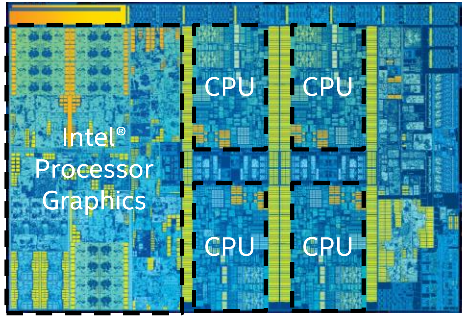
INTEGRATING OPENCL™ KERNELS INTO THE PROGRAM FLOW

Alexei Katranov

IWOCL '16, April 21, 2016, Vienna, Austria

Hardware: customization, integration, heterogeneity



Multicore CPU + integrated units for graphics, media and compute

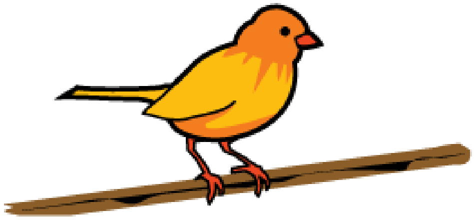
Discrete co-processors and accelerators

FPGAs, fixed function devices, domain-specific compute engines, etc…

Diverse and heterogeneous environments with multiple compute resources

Intel® Threading Building Blocks (Intel® TBB)

- Widely used C++ template library
- Rich feature set for general purpose parallelism
- For Windows*, Linux*, OS X*, Android*, etc.
- Both commercial and open-source licenses
- Commercial support for Intel® Atom™, Core™, Xeon® processors, and for Intel® Xeon Phi™ coprocessors
- Community contributions for non-Intel architectures



http://software.intel.com/intel-tbb

http://threadingbuildingblocks.org

Optimization Notice

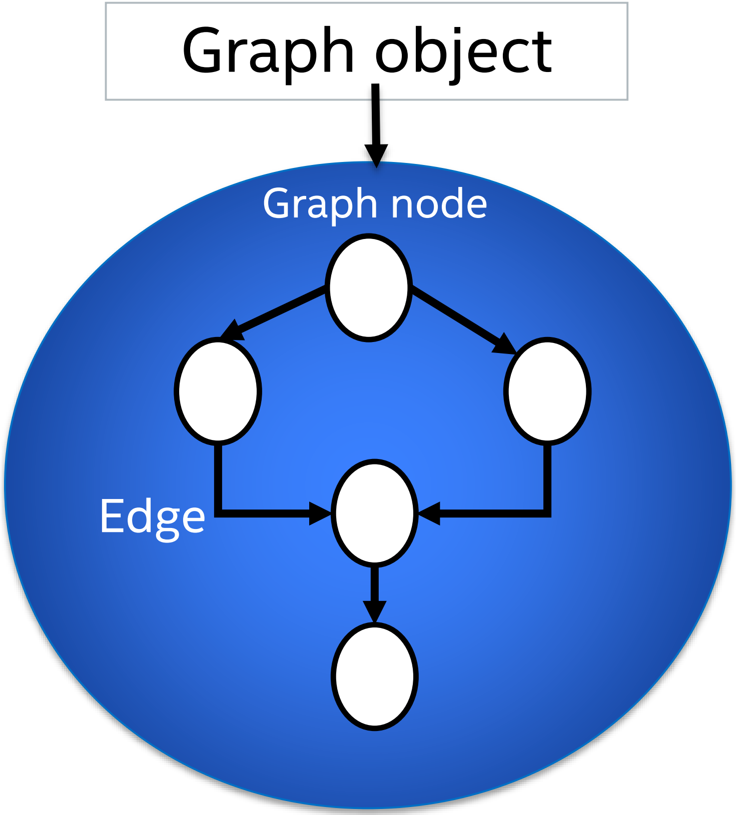
Rich Feature Set for Parallelism

- Parallel algorithms and data structures
- Threads and synchronization
- Memory allocation and task scheduling

Generic Parallel Algorithms	Efficient scalable way to exploit the power of multi-core without having to start from scratch
Flow Graph	A set of classes to express parallelism as a graph of compute dependencies and/or data flow
Concurrent Containers	Concurrent access, and a scalable alternative to serial containers with external locking
Synchronization Primitives	Atomic operations, a variety of mutexes with different properties, condition variables
Task Scheduler	Sophisticated work scheduling engine that empowers parallel algorithms and the flow graph
Thread Local Storage	Unlimited number of thread-local variables
Threads	OS API wrappers
Miscellaneous	Thread-safe timers and exception classes
Memory Allocation	Scalable memory manager and false-sharing free allocators

Intel TBB Flow Graph at glance

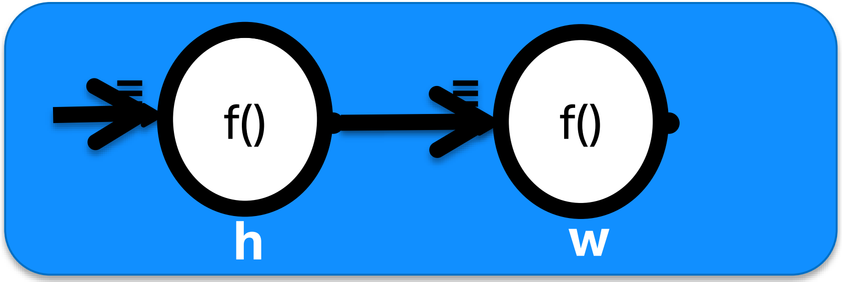
- Intel TBB Flow Graph is an abstraction built on top of TBB task scheduler API
  - Like an additional programming model
  - Explicitly defined control and data dependencies between computations
  - Parallelism is automatically extracted
- Intel TBB flow graph is targeted to multicore shared memory systems.



Hello World Example

Users create nodes and edges, interact with the graph and wait for it to complete

```
tbb::flow::graph g;
tbb::flow::continue_node< tbb::flow::continue_msg >
  h( g, []( const continue_msg & ) { std::cout << "Hello "; } );
tbb::flow::continue_node< tbb::flow::continue_msg >
  w( g, []( const continue_msg & ) { std::cout << "World\n"; } );
tbb::flow::make_edge( h, w );
h.try_put(continue_msg());
g.wait_for_all();
```

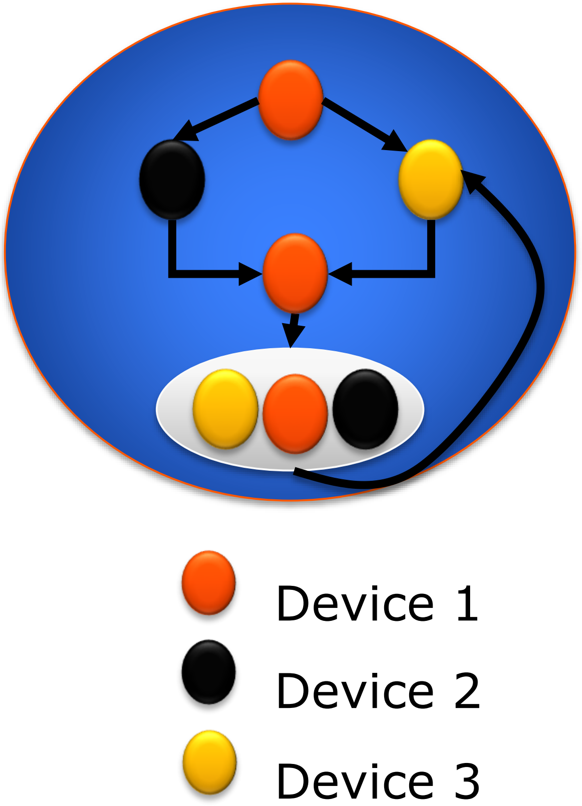


Optimization Notice

COMBINING OPENCL™ AND INTEL TBB

Idea of Heterogeneous Flow Graph

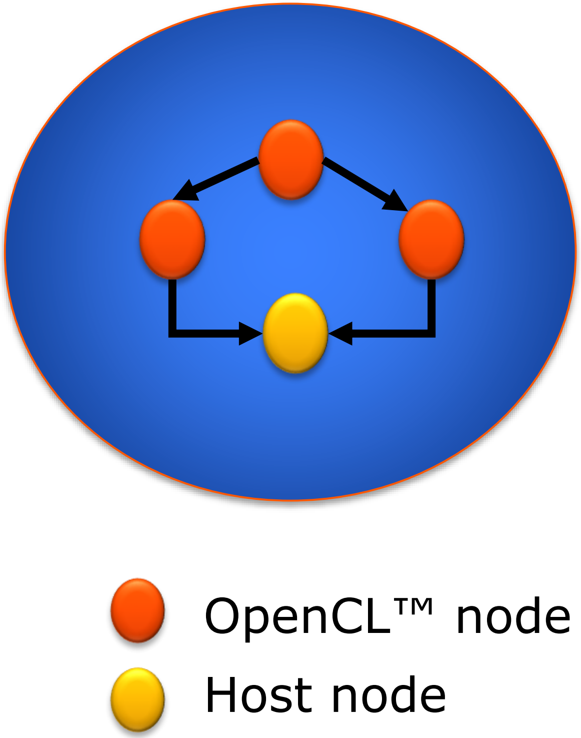
- TBB flow graph as a coordination layer
- Be the glue that connects hetero HW and SW IP together
- Expose parallelism between blocks; simplify integration



OpenCL™ node

Core functionality:

- enumerate & query OpenCL™ devices
- select a device to be used for program execution
- transfer data to/from the device
- execute a given kernel there
- support efficient kernel chaining (no excessive data transfer)



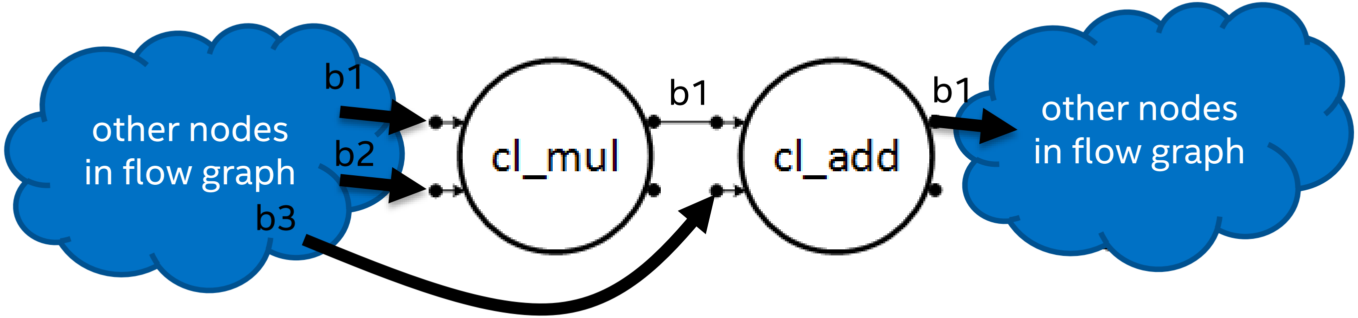
OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.

Hello World example for OpenCL node

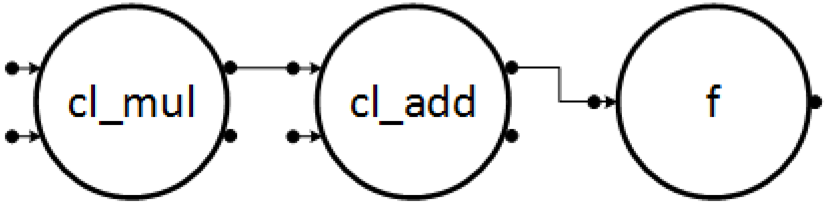
```
// A graph with OpenCL support.
opencl_graph g;

const char str[] = "Hello, World!";
// OpenCL buffer for the string
opencl_buffer<cl_char> b(g, sizeof(str));
// Copy the string to the buffer
std::copy_n(str, sizeof(str), b.begin());

// A node that outputs the content of an incoming buffer
opencl_node<tuple<opencl_buffer<cl_char>>> clPrint(g, "hello_world.cl", "print");
k.set_ndranges({1});

// Send the buffer as the node input
input_port<0>(clPrint).try_put(b);
// Wait for work completeion.
g.wait_for_all();
```

```
// hello_world.cl
kernel void print( global char *str ) {
    printf("OpenCL says '");
    for ( ; *str; ++str ) printf("%c", *str);
    printf("'\n");
}
```

OpenCL node pipeline example

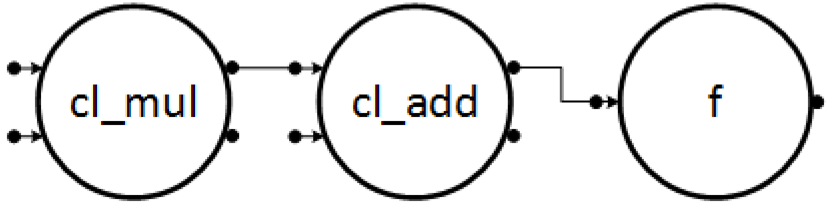


OpenCL node pipeline example




```
typedef opencl_buffer<cl_int> cl_buffer_t;
typedef opencl_node < tuple<cl_buffer_t, cl_buffer_t> > cl_node_t;

// Create nodes
cl_node_t cl_mul( g, "program.cl", "mul" );
cl_node_t cl_add( g, "program.cl", "add" );
function_node_t f( g, unlimited, []( const cl_buffer_t &t ) {...} );

// Create dependencies between nodes
make_edge( cl_mul, cl_add );
make_edge( cl_add, f );

// Put buffers to the graph
cl_buffer_t b1( g, N ), b2( g, N ), b3( g, N );
input_port<0>( cl_mul ).try_put( b1 );
input_port<1>( cl_mul ).try_put( b2 );
input_port<1>( cl_add ).try_put( b3 );
```

Under the hood



Your code

```
cl_node_t cl_mul
```

Real work

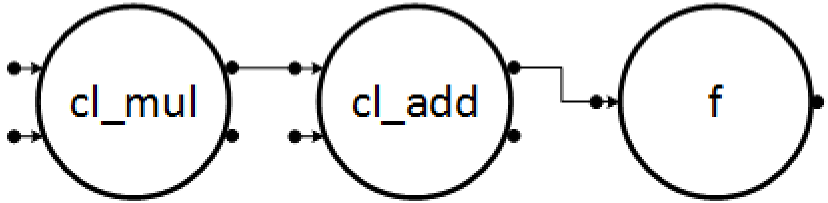
OpenCL intialization

1. Query the available devices
2. Create context
3. Create queue

Create a kernel:

1. Prepare the list of devices
2. Read file
3. Prepare program
4. Build program
5. Print error if observed
6. Get a kernel

Under the hood



Your code

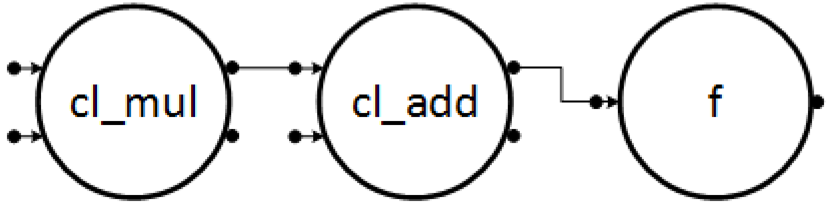
```
cl_node_t cl_mul
```

```
cl_node_t cl_add
```

Real work

- OpenCL intialization
- Create a kernel
- Create a kernel

Under the hood



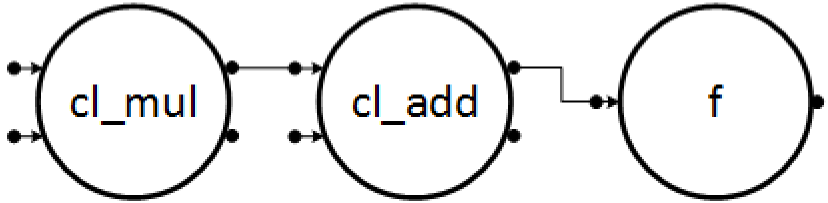
Your code

```
cl_node_t cl_mul
cl_node_t cl_add
cl_buffer_t b1,b2,b3
```

Real work

- OpenCL intialization
- Create a kernel
- Create a kernel
- Create a buffer
- Create a buffer
- Create a buffer

Under the hood



Your code

`cl_node_t cl_mul`

`cl_node_t cl_add`

`cl_buffer_t b1,b2,b3`

`cl_mul<0>.put( b1 )`

Real work

OpenCL init

Kernel x2

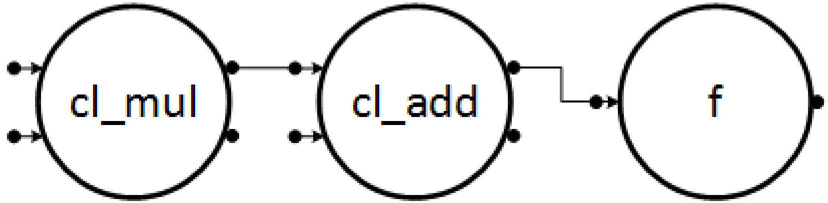
Buffer x3

Send msg

Work in parallel

Move data to device

Under the hood



Your code

. . .

`cl_mul<0>.put( b1 )`

`cl_mul<1>.put( b2 )`

Real work

. . .

Send msg

Send msg

Work in parallel

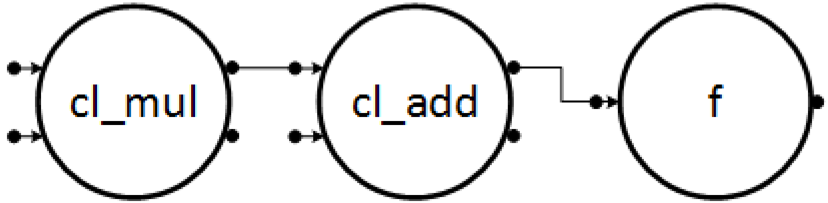
Move data to device

Move data to device

Run kernel "`cl_mul`"

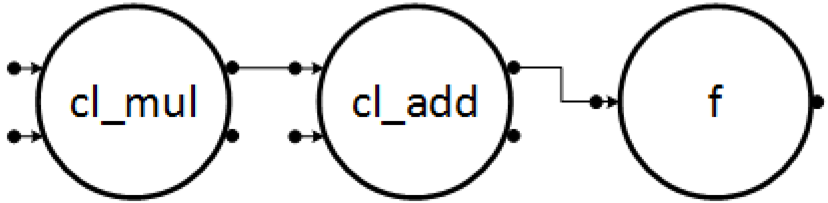
1. Set arguments
2. Enqueue kernel
3. Put "`b1`" to "`cl_add`"

Under the hood



Your code	Real work	Work in parallel
. . .	. . .	
`cl_mul<1>.put( b2 )`	Send msg	Move data to device
		Run kernel "`cl_mul`"
`cl_add<1>.put( b3 )`	Send msg	Move data to device
		Sync with previous kernel "`cl_mul`"
		Run kernel "`cl_add`"

Under the hood



Your code

. . .

`cl_add<1>.put( b3 )`

Real work

. . .

Send msg

Work in parallel

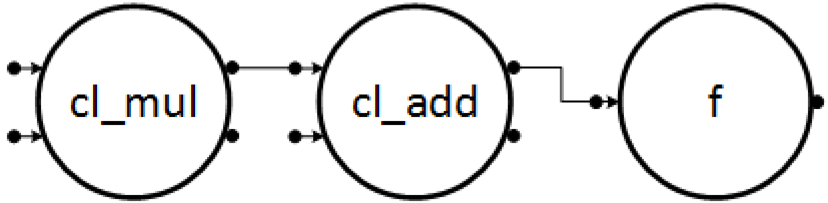
Move data to device

Sync with previous kernel "`cl_mul`"

Run kernel "`cl_add`"

Sync with previous kernel "`cl_add`"

Under the hood



Your code

. . .

`cl_add<1>.put( b3 )`

Real work

. . .

Send msg

Work in parallel

Move data to device

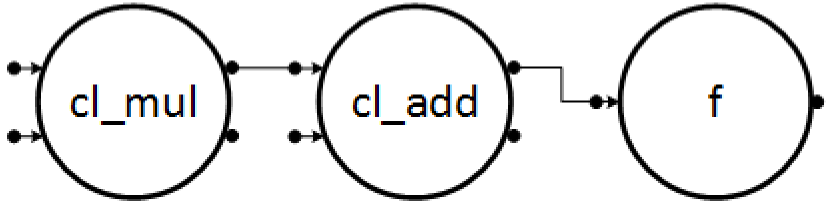
Sync with previous kernel "`cl_mul`"

Run kernel "`cl_add`"

Sync with previous kernel "`cl_add`"

Move data and run "`f`"

Under the hood



Your code

```
cl_node_t cl_mul
cl_node_t cl_add
cl_buffer_t b1,b2,b3
cl_mul<0>.put( b1 )
cl_mul<1>.put( b2 )
cl_mul<2>.put( b3 )
```

Real work

OpenCL intialization

Create a kernel:

1. Prepare the list of devices
2. Read file
3. Prepare program
4. Build program
5. Print error if observed

Create buffer

Send message

Work in parallel

Move data to device

Run kernel "cl_mul"

Run kernel "cl_add"

1. Set arguments
2. Enqueue kernel
3. Put "b1" to "cl_add"

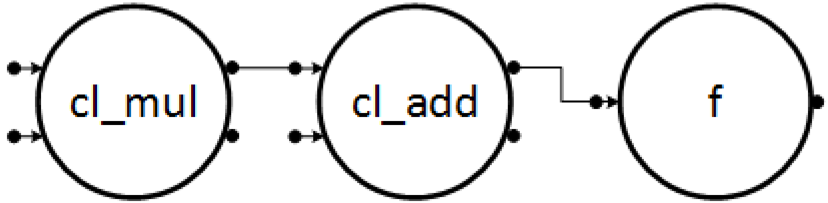
Sync with previous kernel

Run kernel "cl_add"

Sync with previous kernel "cl_add"

Move data and run "f"

Under the hood



Your code

```
cl_node_t cl_mul
cl_node_t cl_add
cl_buffer_t b1,b2,b3
cl_mul<0>.put( b1 )
cl_mul<1>.put( b2 )
cl_mul<2>.put( b3 )
```

Real work

OpenCL intialization

Create a kernel:

Create

1.

2.

5.

Create b

Send message

Send message

Send message

Work in parallel

Move data to device

ul"

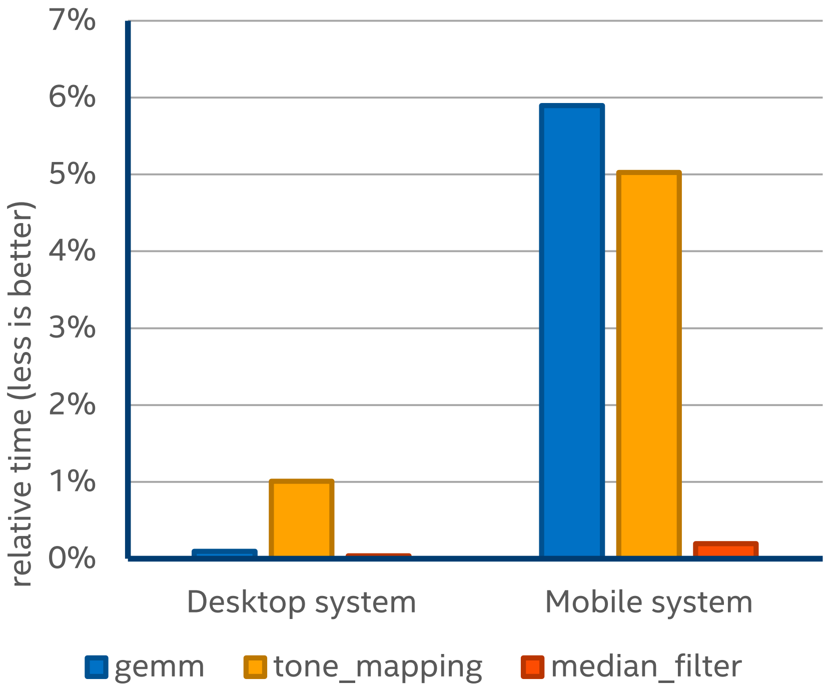
s kernel

Move data and run "f"

Intel TBB work

PERFORMANCE EVALUATION

OpenCL node overheads



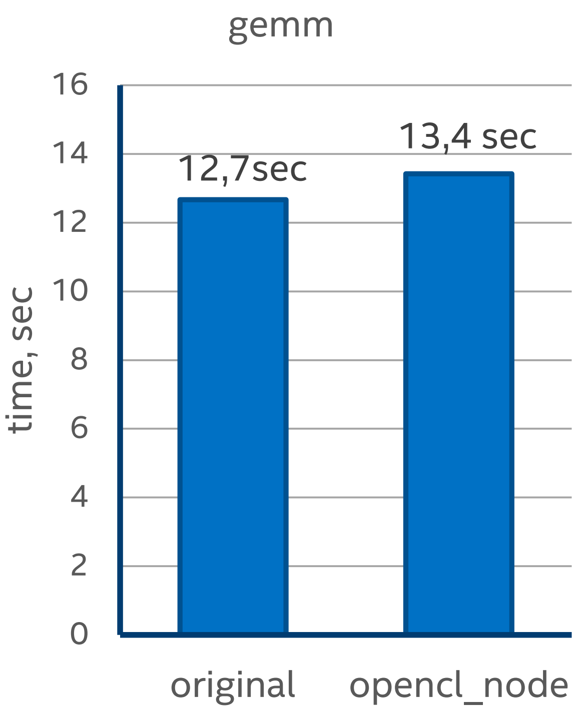
Configuration info:

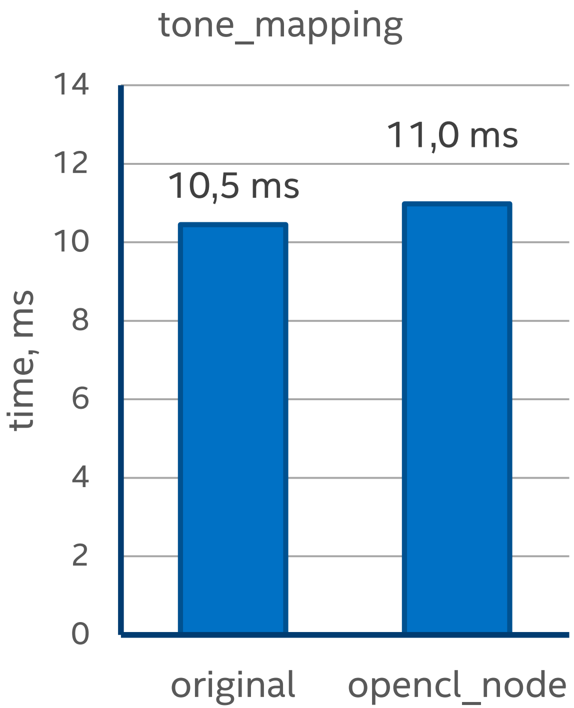
Desktop system: Hardware: Intel® Core™ i7-6700K CPU @4.00Ghz, 16 GB RAM; Software: Microsoft* Windows 10 Enterprise, Microsoft Visual Studio* Professional 2015 Update 1, Intel HD Graphics Driver for Windows 15.40.

Mobile system: Intel Core i5-4300U CPU @1.90Ghz, 8 GB RAM; Software: Microsoft Windows 8.1 Enterprise, Microsoft Visual Studio Professional 2015 Update 2, Intel HD Graphics Driver for Windows 15.36

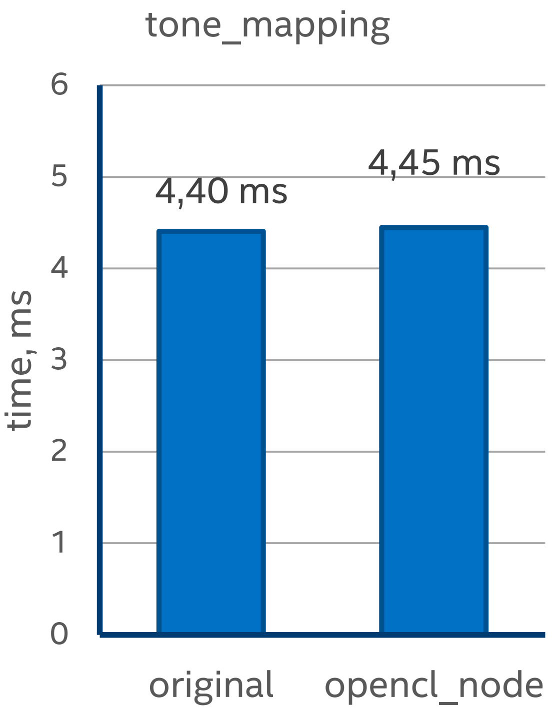
OpenCL node overheads in detail

Mobile system

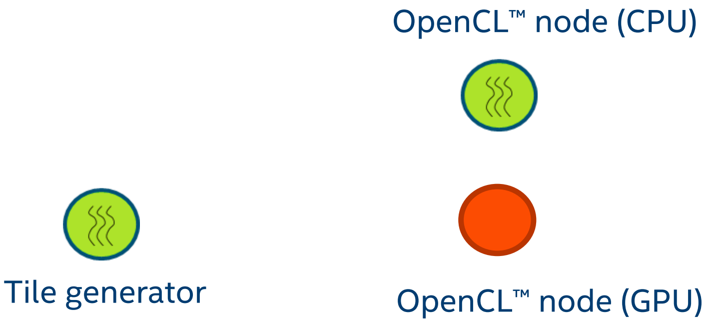




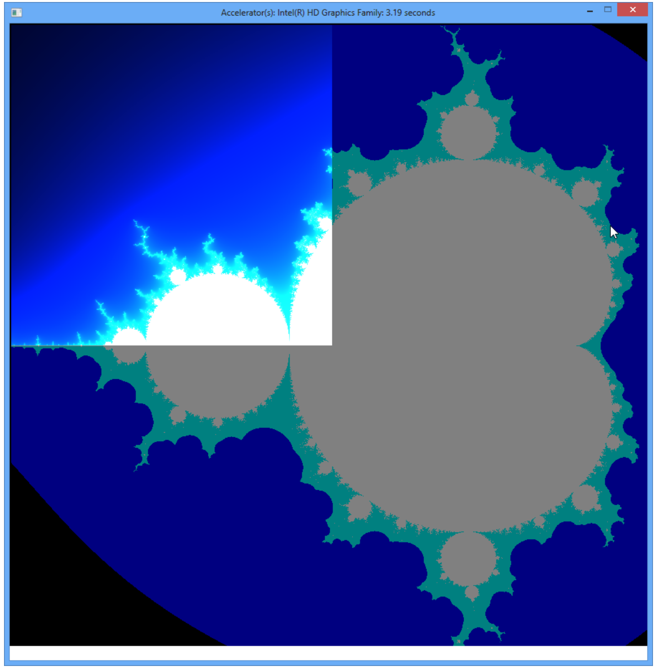
Desktop system



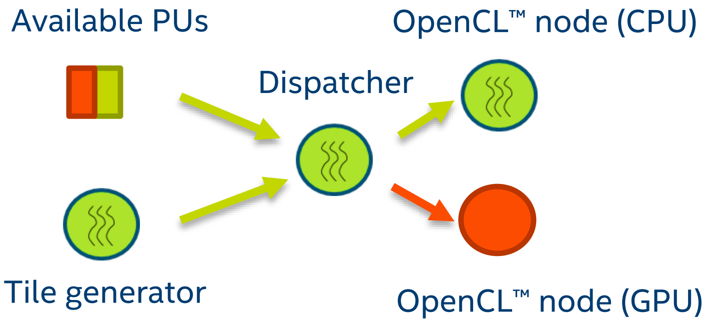
Load balancing CPU and GPU



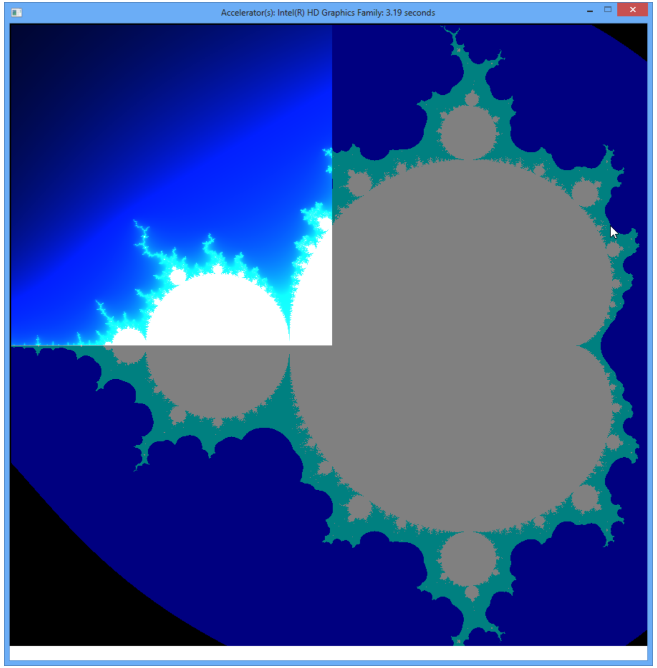
Generic support makes coordinating with any model easier and efficient



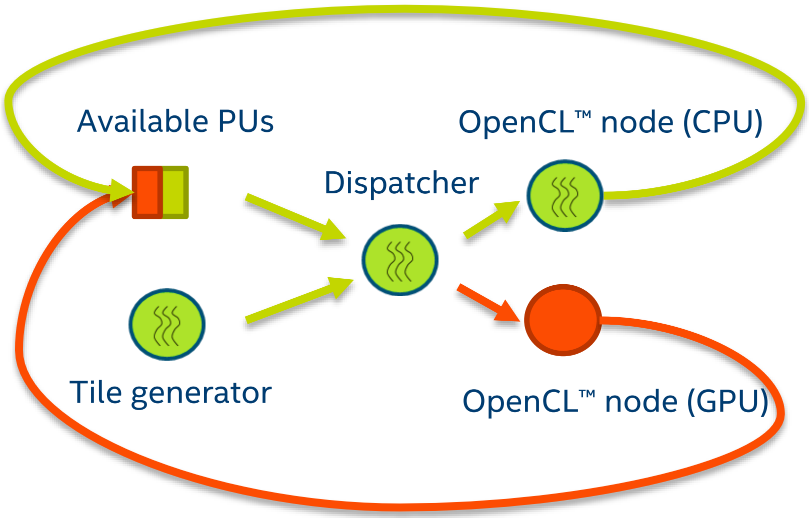
Load balancing CPU and GPU



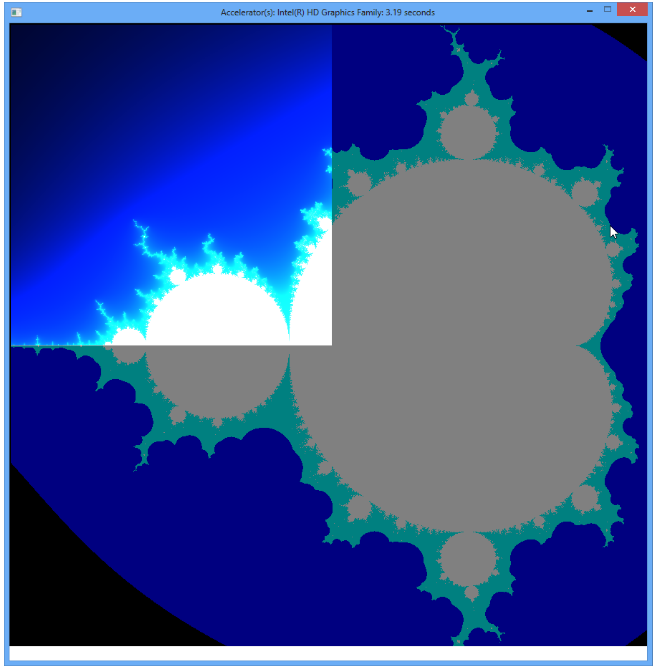
Generic support makes coordinating with any model easier and efficient



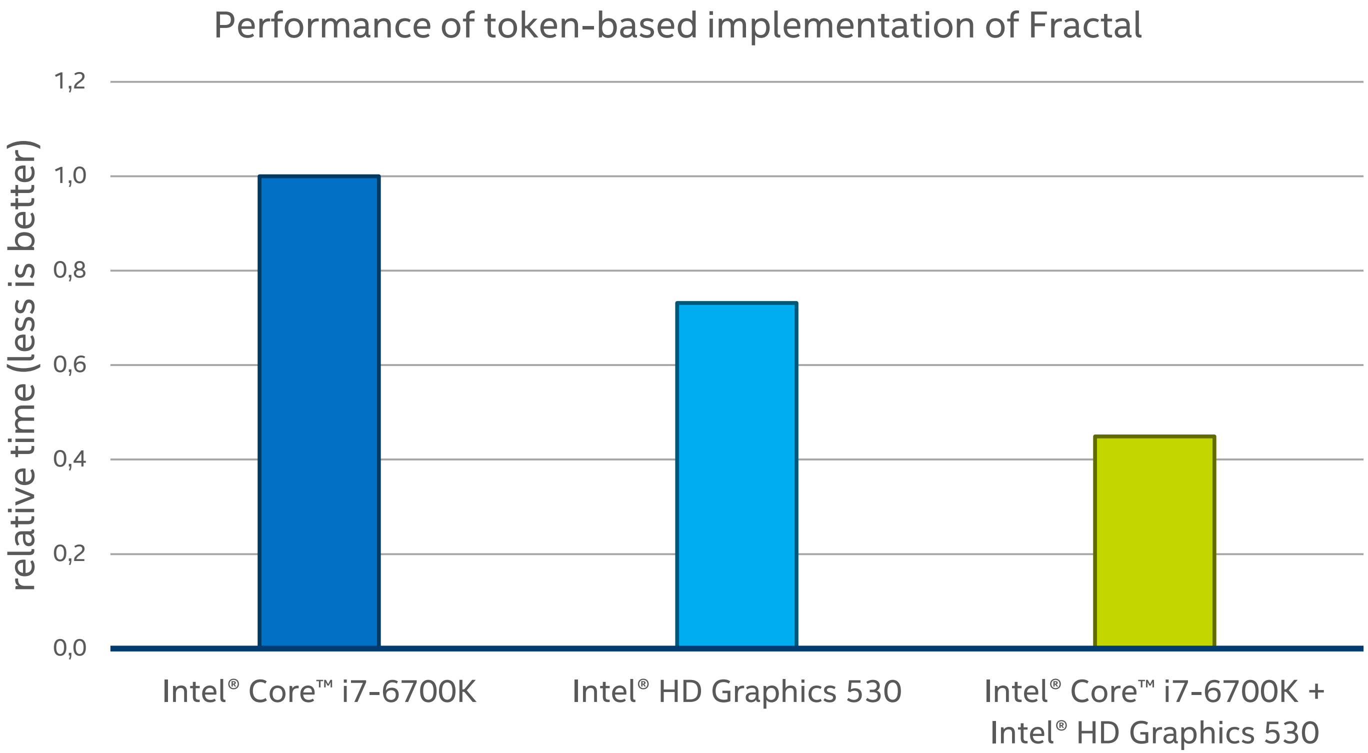
Load balancing CPU and GPU



Generic support makes coordinating with any model easier and efficient



Load balancing CPU and GPU: performance



Optimization Notice

SUMMARY

Summary

Intel TBB flow graph is a coordination layer on heterogeneous systems:

- First class support for OpenCL (opencl_node overview: https://software.intel.com/en-us/blogs/2015/12/09/opencl-node-overview)
- Reasonable performance overheads (about 1% for 4 ms workload on a desktop system)
- Declarative "language" to express unstructured parallelism, e.g. token-based balancing scheme

Intel TBB is open source and freely available on

https://www.threadingbuildingblocks.org/

Acknowledgments

Our thanks to

- Alexey Kukanov for co-authoring and thorough review
- Michael Voss for material contribution to the features described in this presentation
- Robert Ioffe for evaluating our work and providing valuable feedback
- Others who helped in developing the functionality

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

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

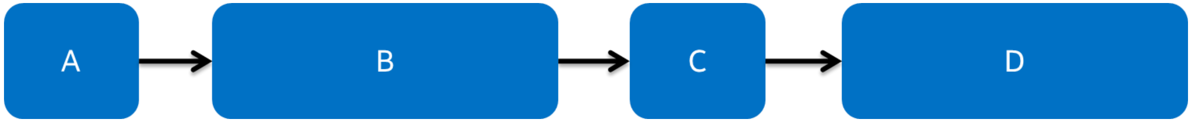
Notice revision #20110804

intel®

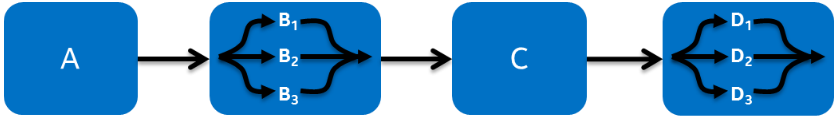
Backup: Motivation for data flow and graph-parallelism

```
x = A();
y = B(x);
z = C(x);
D(y,z);
```

Serial implementation (perhaps vectorized)



Loop-parallel implementation



Loop- and graph-parallel implementation