Propel with OpenCL

A Deep Dive Workshop to Create, Debug, Analyze and Optimize OpenCL Applications using Intel® Tools

Anita Banerjee, Julia Fedorova, Uri Levy, Alexandr Kurylev,
Sonal Sharma, Michael Stoner, Robert Ioffe

Legal

Agenda

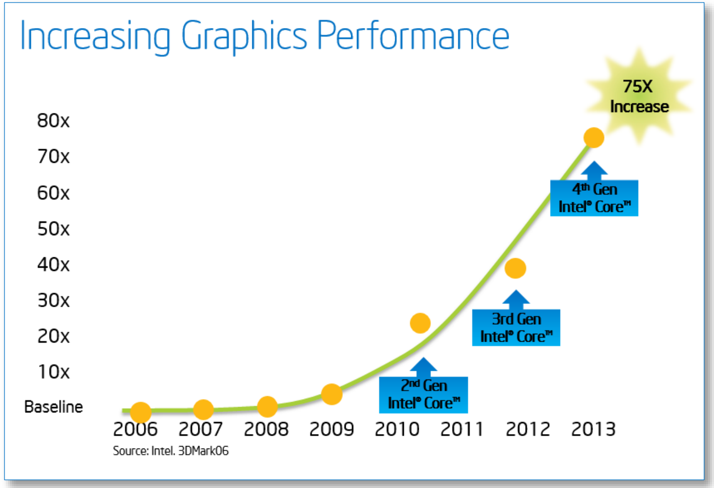
- Intel® Graphics Overview
- The Intel® OpenCL™ Code Builder
- Intel® VTune™ Amplifier 2015
- Optimization Techniques and Examples
- OpenCL™ 2.0 Overview
- Summary / Questions

Agenda

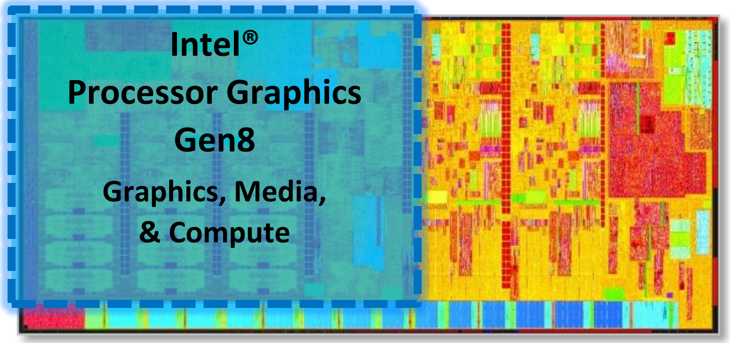
- Intel® Graphics Overview ---- ***Presenter: Julia Fedorova***
- Intel® OpenCL™ Code Builder
- Intel® VTune™ Amplifier 2015
- Optimization Techniques and Examples
- OpenCL™ 2.0 Overview
- Summary / Questions

Intel® Processor Graphics?

- Intel® Processor Graphics:
  3D Rendering, Media, *and Compute*
- Discrete class performance but…
  integrated on-die for true heterogeneous computing, SoC power efficiency, and a fully connected system architecture
- Some products are near TFLOPS performance
- Highly threaded, data parallel compute engine

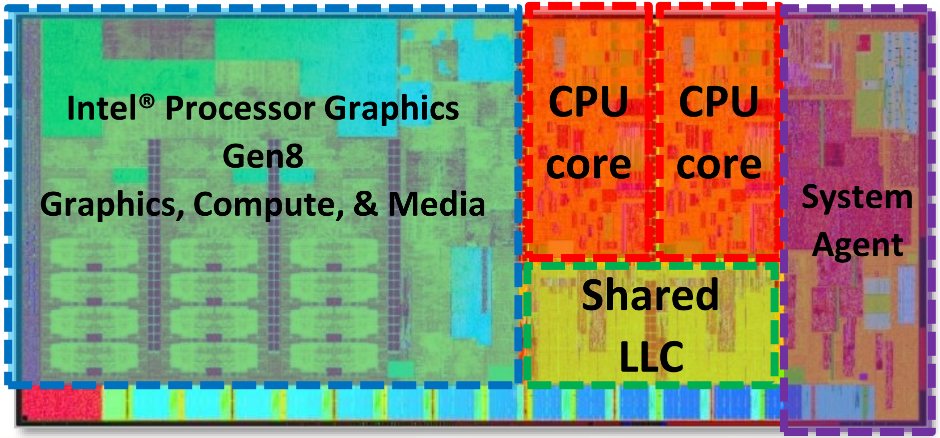


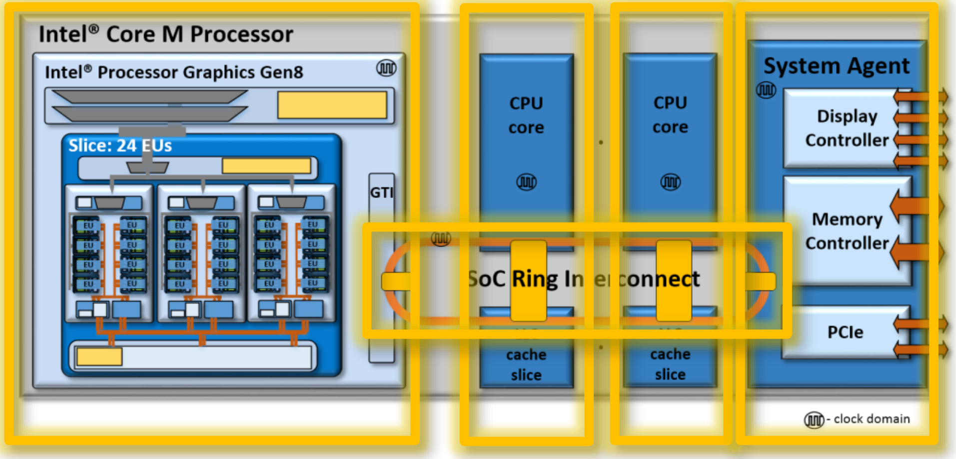
Intel® Core™ M:



Intel Processor Graphics is a key Compute Resource

Example Chip Level Architecture: Intel® Core™ M



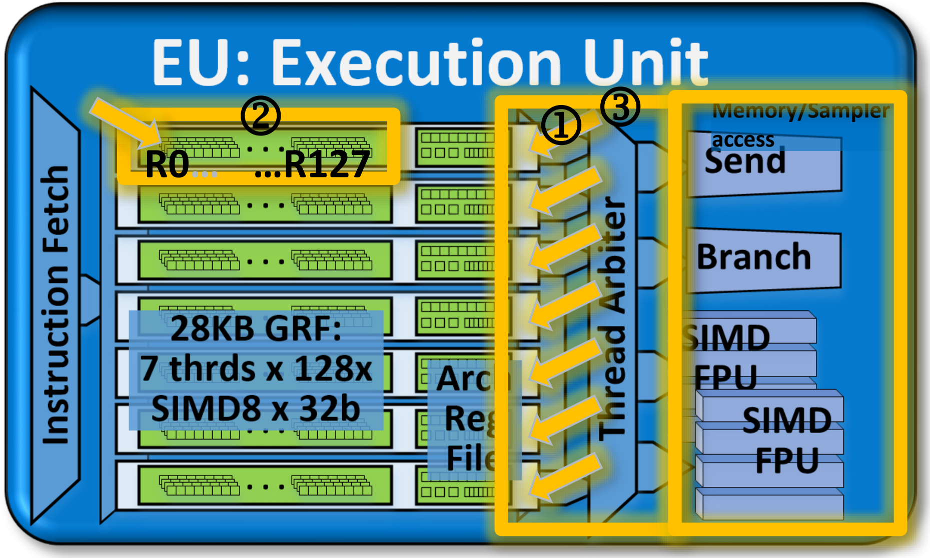


Many different processor products, with different processor graphics configs

Multiple CPU cores, shared LLC, system agent

Multiple clock domains, target power where it's needed

EU: The Execution Unit



① Gen8: Seven hardware threads per EU

② 128 general purpose registers per thread

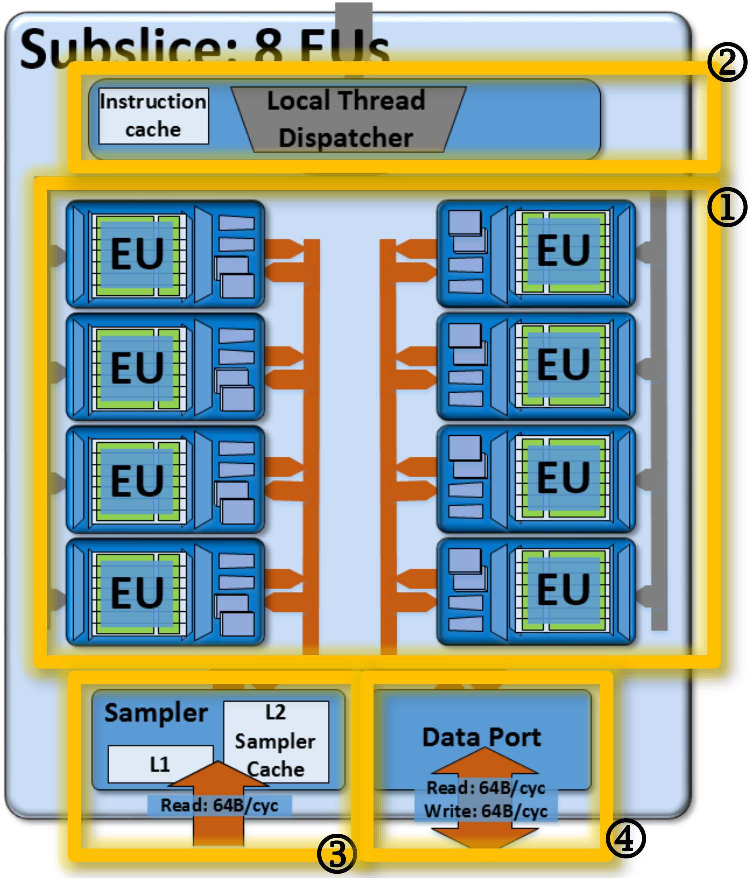
- 4K registers/thread or 28K/EU
- Each register : 32 bytes wide
  - 8 x 32b floats, 8 x 32b integers
  - 16 x 16b half-floats, 16 x 16b shorts

③ Thread Arbiter picks instructions to run from runnable thread(s)

- Each cycle: can co-issue multiple instructions, from up to four different threads
- Dispatches instruction to appropriate functional unit

The fine grain threaded nature of the EUs ensures continuous streams of ready to execute instructions, while also enabling latency hiding of longer operations such as memory scatter/gather, sampler requests, or other system communication.

Subslice: An Array of 8 EU's



Each: Subslice

① Eight Execution Units

② Local Thread Dispatcher & Inst $

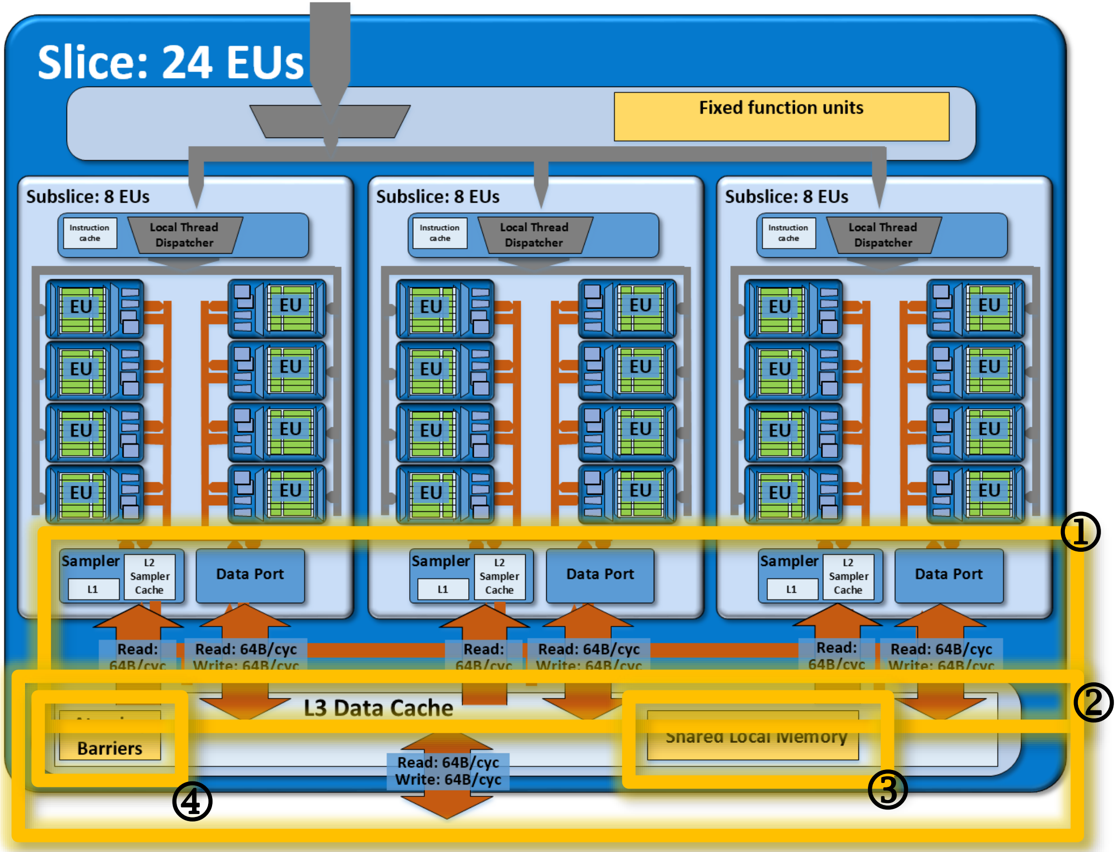
③ Texture/Image Sampler Unit:

- Includes dedicated L1 & L2 caches
- Dedicated logic for dynamic texture decompression, texel filtering, texel addressing modes
- 64 Bytes/cycle read bandwidth

④ Data Port:

- General purpose load/store memory unit
- Memory request coalescence
- 64 Bytes/cycle read & write bandwidth

Slice: 3x Subslices



Each Slice: 3 x 8 = 24 EU's

- 3 x 8 x 7 = 168 HW threads/slice

① Dedicated interface for every sampler & data port

② Level-3 (L3) Data Cache:

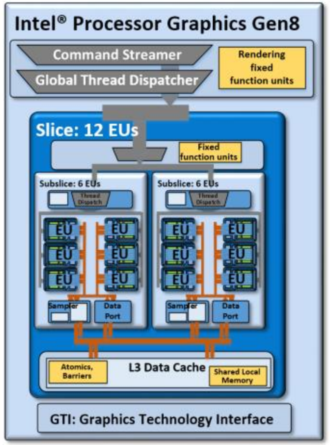
- Typically 384 KB / slice, though Allocations are app reconfigurable
- 64 byte cachelines
- Monolithic, but distributed cache
- 64 bytes/cycle read & write

③ Shared Local Memory:

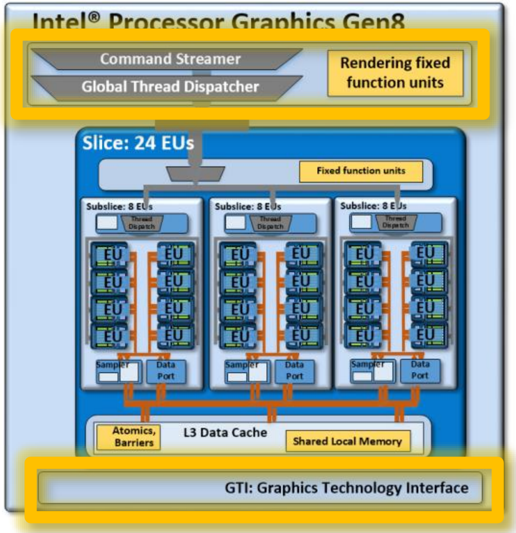
- 64 KB / subslice
- More highly banked than rest of L3

④ Hardware Barriers, 32bit atomics

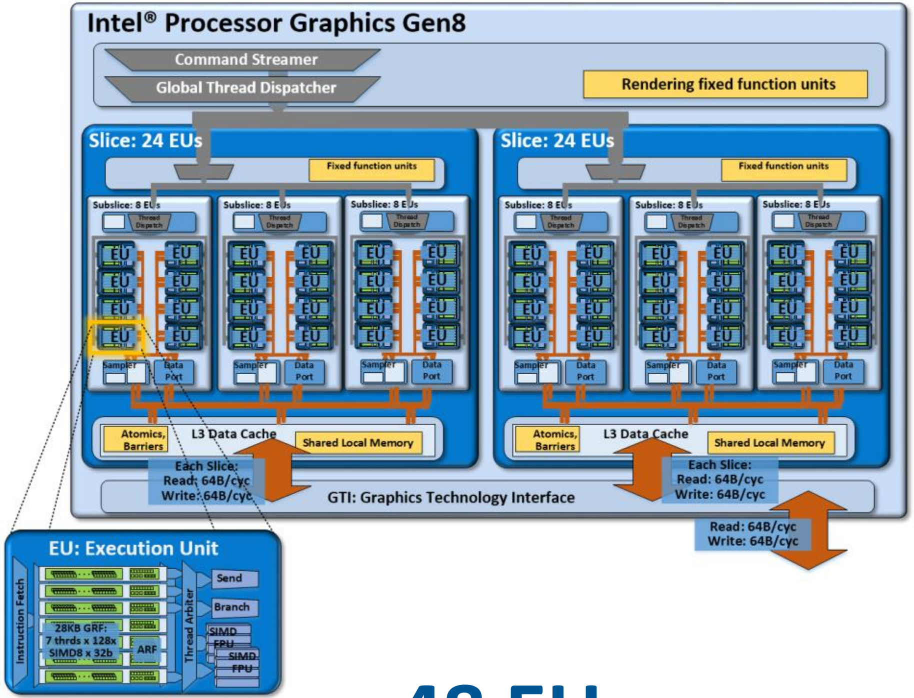
Product Configuration Examples



12 EUs



24 EUs



48 EUs

OpenCL™ Execution Model maps to Intel® Graphics Architecture

OpenCL* Kernels run on an Execution Unit (EU)

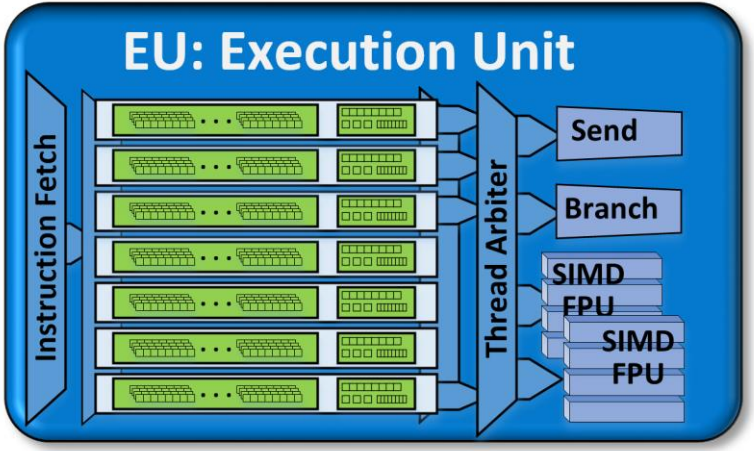
Each EU is a Multi-Threaded SIMD Processor

Up to 7 threads per EU

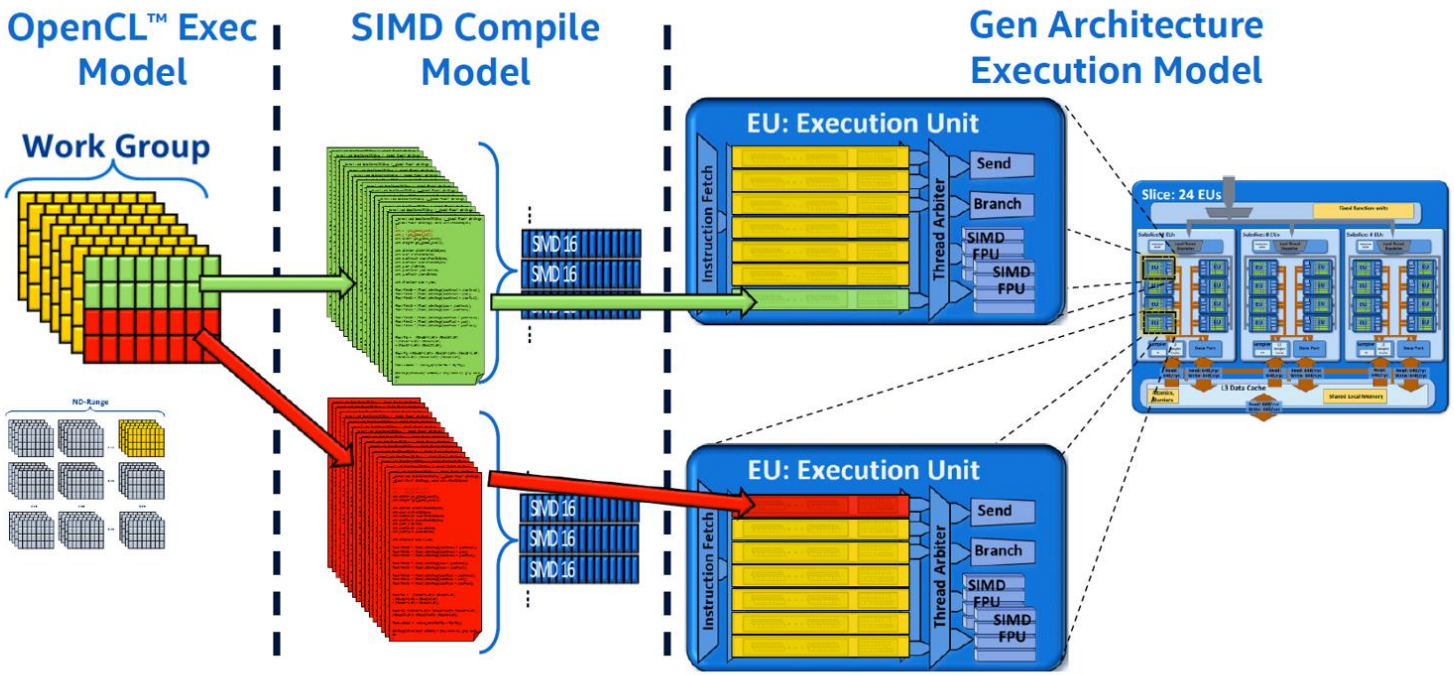
- 128 x 8 x 32-bit registers per thread

Up to 8, 16, or 32 OpenCL* work items per thread (compiler-controlled)

- “SIMD8”, “SIMD16”, “SIMD32”
- SIMD8 → More Registers
- SIMD16 and SIMD32 → Better Efficiency



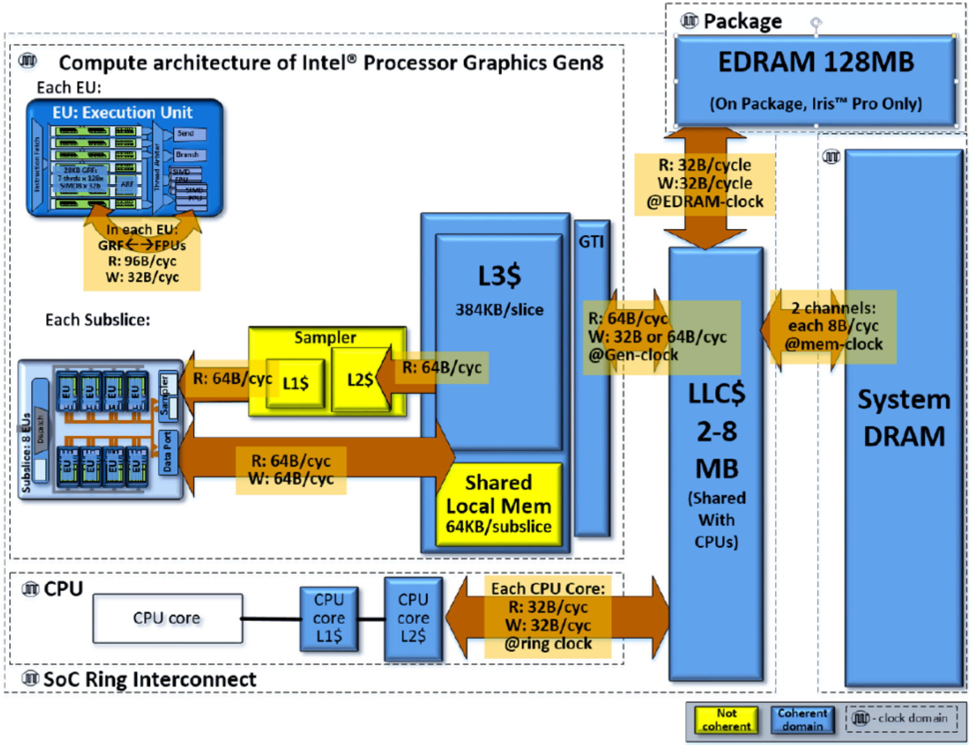
OpenCL™ Execution Model maps to Intel® Graphics Architecture



Memory Hierarchy and Sharing

- Intel® Processor Graphics has full performance access to system memory
- “Zero Copy” CPU & Graphics data sharing
- Shared Virtual Memory – new in Gen8

  Facilitated by OpenCL™ 2.0 Shared Virtual Memory:
  - Coarse & fine grained SVM
  - CPU & GPU atomics as synchronization primitives
  - System SVM as soon as OSVs are ready



Agenda

- Intel® Iris™ Graphics Overview
- The Intel® OpenCL™ Code Builder ---- ***Presenter: Uri Levy***
- Intel® VTune™ Amplifier 2015
- Optimization Techniques and Examples
- OpenCL™ 2.0 Overview
- Summary / Questions

OpenCL™ Code Builder

A comprehensive developers' tool-chain for OpenCL™ and Intel® Graphics compute

Supports Each State of the OpenCL™ Code Development

Getting Started

Identify opportunities for accelerations

- Code Sample
- Case Studies
- Identify performance bottlenecks in applications
- Redesign algorithms

* This stage is largely manual

Build

Quickly build OpenCL code

- Kernel development framework
- OpenCL 2.0 development environment
- Pre-defined projects
- IDE integration – auto compilation, syntax highlighting
- Offline compilation with errors reports

Debug

Debug your code – both functional and performance

- API level debugging
- API calls and commands tracing and analysis
- Step-by-step kernel debugging (preview)
- Kernel statistics views
- Memory debug

Tune

Tune the application

- Application level profiling
- Platform level timeline view
- Platform performance counters
- Correlate CPU & GPU activities
- Kernel hot-spots analysis

Carry-on performance optimizations in each step of the development

OpenCL™ Code Builder

What Is New in Version 2015?

OpenCL™ Code Builder fully integrated in Intel's development suites

- Available with Intel® Integrated Native Developer Experience (Intel® INDE)
- Available with Intel® Media Server Studio
- Advanced new editing and debugging features with Microsoft* Visual Studio plug-in
- Free Windows & Android development with Intel INDE starter edition

Commercial OpenCL 1.2 Linux* driver for Intel® Graphics

- Available with Intel® Media Server Studio

OpenCL 2.0 on 5th Generation Intel® Core™ Processors

- Fine-grained shared virtual memory (SVM) support
- Support Intel® HD Graphics 5500/6000 and Intel® Iris™ Graphics 6100
- Available with Intel® Integrated Native Developer Experience (Intel® INDE)

Where did The Intel® SDK for OpenCL™ Applications go?

- Intel® SDK for OpenCL™ Applications is now available as OpenCL™ Code Builder
- All SDK's capabilities are now integrated into Intel's suites for developers through the OpenCL™ Code Builder.
- With new suite support, developer now gets OpenCL Code Builder, profiling features, and interoperable products in a single place.
- A standalone Intel® Code Builder for OpenCL™ API is available for support configurations that are not available with the integrated suites
    - E.g Ubuntu* for CPU, Intel® Xeon Phi™ coprocessor, and more

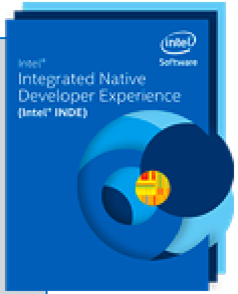
Intel's Portfolio of Tools for OpenCL™ Development

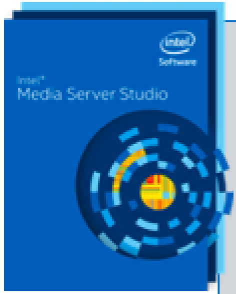
Maximize the power of the platform with OpenCL™ and Intel® Graphics Compute

- *Build high-performance applications*
- *Optimizing tasks with standard APIs and best available compute engines*
- *Tap into a comprehensive developers' tool-chain, IDE integration, and more.*

Intel® INDE

- For mobile and PC client applications
- Create & Debug with OpenCL™ Code Builder
- Analyze with INDE Analyze capabilities
- Supports:
  - OpenCL 1.2 & 2.0
  - Windows*, Android*
  - Intel® Core™ & Atom™
  - Intel® Graphics Compute





Intel® Media Server Studio

- For enterprise media solution
- Create & Debug with OpenCL™ Code Builder
- Analyze with Intel® VTune™ Amplifier XE
- Supports:
  - OpenCL 1.2 & 2.0
  - Linux*, Windows*
  - Intel® Xeon® E3 & Core™ i7
  - Intel® Graphics Compute

Intel continues to support Intel Xeon® E5 & E7, and Intel Xeon Phi through a standalone
Intel® Code Builder for OpenCL™ API

Don't leave performance on the platform!

Which Suite To Download?

	Tool suite	Intel® INDE Starter edition: free	Intel® Media Server Studio Essential edition: $499	Intel® Code Builder for OpenCL™ API for Linux* free
Supported devices	Intel® Graphics (GPU)	X	X	
	Intel® processors (CPU)	X	X	X
	Intel® Xeon Phi™ coprocessors			X
Target OS	Windows*	X	X	
	Android*	X		
	Linux*		X	X
Host OS (Development environment)	Windows*	X	X	
	Android*			
	Linux*		X	X
IDE Integration	Microsoft Visual Studio*	X	X	
	Eclipse*	X	X	
	Standalone UI	X	X	X

Select the tool that best fit your target applications and OS matrix:

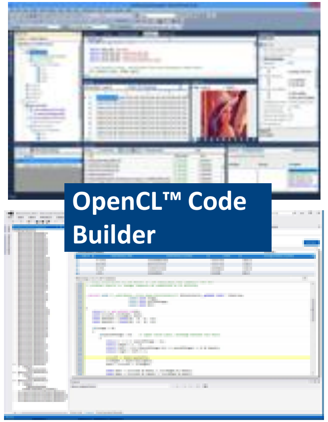
- PC & Mobile applications → Use Intel® INDE
- Enterprise media applications → Use Intel® Media Server Studio
- HPC apps → Use Intel® Code Builder for OpenCL™ APIs

Tools for OpenCL™ Development

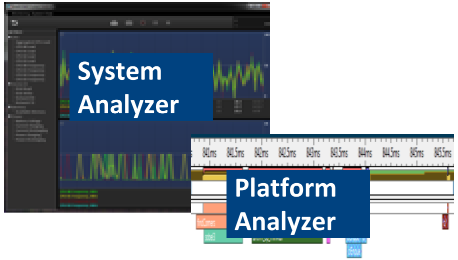
The Development Flow

Availability	Intel® INDE Starter	Intel® INDE Pro	Intel® Media Server Studio\	Intel® VTune Amplifier XE
OpenCL™ Code Builder	V	V	V	
System And Platform Analyzer		V		
Intel® VTune Amplifier XE (with Platform Analyzer features)			V	V

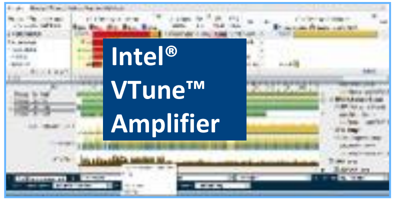
When I
CREATE
my OpenCL Code



When I first **ANALYSE**
my Application



When I
TUNE
my entire system



Don't leave performance on the table!

Get optimized code faster - use performance tools during each step of the development

OpenCL™ Code Builder

Development Tools Features

OpenCL™ Code Builder - Features and Support Matrix

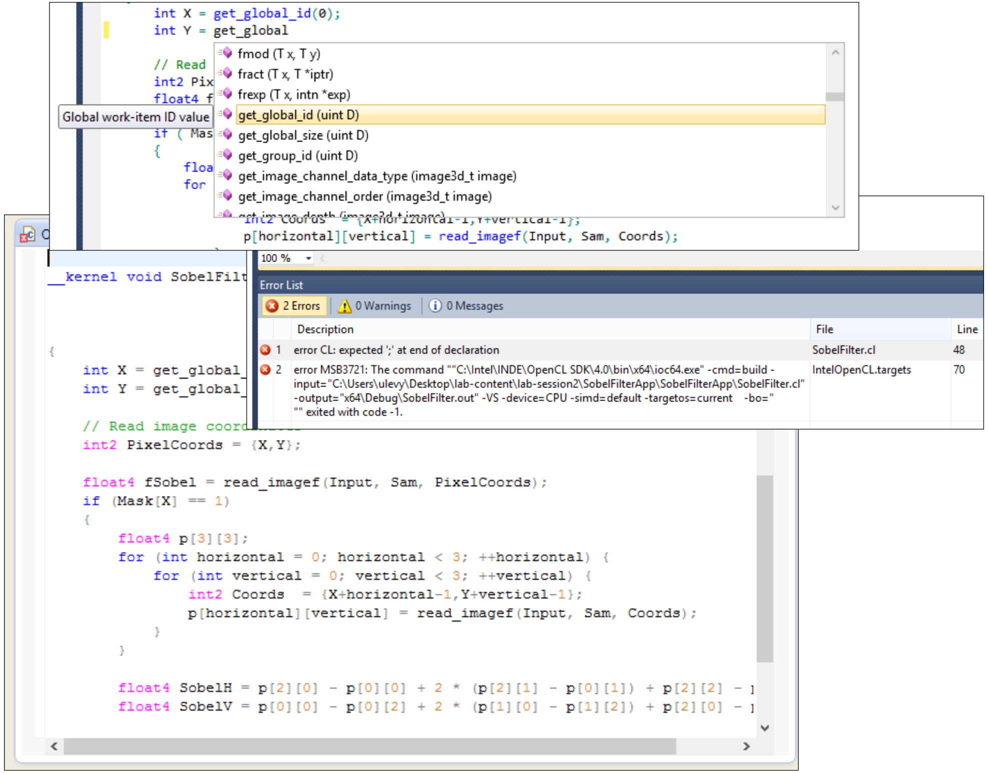
Development Environment

VISUAL STUDIO*	ECLIPSE*	STANDALONE	FEATURES:	PREVIEW FEATURE	
•	•	•	OpenCL™ 1.2 support		Create & Build
•		•	OpenCL™ 2.0 support with Intel® Core™ M and 5th Gen Intel® Core™ Processors		
•		•	OpenCL™ 2.0 development environment on previous CPU generations		
•	•	•	Kernel Development Framework		
•			New OpenCL Project wizard		
•	•	•	Syntax highlighting		
•	•	•	Code auto completion		
•	•	•	Offline compilation		
•	•	•	SPIR* 1.2 generation and consumption		
•			Remote development for Android*		
•			API-level debugging		Debug
•			Image and memory view		
•			API calls tracing		
•			Step-by-step debugging for CPU kernels		
		•	Step-by-step debugging for GPU kernels	•	
•		•	API calls and memory command analysis	•	Analyze
•			Kernel occupancy and latency analysis	•	

Code Builder BUILD

Code Editing and Compiling

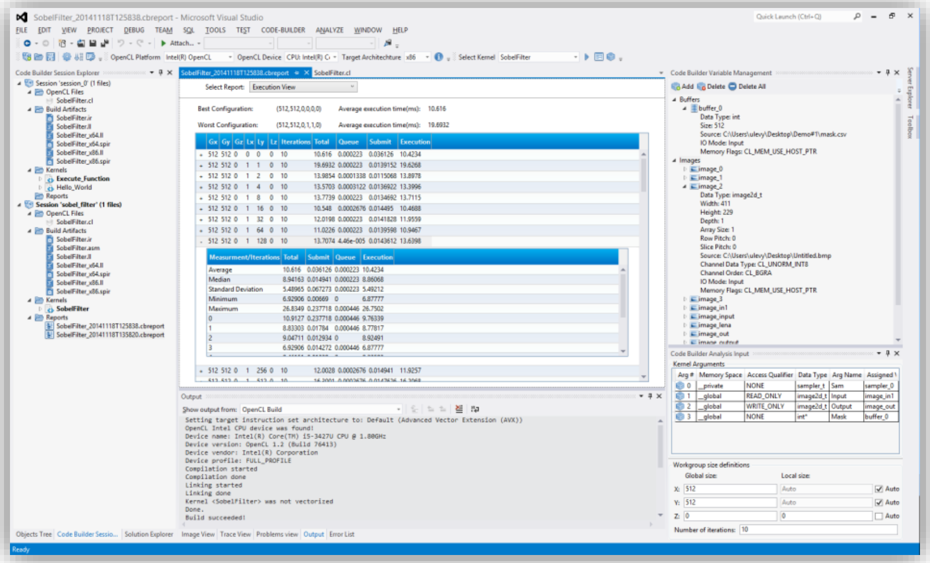
- IDE integration (Visual Studio and Eclipse)
- Offline compilation and binary generation of OpenCL™ kernels
- Syntax checking and compile error reports.
- Project wizards
- Offline build for the Android* target.



Code Builder BUILD

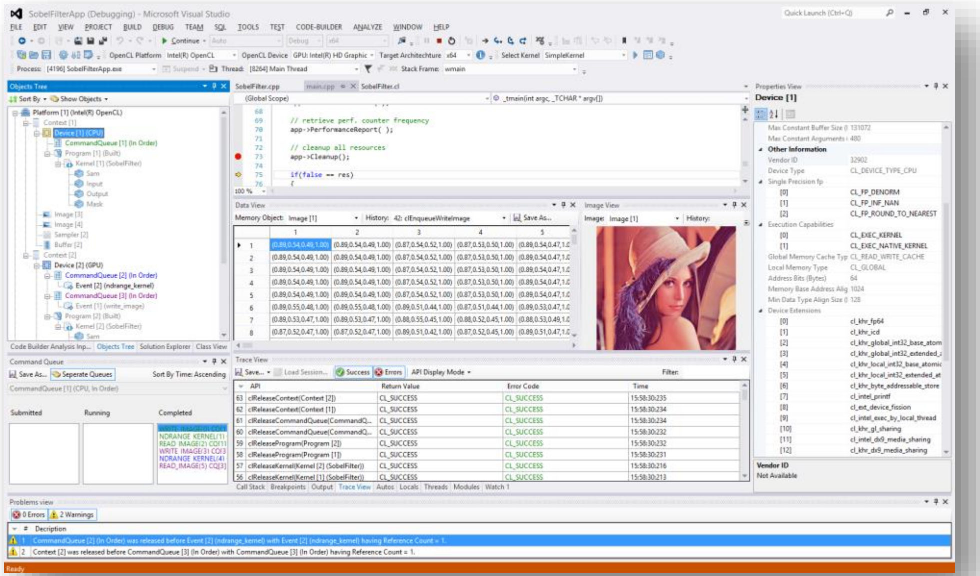
Kernel Development Framework

- Create and build OpenCL Kernels on standalone environment
- Assign variables to the kernel and check its correctness
- Show the input and output values
- Analyze kernel’s performance with “What-if” analysis on work group sizes
- Remote development on Android devices



API Level Debugging

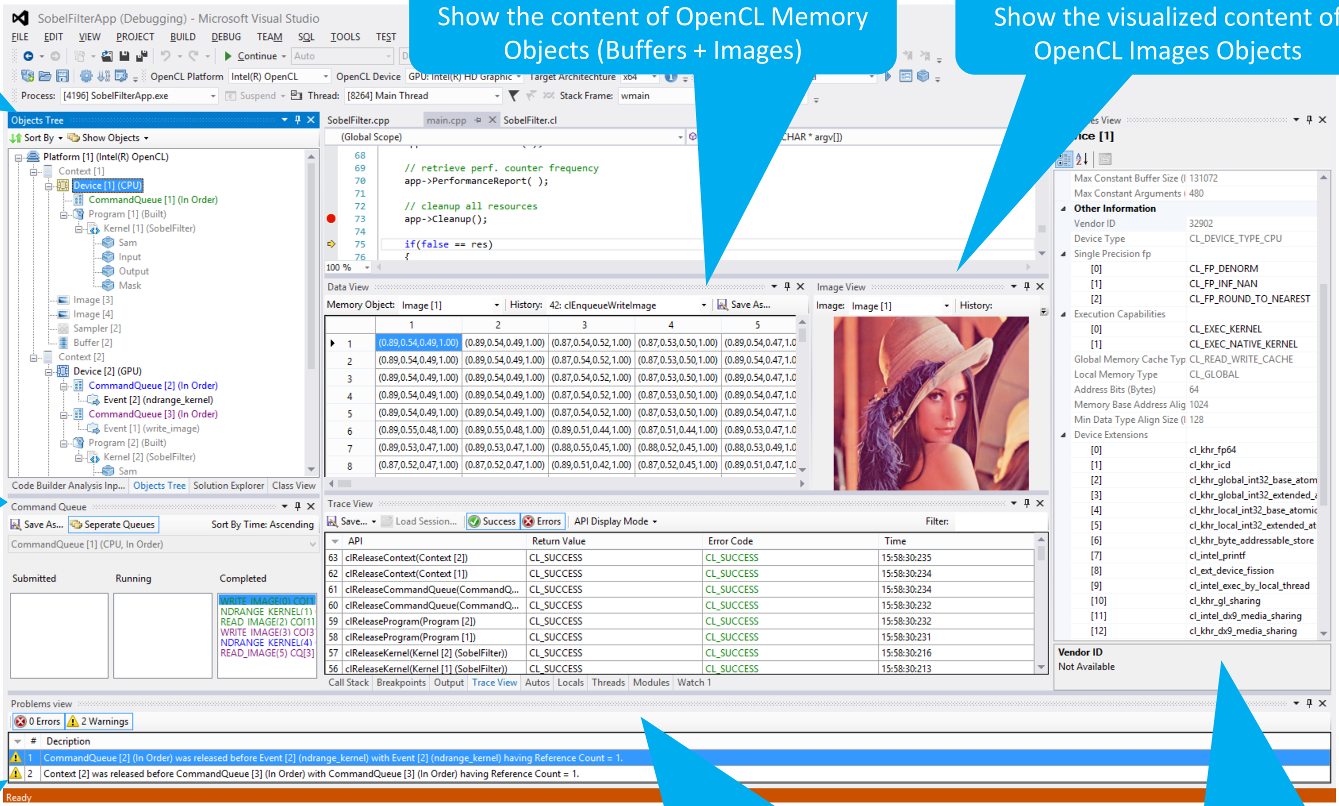
- Seamless debugging of OpenCL™ API calls, objects, and queues
- Enables monitoring and understanding the OpenCL environment of an application throughout execution
- OpenCL API calls tracing
- Images and memory objects view



Objects Tree View
Explore all OpenCL Objects in memory and their properties

Date View
Show the content of OpenCL Memory Objects (Buffers + Images)

Image View
Show the visualized content of OpenCL Images Objects



Commands Queue View
Examine commands queue status and their commands' state

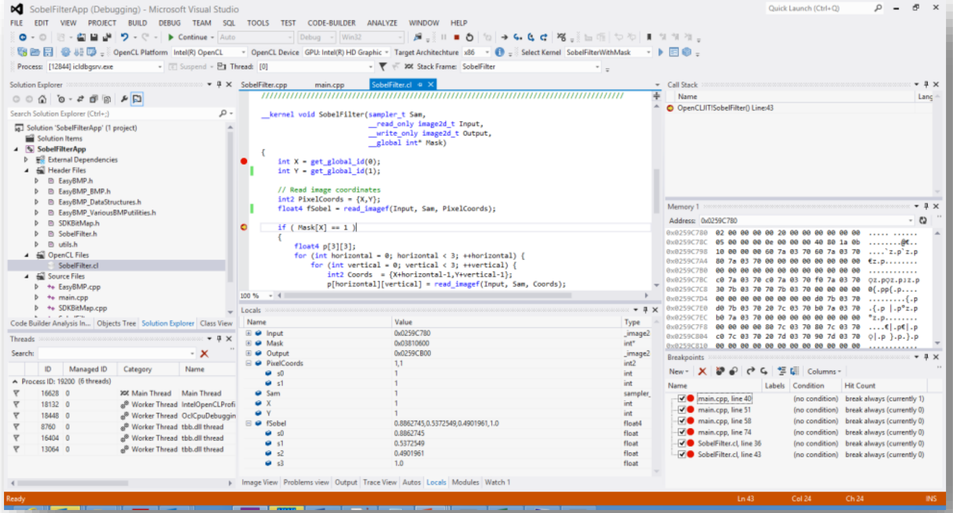
Problems View
Look for hints for potential error or warnings during execution

Trace View
Trace application's OpenCL API calls and their return values

Properties View
View the properties of the slected OpenCL objects

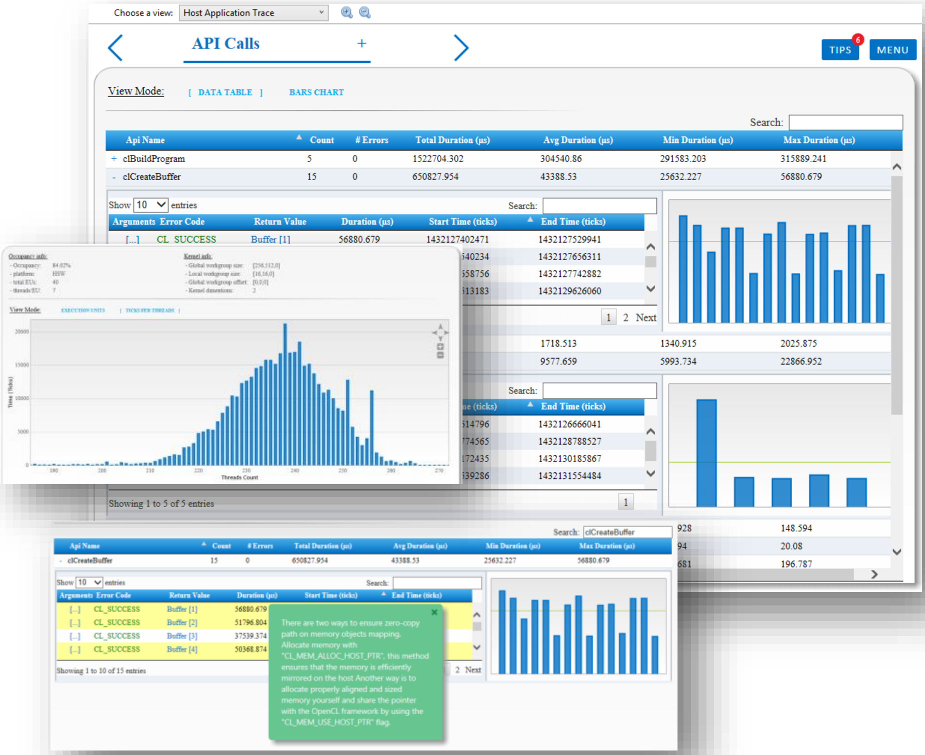
Kernel Level Debugging on the CPU

- Step into Kernels running on the CPU
- Supports existing debugging capabilities
  - Breakpoints
  - Memory view
  - Watch variables – including OpenCL types like float4, int4, etc.
  - Call stack
  - Auto and local variables views



Code Builder – Application Analysis

- Host level analysis
  - Identify performance bottlenecks in the API calls
  - Optimize the host code to reduce API execution time and kernels run time
- Kernel level analysis
  - Optimize the kernel code to get better utilization and reduce the latency
  - Measure compute metrics (latency, and utilization)



Demo Session

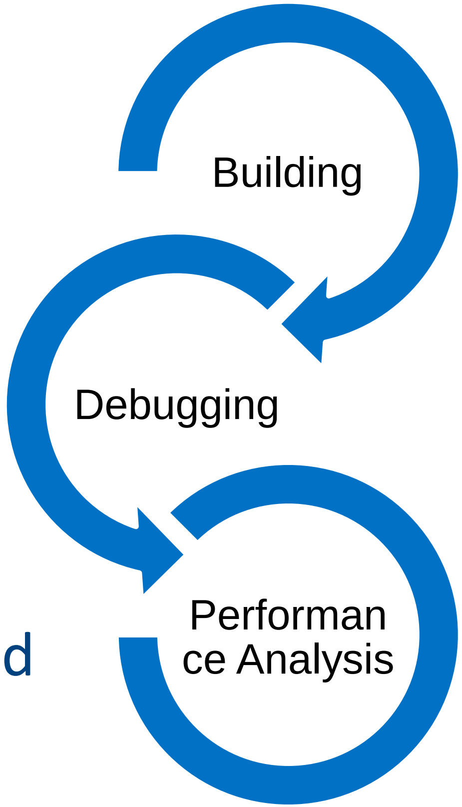
Introduction with OpenCL™ Code Builder
A Walkthrough

Code Builder Walkthrough Session Content

Create, Build and Analyze my OpenCL kernel with **Kernel Development Framework**

Debug my OpenCL host application and Kernel with **OpenCL® Debugger**

Analyze and Optimize your OpenCL application and kernel code with **OpenCL® Code Analyzer**

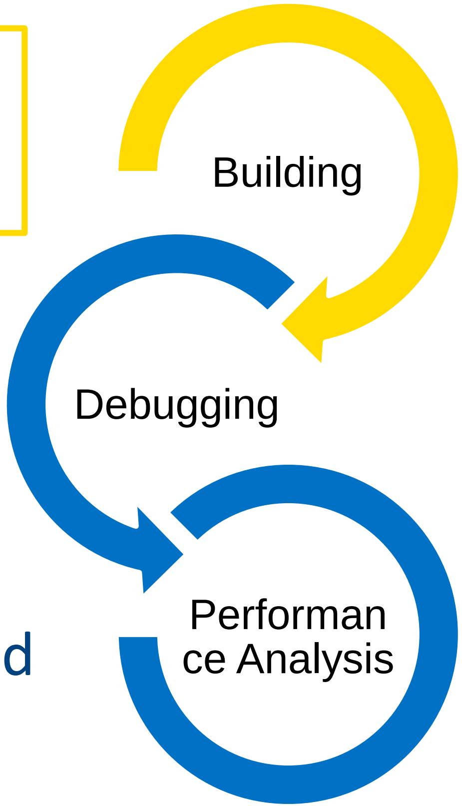


Code Builder Walkthrough Session Content

Create, Build and Analyze my OpenCL kernel with **Kernel Development Framework**

Debug my OpenCL host application and Kernel with **OpenCL® Debugger**

Analyze and Optimize your OpenCL application and kernel code with **OpenCL® Code Analyzer**



Create, Build and Analyze your OpenCL Kernel

with **Kernel Development Framework**

Our case study: **Sobel Filter Kernel**

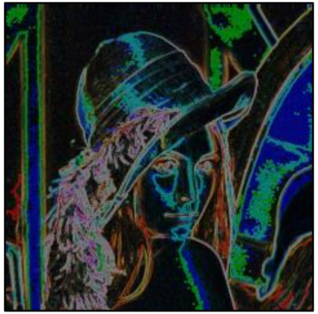
- Edge detection algorithm
- Discrete differentiation operator, computing an approximation of the gradient of the image intensity function

Source Image

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

horizontal and vertical derivative approximations

$$\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2}$$

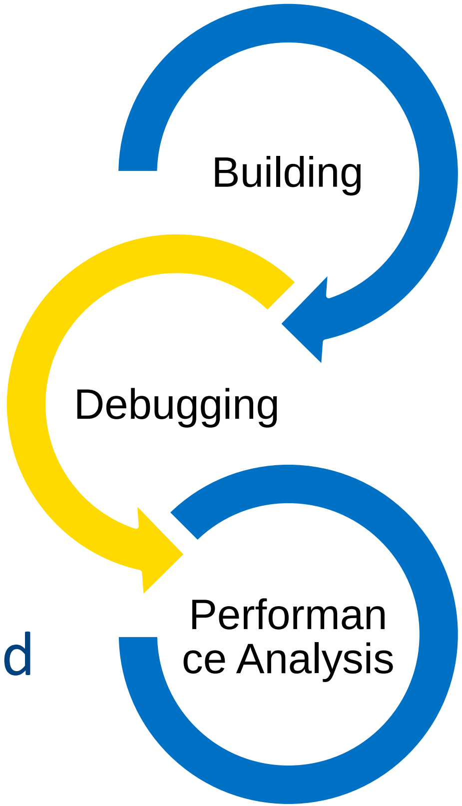


Code Builder Walkthrough Session Content

Create, Build and Analyze my OpenCL kernel with **Kernel Development Framework**

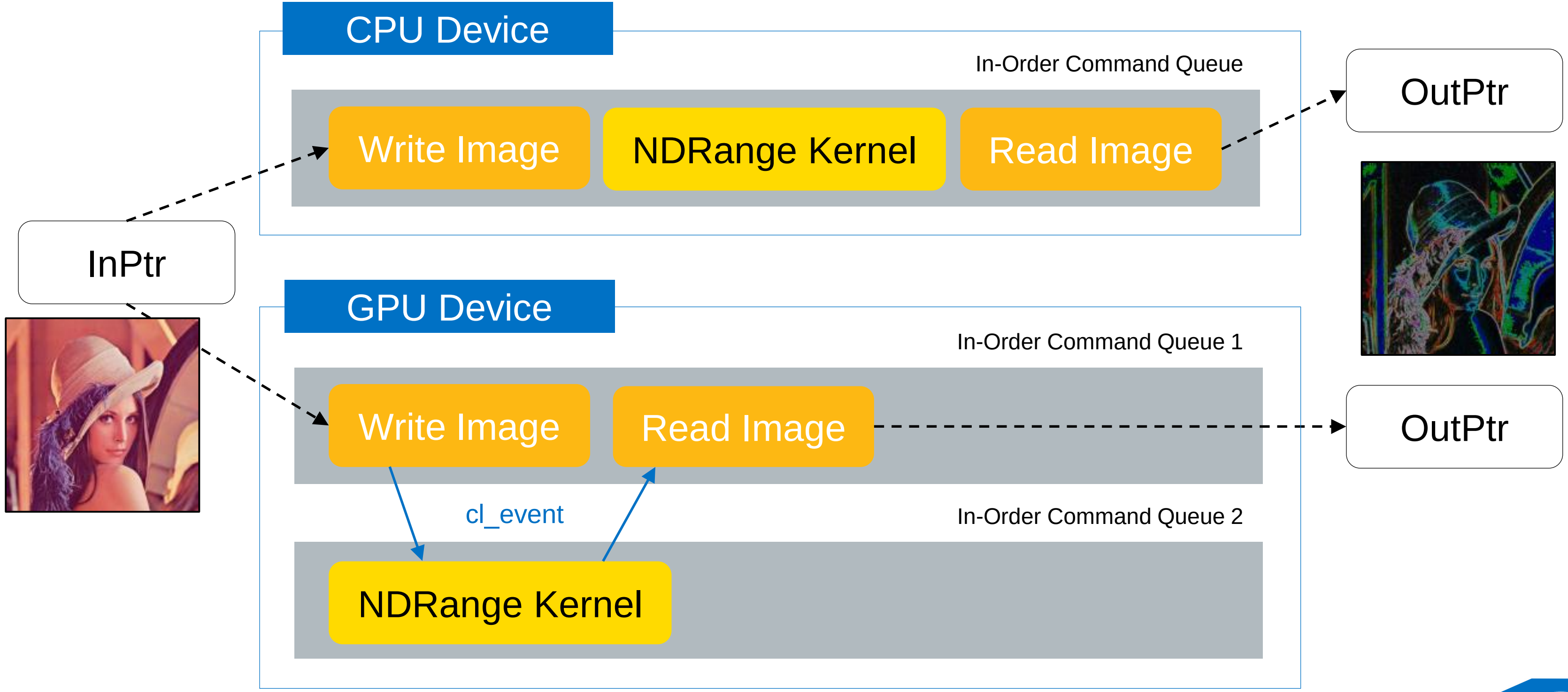
Debug my OpenCL host application and Kernel with **OpenCL® Debugger**

Analyze and Optimize your OpenCL application and kernel code with **OpenCL® Code Analyzer**



Debug OpenCL host applications and kernel code
with **OpenCL Debugger**

Our case study: **Sobel Filter App**

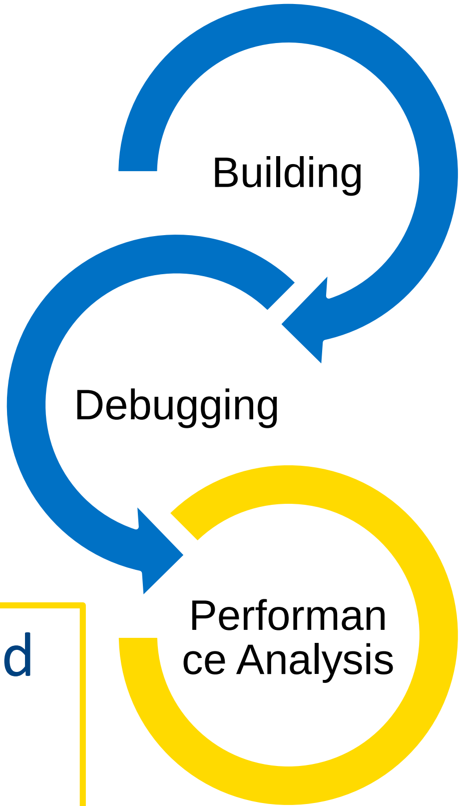


Code Builder Walkthrough Session Content

Create, Build and Analyze my OpenCL kernel with **Kernel Development Framework**

Debug my OpenCL host application and Kernel with **OpenCL® Debugger**

Analyze and Optimize your OpenCL application and kernel code with **OpenCL® Code Analyzer**



Analyze OpenCL host applications and kernel code

with **OpenCL Code Analyzer**

1st case study: Host level optimization on trivial “Hello World” application

- Serial execution of 4 basic compute workloads (OpenCL kernels)

→ kernel1 → kernel2 → kernel3 → kernel4 →

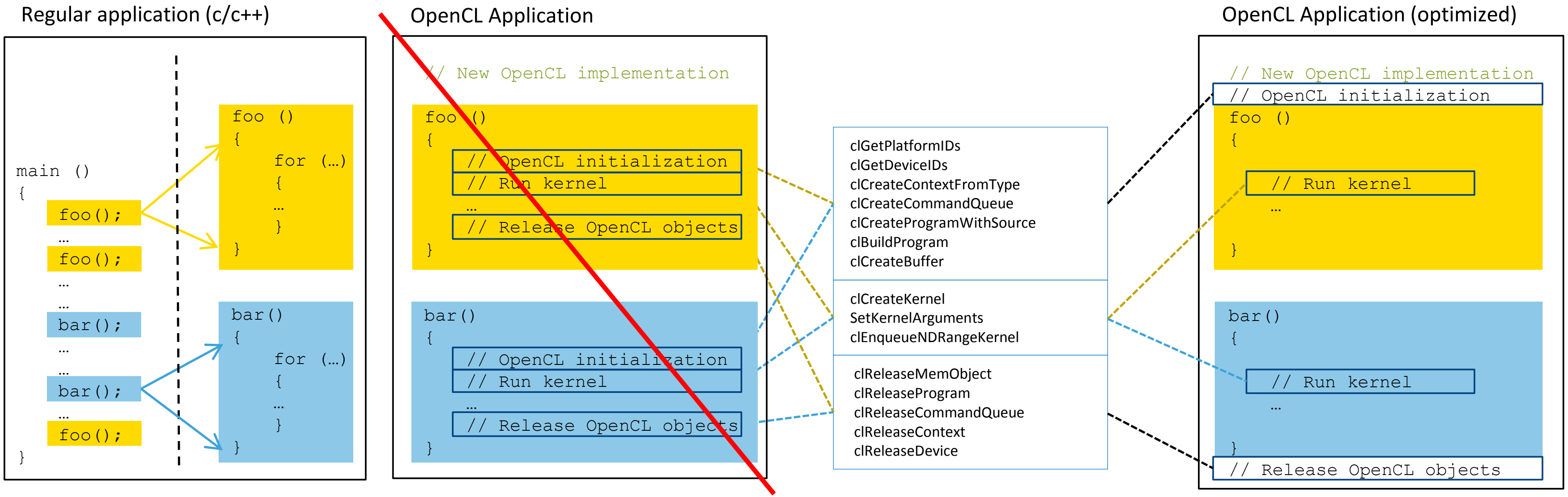
kernel1	kernel2	kernel3	kernel4
[C]=[A]+[B]	[C]=[A]-[B]	[C]=[A]||[B]	[C]=[A]&[B]

- Non optimized host code
- 4 optimization step

1st optimization – Wrong API Usage

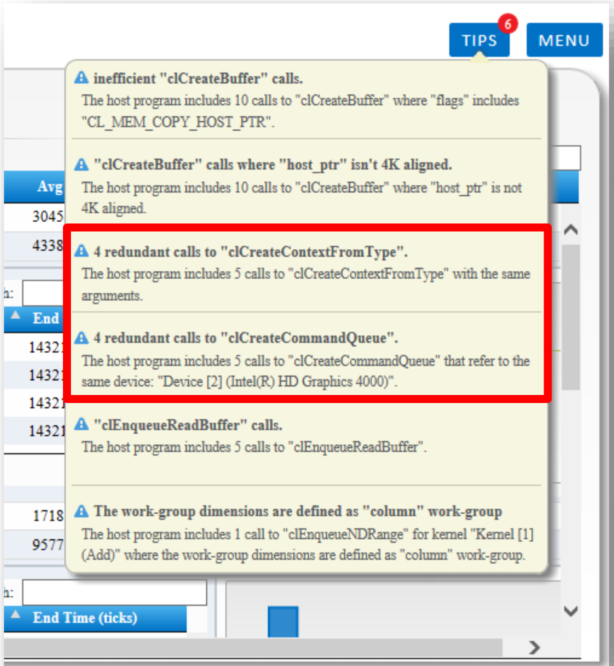
Avoid redundant usage of API calls

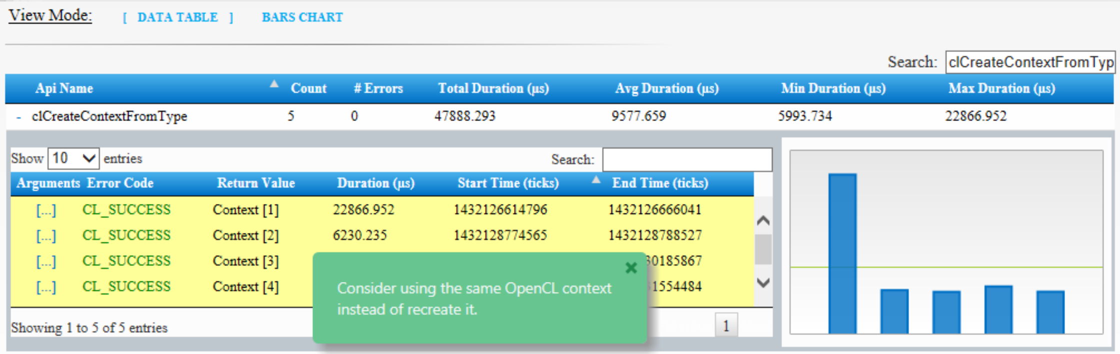
clBuildProgram calls takes ~40% of total execution time



1st optimization – Wrong API Usage

Avoid redundant usage of API calls





2nd optimization - Memory Access Patterns

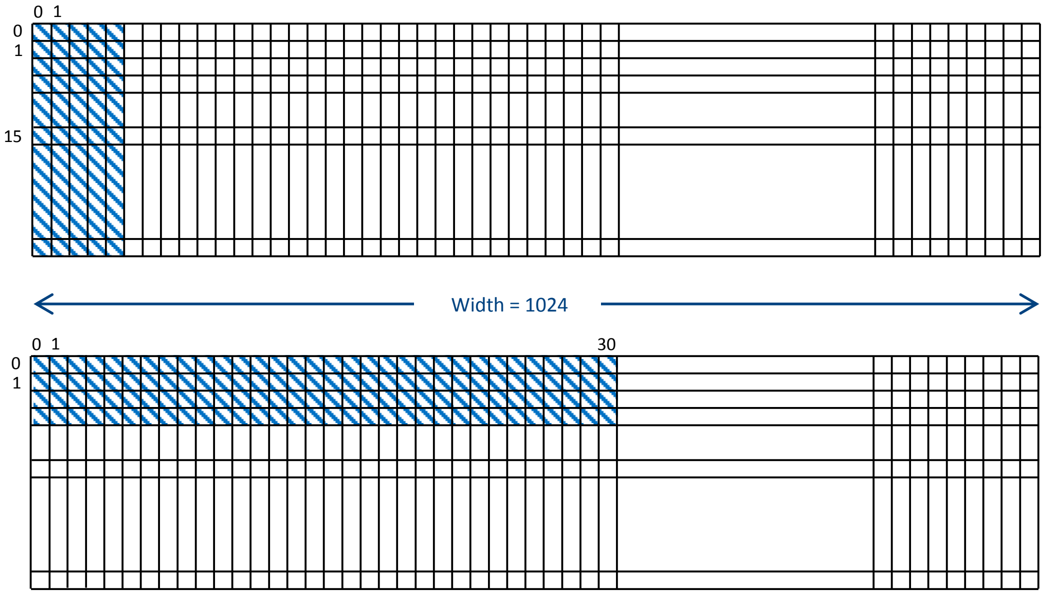
Row memory access VS column memory access

Host

```
size_t localWorkSize[2] = {1, 16};
clEnqueueNDRangeKernel(…, localWorkSize,…);
```

Kernel

```
const int id = y * width + x;
local = buffer[id];
```



~~LocalWorkSize[2] = {1,16}~~

{ x , y } =	0,0	0,1	. . .	0,15
y * width + x =	0	1024	. . .	15360

~~Y is different for every work-item in the work-group; every read operation comes from a different cache line for every work-item in the work-group~~

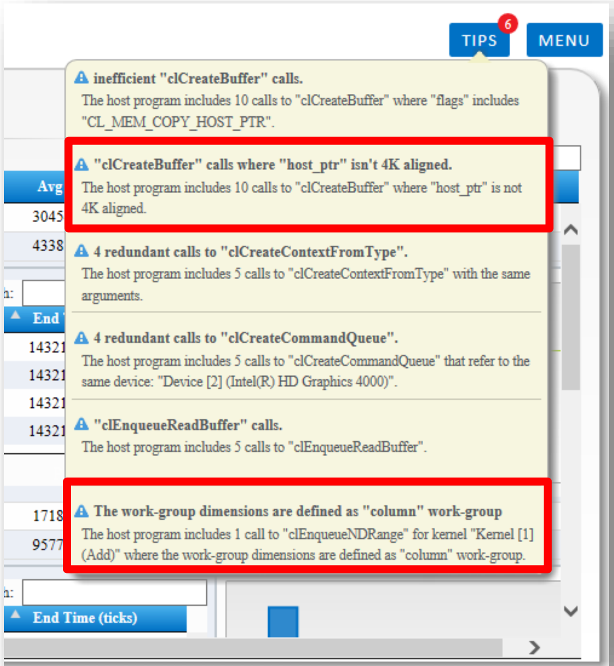
LocalWorkSize[2] = {16,1}

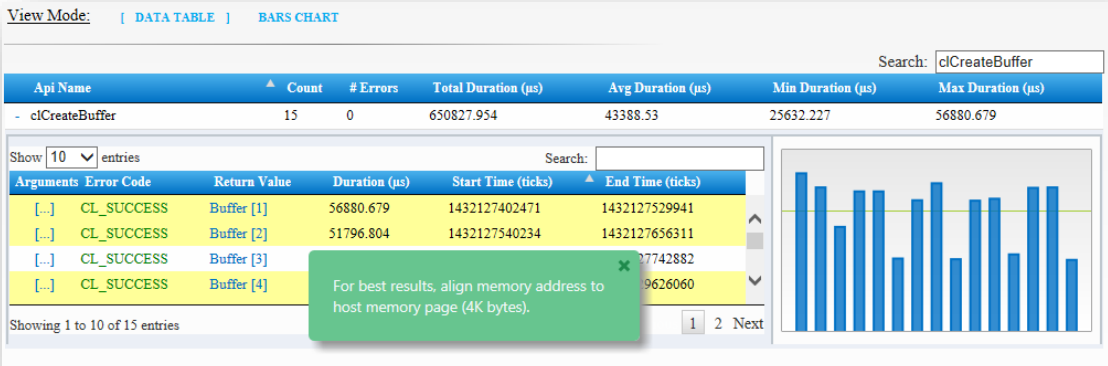
{ x , y } =	0,0	1,0		15,0
y * width + x =	0	1	. . .	15

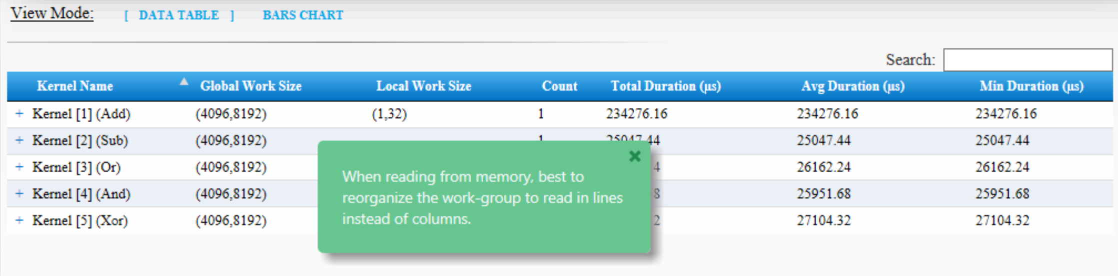
Y is constant for all work-items in the work-group. Id increases monotonically across the entire work-group, which means that the read operations comes from a single L3 cache line (16 x sizeof(int) = 64 bytes).

2nd optimization - Memory Access Patterns

Row memory access VS column memory access



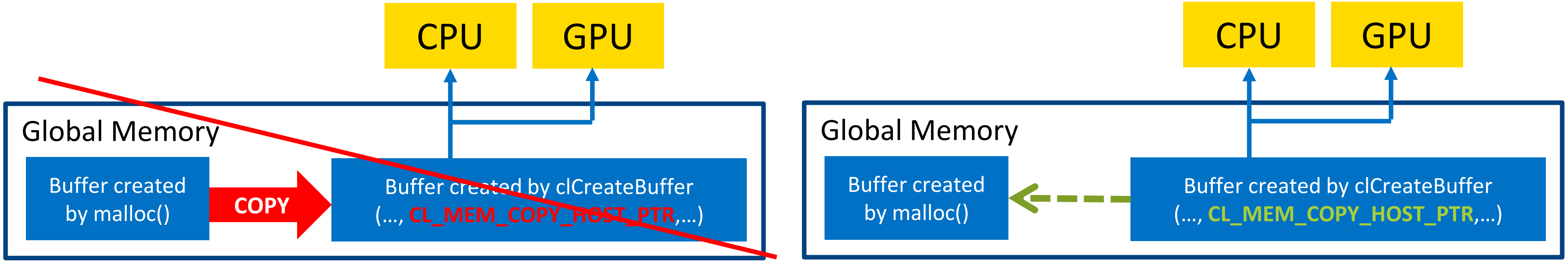




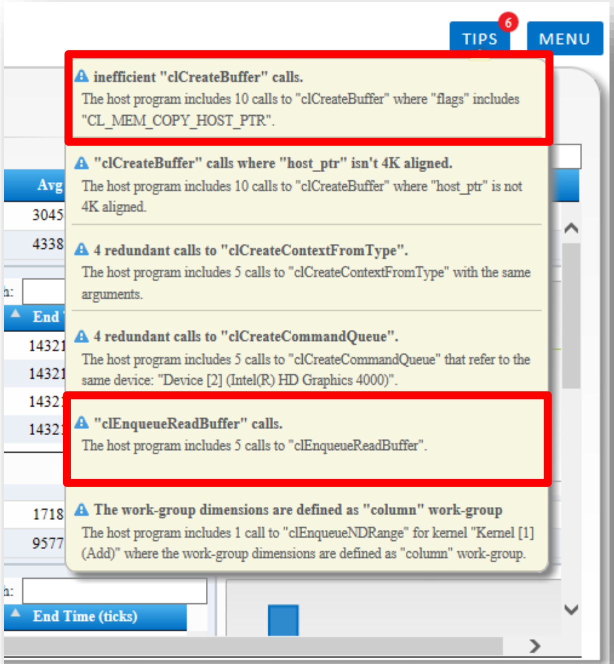
3rd optimization - Host to Device Transfers

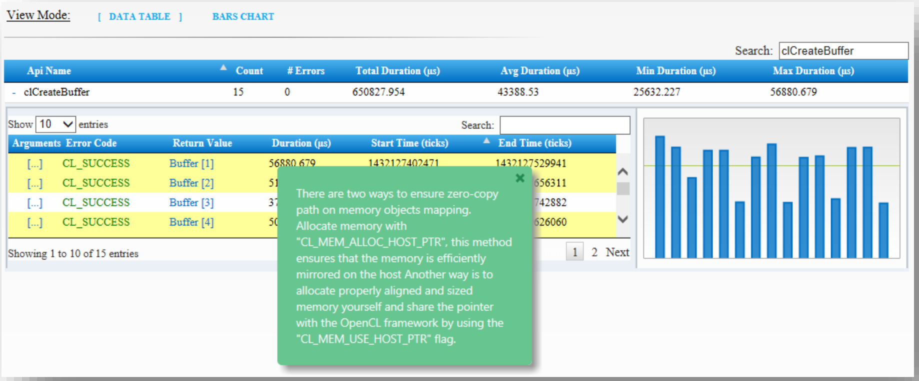
```
clCreateBuffer(..., CL_MEM_COPY_HOST_PTR, ...);
...
clEnqueueReadBuffer(...);
```

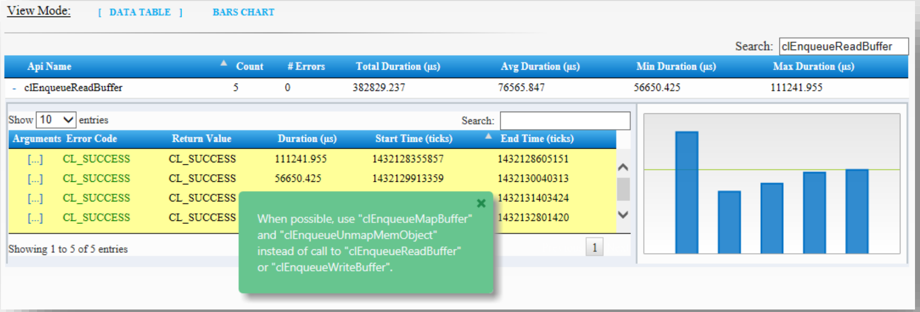
- clCreateBuffer
  - CL_MEM_USE_HOST_PTR flag enables the application to share its memory allocation with the OpenCL™ runtime implementation, and avoid memory copies of the buffer.



3rd optimization - Host to Device Transfers

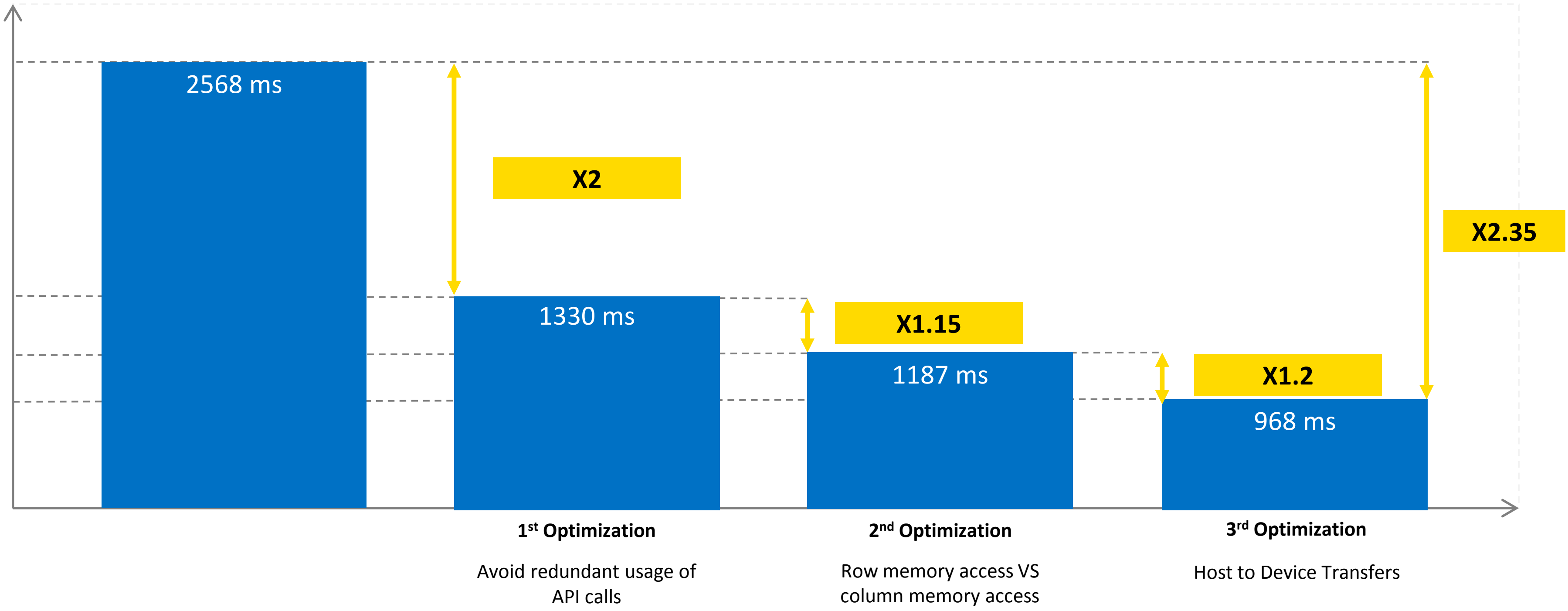






Host level optimization – Summary

On Intel® Iris™ Graphics 5100



Analyze OpenCL host applications and kernel code

with **OpenCL Code Analyzer**

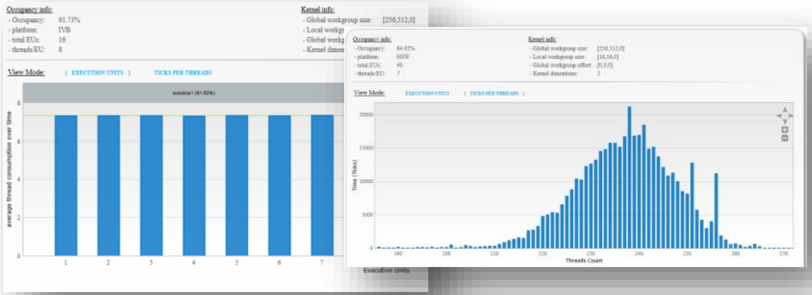
2nd case study: Kernel level analysis with:

- **Occupancy View** – how much the GPU is busy

  Higher is better

- **Latency View** - execution time of each kernel instruction (especially memory)

  Lower is better



- Top latency lines:

Line #	Total Latency (%)	Total latency (cycles)	Count	Average Latency (cycles)
35	60.05	304445328	131072	2322.7
36	39.95	202546628	131072	1545.3

Showing 1 to 2 of 2 entries

```
18  // Intel Corporation is the author of the Materials, and requests that all
19  // problem reports or change requests be submitted to it directly
20
21  // Latency Kernels:
22  ////////////////////////////////////////////////////////////////////////////////////
23
24  __kernel void LatencyTest(__global int* a, __global int* b, __global int* c, __global int* d, __global int* out)
25  {
26      size_t x      = get_global_id(0);
27      size_t y      = get_global_id(1);
28      size_t width  = get_global_size(0);
29      size_t height = get_global_size(1);
30
31      size_t id_a = y * width  + x;
32      size_t id_b = x * height + y;
33      //size_t id_b = y * width  + x;
34
35      int lb = a[id_b];
36      int la = b[id_a + lb];
37
38      out[id_a] =   la+lb;
39  }
```

Download, Learn, Code, Optimize

- Free download at: intel.com/software/opencl
- Follow us: @IntelOpenCL
- Contact as through our forum:

http://software.intel.com/en-us/forums/opencl

Try related products

- Native client development with Intel® Integrated Native Developer Experience (Intel® INDE)
- Performance tuning with the Intel® VTune™ Amplifier XE
- Media performance with the Intel® Media SDK

OpenCL

What is available online?

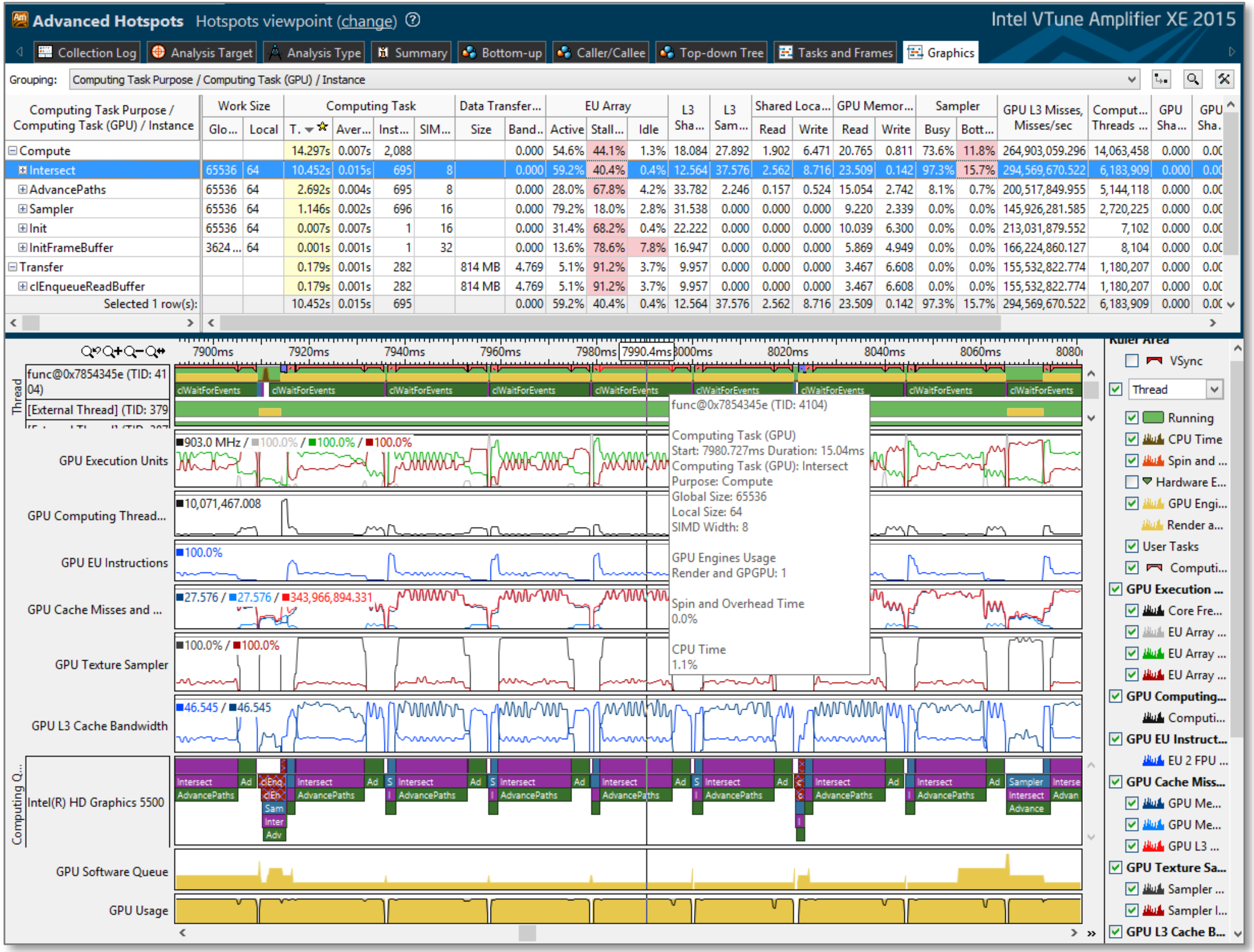
- ✓ Free Downloads
- ✓ Code Samples
- ✓ Documentation
- ✓ Tech Articles
- ✓ Reviews
- ✓ Forums and Support
- ✓ Webinars

Agenda

- Intel® Iris™ Graphics Overview
- Intel® OpenCL™ Code Builder
- Intel® VTune™ Amplifier 2015 ---- ***Presenter: Alexandr Kurylev***
- Optimization Techniques and Examples
- OpenCL™ 2.0 Overview
- Summary / Questions

What We Are Going To Talk About

- Introduction into GPU Analysis for OpenCL* applications with Intel® VTune™ Analyzer XE
- New features in recent releases
- Case study: an OpenCL* kernel optimization
- Summary / Questions



OpenCL kernel and data transfers

GPU hardware metrics attributed to kernels

GPU hardware metrics

OpenCL queue

GPU software queue

https://software.intel.com/en-us/articles/intel-vtune-amplifier-xe-getting-started-with-opencl-performance-analysis-on-intel-hd-graphics

GPU Analysis Features Overview

- Intel VTune Amplifier is a powerful performance debugging tool with mature GPU profiling capabilities
- Shows host and GPU activities correlated
- OpenCL kernel queue graphical view
- Allows to see kernels and their characteristics
- Goes down to hardware level by showing both GPU and CPU hardware metrics

Metric Presets For Intel® Graphics Analysis

- “Overview”: useful for Graphics and Compute and indeed, provides an overview
- “Compute Basic”: details about compute as Occupancy, IPC, FPUs Active etc.

New!

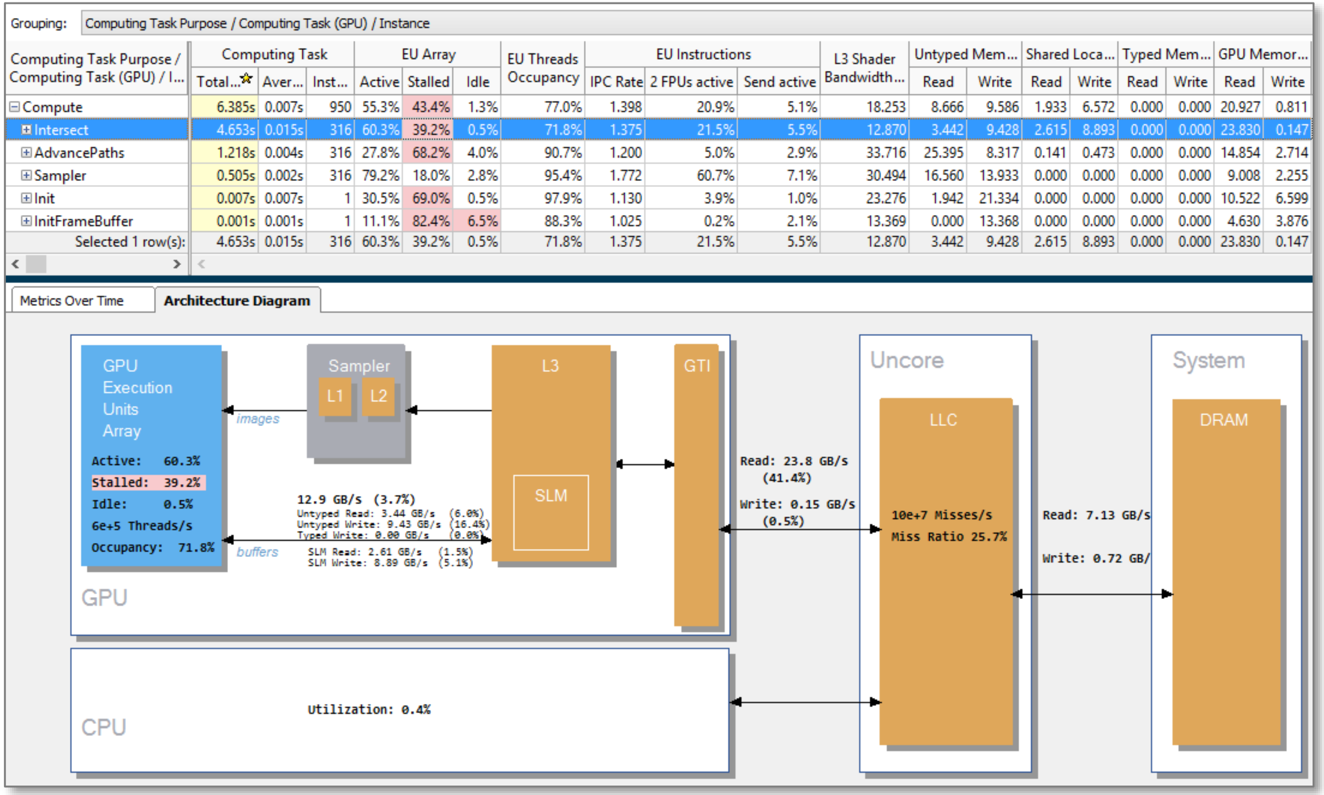
- “Compute Extended”: Memory Access and Coalescence metrics

Grouping: Computing Task Purpose / Computing Task (GPU) / Instance

Computing Task Purpose / Computing Task (GPU) / I...	Computing ...		Data Transfe...		EU Array			EU Threads Occupancy	EU Instructions			L3 Shader Bandwidth...	Memory Transactions Coalesce...				Shared Loca...		GPU Shader Barriers	GPU Shader Atomics
	Tot...	Aver...	Size	Ban..	Active	Stall...	Idle		IPC Rate	2 FPUs active	Send active		Untyped...	Untyped...	Typ.	Typ.	Read	Write		
⊟ Compute	15.061s	0.007s		0.000	54.8%	43.8%	1.3%	76.9%	1.400	20.8%	5.0%	18.167	46.9%	63.9%			1.912	6.504	0.000	0.000
⊞ Intersect	11.022s	0.015s		0.000	59.4%	40.2%	0.5%	71.8%	1.374	21.1%	5.4%	12.615	59.8%	68.4%			2.572	8.752	0.000	0.000
⊞ AdvancePaths	2.827s	0.004s		0.000	28.2%	67.5%	4.3%	90.3%	1.205	5.2%	3.0%	34.054	39.3%	46.3%			0.157	0.525	0.000	0.000
⊞ Sampler	1.204s	0.002s		0.000	79.7%	17.5%	2.8%	95.5%	1.781	61.8%	7.1%	31.670	49.9%	62.2%			0.000	0.000	0.000	0.000
⊞ Init	0.007s	0.007s		0.000	30.7%	68.7%	0.6%	97.7%	1.126	3.8%	1.1%	24.329	24.5%	7.2%			0.000	0.000	0.000	0.000
⊞ InitFrameBuffer	0.001s	0.001s		0.000	9.1%	85.6%	5.3%	89.6%	1.026	0.2%	1.7%	7.917		25.0%			0.000	0.000	0.000	0.000
⊟ Transfer	0.183s	0.001s	860 MB	4.932	5.3%	91.1%	3.6%	90.4%	1.009	0.0%	1.3%	10.821	25.0%	12.5%			0.000	0.000	0.000	0.000
⊞ clEnqueueReadBuffer	0.183s	0.001s	860 MB	4.932	5.3%	91.1%	3.6%	90.4%	1.009	0.0%	1.3%	10.821	25.0%	12.5%			0.000	0.000	0.000	0.000
Selected 1 row(s):	11.022s	0.015s		0.000	59.4%	40.2%	0.5%	71.8%	1.374	21.1%	5.4%	12.615	59.8%	68.4%			2.572	8.752	0.000	0.000

Architecture Diagram

New!



Grouping: Computing Task Purpose / Computing Task (GPU) / Instance

Computing Task Purpose / Computing Task (GPU) / I...	Computing Task: Total...	Computing Task: Aver...	Computing Task: Inst...	EU Array: Active	EU Array: Stalled	EU Array: Idle	EU Threads Occupancy	EU Instructions: IPC Rate	EU Instructions: 2 FPUs active	EU Instructions: Send active	L3 Shader Bandwidth...	Untyped Mem...: Read	Untyped Mem...: Write	Shared Loca...: Read	Shared Loca...: Write	Typed Mem...: Read	Typed Mem...: Write	GPU Memor...: Read	GPU Memor...: Write
⊟ Compute	6.385s	0.007s	950	55.3%	43.4%	1.3%	77.0%	1.398	20.9%	5.1%	18.253	8.666	9.586	1.933	6.572	0.000	0.000	20.927	0.811
⊞ Intersect	4.653s	0.015s	316	60.3%	39.2%	0.5%	71.8%	1.375	21.5%	5.5%	12.870	3.442	9.428	2.615	8.893	0.000	0.000	23.830	0.147
⊞ AdvancePaths	1.218s	0.004s	316	27.8%	68.2%	4.0%	90.7%	1.200	5.0%	2.9%	33.716	25.395	8.317	0.141	0.473	0.000	0.000	14.854	2.714
⊞ Sampler	0.505s	0.002s	316	79.2%	18.0%	2.8%	95.4%	1.772	60.7%	7.1%	30.494	16.560	13.933	0.000	0.000	0.000	0.000	9.008	2.255
⊞ Init	0.007s	0.007s	1	30.5%	69.0%	0.5%	97.9%	1.130	3.9%	1.0%	23.276	1.942	21.334	0.000	0.000	0.000	0.000	10.522	6.599
⊞ InitFrameBuffer	0.001s	0.001s	1	11.1%	82.4%	6.5%	88.3%	1.025	0.2%	2.1%	13.369	0.000	13.368	0.000	0.000	0.000	0.000	4.630	3.876
Selected 1 row(s):	4.653s	0.015s	316	60.3%	39.2%	0.5%	71.8%	1.375	21.5%	5.5%	12.870	3.442	9.428	2.615	8.893	0.000	0.000	23.830	0.147

Shows GPU blocks & CPU utilization and Uncore bandwidth while specific GPU task was running

Case Study: Gaussian Blur Kernel

- Convolves the image with Gaussian function

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

- Reduces noise and details
- Good representative of image processing kernels

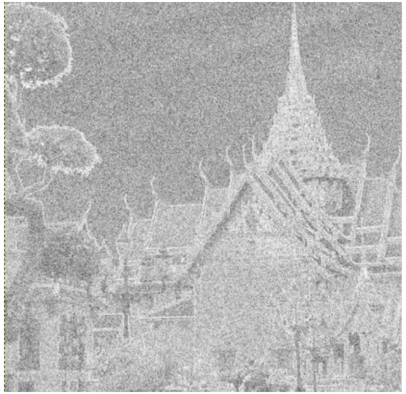
The sample matrix, produced by sampling the Gaussian filter kernel (σ = 0.840896) at the midpoints of each pixel and then normalizing:

0.00000067	0.00002292	**0.00019117**	0.00038771	**0.00019117**	0.00002292	0.00000067
0.00002292	0.00078634	0.00655965	0.01330373	0.00655965	0.00078633	0.00002292
0.00019117	0.00655965	0.05472157	0.11098164	0.05472157	0.00655965	**0.00019117**
0.00038771	0.01330373	0.11098164	**0.22508352**	0.11098164	0.01330373	0.00038771
0.00019117	0.00655965	0.05472157	0.11098164	0.05472157	0.00655965	**0.00019117**
0.00002292	0.00078633	0.00655965	0.01330373	0.00655965	0.00078633	0.00002292
0.00000067	0.00002292	**0.00019117**	0.00038771	**0.00019117**	0.00002292	0.00000067

Source: http://en.wikipedia.org/wiki/Gaussian_blur

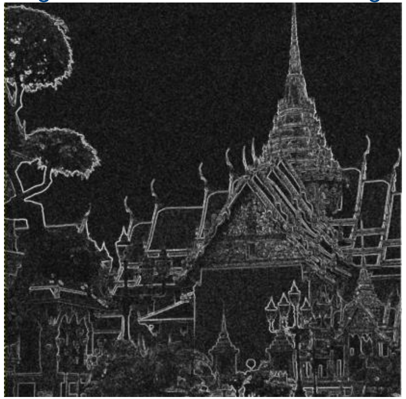
Original Image

Edge Detection on Original Image



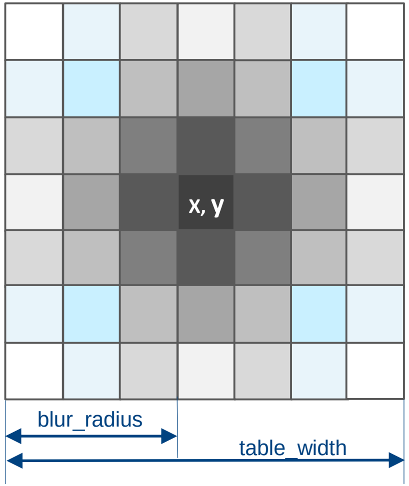
Blurred Image

Edge Detection on Blurred Image



Naïve implementation

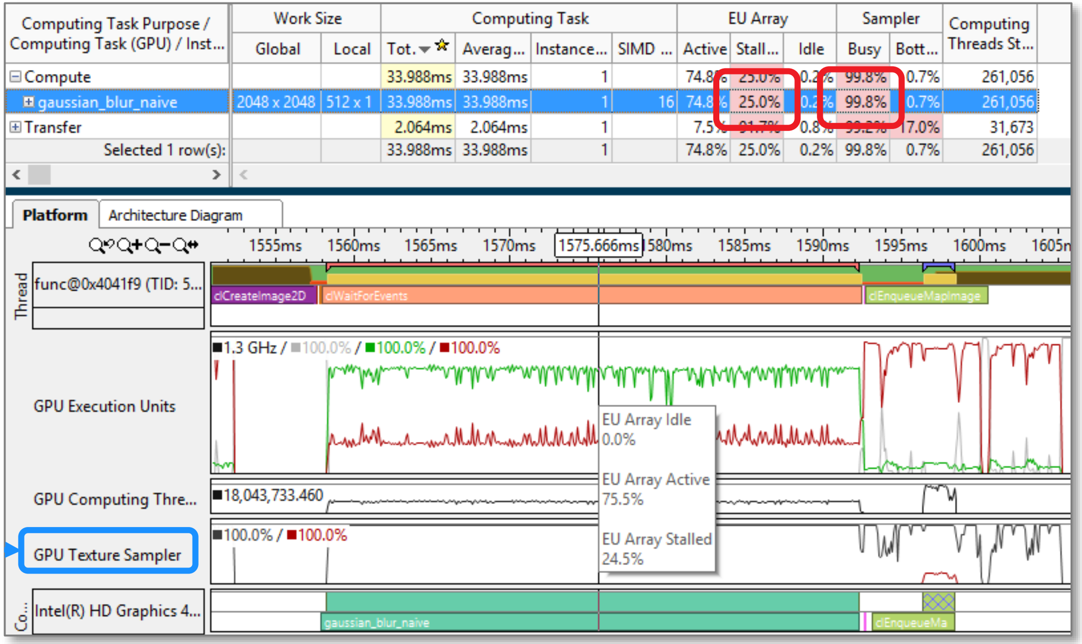
- Uses Sampler
- Processes one pixel per a work-item




```
const sampler_t samplerA = CLK_FILTER_NEAREST ;

__kernel void gaussian_blur_naive(read_only image2d_t src,
                                  __global float* table,
                                  const int blur_radius,
                                  write_only image2d_t dst)
{
    float4 dst_val = { 0, 0, 0, 0}, src_val  = { 0, 0, 0, 0};
    int i, k, h, w ;

    int x = get_global_id(0);
    int y = get_global_id(1);

    int table_width = blur_radius*2 + 1;

    for (i = 0; i < table_width; ++i)
    {
        w = i - blur_radius;
        for (k = 0; k < table_width; ++k)
        {
            h = k - blur_radius;
            src_val = read_imagef(src, samplerA, (int2)(x + w, y + h));

            dst_val += src_val * table[i*table_width + k];
        }
    }
    write_imagef(dst, (int2)(x, y), dst_val);
}
```

What Can We Learn From The Tool?

```
const sampler_t samplerA = CLK_FILTER_NEAREST ;

__kernel void gaussian_blur_naive(read_only image2d_t src,
                                  __global float* table,
                                  const int blur_radius,
                                  write_only image2d_t dst)
{
    float4 dst_val = { 0, 0, 0, 0}, src_val  = { 0, 0, 0, 0};
    int i, k, h, w ;

    int x = get_global_id(0);
    int y = get_global_id(1);

    int table_width = blur_radius*2 + 1;

    for (i = 0; i < table_width; ++i)
    {
        w = i - blur_radius;
        for (k = 0; k < table_width; ++k)
        {
            h = k - blur_radius;
            src_val = read_imagef(src, samplerA, (int2)(x + w, y + h));

            dst_val += src_val * table[i*table_width + k];
        }
    }
    write_imagef(dst, (int2)(x, y), dst_val);
}
```



* Code source by Intel

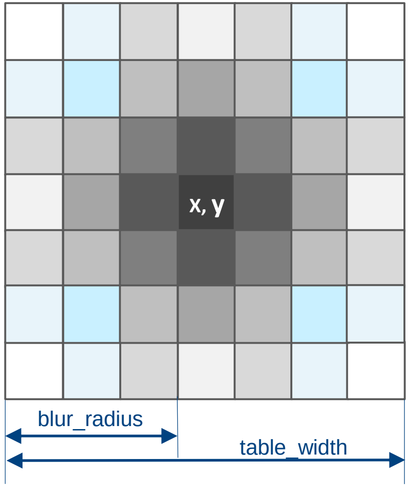
EU Stalled ~ 25% → EUs are waiting 25% of the time

Optimization Considerations

25% stalls due to Sampler accesses

However:

- regular access pattern allows using plane buffers instead of images (memory buffers are faster to access than Sampler)



- can use Gaussian Blur's separability property
  - make two kernels (instead of one): horizontal and vertical pass
  - access image data from linear buffers

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Gaussian Blur: Two Passes

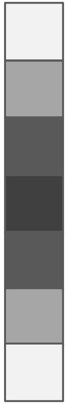
Taking advantage of Gaussian Blur’s separable property

```
float4 _unpack_uchar4(uchar4 src)
{
    private uchar4 temp = src;
    float4 res;
    res.x = (float)temp.x;
    res.y = (float)temp.y;
    res.z = (float)temp.z;
    res.w = (float)temp.w;
    return res;
}
```

```
uchar4 _pack_float4(float4 src)
{
    uchar4 res;
    res.x = (uchar)src.x;
    res.y = (uchar)src.y;
    res.z = (uchar)src.z;
    res.w = (uchar)src.w;
    return res;
}
```

```
__kernel void gaussian_blur_hor_1(__global read_only uchar4* src,
                                  __global read_only float* table,
                                  const int blur_radius,
                                  __global write_only uchar4* dst)
{
    float4 dst_val = { 0, 0, 0, 0}, src_val  = { 0, 0, 0, 0};
    int x = get_global_id(0);
    int y = get_global_id(1);
    int image_width = get_global_size(0);

    for (int k = 0; k < blur_radius*2 + 1; ++k)
    {
        int w = x + k - blur_radius;
        if ( w >= 0 && w < image_width )
        {
            src_val = _unpack_uchar4(src[image_width*y + w]);
        }
        else if (w < 0)
        {
            src_val = _unpack_uchar4(src[image_width*y]);
        }
        else if (w >= image_width)
        {
            src_val = _unpack_uchar4(src[image_width*y + image_width-1]);
        }
        float4 mult = (float4)table[k];
        dst_val  += src_val * mult;
    }
    dst[y*image_width + x] = _pack_float4(dst_val);
}
```

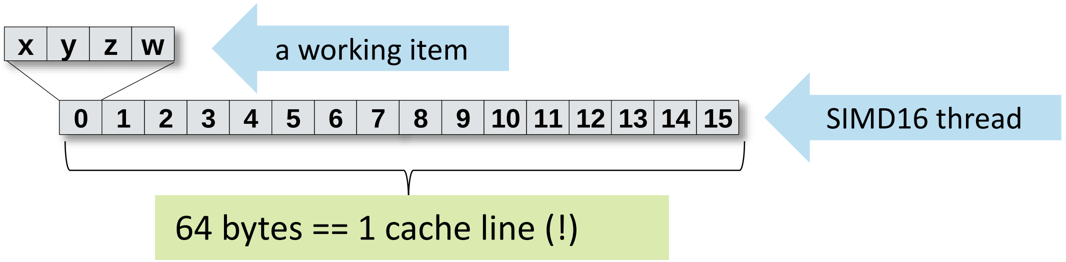
```
__kernel void gaussian_blur_vert_1(__global uchar4* src,
                                   __global float* table,
                                   const int blur_radius,
                                   __global uchar4* dst)
{
    float4 dst_val = { 0, 0, 0, 0}, src_val  = { 0, 0, 0, 0};
    int x = get_global_id(0);
    int y = get_global_id(1);
    int image_width  = get_global_size(0);
    int image_height = get_global_size(1);

    for (int k = 0; k < blur_radius*2 + 1; ++k)
    {
        int w = y + k - blur_radius;

        if ( w >= 0 && w < image_height )
        {
            src_val = _unpack_uchar4(src[image_width*w + x]);
        }
        else if (w < 0)
        {
            src_val = _unpack_uchar4(src[ x]);
        }
        else if (w >= image_height)
        {
            src_val = _unpack_uchar4(src[(image_width)*(image_height-1) + x]);
        }
        float4 mult = (float4)table[k];
        dst_val  += src_val * mult;
    }
    dst[y*image_width + x] = _pack_float4(dst_val);
}
```



Memory Access Pattern

Read/Write from/to memory by 4 bytes

Optimizes memory access and makes it coalesced




```
float4 _unpack_uchar4(uchar4 src)
{
    private uchar4 temp = src;
    float4 res;
    res.x = (float)temp.x;
    res.y = (float)temp.y;
    res.z = (float)temp.z;
    res.w = (float)temp.w;
    return res;
}
```

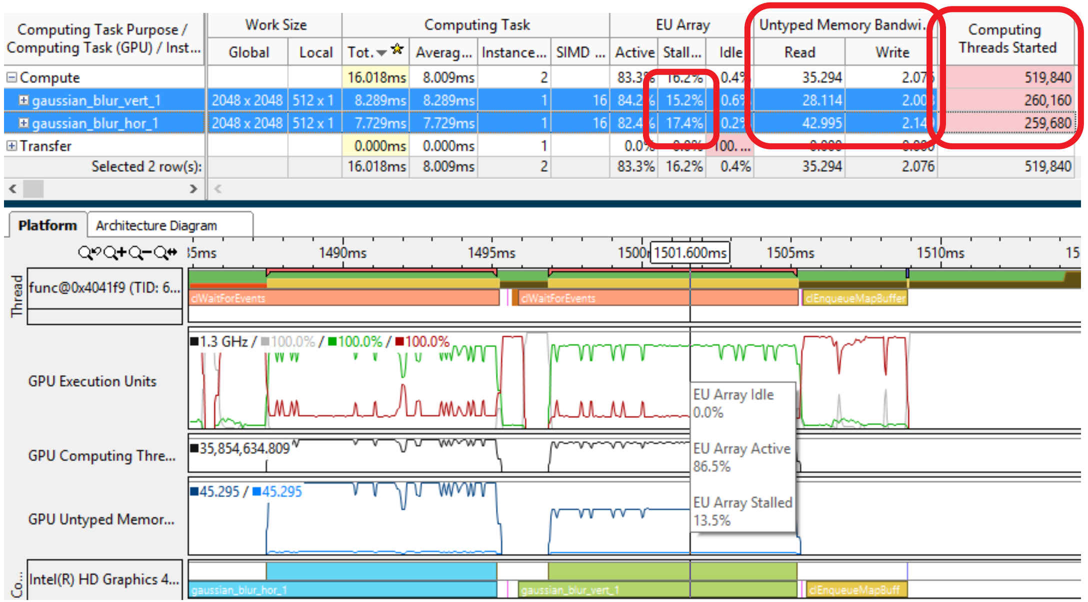
```
uchar4 _pack_float4(float4 src)
{
    uchar4 res;
    res.x = (uchar)src.x;
    res.y = (uchar)src.y;
    res.z = (uchar)src.z;
    res.w = (uchar)src.w;
    return res;
}
```

```
    for (int k = 0; k < blur_radius*2 + 1; ++k)
    {
        int w = x + k - blur_radius;
        if ( w >= 0 && w < image_width )
        {
            src_val = _unpack_uchar4(src[image_width*y + w]);
        }
        else if (w < 0)
        {
            src_val = _unpack_uchar4(src[image_width*y]);
        }
        else if (w >= image_width)
        {
            src_val = _unpack_uchar4(src[image_width*y + image_width-1]);
        }
        float4 mult = (float4)table[k];
        dst_val  += src_val * mult;
    }
    dst[y*image_width + x] = _pack_float4(dst_val);
}
```

```
    for (int k = 0; k < blur_radius*2 + 1; ++k)
    {
        int w = y + k - blur_radius;

        if ( w >= 0 && w < image_height )
        {
            src_val = _unpack_uchar4(src[image_width*w + x]);
        }
        else if (w < 0)
        {
            src_val = _unpack_uchar4(src[ x]);
        }
        else if (w >= image_height)
        {
            src_val = _unpack_uchar4(src[(image_width)*(image_height-1) + x]);
        }
        float4 mult = (float4)table[k];
        dst_val  += src_val * mult;
    }
    dst[y*image_width + x] = _pack_float4(dst_val);
}
```

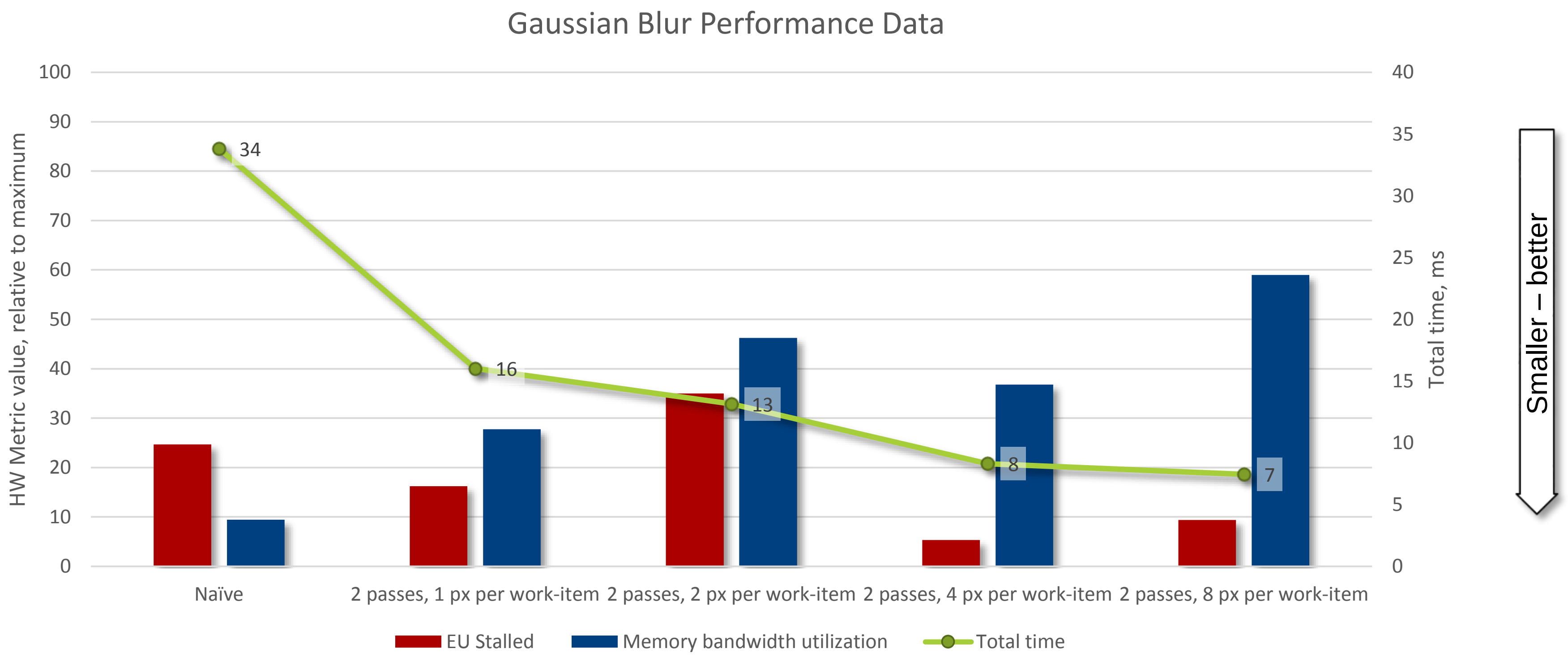
Two Passes. Is more speed-up possible?

- EU↔L3 memory bandwidth is far from its peak value (43 + 28 = 71 vs 128 GB/s for 2 slices @ 1 GHz)
- Process more pixels in one work item



Use available memory bandwidth to speed up the kernels

Gaussian Blur Optimization Steps



Hardware metrics guide optimization of GPU bound code

What To Optimize For

Optimization Target	Metric(s) to Watch	Traps
Memory Throughput	Untyped Read/Write → for 128 GB/s at 1 GHz per 1 slice	Non-coalesced accesses might consume additional bandwidth
Occupancy	Kernels with too small work for working item EU Idle → 0 % EU Stalled → 0 %	More EU threads is not good when the working item is doing too small work
Compute Throughput	EU Active → 100 % EU Stalled → 0 % EU Idle → 0 %	Redundant calculation might elevate EUActive

Operating Systems Support

Feature	Window*	Linux*	Android*
GPU usage, per engine	√	√	√
GPU usage items attributed to OpenCL	√	√	
GPU Hardware Metrics	√		√
Media Server Studio CPU-side APIs support		√	
OpenCL 1.2 support	√	√	
OpenCL 2.0 basic support	√	√	
GPU Architecture Diagram	√		

Summary

Use VTune to analyze OpenCL* applications running on Intel® Graphics

- Watch for hot kernels and possible inefficient CPU↔GPU interactions

Optimize Hottest OpenCL* Kernels using Intel Graphics Hardware Metrics

Watch for

- Memory Access Pattern
- Occupancy
- EU utilization

VTune helps use full potential of Intel Iris Graphics with your OpenCL* application

Agenda

- Intel® Iris™ Graphics Overview
- Intel® OpenCL™ Code Builder
- Intel® VTune™ Amplifier 2015
- Optimization Techniques and Examples ---- ***Presenter: Anita Banerjee***
- OpenCL™ 2.0 Overview
- Summary / Questions

Optimization Techniques and Examples

- Host Side Optimizations
- Memory Matters
  - Host to Device
  - Device Access
- Compute Characteristics
  - Maximizing Gflops
- Device Side Optimizations

OpenCL* Host Side Optimizations

- Pre-compile kernels if possible
  - Compile once and save binary – load at app start
- Enqueue multiple commands in queue
  - Use in-order queues
  - No need to wait or “clFinish()” on every kernel
  - Allows OpenCL runtime to minimize overhead
- Use null for LWS
  - Let driver to choose the best LWS for you if you are not sure

Optimizing Host to Device Transfers

Host (CPU) and Device (GPU) share the same physical memory

For OpenCL* buffers:

- No transfer needed (zero copy)!
- Allocate memory aligned to a cache line (64 bytes) and multiple of 4KB (page size)
- Create buffer with system memory pointer and **`CL_MEM_USE_HOST_PTR`**
- Use **`clEnqueueMapBuffer()`** to get pointer to access data from CPU
- Use **`clEnqueueUnmapMemObject()`** to give the pointer back to GPU before using at kernels.

For OpenCL* images:

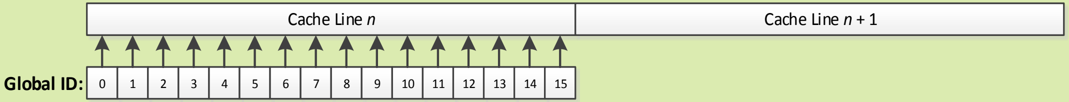
- Use cl_khr_image2d_from_buffer Ext.
- http://www.khronos.org/message_boards/viewtopic.php?t=5545

`__global` and `__constant` Memory Access Examples

Adjacent work items should ideally read/store adjacent memory locations

x = data[get_global_id(0)]

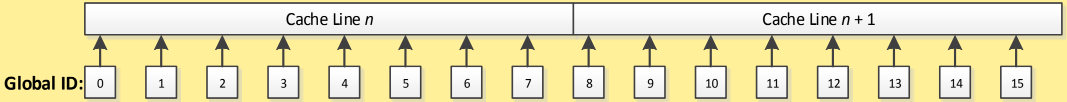
- One cache line, full bandwidth



Especially avoid the work items reading/storing skipping memory or vertically

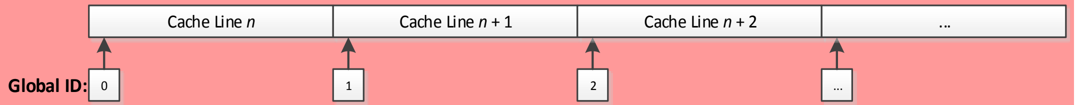
x = data[get_global_id(0) * 2]

- Strided, half bandwidth



x = data[get_global_id(0) * 16]

- Very strided, worst-case



`__global` and `__constant` Memory

Global memory access performance depends of size and alignment.

Best: Load/Store 16 bytes of data at a time, starting from a cache line aligned address

OK: Load/Store at least 4 bytes of data at a time, starting from a 4 bytes aligned address

`__local` Memory

	L3$	SLM	
`data[get_global_id(0)];`	1 cache line Full bandwidth	16 banks Full bandwidth	`data[get_local_id(0)];`
`data[get_global_id(0) + 1];`	2 cache lines Half bandwidth	16 banks Full bandwidth	`data[get_local_id(0) + 1];`
`data[get_global_id(0) * 2];`	2 cache line Half bandwidth	8 banks Half bandwidth	`data[get_local_id(0) * 2];`
`data[get_global_id(0) * 16];`	16 cache lines Worst case!	1 bank Worst case!	`data[get_local_id(0) * 16];`
`data[get_global_id(0) * 17];`	16 cache line Worst case!	16 banks Full bandwidth	`data[get_local_id(0) * 17];`

When picking a memory type, consider access patterns!

`__private` Memory

Compiler can *usually* allocate Private Memory in the Register File

- Even if Private Memory is dynamically indexed
- Good Performance

Fallback: Private Memory allocated in Global Memory

- Accesses are very strided
- Bad Performance

Intel® HD Graphics Memory Hierarchy

Intel® HD Graphics Register File	**Shared Local memory (SLM)**	**Main memory**
Private	**Local**	**Global/Constant**
Minimal latency	**Low latency**	**Longest latency**

Use Built-in Functions

```
dp = V0.x * V1.x + V0.y * V1.y
   + V0.z * V1.z + V0.w + V1.w;
```

→

```
dp = dot(V0, V1);
```

```
C = sqrt(A * A + B * B);
```

→

```
C = hypot(A, B);
```

```
cl =
fmin(fmax(X,minVal),maxVal);
```

→

```
cl = clamp(X,
minVal, maxVal);
```

Allow the Compiler to Optimize Better

Trade Accuracy vs. Speed

```
c[id] += a[id]*b[id];
```
→
```
c[id] =
fma(a[id],b[id],c[id]);
```

```
a[id] = sin(a[id]);
```
→
```
a[id] =
native_sin(a[id]);
```

- Use mad()/fma(): Either explicitly with built-ins or via -cl-mad-enable build option
- Use native_* versions of trigonometry functions
  or compile with –cl-fast-relaxed-math build option
- floats processing is about ~2x of throughput than int – HD5200 and older

Avoid Byte/Short Load and Stores

```
__kernel void k(global uchar* a, global uchar* b)
{
  int gid = get_global_id(0);
  a[gid] *= b[gid];
  . . .
}
```

```
__kernel void k(global uint4* a, global uint4* b)
{
  int gid = get_global_id(0);
  a[gid] *= b[gid];
   . . .
}
```

Avoid byte or short loads. Load and store in greater chunk

Agenda

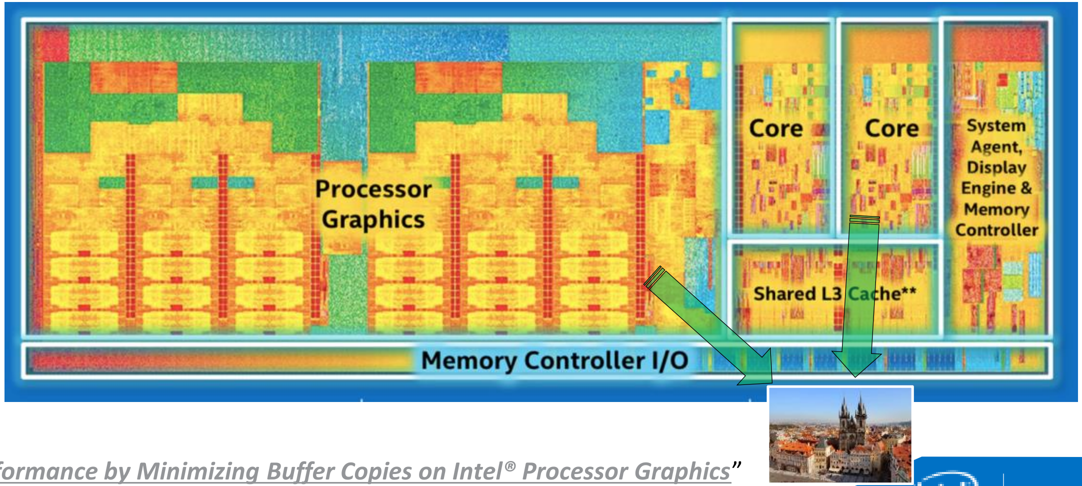
- Intel® Iris™ Graphics Overview
- Intel® OpenCL™ Code Builder
- Intel® VTune™ Amplifier 2015
- Optimization Techniques and Examples
- OpenCL™ 2.0 Overview ---- ***Presenter: Sonal Sharma & Michael Stoner***
- Summary / Questions

Shared Virtual Memory *(Pre-history)*

Builds upon "shared physical memory" (SPM) feature

- *SPM* established with **OpenCL 1.0** => *CL_MEM_USE_HOST_PTR* flag
- Supported on Intel 3rd Gen processors with HD Graphics
- Eliminated buffer copy costs, aka "zero-copy" buffers*
- Buffer must have 4k byte alignment and size divisible by 64

SPM available since 2011, but many OpenCL apps still not using it…



* See "*Getting the Most from OpenCL™ 1.2: How to Increase Performance by Minimizing Buffer Copies on Intel® Processor Graphics*"

Shared Virtual Memory

Allows de-referencing of host-allocated virtual memory pointers directly on the GPU

Enables GPU offload of pointer-oriented algorithms (e.g. using trees or linked lists)

```
//
//        Array of Element                           Array of floats
//           structures
//
//     ||===================||
//     ||    .............  ||                     ||===========||
//     ||    .............  ||<-----+              ||  ......... ||
//     ||===================||      |              ||    float   ||
//     ||  float* internal--||------+              ||    float   ||
//     ||  float* external--||-------------------->||    float   ||
//     ||  float value <----||------+              ||  ......... ||
//     ||===================||      |              ||  ......... ||
//     ||    .............  ||      |              ||    float   ||
//     ||    .............  ||      |              ||    float   ||
//     ||===================||      |              ||    float   ||
//     ||===================||      |              ||    float   ||
//     ||  float* internal--||------+              ||    float   ||
//     ||  float* external--||-------------------->||    float   ||
//     ||  float value      ||                     ||    float   ||
//     ||===================||                     ||    float   ||
//     ||    .............  ||                     ||  ......... ||
//     ||    .............  ||                     ||===========||
//     ||===================||
//
```

SVM buffer SVM buffer

3 types of SVM

Coarse-grain buffers (Intel 5th Gen Processors w/ HD Graphics 5300)

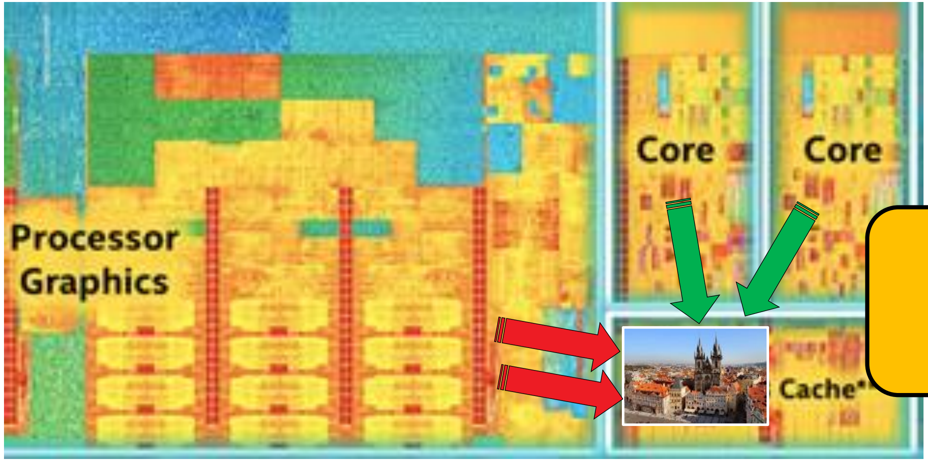
- SVM buffers are mapped to either CPU or GPU at any given time
- Access is controlled by clEnqueueMap/Unmap commands

***Un-mapped* state: Only GPU can access buffer**

3 types of SVM

Coarse-grain buffers (Intel 5th Gen Processors w/ HD Graphics 5300)

- ➢ SVM buffers are mapped to either CPU or GPU at any given time
- ➢ Access is controlled by clEnqueueMap/Unmap commands



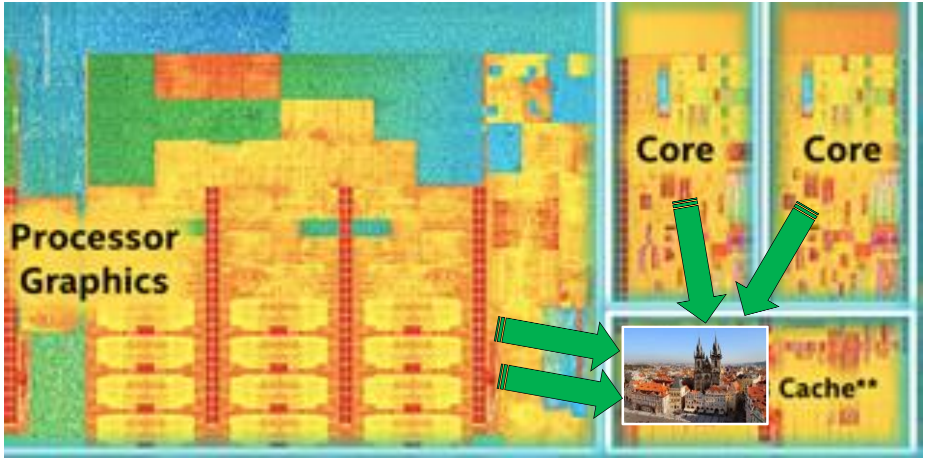
***Mapped* state: Only CPU can access buffer**

3 types of SVM

Fine-grain buffers (Intel 5th Gen Processors w/ HD Graphics 5500+)

- SVM buffers can be accessed from either CPU or GPU at any time
- Can use *atomics* to avoid 'race' conditions

  Check if device supports (CL_DEVICE_SVM_FINE_GRAIN_BUFFER & CL_DEVICE_SVM_ATOMICS flags)

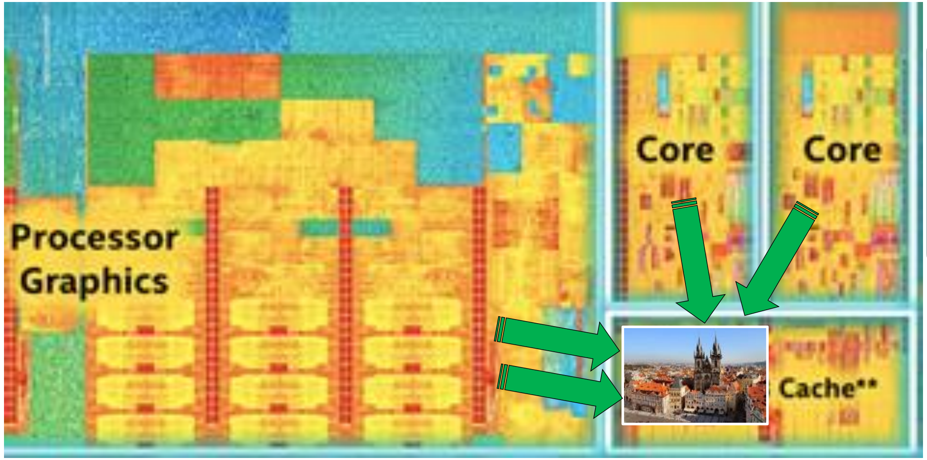


Fine grain SVM buffer allows simultaneous access from CPU & GPU

3 types of SVM

Fine-grain system memory (Future Intel Processors)

- CPU & GPU can share anything allocated from the C-runtime 'heap' (i.e. *malloc/new*)
- **Ideal end-state** – requires convergence of OS, H/W, and API support



Full CPU/GPU memory coherency for all heap allocations

Shared Virtual Memory – API Basics

Allows host-allocated VM pointers to be de-referenced directly on the GPU

- No need for clCreateBuffer() => *cl_mem* object to encapsulate a buffer
- Use `clSVMAlloc`() to allocate memory. To create Fine-grained buffer, use flag `CL_MEM_SVM_FINE_GRAIN_BUFFER`

GPU

```
__kernel void svmbasic
    (global float *inputFloats)
{
    ...
    float x = *inputFloats;
    ...
}
```

CPU host

```
// Allocate SVM
float* inputFloats = (float*) clSVMAlloc(
            context,
            CL_MEM_READ_ONLY,    // input buffer
            size*sizeof(float), // size (bytes)
            0 );                 // alignment
```

Shared Virtual Memory – Kernel setup

Two ways to pass SVM pointer to a kernel

- Use `clSetKernelArgSVMPointer`() to pass the pointer directly
- If this buffer contains pointers to additional SVM regions, use `clSetKernelExecInfo` () with `CL_KERNEL_EXEC_INFO_SVM_PTRS` flag

CPU host

```
// inputElements is an SVM allocation
err = clSetKernelArgSVMPointer(kernel, 0, inputElements);

// inputElements contains pointers to inputFloats
err = clSetKernelExecInfo(kernel,
    CL_KERNEL_EXEC_INFO_SVM_PTRS,
    sizeof(inputFloats),
    &inputFloats
);
```

Nested Parallelism

- “Device-side enqueue”
  - OpenCL kernels can launch ‘child’ kernels on the device without returning control to the CPU host
- Enables flexible work scheduling entirely on the GPU
  - Recursive algorithms (e.g. quickSort, Sierpinski’s Carpet, etc.)
- Also, meets competitive challenge with CUDA’s implementation
- Device-side enqueue Building blocks:
  - Host side API: creating a default device queue from the host
  - Block Syntax: simplifies device side enqueue
  - Device side API: enqueue_kernel

Sierpiński Carpet

The **Sierpiński carpet** is a plane fractal first described by Wacław Sierpiński in 1916.

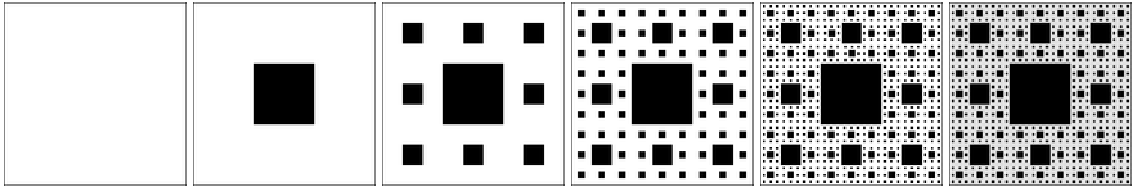
Start with a white square.

Divide the square into 9 sub-squares in a 3-by-3 grid

Paint the central sub-square black.

Apply the same procedure recursively to the remaining 8 sub-squares

And so on …



See http://en.wikipedia.org/wiki/Sierpinski_carpet for more info

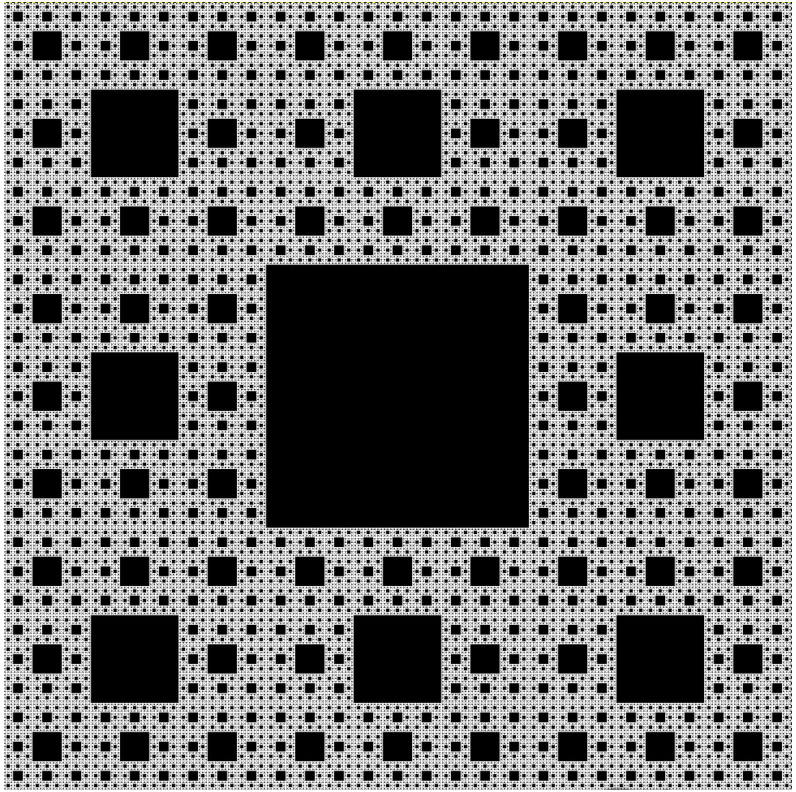
* Sierpiński Carpet image sequence above from http://en.wikipedia.org/wiki/Sierpinski_carpet

Sierpiński Carpet Kernel in OpenCL 2.0

```
__kernel void sierpinski(__global char* src, int width, int offsetx, int offsety)
{
        int x = get_global_id(0);
        int y = get_global_id(1);
        queue_t q = get_default_queue();

        int one_third = get_global_size(0) / 3;
        int two_thirds = 2 * one_third;

        if (x >= one_third && x < two_thirds && y >= one_third && y < two_thirds)
        {
                src[(y+offsety)*width+(x+offsetx)] = BLACK;
        }
        else
        {
                src[(y+offsety)*width+(x+offsetx)] = WHITE;

                if (one_third > 1 && x % one_third == 0 && y % one_third == 0)
                {
                        const size_t  grid[2] = {one_third, one_third};
                        enqueue_kernel(q, 0, ndrange_2D(grid), ^{ sierpinski(src, width, x+offsetx, y+offsety); });
                }
        }
}
```

Easy to translate recursive algorithm to implementation

Sierpiński Carpet - Result



2187x2187 image: 8^6 = 299592 enqueue_kernel calls!

Conference Session on OpenCL 2.0

Achieving Performance with OpenCL 2.0 on Intel Processor Graphics

Presented By: Robert Ioffe, Sonal Sharma and Michael Stoner

Date and Time: May 13th 2015, 10:40am

Agenda

- Intel® Iris™ Graphics Overview
- Intel® OpenCL™ Code Builder
- Intel® VTune™ Amplifier 2015
- Optimization Techniques and Examples
- OCL 2.0 Overview
- Summary / Questions

Additional Resources

intel®