

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

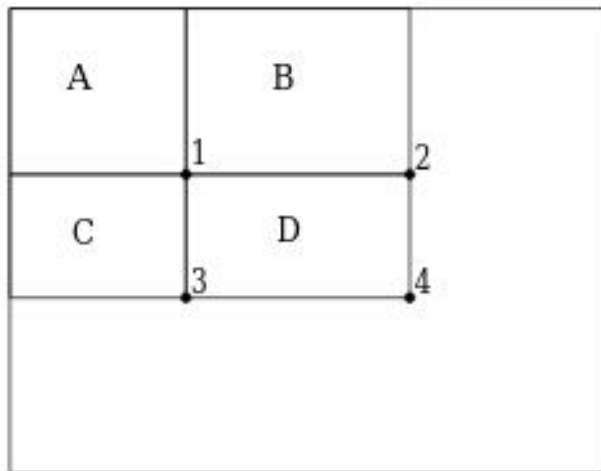
Viola-Jones Face Detection - Overview

Goal: 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

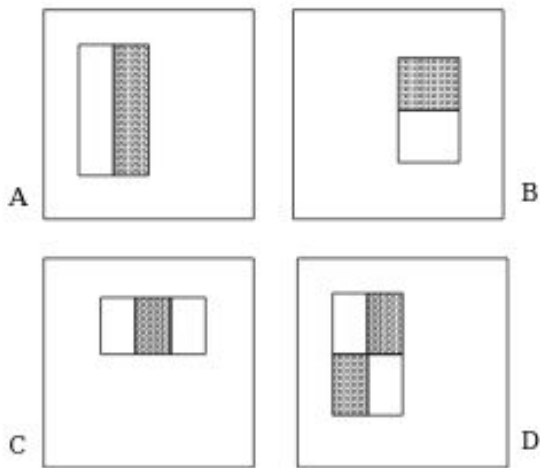
- An intermediate representation of an image
- Each pixel in the integral image is the inclusive sum of all pixels above and to the left



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

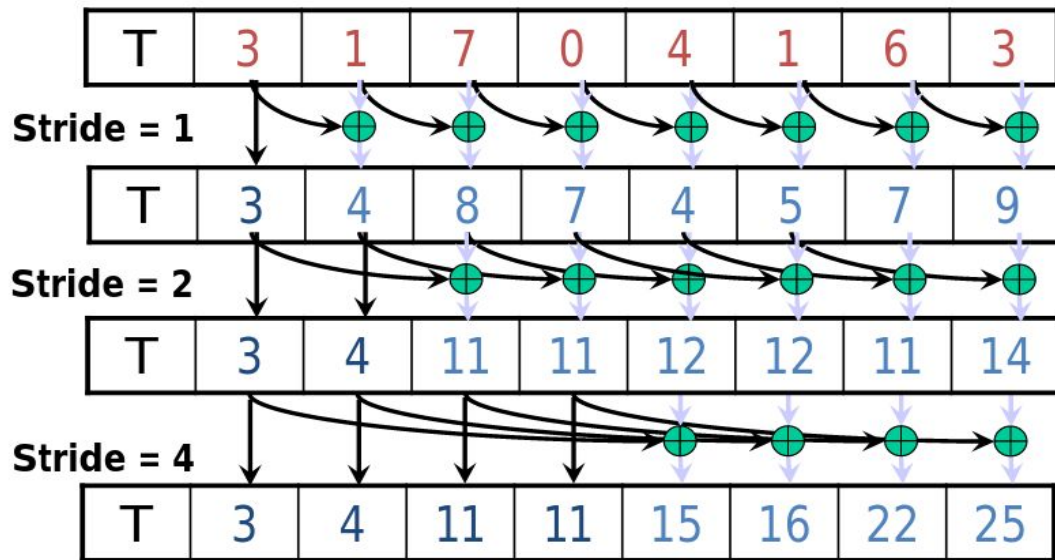


Computing Integral Images

```
157 struct Mat* compute_integrals(struct Mat *images, int total_samples) {
158     int rows;
159     int cols;
160     struct Mat *integral_images;
161
162     rows = images[0].rows;
163     cols = images[0].cols;
164
165     float row_sum[rows][cols];
166
167     integral_images = (struct Mat*) malloc(sizeof(struct Mat) * total_samples);
168
169     for (int i=0; i<total_samples; ++i) {
170         integral_images[i].values = (float*) malloc(sizeof(float) * rows*cols);
171         integral_images[i].rows = images[i].rows;
172         integral_images[i].cols = images[i].cols;
173
174         for (int r=0; r<rows; ++r){
175             for (int c=0; c<cols; ++c) {
176                 row_sum[r][c] = images[i].values[(r*rows)+c];
177
178                 if (c>0) {
179                     row_sum[r][c] += row_sum[r][c-1];
180                 }
181
182                 integral_images[i].values[(r*rows)+c] = row_sum[r][c];
183
184                 if (r>0) {
185                     integral_images[i].values[(r*rows)+c] += integral_images[i].values[((r-1)*rows)+c];
186                 }
187             }
188         }
189     }
190
191     return integral_images;
192 }
```

- 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 $\text{ceil}(n/2.0)$. Threads *stride* to $n-1$ active: add pairs of elements that are *stride* elements apart.
3. Write output from shared memory to device memory

Iteration #3
Stride = 4

Initial Implementation

```
24 __global__ void parallel_scan_pitched(float *images, int rows, int cols, size_t pitch) {
25
26     float temp_val = 0.0;
27     int offset = 0;
28     int max_stride = ceil(blockDim.x/2.0);
29
30     // get image for current block
31     float *img = (float*) ((char*) images + blockIdx.x*pitch);
32
33     // build image integral per block (576 threads) via Kogge-Stone Parallel Scan Algo (w/o double buffering)
34     //#pragma unroll 1
35     for (int stride=1; stride<=max_stride; stride*=2) {
36         __syncthreads();
37         offset = threadIdx.x - stride;
38         if (offset >= 0) {
39             temp_val = img[(threadIdx.y*rows)+threadIdx.x] + img[(threadIdx.y*rows)+offset];
40         }
41
42         __syncthreads();
43         if (offset >= 0) {
44             img[(threadIdx.y*rows)+threadIdx.x] = temp_val;
45         }
46     }
47 }
```

- 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

```
48 __global__ void parallel_scan_shared_mem_sb_pitched(float *images, int rows, size_t pitch) {
49
50     // create shared memory array
51     __shared__ float temp[576];
52
53     float temp_val = 0.0;
54     int offset = 0;
55     int max_stride = ceil(blockDim.x/2.0);
56
57     // get image for current block
58     float *img = (float*) ((char*) images + blockIdx.x*pitch);
59
60     // each thread pulls one pixel into shared
61     temp[(threadIdx.y*rows)+threadIdx.x] = img[(threadIdx.y*rows)+threadIdx.x];
62
63     // build image integral per block (576 threads) via Kogge-Stone Parallel Scan Algo (w/o double buffering)
64     for (int stride=1; stride<=max_stride; stride*=2) {
65         __syncthreads();
66         offset = threadIdx.x - stride;
67         if (offset >= 0) {
68             temp_val = temp[(threadIdx.y*rows)+threadIdx.x] + temp[(threadIdx.y*rows)+offset];
69         }
70
71         __syncthreads();
72         if (offset >= 0) {
73             temp[(threadIdx.y*rows)+threadIdx.x] = temp_val;
74         }
75     }
76
77     img[(threadIdx.y*rows)+threadIdx.x] = temp[(threadIdx.y*rows)+threadIdx.x];
78
79 }
```

- use shared memory to store
all the pixel values

Adding Double Buffering

```
81 __global__ void parallel_scan_shared_mem_db_pitched(float *images, int rows, size_t pitch) {
82     // create shared memory arrays
83     __shared__ float temp0[576];
84     __shared__ float temp1[576];
85
86     // get image for current block
87     float *img = (float*) ((char*) images + blockIdx.x*pitch);
88
89     // create pointers to shared memory arrays for double buffering
90     float *source = temp0;
91     float *dest = temp1;
92     float *swap;
93
94     int temp_val;
95     float part_sum;
96     int offset = 0;
97     int max_stride = ceil(blockDim.x/2.0);
98
99     // each thread pulls one pixel into shared
100     temp_val = img[(threadIdx.y*rows)+threadIdx.x];
101     temp0[(threadIdx.y*rows)+threadIdx.x] = temp_val;
102     temp1[(threadIdx.y*rows)+threadIdx.x] = temp_val;
103
104     // build image integral per block (576 threads) via Kogge-Stone Parallel Scan Algo (w/ double buffering)
105     for (int stride=1; stride<=max_stride; stride*=2) {
106         __syncthreads();
107         offset = threadIdx.x - stride;
108         part_sum = source[(threadIdx.y*rows)+threadIdx.x];
109
110         if (offset >= 0) {
111             part_sum += source[(threadIdx.y*rows)+offset];
112         }
113
114         dest[(threadIdx.y*rows)+threadIdx.x] = part_sum;
115
116         swap = dest;
117         dest = source;
118         source = swap;
119     }
120
121     img[(threadIdx.y*rows)+threadIdx.x] = source[(threadIdx.y*rows)+threadIdx.x];
122 }
```

- 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
- *Caveat: 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

```
125 __global__ void parallel_scan_shared_mem_db(struct Mat *images, int rows) {
126     // create shared memory arrays
127     __shared__ float temp0[576];
128     __shared__ float temp1[576];
129
130     // create pointers to shared memory arrays for double buffering
131     float *source = temp0;
132     float *dest = temp1;
133     float *swap;
134
135     int temp_val;
136     float part_sum;
137     int offset = 0;
138     int max_stride = ceil(blockDim.x/2.0);
139
140     // each thread pulls one pixel into shared
141     temp_val = images[blockIdx.x].values[(threadIdx.y*rows)+threadIdx.x];
142     temp0[(threadIdx.y*rows)+threadIdx.x] = temp_val;
143     temp1[(threadIdx.y*rows)+threadIdx.x] = temp_val;
144
145     // build image integral per block (576 threads) via Kogge-Stone Parallel Scan Algo (w/ double buffering)
146     for (int stride=1; stride<=max_stride; stride*=2) {
147         __syncthreads();
148         offset = threadIdx.x - stride;
149         part_sum = source[(threadIdx.y*rows)+threadIdx.x];
150
151         if (offset >= 0) {
152             part_sum += source[(threadIdx.y*rows)+offset];
153         }
154
155         dest[(threadIdx.y*rows)+threadIdx.x] = part_sum;
156
157         swap = dest;
158         dest = source;
159         source = swap;
160     }
161
162     images[blockIdx.x].values[(threadIdx.y*rows)+threadIdx.x] = source[(threadIdx.y*rows)+threadIdx.x];
163 }
164 }
```

```
17 struct Mat {
18     float *values;
19     int rows;
20     int cols;
21 };
22 }
```


Transposition

```
3 __global__ void transpose_patched(float *images, size_t pitch) {
4
5     // get image for current block
6     float *img = (float*) ((char*) images + blockIdx.x*pitch);
7
8     float temp = img[(threadIdx.y*blockDim.x) + threadIdx.x];
9
10    __syncthreads();
11
12    img[(threadIdx.x*blockDim.x) + threadIdx.y] = temp;
13 }
14
15 __global__ void transpose(struct Mat *images) {
16
17     float temp = images[blockIdx.x].values[(threadIdx.y*blockDim.x) + threadIdx.x];
18
19     __syncthreads();
20
21     images[blockIdx.x].values[(threadIdx.x*blockDim.x)+threadIdx.y] = temp;
22 }
23
```

Feature Building

```
23 struct section {  
24     int row;  
25     int col;  
26     int height;  
27     int width;  
28     int sign;  
29 };  
30  
31 struct feature {  
32     struct section *sections;  
33     int num_sections;  
34 };  
35
```

- 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 $(x_1, y_1), \dots, (x_n, y_n)$ 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 $t = 1, \dots, T$:

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 $\alpha_t = \log \frac{1}{\beta_t}$

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 $(x_1, y_1), \dots, (x_n, y_n)$ 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 $t = 1, \dots, T$:

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

```
664 for (int k=0; k<total_features; ++k) {
665     for (int i=0; i<data.total_samples; ++i) {
666         struct feature feat = weak_classifiers[k].feat;
667         struct Mat img = data.images[i];
668
669         int num_sections = feat.num_sections;
670         struct section sect;
671         sum = 0.0;
672
673         for (int s=0; s<num_sections; ++s) {
674             sect = feat.sections[s];
675             A = ((sect.row*rows) + sect.col) - (rows + 1);
676             B = ((sect.row*rows) + (sect.col+sect.width-1)) - rows;
677             C = (((sect.row+sect.height-1)*rows) + sect.col) - 1;
678             D = (((sect.row+sect.height-1)*rows) + (sect.col+sect.width-1));
679
680             sum += sect.sign * (img.values[D] - img.values[B] - img.values[C] + img.values[A]);
681         }
682
683         res[k][i][0] = sum;
684         res[k][i][1] = weights[i];
685     }
686 }
```

- 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

```
5 __global__ void train(float *res, struct weak_classifier *weak_classifiers, float *images, int rows, size_t pitch, int total_samples, int total_features) {
6     int sample_ind = ((blockIdx.x*blockDim.x) + threadIdx.x) % total_samples;
7     int feat_ind = floorf(((blockIdx.x*blockDim.x) + threadIdx.x) / (float)total_samples);
8
9     if (feat_ind < total_features) {
10         struct feature feat = weak_classifiers[feat_ind].feat;
11         float *img = (float*) ((char*) images + sample_ind*pitch);
12
13         int A,B,C,D;
14         int num_sections = feat.num_sections;
15         struct section sect;
16         float sum = 0.0;
17
18         for (int s=0; s<num_sections; ++s) {
19             sect = feat.sections[s];
20             A = ((sect.row*rows) + sect.col) - (rows + 1);
21             B = ((sect.row*rows) + (sect.col+sect.width-1)) - rows;
22             C = (((sect.row+sect.height-1)*rows) + sect.col) - 1;
23             D = (((sect.row+sect.height-1)*rows) + (sect.col+sect.width-1));
24
25             sum += sect.sign * (img[D] - img[B] - img[C] + img[A]);
26
27         }
28
29         res[(feat_ind*total_samples*2) + (sample_ind * 2)] = sum;
30         res[(feat_ind*total_samples*2) + (sample_ind * 2) + 1] = weights_d[sample_ind];
31     }
32 }
```

Block_size = 1024

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

1. Training Using Shared Memory

```
34 __global__ void train_shared(float *res, struct weak_classifier *weak_classifiers,
35     float *images, int rows, size_t pitch, int total_samples, int total_features, int iteration) {
36     int sample_ind = (iteration*blockDim.x) + threadIdx.x;
37     int feat_ind = blockDim.x;
38
39     if (sample_ind < total_samples) {
40
41         __shared__ struct weak_classifier wc;
42
43         if (threadIdx.x==0) {
44             wc = weak_classifiers[feat_ind];
45         }
46         __syncthreads();
47
48         float *img = (float*) ((char*) images + sample_ind*pitch);
49
50         int A,B,C,D;
51         int num_sections = wc.feat.num_sections;
52         struct section sect;
53         float sum = 0.0;
54
55         for (int s=0; s<num_sections; ++s) {
56             sect = wc.feat.sections[s];
57             A = ((sect.row*rows) + sect.col) - (rows + 1);
58             B = ((sect.row*rows) + (sect.col+sect.width-1)) - rows;
59             C = (((sect.row+sect.height-1)*rows) + sect.col) - 1;
60             D = (((sect.row+sect.height-1)*rows) + (sect.col+sect.width-1));
61
62             sum += sect.sign * (img[D] - img[B] - img[C] + img[A]);
63
64         }
65
66         res[(feat_ind*total_samples*2) + (sample_ind * 2)] = sum;
67         res[(feat_ind*total_samples*2) + (sample_ind * 2) + 1] = weights_d[sample_ind];
68     }
69 }
```

```
622 // launch kernel
623 total_iters = (int) ceil(data.total_samples/(float)block_size);
624 for (int i=0; i<total_iters; ++i) {
625     train_shared<<<grid_size,block_size>>>(res_d, wcs_d,
626         images_d, rows, dpitch, data.total_samples, total_features, i);
627 }
628 }
```

5. Classification

```
803 for (int k=0; k<total_features; ++k) {
804     error = 0.0;
805     struct feature feat = weak_classifiers[k].feat;
806     struct section sect;
807     num_sects = feat.num_sections;
808
809     for (int i=0; i<data.total_samples; ++i) {
810         struct Mat img = data.images[i];
811         sum = 0.0;
812
813         for (int s=0; s<num_sects; ++s) {
814             sect = feat.sections[s];
815             A = ((sect.row*rows) + sect.col) - (rows + 1);
816             B = ((sect.row*rows) + (sect.col+sect.width-1)) - rows;
817             C = (((sect.row+sect.height-1)*rows) + sect.col) - 1;
818             D = (((sect.row+sect.height-1)*rows) + (sect.col+sect.width-1));
819
820             sum += sect.sign * (img.values[D] - img.values[B] - img.values[C] + img.values[A]);
821         }
822
823         label = (sum * weak_classifiers[k].parity) < (weak_classifiers[k].thresh * weak_classifiers[k].parity);
824
825         results[k][i] = abs(label-(data.labels[i]));
826         error += (weights[i] * results[k][i]);
827     }
828
829     errors[k] = error;
830 }
831 }
```

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

- 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

```
70 __global__ void classify(float *errors, int *results, struct weak_classifier *weak_classifiers,
71 float *images, int *labels, int rows, size_t pitch, int total_samples, int total_features) {
72     int sample_ind = ((blockIdx.x*blockDim.x) + threadIdx.x) % total_samples;
73     int feat_ind = floorf(((blockIdx.x*blockDim.x) + threadIdx.x) / (float)total_samples);
74
75     if (feat_ind < total_features) {
76         weak_classifier wc = weak_classifiers[feat_ind];
77         float *img = (float*) ((char*) images + sample_ind*pitch);
78
79         int A,B,C,D;
80         int num_sections = wc.feats.num_sections;
81         struct section sect;
82         int label;
83         int res;
84         float sum = 0.0;
85
86         for (int s=0; s<num_sections; ++s) {
87             sect = wc.feats.sections[s];
88             A = ((sect.row*rows) + sect.col) - (rows + 1);
89             B = ((sect.row*rows) + (sect.col+sect.width-1)) - rows;
90             C = (((sect.row+sect.height-1)*rows) + sect.col) - 1;
91             D = (((sect.row+sect.height-1)*rows) + (sect.col+sect.width-1));
92
93             sum += sect.sign * (img[D] - img[B] - img[C] + img[A]);
94         }
95
96         label = (sum * wc.parity) < (wc.thresh * wc.parity);
97
98         res = abs(label-(labels[sample_ind]));
99         results[(feat_ind*total_samples) + sample_ind] = res;
100         errors[(feat_ind*total_samples) + sample_ind] = (weights_d[sample_ind] * res);
101     }
102 }
```

Block_size = 1024

Grid_size =

$\text{ceil}((\text{total_samples} * \text{total_features}) / \text{block_size})$

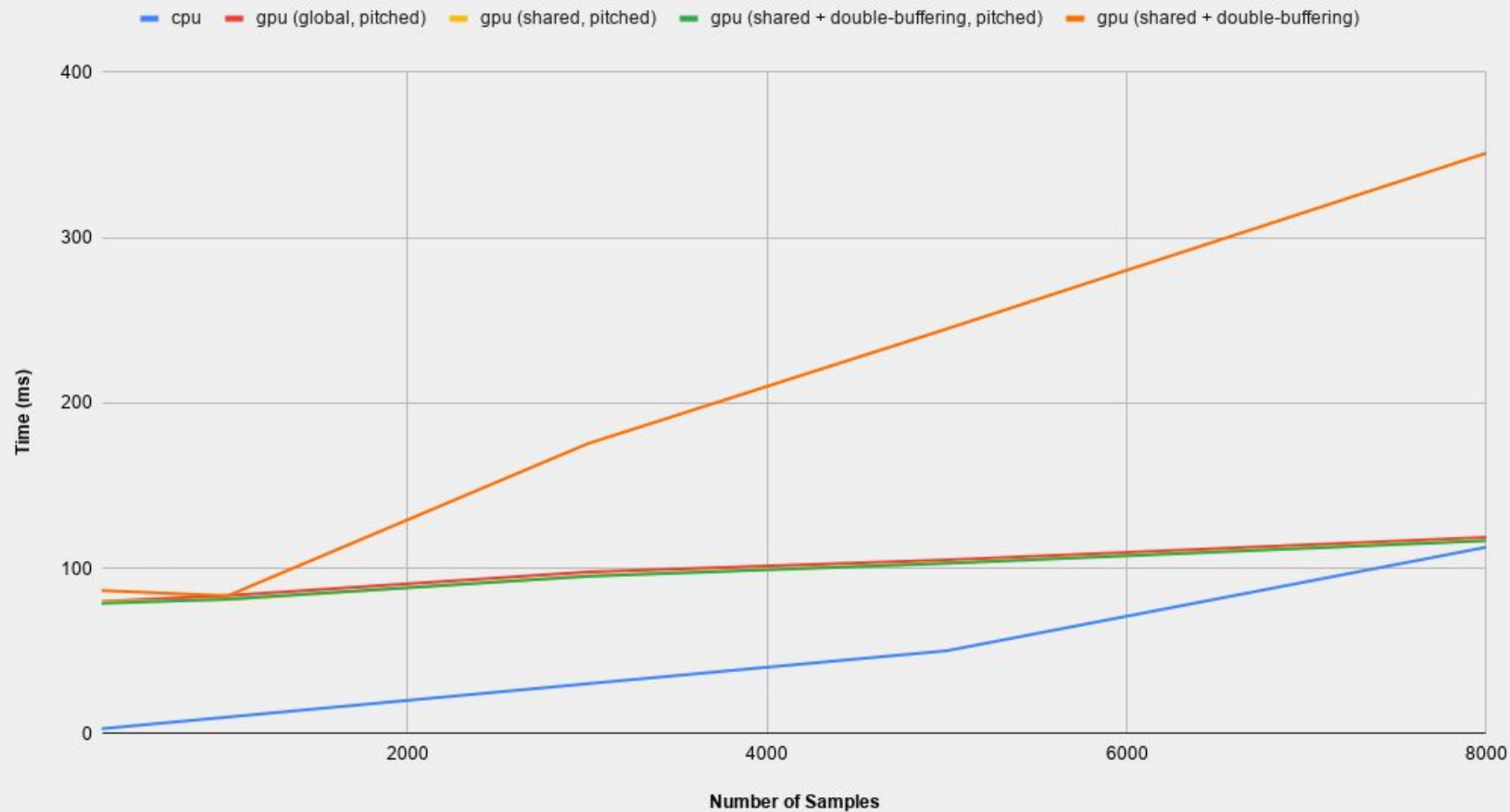
Classification Using Shared Memory

```
104 __global__ void classify_shared(float *errors, int *results, struct weak_classifier *weak_classifiers,
105                               float *images, int *labels, int rows, size_t pitch, int total_samples, int total_features, int iteration) {
106     int sample_ind = (iteration*blockDim.x) + threadIdx.x;
107     int feat_ind = blockIdx.x;
108
109     if (sample_ind < total_samples) {
110
111         __shared__ struct weak_classifier wc;
112
113         if (threadIdx.x==0) {
114             wc = weak_classifiers[feat_ind];
115         }
116         __syncthreads();
117
118         float *img = (float*) ((char*) images + sample_ind*pitch);
119
120         int A,B,C,D;
121         int label;
122         int res;
123         int num_sections = wc.feats.num_sections;
124         struct section sect;
125         float sum = 0.0;
126
127         for (int s=0; s<num_sections; ++s) {
128             sect = wc.feats.sections[s];
129             A = ((sect.row*rows) + sect.col) - (rows + 1);
130             B = ((sect.row*rows) + (sect.col+sect.width-1)) - rows;
131             C = (((sect.row+sect.height-1)*rows) + sect.col) - 1;
132             D = (((sect.row+sect.height-1)*rows) + (sect.col+sect.width-1));
133
134             sum += sect.sign * (img[D] - img[B] - img[C] + img[A]);
135
136         }
137
138         label = (sum * wc.parity) < (wc.thresh * wc.parity);
139
140         res = abs(label-(labels[sample_ind]));
141         results[(feat_ind*total_samples) + sample_ind] = res;
142         errors[(feat_ind*total_samples) + sample_ind] = (weights_d[sample_ind] * res);
143     }
144 }
```

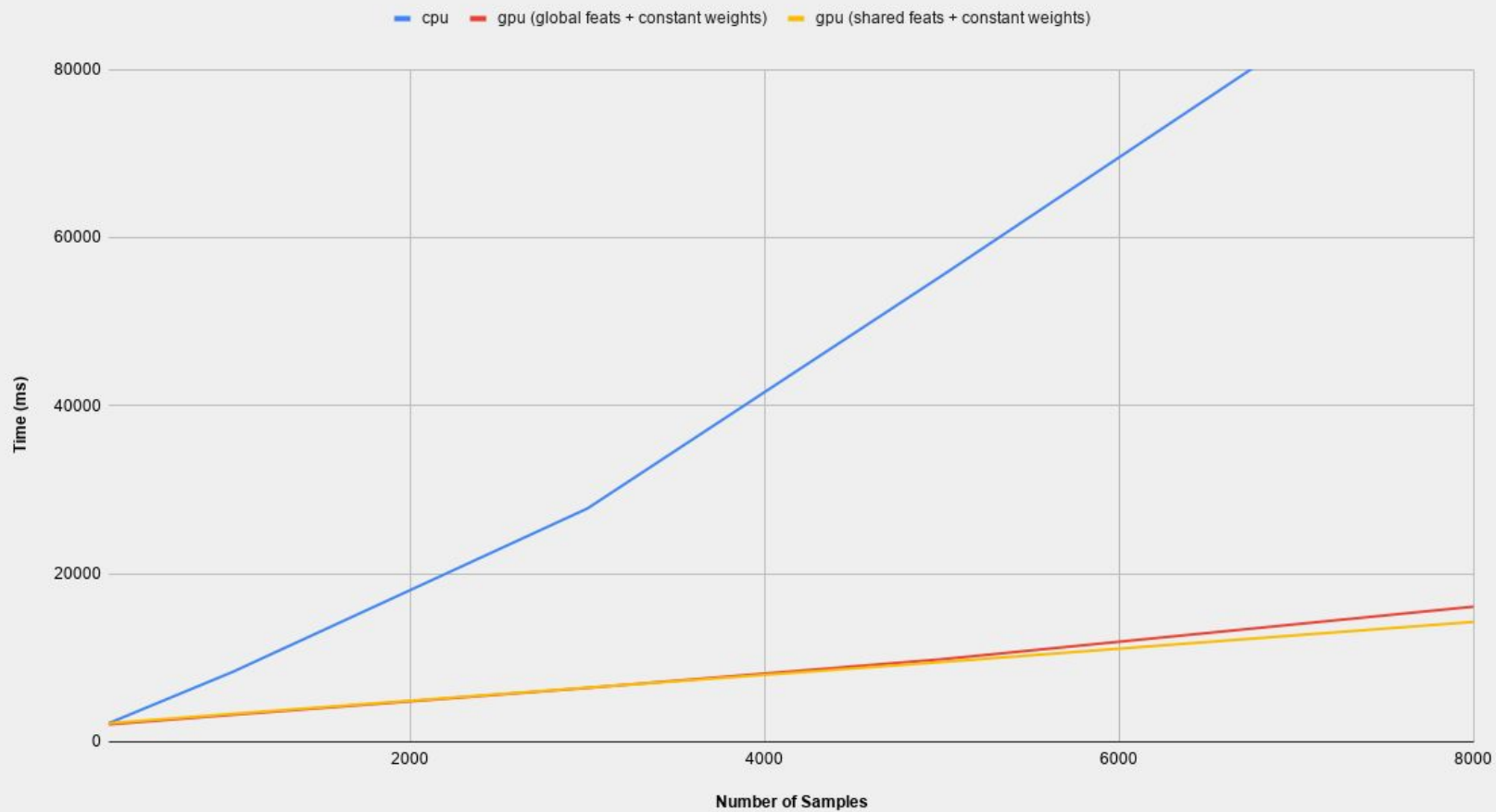
```
828 // launch kernel
829 total_iters = (int) ceil(data.total_samples/(float)block_size);
830 for (int i=0; i<total_iters; ++i) {
831     classify_shared<<<grid_size,block_size>>>>(errors_d, results_d, wcs_d,
832                                                images_d, labels_d, rows, dpitch, data.total_samples, total_features, i);
833 }
834 }
```


Results

Integral Images



Training



Classification

