DYNAMIC PARTITIONING-BASED JPEG DECOMPRESSION ON HETEROGENEOUS MULTICORE ARCHITECTURES

Wasuwee Sodsong¹, Jingun Hong¹, Seongwook Chung¹, Yeongkyu Lim², Shin-Dug Kim¹ and Bernd Burgstaller¹

¹Yonsei University
²LG Electronics
JPEG Decompression

Entropy Coded Data

```
110010001010011010
101001011010100100
110000101000010011
010010101001000111
010010101001000111
```
JPEG Decompression

Entropy Coded Data

Huffman Decoding

Frequency Domain
JPEG Decompression

Entropy Coded Data

Huffman Decoding

Frequency Domain

IDCT

Spatial Domain (YCbCr)
JPEG Decompression

Entropy Coded Data

Huffman Decoding

Frequency Domain

IDCT

Spatial Domain (YCbCr)

Upsampling

YCbCr Color
JPEG Decompression

Entropy Coded Data

Huffman Decoding

Frequency Domain

IDCT

Spatial Domain (YCbCr)

Upsampling

Color Conversion

RGB Color

YCbCr Color
JPEG Decompression

Entropy Coded Data

Huffman Decoding

Frequency Domain

IDCT

Spatial Domain (YCbCr)

Upsampling

Output

RGB Color

Color Conversion

YCbCr Color

Bitmap Image
Sequential JPEG Decompression

- JPEG is an **asymmetric compression**
  - Compression performs **once per image**
  - Decompression performs **once per use**
- 463 out of the 500 most popular websites use JPEG images
- Operates in blocks of 8x8 pixels
- Sequential JPEG decoders apply IDCT, upsampling and color conversion **block-by-block**
Parallelism in JPEG Decompression

Sequential Part
- Huffman decoding
- **NOT** suitable for data-parallelism
  - Codewords have variable lengths.
  - The starting bit of a codeword in the encoded bitstream is only known once the previous codeword has been decoded.
Parallelism in JPEG Decompression

Sequential Part
- Huffman decoding
- NOT suitable for data-parallelism
  - Codewords have variable lengths
  - The starting bit of a codeword in the encoded bitstream is only known once the previous codeword has been decoded

Parallelizable Part
- IDCT, upsampling and color conversion
- Suitable for GPU computing and SIMD operations on CPU
  - low data dependency
  - operates same instructions repeatedly
  - has fixed input and output sizes
Research Question

How to orchestrate JPEG decompression on CPU+GPU architectures?

- Input image characterized by
  - Width
  - Height
  - Entropy

- Need: work partitioning, schedule, execution infrastructure
Our Contributions

- Heterogeneous JPEG decoder on CPU+GPU architectures
  - profiling based performance model
  - dynamic partitioning scheme that automatically distributes the workload at run-time
- Pipelined execution model overlaps sequential Huffman decoding with GPU computations
- Parallelizable part is distributed across CPU and GPU
  - data-, task- and pipeline-parallelism
- GPU kernels designed to minimize memory access overhead
- libjpeg-turbo implementation and experimental evaluation for libjpeg-turbo library
libjpeg & libjpeg-turbo

- libjpeg is a sequential JPEG compression reference implementation by Independent JPEG group
  - First version released in 1991

- libjpeg-turbo is a re-implementation of libjpeg
  - Utilizes SIMD instructions on x86 and ARM platforms.
  - Used by Google Chrome, Firefox, Webkit, Ubuntu, Fedora and openSUSE

- Both libraries strictly designed to conserve memory
  - Inhibits coarse-grained parallelism
  - A non-goal with today's target architectures
Re-engineering libjpeg-turbo

**libjpeg-turbo**

- To conserve memory, libjpeg-turbo decodes images in units of 8 pixel rows:
- 8 rows at a time do not contain enough computations to keep the data-parallel execution units of a GPU busy.
- Significant constant overhead per kernel invocation and data transfer (host→device→host).

**Our Approach**

- Store an entire image in memory:
- Fully utilizes all GPU cores by processing several larger image chunks.
- Reduce number of kernel invocations and data transferring overhead.
Heterogeneous JPEG Decompression Overview

**Motivation:** One architecture is unutilized when the other is processing

**Observation:** No dependency among 8x8 pixel blocks. Thus, the CPU and the GPU can compute in parallel

**Goal:** Find partitioning size at runtime such that the load on the CPU and the GPU are balanced

**Requirement:** Performance model through offline profiling
Performance Model

- Offline profiling step on image training set
  - 19 master images cropped to various sizes
  - Maximum image size is 25 megapixels

- Profile execution time of the sequential part and the parallelizable part on CPU and GPU

- Model all decompression steps using multivariate polynomial regression up to degree 7

- Select the best-fit model by comparing Akaike information criterion (AIC) values
Performance Model for the Parallelizable Part

- Linearly scales as image size increased
- Image dimension is known at the beginning of the decompression step
- **Parameters**: width and height

![Graph showing time vs. pixels for different subsampling ratios (4:2:2 and 4:4:4)]
Unlike the parallelizable part, Huffman decoding time does **NOT** have a high correlation with image width and height.
Huffman decoding time has a high correlation with the size of entropy coded data.

We have observed a linear trend as entropy density increased, entropy size in bytes per pixel.

**Parameters:** width, height and entropy size

Entropy size can be roughly approximated from JPEG file size.
Overlapped Partitioning Scheme

- Sharing workload of the parallelizable part between CPU and GPU
Overlapped Partitioning Scheme

- **Idea:** Share workload of the parallelizable part on the CPU and the GPU.

- Partitioning equation can be formulated as
  \[ f(x) = T_{disp}(w, h - x) + P_{CPU}(w, x) - P_{GPU}(w, h - x), \]
  where \( x \) is number of rows given to CPU, and
  
  \( w, h \) are image width and height.

- When \( f(x) = 0 \), the time spent on the CPU and GPU are equaled.

- \( w \) and \( h \) are known at runtime. We can use Newton’s method to solve for \( x \).

- **Problem:** GPU is unutilized during Huffman decoding.
Pipelined Partitioning Scheme

- Sharing workload of the parallelizable part between CPU and GPU
- Increase parallelism by performing Huffman decoding and GPU kernel in pipelined fashion
Pipelined Partitioning Scheme

- **Idea:** Execute Huffman decoding in a pipelined fashion with GPU kernel.
- **Split an image into several chunks of \( c \) rows.
- **An optimal chunk size is found through a profiling.**
- **We can start kernel invocation as soon as an image chunk is decoded.**
- **On a fast GPU, only the execution time of last chunk is visible to users.**

- **Problem:** Does **NOT guarantee** improvement over CPU computation.
Combined Partitioning Scheme

- **GPU-Only**
  - CPU: Huffman Decoding
  - GPU: GPU Kernel
  - CPU and GPU are connected by dashed arrows indicating data transfer.

- **Overlapped**
  - CPU: Dispatch, Huffman
  - GPU: GPU Kernel, SIMD
  - CPU and GPU are connected by solid arrows indicating execution.

- **Pipeline**
  - CPU: Dispatch, Huffman 1, Huffman 2, Huffman 3
  - GPU: Kernel (Huffman 1), Kernel (Huffman 2), Kernel (Huffman 3)
  - CPU and GPU are connected by solid arrows indicating execution.

- **Combined**
  - CPU: Dispatch, Huffman 1, Huffman 2, Huffman 3
  - GPU: Kernel (Huffman 1), Kernel (Huffman 2), Kernel (Huffman 3), Kernel (Huffman 4)
  - CPU and GPU are connected by solid arrows indicating execution.
Combined Partitioning Scheme

- Combining overlapped and pipelined model to guarantee improvement.
  
  \[ f(x) = T_{Huff}(w, h - c, d) + T_{disp}(w, h - x) + P_{CPU}(w, x) - P_{GPU}(w, h - x), \]

- Where \( c \) is number of rows in a chunk, and \( d \) is entropy density in bytes per pixel.

- Using Newton’s method to solve for \( x \) at runtime.

- Estimation errors from Huffman decoding:
  - Assume the same Huffman decoding time for every pixel across an image
  - Entropy is not distributed evenly in practice.

- Perform re-partitioning before Huffman decoding for the last GPU kernel.
We implemented GPU kernels for IDCT, upsampling and color conversion.

- **Optimizations**
  - Vectorization to reduce number of reads/writes to global memory
  - Store intermediate results in local memory (NVIDIA’s shared memory)
  - Map work-items with consideration of coalesced memory access
  - Combine color conversion kernel with the prior kernel to avoid global memory store between kernels.
Experimental Setup

- Test set
  - A new set of images. No images are reused from the training set
  - 3591 images of various size
  - Maximum image size is 25 megapixels

- Hardware specification

<table>
<thead>
<tr>
<th>Machine Name</th>
<th>GT 430</th>
<th>GTX 560</th>
<th>GTX 680</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU model</td>
<td>Intel i7-2600k</td>
<td>Intel i7-2600k</td>
<td>Intel i7-3770k</td>
</tr>
<tr>
<td>CPU frequency</td>
<td>3.4 GHz</td>
<td>3.4 GHz</td>
<td>3.5 GHz</td>
</tr>
<tr>
<td>GPU model</td>
<td>NVIDIA GT 430</td>
<td>NVIDIA GTX 560Ti</td>
<td>NVIDIA GTX 680</td>
</tr>
<tr>
<td>GPU core frequency</td>
<td>700 MHz</td>
<td>822MHz</td>
<td>1006MHz</td>
</tr>
<tr>
<td>No. of GPU cores</td>
<td>96</td>
<td>384</td>
<td>1536</td>
</tr>
<tr>
<td>GPU memory size</td>
<td>1024 MB</td>
<td>1024 MB</td>
<td>2048 MB</td>
</tr>
<tr>
<td>Compute Capability</td>
<td>2.1</td>
<td>2.1</td>
<td>3.0</td>
</tr>
</tbody>
</table>
Speedup Over SIMD Version

![Graph showing speedup over SIMD version for different GPU models and pixel counts. The graph compares GPU-only, overlapped, pipelined, and combined approaches.](image-url)
Amdahl's Law, Theoretical Maximum Speedup

- Up to 95% of theoretical maximum speedup
- Average of 88% when the problem size is large enough

Intel - i7 3770k + Nvidia GTX 680
Partitioning Errors

Average Execution Time (ms)

CPU Time vs GPU Time for GT 430 and GTX 680

Pixels

0 10M 20M

0 10 20 30 40

0 25 50 75
Conclusions

- JPEG decoding contains high amount of massive data computation
- We proposed JPEG decoding scheme for heterogeneous architectures
  - Performance model using polynomial regression
  - Dynamic partitioning scheme
  - Up to 4.2x (2.5x average) speedup over the SIMD version of libjpeg-turbo
  - Guaranteed improvement regardless of CPU+GPU combinations
  - Workload is well distributed across CPU and GPU
- Our combined partitioning scheme achieves up to 95% of the theoretically attainable speedup, with an average of 88%

- Future work
  - Extension to mobile systems
Thank you
Backup Slides
Related Work

Parallel JPEG Decoding

[1] Parallel image processing based on CUDA [Yang ‘08]

[2] Design, implementation and evaluation of a task-parallel JPEG decoder for the libjpeg-turbo library [Hong ’12]


Heterogeneous Computing

[1] Architectural exploration of heterogeneous multiprocessor systems for JPEG [Shee ‘08]


[3] Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping [Luk ‘09]


GPU Results

- An 2048x2048 image

![Normalized Time Graphs for GT 430, GTX 560, GTX 680](image_url)

- **GT 430**
  - CPU
  - SIMD
  - GPU

- **GTX 560**
  - CPU
  - SIMD
  - GPU

- **GTX 680**
  - CPU
  - SIMD
  - GPU

- Color Coding:
  - Huffman
  - Upsampling
  - Host to Device
  - IDCT
  - Color Conversion
  - Upsampling & Color Conversion
  - Device to Host