Debugging And Optimizing OpenCL* Applications

Best Practices and Tools

Yuval Eshkol (yuval.eshkol@intel.com)
Legal Notices and Disclaimers

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES REGARDING THE FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information. The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYMark 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.
- All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.
- All products, platforms, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice. All dates specified are target dates, are provided for planning purposes only and are subject to change.
- This document contains information on products in the design phase of development. Do not finalize a design with this information. Revised information will be published when the product is available. Verify with your local sales office that you have the latest datasheet before finalizing a design.
- Code names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third parties are not authorized by Intel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel's internal code names is at the sole risk of the user.
- Intel, Intel Inside, Intel Atom and Intel Core are trademarks of Intel Corporation in the U.S. and other countries.
- Other names and brands may be claimed as the property of others.
- Copyright © 2015-2016, Intel Corporation. All rights reserved.
- OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos

Optimization Notice
Copyright © 2016, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.
Objectives

- Understand the architecture characteristics relevant to compute applications on Intel® Processor Graphics
- Learn techniques for optimizing OpenCL* applications for Intel® Processor Graphics
- Introduce with Intel® tools for development, debugging and optimizing OpenCL* applications
Agenda

- Intel® Processor Graphics introduction
- Optimization techniques for OpenCL* applications
- Develop OpenCL* applications with Intel® SDK for OpenCL™ Applications
- Debug OpenCL* applications with Intel® SDK for OpenCL™ Applications
- Optimize OpenCL* application with Intel tools
  - Intel® VTune™ Amplifier XE
  - Intel® SDK for OpenCL* Applications
Introduction
Intel® Processor Graphics Architecture

- Today, our focus is on Intel® Iris™ Graphics and Intel® HD Graphics in 6th Generation Intel® Core™ Processors
  - Or, Intel Processor Graphics Gen9
- For more details, see our whitepaper, The Compute Architecture of Intel Processor Graphics Gen7.5/Gen8.0/Gen9.0
Intel® Processor Graphics Architecture

- Outstanding rendering and media performance
- High-throughput general purpose compute capabilities
- High bandwidth memory hierarchy
- Deep integration with on-die CPUs and other SoC devices
Intel® Processor Graphics Architecture

- Modular architecture
- Scalability for a range of products
GEN9 Core Processor

*Other names and brands may be claimed as the property of others.

An Intel® Core™ i7 processor 6700K SoC and its ring interconnect architecture.
- Modular architecture, which enables scalability across a wide range of target products
Intel® Graphics Architecture Building Blocks

A potential product design composed of three slices, each with three sub-slices, for a total of 72 EUs

*Other names and brands may be claimed as the property of others.

Intel® Core™ i7-6970HQ Processor with Intel® Iris™ Pro Graphics 580
Intel® Graphics Architecture Building Blocks - EU

- 7 threads with
  - 128 GRF of 32 bytes, Accessible as SIMD-8 32-bit

- Can co-issue up to 4 instruction processing units including:
  - 2 FPUs
  - Branch unit
  - Message send unit
Intel® Graphics Architecture Building Blocks - Subslice

- 8 EUs (can be changed for scalability) X 7 threads
  - Dedicated hardware resources and register files for 56 simultaneous threads
- Local thread dispatcher unit
- Supporting instruction caches
- Sampler
  - Read-only memory fetch unit includes 2-level caches
- Data port
  - A memory load/store unit
Intel® Graphics Architecture Building Blocks - Slice

- 3 sub-slices for a total of 24 EUs
- Banked L3 cache
- Cachelines are 64 bytes each
- Smaller highly-banked shared local memory
- For sharing among EU hardware threads within the same subslice
- Same latency as L3 data cache
- Can yield full bandwidth for access patterns that may not be 64-byte aligned
Intel® Graphics Architecture Building Blocks - Product

- SoC product instantiates a single slice or groups of slices.
- Additional front end logic
  - Manage command submission
- Fixed-function logic
- Support 3D rendering, and media pipelines
- Graphics technology interface (GTI)
- Interfaces to the rest of the SoC components (memory, CPU, …)

The Intel® Core™ i7 processor 6700K with Intel® HD Graphics 530.
Intel® Graphics Architecture - Memory

- Share DRAM physical memory with the CPU
- Zero copy
- Shared memory coherency and consistency

SoC chip level memory hierarchy and its theoretical peak bandwidths for Intel processor graphics gen9.
How OpenCL* Maps To Intel® Processor Graphics
Executing OpenCL* Kernels

- OpenCL* work items map to SIMD lanes of a hardware thread
- Compiler may decide to compile a kernel SIMD32, SIMD16, or SIMD8
  - A compiler heuristic will choose a SIMD width that best maximizes register footprint within a single hardware thread and avoid needs for register spill/fill.
  - Typically short kernels that need less than 128 bytes of private memory will compile to SIMD32
Executing OpenCL* Kernels

SIMD8 Compilation

Example: work group with 64 work items

512 Bytes of Register Space Per Work Item

SIMD16 Compilation

256 Bytes of Register Space Per Work Item

Compiler can trade register space for IPC!
Executing OpenCL* Kernels

- Example: SIMD16 compile, 64 work items per work group

Work Group A

Work Group B

Work Group C
Optimize OpenCL* applications

Best Practices
Two Levels of Optimizations for OpenCL*

- Application level optimization
  - Optimization is ~vendor agnostic, tools are ~similar
  - Many API level tricks are ~portable

- Kernel level optimization side
  - Entirely vendor-specific (and so are tools)
  - Kernels optimizations are generally less portable
Optimization Factors

- Optimize host API calls
- Reduce host <> device memory traffic and bandwidth
- Optimizing memory access
- Maximizing occupancy and computation
- Kernel algorithm optimization

OpenCL* Developer Guide for Intel® Processor Graphics:

https://software.intel.com/en-us/iocl_opg
Optimization Factors

- Optimize host API calls
- Reduce Host <> Device memory traffic and bandwidth
- Optimizing memory access
- Maximizing occupancy
- Maximizing computation
- Kernel algorithm optimization
Intro to Host-Side API Optimization

- “Wall-clock” time: wrap OpenCL* API calls with timestamps + printfs
  - Most “device” OpenCL* APIs (like clEnqueueNDRangeKernel) just put a call into a queue and immediately return
    - To measure actual execution time, need to synchronize on completion

- Better solution with profiling events
  - OpenCL* profiling info from events associated with all queued commands:
    - Time spent in the command queue, driver and actual hardware exec
  - Enable queue for profiling with CL_QUEUE_PROFILING_ENABLE
  - Wait for the command completion before querying the event stats

- Best solution
  - Use Intel® Code Builder to get API profiling report and optimization tips
Avoid Redundant Usage of API Calls

Regular application (c/c++)

```c
main ()
{
    foo();
    ...
    foo();
    ...
    bar();
    ...
    bar();
    ...
    foo();
}
```

OpenCL Application

```c
foo ()
{
    // OpenCL initialization
    // Run kernel
    // Release OpenCL objects
}
```

Optimization Notice

Code snippets provided in this presentation are for illustrative purposes only. Intel disclaims any and all implied or express warranties associated with the code snippets, and any and all use of such code snippets is at the sole discretion and exclusive risk of the user.

*Other names and brands may be claimed as the property of others.*
Reusing Compilation Results

- Reusing compilation results is typically faster than recreating the program from the source
- Check it for your specific program and device
- Cache the resulting binaries after the first OpenCL* compilation and reuse them by calling clCreateProgramWithBinary
  - To retrieve binaries generated from CreateProgramWithSource and clBuildProgram:
    - Call clGetProgramInfo with the CL_PROGRAM_BINARIES parameter
- A better way – Pre compile your program offline with Intel® Code Builder for OpenCL™ API and save intermediate binaries
Code Builder API Calls Report

![Code Builder API Calls Report Image]

- **Optimization Notice**
  
  *Other names and brands may be claimed as the property of others.
## Code Builder API Calls Report

![Graphical View of Code Builder API Calls](image)

### Table: Code Builder API Calls Report

<table>
<thead>
<tr>
<th>Api Name</th>
<th>Count</th>
<th># Errors</th>
<th>Total Duration (ms)</th>
<th>Avg Duration (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clBuildProgram</code></td>
<td>5</td>
<td>0</td>
<td>1800529.69</td>
<td>36105.93</td>
</tr>
<tr>
<td><code>clCreateBuffer</code></td>
<td>14</td>
<td>0</td>
<td>505422.278</td>
<td>40361.488</td>
</tr>
<tr>
<td><code>clCreateContextFromType</code></td>
<td>5</td>
<td>0</td>
<td>42899.852</td>
<td>8579.97</td>
</tr>
<tr>
<td><code>clCreateKernel</code></td>
<td>5</td>
<td>0</td>
<td>73049.299</td>
<td>14609.86</td>
</tr>
<tr>
<td><code>clCreateProgramWithSource</code></td>
<td>5</td>
<td>0</td>
<td>1173.758</td>
<td>234.752</td>
</tr>
<tr>
<td><code>clCreateProgramWithSource</code></td>
<td>5</td>
<td>0</td>
<td>491.837</td>
<td>98.367</td>
</tr>
<tr>
<td><code>clEnqueueNDRangeKernel</code></td>
<td>5</td>
<td>0</td>
<td>560.809</td>
<td>112.162</td>
</tr>
<tr>
<td><code>clEnqueueReadBuffer</code></td>
<td>5</td>
<td>0</td>
<td>175029.128</td>
<td>35005.828</td>
</tr>
<tr>
<td><code>clFinish</code></td>
<td>10</td>
<td>0</td>
<td>308955.786</td>
<td>30895.786</td>
</tr>
<tr>
<td><code>clGetDeviceIDs</code></td>
<td>5</td>
<td>0</td>
<td>10.264</td>
<td>2.053</td>
</tr>
<tr>
<td><code>clGetPlatformIDs</code></td>
<td>2</td>
<td>0</td>
<td>47.213</td>
<td>23.607</td>
</tr>
<tr>
<td><code>clGetPlatformInfo</code></td>
<td>10</td>
<td>0</td>
<td>16.422</td>
<td>1.642</td>
</tr>
<tr>
<td><code>clReleaseCommandQueue</code></td>
<td>5</td>
<td>0</td>
<td>19446.86</td>
<td>3889.372</td>
</tr>
<tr>
<td><code>clReleaseContext</code></td>
<td>5</td>
<td>0</td>
<td>15139.796</td>
<td>3027.059</td>
</tr>
<tr>
<td><code>clReleaseDevice</code></td>
<td>5</td>
<td>0</td>
<td>15.19</td>
<td>3.038</td>
</tr>
<tr>
<td><code>clReleaseKernel</code></td>
<td>5</td>
<td>0</td>
<td>27.917</td>
<td>5.583</td>
</tr>
<tr>
<td><code>clReleaseMemObject</code></td>
<td>15</td>
<td>0</td>
<td>256074.68</td>
<td>17071.645</td>
</tr>
</tbody>
</table>

### Optimization Notice

*4 redundant calls to "clCreateKernel".*

The host program includes 5 calls to `clCreateKernel` with the same arguments.

*4 redundant calls to "clCreateContextFromType".*

The host program includes 5 calls to `clCreateContextFromType` that refer to the same device. "Device [1] (Intel(R) HD Graphics 4400)"

*4 redundant calls to "clCreateBuffer".*

The host program includes 10 calls to `clCreateBuffer` where the "flags" includes "CL_MEM_COPY_HOST_PTR".
Optimization Factors

- Optimize host API calls
- **Reduce Host <> Device memory traffic and bandwidth**
- Optimizing memory access
- Maximizing occupancy
- Maximizing computation
- Kernel algorithm optimization
Zero Copy

- The key hardware feature that enables zero copy is the fact that the CPU and GPU have shared physical memory
- Memory shared between the CPU and GPU can be efficiently accessed by both devices
Zero Copy

- Always improves performance
- To create zero copy buffers, do one of the following:
  - Use CL_MEM_ALLOC_HOST_PTR
    - Let the runtime handle creating a zero copy allocation buffer for you
  - Use CL_MEM_USE_HOST_PTR with:
    - Buffer allocated at a 4096 byte boundary (aligned to a page and cache line boundary)
    - Total size that is a multiple of 4096 byte (page size)

```c
int *pbuf = (int *)aligned_malloc(sizeof(int) * 1024, 4096);
cl_mem myZeroCopyCLMemObj = clCreateBuffer(ctx,...CL_MEM_USE_HOST_PTR...);
```

Zero Copy - Accessing the Buffer on the Host

- Use `clEnqueueMapBuffer()` and `clEnqueueUnmapMemObject()`
- Don't use:
  - `clEnqueueReadBuffer()`
  - `clEnqueueWriteBuffer()`

* This behavior may not be the same on all platforms.
## Code Builder API Calls Report

### Api Calls: [Data Table] Graphical View

<table>
<thead>
<tr>
<th>Api Name</th>
<th>* Count</th>
<th># Errors</th>
<th>Total Duration (µs)</th>
<th>Avg Duration (µs)</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ clBuildProgram</td>
<td>5</td>
<td>0</td>
<td>1800529.69</td>
<td>360105.938</td>
<td></td>
</tr>
<tr>
<td>+ clCreateBuffer</td>
<td>15</td>
<td>0</td>
<td>605422.278</td>
<td>40361.485</td>
<td></td>
</tr>
<tr>
<td>+ clCreateCommandQueue</td>
<td>5</td>
<td>0</td>
<td>42899.852</td>
<td>8570.97</td>
<td></td>
</tr>
<tr>
<td>+ clCreateContextFromType</td>
<td>5</td>
<td>0</td>
<td>73049.299</td>
<td>14609.96</td>
<td></td>
</tr>
<tr>
<td>+ clCreateKernel</td>
<td>5</td>
<td>0</td>
<td>1173.738</td>
<td>234.752</td>
<td></td>
</tr>
<tr>
<td>+ clCreateProgramWithSource</td>
<td>5</td>
<td>0</td>
<td>491.837</td>
<td>98.367</td>
<td></td>
</tr>
<tr>
<td>+ clEnqueueNDRangeKernel</td>
<td>5</td>
<td>0</td>
<td>560.809</td>
<td>112.162</td>
<td></td>
</tr>
<tr>
<td>+ clEnqueueReadBuffer</td>
<td>5</td>
<td>0</td>
<td>175029.128</td>
<td>35005.826</td>
<td></td>
</tr>
<tr>
<td>+ clFinish</td>
<td>10</td>
<td>0</td>
<td>30895.786</td>
<td>30895.579</td>
<td></td>
</tr>
<tr>
<td>+ clGetDeviceIDs</td>
<td>5</td>
<td>0</td>
<td>10.264</td>
<td>2.063</td>
<td></td>
</tr>
<tr>
<td>+ clGetPlatformIDs</td>
<td>2</td>
<td>0</td>
<td>47.213</td>
<td>23.607</td>
<td></td>
</tr>
<tr>
<td>+ clGetPlatformInfo</td>
<td>10</td>
<td>0</td>
<td>16.422</td>
<td>1.642</td>
<td></td>
</tr>
<tr>
<td>+ clReleaseCommandQueue</td>
<td>5</td>
<td>0</td>
<td>19448.86</td>
<td>3889.372</td>
<td></td>
</tr>
<tr>
<td>+ clReleaseContext</td>
<td>5</td>
<td>0</td>
<td>15139.796</td>
<td>3027.959</td>
<td></td>
</tr>
<tr>
<td>+ clReleaseDevice</td>
<td>5</td>
<td>0</td>
<td>15.19</td>
<td>3.038</td>
<td></td>
</tr>
<tr>
<td>+ clReleaseKernel</td>
<td>5</td>
<td>0</td>
<td>27.917</td>
<td>5.583</td>
<td></td>
</tr>
<tr>
<td>+ clReleaseMemObject</td>
<td>15</td>
<td>0</td>
<td>256074.68</td>
<td>17071.645</td>
<td></td>
</tr>
</tbody>
</table>

**Optimization Notice**

Copyright © 2016, Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.*
There are two ways to ensure zero-copy path on memory objects mapping. Allocate memory with "CL_MEM_ALLOC_HOST_PTR", this method ensures that the memory is efficiently mirrored on the host. Another way is to allocate properly aligned and sized memory yourself and share the pointer with the OpenCL framework by using the "CL_MEM_USE_HOST_PTR" flag.

For best results, align memory address to host memory page (4K bytes).
Shared Virtual Memory (SVM)

- Supported from OpenCL* 2.0
- Enables the host and device to seamlessly share pointers and complex pointer-containing data-structures
  - Linked lists or trees
- Tight Host-Kernel synchronization using atomics
  - Just like two distinct cores in a CPU
Shared Virtual Memory (SVM)

- Basically a productivity feature
  - Targeted to fulfill the needs of developers for tighter host-device synchronization beyond enqueuing commands and synchronizing through events
- Also a very important performance feature
  - Go to “GPU daemon – Road to Zero Cost Submission”
    by Michal Mrozek and Zbigniew Zdanowicz (Intel) on THURSDAY 21\textsuperscript{st} APRIL 16:30 – 17:00
Shared Virtual Memory (SVM)

- Requires dedicated hardware coherency support
  - Such as enabled in Intel Core Processors with Intel® Graphics Gen8/Gen9 compute architecture
- There are different levels of SVM support depending on OpenCL* platform hardware capabilities
  - Tradeoff between productivity and portability
- Not all OpenCL* platforms support all SVM features
- OpenCL* 2.0 specification defines a minimum level of required SVM support
  - Other features are optional
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 clEnqueueSVMMap/Unmap commands

- Fine-grain buffers (Intel 5th Gen Processors w/ HD Graphics 5500+)
  - SVM buffers can be accessed from either CPU or GPU at any time
  - Use atomics to control access (if CPU & GPU may try to modify the same memory location)
  - Check CL_DEVICE_SVM_FINE_GRAIN_BUFFER for fine-grained buffer SVM support, CL_DEVICE_SVM_ATOMICS is for atomics support

- Fine-grain system memory (Future Intel Processors)
  - CPU & GPU can share anything allocated from the C-runtime ‘heap’ (i.e. malloc/new)
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-map state:
Only GPU can access the 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

Map state:
Only GPU can access the buffer
3 types of SVM

- Fine-grain buffers (Intel 5th Gen Processors w/ HD Graphics 5500+)
  - SVM buffers can be accessed from both CPU and GPU at any time
  - Can use atomics to avoid ‘race’ conditions
3 types of SVM

- Fine-grain system memory (Future Intel Processors)

Refer to OpenCL* 2.0 Shared Virtual Memory Overview for more information
SVM in VTune Amplifier XE Views

<table>
<thead>
<tr>
<th>Computing Task Purpose / Computing Task (GPU) / Instance</th>
<th>Work Size</th>
<th>Computing Task</th>
<th>SVM Usage Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Global</td>
<td>Local</td>
<td>Total Time</td>
</tr>
<tr>
<td>Compute</td>
<td>499.66s</td>
<td>0.038s</td>
<td>13,005</td>
</tr>
<tr>
<td>ReadWriteCopy_NoAlignPartWrite</td>
<td>2097152</td>
<td>256</td>
<td>133.55s</td>
</tr>
<tr>
<td>ReadWriteCopy_NoAlignPartWrite</td>
<td>2097152</td>
<td>256</td>
<td>61.267s</td>
</tr>
<tr>
<td>ReadWriteCopy</td>
<td>2097152</td>
<td>256</td>
<td>47.392s</td>
</tr>
<tr>
<td>ReadWriteCopyUnRoll</td>
<td>2097152</td>
<td>256</td>
<td>34.491s</td>
</tr>
<tr>
<td>ReadOnly</td>
<td>2097152</td>
<td>256</td>
<td>34.422s</td>
</tr>
<tr>
<td>ReadWriteCopy</td>
<td>2097152</td>
<td>256</td>
<td>32.639s</td>
</tr>
<tr>
<td>ReadWriteCopy</td>
<td>2097152</td>
<td>256</td>
<td>31.976s</td>
</tr>
</tbody>
</table>
Avoiding Needless Synchronization

- Merging kernels reduces memory traffic
  - But mind instruction cache size
- Continue executing kernels until you really need to read the results
  - Use in-order queue and blocking call to `clEnqueueMapXXX`
- Merging multiple kernels in a pipeline
Optimization Factors

- Optimize host API calls
- Reduce Host <> Device memory traffic and bandwidth
- **Optimizing memory access**
- Maximizing occupancy
- Maximizing computation
- Kernel algorithm optimization
Optimizing Memory Access

- Accesses to **global** and **constant** memory go through
- L3 cache: GPU-specific, Cache line is 64 bytes
- LLC: CPU and GPU shared
Optimizing Memory Access

- **Local** memory is allocated directly from the L3 cache
- Divided into 16 banks at a 32-bit granularity
- **private** memory that is allocated to registers is very efficient to access

- **Private** memory that spills from registers do the same as **Global** memory
  - The performance in this can be very poor
  - There is no locality for **private memory** accesses
  - each work-item accesses a unique cache line for every access to **private memory**
Global Memory and Constant Memory

- Global, and constant memory bandwidth is determined by the number of the accessed L3 cache lines.
- If two L3 cache lines are accessed from different work items in the same hardware thread, memory bandwidth is \( \frac{1}{2} \) of the memory bandwidth in case when only one L3 cache line is accessed.
- Affected by two factors
  - The access pattern function of the work-item global id(s)
  - The work-group dimensions
**Access pattern** function example:

Workgroup = \(<16,1,1>\)

\[
x = \text{myArray}[\ \text{get\_global\_id}(0) \ ];
\]

\[
x = \text{myArray}[\ \text{get\_global\_id}(0) + 1 \ ]
\]
Global Memory and Constant Memory

```c
size_t localWorkSize[2] = {1, 16};
c1EnqueueNDRangeKernel(..., localWorkSize, ...);
```

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

**Host**

**Kernel**

**Optimization Notice**

Copyright © 2016, Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.

Code snippets provided in this presentation are for illustrative purposes only. Intel disclaims any and all implied or express warranties associated with the code snippets, and any and all use of such code snippets is at the sole discretion and exclusive risk of the user.
Local Memory

- Highly-banked
  - More important to minimize bank conflicts than to minimize the number of L3 cache lines accesses

- Local memory accesses have latencies similar to L3$ hits
  - Using only local memory as a cache is often not productive

- But, local memory and L3$ are organized differently
Private Memory

- Each work item in an OpenCL* kernel has access to up to 512 bytes of register space
- Bandwidth to registers is faster than any memory
- Loading and processing blocks of pixels in registers is very efficient!
  - Example: non-separable convolution (filter2D) in OpenCV*

```c
float sum[PIX_PER_WI_X] = { 0.0f };  
float k[KERNEL_SIZE_X];  
float d[PIX_PER_WI_X + KERNEL_SIZE_X];  
// Load filter kernel in k, input data in d
...
// Compute convolution
for (px = 0; px < PX_PER_WI_X; ++px)
  for (sx = 0; sx < KERNEL_SIZE_X; ++sx)
    sum[px]= mad(k[sx], d[px + sx], sum[px]);
```

Use available registers (up to 512 bytes) instead of memory, where possible!

Achieved up to 5.7X!
Optimization Factors

- Optimize host API calls
- Reduce Host <-> Device memory traffic and bandwidth
- Optimizing memory access
- Maximizing occupancy
- Maximizing computation
- Kernel algorithm optimization
Maximizing Occupancy

- Occupancy is a measure of utilization
- The goal is to keep a sufficient number of work-groups active
  - If one is stalled, another can run on its hardware resource.
Maximizing Occupancy

- Two primary things to consider:
  - Launch enough work items to keep GPU units busy
    - Compiler may pack up to 32 work items per thread (with SIMD-32).
  - Let the kernel do enough work
    - In short kernels: use short vector data types and compute multiple pixels to better amortize thread launch cost
  - Use Vload and Vstore

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

1 pixel per work item

```c
__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);
```

4 pixels per work item
Maximizing Occupancy

- More subtle occupancy issues (when using barriers or local memory):
  - Sub-slices will not run partial workgroups
    - Can be a limiting factor for very large work groups
  - Sub-slices will not run more than 16 (32 on Gen9) work groups
    - Can be a limiting factor for very small work groups
  - Shared Local Memory (SLM) – 64KB SLM per sub-slice
    - Can be a limiting factor for kernels that use a lot of local memory
- General advice when using barriers or local memory
  - Experiment with workgroup sizes of 64, 128, or 256
  - Use less than 64 bytes of local memory per work item
Optimization Factors

- Optimize host API calls
- Reduce Host <> Device memory traffic and bandwidth
- Optimizing memory access
- Maximizing occupancy
- **Maximizing computation**
- Kernel algorithm optimization
Maximizing Compute Performance

- Prefer float over int, if possible
- Trade accuracy for speed, where appropriate
  - Use native_* and built-ins (or use -cl-fast-relaxed-math)
  - Compiler optimization options that enable optimizations for floating-point arithmetic for the whole OpenCL* program:
    - For example: -cl-mad-enable, -cl-fast-relaxed-math

\[ x = \cos(i); \quad \Rightarrow \quad x = \text{native}_\cos(i); \]

*Used to speedup OpenCV* SURF and HOG!*

*Other names and brands may be claimed as the property of others.*

*Copyright © 2016, Intel Corporation. All rights reserved.*
Using OpenCL* 2.0 Workgroup Functions

- The OpenCL* 2.0 standard offers new workgroup built in functions
  - Parallel Primitive – popular parallel primitives (scan, reduction)
    - Operations available: add, min, max
    - Allows reductions and scans without exposed local memory or barriers
  - Broadcast – Transmit data from one work item to all work items within the workgroup
  - Predicate – evaluate a predicate for all work items in a workgroup (any, all)

- Convenient
  - much simpler to use

- Performance efficient
  - Device-specific implementation optimized for the hardware
Using OpenCL* 2.0 Workgroup Functions

- Predicate workgroup functions
  - work_group_reduce_<op>
  - work_group_scan_exclusive_<op>
  - work_group_scan_inclusive_<op>
    - Operations available: add, min, max

```c
__kernel void foo(int *p)
{
    ...
    int prefix_sum_val = work_group_scan_inclusive_add(p[get_local_id(0)]);
    ...
}
```

P = [3 1 7 0 4 1 6 3]  
prefix_sum_val = [3 4 11 11 15 16 22 25]
Example: work_group_reduce_add

```c
__local float smem[256];
unsigned int id = get_local_id(0);
float smem[id] = sum = input;

if (id < 128) smem[id] = sum = sum + smem[id + 128]; barrier(CLK_LOCAL_MEM_FENCE);
if (id < 64) smem[id] = sum = sum + smem[id + 64]; barrier(CLK_LOCAL_MEM_FENCE);
if (id < 32) smem[id] = sum = sum + smem[id + 32]; barrier(CLK_LOCAL_MEM_FENCE);
if (id < 16) smem[id] = sum = sum + smem[id + 16]; barrier(CLK_LOCAL_MEM_FENCE);
if (id < 8) smem[id] = sum = sum + smem[id];
if (id < 4) smem[id] = sum = sum + smem[id];
if (id < 2) smem[id] = sum = sum + smem[id];
if (id < 1) smem[id] = sum = sum + smem[id];
sum = smem[0];

sum = work_group_reduce_add(input);
```

- **No exposed local memory or barriers**
- **Code written independent of workgroup size**
- **Intel optimized for Processor Graphics**
Using OpenCL* 2.0 Workgroup Functions

- Predicate workgroup functions
  - work_group_all()
  - work_group_any()

```c
__kernel void foo (int *in, int *out)
{
    ...
    int gid = get_global_id(0);
    int result = work_group_all(in[gid] < in[gid+1])
    ...
}
```
Develop OpenCL® applications with Intel® SDK for OpenCL™ Applications
OpenCL* is Great!

- However…
  - Development is not trivial
  - Debugging of parallel processing applications is difficult
  - Optimization of OpenCL* applications is platform-dependent and very challenging
Intel® SDK for OpenCL™ Applications

The OpenCL® development environment for Intel® based platforms

Available for free
Intel® Code Builder for OpenCL™ API

Comprehensive development environment for the build, debug, and analysis of an OpenCL* applications

- **BUILD AND CREATE**
  - JumpStart Kit
    - A wizard for creating an OpenCL* project
  - Kernel Development Framework (KDF)
    - Stand alone environment for write, compile and run kernels
Intel® Code Builder for OpenCL™ API

Comprehensive development environment for the build, debug, and analysis of an OpenCL* applications

- **DEBUG**
  - Seamless debugging tool for OpenCL* applications
    - OpenCL* API debugger for host side debugging
    - OpenCL* Kernel Debugger for device side debugging
Intel® Code Builder for OpenCL™ API

Comprehensive development environment for the build, debug, and analysis of an OpenCL* applications

- **ANALYZE**
  - Easy and simple performance debugging tool
  - Collect performance data from both the host side and the kernel side
Intel® Code Builder for OpenCL™ API

Comprehensive development environment for the build, debug, and analysis of an OpenCL® applications

- Integration
  - A single framework for all the functionality that the developer needs
  - Smooth path between all components
  - IDE native integration
Build and create
Create new OpenCL* Project

- Create OpenCL* project with Jump-Start wizard
  - Very simple wizard for creating new OpenCL* project
  - Intended for developers that write an OpenCL* application from scratch
  - Plug in for Visual Studio*
Create new OpenCL* Project

- In a few clicks you can generate:
  - An empty project ready for you to implement host and kernel code
  - Full host + kernel code project ready for build
Kernel Development Framework (KDF)

- Standalone environment for kernel development
- Syntax checking and auto-completion for OpenCL* C language
- Offline compilation and binary generation of OpenCL* kernels
- Compilation error reports
Kernel Development Framework

Gen assembly <=> OpenCL*-C Line mapping
Kernel Development Framework

- Run and review the results
  - Assign variables to the kernel and check its correctness
  - Show the input and output values

- Capture kernel session from exiting OpenCL* application
  - Store the kernels code with its inputs (buffer or images)

- Coming soon:
  - Generate host code from session
  - Validate kernel outputs versus a reference
Capture & Reply Kernel Sessions
Capture & Reply Kernel Sessions

- Very useful when kernel inputs are not available
  - Created in run-time
  - Output of a previous kernel
- Very useful when the application requires user interaction to execute the kernel
- Eliminates the need to run the application for any kernel change
- **Sobel Kernel**
- Edge detection algorithm
- Discrete differentiation operator, computing an approximation of the gradient of the image intensity function

\[
G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \ast A \quad \text{and} \quad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \ast A
\]

Source Image

horizontal and vertical derivative approximations

\[G = \sqrt{G_x^2 + G_y^2}\]
DEMO – Session Generation

- Generate session from CyberLink Power Director*
  A High Performance Video Editing suite
Debug
Host Level Debugging

- Seamless debugging of OpenCL* API calls, objects, and queues
- Enables monitoring and understanding the OpenCL* environment of an application execution
- OpenCL* API call tracing
- Images and memory objects view
- Extension to the Visual Studio* debugger
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* image 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 selected OpenCL* objects
Host Level Debugging

- **Image view**
  - Show the visualized content of OpenCL* Image objects when hitting the break point
  - Option to see the image content in different stages of the program
  - Channel filter

- **Data view**
  - Show the content of OpenCL* memory objects (buffers + images)
Host Level Debugging

- Object tree view
  - Hierarchical view of all OpenCL* objects in memory
  - Filter objects by type

- Properties view
  - View the properties of the selected OpenCL* objects
Host Level Debugging

- Problems view
  - Look for hints for potential error or warnings during execution
  - Filter for errors/warning
Host Level Debugging

- **Trace view**
  - Show for any executed OpenCL* API:
    - Name and arguments
    - Error code
    - Return value
    - Execution time
  - Filter by errors/success

<table>
<thead>
<tr>
<th>API</th>
<th>Return Value</th>
<th>Error Code</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>clCreateImage([Context [1]. CL_MEM_READ_ONLY... Image [3]]</td>
<td>CL_SUCCESS</td>
<td>10:03:40:720</td>
</tr>
</tbody>
</table>
Host Level Debugging

- Commands queue view
  - Examine commands queue status and their commands’ state
  - Help understand the commands flow thru the various queues during the application
Kernel Level Debugging on the GPU

- Enables source and assembly level debugging on GPU
- Provide all the conveniences of the modern debugger
  - Step-in, break and continue, show variables, switch between threads, etc.
- Enhanced for the specifics of OpenCL
  - Ability to view the content of vector variables like float4, uchar16 etc.
- Remote debugging only (host vs. target)
- GDB based
- Microsoft Visual Studio* 2015 integration
- Supported on Gen9 and above (Beta version) on Windows
Kernel Level Debugging on the GPU
Kernel Level Debugging on the GPU
Kernel Level Debugging on the GPU
Kernel Level Debugging on the GPU

Step by Step source level debugging of the GPU EU threads

Switch between GPU EU threads

Step by step kernel source-level debugger

Local and Global variables view

Assembly level debugging

Global memory view

Inspect GPU register values
Analyze
Performance Analysis with Code Builder

- 2 ways for performance analysis with the Code Builder
  - Kernel level analysis only with the Kernel Development Framework
  - Full application analysis (host + kernels)
Code Builder – kernel Analysis with KDF

- Kernel Development Framework enable a standalone environment for performance analysis of kernels
  - Enables What-if analysis
  - Provides a lot of performance data:
    - Throughput
    - Memory bandwidth
    - GPU utilization
    - Occupancy
    - Latency for memory operation
DEMO – Performance Analysis with KDF

- Generate session from CyberLink Power Director*
  A High Performance Video Editing suite

*Other names and brands may be claimed as the property of others.
DEMO – Case Study

- Optimization of Sobel Kernel
  - Uchar -> Uchar16
  - Int -> float

\[
G_x = \begin{bmatrix}
  +1 & 0 & -1 \\
  +2 & 0 & -2 \\
  +1 & 0 & -1
\end{bmatrix} \ast A \quad \text{and} \quad G_y = \begin{bmatrix}
  +1 & +2 & +1 \\
  0 & 0 & 0 \\
  -1 & -2 & -1
\end{bmatrix} \ast A
\]

Source Image

horizontal and vertical derivative approximations

\[
G = \sqrt{G_x^2 + G_y^2}
\]
Code Builder - Full Application Analysis

- Guided performance debugging and source level analysis capabilities
  - a “wizard-like” profiling tool with runtime hints and drill down analysis (host to kernel)
  - Command line tool
  - Fully integrated to Visual Studio*

*Other names and brands may be claimed as the property of others.
DEMO – Full Application Analysis with Code Builder

Target application: Optical Flow (OpenCV* implementation)

Given a set of points in an image > find those same points in another image
Target application: Optical Flow (OpenCV* implementation)

Given a set of points in an image > find those same points in another image

- Can be used to:
  - Find an object from one image in another
  - Determine how an object/camera moved
  - Resolve depth from a single camera.
  - More..

- Use a lot of OpenCL* kernels
OpenCL™ Command Queue View
## SVM Usage Info

### GPU OpenCL Info
- **Version:** OpenCL 1.2.0
- **Max Compute Units:** 24
- **Max Work Group Size:** 256
- **Local Memory:** 64 KB
- **SVM Capabilities:** Fine-grained buffer with atomics

<table>
<thead>
<tr>
<th>Grouping</th>
<th>Computing Task Purpose / Computing Task (GPU) / Instance</th>
<th>Work Size</th>
<th>Total Time</th>
<th>Average Time</th>
<th>Instance Count</th>
<th>SIMD Width</th>
<th>SVM Usage Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReadWriteCopy_NoAlignPartWrite</td>
<td></td>
<td>2097152</td>
<td>256</td>
<td>499.664s</td>
<td>0.038s</td>
<td>13,005</td>
<td></td>
</tr>
<tr>
<td>ReadWriteCopy</td>
<td></td>
<td>2097152</td>
<td>256</td>
<td>133.350s</td>
<td>0.157s</td>
<td>849</td>
<td></td>
</tr>
<tr>
<td>ReadWriteCopyUnRoll</td>
<td></td>
<td>2097152</td>
<td>256</td>
<td>34.491s</td>
<td>0.041s</td>
<td>850</td>
<td></td>
</tr>
<tr>
<td>ReadOnly</td>
<td></td>
<td>2097152</td>
<td>256</td>
<td>34.422s</td>
<td>0.020s</td>
<td>1,700</td>
<td></td>
</tr>
<tr>
<td>ReadWriteCopy</td>
<td></td>
<td>2097152</td>
<td>256</td>
<td>32.593s</td>
<td>0.038s</td>
<td>850</td>
<td></td>
</tr>
<tr>
<td>ReadWriteCopy</td>
<td></td>
<td>2097152</td>
<td>256</td>
<td>31.976s</td>
<td>0.038s</td>
<td>850</td>
<td></td>
</tr>
</tbody>
</table>

- Fine-Grained Buffer
- Coarse-Grained Buffer

[Image of the table with highlighted rows and columns]
Gaussian Blur

Naïve implementation

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

```c
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];
        }
    }
    dst_val += src_val;
    dst[x, y] = dst_val;
}
```

* Code source by Intel
What Can We Learn from VTune?

```c
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

EUStalled ~ 0.2 => EUs are waiting 20% of the time
Gaussian Blur: Can We Do Faster?

Use memory buffers instead of images

- Memory buffers are faster to access than Sampler

Take advantage of Gaussian Blur’s separability property

- Two kernels (instead of one):
  - Horizontal pass
  - Vertical pass
Gaussian Blur: Two Passes

Two passes give 21 ms of device time instead of 30 ms!
EUActive increased from 0.8 to 0.9

Code source by Intel
Gaussian Blur: Two Passes

<table>
<thead>
<tr>
<th>Computing Task Purpose / Computing</th>
<th>Work Size</th>
<th>Computing Task</th>
<th>Data Transf</th>
<th>EU Array</th>
<th>Untyped Memory Bandwidth, GB/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>Local</td>
<td>Total</td>
<td>Average</td>
<td>Inst.</td>
<td>Size</td>
</tr>
<tr>
<td>Transfer</td>
<td></td>
<td>0.001ms</td>
<td>0.001ms</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>Compute</td>
<td></td>
<td>20.472ms</td>
<td>10.236ms</td>
<td>2</td>
<td>32</td>
</tr>
<tr>
<td><em>gaussian_blur_vert_1</em></td>
<td>2048 x 2048</td>
<td>10.619ms</td>
<td>10.619ms</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td><em>gaussian_blur_hor_1</em></td>
<td>2048 x 2048</td>
<td>9.853ms</td>
<td>9.853ms</td>
<td>1</td>
<td>32</td>
</tr>
</tbody>
</table>

EU <-> L3 memory bandwidth is far from its peak value (~37 Gb/s vs. 150 Gb/s)
## Gaussian Blur Optimization Steps

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time, ms</th>
<th>EUActive</th>
<th>EUStalled</th>
<th>EUIdle</th>
<th>L3 Reads, Gb/s</th>
<th>L3 Writes, Gb/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>30</td>
<td>0.78</td>
<td>0.21</td>
<td>0.014</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Hor Pass Simple</td>
<td>9.9</td>
<td>0.89</td>
<td>0.10</td>
<td>0.019</td>
<td>30</td>
<td>1.5</td>
</tr>
<tr>
<td>1 pixel per work-item</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vert Pass Simple</td>
<td>11</td>
<td>0.85</td>
<td>0.091</td>
<td>0.059</td>
<td>20</td>
<td>1.5</td>
</tr>
<tr>
<td>1 pixel per work-item</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total*</td>
<td>21.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hor Pass</td>
<td>5.6</td>
<td>0.89</td>
<td>0.094</td>
<td>0.015</td>
<td>54</td>
<td>3.0</td>
</tr>
<tr>
<td>4 pixels per work-item</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vert Pass</td>
<td>5.8</td>
<td>0.68</td>
<td>0.68</td>
<td>0.006</td>
<td>25</td>
<td>2.9</td>
</tr>
<tr>
<td>4 pixels per work-item</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total*</td>
<td>13.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hor Pass</td>
<td>4.6</td>
<td>0.93</td>
<td>0.064</td>
<td>0.006</td>
<td>68</td>
<td>7.3</td>
</tr>
<tr>
<td>8 pixels per work-item</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vert Pass</td>
<td>4.9</td>
<td>0.96</td>
<td>0.038</td>
<td>0.003</td>
<td>53</td>
<td>6.7</td>
</tr>
<tr>
<td>8 pixels per work-item</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total*</td>
<td>10.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sum of kernels duration + time between the kernels

Performance data where collected on Intel® 4th Generation Intel® Core™ Processor with Intel® HD Graphics 5000