JPEG decoder example design
Motivation

- Increasing storage requirements for images
- Customer application:
  - Provide scaled versions of images based on requests
  - Tradeoff between *compute* and *storage*
Compiling OpenCL to FPGAs

Kernel Programs

```c
__kernel void sum(__global const float *a, __global const float *b, __global float *answer)
{
int xid = get_global_id(0);
answer[xid] = a[xid] + b[xid];
}
```

Host Program

```c
main()
{
read_data_from_file( ... );
manipulate_data( ... );
clEnqueueWriteBuffer( ... );
clEnqueueKernel(..., sum, ...); clEnqueueReadBuffer( ... );
display_result_to_user( ... );
}
```
```c
__kernel void
sum(__global const float *a,
__global const float *b,
__global float *answer)
{
  int xid = get_global_id(0);
  answer[xid] = a[xid] + b[xid];
}
```
__kernel void
sum(const float *a,
const float *b,
float *answer)
{
int xid = get_global_id(0);
answer[xid] = a[xid] + b[xid];
}

main()
{
read_data_from_file(...);
manipulate_data(...);
clEnqueueWriteBuffer(...);
clEnqueueKernel(..., sum, ...);
clEnqueueReadBuffer(...);
display_result_to_user(...);
}
__kernel void
sum(__global const float *a,
__global const float *b,
__global float *answer)
{
    int xid = get_global_id(0);
    answer[xid] = a[xid] + b[xid];
}

main()
{
    read_data_from_file( ... );
    manipulate_data( ... );

    clEnqueueWriteBuffer( ... );
    clEnqueueKernel(..., sum, ...);
    clEnqueueReadBuffer( ... );
    display_result_to_user( ... );
}
```c
__kernel void
sum(__global const float *a, __global const float *b, __global float *answer)
{
    int xid = get_global_id(0);
    answer[xid] = a[xid] + b[xid];
}
```

```c
main()
{
    read_data_from_file(...);
    manipulate_data(...);
    clEnqueueWriteBuffer(...);
    clEnqueueKernel(..., sum, ...);
    clEnqueueReadBuffer(...);
    display_result_to_user(...);
}
```
Concurrent processing and transfers

```c
__kernel void sum(__global const float *a, __global const float *b, __global float *answer)
{
    int xid = get_global_id(0);
    answer[xid] = a[xid] + b[xid];
}
```

```c
main()
{
    read_data_from_file( ... );
    manipulate_data( ... );
    clEnqueueWriteBuffer( ... );
    clEnqueueKernel(..., sum, ...);
    clEnqueueReadBuffer( ... );
    display_result_to_user( ... );
}
```

Host executable

Altera OpenCL library

Operating system (Linux / Windows)
Concurrent processing and transfers (cont’d)

- Host schedules transfers and kernel execution on different data sets
  - Can hide transfer latency

- JPEG decoder leverages concurrent transfers and computation
  - Performance determined by slowest kernel execution or transfer
**JPEG decoder**

- **Data flow: decompresses images as a first step**
  - i.e. front-end for other image processing blocks running on the FPGA
  - FPGA architecture allows several accelerators to run *concurrently*
  - Accelerator performance decoupled of subsequent processing

- **Throughput is proportional with FPGA resources used**
  - Implemented a decoder that uses one third of a Stratix V A7 FPGA resources
  - Leaves a large number of resources available for subsequent image processing
Results

- **Baseline JPEG decompression**
  - 4:4:4 and 4:2:0 downsampling

- **Arbitrary image resolution**
  - Decoding speed up to 900 Mpixels / s (2.7GB / s)

**Measured performance**

<table>
<thead>
<tr>
<th>Image</th>
<th>Image size</th>
<th>FPGA Throughput (images / s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.jpeg</td>
<td>2048 x 1536</td>
<td>287</td>
</tr>
<tr>
<td>2.jpeg</td>
<td>308 x 231</td>
<td>6265</td>
</tr>
<tr>
<td>3.jpeg</td>
<td>1024 x 768</td>
<td>1044</td>
</tr>
<tr>
<td>4.jpeg</td>
<td>768 x 1024</td>
<td>1059</td>
</tr>
</tbody>
</table>
Other performance metrics

- **Significantly lower power consumption**
  - FPGA power envelope is ~25W

- **Overlaps image transfer and computation**
  - Host simply reads images from disk, inspects header and transfers the images to the FPGA
  - JPEG decoder is always active, decoding previously transferred images

- **Scalable implementation**
  - Performance is proportional to the number of Huffman decoders
  - 6-way decoder in this implementation

- 36% logic
  - 75.4 k ALMs
  - 166 DSPs

- 13% logic

6-way JPEG decoder

**Stratix V A7 FPGA**
How to decode a JPEG image?

**Encoding steps:**
- Image separated into Y, C_b and C_r components
- 8 x 8 blocks on each component are transformed using DCT transform
- Image data is entropy encoded
  - Run-length coding
  - Huffman coding

**Split JPEG decoding into several kernels**
- Kernel for Huffman and run-length decoding
- Kernel for inverse quantization, iDCT and RGB conversion
- All these kernels run concurrently
Huffman / Run-length decoder details

- **Serial process, codes are decoded one at a time**
  - All codes in a 8x8 block are adjacent in the encoded stream
  - Once all codes in block are decoded, they are forwarded to iDCT

- **Single threaded OpenCL**
  - Essentially C code
  - Decodes 1 coefficient / loop iteration
  - Translates into one or more DCT coefficients / cycle
    - Takes into account run-length decoding that skips coefficients equal to 0

- **Huffman codes are different for each JPEG image**
  - Loaded at accelerator initialization

- **Multiple decoders can process independent images**
  - Current implementation uses 6 parallel Huffman decoders
  - Decoders are truly independent, not alike GPUs that impose SIMD restrictions
Inverse quantization / iDCT / RGB conversion

- **Processes 16 coefficients / cycle**
  - Receives data from multiple Huffman decoders

- **Expressed as an OpenCL multi-threaded kernel**
  - Multi-threaded C
  - Synchronization points

- **Stores 6 blocks of 8 x 8 points**
  - Supports both 4:4:4 and 4:2:0 formats
  - Generates a 8 x 16 or 16 x 16 tile of the output
JPEG decoder – Application architecture

Functional flow

1. Reading file headers
2. Huffman decoding
3. Run-length decoding
4. Inverse quantization
5. Inverse DCT
6. YCbCr to RGB

Hardware architecture

- Reading file headers to PCIe
- Fetch from DDR3
- Huffman decoding
- Multiplexing streams
- iDCT + RGB
- DDR3
- FPGA
- Reads from memory
- Initial data parsing

Round-robin multiplexing ensures that iDCT receive full blocks

© 2014 Altera Corporation—Public
Channels

- **Altera’s extension to OpenCL**
  - Mechanism for direct kernel to kernel communication
  - Preserves DDR3 bandwidth for other components of the application
  - Code example from the multiplexer kernel

```c
kernel void Multiplex() {
    while (true) {
        #pragma unroll
        for (int i = 0; i < COPIES; i++) {
            if (i == source) hc = read_channel_altera(results[i]);
        }
        ...
        write_channel_altera(toDCT, dc);
        ...
    }
}

kernel void DCTandRGB(global uchar *output, uchar write) {
    struct DCTContext dc = read_channel_altera(toDCT);
    ...
}
```

Channels from multiple decoders

Channel to iDCT
End to end performance measurements
- Include host to device and device to host transfers over PCIe
- Divide total time by the number of images
- Batch processing mode sends a group of images for processing
Performance
- Up to 2.7 GB/s of decoded data
- 1/3 of the resources on the A7 FPGA device

Entire decoder implemented in OpenCL

OpenCL implementation allows fast changes to the algorithm, i.e.
- Enhance \textit{algorithm} based on image set properties
- Increase throughput by instantiating more decoders
- Include decoder in a larger design
Thank You