride=1; stride<=max_stride; stride*=2) {
        __syncthreads();
        offset = threadIdx.x - stride;
        if (offset >= 0) {
            temp_val = temp[(threadIdx.y*rows)+threadIdx.x] + temp[(threadIdx.y*rows)+offset];
        }

        __syncthreads();
        if (offset >= 0) {
            temp[(threadIdx.y*rows)+threadIdx.x] = temp_val;
        }
    }

    img[(threadIdx.y*rows)+threadIdx.x] = temp[(threadIdx.y*rows)+threadIdx.x];

}
```

- use shared memory to store all the pixel values

Adding Double Buffering

```
__global__ void parallel_scan_shared_mem_db_pitched(float *images, int rows, size_t pitch) {
    // create shared memory arrays
    __shared__ float temp0[576];
    __shared__ float temp1[576];

    // get image for current block
    float *img = (float*) ((char*) images + blockIdx.x*pitch);

    // create pointers to shared memory arrays for double buffering
    float *source = temp0;
    float *dest = temp1;
    float *swap;

    int temp_val;
    float part_sum;
    int offset = 0;
    int max_stride = ceil(blockDim.x/2.0);

    // each thread pulls one pixel into shared
    temp_val = img[(threadIdx.y*rows)+threadIdx.x];
    temp0[(threadIdx.y*rows)+threadIdx.x] = temp_val;
    temp1[(threadIdx.y*rows)+threadIdx.x] = temp_val;

    // build image integral per block (576 threads) via Kogge-Stone Parallel Scan Algo (w/ double buffering)
    for (int stride=1; stride<=max_stride; stride*=2) {
        __syncthreads();
        offset = threadIdx.x - stride;
        part_sum = source[(threadIdx.y*rows)+threadIdx.x];

        if (offset >= 0) {
            part_sum += source[(threadIdx.y*rows)+offset];
        }

        dest[(threadIdx.y*rows)+threadIdx.x] = part_sum;

        swap = dest;
        dest = source;
        source = swap;
    }

    img[(threadIdx.y*rows)+threadIdx.x] = source[(threadIdx.y*rows)+threadIdx.x];
}
```

- Use two shared memory buffers
- Each thread copies its pixel value from the image to both buffers
- One starts as the source buffer and the other is used as the destination buffer to store the intermediate results

*<u>Caveat</u>: need to copy over current value from source to destination regardless if the current thread is performing an addition

Using pointers instead of pitched data copies

```
__global__ void parallel_scan_shared_mem_db(struct Mat *images, int rows) {
    // create shared memory arrays
    __shared__ float temp0[576];
    __shared__ float temp1[576];

    // create pointers to shared memory arrays for double buffering
    float *source = temp0;
    float *dest = temp1;
    float *swap;

    int temp_val;
    float part_sum;
    int offset = 0;
    int max_stride = ceil(blockDim.x/2.0);

    // each thread pulls one pixel into shared
    temp_val = images[blockIdx.x].values[(threadIdx.y*rows)+threadIdx.x];
    temp0[(threadIdx.y*rows)+threadIdx.x] = temp_val;
    temp1[(threadIdx.y*rows)+threadIdx.x] = temp_val;

    // build image integral per block (576 threads) via Kogge-Stone Parallel Scan Algo (w/ double buffering)
    for (int stride=1; stride<=max_stride; stride*=2) {
        __syncthreads();
        offset = threadIdx.x - stride;
        part_sum = source[(threadIdx.y*rows)+threadIdx.x];

        if (offset >= 0) {
            part_sum += source[(threadIdx.y*rows)+offset];
        }

        dest[(threadIdx.y*rows)+threadIdx.x] = part_sum;

        swap = dest;
        dest = source;
        source = swap;
    }

    images[blockIdx.x].values[(threadIdx.y*rows)+threadIdx.x] = source[(threadIdx.y*rows)+threadIdx.x];
}
```

```
struct Mat {
        float *values;
        int rows;
        int cols;
};
```

Transposition

```
__global__ void transpose_pitched(float *images, size_t pitch) {

    // get image for current block
    float *img = (float*) ((char*) images + blockIdx.x*pitch);

    float temp = img[(threadIdx.y*blockDim.x) + threadIdx.x];

    __syncthreads();

    img[(threadIdx.x*blockDim.x) + threadIdx.y] = temp;
}

__global__ void transpose(struct Mat *images) {

    float temp = images[blockIdx.x].values[(threadIdx.y*blockDim.x) + threadIdx.x];

    __syncthreads();

    images[blockIdx.x].values[(threadIdx.x*blockDim.x)+threadIdx.y] = temp;
}
```

Feature Building

```c
struct section {
    int row;
    int col;
    int height;
    int width;
    int sign;
};

struct feature {
    struct section *sections;
    int num_sections;
};
```

- Feature is made up of several sections (storing the count for later use)
- A section is a description of a rectangular section of a feature with a sign value (-1 or 1)

AdaBoosting

- initialize the weights
- for each weak classifier we want to obtain
    - normalize the weights
    - train each feature on the samples
    - choose the classifier with the lowest error
    - update the weights
- put together final strong classifier

- Given example images $(x_1, y_1), \ldots, (x_n, y_n)$ where $y_i = 0, 1$ for negative and positive examples respectively.
- Initialize weights $w_{1,i} = \frac{1}{2m}, \frac{1}{2l}$ for $y_i = 0, 1$ respectively, where m and l are the number of negatives and positives respectively.
- For $t = 1, \ldots, T$:
    1. Normalize the weights,
    $$w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$$
    so that w_t is a probability distribution.
    2. For each feature, j, train a classifier h_j which is restricted to using a single feature. The error is evaluated with respect to w_t, $\epsilon_j = \sum_i w_i \left| h_j(x_i) - y_i \right|$.
    3. Choose the classifier, h_t, with the lowest error ϵ_t.
    4. Update the weights:
    $$w_{t+1,i} = w_{t,i} \beta_t^{1-e_i}$$
    where $e_i = 0$ if example x_i is classified correctly, $e_i = 1$ otherwise, and $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$.
- The final strong classifier is:
$$h(x) = \begin{cases} 1 & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & \text{otherwise} \end{cases}$$
where $\alpha_t = \log \frac{1}{\beta_t}$

AdaBoosting

- initialize the weights
- for each weak classifier we want to obtain
    - normalize the weights
    - train each feature on the samples
    - choose the classifier with the lowest error
    - update the weights
- put together final strong classifier

- Given example images $(x_1, y_1), \ldots, (x_n, y_n)$ where $y_i = 0, 1$ for negative and positive examples respectively.
- Initialize weights $w_{1,i} = \frac{1}{2m}, \frac{1}{2l}$ for $y_i = 0, 1$ respectively, where m and l are the number of negatives and positives respectively.
- For $t = 1, \ldots, T$:
    1. Normalize the weights,
    $$w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$$
    so that w_t is a probability distribution.
    2. For each feature, j, train a classifier h_j which is restricted to using a single feature. The error is evaluated with respect to w_t, $\epsilon_j = \sum_i w_i |h_j(x_i) - y_i|$.
    3. Choose the classifier, h_t, with the lowest error ϵ_t.
    4. Update the weights:
    $$w_{t+1,i} = w_{t,i}\beta_t^{1-e_i}$$
    where $e_i = 0$ if example x_i is classified correctly, $e_i = 1$ otherwise, and $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$.
- The final strong classifier is:
$$h(x) = \begin{cases} 1 & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & \text{otherwise} \end{cases}$$
where $\alpha_t = \log \frac{1}{\beta_t}$

1. train each feature
2. sort the (results,weights) vector
3. build the cumulative weights for the feature
4. set the best threshold and parity for the classifier
5. run classification and report back the errors

1. Training

```
    for (int k=0; k<total_features; ++k) {
        for (int i=0; i<data.total_samples; ++i) {
            struct feature feat = weak_classifiers[k].feat;
            struct Mat img = data.images[i];

            int num_sections = feat.num_sections;
            struct section sect;
            sum = 0.0;

            for (int s=0; s<num_sections; ++s) {
                sect = feat.sections[s];
                A = ((sect.row*rows) + sect.col) - (rows + 1);
                B = ((sect.row*rows) + (sect.col+sect.width-1)) - rows;
                C = (((sect.row+sect.height-1)*rows) + sect.col) - 1;
                D = (((sect.row+sect.height-1)*rows) + (sect.col+sect.width-1));

                sum += sect.sign * (img.values[D] - img.values[B] - img.values[C] + img.values[A]);
            }

            res[k][i][0] = sum;
            res[k][i][1] = weights[i];
        }
    }
```

- For each feature and data sample
    - loop through each section of the feature
    - apply the section to the integral image
    - store the pair of resulting sum of all sections and the weight in a list

1. Training on the GPU

```
__global__ void train(float *res, struct weak_classifier *weak_classifiers, float *images, int rows, size_t pitch, int total_samples, int total_features) {
    int sample_ind = ((blockIdx.x*blockDim.x) + threadIdx.x) % total_samples;
    int feat_ind = floorf(((blockIdx.x*blockDim.x) + threadIdx.x) / (float)total_samples);

    if (feat_ind < total_features) {
        struct feature feat = weak_classifiers[feat_ind].feat;
        float *img = (float*) ((char*) images + sample_ind*pitch);

        int A,B,C,D;
        int num_sections = feat.num_sections;
        struct section sect;
        float sum = 0.0;

        for (int s=0; s<num_sections; ++s) {
            sect = feat.sections[s];
            A = ((sect.row*rows) + sect.col) - (rows + 1);
            B = ((sect.row*rows) + (sect.col+sect.width-1)) - rows;
            C = (((sect.row+sect.height-1)*rows) + sect.col) - 1;
            D = (((sect.row+sect.height-1)*rows) + (sect.col+sect.width-1));

            sum += sect.sign * (img[D] - img[B] - img[C] + img[A]);

        }

        res[(feat_ind*total_samples*2) + (sample_ind * 2)] = sum;
        res[(feat_ind*total_samples*2) + (sample_ind * 2) + 1] = weights_d[sample_ind];
    }
}
```

Block_size = 1024
Grid_size = ceil((total_samples*total_features) / block_size)

1. Training Using Shared Memory

```
__global__ void train_shared(float *res, struct weak_classifier *weak_classifiers,
        float *images, int rows, size_t pitch, int total_samples, int total_features, int iteration) {
    int sample_ind = (iteration*blockDim.x) + threadIdx.x;
    int feat_ind = blockIdx.x;

    if (sample_ind < total_samples) {

        __shared__ struct weak_classifier wc;

        if(threadIdx.x==0) {
            wc = weak_classifiers[feat_ind];
        }
        __syncthreads();

        float *img = (float*) ((char*) images + sample_ind*pitch);

        int A,B,C,D;
        int num_sections = wc.feat.num_sections;
        struct section sect;
        float sum = 0.0;

        for (int s=0; s<num_sections; ++s) {
            sect = wc.feat.sections[s];
            A = ((sect.row*rows) + sect.col) - (rows + 1);
            B = ((sect.row*rows) + (sect.col+sect.width-1)) - rows;
            C = (((sect.row+sect.height-1)*rows) + sect.col) - 1;
            D = (((sect.row+sect.height-1)*rows) + (sect.col+sect.width-1));

            sum += sect.sign * (img[D] - img[B] - img[C] + img[A]);

        }

        res[(feat_ind*total_samples*2) + (sample_ind * 2)] = sum;
        res[(feat_ind*total_samples*2) + (sample_ind * 2) + 1] = weights_d[sample_ind];
    }
}
```

```
    // launch kernel
    total_iters = (int) ceil(data.total_samples/(float)block_size);
    for (int i=0; i<total_iters; ++i) {
        train_shared<<<grid_size,block_size>>>(res_d, wcs_d,
                images_d, rows, dpitch, data.total_samples, total_features, i);
    }
```

5. Classification

```
for (int k=0; k<total_features; ++k) {
    error = 0.0;
    struct feature feat = weak_classifiers[k].feat;
    struct section sect;
    num_sects = feat.num_sections;

    for (int i=0; i<data.total_samples; ++i) {
        struct Mat img = data.images[i];
        sum = 0.0;

        for (int s=0; s<num_sects; ++s) {
            sect = feat.sections[s];
            A = ((sect.row*rows) + sect.col) - (rows + 1);
            B = ((sect.row*rows) + (sect.col+sect.width-1)) - rows;
            C = (((sect.row+sect.height-1)*rows) + sect.col) - 1;
            D = (((sect.row+sect.height-1)*rows) + (sect.col+sect.width-1));

            sum += sect.sign * (img.values[D] - img.values[B] - img.values[C] + img.values[A]);
        }

        label = (sum * weak_classifiers[k].parity) < (weak_classifiers[k].thresh * weak_classifiers[k].parity);

        results[k][i] = abs(label-(data.labels[i]));
        error += (weights[i] * results[k][i]);

    }

    errors[k] = error;
}
```

$$h_j(x) = \begin{cases} 1 & \text{if } p_j f_j(x) < p_j \theta_j \\ 0 & \text{otherwise} \end{cases}$$

- For each feature and data sample
    - loop through each section of the feature
    - apply the section to the integral image
    - calculate label
    - store result
    - store classification error

5. Classification on the GPU

```
__global__ void classify(float *errors, int *results, struct weak_classifier *weak_classifiers,
        float *images, int *labels, int rows, size_t pitch, int total_samples, int total_features) {
    int sample_ind = ((blockIdx.x*blockDim.x) + threadIdx.x) % total_samples;
    int feat_ind = floorf(((blockIdx.x*blockDim.x) + threadIdx.x) / (float)total_samples);

    if (feat_ind < total_features) {
        weak_classifier wc = weak_classifiers[feat_ind];
        float *img = (float*) ((char*) images + sample_ind*pitch);

        int A,B,C,D;
        int num_sections = wc.feat.num_sections;
        struct section sect;
        int label;
        int res;
        float sum = 0.0;

        for (int s=0; s<num_sections; ++s) {
            sect = wc.feat.sections[s];
            A = ((sect.row*rows) + sect.col) - (rows + 1);
            B = ((sect.row*rows) + (sect.col+sect.width-1)) - rows;
            C = (((sect.row+sect.height-1)*rows) + sect.col) - 1;
            D = (((sect.row+sect.height-1)*rows) + (sect.col+sect.width-1));

            sum += sect.sign * (img[D] - img[B] - img[C] + img[A]);
        }

        label = (sum * wc.parity) < (wc.thresh * wc.parity);

        res = abs(label-(labels[sample_ind]));
        results[(feat_ind*total_samples) + sample_ind] = res;
        errors[(feat_ind*total_samples) + sample_ind] = (weights_d[sample_ind] * res);
    }
}
```

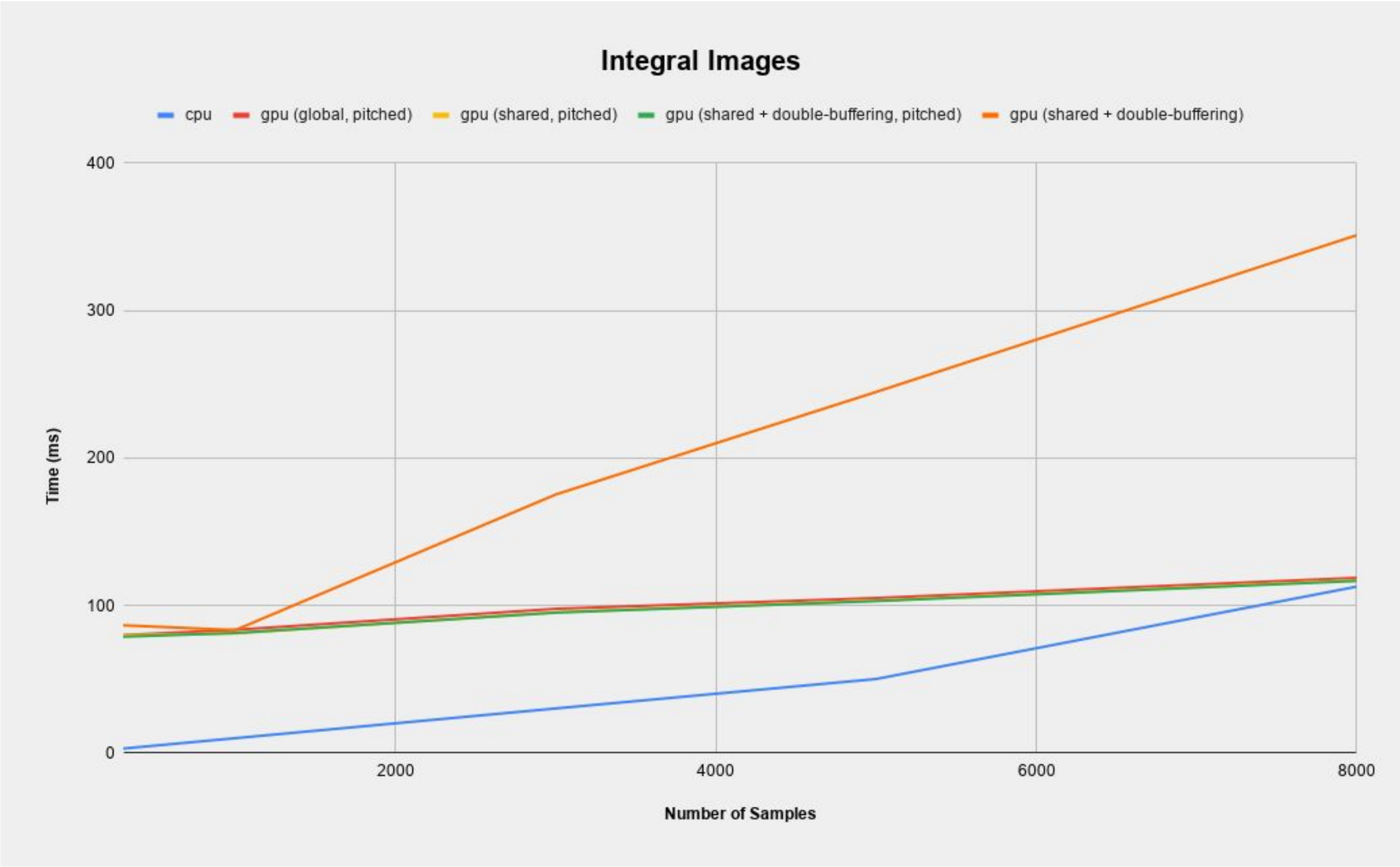
Block_size = 1024
Grid_size =
ceil((total_samples*total_features)/block_size)

Classification Using Shared Memory

```
__global__ void classify_shared(float *errors, int *results, struct weak_classifier *weak_classifiers,
        float *images, int *labels, int rows, size_t pitch, int total_samples, int total_features, int iteration) {
    int sample_ind = (iteration*blockDim.x) + threadIdx.x;
    int feat_ind = blockIdx.x;

    if (sample_ind < total_samples) {

        __shared__ struct weak_classifier wc;

        if(threadIdx.x==0) {
            wc = weak_classifiers[feat_ind];
        }
        __syncthreads();

        float *img = (float*) ((char*) images + sample_ind*pitch);

        int A,B,C,D;
        int label;
        int res;
        int num_sections = wc.feat.num_sections;
        struct section sect;
        float sum = 0.0;

        for (int s=0; s<num_sections; ++s) {
            sect = wc.feat.sections[s];
            A = ((sect.row*rows) + sect.col) - (rows + 1);
            B = ((sect.row*rows) + (sect.col+sect.width-1)) - rows;
            C = (((sect.row+sect.height-1)*rows) + sect.col) - 1;
            D = (((sect.row+sect.height-1)*rows) + (sect.col+sect.width-1));

            sum += sect.sign * (img[D] - img[B] - img[C] + img[A]);

        }

        label = (sum * wc.parity) < (wc.thresh * wc.parity);

        res = abs(label-(labels[sample_ind]));
        results[(feat_ind*total_samples) + sample_ind] = res;
        errors[(feat_ind*total_samples) + sample_ind] = (weights_d[sample_ind] * res);
    }
}
```

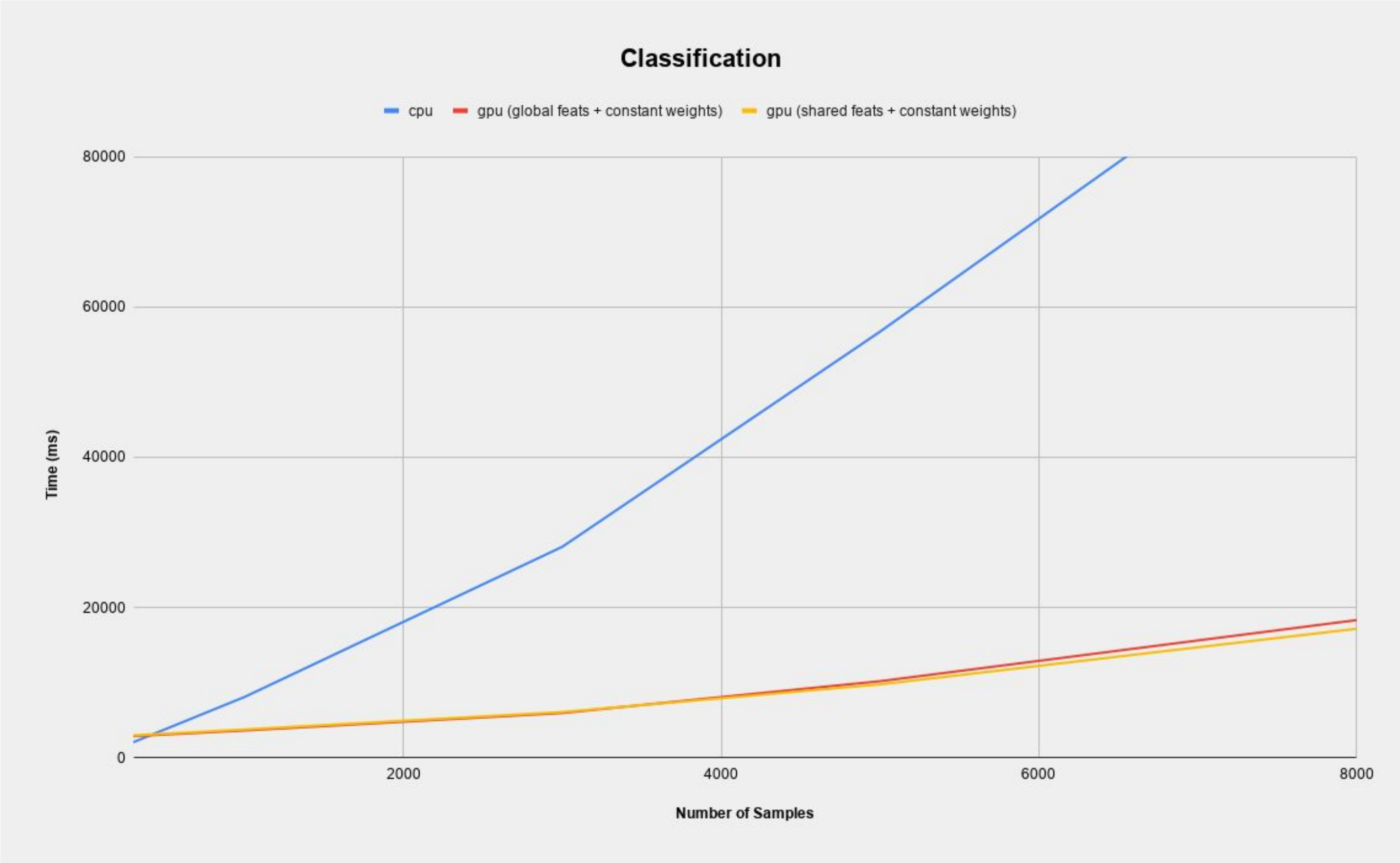
```
    // launch kernel
    total_iters = (int) ceil(data.total_samples/(float)block_size);
    for (int i=0; i<total_iters; ++i) {
        classify_shared<<<grid_size,block_size>>>(errors_d, results_d, wcs_d,
                images_d, labels_d, rows, dpitch, data.total_samples, total_features, i);
    }
```

Results



Integral Images
cpu
gpu (global, pitched)
gpu (shared, pitched)
gpu (shared + double-buffering, pitched)
gpu (shared + double-buffering)
400
300
200
100
0
Time (ms)
2000
4000
6000
8000
Number of Samples

Training
cpu
gpu (global feats + constant weights)
gpu (shared feats + constant weights)
80000
60000
40000
20000
0
Time (ms)
2000
4000
6000
8000
Number of Samples


Classification
cpu
gpu (global feats + constant weights)
gpu (shared feats + constant weights)
80000
60000
40000
20000
0
Time (ms)
2000
4000
6000
8000
Number of Samples