

Compute for Mobile Devices: Performance-focused Hands-On Tutorial

Maxim Shevtsov
Intel
maxim.y.shevtsov@intel.com



SA2014.SIGGRAPH.ORG

SPONSORED BY



Hi everyone, my name is Maxim.

At Intel I'm driving the team responsible for tracking OpenCL perf/benchmarks, creating collaterals and doing technology evangelization.

But today I'm speaking solely on MY own Behalf

Today Android and Windows phones and tablets, as well as iOS devices and many other things in your hands are actually a full-blown, general-purpose computers.

Even though these may be dirt-cheap phones, for many people in emerging markets these would be their first computer.

Agenda

- ▶ Intro to (mobile) Compute
- ▶ RenderScript*
- ▶ OpenCL™
- ▶ Metal*
- ▶ OpenGL ES™ Pixel Shaders
- ▶ OpenGL ES™ Compute Shaders
- ▶ Conclusions
- ▶ Common perf tricks

Bonus: Optimizing OpenCL code in OpenCV <if time permits>

SA2014.SIGGRAPH.ORG

SPONSORED BY



So in this talk we will characterize the opportunities and limitations of mobile thru the perspective of the industry-adopted compute APIs.

We have really packed agenda, so I would ask to delay any questions to the very end

Why Compute In Mobile

In *games*, because the graphics pipeline is still so specialized to GPU

- ▶ Physics and AI are often computed elsewhere

[GPU Compute in Games Wed, 03 December 14:15 - 18:00, Peony Hall](#)

Computationally-intensive usage *beyond graphics*

- ▶ Media: image/video processing and codecs, computational photography
- ▶ Frameworks effects
- ▶ Computer Vision (e.g. gesture recognition, eye tracking, AR)

Lot's of specific hardware (ISP/DSP) to expose

SA2014.SIGGRAPH.ORG

SPONSORED BY



Why do we need compute in mobile?

The simplest basic answer is - we have compute usages that do not fit *graphics pipeline*

We have things like LiquidFun solver in Android RS, or Bullet engine in OpenCL

Also there are various media things plus fancy UI effects like animation, shadows or ripples

Finally, computationally intensive things like Speech Recognition and all sorts of CV

And it is not only about usages- also there is a good bulk of hardware to expose

Quick environmental scan, 1

Consider old-school HPC Compute:

- Abundant parallelism, huge data and compute demand
- Very specific platforms/tuning
- ...FORTRAN*, MPI*, OpenMP*...recently CUDA*, OpenCL™...

In contrast, Mobile is huge variety of devices and capabilities:

- Everything should run on every device
- No low-level performance tuning
- Finally, different APIs! (partly driven by two points above)

Forget everything you know about performance from HPC

SA2014.SIGGRAPH.ORG

SPONSORED BY



Again, the focus of the talk is compute on the end-user devices, not things like thin client or cloud computing.

From this point of view, HPC always less cares about device compatibility- mostly sticking to the specific vendor and/or API.

I myself came from a regular “client” segment rather than from HPC, but still recognized mobile realities are very different:

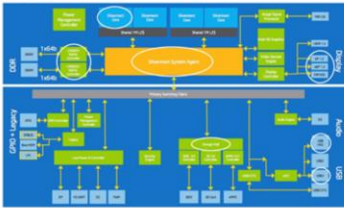
- Different vendors’ architectures are substantially more diverged in mobile (considering things ILP vs TLP, different SIMD options, tile-based architectures, etc) which limits the optimization path.

Quick environmental scan, 2

What about conventional desktops/laptops/etc?

- Unified Memory makes the life easier similarly to mobile
- Compute results are usually visualized in some way

Bay Trail SOC Block Diagram



Also, mobile device is more than CPU and GPU

- Camera (often with dedicated compute-capable ISP)
- Video/Audio/etc, maybe with multiple DSPs
- Overall SoC power tradeoffs

Mobile requires potentially different capabilities/APIs for Compute

SA2014.SIGGRAPH.ORG

SPONSORED BY



As I said before your Android or iOS device is in fact *very similar* to your home or office computer: it is effectively tiny, low-power laptop, with touch screen replacing the keyboard

So most laptops and phone/tablets enjoy the CPU-GPU Unified Memory (and things like shared caches) which make the life easier, comparing to NUMA and other oddities that are regular to HPC.

Also just like for say laptops, the results of mobile Compute are often related to graphics by means of being displayed in some way

Finally Also keep in mind that your mobile processor is more than just CPU and GPU

- ISP in camera and maybe multiple DSPs- all the things your compute API of choice should efficiently do interoperability with

And of course- different tradeoffs for power

One important implication of these differences is that Mobile does require different API for Compute to accommodate these challenges

But first, let's see what a typical GPU is capable for

Few words on the mobile capabilities

Keep in mind the following mobile constraints

- ▶ Typical mem Bandwidth is low (2-4 Gbytes/sec)
- ▶ GPU Gflops (ballpark of ~100 Gflops)
- ▶ Screen resolutions and camera frame rates are high

GPU is really busy with rendering:

With mobile devices the opportunities to use CPU + GPU are even greater

SA2014.SIGGRAPH.ORG

SPONSORED BY



General interactivity of the Mobile implies real-time frame rates

Current GPU ballpark is of ~100 Gflops, but ~300 gflops are NOT unheard

Bandwidth figures remain way behind and will stay that way for the predictable future
(SIGGRAPH-2013-SamMartinEtAl-Challenges.pdf)

This is especially important as only the rendering leftovers are available for compute and any significant compute horsepower demand can be really prohibitive for real-time
One immediate implication Implication: in the mobile the CPU is nowhere “just a fallback” device.

Is There Any Simplest Mobile Compute API Ever?

- ▶ No workgroups/threads/warps, no shared memory
 - ▶ No async queues ...finally no notion of a “device”
- ▶ Minimalistic “allocate/execute/copy” API
 - ▶ Built-in lib functions for typical usage scenarios
 - ▶ Zero-copy sharing with other APIs
- ▶ Leverages GPU but runs everywhere
 - ▶ with transparent CPU fallback
- ▶ Simple kernel lang
- ▶

SA2014.SIGGRAPH.ORG

SPONSORED BY



If you do a lot of processing on CPU today and just seek for a speedup ...you probably not very interested

We will keep discussing APIs along the dimensions from this foil and provide summary in the very end

Google RenderScript*

Android, iOS, Windows, Linux, OS X

SA2014.SIGGRAPH.ORG

SPONSORED BY



RenderScript seems to be designed exactly around the wishes from the previous section.

RenderScript*

- ▶ User-scripts language - based on C99 with extensions
 - ▶ Rich set of built-in functions
 - ▶ Many GPU-specific restrictions are relaxed
- ▶ Intrinsic - the most popular feature today
 - ▶ Built-in high-level library functions for image processing (Blur, YuvToRGB, etc)
 - ▶ Objective: drivers ship optimized intrinsic implementation
- ▶ Both Java API and NDK bindings
 - ▶ Each user script also gets a Java “glue” class with accessors
- ▶ Support library (for devices running Android 2.2 and higher)

SA2014.SIGGRAPH.ORG

SPONSORED BY



It operates notion of user scripts that are actually written using C-like language, with many CPU-attributed features like support for recursion
Most host APIs (e.g. execute and data copying but of course not alloc) have script-level equivalents

Also there are Intrinsic that are built-in functions that perform well-defined operations for image processing
Some intrinsic may run on special-purpose processors

What I really love in Rs is the reflected layer APIs that are a set of classes that are generated from your Renderscript code by build tools. This auto-generated layer is basically a wrapper around the Renderscript code (including script global vars).

RenderScript* example

```
//test.rs
#pragma version(1)
#pragma rs java_package_name(com.android.example.siggraph)
uchar4 __attribute__((kernel)) root(uchar4 ain)
{
    float4 f4 = rsUnpackColor8888(ain);
    const float3 conv= {0.299f, 0.587f, 0.114f};
    float3 luma = dot(f4.rgb, conv);
    return rsPackColorTo8888(luma);
}

//in the app code, init
RenderScript mRS = RenderScript.create(theActivity);
ScriptC_test mScript = new ScriptC_test(mRS, getResources(), R.raw.test);
Allocation mInAllocation = Allocation.createFromBitmap(mRS, mBitmapIn,... Allocation.USAGE_SCRIPT);
Allocation mOutAllocation = Allocation.createTyped(mRS, mInAllocation.getType());

//in the app code, run
mScript.forEach_root(mInAllocation, mOutAllocation);
mRS.finish();
mOutAllocation.copyTo(mBitmapOut);
```

*dynamic (script-from script) parallelism discussion in the backup

SA2014.SIGGRAPH.ORG

SPONSORED BY



Here is example... let's inspect the **kernel code first**

1)The function root() is conceptually similar to main() in C. When a script is invoked by the runtime, this is the function that will be called for each item in the incoming allocation In this simple example you still can spot few kernel language features: 2) unpack/pack pixel functions and

3)math built-in . It is also possible to call scripts from the scripts, but I would refer to the foils in backup on the advanced topics

On the host side

4) Renderscript applications need a context object. 5)Notice the glue class , which is generated by the compiler. Scripts are raw resources in an application's APK stored as LLVM code, so that scripts are compiled on the host with clang, and all the high level optimizations are happening at the application compilation stage. The translation to actual device code and device-specific optimizations still happens in the runtime. 6)The next function calls create a compute allocation from the Bitmap and the output allocation. We also specify the potential uses so the system will choose the correct type of memory. We will cover usage flags in details later. 7) Finally we invoke a root function so it does a compute launch. The forEach will run across multiple threads if the device supports these, it may actually occur on the CPU/GPU or DSP. 8) Finally we wait for completion and copy the result to some output bitmap. We will cope with this copying later.

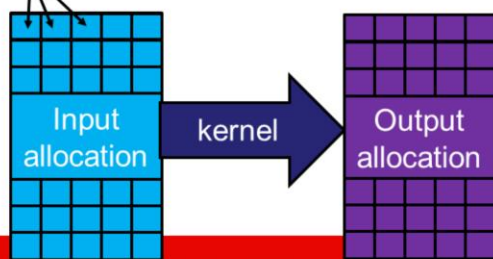
Notice that this Java code is really simple!

RenderScript*: Transparent Parallelism

forEach is executing the kernel for each individual item in the mem allocation:

- No need to specify dimensions
- Compiler is free to pack/vectorize
- Run-time is free to
 - Run multiple threads
 - Select device

elements



```
// kernel declaration with API level prior to 17
void root(const uchar4 *ain, uchar4 *aout, const void* p)
{}

// kernel declaration with newer API levels
//additional allocation as a script global
rs_allocation alloc;
uchar4 attribute ((kernel)) transform2 (uchar4 in,
uint32 tx, uint32 ty)
{
uchar4 pixel = rsGetElementAt_uchar4(alloc, x, y);
return pixel;
}
```

SA2014.SIGGRAPH.ORG

SPONSORED BY



A root function does a compute launch doing implicit looping over each and every element of the input allocation.

Declarations...

- 1) A script may have an input [Allocation](#), an output [Allocation](#), or both.
- 2) If more than one input or output is required, those objects should be bound to script globals. Notice the FilterScript style of script declarations like having no pointers, even though the FilterScript is deprecated this style is generally advised as GPU compilers friendlier.
- 3) A script may access the coordinates of the current execution using the x, y, and z arguments. These arguments are optional

This concept of the implicit parallelism will be really important for the rest of presentation.

ScriptGroups

- ▶ Group scripts together, execute them all with a single call
 - ▶ The output from one script becomes the input of another
- ▶ Enables some behind the scenes optimizations
 - ▶ Like memory tiling or kernel fusion
 - ▶ Runtime analyzes the dependency DAG of the group
- ▶ Possibly automatically optimizes any intermediate allocations away

```
//1. specify the chain of kernels
ScriptGroup.Builder b = new ScriptGroup.Builder(mRS);
b.addKernel(mYuv.getKernelID());
b.addKernel(mAnotherGroup.getKernelID());
mGroup = b.create();

//2. specify inputs/outputs and execute
mGroup.setInput(mYuv.getKernelID_root(), mAllocIn);
mGroup.setOutput(mYuv.getKernelID_root(), mAllocOut);
...
mGroup.execute();
```

SA2014.SIGGRAPH.ORG

SPONSORED BY



You can group scripts together and execute them all with a single call as if they were part of a single script.

This allows Renderscript to optimize execution of the scripts in interesting ways: for example fuse pipeline to improve data locality which somewhat alleviates missing support for execution groups and shared memory – sort of explicit tiling available in for example in OpenCL

The API is straightforward:

- 1) You create a ScriptBuilder,
 - 2) add the scripts
 - 3) And finalize the sequence
- Then you 4) specify the inputs/outputs
- 5) fire and forget

Few RenderScript* Samples and Tests

- ▶ <https://android.googlesource.com/platform/frameworks/rs/+master/java/tests/>
- ▶ Physics and VP9 benchmarks may also appear
- ▶ ComputeBenchmark is a built-in functions “speedometer”
- ▶ ImageProcessing versions are the most popular benchmarks
 - ▶ [Java and native numbers single and multi-threaded vs RenderScript on the CPU and GPU](#) for the ImageProcessing



SA2014.SIGGRAPH.ORG

SPONSORED BY



As information and especially non-toy examples of RS are really scattered, here is the link to Google’s example and test in the Android repository.

At the moment, since general focus of RS is image processing or computational photography, so the ImageProcessing seems to an important benchmark in the community and probably Google

And I put a link with very impressive numbers for the ImageProcessing benchmark.

In general it clearly demonstrates that GPUs are generally much better for things like typical images manipulations.

But as you can imagine there are opposite cases for algorithms are really faster on CPU and this is promise of RS runtime to pick up best device for any particular task automatically.

Common RenderScript* Optimizations : USAGE_SCRIPT + USAGE_IO_OUTPUT

```
//Tell runtime that the Allocation serves as a SurfaceTexture producer  
Init SurfaceTexture outSurfText = ((TextureView) findViewById(R.id.textureView)).getSurfaceTexture();  
allocationOut = Allocation.createTyped(rs, builder.create(), USAGE_SCRIPT | USAGE_IO_OUTPUT);  
allocationOut.setSurface(new Surface(outSurfText));
```

```
script.forEach_root(allocationIn, allocationOut); //call the script  
rs.finish(); //wait for completion  
Update //the data in the Allocation is undefined after this operation:  
allocationOut.ioSend(); // send the buffer  
outSurfText.updateTexImage(); //notify the texture on the update
```

Warning: RenderScript
Support Library doesn't
support USAGE_IO_INPUT
/USAGE_IO_OUTPUT

SA2014.SIGGRAPH.ORG

SPONSORED BY



Since the time the RS was first shipped in Android 3.0 (and was in use by platform apps such as wallpapers), the major leap was Android 4.2 with RS GPU acceleration
Now it offers dedicated flag for RS output sharing, e.g with OpenGL
Notice the USAGE_SCRIPT flag that indicates that the Allocation will be bound to and accessed by scripts (this is remnant of time when RS was also a graphics API)

The general point of sharing is 1) setting the underlying surface for Allocation
2)and synch upon update

USAGE_IO_INPUT

android.hardware.camera2

The android.hardware.camera2 package provides an interface to individual camera devices connected to an Android device. It defines a camera device as a pipeline, which takes in input requests for capturing a single frame or a video stream, plus a set of output image buffers, video frames, having multiple requests in flight, and open available camera devices. Camera devices provide a set of static properties. The CameraCharacteristics object is used to query and open available camera devices. Camera devices provide a set of static properties. The CameraCharacteristics object is used to query and open available camera devices. Camera devices provide a set of static properties. The CameraCharacteristics object is used to query and open available camera devices.

This package requires API level 21 or higher.

This document is hidden because your selected API documentation is 19. You can change the documentation selector above the left navigation.

developer.android.com/guide/topics/media/camera.html

Develop > API Guides > Camera

- Introduction
- App Components
- App Resources
- App Manifest
- User Interface
- Animation and Graphics
- Computation
- Media and Camera

The Basics

The Android framework supports capturing images and video through the `android.hardware.camera2` API or camera `Intent`. Here are the relevant classes:

- `android.hardware.camera2`
This package is the primary API for controlling device camera. It can be used to take pictures or videos when you are building a camera application.
- `Camera`
This class is the older deprecated API for controlling device cameras.
- `SurfaceView`

1. Create an Allocation with YUV type and the USAGE_IO_INPUT
2. Obtain the Surface with Allocation.getSurface()
3. Pass the Surface to configureOutputs() of Camera2
4. Wait for the data to arrive with Allocation.ioReceive()

SA2014.SIGGRAPH.ORG
SPONSORED BY

Now with new Camera HAL in Lollipop the USAGE_IO_INPUT is also supported. It is conceptually similar to output sharing, but now you get surface from the Allocation and pass it to some producer. Then you just wait for the data to arrive

Until API Level 21 the IO_INPUT worked only for Canvas

```

mAllocation.getSurface();
    Canvas canvas = surface.lockCanvas();
    //draw
    ....
    surface.unlockCanvasAndPost(canvas);

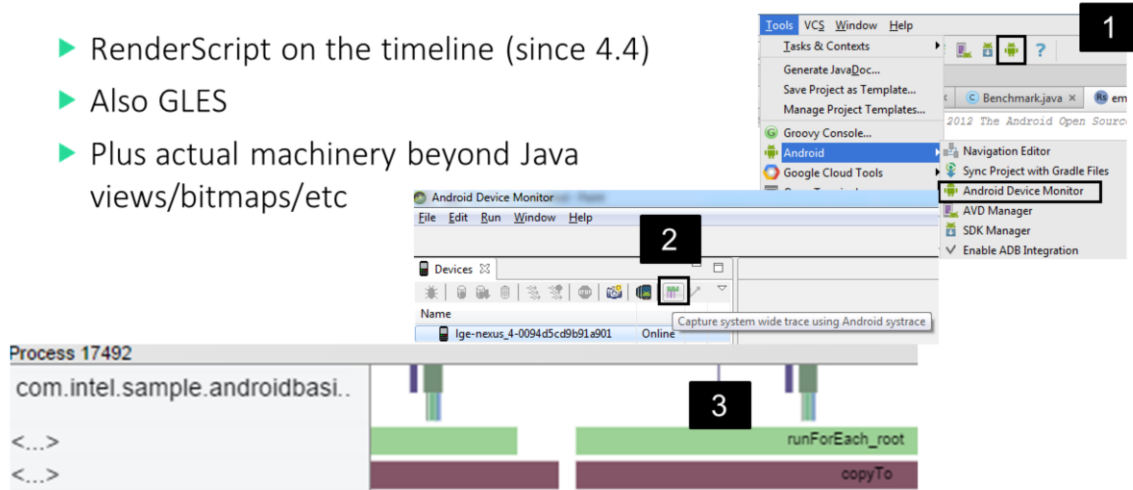
```

mAllocation.ioReceive();

Warning: this code is not HW accelerated

Android systrace: The Very First Tool for RenderScript*

- ▶ RenderScript on the timeline (since 4.4)
- ▶ Also GLES
- ▶ Plus actual machinery beyond Java views/bitmaps/etc



SA2014.SIGGRAPH.ORG

SPONSORED BY



Maybe the only truly HW-vendor agnostic tool, available as a part of Android Device Monitor in Android Studio and Eclipse (first screenshot), Then you should select tracing in the ADM itself (second picture) and finally you can look to the trace in the Chrome (3)

It is the only tool by the best of my knowledge that supports displaying all RS calls on the timeline – for example here I can see how forEach for my kernel root function runs in parallel with copying the results from prev. frame Also it supports OpenGL (but still not the Compute Shaders) and what I love most: some machinery behind GUI elements like views and bitmaps

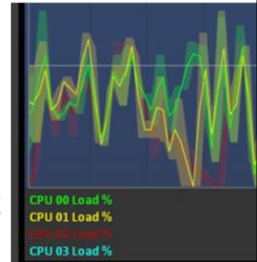
Where RenderScript* Actually Runs?

- ▶ Look for RenderScript diagnostics in the logcat output:

```
V/RenderScript 0x44184368 Launching thread(s), CPUs 4
```

```
E/RenderScript : Successfully loaded runtime: libRSDriver_adreno.so
```

- ▶ Look to the CPU/GPU activity with any tool, e.g. Intel® GPA
- ▶ Remember that things like recursion trigger the CPU fallback
- ▶ Intrinsics presumably may run on DSP
- ▶ Opportunity for the real scheduler (CPU/GPU/DSPs)



SA2014.SIGGRAPH.ORG

SPONSORED BY



Well I was having hard times figuring out where a script actually run?

Look for RenderScript diagnostics in the logcat output:

Look to any tool that tracks CPU/GPU activity for indirect indications

Remember that things like recursion trigger the CPU fallback

And intrinsics presumably may run on DSP

The potential downside of any automatic scheduler is that if it decided to run some RS on the CPU concurrently with the rest of your Java or Native parallel code this would easily oversubscribe the CPU, so a general Android tip to never run heavy things on main thread holds true

RenderScript* summary	
Support for devices beyond GPU	Yes
Explicit jobs scheduling	No
Simple API	Yes
Rich kernel language	Yes
OS integration	Yes
Interoperability with other APIs	Yes
Low-level tuning	No
Portable across OSes	No



Summary is that RenderScript is really high-level API, with deep integration into the underlying OS. This allows to keep the API really focused and minimalistic, but limits numbers of low-level optimizations, I mean the missing support for the shared or local memory.

Also though multiple devices are supported in theory, but manual scheduling or even device selection is not possible.

Khronos OpenCL™

Android, iOS, Windows, Linux, OS X

SA2014.SIGGRAPH.ORG

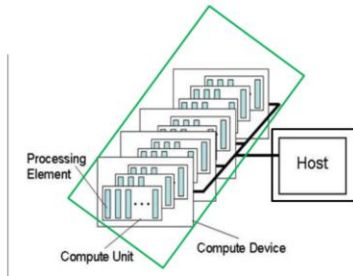
SPONSORED BY



Now let's discuss OpenCL which is somewhat opposite case by many points

OpenCL™ Is Really Low-Level API For Compute

Image: https://www.khronos.org/assets/uploads/developers/library/overview/opencl_overview.pdf



- Clearly separates *host* from the devices
 - Abstracts DSPs/ISPs/FPGAs, not just GPUs
 - Multi-device “shared” contexts (in backup)
- Dedicated C-based kernel lang (like other APIs)
 - Much richer set of math functions (~as in CUDA*)
- FP arithmetic based on IEEE rules
- Many ways to query the underlying device info/config*
 - E.g. available memory, number of cores/units, data alignments, prefer. datatypes

*more comparison to CUDA in the slide notes

SA2014.SIGGRAPH.ORG

SPONSORED BY



OpenCL does have devices, but highly abstracts the entire notion. So you have 1) host which is where your main program runs and the 2) OpenCL devices that look identically from the SW perspective, devices may include multi-core CPUs, GPUs, Cell-type architectures **and other parallel processors such as DSPs**, or other fixed function blocks

Feature-wise OpenCL is quite similar to CUDA, but being entirely vendor-controlled language the CUDA is typically ahead of OpenCL, since there is no need for agreement on features - like discussions Khronos group drives for OpenCL each time, leading to Least Common Denominator solution in many cases.

But of course CUDA supported just NV GPUs

Also unlike CUDA where you need the entire toolchain from NV, with OpenCL you can just use your favorite host compiler, 3) as basically you only link with the OpenCL library from Khronos and that's it.

OpenCL™ *n*-DRange Concept

Kernel invocation for each data point, for example

- ▶ Process a 1024 x 1024 image
- ▶ Global size dimensions: 1024 x 1024
- ▶ 1 kernel execution per pixel: 1,048,576 total executions (work-items)

```
void
scalar_mul(int n,
           const float *a,
           const float *b,
           float *c)
{
    int i;
    for (i=0; i<n; i++)
        c[i] = a[i] * b[i];
}
```



```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
    size_t id = get_global_id(0);
    c[id] = a[id] * b[id];
} // execute over "n" work-items
```

SA2014.SIGGRAPH.ORG

SPONSORED BY

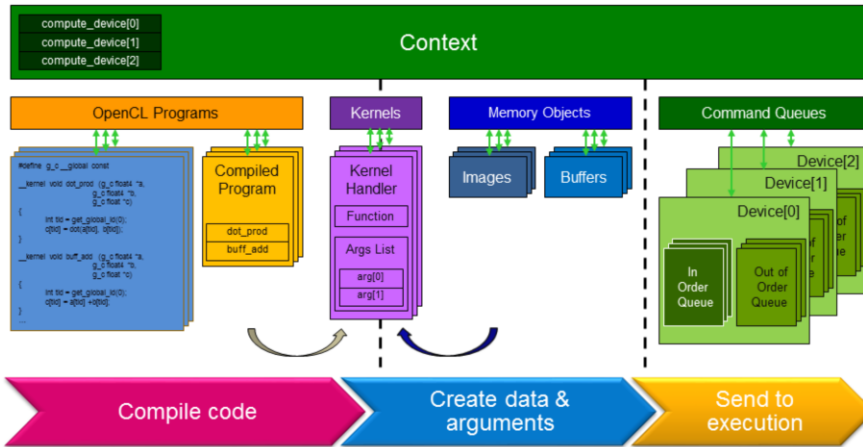


The big idea behind OPENCL is generalized notion of n-dimensional space of work-items to operate on.

So instead of explicit loops 1) in your C-code now you specify NDRanges and you write kernels that are automatically executed for each point of the NDRange's index space
2) each point is having unique ID

This is very similar to RenderScript

OpenCL™ Runtime Objects



SA2014.SIGGRAPH.ORG

SPONSORED BY



Everything in runtime happens within the OpenCL context: Context is a sandbox for your OpenCL code and resources.

OpenGL is nowhere easier if you want plain compute, given the binding, targets, usages, need for quad setup and fake rendering if you use pixel shaders, etc

<more OpenCL™ foils in backup>

The image displays a sequence of overlapping presentation slides illustrating the OpenCL workflow. The slides are as follows:

- OpenCL™: Setting up the environment**
 - Select an
 - Select a
 - Create a
 - Create a
- OpenCL™: Create and Build the Kernels**
 - Define source o
 - Build the progra
 - Fetch and print
 - Kernels can also
- OpenCL™: Setup Memory Objects**
 - For the input bu
 - A_buffer =
 - B_buffer =
 - D_buffer =
- OpenCL™: Define the kernel**
 - Create ke
 - Attach an
- OpenCL™: Submit commands / Execute**
 - Enqueue t
 - Read back
 - This API ca beyond the
- OpenCL™ primer on multi-device support**
 - Multi-device support is leveraged thru the [shared contexts](#)
 - Objects are shared between devices in the context
 - Objects cannot be updated concurrently—do explicit synchronizations
 - OpenCL 1.2 features sub-buffers for writes to the non-overlapping regions
 - Still, a separate queue per device: no “shared queue”
 - Any load-balancing logic left to your app

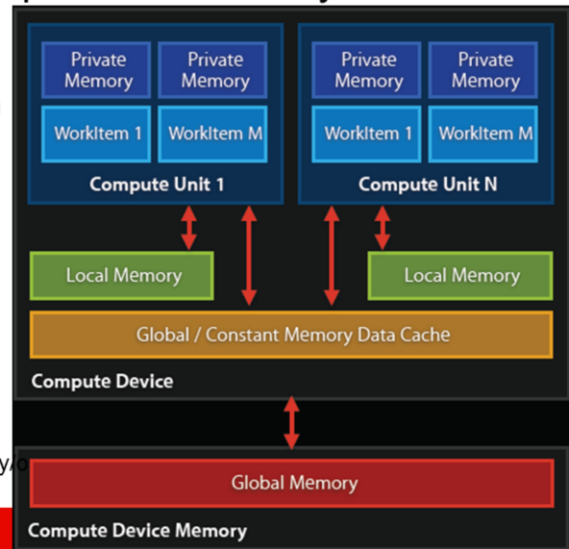
SA2014.SIGGRAPH.ORG

I wouldn't go much into details on OpenCL workflow- just refer to the backup offline, as relations of the OpenCL runtime objects are fairly complex, however this is quite similar to CUDA again

Work-groups and General OpenCL™ Memory Model

- ▶ In turn, work-items are executed in groups (hence **work-groups**)
- ▶ General execution order for work-items is totally impl-defined
- ▶ Each work-group has data sharing between work-items via **local memory**

Image from:
https://www.khronos.org/assets/uploads/developers/library/view/opencl_overview.pdf



SA2014.SIGGRAPH.ORG

OpenCL spec doesn't mandate any particular order of execution for individual work-items, which opens doors for efficient vectorization and threading. But still spec allows you to group work-items in work-groups to share local memory and work-group barriers.

The local (or "shared" in OpenGL/CUDA parlance) memory is essentially a scratchpad used for explicit caching. It is quite vendor-specific trick though, as in some GPUs it is actually mapped to the same system mem (like for low-power Freescale's multimedia processors you can find in Kindle).

For the rest of OpenCL memory types...well, for System On a Chip solutions the global OpenCL mem is basically backed with regular system mem, and the private mem is just like stack which is cached by HW or simply kept in registers (if the size permits)

(No) Mobile OpenCL™ Support?

Official list of the vendors and OpenCL conformant devices - [here](#)

- ▶ Officially *no* for iOS and Android
 - ▶ <http://streamcomputing.eu/blog/2014-06-30/openc1-support-recent-android-smartphones/>
 - ▶ <http://stackoverflow.com/questions/18847255/is-available-openc1-on-ios>
- ▶ Still many vendors and OEMs offer Android OpenCL drivers
 - ▶ With BayTrail Intel offers the OpenCL 1.2 Full Profile support
 - ▶ [Imagination also offers](#) OpenCL support
 - ▶ <https://developer.qualcomm.com/mobile-development/maximize-hardware/mobile-gaming-graphics-adreno/tools-and-resources> (OpenCL 1.2)
 - ▶ <http://developer.sonymobile.com/tag/openc1/> (OpenCL 1.1)
 - ▶ <http://malideveloper.arm.com/develop-for-mali/sdks/mali-openc1-sdk/> (1.1)

SA2014.SIGGRAPH.ORG

SPONSORED BY



Finally a question on which devices OpenCL is supported

You should separate the formal OpenCL support for any particular device, and actual driver availability for the SPECIFIC OS.

For example you can find the device you are looking for in the official khronos list which is the first link on the page, but be careful as it may be conformant on some weird Embedded Linux installation, like for Vivante GPUs, while Android OpenCL drivers may also exist (but not passing all the official conformance routine, just like MediaTek's products).

And there is no consistency here – for example Sony offers just OCL 1.1 drivers for it's Snapdragon based phones, while QCOM already offer OCL 1.2 drivers

OpenCL™ Portability

- ▶ Carefully track the OpenCL versions/extensions support by devices... just like for OGL/GLES
 - ▶ OpenCL 1.1/1.2 support currently dominate, not 1.0 or 2.0
- ▶ Beware of possible endianness issues (avoid bitcasts in the kernels)
- ▶ A number of edge cases for data type conversions and math functions that are more rigorously defined than in C99

For OpenCL 1.1->1.2->2.0 details refer to the backup

SA2014.SIGGRAPH.ORG

SPONSORED BY



Contrary to a popular belief, the fact the OpenCL natively talks to a larger range of devices, doesn't mean that your code will actually run on all of them. So keep OpenCL versions and profiles in mind

With CUDA it is less apparent as range of hardware supported is smaller, you still need to distinguish the architectures (pre-Fermi, Fermi, Kepler, and finally Maxwell for the next Tegra).

And yes there are caveats even when porting from native code – this is about last bullet

OpenCL™ 1.2 **Embedded** Profile

- ▶ 64-bit data types i.e. long, double etc are optional
- ▶ Support for 3D images is optional
- ▶ CL_FLOAT images support only CL_FILTER_NEAREST sampling
- ▶ Relaxed rules for NaN, INFs, etc,
- ▶ Relaxed precision of the math functions is totally impl-defined
- ▶ All sort of built-in atomic functions are optional

SA2014.SIGGRAPH.ORG

SPONSORED BY



To summarize the difference of Embedded profile: many many things are **OPTIONAL** and accessed thru extensions, 64-bit data types like long or double types for example and also all sort of atomics

EP can also be not fully IEEE compliant

There is no Embedded profile in CUDA btw, so Tegra offers the same CUDA as you may find in the full-blown HPC server solutions but mabe some things are backed by SW

OpenCL™ 1.2 **Embedded** Profile, continued

- ▶ For local mem the min mandated value is just 1 KB
- ▶ Min allowed read-only images in kernel is 8 (for writes- just 1!)
- ▶ Maximum image width/height is just 2048

Notice that list of mandated image formats is really short:

- ▶ RGBA CL_UNORM_INT8/UNORM_INT16
- ▶ SIGNED_INT8/_INT16/_INT32
- ▶ UNSIGNED_INT8/_INT16/_INT32
- ▶ HALF_FLOAT/FLOAT

SA2014.SIGGRAPH.ORG

SPONSORED BY



Few more examples of hardware-implied limitations that basically match the typical OpenGL-compute limits as we will study shortly
But unlike OpenGL ES the number of channels and data types for images in OPenCL is really limited

Important OpenCL™ Extensions: OpenCL-OpenGL sharing with khr_gl_sharing

- ▶ Create an OpenGL context and afterwards an OpenCL context

```
clGetGLContextInfoKHR(... CL_DEVICES_FOR_GL_CONTEXT_KHR); //get a device associated with GL context
```

```
clCreateContext(gl_context...device);
```

- ▶ Resources might fail sharing on some platforms otherwise

```
cl_mem mem =clCreateFromGLBuffer (...cl_mem_flags, GLuint,...);
```

```
cl_mem mem =clCreateFromGLTexture (...cl_mem_flags, GLenum tex_target,GLint miplevel, GLuint tex...);
```

- ▶ Beware of synch caveats between APIs

```
clEnqueueAcquireGLObjects(queue,.. mem);
```

```
clEnqueueReleaseGLObjects(queue, &mem...);
```

Other important extension:
cl_khr_egl_image

SA2014.SIGGRAPH.ORG

SPONSORED BY



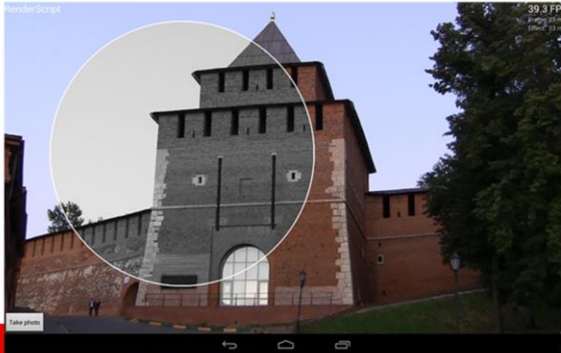
Some vendors provide extended functionality over the standard OpenCL spec via extensions. These are still blessed by Khronos but provided by vendors within their drivers/SDKs.

Specifically, interoperability with the OpenGL is managed through dedicated extension. There are many caveats to get actual zero-copy interoperability, for example order of context creation and resource allocations, synchronization and so on.

There are also extensions like `cl_khr_egl_image` which allows to share data thru `EGLImage` (with camera, video decoders, native surfaces, etc)

Considerations For Optimizations

- On GPUs, whether OpenCL or RenderScript is faster varies greatly between devices
 - The same for sustainability of FPS
 - Be careful as RS may run on CPU
- There are performance inversions on some devices (plus different throttling behavior)



Example: same image effect in RS and OpenCL

<https://software.intel.com/en-us/articles/renderscript-basic-sample-for-android-os>

<https://software.intel.com/en-us/android/articles/opencl-basic-sample-for-android-os>

Look for specific data? Try benchmarks like [CompuBench](#) from Kishonti: OpenCL perf directly [comparable](#) to RenderScript

SA2014.SIGGRAPH.ORG

SPONSORED BY



There are of benchmarks comparing OpenCL to RenderSript and the rest of APIs for example Compubench from Kishonti.

I intentionally didn't put any specific perf numbers here, as these are fluctuating greatly between devices:

On GPUs, whether OpenCL or RenderScript is faster varies greatly, but typically no clear performance advantage, unless you tune the code.

Another interesting observation is that in all my experiments the stability of FPS is also very device-dependent (supposedly due to different level of interaction with 3D and rest of Android for each implementation)

IF you want to play some identical coded in both APIs yourself, try the code from the links.

OpenCL™ summary	
Support for devices beyond GPU	Yes
Explicit jobs scheduling	Yes
Simple API	No
Rich kernel language	Yes
OS integration	No
Interoperability with other APIs	Yes
Low-level tuning	Yes
Portable across OSes	Yes



Summary is that OpenCL is really low-level dedicated API for Compute, everything in your hand, so all sort of scheduling and tuning are possible.

Having no OS integration might be troublesome as OS might need to have control in many situations (e.g. when the battery is low).

Another implication is that OpenCL typically requires installing a separate driver and separate libraries.

Keeping the actual support question separated for a while, the OpenCL seem to have the strongest portability promise anyway.

Apple Metal*

Android, iOS(8), Windows, Linux, OS X

SA2014.SIGGRAPH.ORG

SPONSORED BY



What is Apple's Metal*?



- ▶ New graphics and compute API from Apple
 - ▶ Currently only for iOS 8 (Apple A7 and later hardware)
- ▶ Covers much of same functionality of OpenGL and OpenCL
 - ▶ C++11-derived kernel language compiled to Apple IR (AIR)
- ▶ No discussion of Metal graphics API for the rest of the preso
 - ▶ Parallel encoding of commands on multiple threads (like D3D12/Mantle)
 - ▶ Some choices motivated by tiling architecture
- ▶ Finally NO EXTENSIONS, Hurrah!



Apple took a clean-room way with it's new unified graphics and compute API
From the kernel language persp, the Metal uses C++11 both for shading and compute kernels, which similarly to RS are compiled to IR (AIR).

Metal is not related to Swift, it's original API is ObjC, but Metal is callable from Swift (no option for straight C/C++ by the best of my knowledge)
I'll use Swift syntax for subsequent examples

We wouldn't cover the graphics side of things in this preso. However Metal exposes many similarities to DX12 or Mantle, for example textures and samplers are fully separated (like D3D, unlike GL)

Apple Metal* Shading Language

- ▶ Kernel language similar to GLES/OpenCL, but with subset of C++
- ▶ Generally the precision of math routine matches the OpenCL
- ▶ But Metal is compliant just to a subset of the IEEE 754 standard
 - ▶ Denorms may be flushed to zero
 - ▶ Rounding Mode is either RTZ or RTN
- ▶ Quite limited atomics support (int/uint only, just like in GLES)

SA2014.SIGGRAPH.ORG

SPONSORED BY



Overall, Metal is similar in functionality to OpenCL and it is more about having niceties such as C++11 support in the kernel language (the static subset) so you can use templates, overloading, some static usage of classes etc. For the rest it's capabilities approximately matches those of OpenGL Compute or OpenCL

In contrast to Metal the OpenCL 2.0 also supports:

`atomic_(u)long*`

`atomic_float` and `atomic_double*`

`atomic_(u)intptr_t*`, `atomic_size_t*`, `atomic_ptrdiff_t*`

*requires additional OpenCL extensions

Metal* Compute Kernel Example

- ▶ Memory address-space qualifiers familiar from OpenCL C
- ▶ Uses C++11 attribute syntax for metadata

```
constant half3 kRec709Luma = half3(0.2126, 0.7152, 0.0722);
kernel void grayscale(texture2d<half, access::read> inTexture [[ texture(0) ]],
                    texture2d<half, access::write> outTexture [[ texture(1) ]],
                    uint2 gid [[ thread_position_in_grid ]] )
{
    half4 inColor = inTexture.read(gid);
    half gray = dot(inColor.rgb, kRec709Luma);
    half4 outColor = half4(gray, gray, gray, 1.0);
    outTexture.write(outColor, gid);
}
```

SA2014.SIGGRAPH.ORG

35 SPONSORED BY



Being the subset of C++11 the Metal Language

Supports what you'd expect: templates, operator overloading, ...

Leaves out what you'd expect: virtual functions, exceptions, stdlib, ...

Unlike RenderScript, you don't have recursion, and no dynamic parallelism either

Metal uses C++11 attribute syntax for linking user-defined parameters to

- 1) API binding slot
- 2) Or connection to system-defined input or output

Metal*: Device and Command Queue

▶ MTLDevice

- ▶ Represents a GPU
- ▶ Serves as a factory for other objects

```
let device = MTLCreateSystemDefaultDevice()
```

▶ MTLCommandQueue

- ▶ Used to submit and order command buffers

```
let queue = device.newCommandQueue()
```

SA2014.SIGGRAPH.ORG

SPONSORED BY



Again, if you have used OpenCL or CUDA, then your skills will transfer easily to Metal.

Unlike OpenCL in Metal the things are GPU-only: Currently only works on GPUs, and not say on a CPU or DSP.

Command queues are similar to multiple streams in CUDA or multiple command queues in OpenCL.

Metal*: Command Buffer and Encoder

- ▶ MTLCommandBuffer

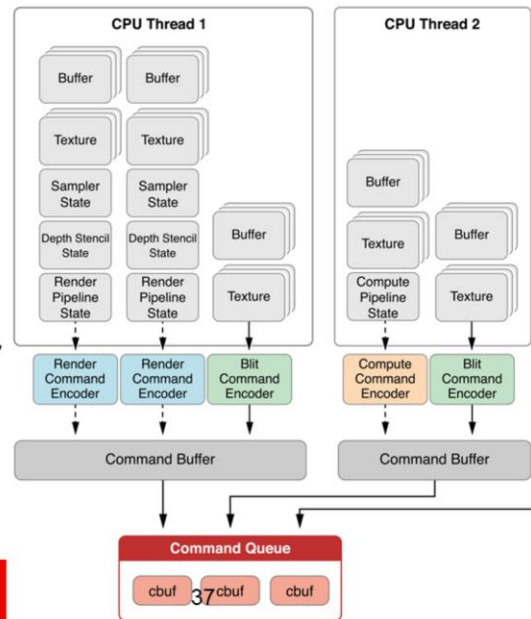
- ▶ Created on a particular queue

```
let cmdBuffer = queue.commandBuffer()
```

- ▶ Granularity for multi-threaded preparation, ordering, and submission work

```
cmdBuffer.enqueue() // add to queue (in order)
```

```
cmdBuffer.commit() // set as ready-to-run
```



SA2014.SIGGRAPH.ORG

Graphics programmers will also appreciate the tight integration with the graphics pipeline, similarly to GL compute.

On the memory model side you can create CPU-GPU shared buffers just like with OpenCL.

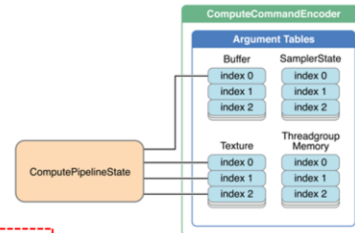
Finally parallelism is achieved by creating multiple command buffers- one or many for compute and another one or many of them) for rendering, which again matches OpenCL

Metal*: Command Encoder

- ▶ One encoder can be “active” and writing to command buffer at a time
- ▶ Distinct sub-types of encoder for rendering (3D), compute, and blit (2D)
- ▶ Example: MTLComputeCommandEncoder

```
let computeEncoder = cmdBuffer.computeCommandEncoder()
computeEncoder.setComputePipelineState(filterKernel);
// Set resources for the kernel
computeEncoder.setTexture(inputImage, atIndex:0);
...

computeEncoder.executeKernelWithWorkGroupSize(MTLSize(16,1,1),
                                                workGroupCount: MTLSize(N,1,1))
computeEncoder.endEncoding()
computeEncoder.commit()
```



http://devstreaming.apple.com/videos/wdc/2014/605/xyycz4pd0h6/605/605_working_with_metal_advanced.pdf

Again, if you know any GPU compute APIs like OpenCL or CUDA etc. you will be at home with Metal, 1) since here you again have work-items organized in work-groups (16kB of local memory) with barrier synch.

Total work is specified as workgroup size * number of workgroups, just like in GLES-compute, unlike OpenCL where you specify your work with finer granularity and can use offsets and other advanced things

[MTLComputeCommandEncoder](#) encodes data-parallel compute processing state and commands that can be executed on the device. You can create “monolithic” pipeline state object for compute (just like for rendering) or use dedicated functins

Metal* summary

Support for devices beyond GPU	No
Explicit jobs scheduling	No
Simple API	No
Rich kernel language	Yes
OS integration	Yes
Interoperability with other APIs	Yes
Low-level tuning	Yes
Portable across OSes	No

Summary is that Metal is clearly GPU-oriented, unified compute/rendering API. It offers advanced kernel language and many low-level features plus tight integration with the specific OS.

Khronos OpenGL ES™, Fragment Shaders

Android, iOS, Windows, Linux, OS X

SA2014.SIGGRAPH.ORG

SPONSORED BY



Doing Compute With Fragment Shaders

Lot of code to init egl

Lot of setup code

Lot of code to do rendering

1. Init EGL and create render surface with EGL
2. Load vertex and fragment shaders
3. Create a program object, attach vert and frag shaders, and link
4. Set the viewport and clear the color/depth buffers
5. Render (e.g. full screen quad)
6. Use the contents of the color buffer (display or read back)

SA2014.SIGGRAPH.ORG

SPONSORED BY



Fragment or pixel shaders actually have been used for a long time to run some level of general compute. Indeed, as long as the data is read by the shader, it can compute more or less whatever it wants from the data, and output the computed result back to the FBO. Probably the biggest limitation here is the lack of scatter support: you cannot write to an arbitrary location in the FBO.

Anyway, the resulting texture can then be used as a source for any other fragment shaders in the rendering pipeline or just displayed.

And there are many things you should do like specifying the quad, texture coordinates, and so on, and if you are so new to graphics and willing to do a simple compute like image processing ...pixel shaders are nowhere easy thing to start with- it's a lot of code especially comparing to say RenderScript. There are plenty of libraries to abstract out much of the boilerplate code, finally there are full-blown game engines that hides all this things from you.

Doing Compute With Fragment Shaders

Example of a shader to output full-screen textured quad

```
#pragma version 200 es
precision mediump float;
uniform sampler2D u_Texture;
varying vec2 v_TexCoordinate;
void main()
{
    gl_FragColor = texture2D(u_Texture, v_TexCoordinate);
};
```

SA2014.SIGGRAPH.ORG

SPONSORED BY



Of course pixel shaders support was a huge leap step in the programmability and that is why we skip the OpenGL versions prior to 2.0 from discussion- even though limited form of compute also had been possible with fixed pipeline, by use of multi-texturing and texture operations, HW-assisted depth, stencil, even limited atomic ops (via blend ops, etc)

I used the texture fetching shader as an example to stress the fact that compact formats (like 10:3:3 or 5:6:5) and compression support are intrinsic in GL, which is very strong side of OpenGL in general,
not available in OpenCL (which is really limited on the image formats and supported bindings for a texture to be shareable).

Few OpenGL ES™ Milestones

- ▶ GLES 2.0 - Initial support for programmable shaders (Mar'07)
 - ▶ With iPhone 3GS in Jun'09 and Android 2.0+ devices in Mar'10
- ▶ GLES 3.0 –Major texturing improvements and MRTs (Aug'12)
 - ▶ Android 4.3+ devices (~Oct'12), iPhone 5s (Sep'13) and other iOS7 devices
- ▶ GLES 3.1- Compute shaders and Indirect draw commands (Mar'14)
 - ▶ Android deices with Lollipop (Nov'14), no official Apple support yet

SA2014.SIGGRAPH.ORG

SPONSORED BY



As you very well know, the OpenGL ES 2.0 is defined relative to the OpenGL 2.0 specification.

In turn OpenGL ES 3.0 is derived from the OpenGL 3.3 specification (and roughly matches DirectX9)

There are multiple enhancements there like support for multiple render targets and standard texture compression

But from the compute persp the primary addition for GLSL ES 3.0 is full support for 32bit integer and floating point types and operations. Previously only lower precisions were supported.

Notice how for 3.0 vendors are now doing much better job with catch up of the new standard comparing more than 2-3 year lag for GLES 2.0 ,

Also notice shorter cadence for recent GLES spec releases (which is general trend of shifting left, already seen for other compute APIs

Doing Compute With Fragment Shaders

Most regular fragments optimizations are not applicable for compute

- ▶ E.g. avoiding overdraw

Some general rendering optimizations that do work

- ▶ Use built-in functions (like “dot)
- ▶ Optimize the use of transparency
- ▶ Reduce texture bandwidth (use tex compression)
- ▶ Avoid dynamic texture lookups
- ▶ Use precision hints

SA2014.SIGGRAPH.ORG

SPONSORED BY



Overdraw which is common graphics perf pitfall occurs when the GPU draws over the same pixel multiple times. Unfortunately things like depth testing back face culling or drawing in the specific order (e.g. front to back order for opaque geometry) are not applicable when doing compute with PS

Well there is bunch of perfectly legal optimizations from regular pixel shaders practices. Saving the BW is most important, as textures consume a large amount of memory bandwidth, and might easily stall the shader exectutions .

It may not seem obvious, but any calculation on the texture coordinates (say for fancy image processing like twirling the image) result in the dependent texture reads, which slows down things especially on the old-fashioned hw.

Finally use low precision which is generally is acceptable for fragment colors for example when you process image data.

Doing Compute With Fragment Shaders: Limitations

- ▶ Limitation on the shader size, control flow, inputs, etc
- ▶ Texture size limits (e.g. GLES™ 2.0 mandates just 2048x2048)
- ▶ glReadPixels and the workarounds (FBO etc)
- ▶ Accuracy is not guaranteed

SA2014.SIGGRAPH.ORG

SPONSORED BY



To fully leverage the portability promise of Pixel shaders, you should respect the Hardware and spec limits: for example control flow in shaders is generally limited to forward branching and to loops where the maximum number of iterations can easily be determined at compile time.

But most fundamental constraint is that you can use pixel shaders for compute, as long as the precision is acceptable.

Unfortunately, for graphics the speed is almost always preferred over the accuracy, thus precision is nowhere close to IEEE standard which makes compute of certain functions (like some transcendentals) impossible or just too slow.

And that is why we are moving to the compute shaders next

GL ES™ Fragments Shaders summary

Support for devices beyond GPU	No
Explicit jobs scheduling	No
Simple API	No
Rich kernel language	No
OS integration	Yes
Interoperability with other APIs	No
Low-level tuning	No
Portable across OSes	Yes

SA2014.SIGGRAPH.ORG

SPONSORED BY



The summary is that from compute perspective Pixel Shaders are low-level old-fashioned way of harnessing the GPU power for non-graphics work.

It is very limited on the optimizations (and here I again means things like missing notion of shared local memory) and scheduling.

BUT: any 2.0-capable GPU out there can run fragment shaders and that really is about every device today.

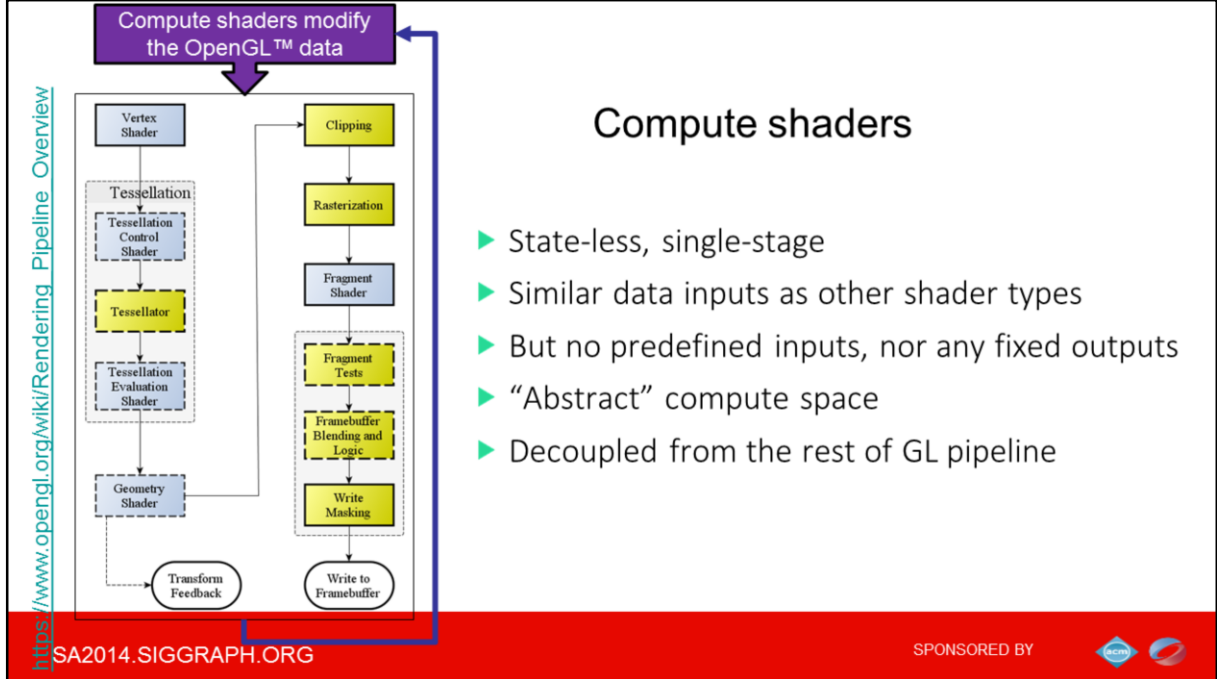
Khronos OpenGL ES™, Compute Shaders

Android, iOS, Windows, Linux, OS-X

SA2014.SIGGRAPH.ORG

SPONSORED BY





Compute shaders

- ▶ State-less, single-stage
- ▶ Similar data inputs as other shader types
- ▶ But no predefined inputs, nor any fixed outputs
- ▶ “Abstract” compute space
- ▶ Decoupled from the rest of GL pipeline

Now we want arbitrary calculations to be executed by the graphics hardware with minimal changes to the GL state machine.

In most respects, a Compute Shader is identical to all other OpenGL shaders, as it has access to many of the same data, such as textures, image textures, atomic counters, and so on.



Yet compute shaders *operate* differently from other shader stages: CS don't have a pre-defined output, but only output things by writing data images, storage buffers or atomic counters.

Finally, Compute shaders are not part of a rendering pipeline and the visible side effects are again through actions on shader storage buffers, image textures and so on.

Well to be fair, the rendering itself is also no longer really a pipeline, with things like transform feedback it is more like the Shenzhen subway map!

Shaders vs Kernels

Compute Shader	OpenCL™
<pre>#version 310 es layout(std430, binding=0) buffer data { int dst[]; }; layout(local_size_x = 16, local_size_y = 16) in; void main() { uint x = gl_GlobalInvocationID.x; dst[x] = 1; }</pre>	<pre>__kernel void foo(__global int* dst) { int x = get_global_id(0); dst[x] = 1; }</pre>

SA2014.SIGGRAPH.ORG SPONSORED BY   49

Conceptually compute shaders are very similar to OpenCL, but the upside is that you use the same GLSL language, something that all OpenGL programmers should be already familiar with .

Again, compute shaders have access to the same data as for example pixel shaders plus the Shader Storage Buffer Objects - a new feature introduced along with compute shaders to allow flexible and arbitrary structured inputs and outputs for compute shaders (well beyond simple integer buffer from this example).

There are still graphics specifics like layouts like buffers binding and additional alignment layout options

Notice that shader needs to declare the number of work-items in a work-group in a special GLSL layout statement, whilst in OpenCL it is typically specified via host APIs. The built-in variables that uniquely identify this particular invocation of the compute shader among *all* invocations of this compute dispatch call are the same to OpenCL.

(You can use the empty brackets, but only on the last element of the buffer)

	OpenGL™ CS	OpenCL™
Context/Device	The same context/device as for rendering	clCreateContext for specific device
Command Queue	No concept of queue	clCreateCommandQueue
Allocating Resources	glGenBuffer, glGenTextures	clCreateBuffer, clCreateImage
Binding	glBindBufferBase(GL_SHADER_STORAGE_BUFFER)	n/a
Kernel/Program APIs	glCreateProgram() glCreateShader(GL_COMPUTE_SHADER) glCompileShader() ... glUseProgram()	clCreateProgramFromSource clBuildProgram clCreateKernel
Kernel Arguments	glUniform1i(glGetUniformLocation(name))	clSetKernelArg()
Kernel Launch	glDispatchCompute(num work groups) glDispatchComputeIndirect(GLintptr)	clEnqueueNDRangeKernel (global work size, local work size)

SA2014.SIGGRAPH.ORG

SPONSORED BY



50

It is quite natural to compare GLES compute to OpenCL from host API level persp as well: For example compute shaders use the same context as does the OpenGL rendering pipeline.

Now comparing to OpenGL the OpenCL setup is quite cumbersome with things like queries for platforms, devices, creating contexts and explicit queues.. There is nothing like this in OpenGL.

And recall that OpenCL typically requires installing a separate driver and separate libraries. While compute Shaders are just there as part of the OpenGL.

The rest of APIs quite similar –like kernel and other general resource management.

For kernel dispatching, the interesting OpenGL compute goodie is DispatchComputeIndirect - the execution parameters for the next shader can be written to a buffer from within previous shader. No need to go back to host.

Advanced Topics: Shared Memory & Barriers

- Similar to OpenCL™ “local” memory

- Shared between all invocations
- Within a work group

```
shared uint histogram[256];  
histogram[id] = 0;  
memoryBarrierShared();  
barrier();
```

- Usual set of memory barriers

- Plus `memoryBarrierShared()` for *shared* variables ordering
- Also `groupMemoryBarrier()` which orders read/writes for the current work group

- Finally `barrier()` as explicit synch

- Between all invocations in the work group

There are some applications, such as image convolution, where threads within a workgroup can share a bulk of data. There is a way to use a shared array that all of the threads in the work-group can access. All this is the same as OpenCL “local” memory. Shared variable access uses the rules for incoherent memory access. This means that the user must perform certain synchronization (like general group barriers or dedicated `memoryBarrierShared()`).

There is usual set of mem barriers to make sure the corresponding memory accesses (e.g. resulting from the use of images/buffers/etc) are completed. While all sorts of memory barriers just synchronize the *memory*, they do not prevent the execution of the threads to cross the barriers. So it is the `barrier()` function specific to compute shaders so work group will not proceed until all invocations have reach this barrier.

Notice that just like with OpenCL barriers the `barrier()` can be called from flow-control, but it can only be called from dynamically *uniform* flow control.

Ordering Compute With The Rest of GLES Commands

```
glBindBufferBase( GL_SHADER_STORAGE_BUFFER, 4, mySSBO);  
glUseProgram( myComputeUpdatePointsProgram );  
glDispatchCompute( TOTAL_POINTS/ GROUP_SIZE, 1, 1 );  
...  
glMemoryBarrier(GL_VERTEX_ATTRIB_ARRAY_BARRIER_BIT);  
glUseProgram( myRenderingProgram );  
glBindBuffer( GL_ARRAY_BUFFER, mySSBO); //use the results as VBO  
...
```

SA2014.SIGGRAPH.ORG

SPONSORED BY



So far we have dispatched the job for compute. Now if we want for example to render the results on a screen we need to make sure the compute shader will finish the job before the actual draw command starts using the VBO buffer updated by the compute shader.

So you define a barrier ordering the commands. The tricky portion is that you have to specify the flag that describes how you intend to *USE* the data altered by CS. So in this particular example we specify that the data will be used for vertex drawing after compute.

The summary is that you need to remember that all jobs are submitted and executed on the GPU in parallel.

On The Support of OpenGL ES TM Compute

- List of devices with official [GLES 3.1](#) support
- Beware of potential driver issues
- Stay tuned for support by recognized industry benchmarks

SA2014.SIGGRAPH.ORG

SPONSORED BY



Even though the official Khronos list of devices is solid, unfortunately we don't have so large list of actual products.

No doubt we'll hear more about this in the coming months.

My experience is that latest GLES with compute shaders driver quality is far less consistent across vendors compared to even OpenCL.

It is entirely possible to run into frustrating driver bugs. From this perspective the OpenCL and GLES pixel shaders are by far more robust.

Also no official benchmarks for compute shaders in the mobile space yet, just fragmented samples, and Rightware just recently launched the Basemark but it is for GLES ES 3.0 only.

OpenGL ES™ Compute Shaders summary

Support for devices beyond GPU	No
Explicit jobs scheduling	No
Simple API	No
Rich kernel language	Yes
OS integration	Yes
Interoperability with other APIs	No
Low-level tuning	Yes
Portable across OSes	Yes

Compute Shaders basically matches the capabilities and general level of tuning available with OpenCL, except that the Shaders support GPUs only.

As we agreed on the pure technical discussion in the beginning I would leave the actual industry adoption and general marketing things aside, that is why I still consider Compute Shaders to be portable solution for GPU-only compute.

Conclusions



Overall summary					
	RenderScript*	OpenCL™	Metal*	GL ES™ Fragments Shaders	GL ES™ Compute Shaders
Support for devices beyond GPU	Yes	Yes	No	No	No
Explicit jobs scheduling	No	Yes	No	No	No
Simple API	Yes	No	No	No	No
Rich kernel language	Yes	Yes	Yes	No	Yes
OS integration	Yes	No	Yes	Yes	Yes
Interoperability with other APIs	Yes	Yes	Yes	No	No
Low-level tuning	No	Yes	Yes	No	Yes
Portable across OSes	No	Yes	No	Yes	Yes

SA2014.SIGGRAPH.ORG

SPONSORED BY



Now the final summary.

It's a lot of data you can always study offline, so here I will give you just few example on how to interpret this

- 1) Only RS and OCL support devices other than GPUs, but just OpenCL allows to explicitly schedule things, while RS relies on the automatic runtime scheduler
- 2) Only OpenCL has no first-class OS support, which is it's main downside, often limiting the support and adoption in mobile
- 3) Neither RenderScript nor Metal are fully portable across OSs, while both OpenCL and Pixel (but not the Compute) Shaders (yet) run on all the major OSs

Few Recommendations

If app is not already using OpenGL ES™, better not to use Compute Shaders

Especially since there is no universal support by devices

OpenCL™ is better for large-bad-optimized things like physics library

Best to write portable code for visual computing apps

GLES fragment shaders is the only portable/ubiquitous way today

Less ways to optimize comparing to the dedicated Compute APIs

If you do a lot of processing with CPU and just seek for a speedup

Look at RenderScript* (Metal* for iOS)

Also look into domain-specific libs like OpenCV*

“Transparent” acceleration with GPU (via OpenCL) , DSPs (via OpenVX*), CPU (via Intel IPP®)

SA2014.SIGGRAPH.ORG

SPONSORED BY



I wouldn't open a rat hole here with the discussion why specific hardware or OS vendors are supporting or not any particular API.

At least it is clear that there is no any era of consolidation ahead of us.

So solely from the technical point of view I see just few recommendations from myself (Don't take them too literally ☺):

if your app is not already using OpenGL, you are probably better off choosing another API for compute scenarios. Similarly DirectCompute is best used with the graphics pipeline on Windows*. BTW Microsoft's AMP is built on top of DirectCompute, it is the certified compute API for Windows Store Applications for windows tablet/phones.

Also even for games, OpenCL is really great for large bad things like physics engine. OpenCL is supported across Windows Desktop, Android, Linux and MacOSX (tat we didn't discuss). OpenCL* is best used for writing portable code across the 4 operating systems for visual computing applications, notice that it is extended with API extensions including interoperability with graphics and media.

For image-processing effects even regular pixel shaders may suffice, and this approach does offers one benefit over all the main approaches here- ubiquity.

If you do a lot of processing with CPU and just seek for a speedup - Look at RenderScript*

(Metal* for iOS)

Also look into domain-specific libs like OpenCV*, that offers transparent acceleration for many cases

What about Intel

Intel opens it's SoC for programmers to maximize the application performance by using the right device for the task in hand. Our vision for Heterogeneous Programming is to make the platform easier to use by exposing and extending industry standard APIs and programming models. To that extent, Intel is supporting standards like Microsoft DirectX, OpenGL, LLVM, OpenCL*, OpenCL* SPIR, and Google Renderscript. On top of these models and APIs, Intel provides programming tools and libraries that allow easier access to Heterogeneous Programming. These tools target Windows, Android and Linux. Intel tools are targeting both CPU and Intel Graphics and other processing elements on the SOC

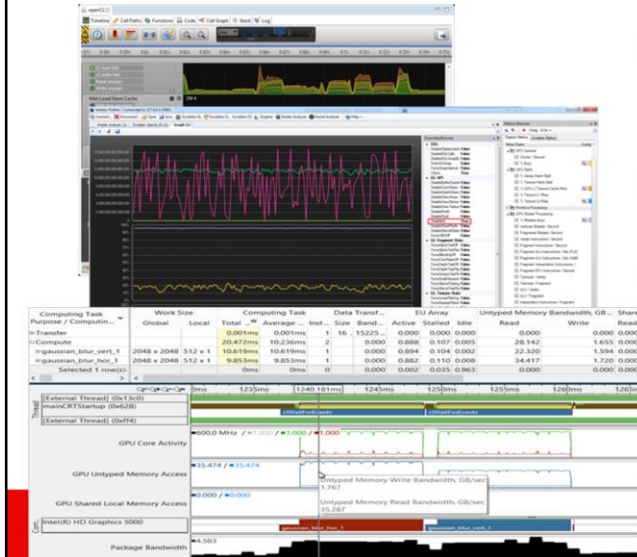
Common Performance Tips (All APIs)

SA2014.SIGGRAPH.ORG

SPONSORED BY



Few Thoughts on Tools



- ▶ OpenGL (ES)™ is first class citizen
 - ▶ But not for Compute Shaders
 - <same for RenderScript* and OpenCL™>
- ▶ API-level analysis is ~vendor agnostic
 - ▶ And many API level tricks are ~portable
- ▶ But actual HW counters are entirely vendor-specific...
 - ▶ Kernels optimizations are generally less portable as well

SPONSORED BY



All the vendors offer plethora of tools for regular GLES vertex and pixel shaders, but not for general sort of compute, for which support is still very limited.
for Compute most vendors have just API tracing and timeline, and no deep HW counters.

And while timeline (API-level) analysis is indeed pretty-vendor agnostic (recall the glTracer in Android), actual HW counters that are critical for kernels optimizations are entirely vendor-specific and require proprietary tools that are very different in look and feel. That is why let's consider common application level techniques first- these are most portable.

(You have lot's of info about various GL states, EGL stats and vertex/fragments dynamic counters like number of pixel processed or time spent in the vertex shader, but compute-related metrics are pretty scarce (e.g. GPU busy, shaders busy, and memory stats).)

Common Optimizations: API level

- ▶ Avoid extensive sync with host (CPU)
 - ▶ Favor larger kernels than sequence of short kernels and save on driver calls
- ▶ Avoid memory copies
 - ▶ Share memory between Compute and the rest of app
 - ▶ Seek for appropriate APIs
- ▶ Avoid (re-)implementing well-defined operations
 - ▶ Use higher-level API functions (like RenderScript intrinsics)
 - ▶ Seek for appropriate extensions

SA2014.SIGGRAPH.ORG

SPONSORED BY



Three things to avoid

Avoid extensive synchronizations with host code and avoid redundant memory transfers between host and device, or between APIs.

Also there is a common pitfall of implementing home-baked routine for already well-defined tasks, for example data conversions routines, or again on interoperability topics.

The entire list is inspired by OpenCL btw (so I omitted OpenCL tips in the corresponding section)

Common Optimizations: Kernel Level

- ▶ Generally images/textures are discouraged
- ▶ Prefer vector data types
 - ▶ Good for data alignment, SIMD, VLIW
- ▶ Play with workgroup size
 - ▶ Local size of 16-256 fits most GPUs
 - ▶ Always try “HW- default” local size (e.g. allowed in OpenCL™)
- ▶ Use built-in functions
- ▶ Consider fast/relaxed math

SA2014.SIGGRAPH.ORG

SPONSORED BY



Here I listed few API-agnostic optimizations for kernels:

There is general question on how to access the memory in compute: for example SSBO which are the same to buffers in OpenCL versus images and textures.

For read-only path the textures often serve better, as backed by HW texture caches . The primary difference is that images always have fixed layout. Buffers, in contrast are don't really have any element definition at all.

Another thing is that buffers always use linear memory, while images/textures usually rely on some tiling to accommodate 2D spatial coherency of access.

So there can be a large difference in performance, but most general recommendation backed by many vendors is to prefer SSBO in OpenGL and buffers in OpenCL over true images/texture.

Most vendors advocate vector data types, so follow this receipt especially when vectors are natural data types for the algorithm (like uchar4 for RGBA)

Always try NULL for local size in OpenCL (some GPUs would fail for other otherwise)
And of course try to trade off precision and speed`

Acknowledgements

- ▶ Thanks Sushma Rao and Tim Foley for the materials
- ▶ Also thank you Avi Bleiweiss, Aaron Kunze, Stephen Junkins and especially Arnon Peleg for the foils reviews



Few acknowledgements and we are done

Bonus: Optimizing OpenCL* code-path in OpenCV*

More at

https://intel.activeevents.com/sz14/connect/sessionDetail.wv?SESSION_ID=1260&tclass=popup

SA2014.SIGGRAPH.ORG

SPONSORED BY



Maximizing Occupancy



- ✦ Occupancy is a measure of utilization
- ✦ Two primary things to consider:
 - ▶ Launch enough work items to keep GPU units busy
 - ▶ In each work item: using short vector data types and computing multiple pixels For example, color conversion:

```
__global uchar* src, dst;  
p = src[src_idx] * B2Y +  
  src[src_idx + 1] * G2Y +  
  src[src_idx + 2] * R2Y;  
dst[dst_idx] = p;
```

Before:
One pixel per work item

```
__global uchar* src_ptr, dst_ptr;  
uchar16 src = vload16(0, src_ptr);  
uchar4 c0 = src.s048c;  
uchar4 c1 = src.s159d;  
uchar4 c2 = src.s26ae;  
uchar4 Y = c0 * B2Y +  
          c1 * G2Y +  
          c2 * R2Y;  
vstore4(Y, 0, dst_ptr);
```

After:
Four pixels per work item

Copyright © 2014, Intel Corporation. All rights reserved. *Other names and brands may be claimed as the property of others.



Optimizing Host to Device Transfers



- ✦ Host (CPU) and Device (GPU) share the same physical memory
- ✦ For OpenCL* buffers:
 - ▶ Create buffer with system memory pointer and `CL_MEM_USE_HOST_PTR`
 - ▶ No transfer needed (zero copy)!
 - ▶ Allocate system memory aligned to a page (4096 bytes)
 - ▶ Allocate a multiple of cache line size (64 bytes)
 - ▶ Use `clEnqueueMapBuffer ()` to access data
 - ▶ OpenCV 3.0 changes make excellent use of this feature!

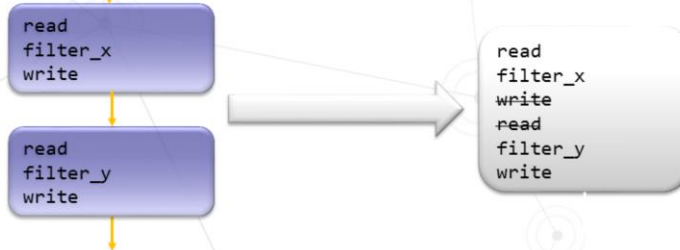
Copyright © 2014, Intel Corporation. All rights reserved. *Other names and brands may be claimed as the property of others.



Optimizing Memory Accesses



- ✦ Merging kernels reduces memory traffic
 - ▶ Computer vision algorithms often form pipelines
 - ▶ Merging multiple kernels in a pipeline can reduce trips to memory
 - + Also reduces runtime overhead!



- ▶ But mind instruction cache size
- ▶ Optimization used for OpenCV[®] separable filters

Copyright © 2014, Intel Corporation. All rights reserved. *Other names and brands may be claimed as the property of others.



Maximizing Compute Performance



- ✦ Use native transcendental functions where possible
- ✦ Use `mad()` / `fma()` (or use `-cl-mad-enable`)
- ✦ Use `floats` instead of `ints` wherever possible to maximize co-issue
- ✦ Avoid `long` and `size_t` data types
 - ▶ Prefer `float` over `int`, if possible
 - ▶ Using `short` data types may improve performance
- ✦ Trade accuracy for speed, where appropriate
 - ▶ "native" built-ins, `-cl-fast-relaxed-math`

```
x = cos(i);
```

```
x = native_cos(i);
```

Used to speedup OpenCV SURF and HOG!

Copyright © 2014, Intel Corporation. All rights reserved. *Other names and brands may be claimed as the property of others.



Legal Disclaimer and Optimization Notice

- ▶ INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.
- ▶ Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
- ▶ © 2014, Intel Corporation. All rights reserved. Intel, the Intel logo, Atom, Core, Iris, VTune, Xeon, and Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804



Backup

SA2014.SIGGRAPH.ORG

SPONSORED BY



Speaker Bio



maxim.y.shevtsov@intel.com

Master Degree in Computer Science in 2003

Academia studies in computer graphics prior to joining Intel in 2005

At Intel:

- Research Scientist: parallel algos for Ray-Tracing/acceleration structures
- Software Architect in the OpenCL performance team



RenderScript*: Advanced parallelism

```
//test.rs
rs_allocation gIn;
rs_allocation gOut;
rs_script gScript;
rs_script_call adv;
void run()
{
  rsForEach(gScript, gIn, gOut, ...&adv);
}
//on the host
mScript.set_gScript(someScript);
mScript.set_gIn(ain);
mScript.set_gOut(aout);
mScript.invoke_run();
```

By default, `forEach` in RS executes the kernel for each individual item in the allocation

Use `rs_script_call` to override this on the managed side

on the host you may use `forEach` ver with `Script.LaunchOptions`, *limited*

SA2014.SIGGRAPH.ORG

SPONSORED BY



- 1) In this kernel example the `run()` function is called from the device side code. It simply triggers an actual compute launch.
- 2) The first parameter is the script to be launched - the root function of this script will be invoked for each element in the allocation. The second and third parameters are the input and output data allocations.
- 3) This allows dynamic parallelism and defining pretty advanced logic to run on the device side without any host intervention.

Always keep in mind that by default, `forEach` in RS is executing the kernel for each individual item in the allocation (i.e. pixel)...

RenderScript*: Advanced parallelism, 2

rs_script_call overrides default strategy on the managed side

```
//test.rs
rs_script_call adv;
void run()
{
    adv.xStart = 16;//some offset
    adv.xEnd =rsAllocationGetDimX(gIn);
    rsForEach(gScript, gIn, gOut,...&adv);
}

typedef struct rs_script_call{
    enum rs_for_each_strategy strategy;
    uint32_t xStart;
    uint32_t xEnd;
    uint32_t yStart;
    uint32_t yEnd;
    uint32_t zStart;
    uint32_t zEnd;
    uint32_t arrayStart;
    uint32_t arrayEnd;
}

enum rs_for_each_strategy {
    RS_FOR_EACH_STRATEGY_SERIAL,
    RS_FOR_EACH_STRATEGY_DONT_CARE,
    RS_FOR_EACH_STRATEGY_DST_LINEAR,
    RS_FOR_EACH_STRATEGY_TILE_SMALL,
    RS_FOR_EACH_STRATEGY_TILE_MEDIUM,
    RS_FOR_EACH_STRATEGY_TILE_LARGE
}
```

SA2014.SIGGRAPH.ORG

SPONSORED BY



- 1) Similarly to the prev foil we have the function that actually triggers compute launch. Here we use additional parameters to change the launch options for the script
- 2) this is also possible from host APIs
- 3) There is always hints to runtime on the efficient access pattern

...But if you want code working on a single row or column instead of an element, you would need to use fake allocation and to define the actual input/output allocations as script globals instead.

OpenCL™ portability: OpenCL 1.1->1.2

- ▶ Custom devices and “built-in” kernels (for HW-assisted impl)
- ▶ New image types: 1D images, 1D/2D image arrays
- ▶ Sampler-less image read functions (with default sampling modes)
- ▶ `printf` function (beware of output buffer limitations)

SA2014.SIGGRAPH.ORG

SPONSORED BY



One notable feature of OCL 1.2 is support for custom device that typically warp some fixed function HW, plus built-in kernels that are similar to RS intrinsics, for example intel offers VME this way. Alos notice improved images (that are OpenCL parlance for textures) support

OpenCL™ portability: OpenCL 1.2->2.0

- ▶ Shared Virtual Memory
- ▶ C1x atomics with scopes and orders, new fences, atomic data types
- ▶ Enqueueing kernels from kernels: entirely on the device
- ▶ Pipes: write/read endpoints for producer-consumer kernels
- ▶ Support for depth and sRGB images
- ▶ OpenCL images from multi-sampled/mip-mapped OpenGL textures

SA2014.SIGGRAPH.ORG

SPONSORED BY



Few words on OCL 2.0

SVM- ability to use regular app-allocated mem in the kernels, the next step in simplifying the porting to OpenCL, as you can keep pointers-based structures like linked list the same on the CPU and GPU.

SVM is also about granularity and coherences of sharing between host and OCL devices, hence atomics in this list

Kernels from kernels for dynamic parallelism entirely on a device and Pipes: (ordered data flow between *kernels*, write/read endpoints for producer-consumer) – are both new execution features.

And finally the improved image (which is OpenCL lingo for textures)...

Also:

Program scope variables in global address space

Generic address space (no need to have local*/global* versions for user funcs in kernels)

Clang blocks (functions that are “local” to other functions, accessing the stack variables)

3D image writes are a core feature

Support for 2D image from buffer

Images with the read_write qualifier

OpenCL™: Setting up the environment

- ▶ Select and grab a platform

```
err = clGetPlatformIDs(1, &platformId, &numPlatforms);
```

- ▶ Select a device the platform provides

```
err = clGetDeviceIDs(platformId, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
```

- ▶ Create a context (can support multiple OR-ed devices)

```
context = clCreateContext(platformId, 1, &device_id, NULL, NULL, &err);
```

- ▶ Create a command queue to feed the device

```
Queue = clCreateCommandQueue(context, device_id, 0, &err);
```

OpenCL™: Create and Build the Kernels

- ▶ Define source code for the kernel program as a string literal or read from a file
- ▶ Build the program object and Compile the program

```
program = clCreateProgramWithSource(context, 1 &KernelSource, NULL, &err);  
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

- ▶ Fetch and print error messages

```
size_t len; char buffer[2048];  
            (program, device_id, CL_PROGRAM_BUILD_LOG,  
             sizeof(buffer), buffer, &len);  
printf("%s\n", buffer);
```

- ▶ Kernels can also be created from binaries

OpenCL™: Setup Memory Objects

- ▶ For the Vector addition example, we need 3 memory objects: 2 input buffers (A and B) and one output buffer (D).

```
A_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,  
                           sizeof(float) * count, A_data, NULL);  
  
B_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,  
                           sizeof(float) * count, B_data, NULL);  
  
D_buffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * count,  
                           NULL, NULL);
```

OpenCL™: Define the kernel

- ▶ Create kernel object from kernel function “vadd”

```
kernel = clCreateKernel(program, "vadd", &err);
```

- ▶ Attach arguments to kernel object

```
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &A_buffer);  
err = clSetKernelArg(kernel, 1, sizeof(cl_mem), &B_buffer);  
err = clSetKernelArg(kernel, 2, sizeof(cl_mem), &D_buffer);  
err = clSetKernelArg(kernel, 3, sizeof(cl_float), &constant);
```

OpenCL™: Submit commands / Execute

- ▶ Enqueue the kernel for execution

```
size_t range[3] = {count, 1, 1}; // Define global size of execution
err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, range, NULL, 0, NULL, NULL);
```

- ▶ Read back results

```
err = clEnqueueReadBuffer(queue, D_buffer, CL_TRUE, 0, sizeof(float) * count, D_data,
0, NULL, NULL);
```

- ▶ This API call is blocking. It returns only after the pipeline (queue) is complete, beyond the read operation.

OpenCL™ primer on multi-device support

Multi-device support is leveraged thru the [shared contexts](#)

- ▶ Objects are shared between devices in the context
 - ▶ Objects cannot be updated concurrently—do explicit synch on writes
 - ▶ OpenCL 1.2 features sub-buffers for writes to the non-overlapping regions
- ▶ Still, a separate queue per device: no “shared queue”
 - ▶ Any load-balancing logic left to your app
 - ▶ Govern work allocation by current load and speed of devices
 - ▶ Devices perf can be affected by OS or driver dynamic freq scaling

SA2014.SIGGRAPH.ORG

SPONSORED BY



There are many approaches to handling multiple devices in OpenCL, well beyond general a context per device way.

You can make multiple devices to share the same context, and most OpenCL objects will be shared between devices as well.

As usual there are many caveats here, for example the memory objs must be explicitly written to a device before being used. And if you transfer between devices, an intermediate host copy may be required (in OCL 1.2 use **clEnqueueMigrateMemObjects**)

Shared context does not imply any “shared queue”. The OpenCL specification requires you to create a separate queue per device and there a lot of associated complexity to orchestrate the work. It is worth to mention that all OpenCL API calls are thread-safe (except `clSetKernelArg`) starting OCL 1.2

Also keep a kernel source same for the devices, use preprocessor to accommodate CPU or GPU specifics.