

OpenCL in Handheld Devices

NOKIA

Kari Pulli

Research Fellow, Nokia Research Center Palo Alto

Material from Jyrki Leskelä, Jarmo Nikula, Mika Salmela

OpenCL 1.0 Embedded Profile

- Enables OpenCL on mobile and embedded silicon
 - Relaxes some data type and precision requirements
 - Avoids the need for a separate “ES” specification
- Khronos APIs provide computing support for imaging & graphics
 - Enabling advanced applications in, e.g., Augmented Reality
- OpenCL will enable parallel computing in new markets
 - Mobile phones, cars, avionics



A camera phone with GPS processes images to recognize buildings and landmarks and provides relevant data from internet

Embedded Profile main differences

- Adapting code for embedded profile
 - Added macro `__EMBEDDED_PROFILE__`
 - `CL_PLATFORM_PROFILE` capability returns the string `EMBEDDED_PROFILE` if only the embedded profile is supported
- Online compiler is optional
- No 64-bit integers
- Reduced requirements for constant buffers, object allocation, constant argument count and local memory
- Image & floating point support matches OpenGL ES 2.0 texturing

The extensions of full profile can be applied to embedded profile

Floating point numbers

- INF and NAN values for floats are not mandated
- Most accuracy requirements are the same, but some single precision floating-point operations are relaxed from full profile:
 - **x / y** ≤ 3 ulp [units in last place]
 - **exp** ≤ 4 ulp
 - **log** ≤ 4 ulp
- Float add, sub, mul, mad can be rounded to zero resulting an error ≤ 1 ulp (helps to keep HW area down)
- Denormalized half floats can be flushed to zero
- The precision of conversions from normalized integers is ≤ 2 ulp for the embedded profile (instead of ≤ 1.5 ulp)

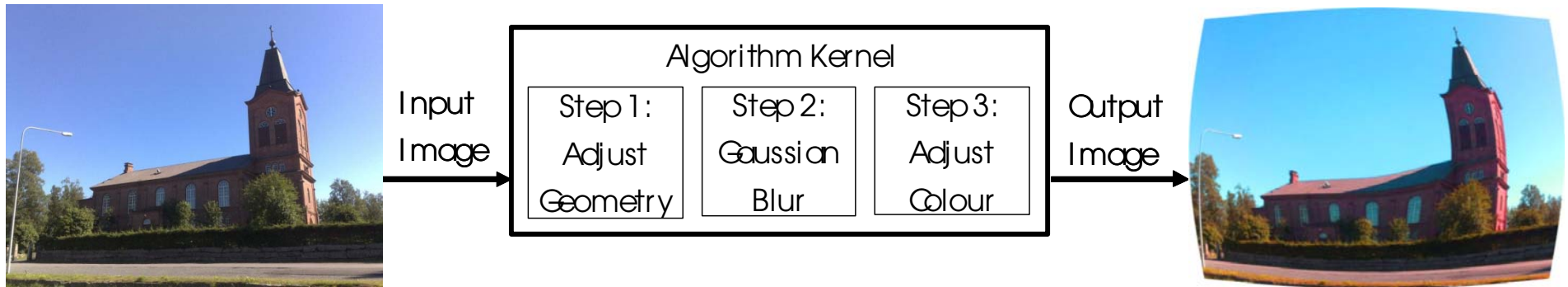
Image support in Embedded Profile

- Image support is an optional feature within an OpenCL device
- If Images are supported, the min reqs for the supported image capabilities are lowered to the level of OpenGL ES 2.0 textures
 - Kernel must be able to read ≥ 8 simultaneous image objects
 - Kernel must be able to write ≥ 1 simultaneous image objects
 - Width and height of 2D image ≥ 2048
 - Number of samplers ≥ 8
- Image formats match OpenGL ES 2.0 texture formats
- Support for 3D images is optional
- Float 2D/3D images only support point sampling (nearest neighbor)

Benefits for mobile devices

- Easier programming in a heterogeneous processor environment
 - Instead of learning different programming methods for CPU, GPU, DSP
 - OpenCL framework handles also event queuing
- Code developed once will run with future hardware
 - If the application conforms to the specification, it will run
 - Can be used with paravirtualized or full virtualized operating systems
 - no manufacturer or HW (even CPU) dependencies
- Area and energy constrained embedded devices
 - In embedded devices it is less likely that other processors (GPU) are 100x faster than main CPU (area consumption, energy constraints)
 - they all are more closer to "sweet spot"
 - OpenCL can allocate workload on multiple processors
 - for example in OMAP 4430 there are 2 x ArmCortexA9, ISP, GPU, DSP

OpenCL for image processing



- A three-step image processing algorithm in a single OpenCL kernel
 - Smaller execution overheads
 - Lower memory bandwidth requirements
 - Better performance comparisons as computation time dominates
- Three tests on TI OMAP 3430 (550 MHz ARM Cortex-A8 CPU & 110 MHz SGX530 GPU)
 - CPU alone
 - GPU alone
 - CPU+GPU in parallel
- Both time and power consumption were measured

Kernel program: Function definition

```
__kernel void imageProcessingTest(  
    float bend,  
    float zoom,  
    float4 adjust_r, // Conversion matrix column 1  
    float4 adjust_g, // Conversion matrix column 2  
    float4 adjust_b, // Conversion matrix column 3  
    float4 gauss_coeff,  
    __read_only image2d_t srcimage,  
    __write_only image2d_t destimage,  
    int width,  
    int height)  
{  
    // Local variables  
    // Step 1  
    // Step 2  
    // Step 3  
}
```

Kernel program: Local variables

```
// Local variables
const sampler_t sampler = CLK_NORMALIZED_COORDS_TRUE |
                           CLK_ADDRESS_CLAMP_TO_EDGE |
                           CLK_FILTER_NEAREST;

float2 size, xy, mycoord;
int2 destPos;
destPos.x = get_global_id( 0 ); // x index
destPos.y = get_global_id( 1 ); // y index
```

Kernel program: Adjust geometry

```
// Step 1
// Coordinate adjustment
mycoord.x = (float) (destPos.x) / (float) (width);
mycoord.y = (float) (destPos.y) / (float) (height);
float2 center(0.5f, 0.5f);
xy = (mycoord - center) / center;
xy = (xy * pow(dot(xy, xy), bend) * zoom + 1.0f) * 0.5f;
```


Kernel program: Read & blur pixels

```
// Step 2
// Read a 3x3 gaussian blurred image
float2 south, east;
south.x = 0.0f;          south.y = zoom / (float)(height);
east.x = zoom / (float)(width);    east.y = 0.0f;
float4 srcColor =
    read_imagef(srcimage, sampler, xy) * gauss_coeff.x;
srcColor += gauss_coeff.y * (
    read_imagef(srcimage, sampler, xy+south) +
    read_imagef(srcimage, sampler, xy-south) +
    read_imagef(srcimage, sampler, xy+east) +
    read_imagef(srcimage, sampler, xy-east) );
srcColor += gauss_coeff.z * (
    read_imagef(srcimage, sampler, xy+south+east) +
    read_imagef(srcimage, sampler, xy-south-east) +
    read_imagef(srcimage, sampler, xy+south-east) +
    read_imagef(srcimage, sampler, xy-south+east) );
```

Kernel program: Adjust colors, write

```
// Step 3
// Adjust contrast and saturation
float4 adjColor;
adjColor.x = dot (adjust_r, srcColor);
adjColor.y = dot (adjust_g, srcColor);
adjColor.z = dot (adjust_b, srcColor);
adjColor.w = srcColor.w;
// Clamp and write image
write_imagef(destimage, destPos, clamp(adjColor, 0.0f, 1.0f));
```

CPU versus GPU considerations

- GPU kernel uses **image2d_t** type to access images
 - maps directly to GPU's HW-based texture access
 - our OpenCL GPU back-end doesn't support memory buffers
- CPU works well with **memory buffers**, with native memory pointers
 - emulating image2d_t behavior in SW is very expensive!
 - the execution time dropped 77% compared to emulated image2d_t
 - also replaced very slow standard pow() function with a fast approximation
- The results are measured using these two separate kernels:
 - GPU: **image2d_t**
 - CPU: **memory buffers**  **More fair comparison**

Performance: Processors alone

- GPU is 3.5X faster than CPU
- GPU's 2.4 s = 0.3 s of input data set-up/transfer
0.4 s of result data read-back
only 1.7 second is consumed on actual execution

Why does GPU win?

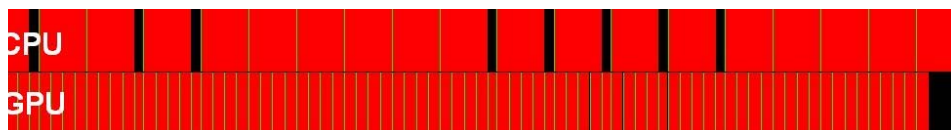
- GPU's single-program-multiple-data architecture fits OpenCL
- GPU has fast vectored floats
 - compared to slow VFPLite of ARM Cortex-A8
- ARM's NEON is not fully utilized
- Algorithm could be implemented also with integers
 - CPU could even outperform GPU

100 Frames 3MPix, 32b RGBA	Alone		Combined	
	CPU	GPU	CPU	GPU
<i>Frames/device</i>	100	100	19	81
<i>Execute time (s)</i>	864.9	240.4	237.4	247.5
<i>Average s/frame</i>	8.6	2.4	12.5	3.1
			2.5	
<i>Min s/frame</i>	6.98	2.33	10.87	2.39
<i>Max s/frame</i>	10.94	2.49	12.62	3.16



Performance: CPU+GPU

- In combined run, frames were scheduled to either processor based on which one becomes free next
- **Running CPU and GPU in parallel gives worse performance than with GPU alone, average frame time 2.5 seconds**
- Graph shows kernel execution of CPU and GPU in red, black bars are idle time caused by scheduling



- Scheduling made CPU to idle more than necessary
- But *why does GPU now need more time for 81 frames than for 100 frames when running alone?*

100 Frames 3MPix, 32b RGBA	Alone		Combined	
	CPU	GPU	CPU	GPU
<i>Frames/device</i>	100	100	19	81
<i>Execute time (s)</i>	864.9	240.4	237.4	247.5
<i>Average s/frame</i>	8.6	2.4	12.5	3.1
			2.5	
<i>Min s/frame</i>	6.98	2.33	10.87	2.39
<i>Max s/frame</i>	10.94	2.49	12.62	3.16

CPU+GPU combined run analyzed

- Transfer times doubled, total data transfer overhead 1.4 s
 - input data set-up / transfer from 0.3 s to 0.6 s
 - result read-back from 0.4 s to 0.8 s
- GPU execution time remained the same as alone, $3.1 \text{ s} - 1.4 \text{ s} = 1.7 \text{ s}$
- Parallel execution doesn't make algorithm to become bandwidth limited
 - **why does the parallel version perform worse?**
- Data transfer overhead is doubled
 - since it is purely CPU bound
 - that *makes GPU idle more between frames*
- Frame computation by CPU cannot compensate that loss

Scheduling is not easy

Better scheduling would run data transfers at full speed and use CPU for frame computation only when it is free from serving GPU data transfers

- 84 frames with GPU and 16 with CPU => the optimum **202 s**:
 - For GPU, consumed time becomes $84 \times 2.4 \text{ s} = 202 \text{ s}$, including also $84 \times 0.7 \text{ s} = 59 \text{ s}$ CPU time due to data transfers
 - For CPU, data transfers leaves $202 \text{ s} - 59 \text{ s} = 143 \text{ s}$ for frame processing, enough for $143 \text{ s} / 8.6 \text{ s} = 16$ complete frames
- Even with current scheduling CPU+GPU combination can give speed-up for an algorithm with more computation per frame
- Optimal scheduling depends on relative speeds of the processors and their dependencies (such as data transfers)
 - E.g., if CPU happens to be faster than GPU, then it may be best to avoid wasting CPU time on data transfers and ignore GPU, running on CPU only

Energy measurements

- 3.8 V input voltage
 - 516 mA idle current was subtracted from the results
- GPU consumes 0.56 J per frame, only 14 % of the CPU's 3.93 J
 - for pure execution without data transfers GPU used only 0.26 J (7%!)
 - over half of the energy in GPU case is due to data transfers run by CPU!
- Combined run is worse than GPU alone

Efficacy	CPU alone	GPU alone	CPU+ GPU
<i>Frames (n)</i>	4	20	20
<i>Time (s)</i>	43.6	48.6	52.1
<i>Current (mA)</i>	95	61	132
<i>Power (mW)</i>	361	232	502
<i>Energy/frame (J)</i>	3.93	0.56	1.31

Analysis of energy measurements

- GPU architecture is better for OpenCL-like parallel data computing
 - all execution units are busy, ALU utilization is maximized
- CPU suffers from poor ALU utilization
 - we didn't have long pipelined vector computations for ARM NEON
 - compiler and our OpenCL do not utilize NEON instructions very well
- GPU's lower operating frequency (110 MHz vs. 550 MHz) saves energy
 - power dissipation increases non-linearly with frequency
 - generally,
 - high parallelism with low frequency
 - is more energy-efficient than
 - low parallelism with high frequency

Summary

- OpenCL 1.0 Embedded Profile is a subset of the full profile
 - Not an "ES" specification of its own
- Easier programming of heterogeneous multi-processor
 - Fast multiprocessor code without portability hassle
- Speedups and energy efficiency possible via parallelism
 - But scheduling can be tricky