



Software

OpenMP* and Threading Building Blocks Task Graphs: unraveling the spaghetti with Intel[®] Advisor – Flow Graph Analyzer

Vasanth Tovinkere | Architect, Flow Graph Analyzer

Intel[®] Corporation

What will be covered today

Task-based parallelism and task graphs

- Challenges

Overview of Intel® Advisor - Flow Graph Analyzer (FGA)

Walking through a sample

Summary

Task-based parallelism and task graphs

Task-based parallelism

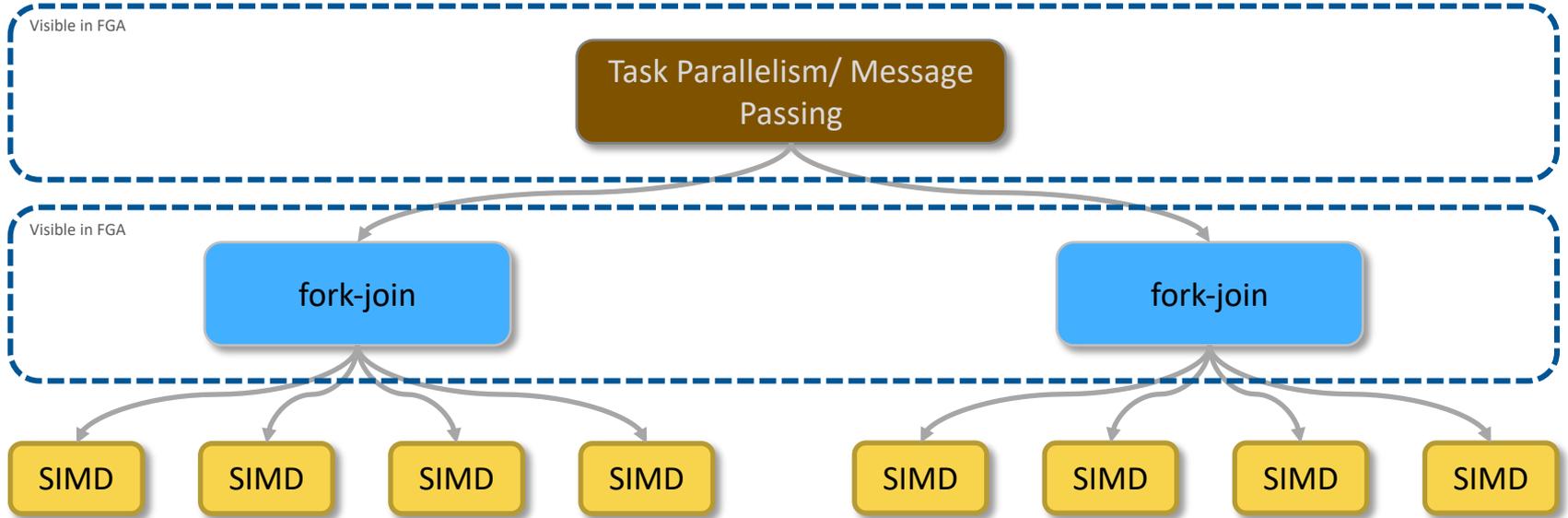
Advantages of task-based parallelism

- Makes parallelization efficient for irregular and runtime dependent execution
- Promotes higher level thinking
- Improves load balancing

Tasks with dependencies

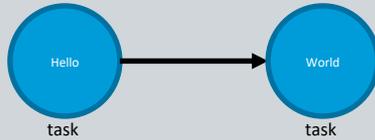
- Fall into two categories: explicit and implicit
- Extends the expressiveness of task-based parallel programming
- Reduces need for global synchronization mechanism such as task barriers

Applications often contain multiple levels of parallelism



Asynchronous task graphs (implicit vs. explicit)

OpenMP*

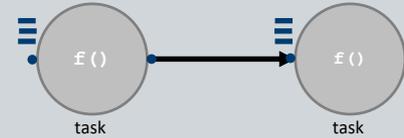


```
#pragma omp parallel
{
  #pragma omp single
  {
    std::string s;
    {
      #pragma omp task depend(out: s)
      {
        s = "Hello ";
        cout << s;
      }
      #pragma omp task depend(out: s)
      {
        s = "World!\n";
        cout << s;
      }
    }
  }
}
```

Implicit

Implicit dependency derived from the depend clause, in this case the variable 's'

Threading Building Blocks (TBB)



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

Explicit

Explicit dependency expressed through the make_edge() call

Challenges with asynchronous task graphs

Creating implicit or explicit task graphs programmatically is easy

- Determining what was created is hard in many cases

New programming paradigm

Allows you to stream data through the graph, which makes debugging challenging

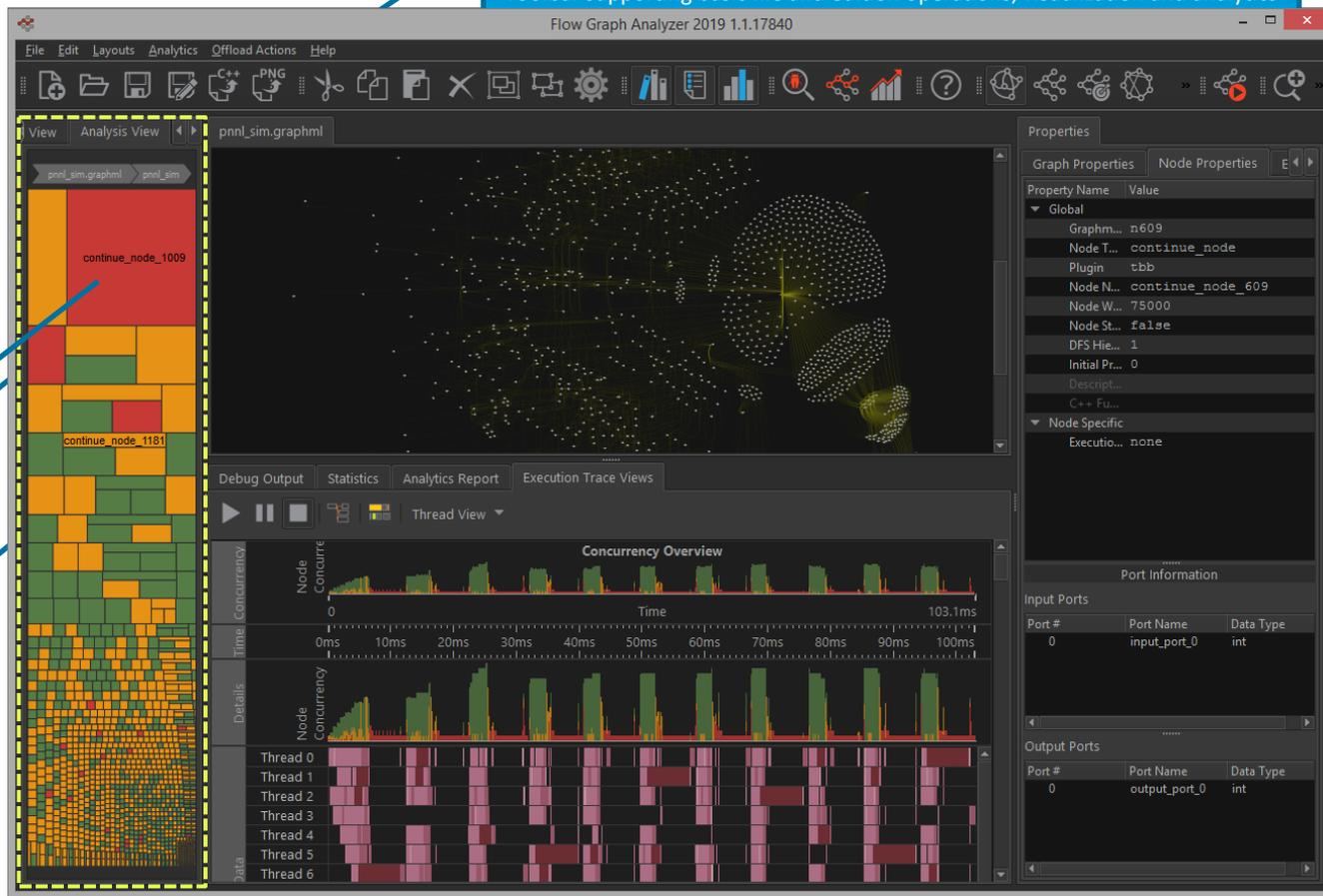
Graph algorithms can be latency-bound or throughput-bound

Parallelism is unstructured in certain types of graphs, so performance analysis can be challenging

Overview of intel[®] Advisor – flow graph analyzer (FGA)

Intel® Advisor – Flow Graph Analyzer

Toolbar supporting basic file and edition operations, visualization and analytics



General health of the graph displayed as a tree-map

The area of the squares represent the CPU time taken by a node as a percentage of the application run and the color indicates the concurrency observed when that node was active

for visualizing graphs

face data, graph generated by
ows interactions
ta

Optimization Notice

Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Overview of intel[®] Advisor – flow graph analyzer

Workflows and UI features

Workflows: Create, Debug, Visualize and Analyze

Design mode

- Allows you to create a graph topology interactively
- Validate the graph and explore what-if scenarios
- Add C/C++ code to the node body
- Export C++ code using Threading Building Blocks (TBB) flow graph API

Analysis mode

- Compile your application (with tracing enabled)
- Capture execution traces during the application run
- Visualize/analyze in Flow Graph Analyzer

Overview of intel[®] Advisor – flow graph analyzer

Creating Asynchronous Task-graphs

Intel® Advisor – Flow Graph Analyzer (Design mode)

Graph Creation

The screenshot displays the Intel Advisor Flow Graph Analyzer in Design mode. The main canvas shows a flow graph with two nodes labeled 'f()' connected by a line. The left sidebar contains a search bar and a tree view with nodes like 'source_node', 'function_node', 'continue_node', and 'multifunction_node'. The bottom-left pane shows a 'Graph' table with columns for Severity, For, Node Name, In-degree, Out-degree, and Total Time (µs). The bottom-right pane shows 'Properties' for the selected node, including 'Graph Name', 'Node Name', 'Plugin', and 'Node Specific' information.

Severity	For	Node Name	In-degree	Out-degree	Total Time (µs)
		Node hello	0	1	1
		Node world	1	0	1

Property Name	Value
Graph Name	n1
Node Name	continue_node
Plugin	tbb
Node Name	hello
Node W...	1
C++ Fu...	[(continue_msg &) ...
Node St...	
Descript...	
Node Specific	
Executio...	none
Initial Pr...	0

Port #	Port Name	Data Type
0	input_port_0	continue_msg

Port #	Port Name	Data Type
0	output_port_0	continue_msg

Drag and Drop Support

Interactive Canvas

Analytics and Modeling

Validation

Code Generation

Optimization Notice

Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.



Intel® Advisor – Flow Graph Analyzer (Design mode)

Serialization

GraphML* file format – uses extensions

```
hello_world.graphml x
35 <<key id="d31" namespace="all::node::tbb::streaming_node" for="node" attr.name="opencl_ndrang
36 <<key id="d32" namespace="all::node::fga::subgraph_node" for="node" attr.name="subgraph_refer
37 <<key id="d33" namespace="all::edge" for="edge" attr.name="graphml_id" attr.type="string"/>
38 <<key id="d34" namespace="all::edge" for="edge" attr.name="edge_name" attr.type="string"/>
39 <<graph id="hello_world_g0" edgedefault="directed">
40 <<data key="d1">2</data>
41 <<data key="d2">1</data>
42 <<node id="n1">
43 <<data key="d3">continue_node</data>
44 <<data key="d16">none</data>
45 <<data key="d5">tbb</data>
46 <<data key="d7">1</data>
47 <<data key="d19">0</data>
48 <<data key="d6">hello</data>
49 <<data key="d12">[( continue_msg &mp; ) { cout && "Hello "; }</data>
50 <<port data_type="continue_msg" name="input_port_0" type="input" offset="0"/>
51 <<port data_type="continue_msg" name="output_port_0" type="output" offset="0"/>
52 <</node>
53 <<node id="n2">
54 <<data key="d3">continue_node</data>
55 <<data key="d16">none</data>
56 <<data key="d5">tbb</data>
57 <<data key="d7">1</data>
58 <<data key="d19">0</data>
59 <<data key="d6">world</data>
60 <<data key="d12">[( continue_msg &mp; ) { cout && "World!\n"; }</data>
61 <<port data_type="continue_msg" name="input_port_0" type="input" offset="0"/>
62 <<port data_type="continue_msg" name="output_port_0" type="output" offset="0"/>
63 <</node>
64 <<edge id="e3" target="n2" source="n1" targetoffset="0" targetport="input_port_0" source
65 <<data key="d34">edge_3</data>
66 <</edge>
67 <</graph>
68 </graphml>
69
```

C/C++ code generated from the graph

```
hello_world.graphml hello_world_stubs.cpp
36
37
38 int build_and_run_hello_world_g0() {
39     graph hello_world_g0;
40     continue_node< continue_msg > hello( hello_world_g0, 0, [( continue_msg & ) { cout << "Hello ";
41     continue_node< continue_msg > world( hello_world_g0, 0, [( continue_msg & ) { cout << "World!\n";
42
43     #if TBB_PREVIEW_FLOW_GRAPH_TRACE
44     hello_world_g0.set_name("hello_world_g0");
45     hello.set_name("hello");
46     world.set_name("world");
47     #endif
48
49     make_edge( hello, world );
50     hello_world_g0.wait_for_all();
51     return 0;
52 }
53
54 int main(int argc, char *argv[]) {
55     return build_and_run_hello_world_g0();
56 }
57
```

Challenges With asynchronous task graphs

Creating implicit or explicit task-graphs programmatically is easy

- Determining what was created is hard in many cases
- ✓ New programming paradigm

Allows you to stream data through the graph, which makes debugging challenging

Graph algorithms can be latency-bound or throughput-bound

Parallelism is unstructured in certain types of graphs, so performance analysis can be challenging

Intel® Advisor – Flow Graph Analyzer (Design mode)

Compiling and collecting traces

Path must be updated so fgtrun.bat and fgt2xml.exe can be run from the command line

```
>c1 hello_world.cpp /O2 /DTBB_USE_THREADING_TOOLS ... /link tbb.lib /OUT:hello_world.exe
```

```
>set FGT_ROOT=<installation-directory>\fga\fgt
```

```
>set INTEL_LIBITNOTIFY64=<installation-directory>\fga\fgt\windows\bin\intel64\<vc-version>\fgt.dll
```

```
>hello_world.exe
```

Traces are saved to a unique directory `_fgt_<date>_<time>`

```
>fgt2xml.exe <name-for-the-trace-data-file>
```

Automatically converts the latest timestamped directory



Overview of intel[®] Advisor – flow graph analyzer

Understanding Graph Execution

Examining the trace data: what's possible?

The screenshot displays the Flow Graph Analyzer interface. The main window shows a flow graph with two nodes labeled 'f()'. A red vertical bar on the left is labeled 'hello' and 'world'. The 'Execution Trace Views' section shows a 'Concurrency Overview' graph and a 'Thread Overview' graph. The 'Code' window shows the source code for 'hello_world_stubs.cpp' with a yellow dashed box highlighting the trace configuration code.

```
85 }
86
87 int build_and_run_hello_world_g0() {
88     continue_node< continue_msg > hello( hello_world_g0, 0, [=]( const continue_msg & ) { cout <<
89     continue_node< continue_msg > world( hello_world_g0, 0, [=]( const continue_msg & ) { cout <<
90
91
92
93     #if TBB_PREVIEW_FLOW_GRAPH_TRACE
94     hello_world_g0.set_name("hello_world_g0");
95     hello.set_name("hello");
96     world.set_name("world");
97     #endif
98
99     make_edge( hello, world );
100     for(int i = 0; i < 1; ++i )
101     continue_node< continue_msg > hello( hello_world_g0, 0, [=]( const continue_msg & ) { cout <<
102     hello_world_g0.wait_for_all();
103     return 0;
104 }
105 }
```

Properties

Property Name	Value
Global	
Graph...	g0::n0
Node T...	

hello_world.graphxml hello_world_stubs.cpp

Debug Output Statistics Analytics Report Execution Trace Views

Concurrency Overview

Thread Overview

0 output_port_0 continue_msg

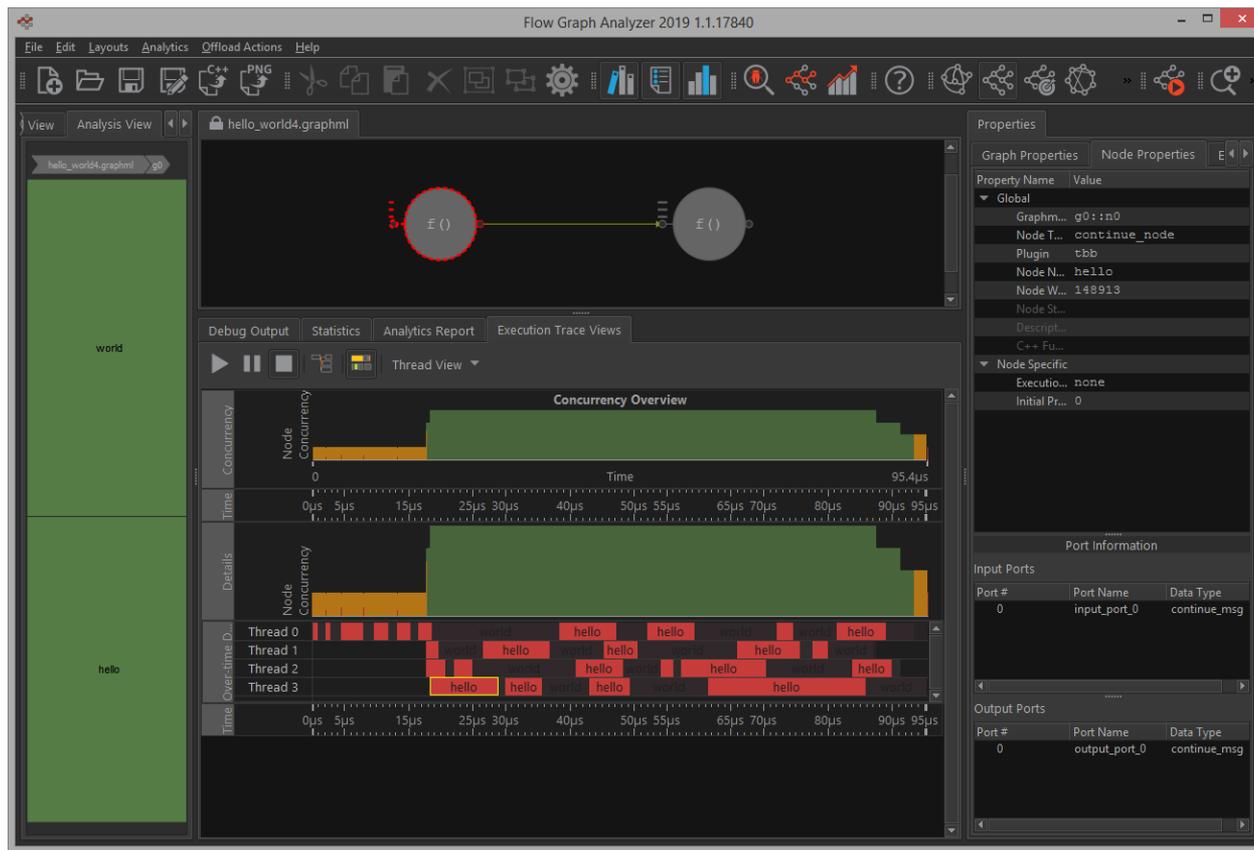
“hello” node in all views that represent different information.

Shows trace information for the case when 1 message is sent to the “hello” node.

Optimization Notice

Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Examining the trace data: correlation



"hello" node in all views that represent different information.

Shows trace information for the case when 25 messages are sent to the "hello" node.

Interacting with the canvas

Clicking on a node on the canvas can highlight the corresponding node's tasks in the timeline. This is turned OFF by default.

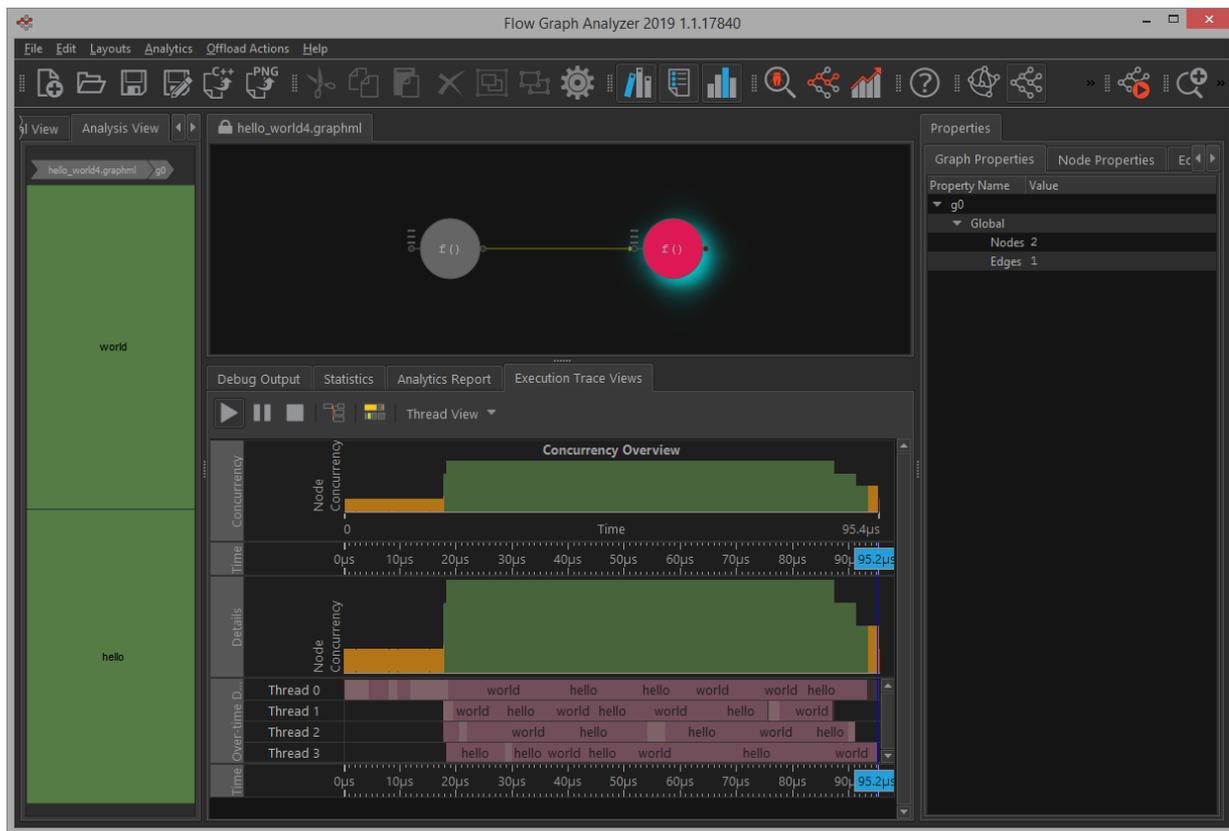
Clicking on a section with low concurrency will highlight the nodes that are active at that time.

These nodes would be the starting point of a cause-and-effect analysis to see if they were responsible for the lower concurrency

Optimization Notice

Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Examining the trace data through Trace Playback



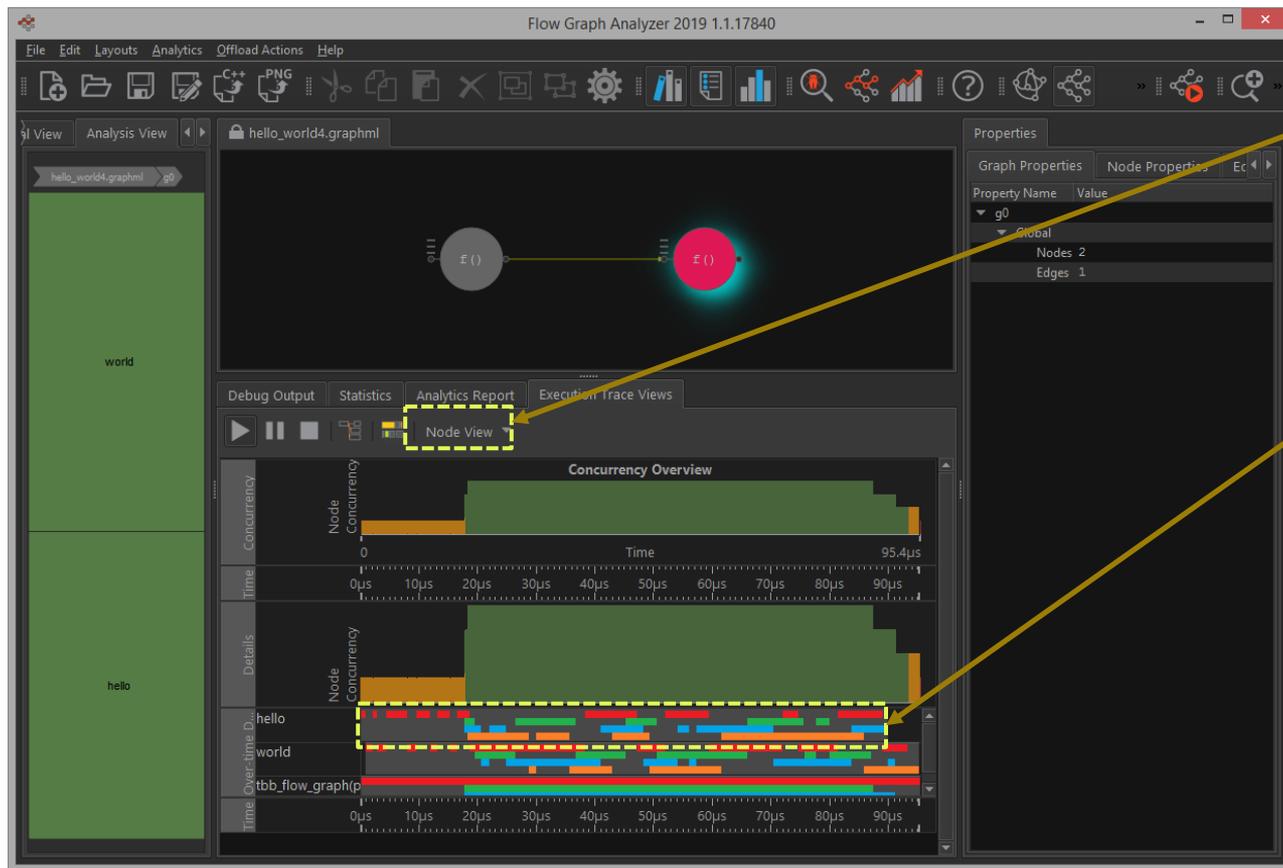
Playback of execution traces to see how data is flowing through the graph.

Allows you to see how the data flows through the graph and what sections of the graph result in good or poor scaling

Optimization Notice

Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Examining the trace data: node view



Node view captures all execution traces for a given node and presents it in a single swim-lane for the node

Each node swimlane is comprised of multiple swimlanes representing the threads which executed an instance of the node.

Provides a compact representation of a node's execution

Challenges With asynchronous task graphs

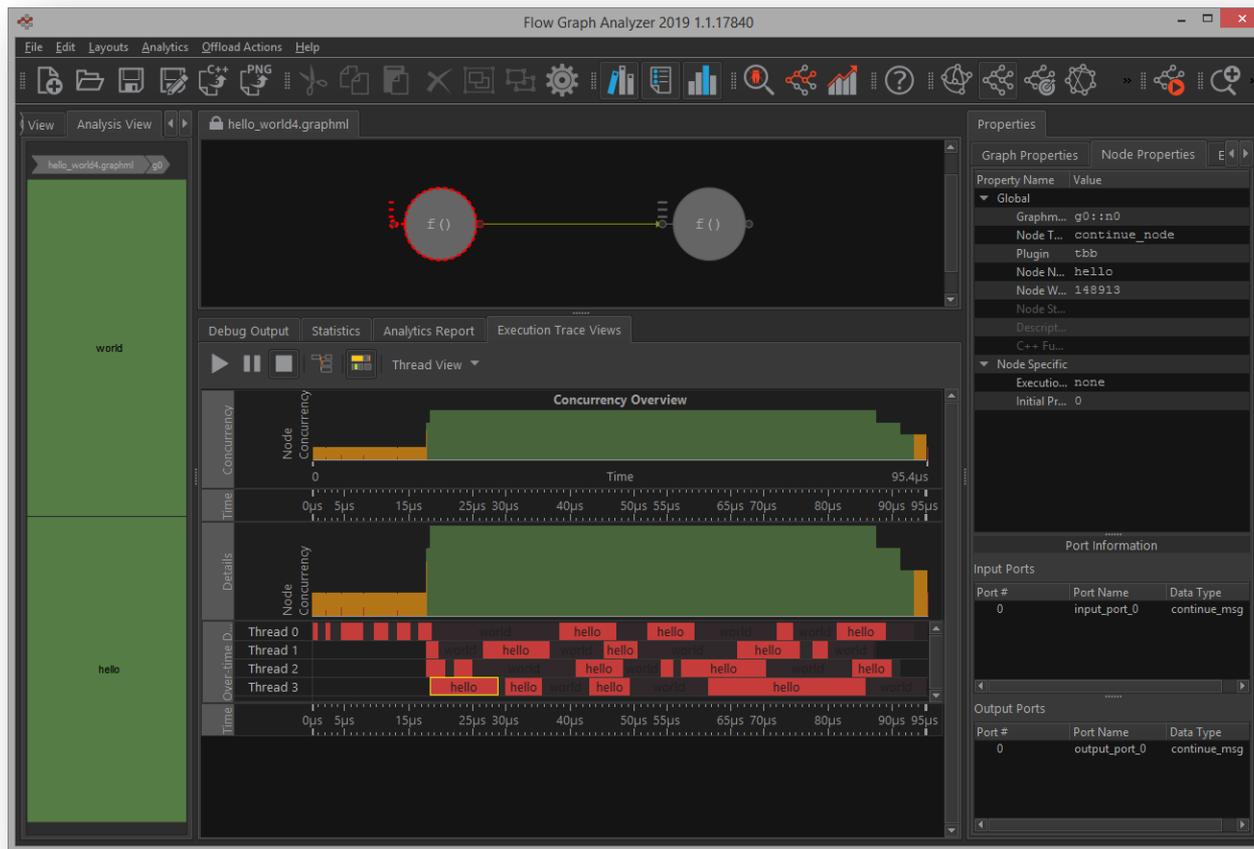
Creating implicit or explicit task-graphs programmatically is easy

- Determining what was created is hard in many cases
- ✓ New programming paradigm
- ✓ Allows you to stream data through the graph, which makes debugging challenging

Graph algorithms can be latency-bound or throughput-bound

Parallelism is unstructured in certain types of graphs, so performance analysis can be challenging

Examining the trace data with data analysis



How do we know which instance of the **Hello** task is in response to which input message?

Helps answer the following questions:
Are the tasks operating on data retiring in order?
Are they out of order?

We need to track the data flowing through the graph

Optimization Notice

Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Examining the trace data with data analysis, cont.

Harder to track the data in dependency graphs as the Data ID cannot be propagated from one node to the next

- **continue_node** requires an input of type **continue_msg**

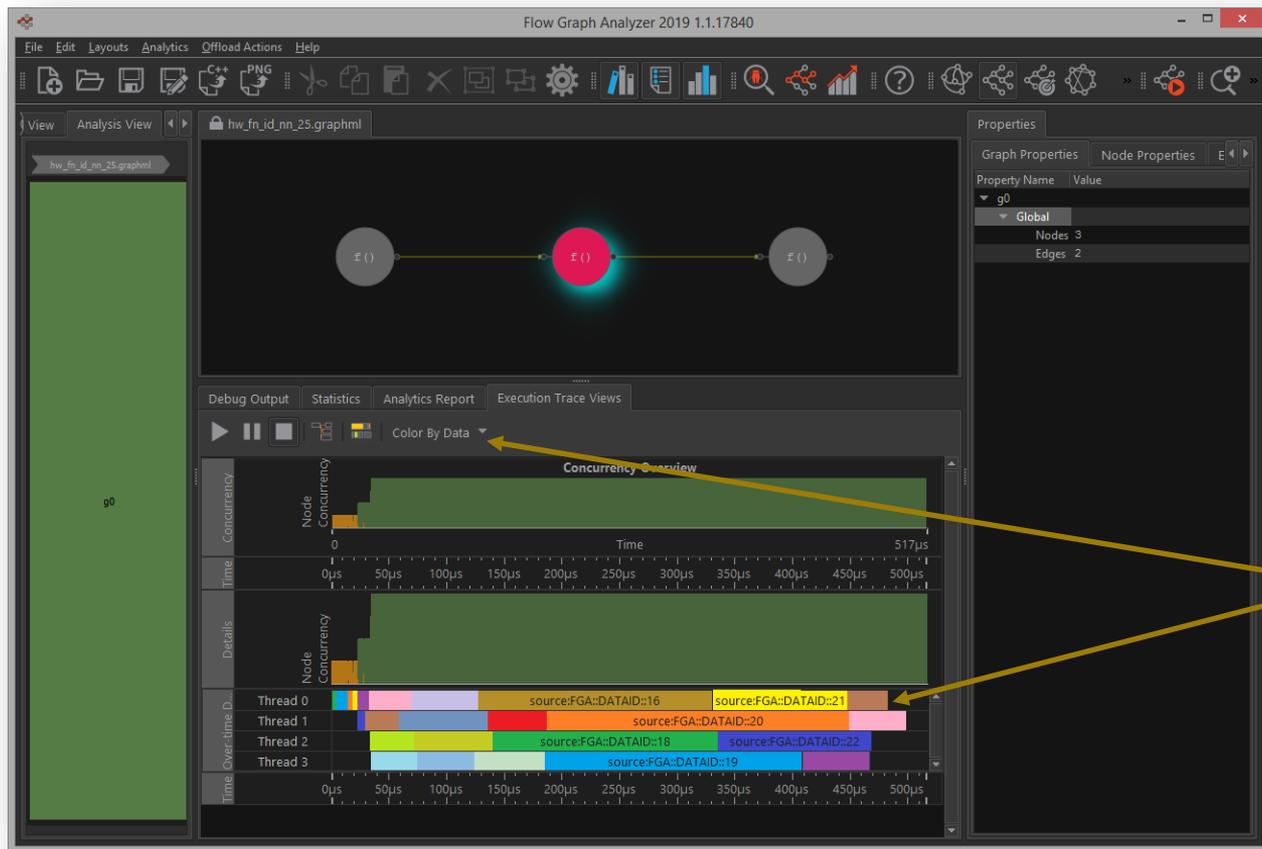
```
continue_node<continue_msg> hello( hello_world_g0, []( continue_msg & ) {
    cout << "Hello ";
} );

continue_node<continue_msg> world(( hello_world_g0, []( continue_msg & ) {
    cout << "World!\n";
} );
```

We are going to convert the Hello World example to use **function_node** instead so we can send the ID from one node to the next

```
+ ..... function_node< int, int > hello( hello_world_g0, 0, [=]( const int &id ) -> int { ...
+ ..... });
+ ..... function_node< int, int > world( hello_world_g0, 0, [=]( const int &id ) -> int { ...
+ ..... });
```

Examining the trace data with data analysis, cont.



Data tracking using an experimental feature will allow you to track which task instance is for which inputs.

1. We changed our graph to use a **function_node** instead of a **continue_node**
2. We have a **source_node** that streams 25 messages/data through the graph
3. We modified the graph to emit the data id from the node **source** to **hello** and **hello** to **world**.
4. We add an user event API to tell the tool which data we are processing in each node.

```
int > hello( hello_world_g0, 0, [=]( const int &id )-> int {  
;  
EM_FLOW_GRAPH_TRACE  
M_MAJOR >= 2019  
iling::event::emit( std::to_string( id ) );  
};
```

Gives you insight into scheduler behavior.

Optimization Notice

Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Examining the trace data with data analysis, cont.

The screenshot shows the Flow Graph Analyzer interface. At the top, there's a menu bar (File, Edit, Layouts, Analytics, Offload Actions, Help) and a toolbar with various icons. The main window displays a graph with three nodes labeled 'f()' connected by edges. Below the graph, there are tabs for 'Debug Output', 'Statistics', 'Analytics Report', and 'Execution Trace Views'. The 'Data Analysis' tab is active, showing a table with columns: Severity, Data #, Node Name, Task Count, Cost (clks), Begin Time (clks), End Time (clks), Duration (clks), Overlap (prev)(%), and Overlap (next)(%). The table lists data for six different data points, each with three results (source, hello, world) and their respective performance metrics.

Severity	Data #	Node Name	Task Count	Cost (clks)	Begin Time (clks)	End Time (clks)	Duration (clks)	Overlap (prev)(%)	Overlap (next)(%)
▼	Data 1	Results(3)	3	4252	3	4266			
		source	1	27	3	30	27	0	0
		hello	1	1340	36	1376	1340	0	0
		world	1	2885	1381	4266	2885	0	0
▼	Data 2	Results(3)	3	10137	4273	14426			
		source	1	9	4273	4282	9	0	0
		hello	1	8518	4291	12809	8518	0	0
		world	1	1610	12816	14426	1610	0	0
▼	Data 3	Results(3)	3	4466	14432	18914			
		source	1	10	14432	14442	10	0	0
		hello	1	2720	14450	17170	2720	0	0
		world	1	1736	17178	18914	1736	0	0
▼	Data 4	Results(3)	3	4500	18921	23435			
		source	1	9	18921	18930	9	0	0
		hello	1	2781	18937	21718	2781	0	0
		world	1	1710	21725	23435	1710	0	0
▼	Data 5	Results(3)	3	7215	22712	28943			
		source	1	7	22712	22719	7	1.03286	0
		hello	1	3550	22728	26278	3550	0.001991549	0
		world	1	3658	26285	29943	3658	0	0
▼	Data 6	Results(3)	3	9574	23445	33032			
		source	1	9	23445	23454	9	7.22	0

Data tracking using an experimental feature will allow you to track which task instance is for which inputs

Statistics for the graph is organized by data operated on and can be seen in Data Analysis tab under Statistics

Using data analysis, the questions posed earlier can be answered.

You can examine the trace data to see if the data is retiring in-order or out-of-order.

If the algorithm is meant to be latency bound, then order is important. If it is throughput bound, data can retire out-of-order

Optimization Notice

Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

Challenges with asynchronous task graphs

Creating implicit or explicit task-graphs programmatically is easy

- Determining what was created is hard in many cases
- ✓ New programming paradigm
- ✓ Allows you to stream data through the graph, which makes debugging challenging
- ✓ Graph algorithms can be latency-bound or throughput-bound

Parallelism is unstructured in certain types of graphs, so performance analysis can be challenging

Walking through an Example

Understanding the performance

A simulation example

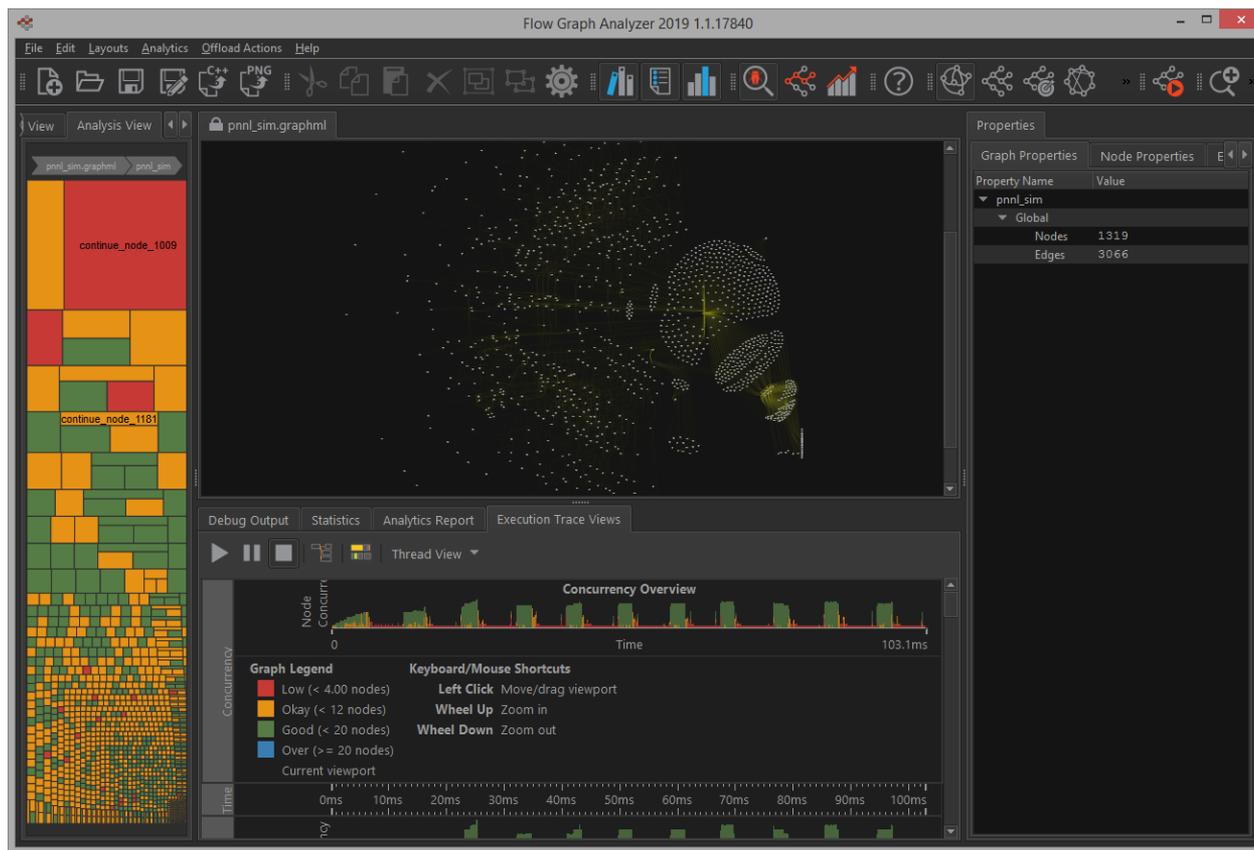
Goes through multiple time steps

Graph is created once programmatically and executed for each time step

- A message is sent to the graph to trigger each time step
- Wait for the graph to process the message (current time step) before the next time step is triggered
- Implemented as a dependency graph using TBB `continue_node`

Measured performance shows some performance scaling w.r.t serial implementation

Example: performance analysis



A complex graph was created programmatically.

Graph has 1319 nodes and 3066 edges.

General health of the graph with a mix of red, yellow and green

Concurrency observed over time ranges from good concurrency where all cores are kept busy to very few kept busy

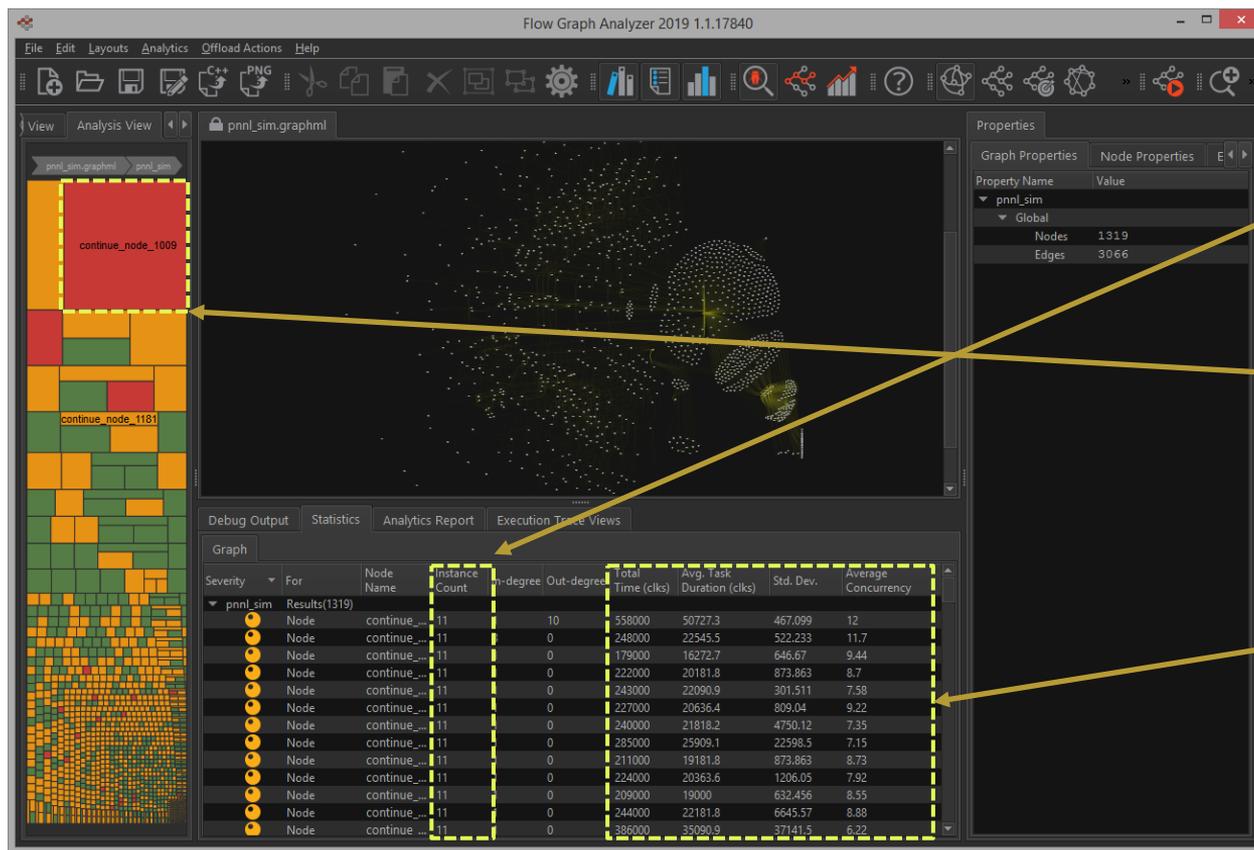
What do the colors mean?

Challenges with asynchronous task graphs

- ✓ Creating implicit or explicit task-graphs programmatically is easy
 - ✓ Determining what was created is hard in many cases
- ✓ New programming paradigm
- ✓ Allows you to stream data through the graph, which makes debugging challenging
- ✓ Graph algorithms can be latency-bound or throughput-bound

Parallelism is unstructured in certain types of graphs, so performance analysis can be challenging

Example: identifying problem areas



What was run and how much was run?

Run captures 11 time steps

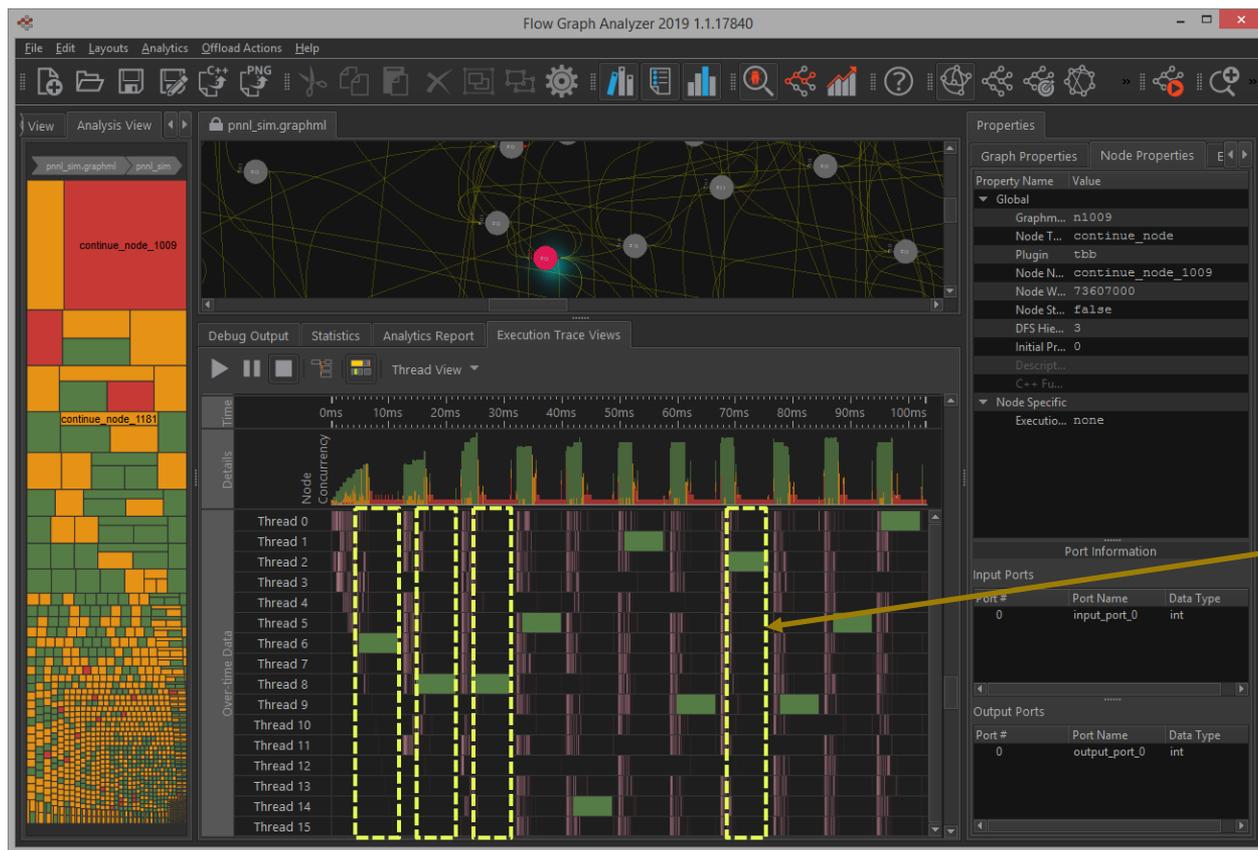
Appears to have one node that consumes a lot of CPU time.

This node also has an observed concurrency that is poor when it executes

Clicking on the node takes you to the node in the graph visualization

You can also sort on the appropriate column in the statistics table.

Example: identifying problem areas, cont.



Clicking on the node takes you to the node in the graph visualization

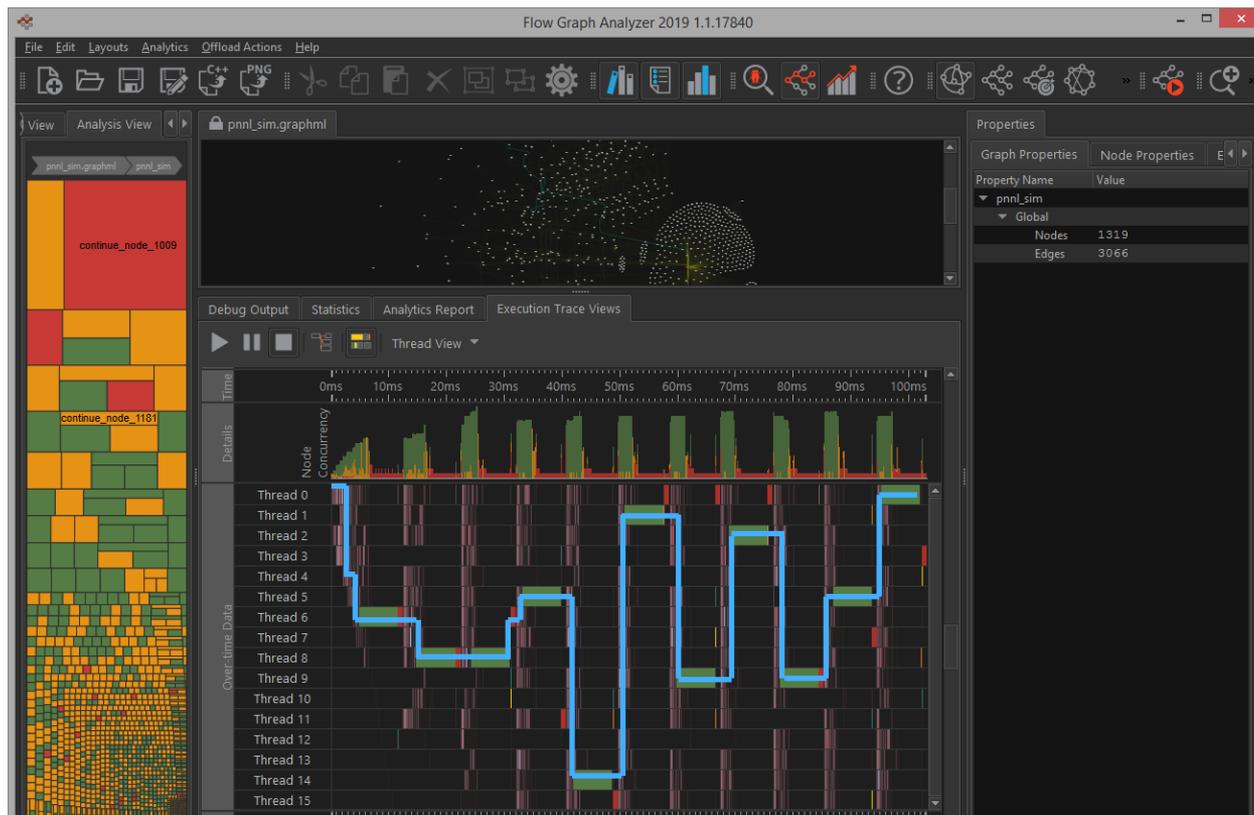
1. To see all tasks belonging to this node in the execution trace, you will have to enable this interaction.
2. Click on the Show/Hide tasks button
3. Now select the node in the canvas

When this node is executing, the resource utilization is very poor.

1. Improving the performance of this one node will substantially improve the performance of the graph.

Optimization Notice

Example: critical path



Analysis features

1. Critical Path
2. Rule-check

Critical Path

Computes the Critical Path(s) for the graph using the execution trace information

The most dominant task that had the maximum CPU Time and a corresponding low concurrency (continue_node_1009) is on this critical path

Critical path reduces the complexity in large graphs by isolating a small set of nodes for analysis and tuning for performance improvements

Optimization Notice

Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

We found the problem Node: Fix it!

What else can we look at?

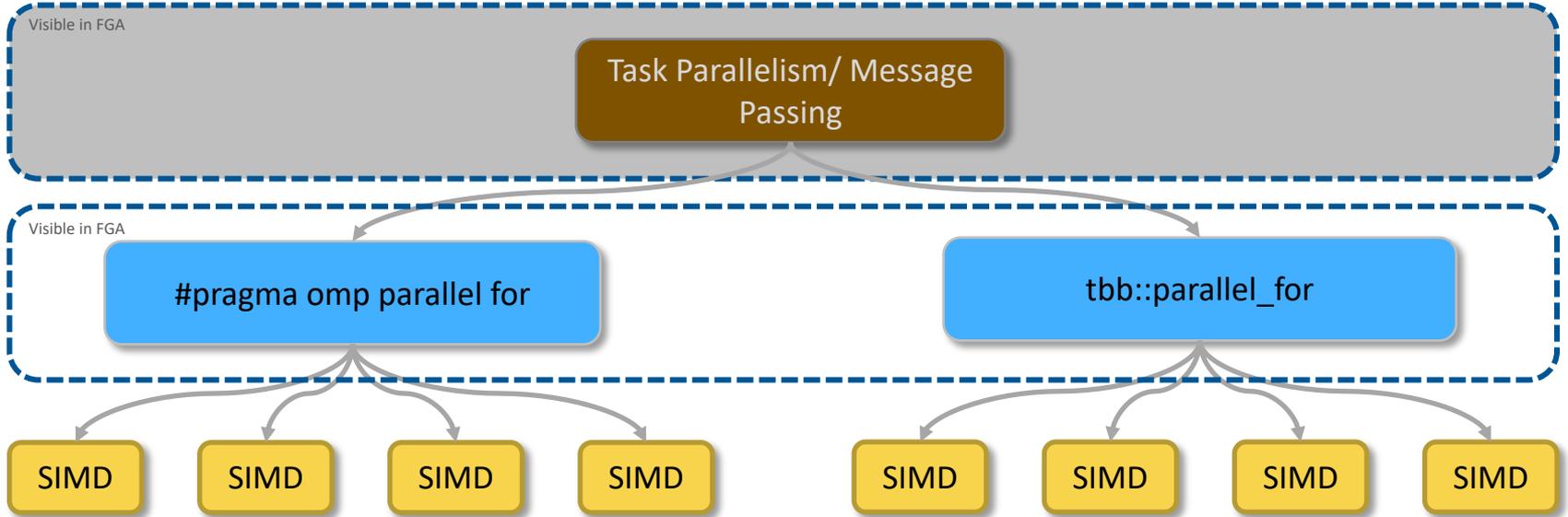
Challenges with asynchronous task graphs

- ✓ Creating implicit or explicit task-graphs programmatically is easy
 - ✓ Determining what was created is hard in many cases
- ✓ New programming paradigm
- ✓ Allows you to stream data through the graph, which makes debugging challenging
- ✓ Graph algorithms can be latency-bound or throughput-bound
- ✓ Parallelism is unstructured in certain types of graphs, so performance analysis can be challenging

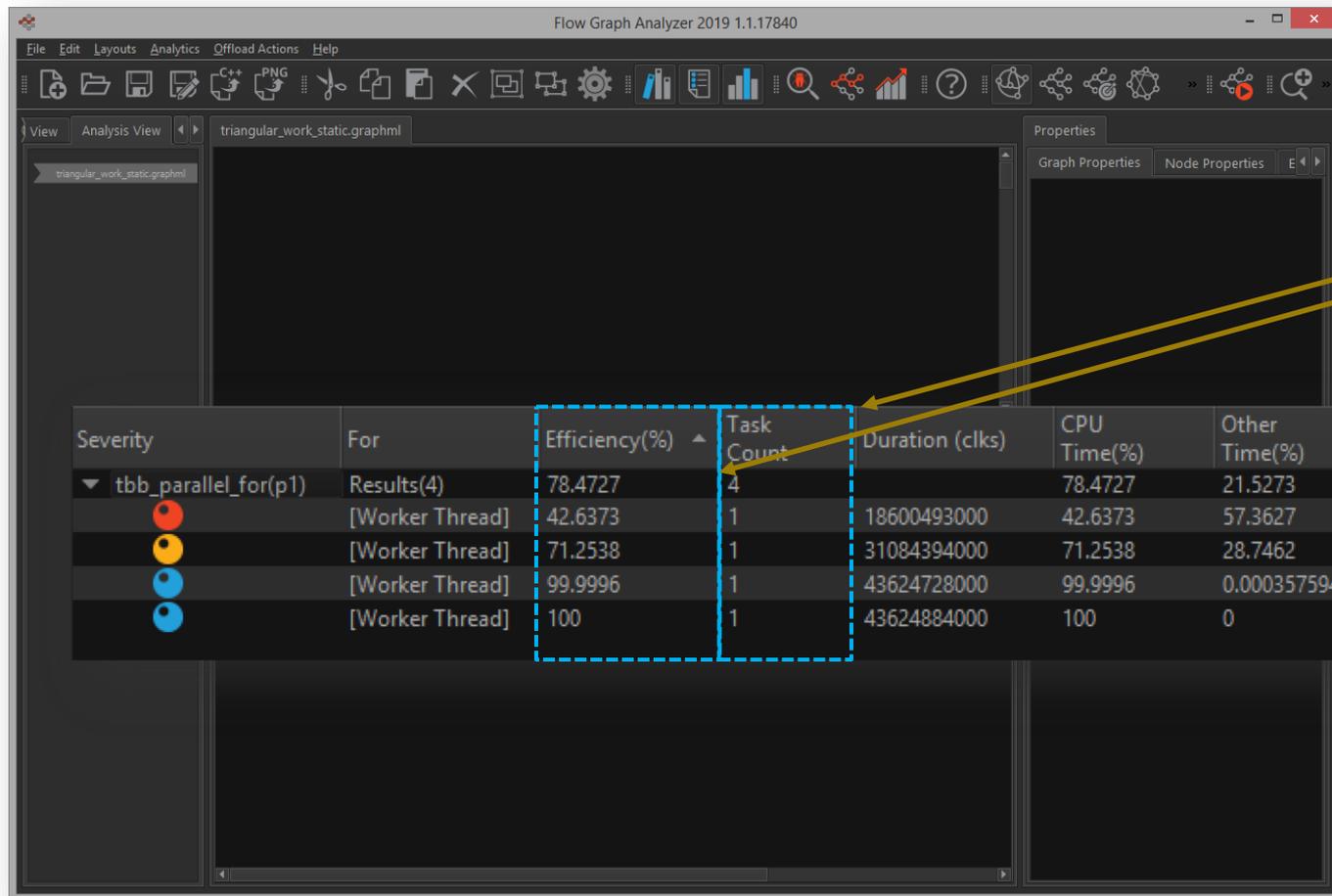
Nested and multi-level parallelism

What does it look like in FGA?

Applications often contain multiple levels of parallelism



Fork-join parallelism: tbb::parallel_for



Captures the execution task-graph for a fork-join construct and provides additional analytics that present information about the construct

1. Imbalance
2. Efficiency

Multi-level parallelism: graph level + fork-join

The screenshot displays the Flow Graph Analyzer interface. The main window shows a graph with a central function node $f()$ highlighted in red and enclosed in a dashed blue box. The graph also includes an input node, a join node, and two output nodes, all connected by green edges. The Properties panel on the right shows the selected node's details, including its name, position, and description.

The Execution Trace Views panel at the bottom shows a table of execution data for the graph and its nested algorithms. The table is filtered to show the execution of the `preprocess_function` node, which is highlighted in blue. The table columns include Severity, For, Efficiency(%), Count, Task, Duration (clks), CPU Time(%), and Other Time(%).

Severity	For	Efficiency(%)	Count	Task	Duration (clks)	CPU Time(%)	Other Time(%)
▼	preprocess_function[tbb_parallel_for(tbb0)]	Resu...	4.39186	597		4.39186	95.6081
		0.766243	32	452475	0.766243	99.2338	
		0.812855	32	465426	0.812855	99.1871	
		3.20528	31	351669	3.20528	96.7947	
		3.67587	32	631674	3.67587	96.3241	
		3.93136	55	1788411	3.93136	96.0686	
		3.98669	95	34862910	3.98669	96.0133	
		4.20178	64	1310382	4.20178	95.7982	
		14.5548	256	8437230	14.5548	85.4452	
▼	detect_A[tbb_parallel_for(tbb63367)]	Resu...	16.8823	511		16.8766	83.1234
		14.8203	127	1390761	14.8032	85.1968	
		15.9021	192	2273907	15.9021	84.0979	
		19.9244	192	2202204	19.9244	80.0756	
▼	preprocess_function[tbb_parallel_for(tbb62387)]	Resu...	18.1363	516		18.1356	81.8644
		1.09215	16	406809	1.09215	98.9079	
		2.59785	16	506949	2.59785	97.4022	
		5.08075	64	1598241	5.08075	94.9193	
		5.26153	65	1648431	5.26153	94.7385	
		5.64582	67	1755909	5.64128	94.3587	
		7.27568	96	6150000	7.27568	92.7243	
		100	192	5339739	100	0	
▼	preprocess_function[tbb_parallel_for(tbb22762)]	Resu...	18.7059	568		18.7059	81.2941
		0.620383	27	216834	0.620383	99.3796	

Timeline shows trace information for the graph and any nested parallelism that is present

Multi-level parallelism in OpenMP*

The screenshot shows the Flow Graph Analyzer interface. The top toolbar includes File, Edit, Layouts, Analytics, Offload Actions, and Help. The main window displays a flow graph for a parallel region (omp0:n0) with a blue dashed box highlighting a specific area. Below the flow graph is a 'Concurrency Overview' chart showing Node Concurrency over Time (0ms to 102.6ms). A blue dashed box highlights the time intervals corresponding to the parallel region. Below the chart is a 'Thread View' showing the execution of threads 0 through 8, with a red box highlighting 'implicit_tasks'.

Double-click on the parallel region node to see the activity within the region

```
1 pipeline_time = prk_wtime();
2 int lic = (m/mc-1) * mc + 1;
3 int ljc = (n/nc-1) * nc + 1;
4 for (int iter = 0; iter<=iterations; iter++) {
5     for (int i=1; i<n; i+=mc) {
6         for (int j=1; j<n; j+=nc) {
7             #pragma omp task depend(in: grid[0], grid[(i-mc)*n+j], \
8                                     grid[i*n+(j-nc)], \
9                                     grid[(i-mc)*n+(j-nc)]) \
10                depend(out: grid[i*n+j])
11                sweep_tile(i, MIN(m, i+mc), j, \
12                          MIN(n, j+nc), n, grid);
13        }
14    }
15    #pragma omp task depend(in: grid[(lic-1)*n+(ljc)]) \
16                depend(out: grid[0])
17        grid[0*n+0] = -grid[(m-1)*n+(n-1)];
18 }
19 #pragma omp taskwait
20 pipeline_time = prk_wtime() - pipeline_time;
```

Intel® Advisor – Flow graph analyzer

Download through Intel® Advisor package

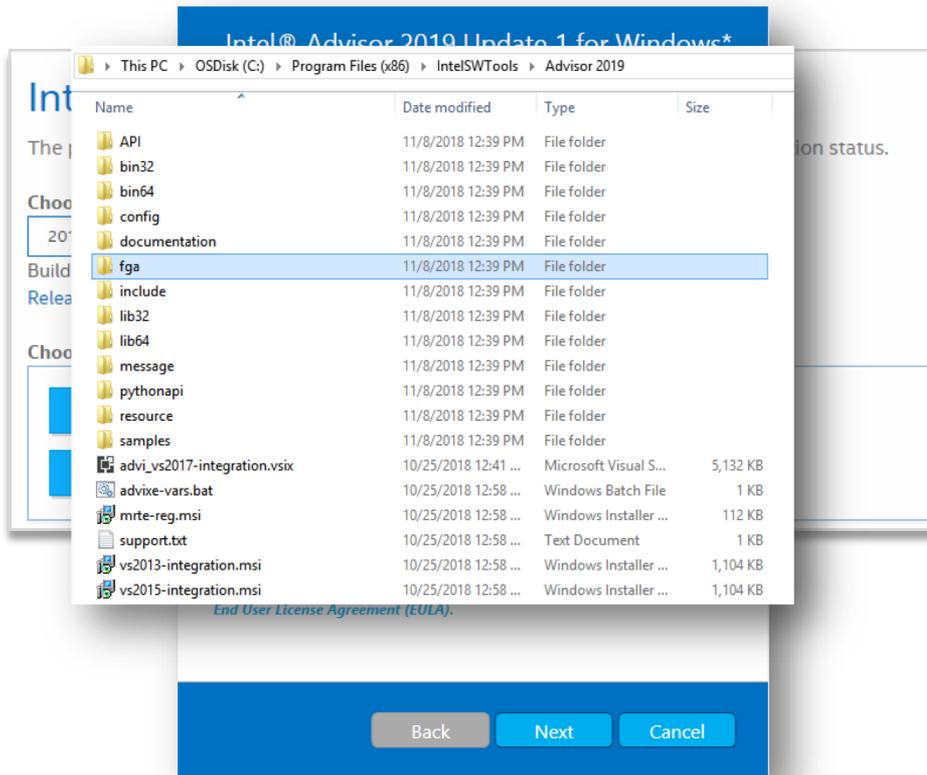
Intel® Advisor – Flow Graph Analyzer

Product feature in Intel® Parallel Studio XE 2019

Tool supports analysis and design of parallel applications using OpenMP* and Threading Building Blocks

Available for Windows*, Linux* and MacOS*

<https://software.intel.com/en-us/articles/getting-started-with-flow-graph-analyzer>



Summary

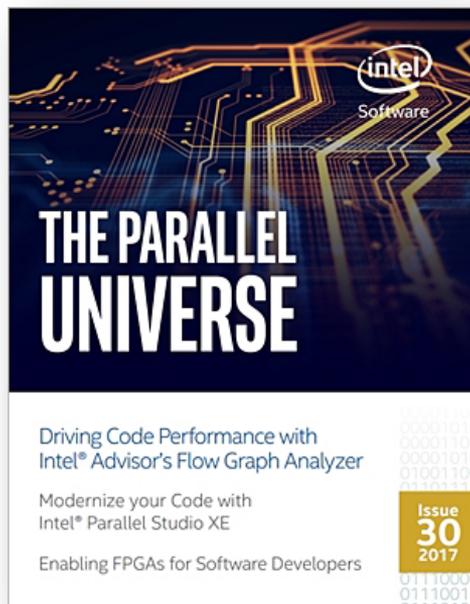
Asynchronous task-graphs improves the efficiency of irregular and runtime dependent execution

- TBB and OpenMP* provide mechanisms to program in this manner

Flow Graph Analyzer helps you create, debug, visualize and analyze such graphs

- Critical path analysis is crucial in reducing the complexity of the analysis problem to a handful of nodes
- Runtime specific analyses, such as the lightweight policy analysis for TBB, target additional performance improvements

Resources



Getting started with FGA

<https://software.intel.com/en-us/articles/getting-started-with-flow-graph-analyzer>

Driving Code Performance with Intel® Advisor's Flow Graph Analyzer

<https://software.intel.com/en-us/download/parallel-universe-magazine-issue-30-october-2017>

IWOMP 2018: Visualization of OpenMP* Task Dependencies Using Intel® Advisor – Flow Graph Analyzer

https://link.springer.com/chapter/10.1007%2F978-3-319-98521-3_12

CPUs, GPUs, FPGAs: Managing the alphabet soup with Intel Threading Building Blocks

<https://software.intel.com/en-us/videos/cpus-gpus-fpgas-managing-the-alphabet-soup-with-intel-threading-building-blocks>

Legal Disclaimer & Optimization Notice

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. For more complete information visit www.intel.com/benchmarks.

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.

Copyright © 2018, Intel Corporation. All rights reserved. Intel, the Intel logo, Pentium, Xeon, Core, VTune, OpenVINO, Cilk, are trademarks of Intel Corporation or its subsidiaries in the U.S. and other countries.

Optimization Notice

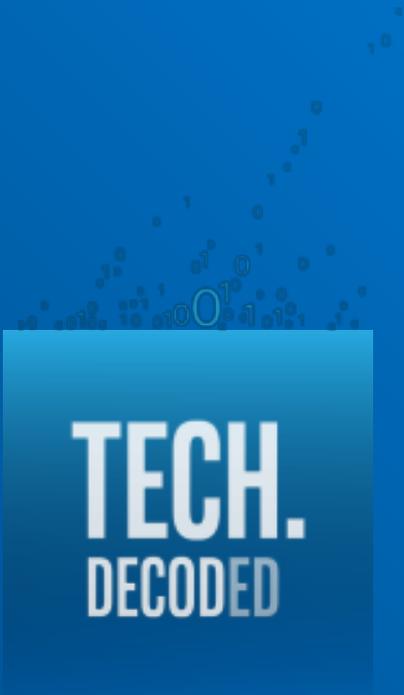
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, 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

Optimization Notice

Copyright © 2018, Intel Corporation. All rights reserved.

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



TECH.
DECODED



intel
Software