Real-Time Semi-Global Matching Disparity Estimation on the GPU

Christian Banz, Holger Blume, Peter Pirsch
Motivation

- Stereo Matching is a frequently required task
- Applications
  - Driver Assistance Systems
  - 3D image processing for real-time feedback
  - Keyhole sugery
  - Subtitle placement

- Implementation on off-the-shelf devices for low-cost availability in desktop computing systems: GPUs
Estimation

Identifying the projection point of a 3-D real-world point in two or more images taken from distinct viewpoints

Existing implementations:

GPU [Gibson '08 / Ernst '08]
3 frames/second

CPU/SIMD [Hirschmüller '08]
0.25 frames/second
New features on current GPU architectures

- Rapid development of GPU architecture
- Fermi architecture has many new features
  - Dual warp scheduler
  - Branch prediction
  - Concurrent kernel execution
  - Improved context switching
  - Significantly more bandwidth and computational power

- Investigate SGM implementations for new GPUs
Performance limiting factors

- **Software:**
  - Effective memory bandwidth usage (i.e. aligned access)
  - Instruction throughput

- **Hardware:**
  - Latency of the memory interface
  - Latency of the arithmetic pipeline
    - issue rate: 1 instruction/cycle, but latency: several ten cycles
  - *Both* has significant influence on performance

- Deep understanding of physical hardware necessary
  - Further reference: Slides by Volkov
Measures for high performance

- Ensure coalesced and aligned memory access
- Maximize data reuse (very little redundant memory access)
- High ratio of computational instructions to ancillary instructions

- Transfer coherent chunks of data to/from memory
- Keep the hardware pipeline filled
  - serial loop with independent arithmetic on local data

- Occupancy is not a good metric
  - Indicates the amount of resources occupied
  - NOT the effectiveness of usage
Semi–Global Matching

- Rank transform and median filter
  - 2D transformations
  - non-separable
  - requires access to local processing window

Semi-Global Matching [Hirschmüller '05]
Non-separable 2D image transform
Median Filter and Rank Transform

128 byte alignment boundary
= no overhead

One warp = tdx in parallel coalesced memory access

n_{ppt} pixel per thread serially more computation per thread
= keep pipeline filled

tdy: increase data reuse more instructions per ALU (cuda core)

local image window in shared memory

Extension of the separable convolution [Podlozhnyuk]
Parameter Study

Best configuration: $tdy = 4$ and $n_{ppt} = 4$

Performance: 0.64 ms

Speed-up > 2

$t�x = 32$ (i.e. warp width)

1280 x 960 pixel image
3x3 median filter
Results (2D transform)

3x3 median filter (greyscale)

"Standard" implementation using texture memory

proposed implementation with optimal configuration set

Speed-Up: 4.3

9x9 rank transform (greyscale)

"Standard" implementation using texture memory

proposed implementation with optimal configuration set

Speed-Up: 4.0
Matching Cost Calculation

for (d=0; d<128; d++) {
    C[x,d] = |left[x] - right[x-d]|;
}

C(x, y, d)
Matching Cost (MC) Implementation Options

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time [ms]</th>
<th>Bandwidth / Max. Bandw.</th>
<th>Performance bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) MC Unaligned</td>
<td>16.32</td>
<td>48.6 GB/s / 144.0 GB/s</td>
<td>Memory Access Scheme</td>
</tr>
</tbody>
</table>

**Unaligned:** Each thread processes one point of \( C(x,y,d) \)
- Only \(|\text{left}[x] - \text{right}[x-d]| \rightarrow \) not compute bound!

**Proposed:** Combination of parallel and serial with aligned memory access

**Datatype:** \( \text{float} \rightarrow \text{uchar} \)
-unpack/pack data overhead (computation only efficient in float/int32)
- quarter memory bandwidth required

Threads process pixels in parallel (aligned, coalesced memory access, parallelization)
Path Cost Calculation

1. Left Image
2. Right Image
3. Rank-Transformation
4. Rank-Transformation
5. Matching Cost
6. Path Cost
7. Disparity calculation
8. Median Filter
Main challenges imposed by Path Cost Calculation

- Main calculation is *recursive* and *not scan-aligned*
- Values off all 8 paths must be kept
  - for the *next pixel along the path*
  - and until *all paths to a pixel are known*

→ Parallelization concept which adheres to principles for fast implementation on the GPU

\[
L_r(p, d) = C(p, d) + \min \{ L_r(p - r, d), \]
\[
L_r(p - r, d + 1) + P_1, \]
\[
L_r(p - r, d - 1) + P_1, \]
\[
\min_i \{ L_r(p - r, i) \} + P_2 \} - \]
\[
\min_i L_r(p - r, i) \]
Path cost calculation

- Same parallelization scheme as for matching cost calculation
  - Major difference: Follow path direction serially

Example

Both Bs can be processed in parallel

Data dependencies:
C can only be processed after A and both Bs
Path Cost Calculation: Parameter Study

Best configuration: $t_y = 16$ and $td_y = 4$
Performance: 39.81 ms
Speed-Up: > 2
Path Cost Calculation: Results

- 1 path direction = 1 kernel
  - Not all streaming multiprocessors used
- Use concurrent kernel execution
  - Available with Fermi architectures
  - Compensates inherently serial computation of the PC kernels

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time [ms]</th>
<th>Bandwidth [GB/s]</th>
<th>Performance bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matching cost + Path cost all 8 path directions sequentially</td>
<td>75.68</td>
<td>20.9</td>
<td>Latency (pipeline is not always filled)</td>
</tr>
</tbody>
</table>
## Summary of Results

Entire disparity estimation using semi-global matching

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Architecture</th>
<th>Com. Cap.</th>
<th>Image Size</th>
<th>Disparity depth</th>
<th>Processing time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposed here</td>
<td>Tesla C2050</td>
<td>2.0</td>
<td>640x480</td>
<td>128</td>
<td>16</td>
</tr>
<tr>
<td>Proposed here</td>
<td>Tesla C2050</td>
<td>2.0</td>
<td>1024x768</td>
<td>128</td>
<td>36</td>
</tr>
</tbody>
</table>
Observations

- Very fast implementation is sometimes a *black art*
  - A single code line can ruin performance
- Very deep knowledge of hardware behavior required
  - ALU and pipeline behavior (stalls, scheduler, etc.)
  - Physical memory properties and memory cache

- Hardware-oriented programming style necessary
- Nvidia should openly communicate this to programmers and provide relevant information
Conclusion

- Efficient parallelization schemes for SGM on the GPU
- Key enablers
  - Combination of parallel and serial implementation (parametrizable)
  - Serial implementation must be local and independent
  - Efficient, aligned, block based memory access
- Can serve as paradigm for programming styles (hardware oriented programming)