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
Rich Feature Set for Parallelism

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

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

```cpp
#include <tbb/flow/tbb_flow.h>

int main()
{
    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();
    return 0;
}
```

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

other nodes in flow graph
b3

cl_mul
b1
b2

cl_add
b1
b1

other nodes in flow graph
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**

```c
cl_node_t cl_mul
```

**Real work**

**OpenCL initialization**
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**

1. **OpenCL initialization**
2. **Create a kernel**
3. **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 initialization**
- 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**
- **Buffer**

Send msg

- **x2**
- **x3**

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

```
... 
cl_mul<1>.put( b2 )
cl_add<1>.put( b3 )
```

Real work

```
... 
Send msg
Send msg
```

Work in parallel

```
Move data to device
Run kernel “cl_mul”
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 initialization**

1. Create a kernel:
   - Prepare the list of devices
   - Read file
   - Prepare program
   - Build program
   - Print error if observed

2. Get a kernel

**Work in parallel**

1. Move data to device
2. Run kernel “cl_mul”
   - Set arguments
   - Enqueue kernel
   - Put “b1” to “cl_add”
3. Sync with previous kernel
4. Run kernel “cl_add”
   - Set arguments
   - Enqueue kernel
   - Put “b1” to “cl_add”
5. Sync with previous kernel
6. 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 initialization
- Create a queue:
  1. Prepare the list of 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
- Get a kernel

**Work in parallel**
- Move data to device
- Move data to device
- Move data to device
- Sync with previous kernel
- Run kernel “cl_mul”
- Move data and run “f”
- Sync with previous kernel
- Run kernel “cl_add”
- Set arguments
- Enqueue kernel
- Put “b1” to “cl_add”
- Send message
- Send message
- Send message
- Send message
- Intel TBB work
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

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>OpenCL node</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>gemm</strong></td>
<td>12.7 sec</td>
<td>13.4 sec</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>OpenCL node</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tone_mapping</strong></td>
<td>10.5 ms</td>
<td>11.0 ms</td>
</tr>
</tbody>
</table>

Desktop system

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>OpenCL node</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tone_mapping</strong></td>
<td>4.40 ms</td>
<td>4.45 ms</td>
</tr>
</tbody>
</table>
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

Available PUs

Tile generator

Dispatcher

OpenCL™ node (CPU)

OpenCL™ node (GPU)

Generic support makes coordinating with any model easier and efficient
Load balancing CPU and GPU: performance

Performance of token-based implementation of Fractal

- Intel® Core™ i7-6700K
- Intel® HD Graphics 530
- Intel® Core™ i7-6700K + Intel® HD Graphics 530
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.

Copyright © 2016, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

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

Notice revision #20110804
Backup: Motivation for data flow and graph-parallelism

Serial implementation (perhaps vectorized)

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

Loop-parallel implementation

Loop- and graph-parallel implementation