HIGH PERFORMANCE PEDESTRIAN DETECTION ON TEGRA X1

Max Lv, NVIDIA
Brant Zhao, NVIDIA
April 7

mlv@nvidia.com
https://github.com/madeye
AGENDA

Histogram of Oriented Gradients on GPU
Optimization Opportunities on a Tegra GPU
Optimization #1: Improve ILP (Instruction Level Parallelism)
Optimization #2: Approximation
Optimization #3: Specialization
Final Results
PEDESTRIAN DETECTION: HOG DESCRIPTOR

Histogram of Oriented Gradients

Gradient-based feature descriptor developed for pedestrian detection

Introduced by Navneet Dalal and Bill Triggs (CVPR’05)

Global descriptor for the complete body

Very high-dimensional: typically ~4000 dimensions

Oriented Gradients: 3x3 Sobel filter with gamma correction

Block Histogram: Pixels vote in proportion to gradient magnitude, with a tri-linear interpolation, in each block (16x16 pixels)

Histograms Normalization: Normalize each block of histogram (36-bin)

Linear SVM: A linear SVM classifier, dot product of each window (7x15 36-bin normalized histograms) and trained coefficients
OPTIMIZATION OPPORTUNITIES
On a 2-SM Maxwell GPU in Tegra X1

Our goal is to improve the performance further based on a well-optimized implementation in VisionWorks.

Trade-offs between ILP (Instruction-level-parallelism) and DLP (Data-level-parallelism).

Trade-offs between precision and computation.

Trade-offs between generalization and specialization.

<table>
<thead>
<tr>
<th>NVIDIA Tegra X1 Maxwell GPU Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Cores</td>
</tr>
<tr>
<td>Texture Units</td>
</tr>
<tr>
<td>ROPs</td>
</tr>
<tr>
<td>GPU Clock</td>
</tr>
<tr>
<td>Memory Clock</td>
</tr>
<tr>
<td>Memory Bus Width</td>
</tr>
<tr>
<td>FP16 Peak</td>
</tr>
<tr>
<td>FP32 Peak</td>
</tr>
<tr>
<td>Architecture</td>
</tr>
</tbody>
</table>
OPTIMIZATION #1
Improve ILP (Instruction Level Parallelism)

Existed GPU kernels optimized for large GPU, improving DLP to saturate SMs

For small GPUs on Tegra, it’s possible to gain perf with larger ILP but smaller DLP

Increase workload in each thread while # of total threads decreases

Try different configs until the best perf is achieved
OPTIMIZATION #1
Example: Best ILP & DLP trade-off for Block Histograms

Various patterns to compute a block of histograms.

Best trade-off: Each thread calculates 3x12 pixels

Not work well on large GPUs like Titan X, but suitable for Tegra X1
32-bit float point of GPU is unnecessary for most of computer vision applications

`--use_fast_math` is enabled by default for our CV projects

Compute in float point, but load and store pixels in integer using `texture` instructions

Sometimes it’s safe to relax the precision even further

---

**OPTIMIZATION #2**

**Approximation**

Compute as FP16/FP32 in **SM**

0, 0.5, 1.0, ...

Conversion / (De)Normalization / Sampling

In **Texture**

0, 128, 255, ...

Store as 8-bit/16-bit Integer in **Memory**
A fast version of `atan2f()` with 3rd order Lagrange polynomial interpolation, and without handling corner cases

```c
float atan2f_lagrange_3rd(const float dy, const float dx) {
    float A = 0.0f, B = 0.0f;
    float Offset = copysignf(float(M_PI), dy);

    if (fabsf(dy) < fabsf(dx)) {
        A = dx; B = dy;
        if (dx >= 0.0f) Offset = 0.0f;
    } else {
        A = -dy; B = dx;
        Offset *= 0.5f;
    }

    const float r = B / A;
    const float p = 1.0f - fabsf(r);

    return ((-0.0663f*p + 0.311f) * p + float(M_PI/4.0)) * r + Offset;
}
```

Comparison between different `atan2f()` implementations:

<table>
<thead>
<tr>
<th></th>
<th>Native</th>
<th>This work</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMA/FADD (op)</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>MUFU.RCP (op)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Handle Corner Case (op)</td>
<td>~30</td>
<td>~5</td>
</tr>
<tr>
<td>Avg. Error (degree)</td>
<td>0.01</td>
<td>0.05</td>
</tr>
</tbody>
</table>
OPTIMIZATION #3
Specialization

Specialize parameters of CV applications to enable further optimization

Unroll the loop fully to eliminate index computation and conditional branches

Allow automatic register blocking by compiler, better instruction scheduling

Allow more tricks to reuse on-chip data

```c
__global__ void kernel(int N) {
    ...
    #pragma unroll
    for (int i = 0; i < N; i++) {
        if (i % 3) {
            ...
            ...
            tmp[i] += ...
        }
    }
    ...
}
```
OPTIMIZATION #3
Example: Transform Linear SVM to 36-layer 7x15 2D Convolutions

Dot products of \((7 \times 15 \times 36)\)-dimension vectors = Sum of 36-layer 7x15 2D convolutions

Load the whole patch to shared memory

Uniform loads of coefficients in constant memory, without any bank conflict

Reuse our well-optimized 2D convolution kernel (aggressive register blocking, GTC’15, Zhao et.al)
OPTIMIZATION #3
Example: Transform Linear SVM to 36-layer 7x15 2D Convolutions

2D convolution on 36 layers

Add up results of all layers

Each element is dot product of each window

Atomic Add =

\( \text{winPerImgX} \times \text{winPerImgY} \)
FINAL RESULTS

214 FPS on Tegra X1

Runtime (ms) of VGA input on Tegra X1, compared to the previous implementation of VisionWorks (https://developer.nvidia.com/embedded/visionworks)

- Oriented Gradients: 1.22 Base, 0.86 Optimized
- Block Histograms: 3.90 Base, 2.23 Optimized
- Histogram Normalization: 0.85 Base, 0.29 Optimized
- Linear SVM: 2.48 Base, 1.01 Optimized
- Overall: 8.73 Base, 4.67 Optimized

1.87x Speedup
THANK YOU

JOIN THE CONVERSATION
#GTC16  

mlv@nvidia.com
https://github.com/madeye
BACKUPS

JOIN THE CONVERSATION
#GTC16  

OPTIMIZATION #2
Example: Fast atan2f() for Oriented Gradients

Employ LOP3 (3-operand logic operations, new instruction of Maxwell arch)

```c
float atan2f_lagrange_3rd(const float dy, const float dx) {
    float flag, z = 0.0f;
    __SET_LT(flag, fabsf(dy), fabsf(dx));

    uint32_t m, t1 = 0x80000000; float t2 = float(M_PI) / 2.0f;
    __LOP3_0x2e(m, __float_as_int(dx), t1, __float_as_int(t2));

    float w = flag * __int_as_float(m) + float(M_PI)/2.0f; float Offset = copysignf(w, dy);
    float t = fminf(fabsf(dx), fabsf(dy)) / fmaxf(fabsf(dx), fabsf(dy));

    uint32_t r, b = __float_as_int(flag) << 2;
    uint32_t mask = __float_as_int(dx) ^ __float_as_int(dy) ^ (~b);
    __LOP3_0xe2(r, mask, t1, __float_as_int(t));

    const float p = fabsf(__int_as_float(r)) - 1.0f;
    return ((-0.0663f*(-p) + 0.311f) * (-p) + float(float(M_PI)/4.0)) * (*(float *)&r) + Offset;
}
```

LOP3 eliminates conditional branches