

Windows AI Performance Analysis

Overview

WinML Perf Analysis

- Load/Bind Times
 - CPU, GPU
- Single Evaluation Time
- Operator Profiling
- Throughput (IPS)
 - CPU, GPU
- Comparison
 - nGraph, OpenVINO

Multi-Model Analysis

- CNN + CNN in WinML
- CNN + RNN in WinML
- Multiple models in OpenVINO
- End-to-End App Scenario

Windows AI CPU Performance Analysis (19H2)

Platform Configuration

SKL Server Core i9 7940X (165W)	KBL NUC Core i7 8809G (35W)	KBL Laptop Core i7 8550U (15W)
14 Cores, 3.1-4.3GHz, HT Off	4 Cores, 3.1-4.1GHz, HT ON	4 Cores, 1.8-3.5GHz, HT On
SSE4.1, SSE4.2, AVX2, AVX-512 (2x FMA)	SSE4.1, SSE4.2, AVX2 (1x FMA)	SSE4.1, SSE4.2, AVX2 (1x FMA)
OMP_NUM_THREADS 8	-	-
OMP_WAIT_POLICY PASSIVE	-	-
Hyper-V Enabled	Hyper-V Enabled	Hyper-V Enabled

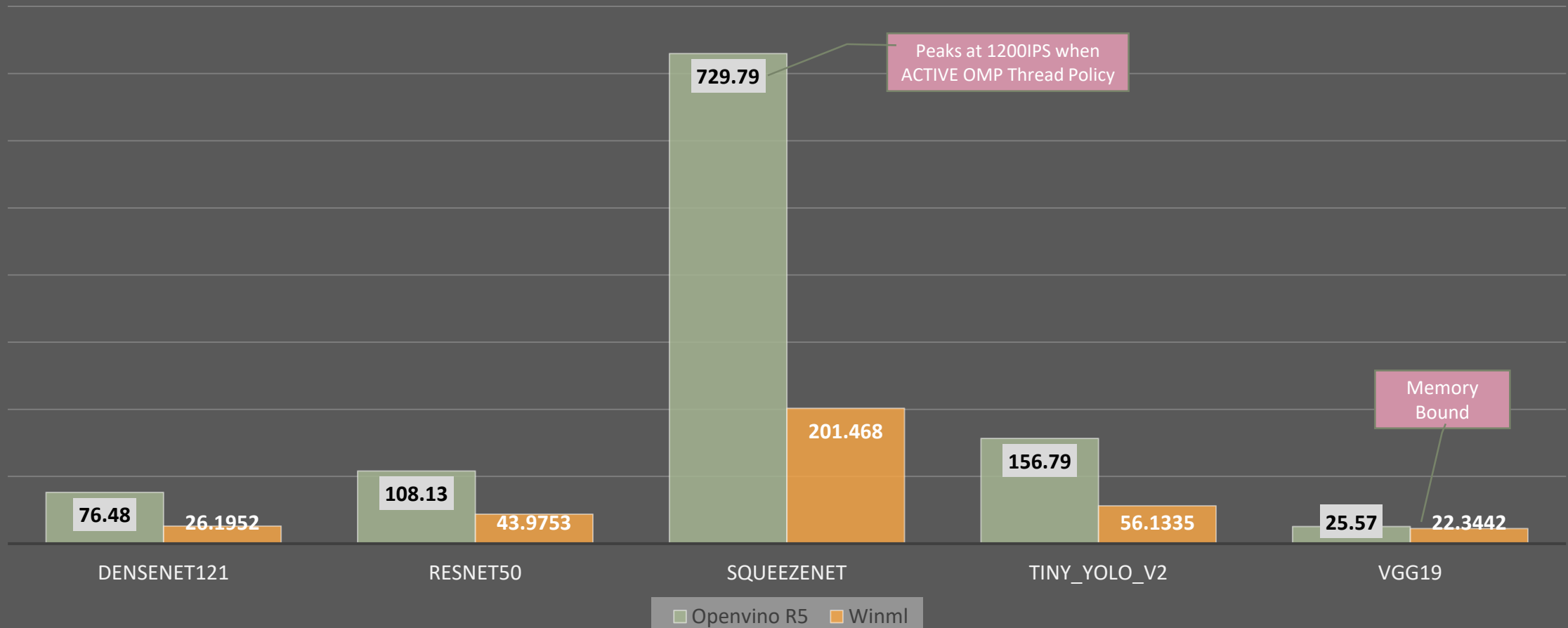
OS Version 19H1-18317-rs-onecore-sigma-mwb

WindowsMLRunner WW04'19-iterations(100)inputBinding(CPU)inputDataType(Tensor)

OpenVino 2019R5

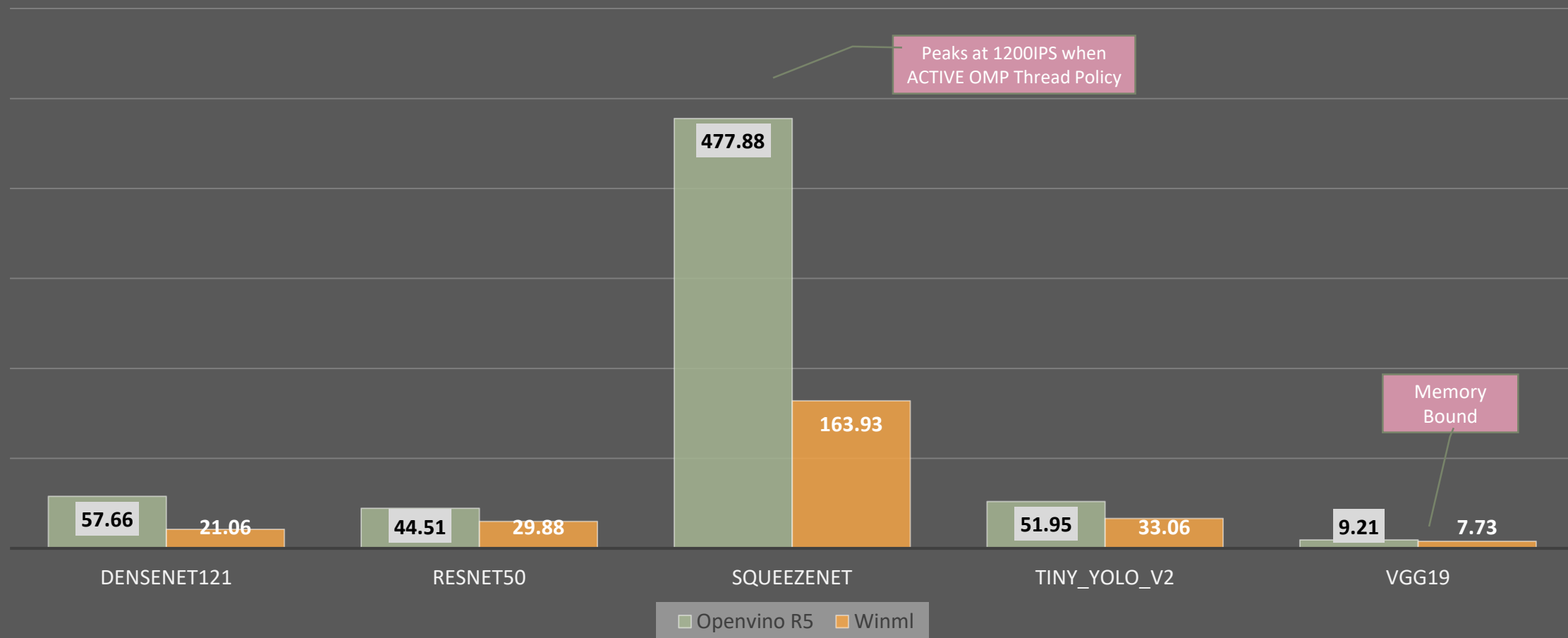
Performance Gap – SKL Server

Inferences per Second (Batch Size 1)



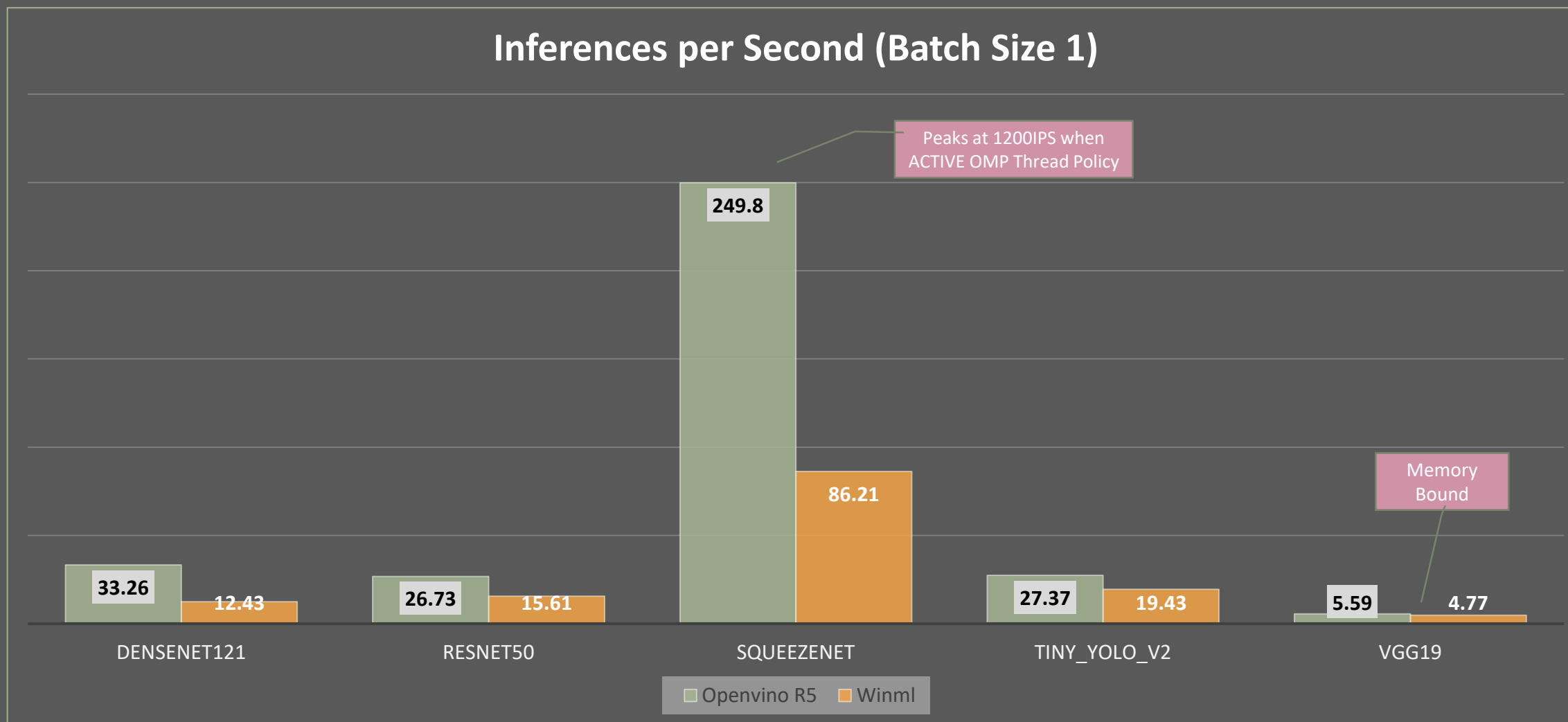
Performance Gap – KBL NUC

Inferences per Second (Batch Size 1)



Performance Gap – KBL Laptop

Inferences per Second (Batch Size 1)

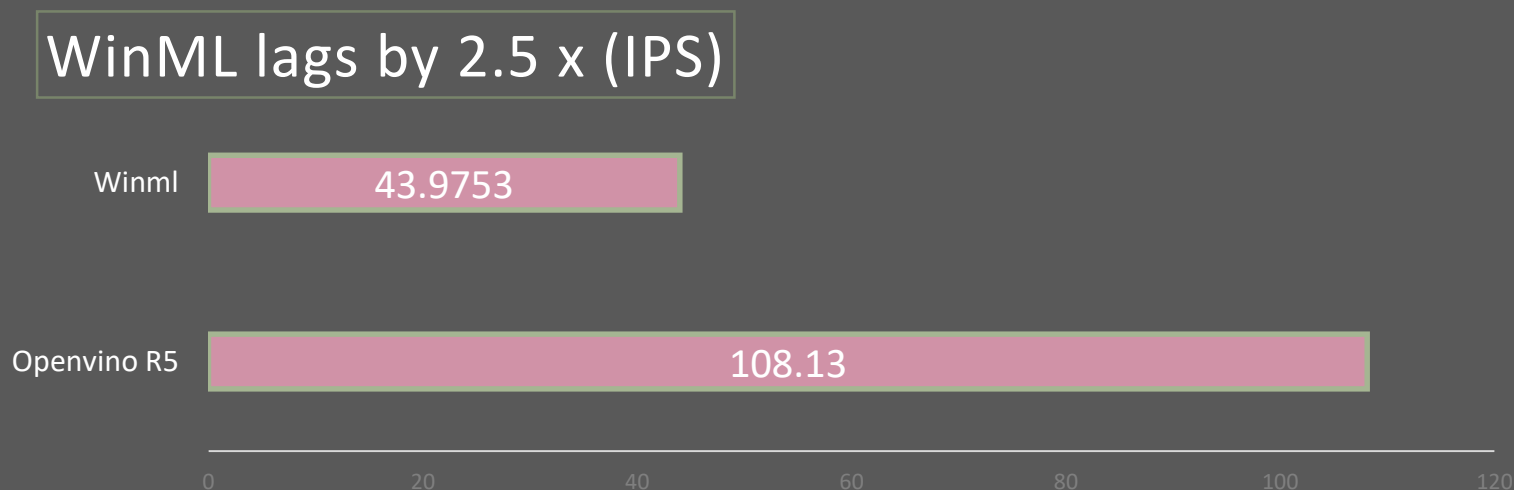


SKL Server

Resnet-50 Analysis

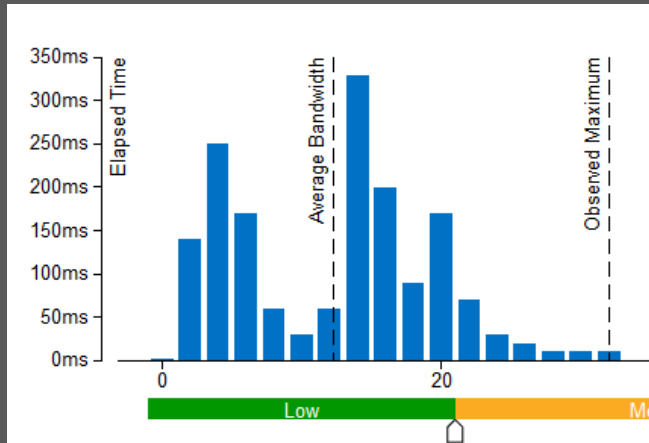
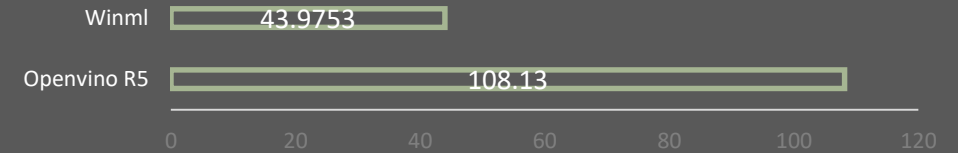
Content

- Memory Analysis
- Threading Analysis
- Micro Architecture Analysis
- Vectorization Analysis
- CPU Frequency Analysis
- Cache Utilization Analysis

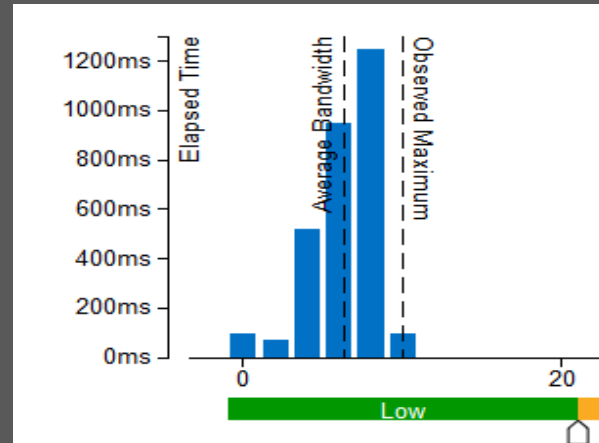


Memory Analysis

WinML lags by 2.5 x (IPS)



Open Vino (GB/s)



WinML(GB/sec)

	Open Vino	WinML
Elapsed Time:	1.651s	2.987s
CPU Time:	7.139s	12.041s
Memory Bound:	35.90%	28.30%
L1 Bound:	3.10%	7.80%
L2 Bound:	2.80%	5.00%
L3 Bound:	8.40%	12.00%
DRAM Bound:	10.90%	3.50%
Local DRAM:	47.60%	3.40%
Loads:	22,988,689,640	22,400,672,000
Stores:	644,019,320	4,256,127,680
LLC Miss Count:	45,503,185	3,500,245
Local DRAM Access Count:	38,502,695	3,500,245
Total Thread Count:	11	15

DRAM BW is more uniform in OpenVino and peaks to 32GB/sec, WinML memory access pattern suggests code is more efficient(also evident from lower CPI) , high in compute density but not well parallelized.

Threading Analysis

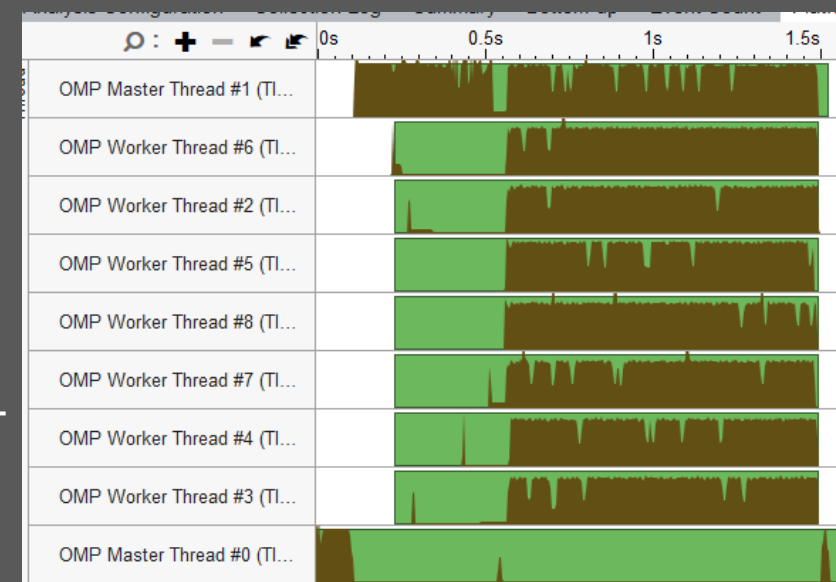
Hotspots HPC Performance Characterization

Analysis ConfigurationCollection LogSummaryBottom-upCaller/CalleeTop-down Treesgemm.cppsequential_executor.cc

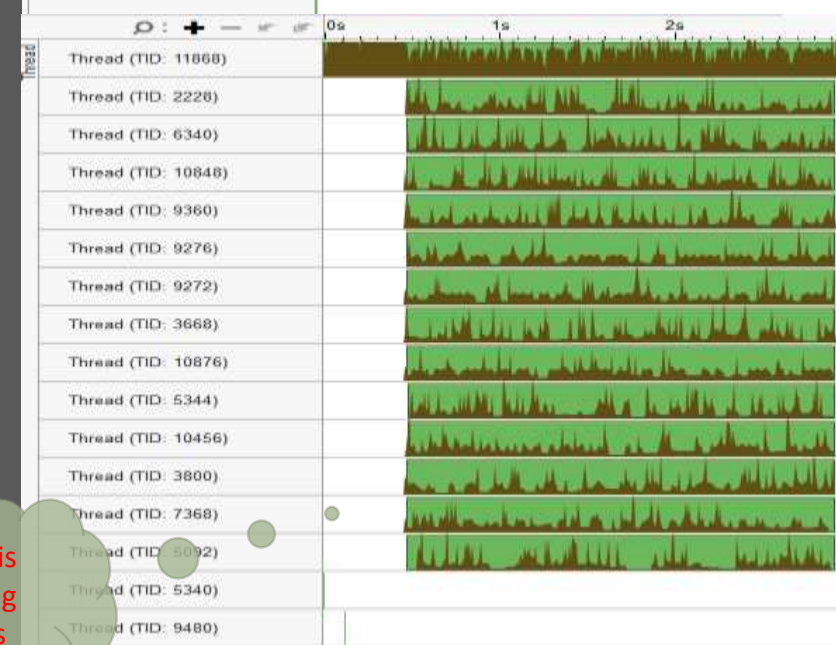
groupingCall Stack

Function Stack	CPU Time: Total					CPU Time: Self	CPI Rate: Total	Vectorization: Total			
	Effective Time by Utilization							% of FP Ops	FP Ops: Packed	FP Ops: Scalar	% of FP Ops
	Idle	Poor	Ok	Ideal	Over						
Total	100.0%					0s	0.514	17.0%	100.0%	0.0%	0.0%
▼ MlasThreadedWorkCallback	58.7%					0.007s	0.486	29.0%	100.0%	0.0%	0.0%
▼ MlasSgemmOperationThreaded	44.7%					0.005s	0.490	29.3%	100.0%	0.0%	0.0%
▼ MlasSgemmOperation	44.7%					0.004s	0.490	29.3%	100.0%	0.0%	0.0%
▼ [Loop at line 896 in MlasSgemmOperation]	44.7%					0.002s	0.490	29.3%	100.0%	0.0%	0.0%
▼ [Loop at line 916 in MlasSgemmOperation]	44.7%					0.018s	0.490	29.3%	100.0%	0.0%	0.0%
▼ MlasSgemmCopyPackB	30.0%					0.010s	0.459	0.0%	0.0%	0.0%	0.0%
▼ [Loop at line 283 in MlasSgemmCopyPackB]	27.8%					0.024s	0.463	0.0%	0.0%	0.0%	0.0%
▼ [Loop at line 293 in MlasSgemmCopyPackB]	27.7%					5.513s	0.463	0.0%	0.0%	0.0%	0.0%
▶ Eigen::internal::pload<union __m128>	10.5%					4.426s	0.425	0.0%	0.0%	0.0%	0.0%
▶ Eigen::internal::pstore<float, union __m128>	4.1%					1.713s	0.682	0.0%	0.0%	0.0%	0.0%
▼ [Loop at line 331 in MlasSgemmCopyPackB]	2.2%					0.428s	0.412	0.0%	0.0%	0.0%	0.0%
▶ Eigen::internal::pstore<float, union __m128>	1.1%					0.450s	0.376	0.0%	0.0%	0.0%	0.0%
▶ Eigen::internal::pload<union __m128>	0.1%					0.052s	0.416	0.0%	0.0%	0.0%	0.0%
MlasZeroFloat32x4	0.0%					0.002s		0.0%	0.0%	0.0%	0.0%
▼ [Loop at line 965 in MlasSgemmOperation]	14.6%					0.036s	0.564	100.0%	100.0%	0.0%	41.7%
▶ MlasSgemmKernelAddAvx512F	8.5%					0.041s	0.480	100.0%	100.0%	0.0%	34.6%
▶ MlasSgemmKernelZeroAvx512F	6.0%					0.037s	0.729	100.0%	100.0%	0.0%	0.0%
▶ guard_dispatch_icall_nop	0.0%					0.004s	2.667	0.0%	0.0%	0.0%	0.0%
▶ _security_check_cookie	0.0%					0.005s	1.667	0.0%	0.0%	0.0%	0.0%
▶ [Loop at line 883 in MlasSgemmOperation]	0.0%					0.001s	0.000	0.0%	0.0%	0.0%	0.0%
▶ MlasConvOperationThreaded	14.0%					0s	0.475	28.1%	100.0%	0.0%	0.0%
▼ onnxruntime::SequentialExecutor::Execute	33.1%					0s	0.542	2.4%	100.0%	0.0%	0.0%
▼ [Loop at line 46 in onnxruntime::SequentialExecutor::Execute]	33.1%					0.004s	0.541	2.4%	100.0%	0.0%	0.0%
▼ [Loop at line 101 in onnxruntime::SequentialExecutor::Execute]	32.7%					0s	0.539	2.4%	100.0%	0.0%	0.0%
▶ onnxruntime::Relu<float>::Compute	15.5%					0s	0.536	0.3%	100.0%	0.0%	0.0%
▶ onnxruntime::Sum_6<float>::Compute	7.8%					0.001s	0.550	0.4%	100.0%	0.0%	0.0%
▶ onnxruntime::Conv<float>::Compute	6.1%					0.002s	0.515	11.3%	100.0%	0.0%	0.0%
▶ onnxruntime::SequentialExecutor::Execute	1.7%					0s	0.541	3.5%	100.0%	0.0%	0.0%
▶ onnxruntime::Pool<float, class onnxruntime::MaxPool<1>>::Compute	1.4%					0s	0.557	1.0%	100.0%	0.0%	0.0%
▶ onnxruntime::Gemm<float, float, float, float>::Compute	0.1%					0s	1.886	28.6%	100.0%	0.0%	0.0%
▶ onnxruntime::Pool<float, class onnxruntime::AveragePool>::Compute	0.1%					0s	0.833	8.0%	100.0%	0.0%	0.0%
▶ MlasConv	0.0%					0s	0.553	10.0%	100.0%	0.0%	0.0%
▶ onnxruntime::Softmax<float>::Compute	0.0%					0s	0.450	0.0%	0.0%	0.0%	0.0%
▶ onnxruntime::logging::Capture::Capture	0.0%					0.001s	1.000	0.0%	0.0%	0.0%	0.0%

OpenVino

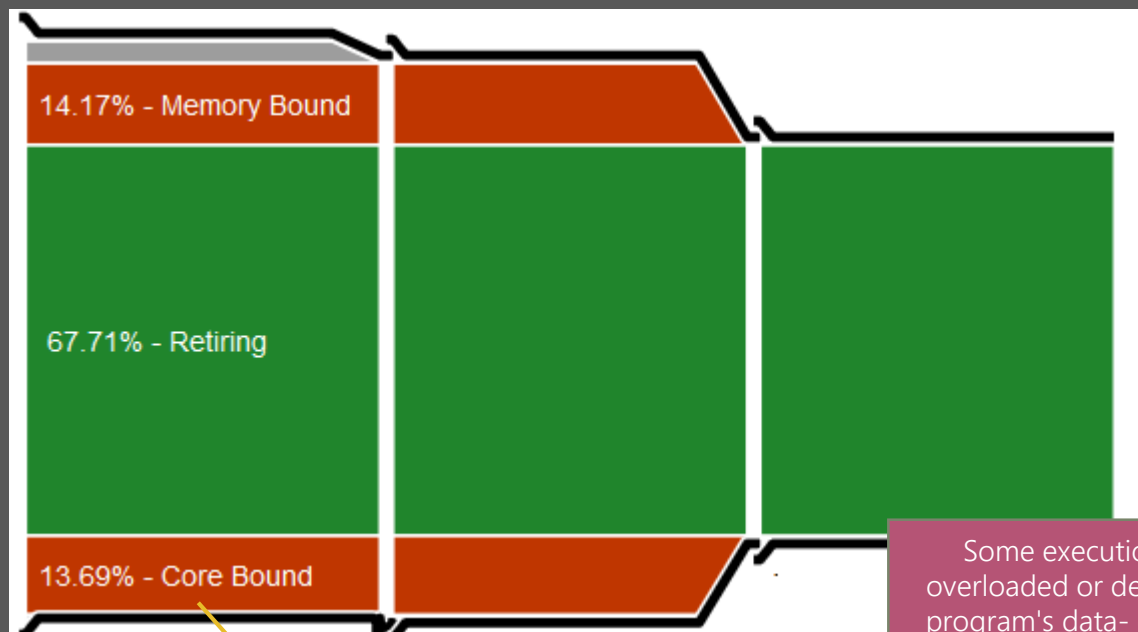


WinML



WinML is
not using
threads
efficiently

Micro Architecture Analysis



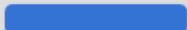
















Some execution units are overloaded or dependencies in program's data- or instruction-flow are limiting the performance

This value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. This metric represents how much Core non-memory issues were of a bottleneck. Shortage in hardware compute resources, or dependencies software's instructions are both categorized under Core Bound. It may indicate the machine ran out of an OOO resources.

Long run of single port

	Open VINO	WindowsML
Elapsed Time:	1.557s	2.954s
Clockticks:	22,010,000,000	38,161,000,000
Instructions Retired:	36,580,000,000	72,679,500,000
CPI Rate:	0.602	0.525
MUX Reliability:	0.622	0.879
Retiring:	52.70%	67.70%
Front-End Bound:	4.70%	3.80%
Bad Speculation:	0.10%	0.60%
Back-End Bound:	42.40%	27.90%
Memory Bound:	35.60%	14.20%
Core Bound:	6.80%	13.70%
Divider:	0.00%	0.00%
Port Utilization:	7.10%	29.30%
Cycles of 0 Ports Utilized:	9.50%	10.90%
Serializing Operations:	0.60%	5.50%
Cycles of 1 Port Utilized:	2.10%	4.10%
Cycles of 2 Ports Utilized:	7.40%	5.50%
Cycles of 3+ Ports Utilized:	23.30%	16.00%
Port 0:	31.90%	27.60%
Port 1:	0.60%	10.90%
Port 2:	25.40%	24.90%
Port 3:	25.40%	25.20%
Port 4:	1.20%	14.30%
Port 5:	31.90%	27.30%
Port 6:	1.20%	17.70%
Port 7:	0.60%	4.10%
Vector Capacity Usage (FPU):	100.00%	100.00%
Average CPU Frequency:	3.1 GHz	3.2 GHz
Total Thread Count:	10	16
Paused Time:	0s	0s

Micro Architecture Analysis

Function / Call Stack	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate	Bad Speculation	Back-End Bound	
						Memory Bound »	Core Bound
► Eigen::internal::pload<union __m128>	8.593s 	28,000,000,000	50,848,000,000	0.551	7.1%	3.0%	6.9%
► [Loop at line 293 in MlasSgemmCopyPackB]	6.998s 	21,548,800,000	43,475,600,000	0.496	0.0%	6.6%	21.3%
► Eigen::internal::pstore<float,union __m128>	4.309s 	13,636,000,000	22,867,600,000	0.596	2.7%	8.7%	17.8%
► [Loop at line 583 in MlasSgemmKernelAddAvx512F]	3.503s 	10,480,400,000	23,671,200,000	0.443	2.7%	14.0%	22.5%
► [Loop at line 582 in MlasSgemmKernelZeroAvx512F]	3.118s 	10,572,800,000	13,977,600,000	0.756	2.9%	54.6%	10.8%
► Eigen::internal::mapbase_evaluator<class Eigen::Block<class	1.538s 	5,801,600,000	8,954,400,000	0.648	25.9%	0.9%	4.4%
► Eigen::internal::pset1<union __m128>	1.146s 	4,340,000,000	13,916,000,000	0.312	0.0%	0.0%	0.0%
► [Loop at line 69 in MlasBiasAdd]	0.952s 	2,903,600,000	5,860,400,000	0.495	2.8%	4.7%	22.9%
► [Loop at line 191 in MlasConvIm2Col]	0.876s 	2,786,000,000	5,367,600,000	0.519	2.3%	7.1%	32.1%
► Eigen::internal::ploadt<union __m128,0>	0.821s 	3,144,400,000	3,973,200,000	0.791	27.4%	3.4%	6.8%
► Eigen::internal::padd<union __m128>	0.752s 	2,595,600,000	3,575,600,000	0.726	15.5%	10.2%	24.9%
► Eigen::internal::scalar_constant_op<float>::packetOp<union _	0.680s 	2,511,600,000	9,410,800,000	0.267	0.0%	6.1%	26.8%
► [Loop at line 176 in MlasConvIm2Col]	0.666s 	2,114,000,000	4,916,800,000	0.430	6.1%	4.0%	18.0%
► Eigen::internal::binary_evaluator<class Eigen::CwiseBinaryOp	0.636s 	2,494,800,000	3,365,600,000	0.741	22.5%	0.0%	16.7%
► Eigen::internal::pmax<union __m128>	0.625s 	2,438,800,000	4,242,000,000	0.575	19.6%	5.7%	8.5%
► Eigen::internal::scalar_max_op<float,float>::packetOp<union _	0.617s 	2,108,400,000	3,452,400,000	0.611	0.0%	0.0%	0.0%
► Eigen::internal::binary_evaluator<class Eigen::CwiseBinaryOp	0.515s 	1,864,800,000	3,766,000,000	0.495	25.4%	0.0%	0.0%

Solution: Resources Contention can be resolved by using more accumulators. This will help fill up the pipeline better.

Vectorization Analysis(*/mlas/lib/x86_64/SgemmKernelAvx512F.S*)

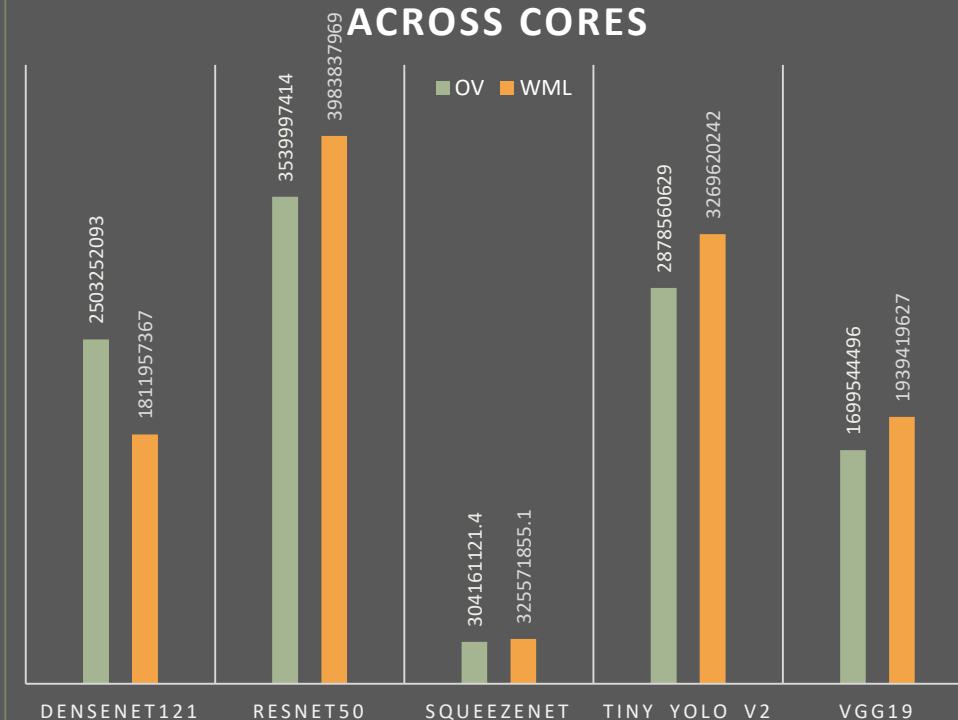
Number of AVX512 Instructions retired reported via event (FP_ARITH_INST_RETIRED.512B_PACKED_SINGLE) are not very different between OpenVino and WindowsML.

However it is important to note following

- vzeroall is used to both clear the accumulators at the start of the loop iteration and provides SSE friendly return state. Vzeroall is slower than vvzeroupper as it was not expected to appear inside loops.
- For remainingCount Using Masking registers is good to hold residual however using them have overheads. Padding to multiples of 16 would provide superior performance.
- Having non-masked loops for M(row major) ≥ 16 & M(row major) ≥ 32 would be better.
- For M ≤ 16 there is potential of a power virus, this shouldn't be more than 15% range, recommend AVX2 for smaller shapes to avoid uarch bubbles.
- Absolute minimum number of accumulators given 2FMA at latency of 4 is 8, however it is better to allocate more. The first three have enough work to hide FMA latency remaining do not.

A	ProcessNextColumnLoop32x12	24 accumulators, with 12 using masks in case of edge
B	ProcessRemainingCountN12	12 accumulators, all using masks
C	ProcessNextColumnLoop32x6	12 accumulators, with 6 using masks
D	ProcessRemainingCountN6	6 accumulators, all using mask
E	ProcessNextColumnLoop32x3	6 accumulators, 3 using mask
F	ProcessRemainingCountN3	3 accumulators all using mask
G	ProcessNextColumnLoop32x1	2 accumulators with 1 using mask
H	ProcessRemainingCountN1	1 accumulator using mask

**AVERAGED AVX512 RETIRED EVENTS
ACROSS CORES**



Vectorization Analysis(*/mlas/lib/x86_64/SgemmKernelAvx512F.S*)

A ProcessNextColumnLoop32x12 24 accumulators, with 12 using masks in case of edge

B ProcessRemainingCountN12 12 accumulators, all using masks

C ProcessNextColumnLoop32x6 12 accumulators, with 6 using masks

D ProcessRemainingCountN6 6 accumulators, all using mask

E ProcessNextColumnLoop32x3 6 accumulators, 3 using mask

F ProcessRemainingCountN3 3 accumulators all using mask

G ProcessNextColumnLoop32x1 2 accumulators with 1 using mask

H ProcessRemainingCountN1 1 accumulator using mask

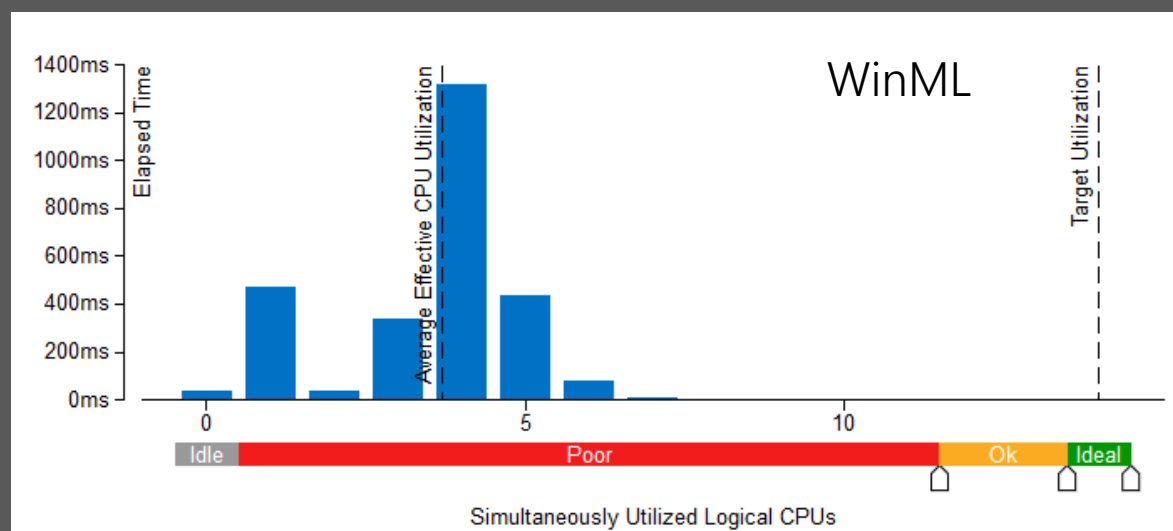
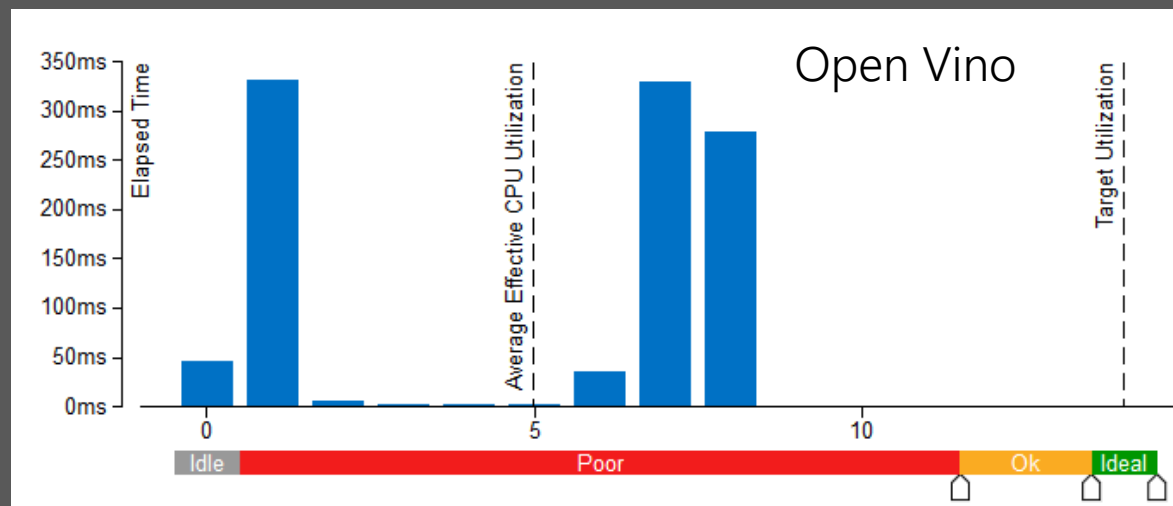
How to get more accumulators and fill up the pipeline better?

- A - unroll k across accumulators and then do a reduction prior to the scale and store
 - Instead of unroll x 4 doing FMA into the same accumulators, do each into a separate set
 - Codepaths D, E, and F would go to 24 and 12 accumulators and there is usually enough adds so that you'll be fully pipelined prior to scale and store
-
- B - unroll additional iteration(s) in N (which also reduces the amount of masking even if nothing else is done)
 - path C becomes 6 x (33 to 48) (24 accumulators) (1 additional path)
 - Path E becomes 3 x (49 to 64), 24 accumulators or 3 x (33 to 48), 9 accumulators (not really an improvement)
 - This starts to look like loop interchange when N small and M large and looks complicated, K unrolling is probably preferred (assuming there is a decent K dimension)
- C- Code paths D, E, F, G, H, MAY be better off using VL = 256, (or AVX2, though VL=256 gives you more registers, masking, etc)
 - this will be higher frequency and the since the FMA is not the bottleneck, we can swallow more accumulators
 - F, G, and H will STILL not hide the FMA latency without some additional help

Lines 331, 510,653

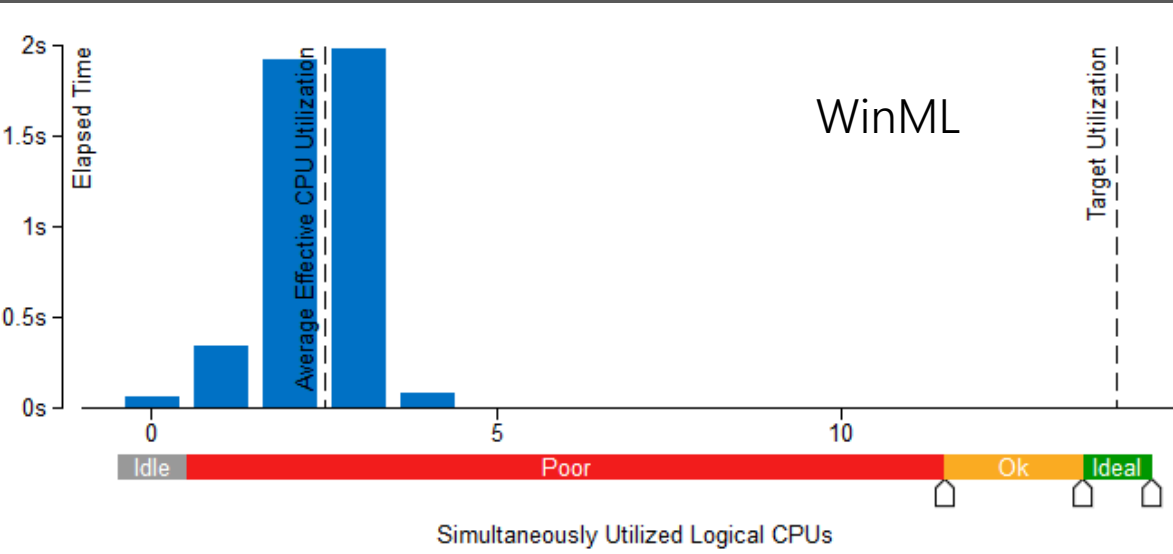
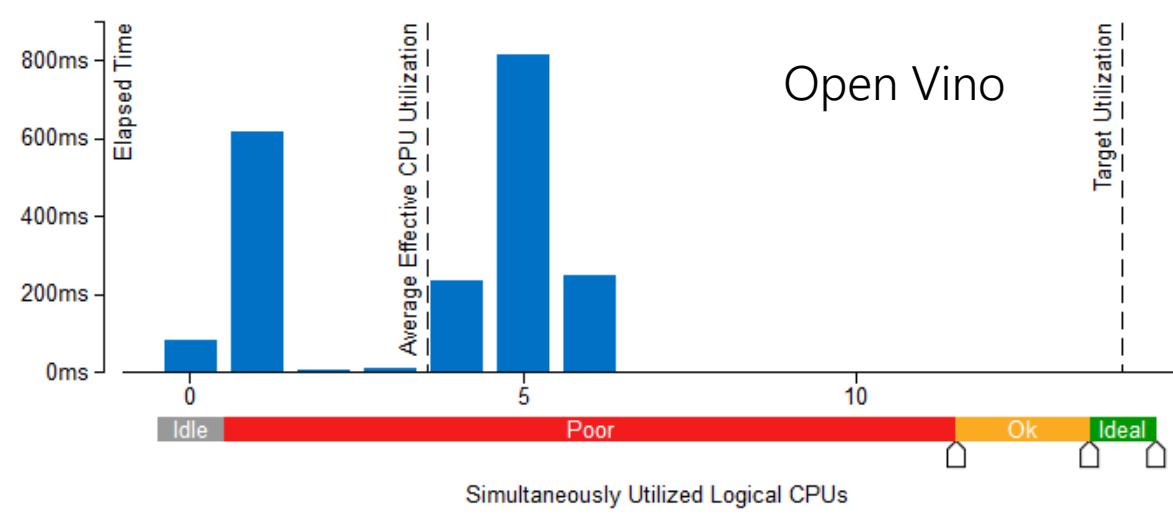
Function	CPU Time: Total ▾	Microarchitecture Usage: Total		Instructions Retired: Self
		Microarchitecture Usage	CPI Rate	
MlasThreadedWorkCallback	24.236s <div></div>	57.1%	0.494	2,800,000
MlasSgemmOperation	22.823s <div></div>	56.7%	0.500	5,600,000
[Loop at line 896 in MlasSgemmOperation]	22.738s <div></div>	57.0%	0.498	8,400,000
[Loop at line 916 in MlasSgemmOperation]	22.734s <div></div>	57.0%	0.498	47,600,000
MlasSgemmOperationThreaded	19.494s <div></div>	56.2%	0.501	0
onnxruntime::SequentialExecutor::Execute	14.735s <div></div>	52.5%	0.557	0
[Loop at line 46 in onnxruntime::SequentialExecutor::Execute]	14.696s <div></div>	52.6%	0.556	16,800,000
MlasSgemmCopyPackB	14.438s <div></div>	63.5%	0.457	56,000,000
[Loop at line 101 in onnxruntime::SequentialExecutor::Execute]	13.811s <div></div>	53.5%	0.544	0
[Loop at line 283 in MlasSgemmCopyPackB]	13.478s <div></div>	63.1%	0.458	263,200,000
[Loop at line 293 in MlasSgemmCopyPackB]	13.446s <div></div>	63.2%	0.458	44,307,200,000
[Loop at line 965 in MlasSgemmOperation]	8.260s <div></div>	45.9%	0.584	84,000,000
Eigen::internal::pload<union __m128>	7.593s <div></div>	63.2%	0.494	50,335,600,000
[Loop at line 415 in Eigen::internal::dense_assignment_loop<class Eigen::i	7.412s <div></div>	54.2%	0.536	2,419,200,000
Eigen::internal::dense_assignment_loop<class Eigen::internal::generic_der	7.412s <div></div>	54.1%	0.536	0
Eigen::internal::generic_dense_assignment_kernel<struct Eigen::internal::e	7.150s <div></div>	54.1%	0.543	1,680,000,000
onnxruntime::Relu<float>::Compute	6.776s <div></div>	53.2%	0.538	0
Eigen::internal::Assignment<class Eigen::Map<class Eigen::Array<float,-1,	6.751s <div></div>	53.3%	0.536	0
Eigen::DenseBase<class Eigen::Map<class Eigen::Array<float,-1,1,0,-1,1>	6.751s <div></div>	53.2%	0.537	0
Eigen::internal::call_assignment<class Eigen::Map<class Eigen::Array<floa	6.751s <div></div>	53.2%	0.537	0
Eigen::internal::call_assignment<class Eigen::Map<class Eigen::Array<floa	6.751s <div></div>	53.2%	0.537	0
Eigen::internal::call_assignment_no_alias<class Eigen::Map<class Eigen::A	6.751s <div></div>	53.2%	0.537	0
Eigen::internal::call_dense_assignment_loop<class Eigen::Map<class Eige	6.750s <div></div>	53.3%	0.536	0
MlasConvOperation	6.047s <div></div>	58.7%	0.481	0
[Loop at line 561 in MlasConvOperation]	6.044s <div></div>	58.7%	0.481	0
MlasConvOperationThreaded	6.017s <div></div>	58.7%	0.482	0
Eigen::internal::binary_evaluator<class Eigen::CwiseBinaryOp<struct Eiger	5.811s <div></div>	58.5%	0.501	3,511,200,000
[Loop at line 577 in MlasConvOperation]	5.487s <div></div>	61.1%	0.473	5,600,000
MlasSgemmKernelAddAvx512F	4.577s <div></div>	52.1%	0.475	137,200,000

Tiny Yolo V2



	Open VINO	WindowsML
Elapsed Time:	1.035s	2.730s
Clockticks:	16,290,500,000	32,054,000,000
Instructions Retired:	25,683,500,000	60,620,500,000
CPI Rate:	0.634	0.529
MUX Reliability:	0.497	0.862
Retiring:	44.90%	77.30%
General Retirement:	44.70%	76.80%
FP Arithmetic:	45.80%	21.40%
FP x87:	0.00%	0.00%
FP Scalar:	0.00%	0.10%
FP Vector:	45.80%	21.30%
Other:	54.20%	78.60%
Microcode Sequencer:	0.20%	0.50%
Front-End Bound:	4.60%	8.70%
Bad Speculation:	0.40%	2.50%
Back-End Bound:	50.10%	11.50%
Memory Bound:	43.60%	5.30%
Core Bound:	6.50%	6.20%
Average CPU Frequency:	3.1 GHz	3.2 GHz
Total Thread Count:	10	17

Densenet121



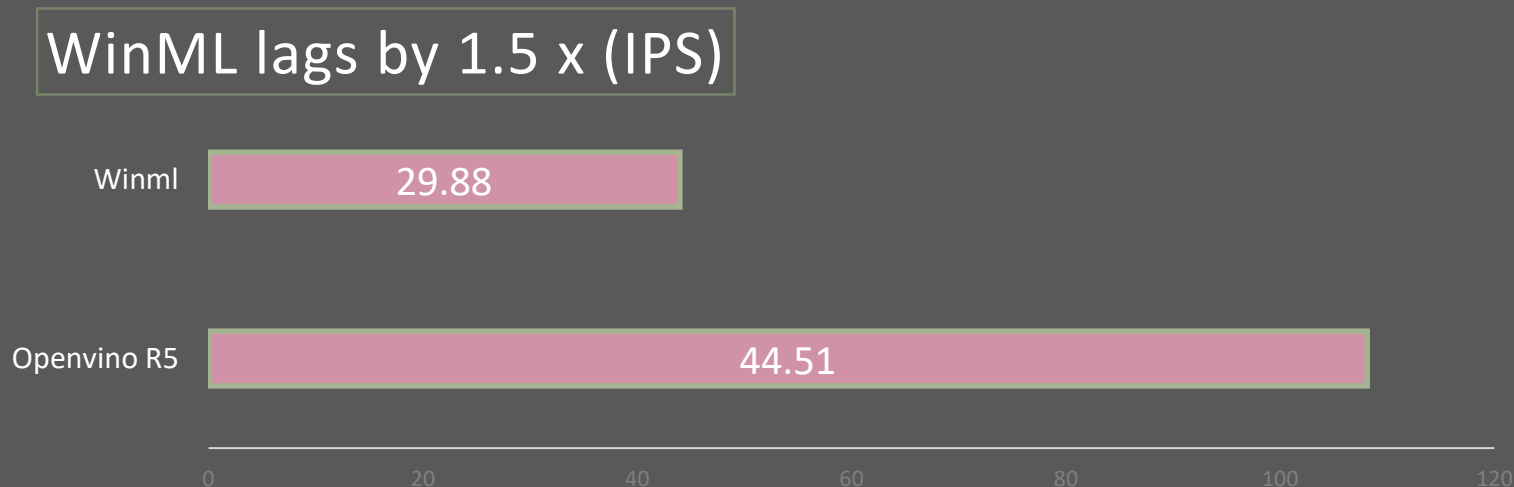
	Open VINO	WindowsML
Elapsed Time:	2.026s	4.402s
Clockticks:	23,250,000,000	34,890,500,000
Instructions Retired:	32,705,000,000	71,625,500,000
CPI Rate:	1	0.487
MUX Reliability:	1	0.955
Retiring:	50.70%	77.00%
General Retirement:	50.10%	76.40%
FP Arithmetic:	54.50%	32.60%
FP x87:	0.00%	0.00%
FP Scalar:	0.00%	0.00%
FP Vector:	54.50%	32.60%
Other:	45.50%	67.40%
Microcode Sequencer:	0.70%	0.60%
Front-End Bound:	9.50%	4.80%
Bad Speculation:	1.30%	0.60%
Back-End Bound:	38.50%	17.60%
Memory Bound:	32.80%	9.00%
Core Bound:	5.70%	8.60%
Divider:	0.00%	0.00%
Port Utilization:	8.40%	27.20%
Average CPU Frequency:	3.1 GHz	3.2 GHz
Total Thread Count:	10	16
Paused Time:	0s	0s

KBL NUC

Resnet-50 Analysis

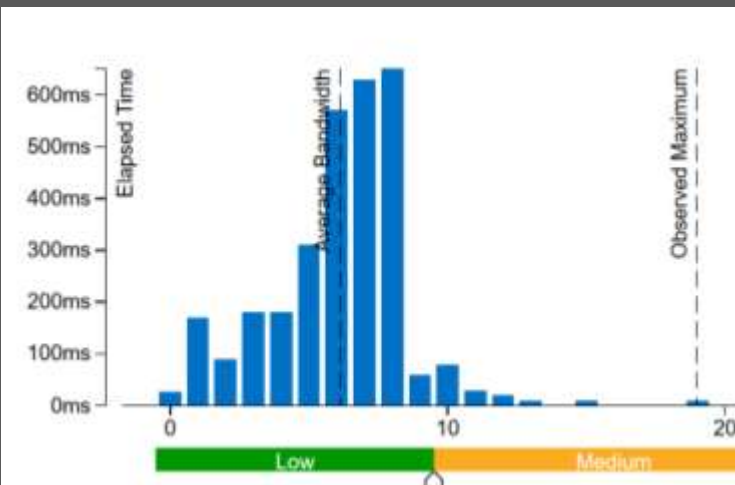
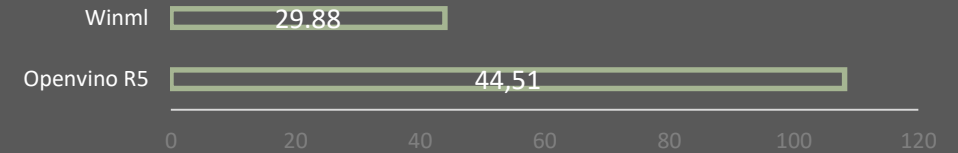
Content

- Memory Analysis
- Threading Analysis
- Micro Architecture Analysis
- Vectorization Analysis
- CPU Frequency Analysis
- Cache Utilization Analysis

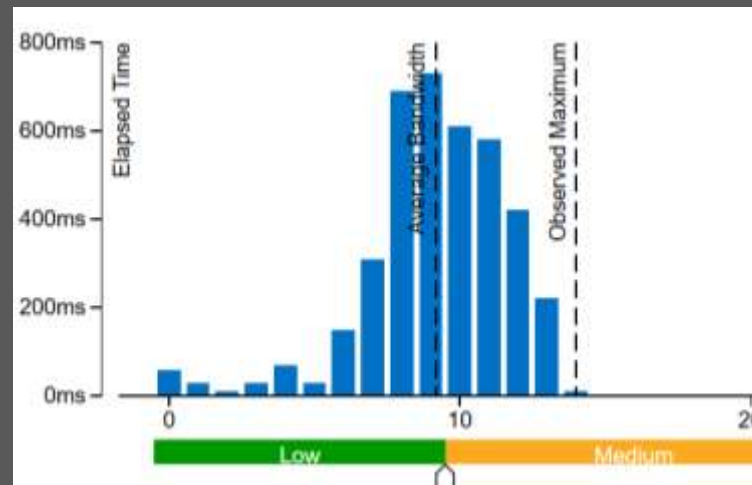


Memory Analysis

WinML lags by 1.5 x (IPS)



Open Vino (GB/s)

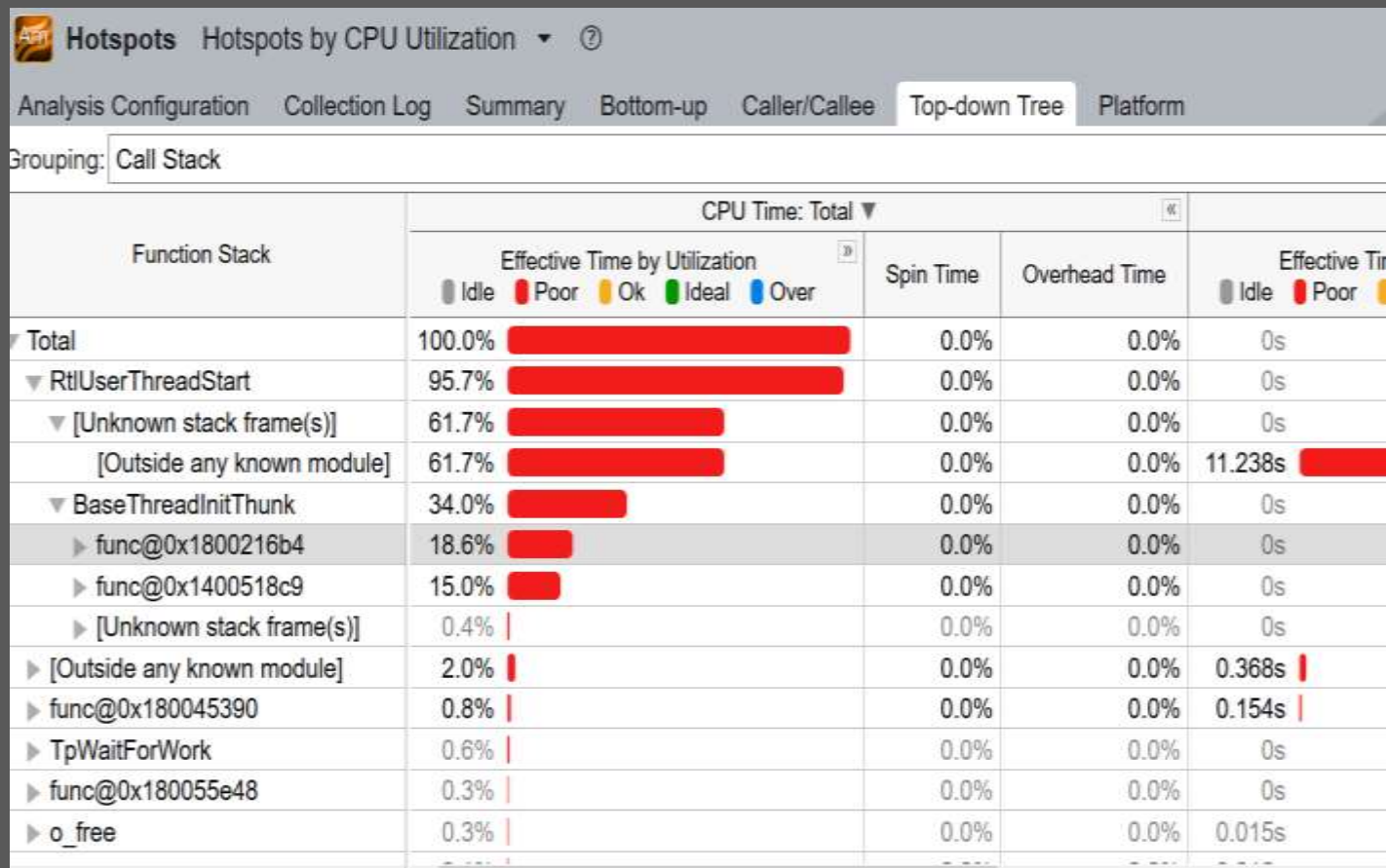


WinML(GB/sec)

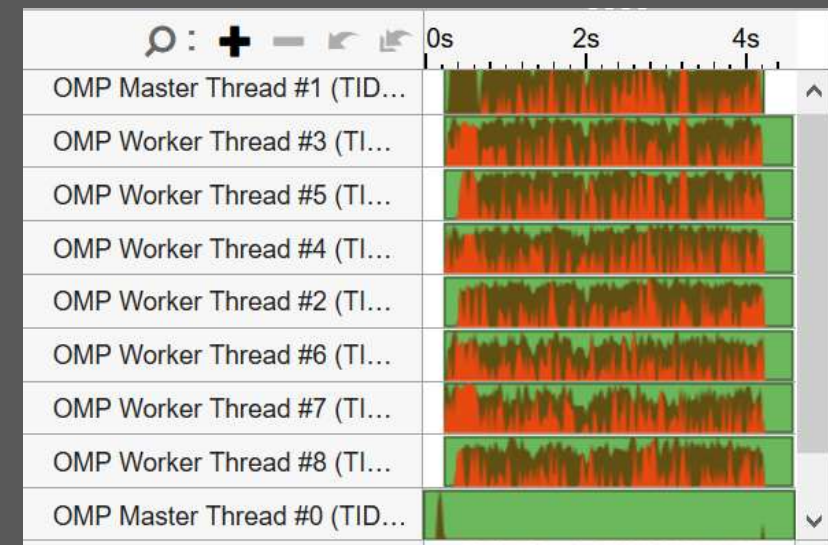
	Open Vino	WinML
Elapsed Time:	3.026s	3.949s
CPU Time:	21.944s	18.494s
Memory Bound:	9.20%	9.20%
L1 Bound:	9.80%	9.00%
L2 Bound:	3.80%	6.00%
L3 Bound:	3.00%	3.30%
DRAM Bound:	3.30%	2.90%
Local DRAM:		
Loads:	35,446,663,368	41,660,449,776
Stores:	2,640,079,200	3,687,710,628
LLC Miss Count:	19,201,344	12,000,840
Local DRAM Access Count:		
Total Thread Count:	13	11

DRAM BW has lower memory demand with a few outliers, WinML memory access pattern suggests code is more efficient(also evident from lower CPI) , high in compute density but not well parallelized.

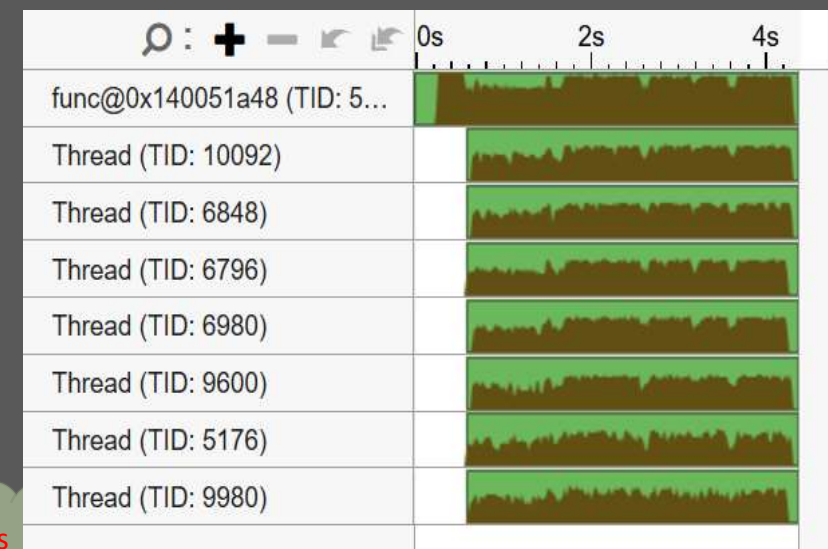
Threading Analysis



OpenVino

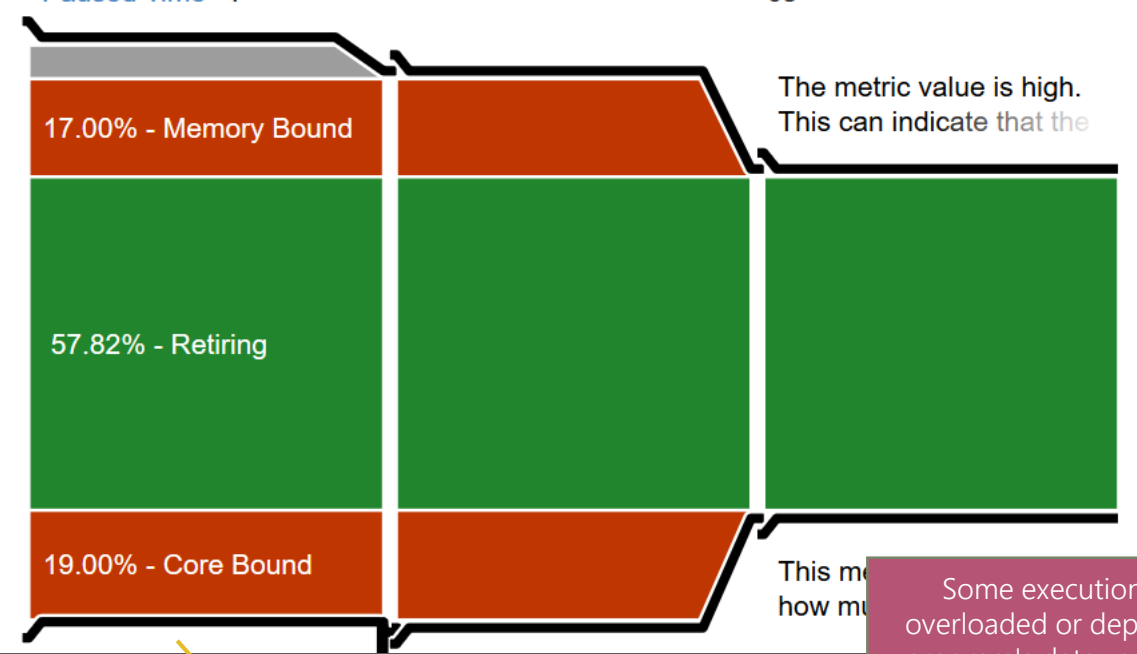


WinML



WinML is not using threads efficiently

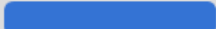











Micro Architecture Analysis



	Open VINO	WindowsML
Elapsed Time:	2.939s	3.932s
Clockticks:	80,262,100,000	72,124,600,000
Instructions Retired:	97,197,400,000	119,191,900,000
CPI Rate:	0.826	0.605
MUX Reliability:	0.962	0.663
Retiring:	71.70%	57.80%
Front-End Bound:	14.60%	5.80%
Bad Speculation:	1.40%	0.30%
Back-End Bound:	12.30%	36.00%
Memory Bound:	5.00%	17.00%
Core Bound:	7.30%	19.00%
Divider:	0.00%	0.00%
Port Utilization:	29.70%	17.10%
Cycles of 0 Ports Utilized:	8.90%	4.80%
Cycles of 1 Port Utilized:	5.70%	3.70%
Cycles of 2 Ports Utilized:	17.30%	11.90%
Cycles of 3+ Ports Utilized:	58.40%	57.70%
Port 0:	70.10%	80.60%
Port 1:	70.40%	80.80%
Port 2:	45.40%	62.90%
Port 3:	46.50%	60.80%
Port 4:	3.70%	13.60%
Port 5:	4.40%	14.40%
Port 6:	7.50%	22.00%
Port 7:	1.70%	3.90%
Vector Capacity Usage (FPU):	100.00%	99.70%
Average CPU Frequency:	3.7 GHz	3.9 GHz
Total Thread Count:	13	11

Micro Architecture Analysis

Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate	Retiring »	Front-End Bound »	Bad Sp
► [Outside any known module]	15.636s 	57,396,500,000	84,112,300,000	0.682	80.6%	12.8%	
► _kmp_hyper_barrier_release	1.803s 	7,154,800,000	1,615,100,000	4.430	18.2%	6.3%	
► func@0x1401d21d1	1.077s 	3,964,900,000	2,858,200,000	1.387	46.7%	17.0%	
► NtDelayExecution	0.689s 	2,659,800,000	2,294,000,000	1.159	72.4%	40.4%	
► func@0x18066e9b6	0.362s 	1,357,800,000	1,453,900,000	0.934	62.6%	33.1%	
► KeDelayExecutionThread	0.244s 	970,300,000	564,200,000	1.720	23.2%	36.1%	
► func@0x1800ed5a6	0.199s 	765,700,000	725,400,000	1.056	0.0%	0.0%	
► func@0x1405e0460	0.195s 	744,000,000	992,000,000	0.750	94.1%	70.6%	
► SleepEx	0.183s 	703,700,000	62,000,000	11.350	32.0%	7.1%	
► func@0x1401c7610	0.111s 	455,700,000	83,700,000	5.444	38.4%	5.5%	
► func@0x180174c02	0.103s 	393,700,000	62,000,000	6.350	12.7%	0.0%	
► func@0x1403412a0	0.087s 	372,000,000	195,300,000	1.905	6.7%	0.0%	

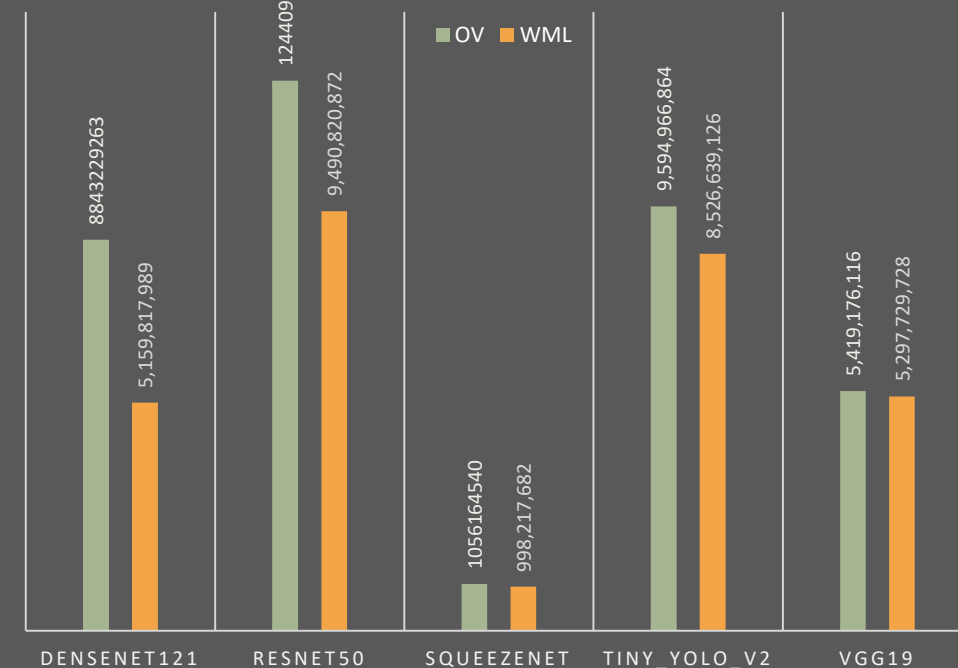
Solution: Resources Contention can be resolved by using more accumulators. This will help fill up the pipeline better.

Vectorization Analysis(*/mlas/lib/x86_64/SgemmKernelAvx256F.S*)

Number of AVX256 Instructions retired reported via event (FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE) are not very different between OpenVino and WindowsML.

AVX2 analysis underway

**AVERAGED AVX2 RETIRED EVENTS
ACROSS CORES**



Layout Optimizations

- Transpose sandwiches
 - Common in models converted from TF.
 - Majority of them can be removed using offline convertor tool.
- NHWC convolutions
 - MLAS Execution provider plans to add support for native NHWC convolutions in 19H2.
- To achieve peak performance on CPUs recommend using **nChw16c** for optimal vectorization and reduced cache thrashing.
 - Reduces number of reorders
 - Improved vectorization
 - Reduced Cache thrashing

Memory Bound Operator Analysis (VGG19)

This CPI is pretty high

Function / Call Stack	CPU Time	Instructions Retired	Microarchitecture Usage	
			Microarchitecture Usage	CPI Rate
[Loop at line 583 in MlasSgemmKernelAddAvx512F]	20.726s	166,994,800,000	70.9%	0.380
Eigen::internal::pload<union __m128>	14.509s	88,396,000,000	68.1%	0.523
[Loop at line 293 in MlasSgemmCopyPackB]	10.543s	65,128,000,000	67.2%	0.480
[Loop at line 830 in MlasSgemmKernelM1TransposeBAvx]	10.472s	5,381,600,000	4.6%	6.466
Eigen::internal::pstore<float,union __m128>	9.125s	49,991,200,000	58.2%	0.553
[Loop at line 176 in MlasConvIm2Col]	6.105s	39,779,600,000	68.9%	0.479
[Loop at line 582 in MlasSgemmKernelZeroAvx512F]	3.154s	17,122,000,000	49.5%	0.601
Eigen::internal::pset1<union __m128>	1.773s	22,167,600,000	100.0%	0.296
[Loop at line 69 in MlasBiasAdd]	1.445s	7,926,800,000	60.4%	0.553
[Loop at line 583 in MlasSgemmKernelAddAvx512F]	1.296s	5,712,000,000	68.8%	0.666
Eigen::internal::pmax<union __m128>	1.116s	6,865,600,000	34.4%	0.640
Eigen::internal::mapbase_evaluator<class Eigen::Block<class Eigen::Map<class Eigen::Matrix<float,-1,-	0.968s	5,171,600,000	40.4%	0.717
[Loop at line 583 in MlasSgemmKernelAddAvx512F]	0.964s	3,399,200,000	31.5%	0.851
Eigen::internal::binary_evaluator<class Eigen::CwiseBinaryOp<struct Eigen::internal::scalar_max_op<fl	0.957s	5,370,400,000	40.6%	0.735
[Loop at line 583 in MlasSgemmKernelAddAvx512F]	0.931s	2,842,000,000	44.7%	0.984
Eigen::internal::scalar_constant_op<float>::packetOp<union __m128>	0.912s	14,669,200,000	93.1%	0.259
[Loop at line 183 in MlasConvIm2Col]	0.892s	4,552,800,000	62.2%	0.579
Eigen::internal::padd<union __m128>	0.873s	3,701,600,000	52.8%	0.679
Eigen::internal::scalar_max_op<float,float>::packetOp<union __m128>	0.868s	5,320,000,000	45.9%	0.656
[Loop at line 156 in MlasConvIm2Col]	0.863s	5,516,000,000	63.0%	0.493
[Loop at line 583 in MlasSgemmKernelAddAvx512F]	0.786s	5,670,000,000	63.4%	0.415
[Loop at line 135 in MlasConvIm2Col]	0.781s	4,387,600,000	59.6%	0.474
[Loop at line 584 in MlasPool2DVectorKernel<struct MLAS_MAXIMUM_POOLING>]	0.695s	5,490,800,000	53.0%	0.527

Load and Bind Time Analysis

System Configuration

Processor	KBL i7 @ 3.60GHz
Core count	4C/ 8T
OS Build	19H1 – 18282
OS power plan	High Performance
Memory	64GB @ 2400MHz
Workload	WinMLRunner

Reference set memory analysis

- Observed up to 10x demand zero page faults when running WinMLRunner compared to other performance workloads.
- Narrowed down the demand zero faults to specific function calls on the stack:
 1. Windows.AI.MachineLearning.dll!onnxruntime::utils::TensorUtils::UnpackTensor<float>
 2. Windows.AI.MachineLearning.dll!std::basic_string<char,std::char_traits<char>,std::allocator<char>>::append
 3. Windows.AI.MachineLearning.dll!std::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string<char,std::char_traits<char>,std::allocator<char>> >
 4. Windows.AI.MachineLearning.dll!onnxruntime::Concat::Compute
- These faults were caused due to copies created by the above function.
- Copies observed to be the same size of the MapFile.

Reference set memory analysis - Example

The following is an example reference set analysis for Alexnet showing the top functions that cause the most number of demand zero faults.

These copies impact both the load time latency and the overall memory footprint of the process on the system.

Function call	Number of pages copied
Windows.AI.MachineLearning.dll!onnxruntime::utils::TensorUtils::UnpackTensor<float>	59536
Windows.AI.MachineLearning.dll!std::basic_string<char,std::char_traits<char>,std::allocator<char>>::append	59499
Windows.AI.MachineLearning.dll!std::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string<char,std::char_traits<char>,std::allocator<char>>	59492
Total	178,527

Opens:

Need for creating multiple copies as opposed to referencing the Map file.

Call stack for UnpackTensor

▼ WinMLRunner.exe (6504)				256,426	253,332
▼ Dynamic				193,554	7,738
VirtualAlloc				183,934	0.000
▼ [Root]				183,919	0.000
▼ - ntdll.dll!RtlUserThreadStart				183,919	0.000
kernel32.dll!BaseThreadInitThunk				183,491	0.000
▼ - WinMLRunner.exe!<PDB not found>				183,486	0.000
▼ - WinMLRunner.exe!<PDB not found>				183,486	0.000
WinMLRunner.exe!<PDB not found>				180,592	0.000
▼ - WinMLRunner.exe!<PDB not found>				121,093	0.000
- Windows.AI.MachineLearning.dll!winrt::impl::produce<winrt::Windows::AI::MachineLearning::factory_implementation::LearningModelSe...				121,093	0.000
Windows.AI.MachineLearning.dll!winrt::Windows::AI::MachineLearning::implementation::LearningModelSession::LearningModelSession				61,586	0.000
▼ - Windows.AI.MachineLearning.dll!onnxruntime::InferenceSession::Impl::Initialize				61,586	0.000
Windows.AI.MachineLearning.dll!onnxruntime::SessionStateInitializer::InitializeAndSave				61,586	0.000
Windows.AI.MachineLearning.dll!onnxruntime::SaveInitializedTensorsWithMemPattern				59,536	0.000
▼ - Windows.AI.MachineLearning.dll!onnxruntime::DeserializeTensorProto				59,536	0.000
Windows.AI.MachineLearning.dll!onnxruntime::utils::TensorProtoToMLValue				59,536	0.000
Windows.AI.MachineLearning.dll!onnxruntime::utils::GetTensorFromTensorProto				59,536	0.000
Windows.AI.MachineLearning.dll!onnxruntime::utils::GetTensorByTypeFromTensorProto<float>				59,536	0.000
Windows.AI.MachineLearning.dll!onnxruntime::utils::TensorUtils::UnpackTensor<float>				59,536	0.000
ucrtbase.dll!memcpy_repmove				59,536	0.000
ntoskrnl.exe!KiPageFault				59,536	0.000
ntoskrnl.exe!MmAccessFault				59,536	0.000
ntoskrnl.exe!MiDispatchFault				59,536	0.000
ntoskrnl.exe!MiResolveDemandZeroFault				59,536	0.000
ntoskrnl.exe!MiResolvePrivateZeroFault				59,536	0.000
ntoskrnl.exe!MiCompletePrivateZeroFault				59,536	0.000
ntoskrnl.exe!MiAllocateWsle				59,536	0.000
ntoskrnl.exe!MiLogAllocateWsleEvent				59,536	0.000
16	▼ File				62,872
17		MapFile	▸ n/a		59,957
18		Image	▸ n/a		2,823
19		MetaFile	▸ n/a		88
20		Prefetcher	n/a	▸ C:\Windows\Pr...	4

The above is the stack for the UnpackTensor call highlighting the copy and also the MapFile

The above is the stack for the std::allocator pointing out another set of copies of 59k pages.
The complete ETL trace attached here: <Path to ETL trace on Aspera>

Compute Shader Caching

- Load time decreases tremendously after setting the EnableComputeShaderCache reg key in the driver
- The driver looks for a compiled shader in the cache instead of compiling it again, thus decreasing the load time in subsequent runs
- Default enabling of EnableComputeShaderCache is a WIP in the driver.

Iterations	HLSL (previous results)	HLSL (ComputeShader cache enabled)
1	629.9 ms	1095.636 ms
2	572.9 ms	53.5535 ms
3	552.3 ms	50.5782 ms

Here, time in ms is the difference between the time(s) of CreateShaderCache task name and the last FindValue task

Operator Profiling

Convolution Operator Profiling – ResNet50

Input	Kernel	WinML	OpenVino	OpenVino (P)
{224,224}	{64,3,7,7}	1923.18	196.4	254
{56,56}	{256,64,1,1}	1114.00	57.65	78
{56,56}	{64,64,1,1}	419.27	127.7	161
{56,56}	{64,64,3,3}	1025.91	151.75	163
{56,56}	{256,64,1,1}	1017.91	113.9	132
{56,56}	{64,256,1,1}	847.27	105.3	135
{56,56}	{64,64,3,3}	1004.27	125.55	160
{56,56}	{256,64,1,1}	861.91	101.05	126
{56,56}	{64,256,1,1}	844.64	108.75	133
{56,56}	{64,64,3,3}	952.36	123.05	160
{56,56}	{256,64,1,1}	860.36	88.75	123
{56,56}	{512,256,1,1}	1088.18	174.1	190
{56,56}	{128,256,1,1}	859.45	158.45	185
{56,56}	{128,128,3,3}	1109.73	95.4	138
{28,28}	{512,128,1,1}	641.73	309.6	382
{28,28}	{128,512,1,1}	705.00	88.05	129
{28,28}	{128,128,3,3}	914.27	118.5	159
{28,28}	{512,128,1,1}	649.55	81.55	123
{28,28}	{128,512,1,1}	712.09	86.25	132
{28,28}	{128,128,3,3}	886.36	118.25	161
{28,28}	{512,128,1,1}	633.18	78.8	125
{28,28}	{128,512,1,1}	730.55	85.95	131
{28,28}	{128,128,3,3}	928.18	118.25	159
{28,28}	{512,128,1,1}	633.45	80.9	128
{28,28}	{1024,512,1,1}	1248.82	144.2	195
{28,28}	{256,512,1,1}	756.09	161.55	193

{28,28}	{256,256,3,3}	1972.00	80.9	130
{14,14}	{1024,256,1,1}	607.55	366.15	424
{14,14}	{256,1024,1,1}	762.18	89.55	131
{14,14}	{256,256,3,3}	1565.36	114.9	162
{14,14}	{1024,256,1,1}	601.09	72.75	130
{14,14}	{256,1024,1,1}	708.18	82.8	135
{14,14}	{256,256,3,3}	1624.18	121.4	154
{14,14}	{1024,256,1,1}	634.27	79.8	123
{14,14}	{256,1024,1,1}	676.18	87.95	137
{14,14}	{256,256,3,3}	1518.09	122	153
{14,14}	{1024,256,1,1}	605.27	75.85	124
{14,14}	{256,1024,1,1}	695.45	90.2	134
{14,14}	{256,256,3,3}	1545.82	124.75	156
{14,14}	{1024,256,1,1}	601.82	78.6	125
{14,14}	{256,1024,1,1}	673.36	88.55	131
{14,14}	{256,256,3,3}	1527.00	115.6	162
{14,14}	{1024,256,1,1}	619.73	72	126
{14,14}	{2048,1024,1,1}	999.36	144.45	195
{14,14}	{512,1024,1,1}	929.64	163.55	216
{14,14}	{512,512,3,3}	1605.09	123.6	163
{7,7}	{2048,512,1,1}	603.82	490.15	542
{7,7}	{512,2048,1,1}	603.09	139.95	179
{7,7}	{512,512,3,3}	1392.27	168.25	206
{7,7}	{2048,512,1,1}	582.64	112.7	149
{7,7}	{512,2048,1,1}	601.82	128.75	168
{7,7}	{512,512,3,3}	1389.82	152.95	201
{7,7}	{2048,512,1,1}	591.27	104.95	144

Convolution Comparison

- OpenVino – R5
- WinML – 18323
- Extracted Conv operator from resnet50 using ConvertOnnxModel tool
- Created xml files for OpenVino using Model Optimizer

Kernel	WinML (FPS)	OpenVino (FPS)
1x1	1482.31	7905.76
3x3	461.98	3571.05
7x7	416.91	2391.83

- Note:
 - Reorders observed during execution

Performance counts (all time in microseconds):

Layer	Prec	Status	Type	Total	Avg batch	Avg image	Kernel
data_0	FP32	NOT_RUN		0	0	0	
output_0	FP32	EXECUTED	Convolution	10505	105	105	jit_avx512_FP32
output_0_nChw16c_nchw_out_output_0	Σ&A·B	EXECUTED	Reorder	14529	145	145	reorder_FP32
out_output_0	Σ&A·B	NOT_RUN	Output	0	0	0	
				25034	250	250	

Graph Optimizations

Per-Operator Execution Time (us) – SKL Server

		Conv	BN	Sum	Add	MaxPool	Relu	AvgPool	SoftMax	Mul	Concat	Gemm	Scaleshift	Reorder	FC
ResNet50	W	999133	0	891636	0	112019	1629087	5928	2407	0	0	14060	0	0	0
	O	159779	0	0	0	1621	0	0	3715	0	0	0	0	721	5698
DenseNet121	W	922400	2995921	0	2257213	125602	2639789	129271	0	2253212	74416	0	0	0	0
	O	150914	0	0	0	3912	41039	0	0	0	0	0	44233	41892	0
VGG19	W	1110301	0	0	0	549897	2417144	0	2643	0	0	169637	0	0	0
	O	392742	0	0	0	7255	1309	0	572	0	0	0	0	731	152705
SqueezeNet	W	176897	0	0	0	196633	449294	8579	2268	0	9057	0	0	0	0
	O	25948	0	0	0	3266	0	0	2196	0	0	0	0	1231	
TinyYolo	W	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	O	82128	0	0	0	6342	0	0	0	0	0	0	0	1486	0

Per-Operator Execution Time – KBL NUC

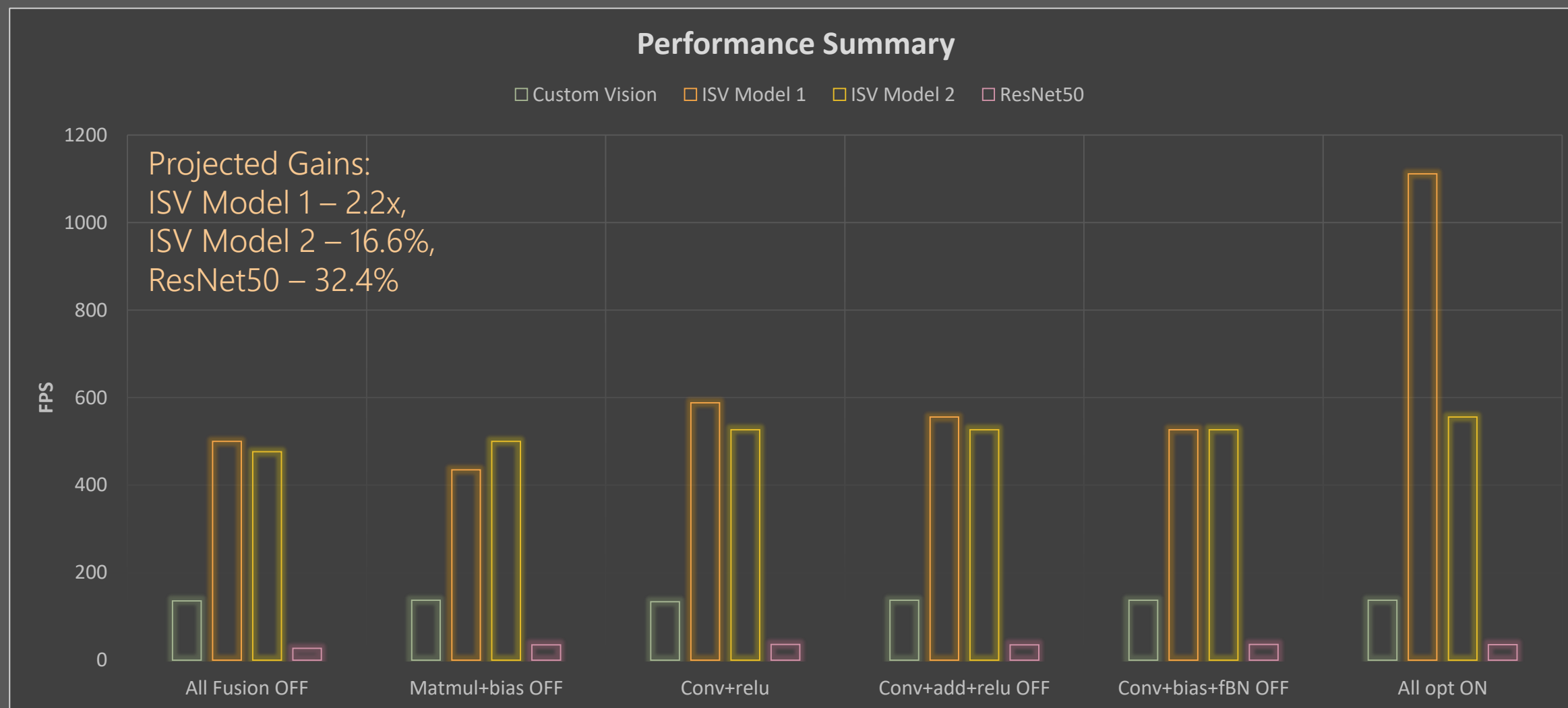
		Conv	BN	Sum	Add	MaxPool	Relu	AvgPool	SoftMax	Mul	Concat	Gemm	Scale shift	Reorder	FC
ResNet50	W	645115	0	47519	0	6271	46241	350	339	0	0	7345	0	0	0
	O	422065	0	0	0	2577	0	0	7406	0	0	0	0	213	19246
DenseNet121	W	648081	64383	0	60224	6691	72405	6790	0	72136	40246	0	0	0	0
	O	285319	0	0	0	4288	11235	0	0	0	0	0	19865	18175	0
VGG19	W	2157027	0	0	0	49347	80003	0	342	0	0	339868	0	0	0
	O	1818303	0	0	0	17791	218	0	499	0	0	0	0	201	348769
SqueezeNet	W	98020	0	0	0	10681	10124	0	529	0	4024	0	0	0	0
	O	36230	0	0	0	2355	0	0	7199	0	0	0	0	92	0
TinyYolo	W	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	O	358971	0	0	0	15943	0	0	0	0	0	0	0	495	0

Fusions

- Learnings based on studies on ngraph and OpenVino.

Fusion Type	Status
MatMul+Bias	POC under test for CPU fusion
Conv+BN+Relu	Under evaluation
Conv+Relu	Under evaluation
Conv+Bias+BN	Enabled in Lotus graph transformer – 19H1
Conv+Relu+EltWise	Under evaluation
Conv+Scale+Bias	Enabled in Lotus graph transformer – 19H1
Conv+Concat	Under evaluation
Transpose+Matmul	Under evaluation
BN+Relu	Enabled only for GPU – 19H1
Eltwise fusion	Under evaluation
MultiLayerRNN	Under evaluation
ConvTranspose+Act	Enabled only for GPU – 19H1
InstanceNorm+Act	Enabled only for GPU – 19H1
MeanVarNorm+Act	Enabled only for GPU – 19H1
Gemm+Act	Enabled only for GPU – 19H1
Matmul+Act	Enabled only for GPU – 19H1

Projected Impact of Key Fusions



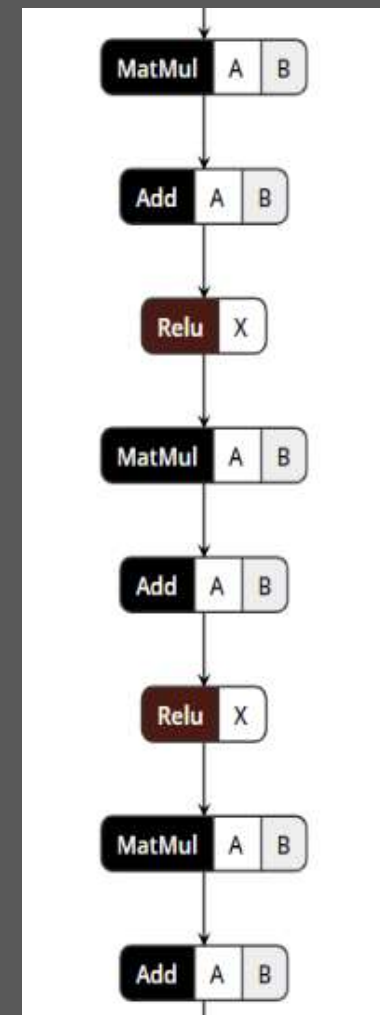
Layout optimization cannot be turned off in nGraph

Redundant fusions help ResNet perf: disable all variants of conv+x to assess impact (slide 10)

Fusion: MatMul + Add

- Source: ISV Model 1
- Status: POC implemented, however gains not realizable for WindowsML.
- Conclusion : Masking inefficiencies in MLAS kernels impede achieving almost **2x perf gain** as noted in other frameworks.

Perf	Enabled	Disabled	Gain (%)
FPS	1111.11	434.78	155



GPU Perf Analysis

HLSL Perf Regression

- Verified extended runtimes in PIX for most operators
- Shader dump analysis underway
- Suspect buffer type mismatch between DML and WinML

Models	GT2 (RS5)	GT2 (19H1)	GT3 (RS5)	GT3 (19H1)
Alexnet	3.58	5.83	5.42	9.83
Densenet121	3.11	4.56	5.15	7.73
Densenet121 FP16	3.07	2.03	5.21	3.14
Emotion Ferplus	15.93	21.13	23.88	35.03
Emotion Ferplus FP16	19.79	8.17	34.47	12.05
FNS-Candy	0.72	0.79	1.26	1.43
FNS-Candy FP16	0.68	0.72	1.22	1.31
Inception v1	11.61	14.85	16.68	24.73
Inception v1 FP16	11.36	5.40	18.54	8.15
Inception v2	5.95	9.94	9.49	16.72
Inception v2 FP16	5.81	3.20	9.57	4.82
Inception v3	3.29	3.93	5.15	6.24
Resnet50	4.95	6.62	7.48	10.70
Resnet50 FP16	4.81	3.14	7.90	4.86
Squeezenet	48.02	64.88	68.77	101.07
Squeezenet FP16	44.26	23.45	68.20	33.75
Tiny Yolo v2	7.08	8.27	11.85	14.28
Tiny Yolo v2 FP16	5.99	6.54	10.76	11.58
Vgg19	0.91	1.15	1.50	2.05

Multi-Model Scenarios

2 concurrent models

	Resnet	ResNet+DenseNet	Delta	Densenet	ResNet+DenseNet	Delta
Load	81.6	106.1	30%	50.4	78.5	56%
Bind	0.07	0.11	57%	0.08	0.12	50%
Eval	36.49	57.96	59%	54.33	78.75	45%

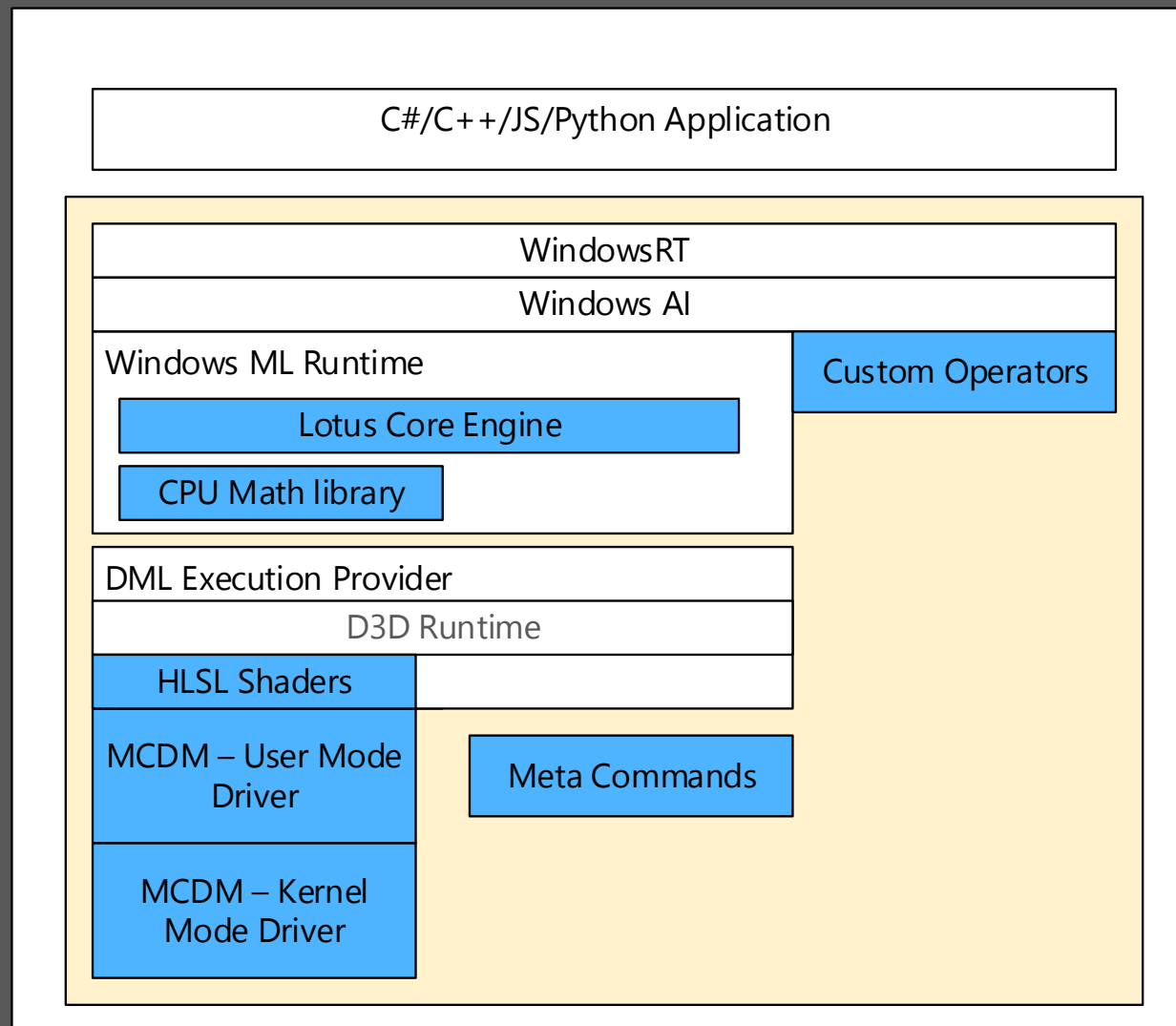
	Resnet	ResNet+SqueezeNet	Delta	Squeezenet	ResNet+SqueezeNet	Delta
Load	81.6	106.1	30%	14.8	21.9	48%
Bind	0.07	0.11	57%	0.06	0.11	83%
Eval	36.49	55.96	53%	6.38	9.93	56%

Thank you!

Number of retired AVX 512 instructions per Core (Resnet-50)						# of Core 14 HT-Disabled											
(FP_ARITH_INST_RETIRED.512B_PACKED_SINGLE)																	
Densenet121	9,292,378,060	4,263,542,926	4,341,669,208	4,353,326,372	98,205,530	44,263,116	14,863,847	4,319,607,416	63,246,587	4,449,138,094	3,094,688	449,232	4,299,778,141	4,339,647,506	4,454,696,637	35,045,529,300	2,503,252,093
Resnet50	9,321,839,853	6,161,726,008	6,154,340,980	6,116,100,700	282,323,849	87,410,574	26,871,337	6,167,829,877	26,890,108	6,158,621,408	3,010,071	806,537	6,104,929,298	6,175,771,975	6,093,331,078	49,559,963,800	3,539,997,414
Squeezenet	9,336,352,342	508,344,598	532,740,328	529,148,030	508,532	0	43,602	542,841,350	561,750	538,145,508	43,602	0	532,702,998	531,127,200	542,048,202	4,258,255,700	304,161,121
Tiny_yolo_v2	9,313,734,434	5,010,614,311	4,873,215,752	4,996,025,242	4,913,646,430	5,032,936,120	5,027,890,949	244,568,226	134,275,922	41,254,914	0	0	4,951,898,661	91,142,197	4,982,380,076	40,299,848,800	2,878,560,629
VGG19	9,325,753,962	2,973,962,634	2,974,006,962	2,973,820,818	144,078,092	0	0	2,943,963,484	0	2,974,017,610	31,139,120	885,676	2,974,027,088	2,973,864,204	2,829,857,252	23,793,622,940	1,699,544,496
Densenet121	9,298,625,760	1,780,713,040	1,837,672,337	1,765,761,917	1,763,244,780	1,804,523,606	1,828,639,901	1,775,486,543	2,663,059,500	2,113,422,132	923,264,640	839,231,681	2,214,005,253	2,049,512,215	2,008,865,599	25,367,403,144	1,811,957,367
Resnet50	9,327,518,745	4,132,036,800	4,146,850,813	4,166,037,419	4,110,830,519	4,154,386,753	3,546,083,571	4,125,105,588	4,181,474,205	4,129,359,084	3,701,582,680	2,911,722,376	4,166,923,824	4,163,725,429	4,137,612,503	55,773,731,564	3,983,837,969
Squeezenet	9,324,176,766	339,947,836	340,195,530	340,576,332	688,506,908	393,982,932	333,698,824	227,049,786	314,554,956	301,254,880	108,733,220	83,611,528	411,763,956	348,782,524	325,346,760	4,558,005,972	325,571,855
Tiny_yolo_v2	9,316,518,111	3,284,807,610	3,357,092,396	3,350,912,318	3,308,973,883	3,277,716,168	3,362,836,880	3,358,183,604	3,345,224,086	3,306,946,294	3,090,305,449	2,978,213,247	3,369,144,650	3,377,512,042	3,006,814,760	45,774,683,387	3,269,620,242
VGG19	9,312,167,112	1,986,512,548	1,515,574,416	1,996,029,998	2,029,717,570	2,014,465,100	2,035,501,956	2,032,299,672	2,032,837,142	2,033,055,532	2,021,037,714	1,436,687,086	2,033,245,246	1,982,552,534	2,002,358,258	27,151,874,772	1,939,419,627

WindowsML Scenarios

Windows AI Stack

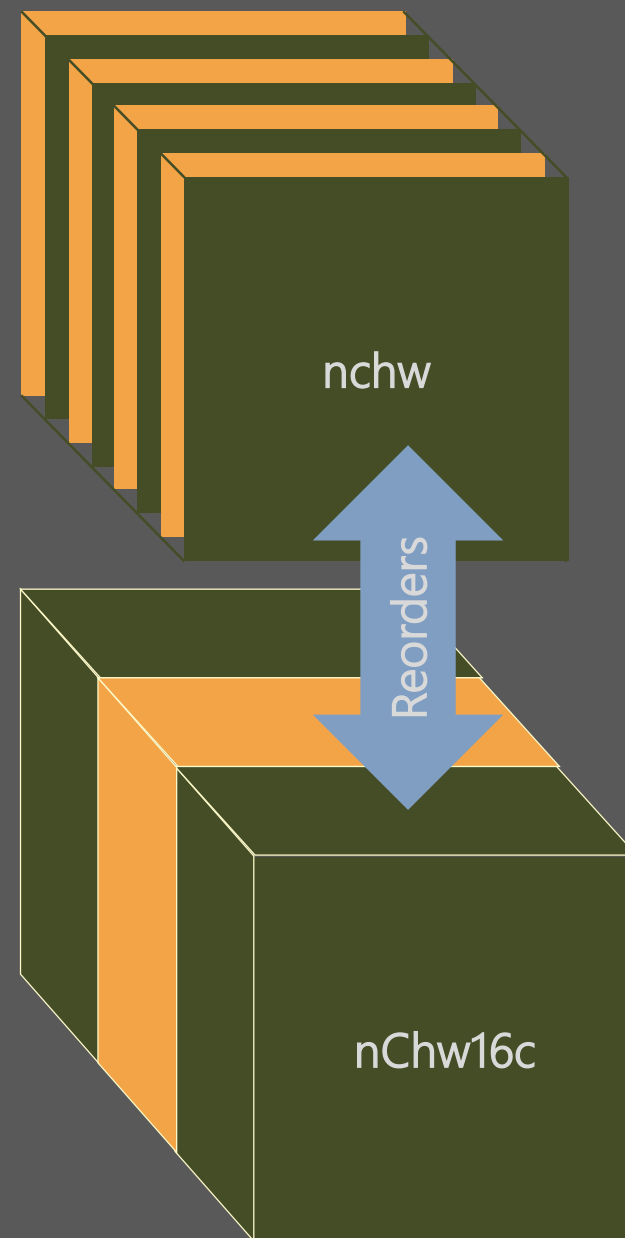


Breakdown of Optimization Opportunities

- Load and bind time: Identify system performance in pre-inference stages which may also be affected by graph restructuring/compilation.
- Inference Time
 - Graph optimizations: Identify opportunities for restructuring to improve inference performance. Further split into
 - hardware-agnostic (Constant folding, NOP removal, algebraic simplification, batch norm fusion)
 - Capability-driven (Operator Fusions, Layout optimizations)
 - Hardware-specific techniques (threading)
 - Operator optimizations: Identify algorithm tweaks to take advantage of libraries and HW capabilities
 - Winograd
 - BLAS level optimizations:
 - vectorization, better LLC
 - threading,
 - removing hotspots

Memory layouts

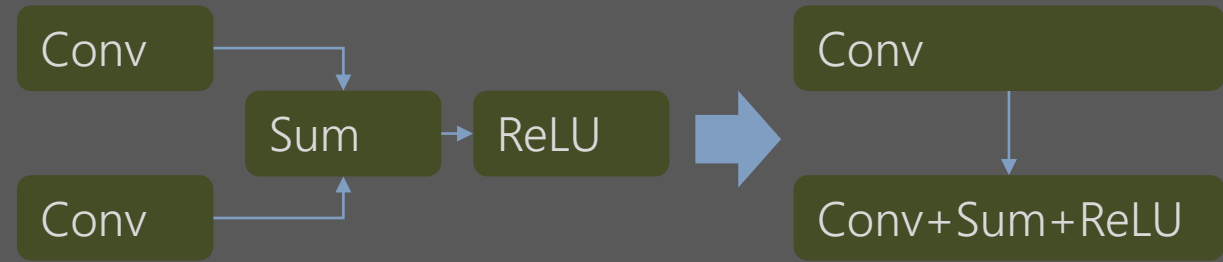
- Most popular memory layouts for image recognition are **nhwc** and **nchw**
 - Directly supported by cuDNN
 - Internal transformations cheaper due to high BW
 - Challenging for IA either for vectorization or for memory accesses (cache thrashing)
- MKL-DNN convolutions use blocked layouts
 - Example: **nChw16c** for single-precision AVX512
 - Convolutions define which layouts are to be used by other primitives
 - FWKs track memory layouts and perform reorders **only** when necessary



Source: MKL Team
Intel MTC Intelligent Compute Team

Fusing computations

- DL workloads may spend high % of time in BW-limited ops
 - ~40% of ResNet-50, even higher for inference
- The solution is to fuse BW-limited ops with convolutions or one with another to reduce the # of memory accesses
 - Conv+ReLU+Sum, BatchNorm+ReLU, etc
 - Done for inference, WIP for training



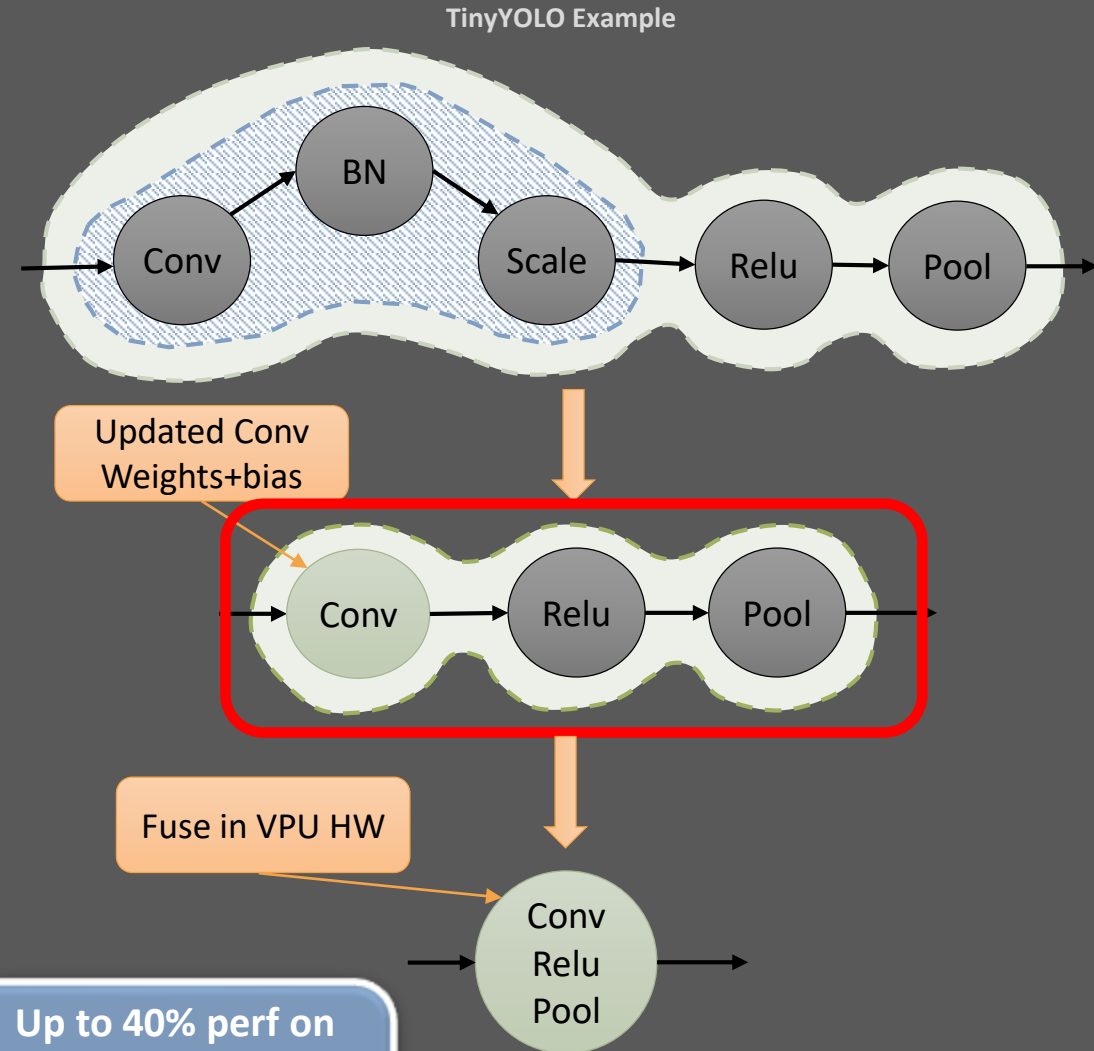
- The FWKs are expected to be able to detect fusion opportunities
 - IntelCaffe already supports this
- Major impact on implementation
 - All the impls. must be made aware of the fusion to get max performance
 - MKL-DNN team is looking for scalable solutions to this problem

Operator Fusion

- Description: Enables HW Fusion of an operator if all the inputs and outputs are known. Also allows stride based fusion so an entire sequence of commands can be run across the data in parallel, further enabling improved performance and optimizing internal caches.

The solution is to fuse BW-limited ops with convolutions or one with another to reduce the # of memory accesses

Conv+ReLU+Sum, BatchNorm+ReLU, etc

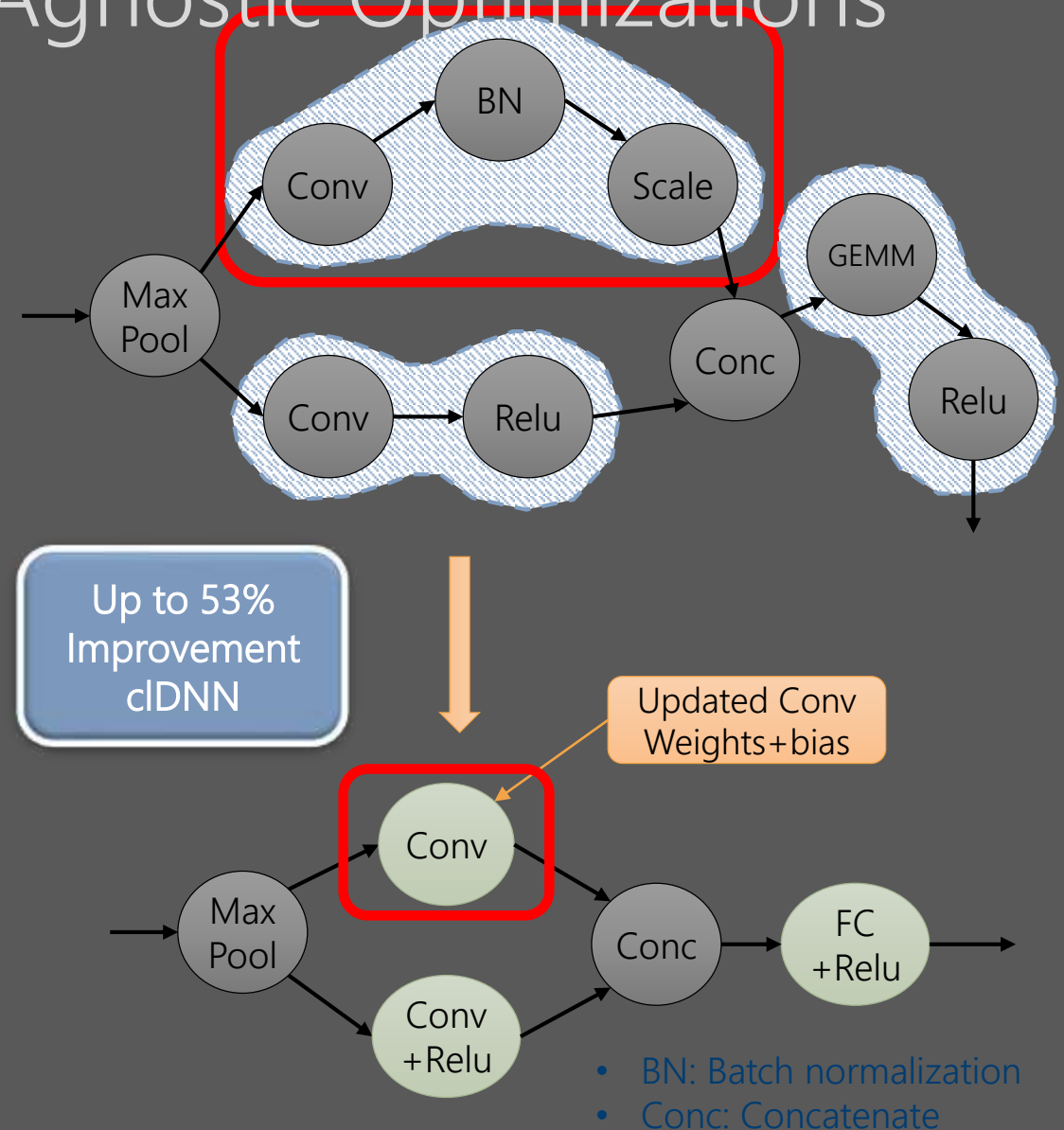


Up to 31% perf on
VPU
With pooling fuse
for tiny yolo

Up to 40% perf on
CPU
With pooling fuse
for tiny yolo

Microsoft to perform HW Agnostic Optimizations

- Description: OS performs operations on the weights to eliminate Scale and BatchNorm operations.
- Details: WinML can fold Scale and BatchNorm operations into other operations to eliminate other nodes.
- This optimization alongside Fusion and Concatenate optimizations show a 53% performance improvement to model evaluation across all cIDNN evaluated models.

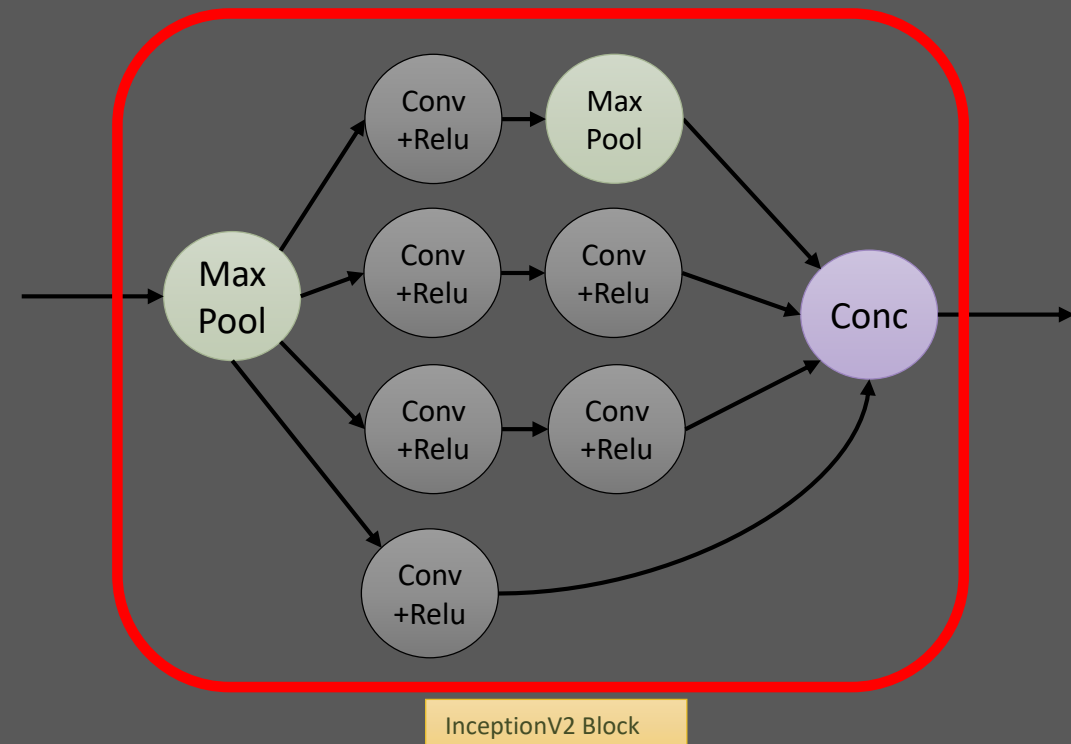
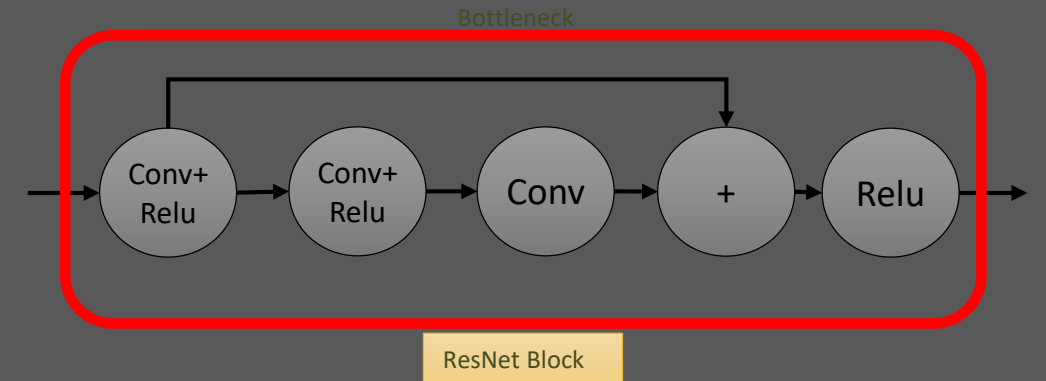


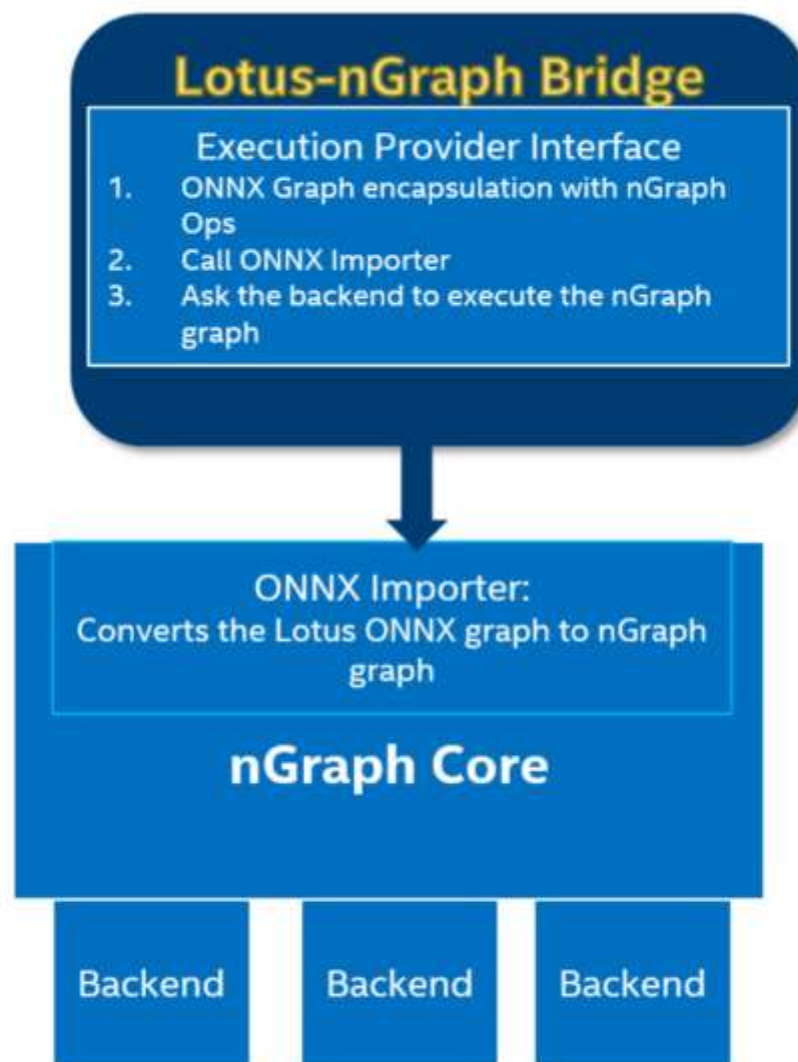
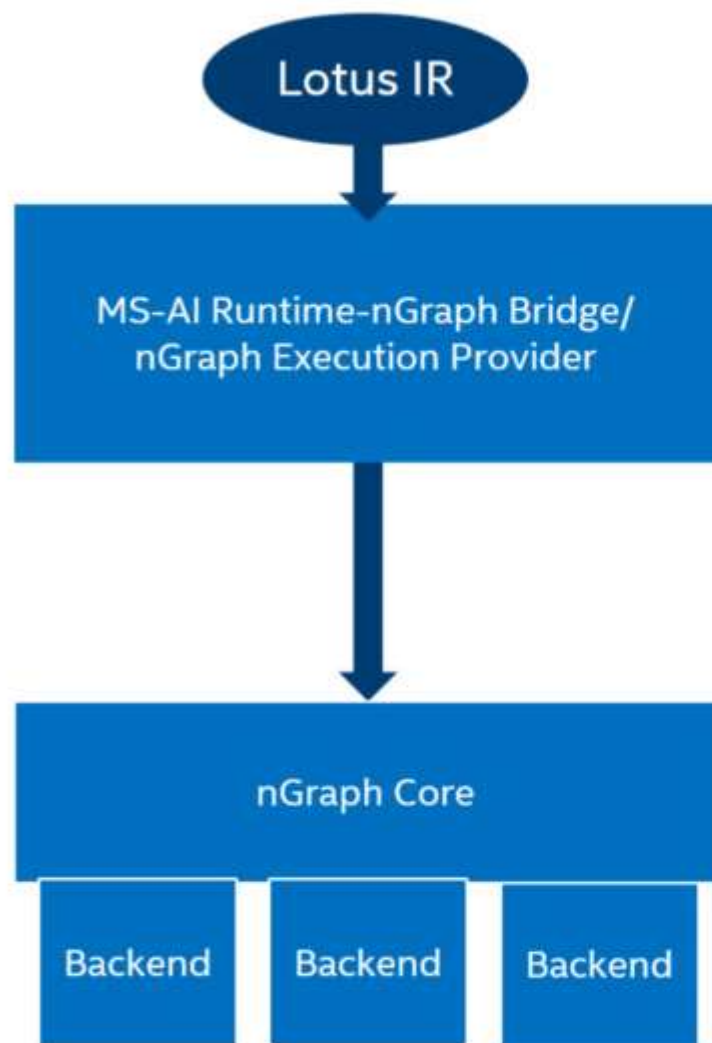
Stride and Vertical Fusion

Details: This will allow for Vertical and Stride Fusion of the functional blocks, to help allow for better utilization of the HW, including limited cache sizes, etc. Also helps enable the hiding of Inter-Layer tensor visibility.

Fusion opportunities already exposed are shown, this would enable the use of more HW specific fusion opportunities.

Example performance numbers are already available through RS5 Meta Command support... further improvements are anticipated with Function Level Support.





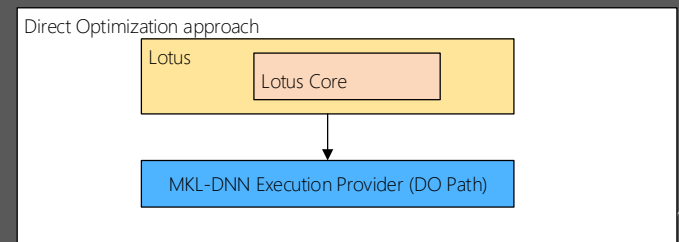
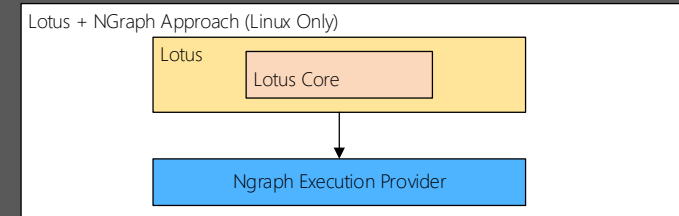
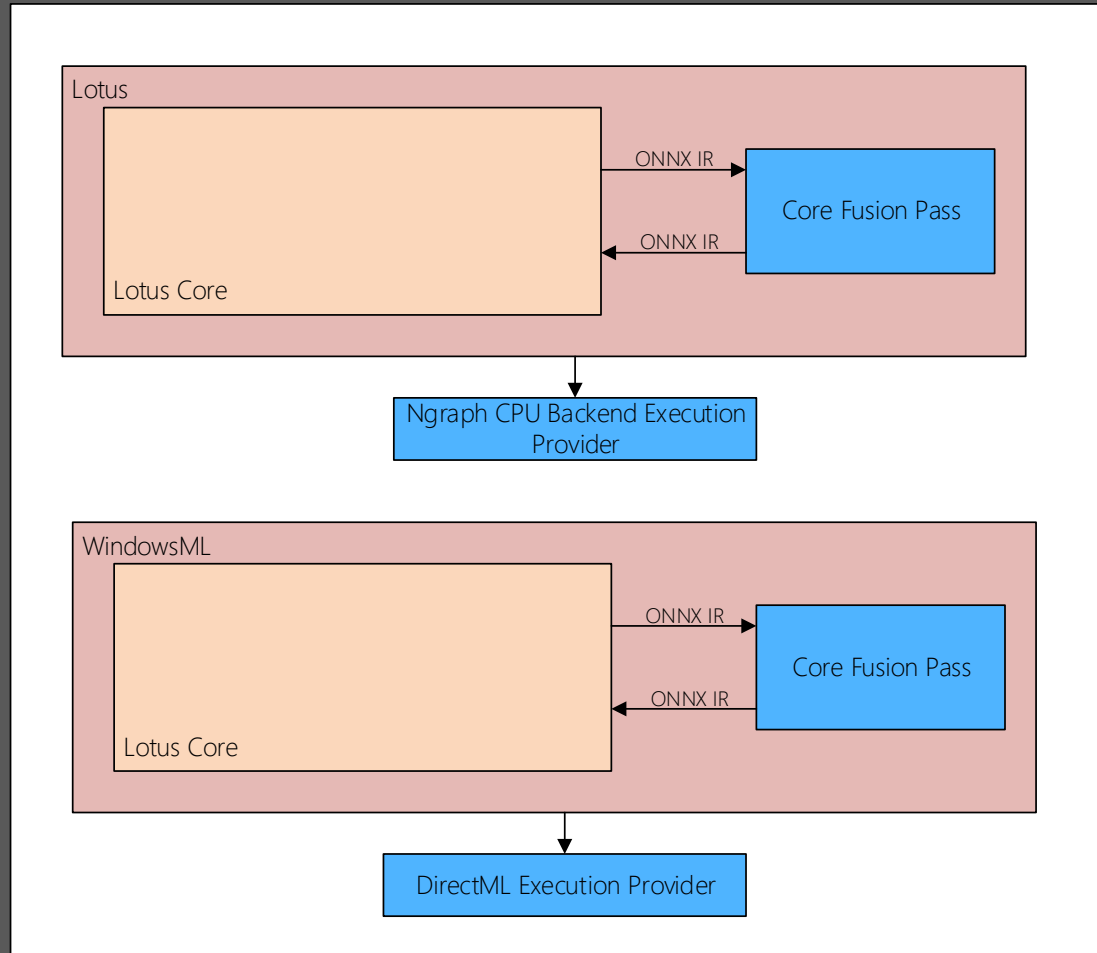
Bridge / EP Functionality

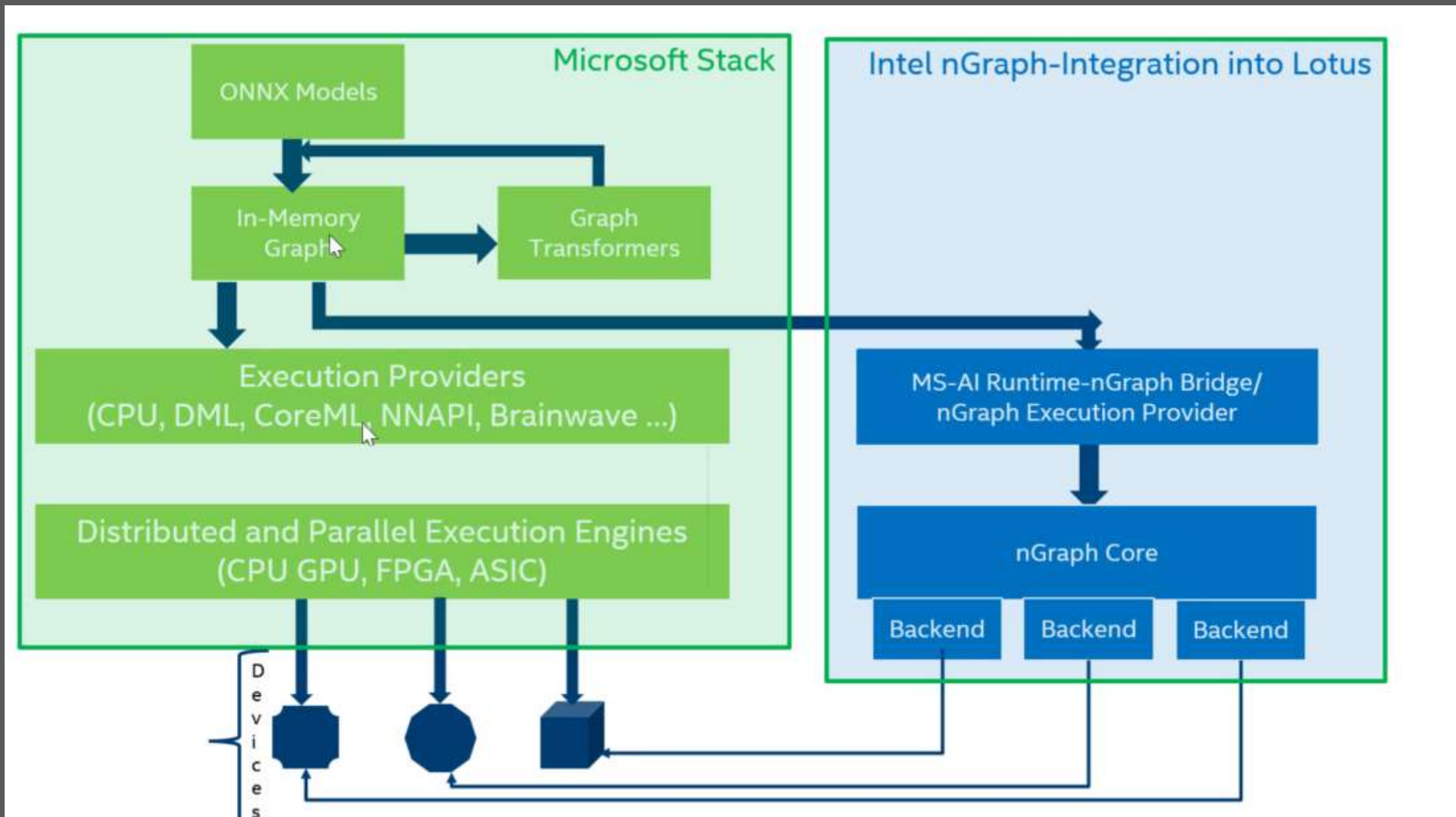
1. ONNX graph transformation to a "custom nGraph ONNX op" that encapsulates all the Ops supported by nGraph for that model
2. Custom Op graph returned to Lotus.
3. Register Ops supported by nGraph
4. Original ONNX graph → ONNX importer → nGraph graph/model
 - a. Models are stored in the model protobuffer along with the metadata of the graph.
5. EP then calls the backend to "compute" the model on the backend device
6. EP receives the results of the compute

Steps to implement a optimization pass

- Match patterns in operations and locate a list of potentially-transformable subgraphs in the given graph.
- Transform the selected candidates into semantically-equivalent subgraphs that execute faster, or with less memory (or both).
- Verify that the optimization pass performs correctly, with any or all expected transformations,
- Measure and evaluate your performance improvements

TR Proposal (Linux and Windows)





Model Optimizer pipeline

