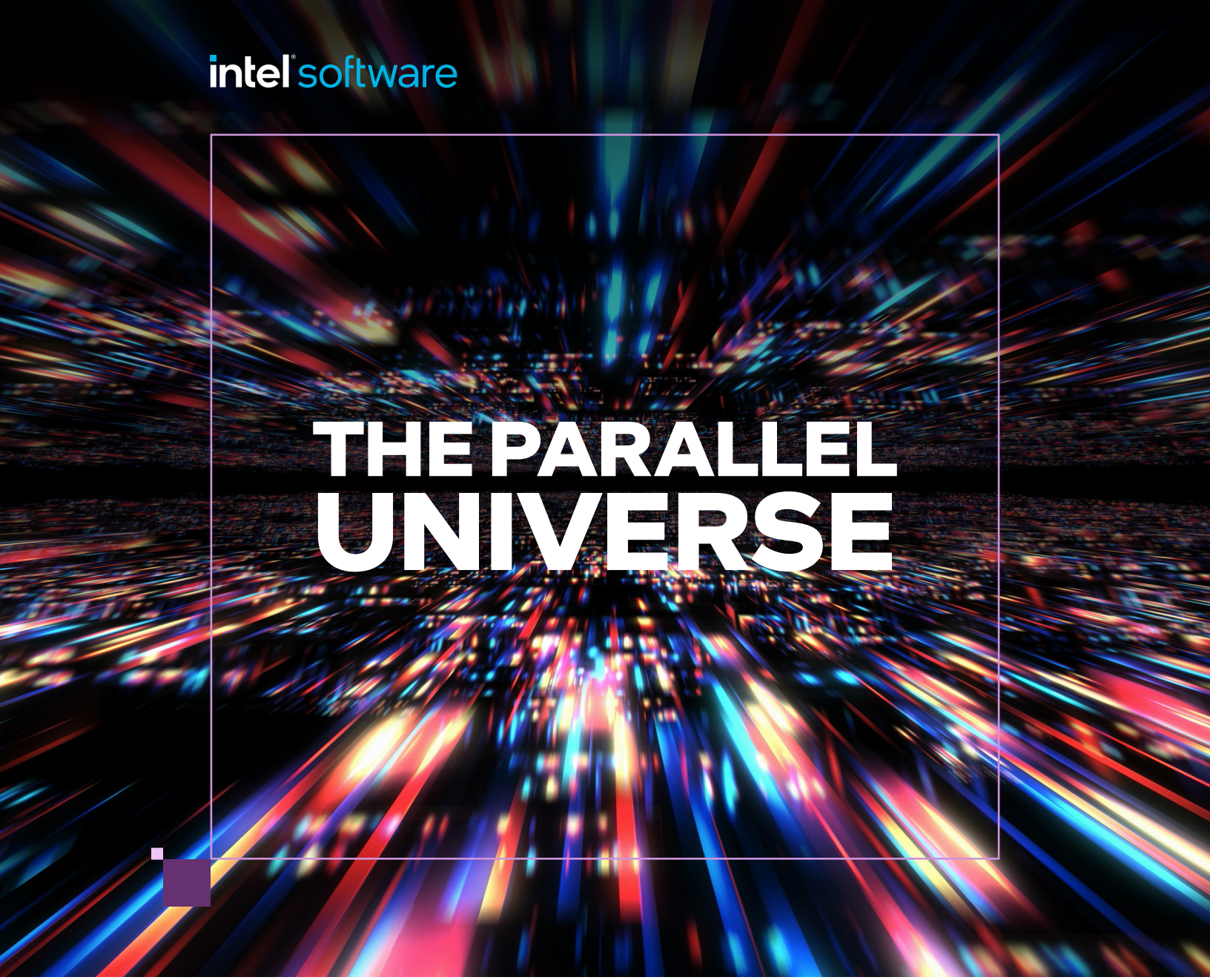


Winning the NeurIPS Billion-Scale Approximate Nearest Neighbor Search Challenge

Optimizing End-to-End Artificial Intelligence Pipelines

Better Artificial Intelligence Performance with Hyperparameter Tuning

Issue
48
2022

Contents

Sign up for future issues

Letter from the Editor

Henry A. Gabb, Senior Principal Engineer at Intel Corporation, is a longtime high-performance and parallel computing practitioner who has published numerous articles on parallel programming. He was editor/coauthor of "Developing Multithreaded Applications: A Platform Consistent Approach" and program manager of the Intel/Microsoft Universal Parallel Computing Research Centers.

Elephants in the SYCL Room

Once upon a time, I was a committed Perl programmer. Then one day, a colleague suggested that I take a look at Python, his favorite new language. "Pshaw," I said. "Why do we need another programming language?" Twenty years later, I'm committed to Python (though I admit that I'm attracted to Julia). The SYCL* specification from the Khronos Group brings heterogeneous data parallelism to C++, but it's facing the same question: Do we really need another programming language? The oneAPI industry initiative adopted SYCL as its direct programming approach for heterogeneous parallelism, so James Reinders (Editor Emeritus, *The Parallel Universe*) and Michael Wong (Distinguished Engineer, Codeplay Software) provide an excellent response to this question and several others in Why SYCL: Elephants in the SYCL Room.

We have several oneAPI and SYCL articles in this issue, but our feature article describes the winning solution to a recent data science competition: **Winning the NeurIPS Billion-Scale Approximate Nearest Neighbor Search Challenge**. This is followed by two more data science articles: **Optimizing End-to-End Artificial Intelligence Pipelines** and **Optimizing Artificial Intelligence Applications**. Artificial intelligence is the glamorous part of analytics pipelines, but data scientists know that it takes a lot of hard work and computation to reach this step. These articles describe how to optimize various parts of the analytics pipeline, up to and including artificial intelligence.

From data science, we turn our attention back to SYCL. James Brodman and John Pennycook, coauthors of the recent book, Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL, describe their top **Five Outstanding Additions Found in SYCL 2020**.

As I mentioned in the last issue of *The Parallel Universe*, I've been experimenting with the ArrayFire heterogeneous parallel library. In this issue, Umar Arshad (ArrayFire, Software Engineer) and I do a side-by-side comparison of the oneAPI and ArrayFire abstractions in **Accelerating the 2D Fourier Correlation Algorithm with ArrayFire and oneAPI**. Readers may also be interested in ArrayFire Interoperability with oneAPI, Libraries, and OpenCL (*The Parallel Universe*, Issue 47).

We close this issue with two articles on the oneAPI Data Parallel C++ Library (oneDPL): **The Maxloc Reduction in oneAPI** and **More Productive and Performant C++ Programming with oneDPL**. These articles describe how to use C++ STL functions in oneDPL for better programmer productivity and heterogeneous parallelism.

As always, don't forget to check out intel.com/oneapi for more information on Intel solutions for code modernization, visual computing, data center and cloud computing, data science, systems and IoT development, and heterogeneous parallel programming with oneAPI.

Henry A. Gabb
April 2022

Winning the NeurIPS Billion-Scale Approximate Nearest Neighbor Search Challenge

Unleashing Intel® Xeon® Processors with Intel® Optane™ Technology to Drastically Improve Search Performance

Mariano Tepper, Cecilia Aguerrebere, and Ted Willke, Intel Labs; Sourabh Dongaonkar and Jawad B Khan, Intel Foundry Services; and Mark Hildebrand, University of California, Davis

Similarity search, also known as approximate nearest neighbor (ANN) search, is the backbone of many AI applications that require search, recommendation, and ranking operations on web-scale databases. Accuracy, speed, scale, cost, and quality-of-service constraints are critical. In this article, we describe a solution that advances these dimensions by leveraging the computational capabilities of Intel® Xeon® processors and Intel® Optane™ memory. To showcase these advances, we participated in the NeurIPS'21 Billion-Scale ANN Search Challenge, winning the Custom Hardware Track. Our results offer an 8x to 19x reduction in CAPEX and five-year OPEX at iso-performance over the next-best solution. This promises to drastically lower the entry barrier and democratizes similarity search in the modern, large-scale, high-accuracy and high-performance scenario.

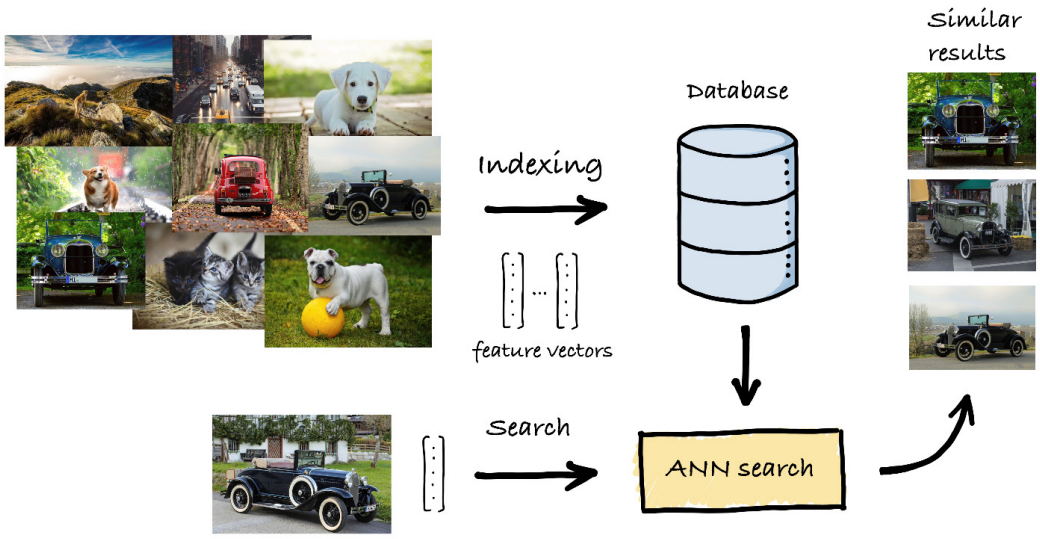


Figure 1. Schematic representation of an ANN search pipeline.

Approximate Nearest Neighbor Search

Given a database of high-dimensional feature vectors and a query vector of the same dimension, the objective of similarity search is to retrieve the database vectors that are most similar to the query, based on some similarity function (**Figure 1**). In modern applications, these vectors represent the content of data (images, sounds, text, etc.), extracted and summarized using deep learning systems such that similar vectors correspond to items that are semantically related.

To be useful in practice, a similarity search solution needs to provide value across different dimensions:

- **Accuracy:** The search results need to be of sufficient quality to be actionable (that is, the retrieved items need to be similar to the query).
- **Performance:** The search needs to be fast, often meeting strict quality-of-service constraints.
- **Scalability:** Databases continue to get larger in terms of the number of items they contain and the dimensionality of those items.
- **Cost:** Being deployed in production and data center scenarios, the solution needs to minimize the total cost of ownership (TCO), often measured as a combination of capital expenditures (CAPEX) and operating expenses (OPEX).

A natural solution is to linearly scan over each vector in the database, compare it with the query, rank the results in descending order of similarity, and then return the most similar vectors. However, the sheer volume and richness of data preclude this approach and make large-scale similarity search an extremely challenging problem that is both compute- and memory-intensive. Better solutions are needed, which commonly involve two phases:

1. During *indexing*, each element in the database is converted into a high-dimensional vector, and then an index is created so that only a fraction of the database is accessed during the search.
2. At *search* time, given a query vector, an algorithm sifts through the database using the index. Its results are used to take different informed actions depending on the final application and based on these semantically relevant results.

The NeurIPS'21 Billion-Scale Approximate Nearest Neighbor Search Challenge

In December 2021, the first billion-scale similarity search competition was organized as part of the NeurIPS conference. The goal of the competition was to provide a comparative understanding of astate-of-the-art similarity search across a curated collection of real-world datasets and to promote the development of new solutions. We participated in the competition's Custom Hardware Track, where we could take full advantage of Intel's hardware offerings. We developed a solution that fully leveraged the capabilities of Intel Xeon processors and Intel Optane persistent memory (PMem), creating a one-two approach that eventually won the competition.

The fundamental metric compared across datasets was TCO, defined as CAPEX + five-year OPEX of the solutions at 90% recall and 100,000 queries-per-second (QPS) throughput. The CAPEX and OPEX are defined by the competition organizers as follows:

- CAPEX = (MSRP of all the hardware components) x (minimum number of systems needed to scale to support 100,000 QPS)
- OPEX = (maximum QPS at or greater than the baseline recall @10 threshold) x (kilowatt-hour/query) x (seconds/hour) x (hours/year) x (five years) x (dollars/kilowatt-hour) x (minimum number of systems needed to scale to support 100,000 QPS)

These metrics balance the energy efficiency (through OPEX) and raw performance (through CAPEX) for each solution.

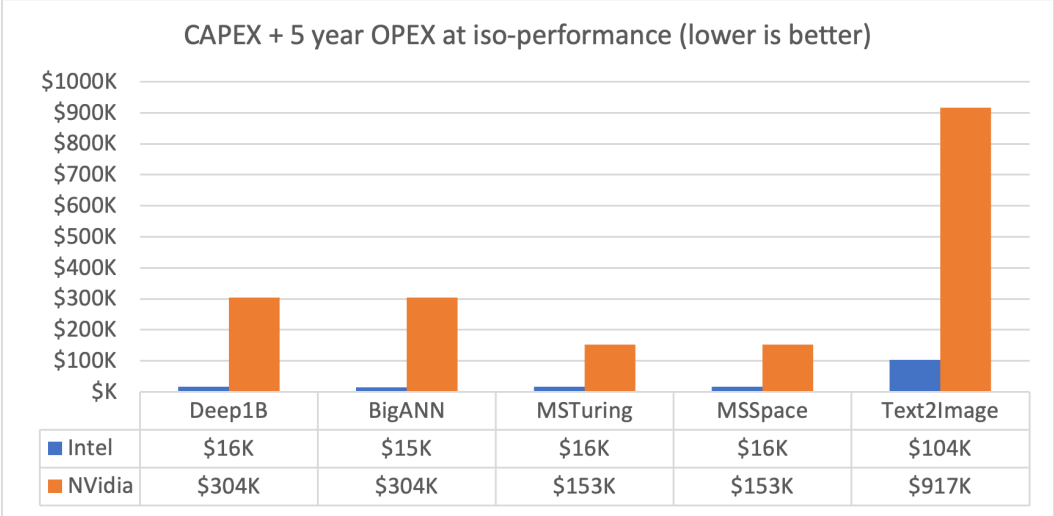


Figure 2. TCO difference between the winning Intel solution and the second place NVIDIA solution, showing up to 20x improvement across five different datasets (x-axis).

Our solution offers a breakthrough improvement in TCO and is between 8.85x and 19.7x better than the second-best solution, NVIDIA's *cuanns_multigpu* that uses a DGX A100 GPU system across multiple datasets (**Figure 2**). The stark difference in efficiency of our solution is apparent when comparing the hardware configurations of our first-place entry and NVIDIA's second-place entry (**Table 1**). A single inexpensive 1U 2S Intel Xeon server with Intel Optane PMem can achieve the same performance as two of NVIDIA's flagship DGX A100 servers with eight A100 GPUs and two 64-core CPUs for ANN search workloads.

	Intel® Xeon® processor + Intel® Optane™ memory	**NVIDIA DGX A100**
CPU	Dual Intel Xeon Gold 6330N processors 56 cores total	Dual AMD Rome 7742 128 cores total
System memory	512GB DDR4 2TB Intel Optane DCPMM 200 Series	2TB DDR4
GPU	None	8x NVIDIA A100 80 GB GPUs
GPU memory	None	640 GB
Power	Up to 1.2 kW	Up to 6.5 kW
Total cost	$14,664	$150,000+

Table 1. Comparing Intel and NVIDIA hardware configurations for the BigANN competition. These configurations achieve similar performance.

In addition to the significantly low CAPEX of the Intel solution at iso-performance, the power efficiency is also significantly better, as shown by the energy per query (in Joules), measured by standard IPMI interface on all machines in the competition. The energy per query of the Intel solution is up to 5x better than the NVIDIA solution (**Figure 3**). This translates to much better OPEX over a long period, as well as a much more sustainable solution to the ANN search problem.

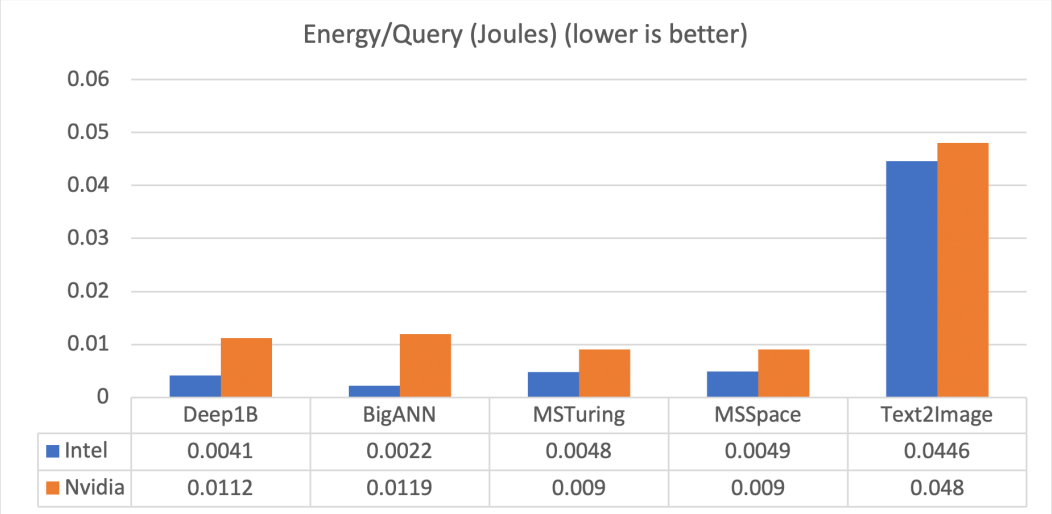


Figure 3. Energy efficiency of the Intel® solution is up to 5x better than the NVIDIA solution across all datasets.

These drastic improvements in TCO are enabled by unique advantages of Intel Xeon processors and the capacity and throughput of Intel Optane PMem, as well as the algorithmic innovation that enables optimal utilization of the hardware resources. In the following sections, we provide the details of how this combined hardware/software approach helped us win this competition by such huge margins.

Algorithmic Approach

We showcase the performance of Intel Xeon processors with Intel Optane PMem for ANN search algorithm, GraphANN. GraphANN is an extension of the graph-based Vamana algorithm that is highly optimized for Intel Optane PMem. It builds a directed graph to index the data points and follows a greedy search to navigate the graph and locate the nearest neighbors of a new query. Throughout the search, two main data structures are used: the graph and the feature vectors. In our solution, we store the graph in Intel Optane PMem and keep the feature vectors in DRAM, when possible. This combination yields impressive throughput and performance per dollar. Moreover, Intel Optane PMem comes in much higher capacities than traditional DRAM, thus providing the necessary scaling for larger datasets. Finally, persistence has the bonus of eliminating the need to load the index into memory, which is quite time-consuming for billion-scale datasets.

Intel® Optane™ Persistent Memory (PMem)

Intel Optane PMem is a storage class memory that can be used in SSDs and persistent memory applications. Historically, there has always been a gap between the memory and storage performance. Intel Optane memory technology is designed to bridge this gap (**Figure 4**). It allows memory cells to be addressed individually, in a dense, transistor-less, stackable 3D design. These features provide a unique combination of affordable capacity and support for data persistence. With innovative technology offering distinctive operating modes, it adapts to different needs across workloads. For example, Intel Optane technology has been used to accelerate storage of logs and caching tier of large-scale applications with storage bottlenecks.

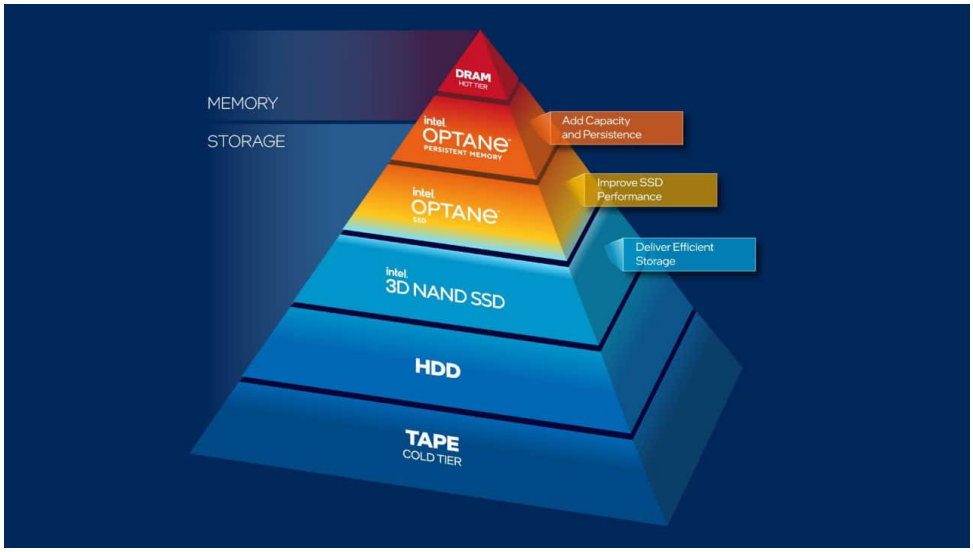


Figure 4. Memory and storage hierarchy and Intel® Optane™ technology's place in it.

Intel Optane PMem has some similarities with DRAM: It is packaged in DIMMs, it resides on the same bus/channels as DRAM, and it can act in the same way as DRAM storing volatile data. It differs from DRAM in the following ways:

- Intel Optane PMem comes in much higher capacities than traditional DRAM. Its modules come in 128GB, 256GB, and 512GB capacities, vastly larger than DRAM modules that typically range from 16GB to 64GB, though larger DRAM capacities exist.
- Intel Optane PMem can also operate in a persistent mode, storing data even without power applied to the module and comes with built-in hardware encryption to help keep data at rest secure. The TCO is greatly improved compared to DRAM on a cost-per-GB basis and the ability to increase the capacity to beyond DRAM's capabilities.

Intel Optane PMem has two operational modes for additional flexibility: Memory Mode and App Direct Mode. Memory Mode expands main memory capacity without persistence. It combines an Intel Optane PMem with a conventional DRAM that serves as a direct-mapped cache for PMem. In App Direct Mode, Intel Optane PMem appears as a persistent memory device that can be addressed separately from DRAM.

ANN with Intel® Optane™ Persistent Memory (PMem)

By studying the access patterns of the Vamana algorithm, we found that data reuse across queries is limited. This discourages the use of Intel Optane Memory Mode, as a cache would provide limited value. Therefore, we used App Direct Mode for this work.

To maximize performance, we organize the data by storing the graph in Intel Optane PMem while keeping the feature vectors in DRAM. The graph accesses follow a highly random and non-local pattern. Here, Intel Optane PMem has a 64-byte block size, which allows us to surgically access the graph elements while retaining performance. We limit the maximum out-degree of each node to 127, as this ensures that exactly four accesses are needed to retrieve the neighbors of any given node. (We use 4B per neighbor and 4B for the number of neighbors.) Storing the data contiguously allows pipelining of these four accesses.

Our optimized version of the Vamana algorithm is written in the Julia programming language. The optimizations can be divided into general (purely software-based) and specific to Intel Optane technology. The general optimizations relate to optimized graph and data representations, use of VNNI instructions for distance computation, static sizing of data vectors, and memory alignment, among others. One such important optimization is "prefetch hoisting," which decouples the distance-computation loop from the vector-fetching loop. In this approach, we use x86 intrinsics to prefetch as much data as possible from memory before beginning the distance computation step. This ensures that memory latency has minimal impact on the queries. The other important optimization results from partitioning the vectors between DRAM and PMem. This is because fetching the vectors for distance computation is the slowest step in the search, and we keep as many vectors as possible in DRAM, which reduces the PMem traffic and provides a significant performance improvement.

Our multithreaded architecture creates small batches of queries that are dynamically load-balanced across worker threads. Each thread processes one query at a time in its batch. Furthermore, all intermediate scratch-space data structures required to process a query are pre-allocated, with each thread owning its own private scratch space. This eliminates dynamic memory allocation during the query processing and minimizes the amount of synchronization among threads.

These optimizations allow us to deliver orders of more than10 to100X improvement in ANN search performance over the previous known best solution (the FAISS algorithm running on GPUs). **Figure 5** shows the improvement achieved by our optimized approach across five different datasets. These datasets encompass different encodings (Int8, UInt8, and Float32), as well as different distance metrics (Euclidean and Inner Product). We can see that across these datasets, GraphANN running on an Intel Xeon processor with Intel Optane memory drastically improves the baseline performance.

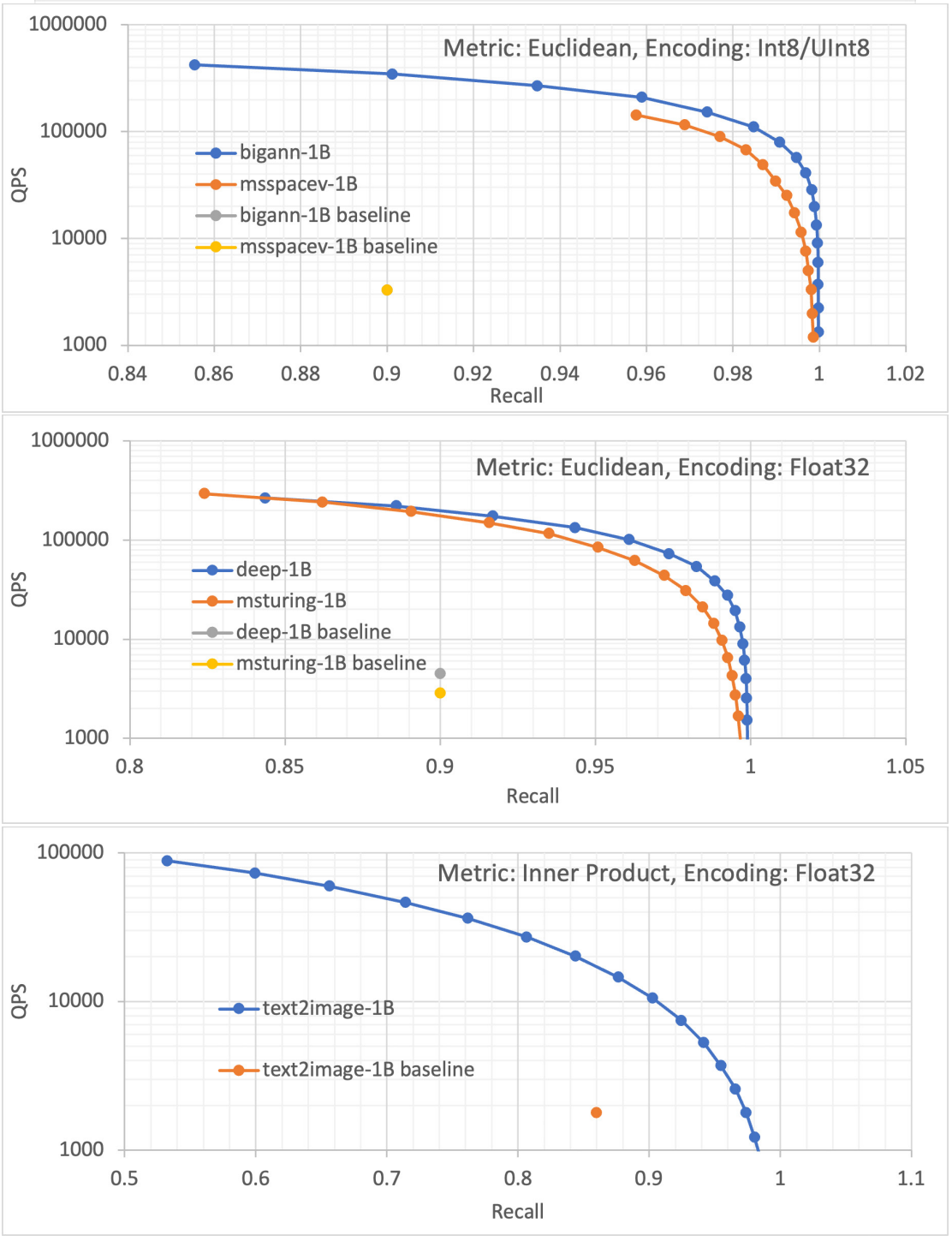
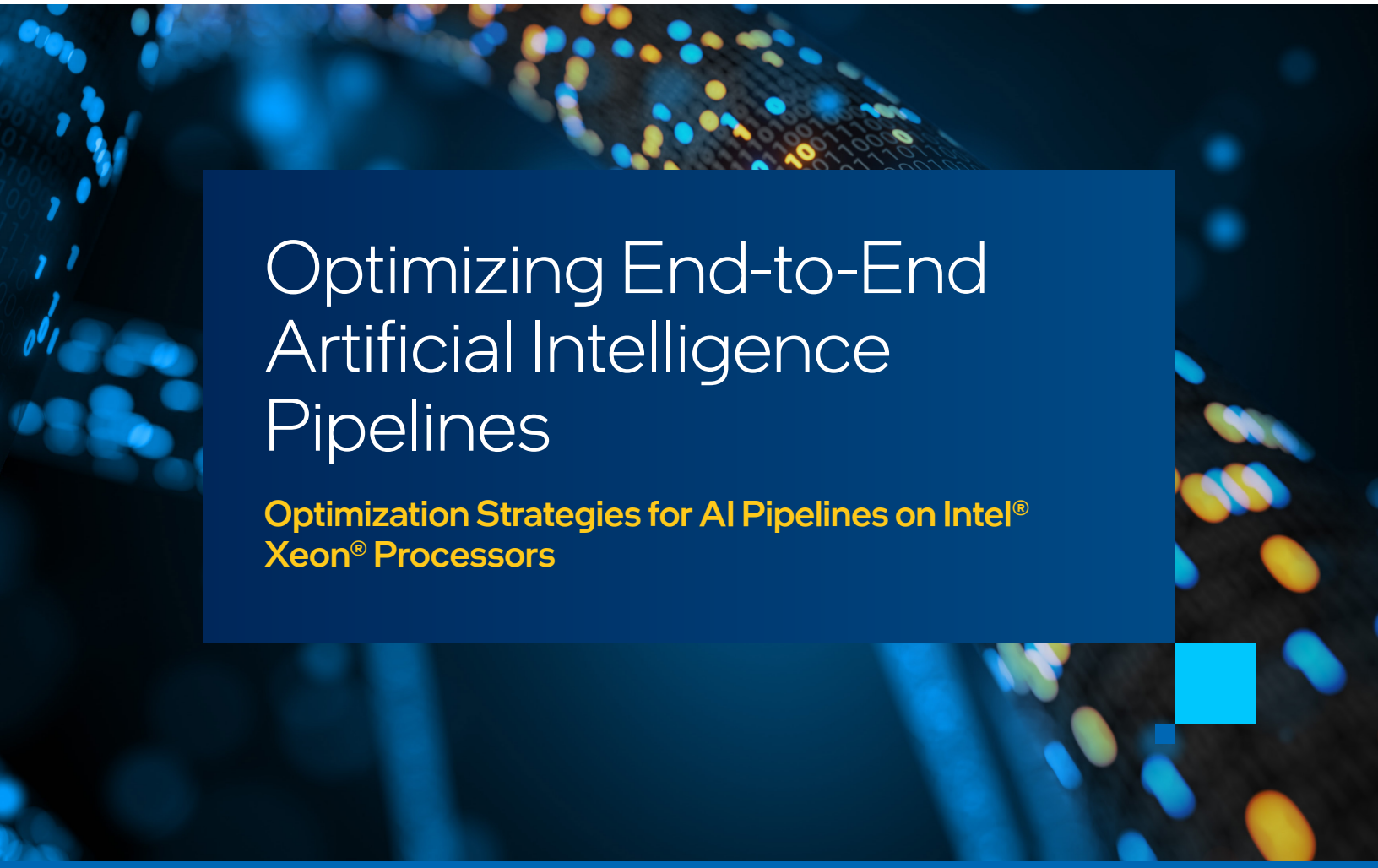


Figure 5. Queries per second (throughput) vs recall plots for the five different datasets, showing the magnitude of improvement of the Intel® Optane™ solution over the previously best software (FAISS) and hardware (GPUs).

Conclusion

In this article, we described the winning algorithm, design choices, and associated hardware setup in the Billion-Scale Approximate Nearest Neighbor Search Challenge in the NeurIPS 2021 (see the public leaderboard). We also showed that Intel Optane PMem can significantly improve the performance of similarity search algorithms across a range of design points, starting from only upgrading the hardware without any associated code changes to a full-blown custom rewrite of code.



Optimizing End-to-End Artificial Intelligence Pipelines

Optimization Strategies for AI Pipelines on Intel® Xeon® Processors

Meena Arunachalam, Principal Engineer; Vrushabh Sanghavi, Senior Deep Learning Software Engineer; Yi A Yao, AI Frameworks Engineer; Yi A Zhou, AI Frameworks Engineer; Lifeng A Wang, AI Frameworks Engineer; Zongru Wen, AI Frameworks Engineer; Niroop Ammbashankar, Senior Deep Learning Software Engineer; Ning W Wang, AI Frameworks Engineer; and Fahim Mohammad, Senior Deep Learning Software Engineer, Artificial Intelligence and Analytics Group, Intel Corporation

End-to-end (E2E) artificial intelligence (AI) pipelines are made up of one or more machine learning (ML)/deep learning (DL) models that solve a problem on a specific dataset and modality, accompanied by multiple preprocessing and postprocessing stages. We apply comprehensive optimization strategies on a variety of modern AI pipelines for ML, natural language processing (NLP), recommendation systems, video analytics, anomaly detection, and face recognition, along with DL/ML model training and inference, optimized data ingestion, feature engineering, media codecs, tokenization, etc. for higher E2E performance. The results across all our candidate pipelines, mostly inference-based, show that for optimal E2E throughput performance, all phases must be optimized. Large memory capacities, AI

acceleration (e.g., Intel® Deep Learning Boost [Intel® DL Boost]), and the ability to run general-purpose code make Intel® Xeon® processors well-suited for these pipelines.

Our optimizations fall broadly into application, framework, and library software; model hyperparameters; model optimization; system-level tuning; and workload partitioning. Tools such as Intel® Neural Compressor offer quantization, distillation, pruning, and other techniques that benefit from Intel DL Boost and other AI acceleration built into Intel Xeon processors. As a result, we see a 1.8x to 81.7x improvement across different E2E pipelines. In addition, we can host multiple parallel streams or instances of these pipelines with the high number of cores and memory capacity available on Intel Xeon processors compared to some memory-limited accelerators that can only host one or a very limited number of parallel streams. In many cases, workload consolidation to CPUs is possible, which also has TCO and power advantages.

E2E AI Applications

We showcase many E2E AI use cases and workloads, each comprising unique pre- and post-processing steps and implemented using a variety of different ML/DL approaches on video, image, tabular, text, and other data types (**Table 1**).

Workload	Application Name	Model	Pre-/Post-processing Stages	Dataset
ML	Census	Ridge Regression	Load data to data frame, drop columns, remove rows, arithmetic ops, type conversion, train/test split	IPUMs Census Data
	PLAsTiCC	Gradient Boosting Tree	Load data, drop columns, groupby aggregation, arithmetic ops, type conversion, train/test split	LSST Simulated Data
	Predictive Analytics in Industrial IoT	Random Forest Classifier	Load data to data frame, drop inessential columns, train/test split	Bosch Production Line
NLP	Document Level Sentiment Analysis	BERT-Large	Load data, initialize tokenizer, data encoding, load model	IMDb
				SST-2
Recommendation System	E2E Deep Interest Evolution Network	DIEN	Data ingestion, label encoding, get history sequence, native sampling, data split, load model	Amazon Books
Video Analytics	Video Streamer	SSD Resnet-34	Video decode, image normalization and resizing, bounding box and labeling, data uploading	Mall video
Anomaly Detection	Anomaly Detection	ResNet50v1.5	Load data, image resizing, image transformations, evaluating feature reconstruction error	MVTec AD
Face Recognition	Face Detection and Recognition	SSD MobileNet, Resnet50v1.5	Load video, frame splitting, resizing, output generation	Soccer celebration

Table 1. E2E AI applications.

E2E AI applications typically involve two broad categories of operations: pre-/post-processing and AI. In **Figure 1**, we see the breakdown range from 4% to 98% pre-/post-processing to 2% to 96% AI as a fraction of the total E2E run-time.

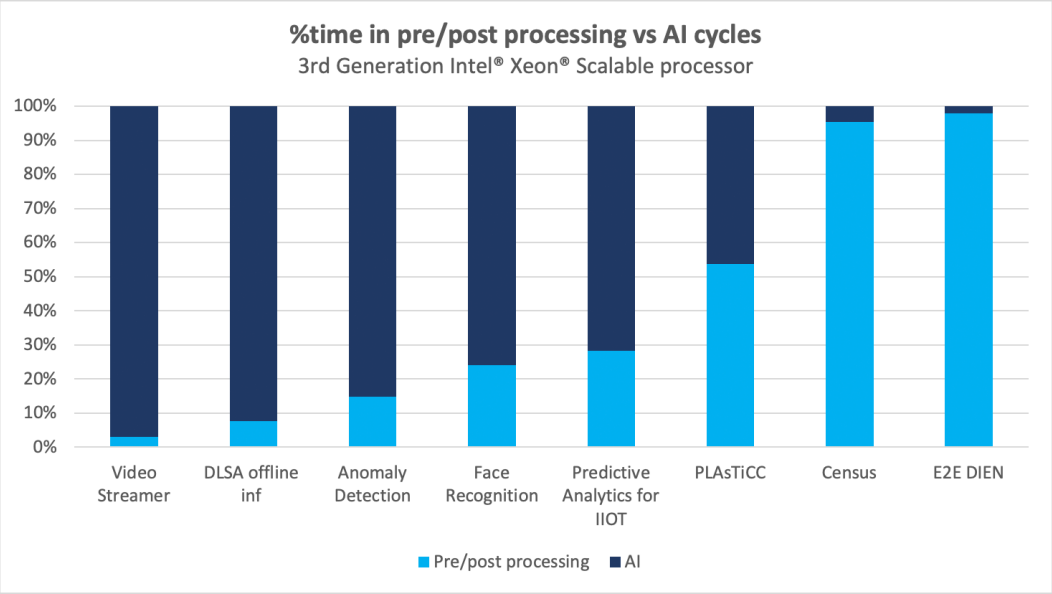


Figure 1. Percent time in pre-/post-processing vs AI.

Census

The Census workload trains a ridge-regression model using the U.S. Census data from the years 1970 to 2010, and predicts the correlation between personal education level and income (**Figure 2**). Prior to ML, it ingests the data, performs data frame operations to prepare the input for model training, and creates a feature set and its subsequent output set[1].

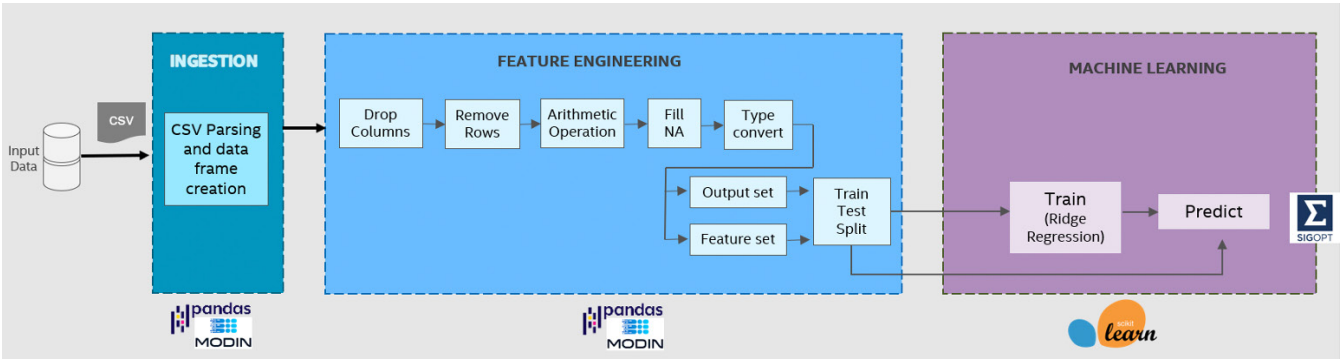


Figure 2. Census application pipeline.

PLAsTiCC

PLAsTiCC is an open data challenge that uses simulated astronomical time-series data to classify objects in the night sky that vary in brightness (**Figure 3**). The pipeline loads the data; manipulates, transforms, and processes the data frames; and uses the histogram tree method from the XGBoost library to train a classifier and perform model inference.

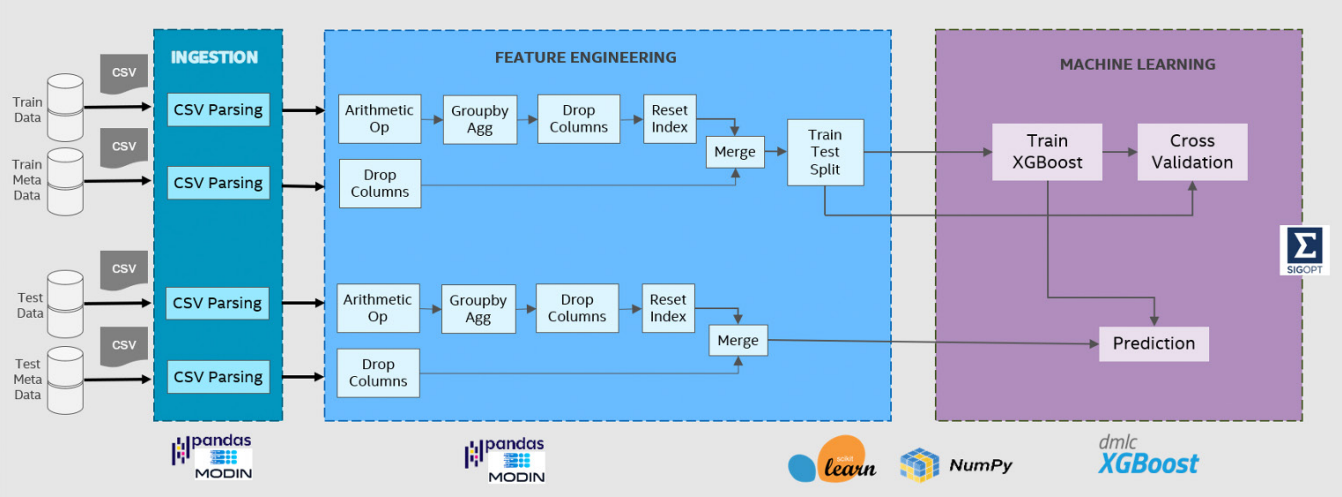


Figure 3. PLAsTiCC application pipeline.

Predictive Analytics in Industrial IoT

This is an E2E unsupervised learning use case in industrial IoT that predicts internal failures during manufacturing, thereby helping maintain the quality and performance of the production line (**Figure 4**). The workflow consists of reading measurements from a CSV file and cleaning them to include only the necessary features. The highly optimized Intel® Distribution of Modin[2] is used for this step. The random forest classifier from Intel® Extension for Scikit-learn[3] is used to generate the model.

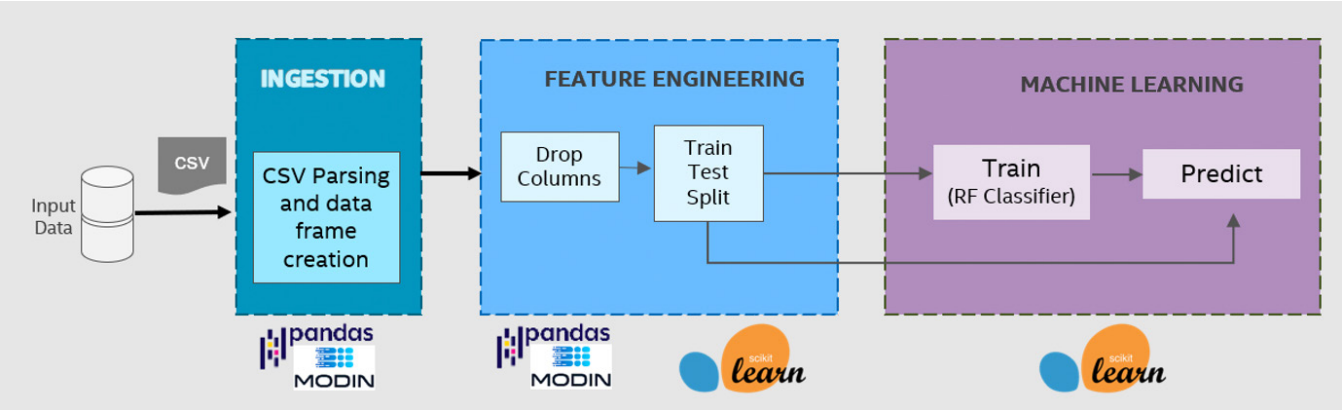


Figure 4. Pipeline for predictive analytics in industrial IoT.

Document Level Sentiment Analysis (DLSA)

The DLSA workflow shown in **Figure 5** is a reference NLP pipeline built using the Hugging Face transformer API to perform document-level sentiment analysis. It uses language models such as BERT-LARGE (uncased), pretrained on a large English text corpus.

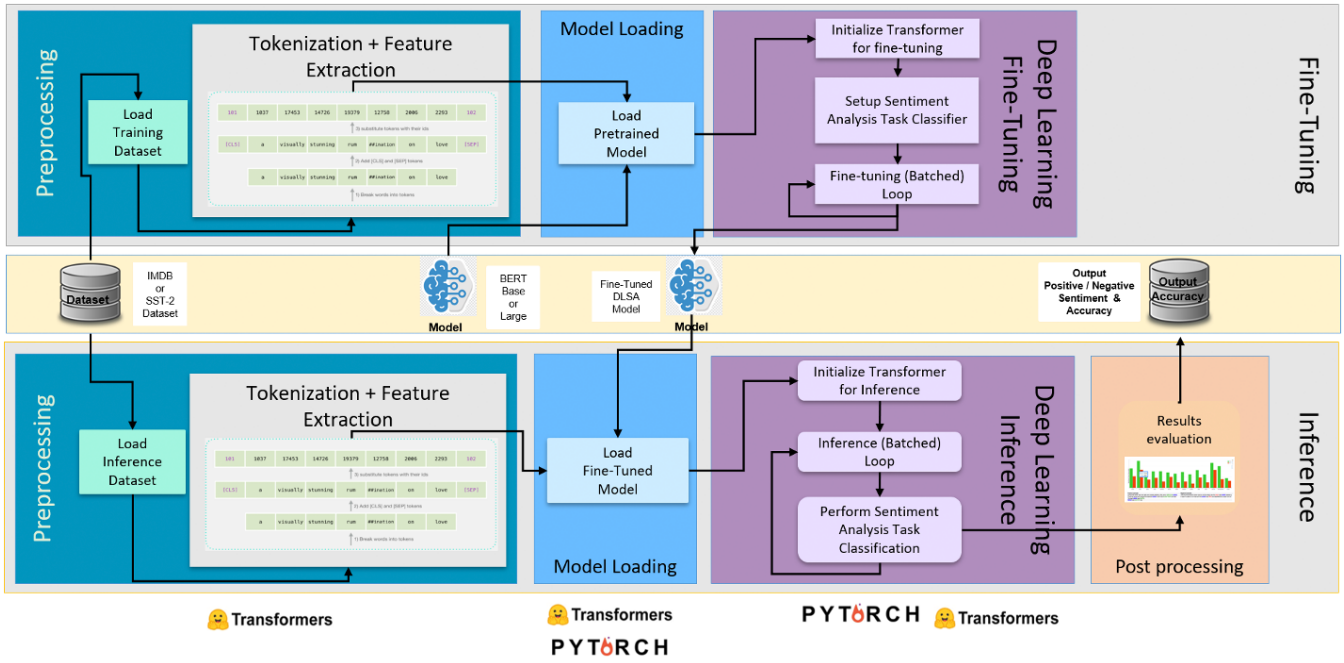


Figure 5. Document level sentiment analysis pipeline.

E2E Deep Interest Evolution Network (DIEN) Recommendation System

The DIEN workflow shown in **Figure 6** is a recommendation inference pipeline that estimates the probability of user clicks at scale.

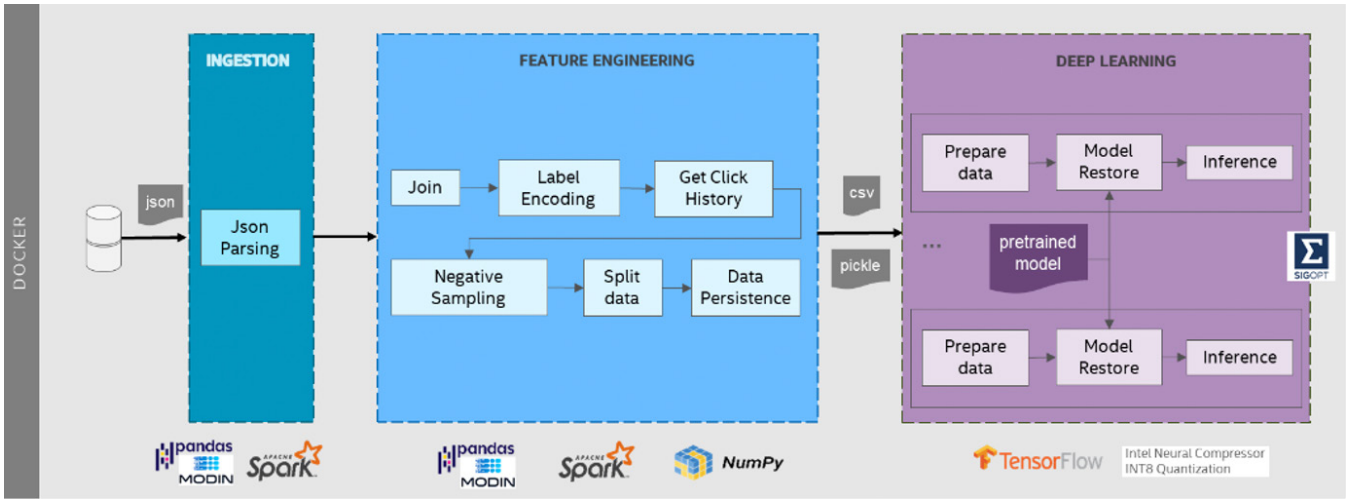


Figure 6. E2E DIEN recommendation system pipeline.

Video Streamer

The video streamer pipeline (**Figure 7**) is designed to mimic real-time video analytics. Real-time data is provided to an inference endpoint that executes single-shot object detection. The metadata created during inference is then uploaded to a database for curation. The pipeline is built upon GStreamer, TensorFlow, and OpenCV. The input video is decoded by GStreamer into images on a frame-by-frame basis. Then, the GStreamer buffer is converted into a NumPy array. TensorFlow does image normalization and resizing, followed by object detection with a pretrained SSD-ResNet34 model. Finally, the results of bounding-box coordinates and class labels are uploaded to a database.

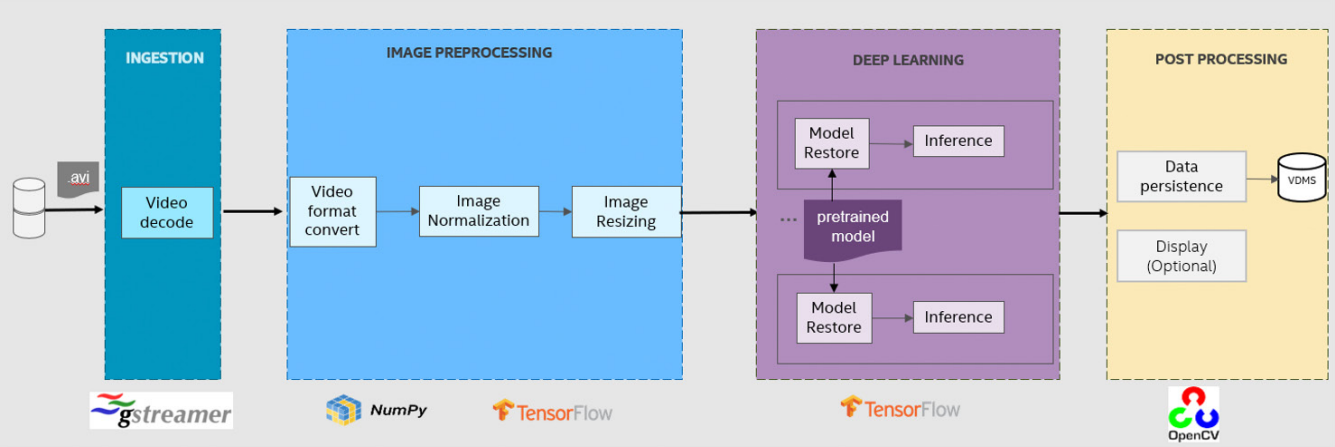


Figure 7. Video streamer application pipeline.

Anomaly Detection

The objective of anomaly detection is to analyze images of parts being manufactured on an industrial production line, using deep neural network and probabilistic modeling to identify rare defects (**Figure 8**). As an out-of-distribution solution, a model of normality is learned over feature maps of the final few layers from normal data in an unsupervised manner. Deviations from the models are flagged as anomalies. Prior to learning distribution, the dimension of the feature space is reduced by using PCA to prevent matrix singularities and rank deficiencies from arising while estimating the parameters of the distribution.

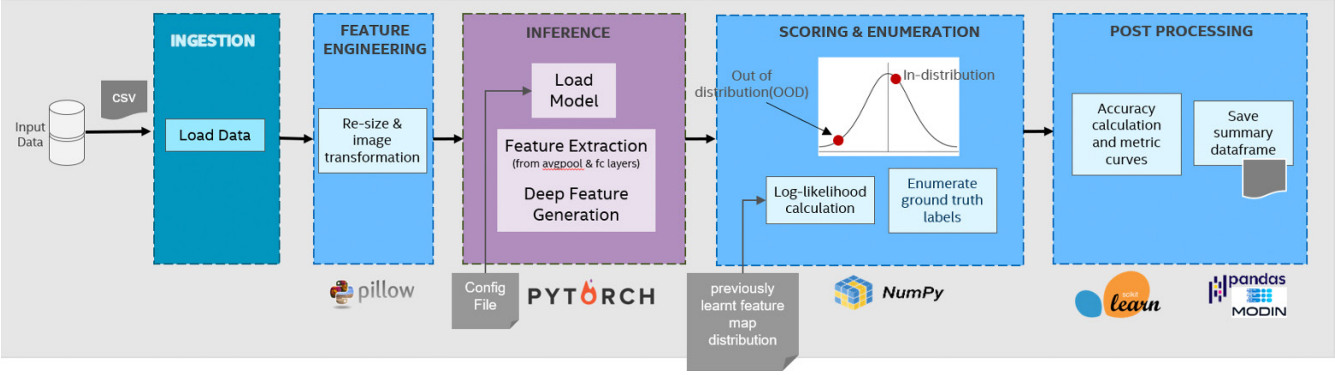


Figure 8. Anomaly detection pipeline.

Face Recognition

This E2E pipeline performs real-time face recognition by cascading two, out-of-the-box, pretrained models: SSD MobileNet and ResNet50v1.5 (**Figure 9**). The input from the camera, as compressed or uncompressed video, undergoes frame splitting and resizing. Each frame is then fed to the detection model (SSD MobileNet), which performs object detection. The NMS bounding boxes are then fed to the recognition model (ResNet50v1.5) to recognize the faces. The output frames with the facial recognitions can either be displayed or saved in databases.

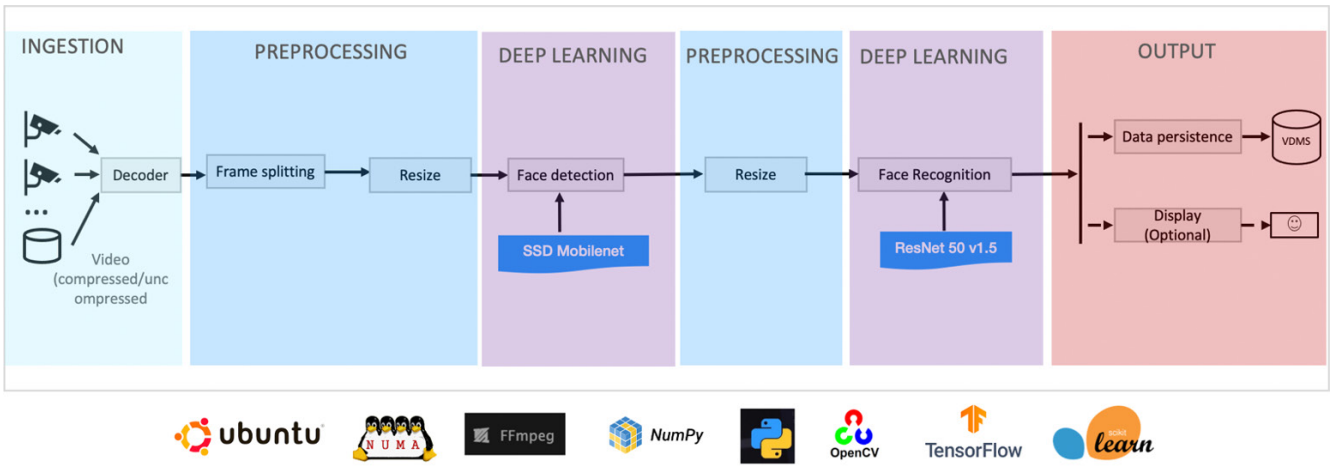


Figure 9. Face recognition pipeline.

How To Do "Efficient-AI": E2E Optimization Strategies

E2E performance-efficient AI requires a coherent optimization strategy consisting of AI software acceleration, system-level tuning, hyperparameter and runtime parameter optimizations, and workflow scaling. All phases (data ingestion, data preprocessing, feature engineering, and model building) need to be holistically addressed to improve user productivity as well as workload performance efficiency (**Figure 10**).

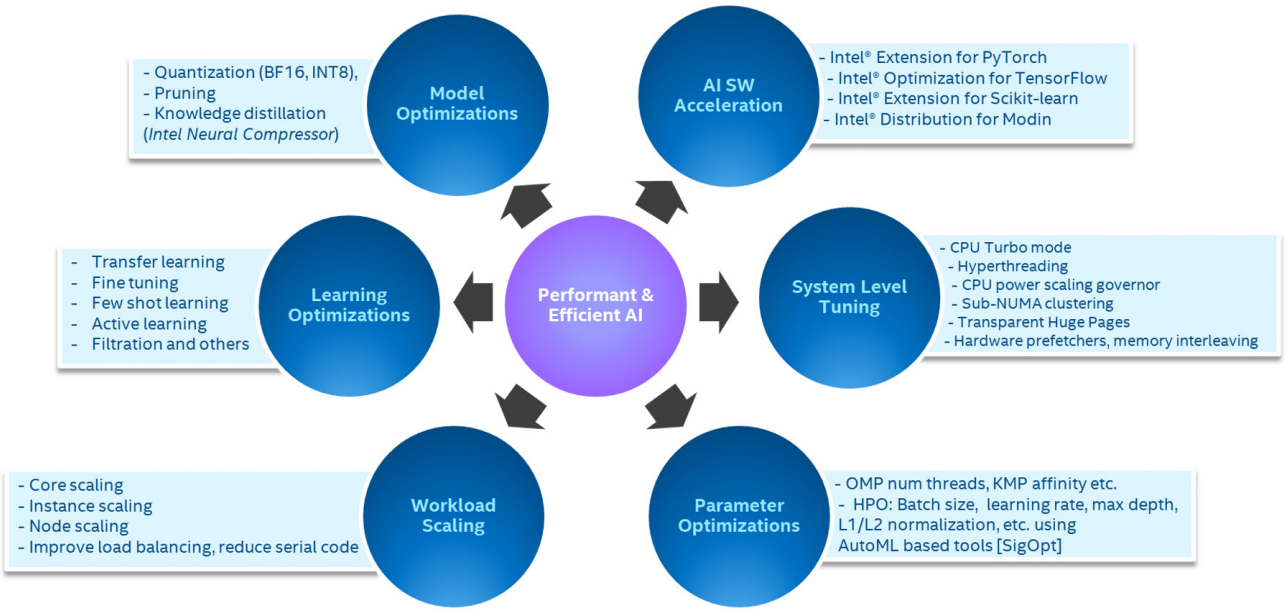


Figure 10. Efficient AI.

AI Software Acceleration

Intel Distribution of Modin is a multithreaded, parallel, and performant data frame library compatible with the pandas API. It performs lightweight, robust data frame and CSV operations, and it scales efficiently with the number of cores, unlike pandas, providing a significant speedup just by changing a couple of lines of code. Data frame operations across different phases speed up from 1.12x to 30x.

Intel Extension for Scikit-learn accelerates common estimators, transformers, and clustering algorithms in classical ML. Ridge regression training and inference in the Census workload is a DGEMM-based memory-bound algorithm that takes advantage of Intel Extension for Scikit-learn's vectorization, cache-friendly blocking, and multithreading to efficiently use multiple CPU cores.

Intel-optimized XGBoost and CatBoost libraries provide efficient parallel tree boosting. The XGBoost kernels are optimized for cache efficiency, remote memory latency, and memory access patterns on Intel processors.

Intel® Extension for PyTorch[4] improves PyTorch performance on Intel processors. With Intel Extension for PyTorch, the Anomaly Detection and DLSA pipelines take advantage of Intel DL Boost. Intel® Optimization for TensorFlow[5] is powered by Intel® oneAPI Deep Neural Network Library (oneDNN), which includes convolution, normalization, activation, inner product, and other primitives vectorized using Intel® AVX-512 instructions. The DIEN, face recognition, and video streamer applications use Intel Optimization for TensorFlow to enable scalable performance on Intel processors through vectorization and optimized graph operations (e.g., ops fusion, batch normalization).

Model Optimizations

Quantization facilitates conversion of high-precision data (32-bit floating point, FP32) to lower precision (8-bit integer, INT8), which enables critical operations such as convolution and matrix multiplication to be performed significantly faster with little to no loss in accuracy. Intel Neural Compressor automatically optimizes low-precision recipes for DL models and calibrates them to achieve optimal performance and memory usage with expected accuracy criteria. DLSA and video streamer applications achieved up to 4x speedup from INT8 quantization alone (**Table 2**).

Parameter Optimizations

The SigOpt model development platform makes it easy to track runs, visualize training, and scale hyperparameter optimization for any pipeline, while also tuning for objectives like maximum throughput at threshold accuracy and/or latency levels. With SigOpt's multi-objective optimization, we can easily obtain insights on the best configurations of the AI pipeline, showing the optimal performance summary and analysis. In the case of PLAsTiCC, "accuracy" and "timing" metrics were optimized, while the model hyperparameters (like the number of parallel threads for XGBoost, number of trees, learning rate, max depth, L1/L2 normalization, etc.) were computed in order to achieve the objective[6]. In DLSA, the number of inference instances and batch size are tuned to achieve high E2E throughput.

Run-time options in TensorFlow also make a big performance impact. It is recommended to control the parallelism within an operation like matrix multiplication or reduction so as to schedule the tasks within a threadpool by setting `intra_op_parallelism_threads` equal to the number of available physical cores and, in contrast, running operations that are independent in the TensorFlow graph concurrently by setting `inter_op_parallelism_threads` equal to the number of sockets. Data layout, OpenMP, and NUMA controls are also available to tune the performance even further[5].

Workload Scaling

Multi-instance execution allows parallel streams of the application to be executed on a single Intel® Xeon® Scalable server. The advantage is demonstrated during anomaly detection, where several cameras can be deployed to detect defects at different stages of the manufacturing pipeline; 10 such streams processing over the standard 30 FPS on a ResNet50 model can be serviced by a single 3rd Gen Intel Xeon Scalable processor. Similarly, E2E DIEN runs with one core/instance with 40 inference instances per socket, while DLSA and DL pipelines run four cores/instance to eight cores/instance with 10 inference streams to five inference streams per socket. This is a unique advantage of CPUs with their large memory capacity.

System-level tuning is available in the BIOS to improve efficiency. Tuning knobs controlling hyperthreading, CPU power scaling governors, NUMA optimizations, hardware prefetchers, and more can be explored to obtain best performance.

	Intel Distribution for Modin	Intel Extension for Scikit-learn	XGBoost	Intel Extension for PyTorch	Intel-optimized TensorFlow	INT8 quantization
Census	6x	59x	-	-	-	-
PLAsTiCC	30x	8x	1x	-	-	-
Predictive Analytics for Industrial IoT	4.8x	113x	-	-	-	-
Document Level Sentiment Analysis	-	-	-	4.15x	-	3.90x
E2E Deep Interest Evolution Network	23.2x	-	-	-	9.82x	-
Video Streamer	-	-	-	-	1.36x	3.64x
Anomaly Detection	1.12x	3.4x	-	1.8x	-	-
Face Recognition	-	-	-	-	1.7x	-

Table 2. Performance improvement from software optimizations and quantization for E2E AI applications.

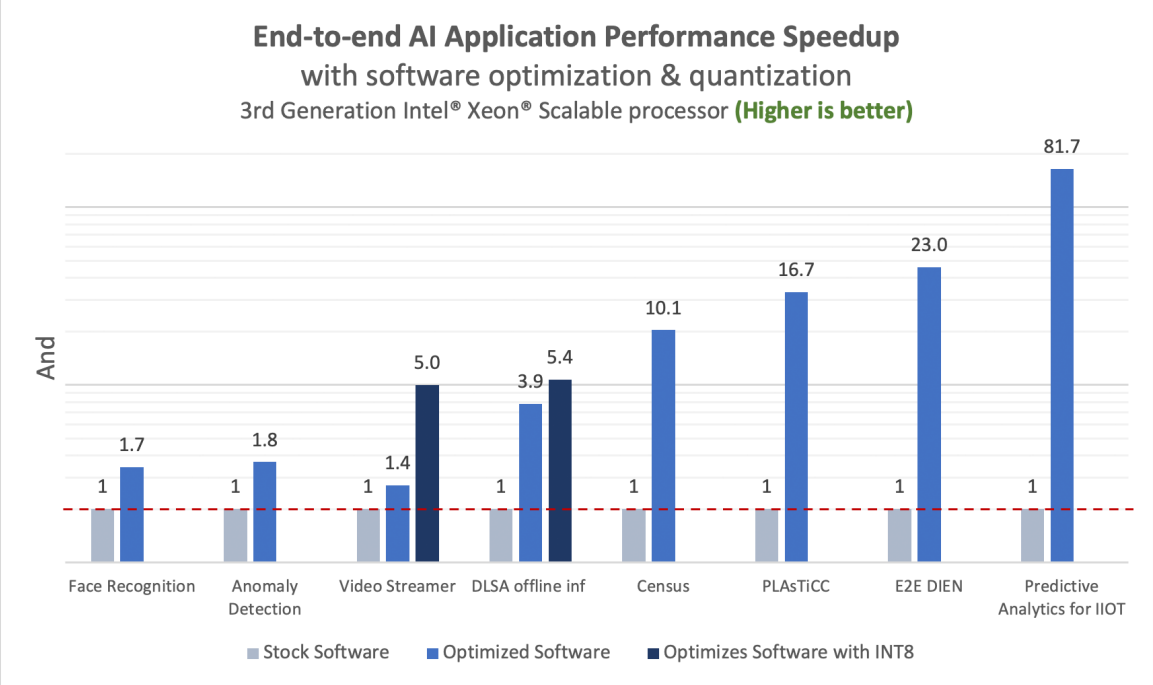


Figure 11. E2E AI application performance speedup.

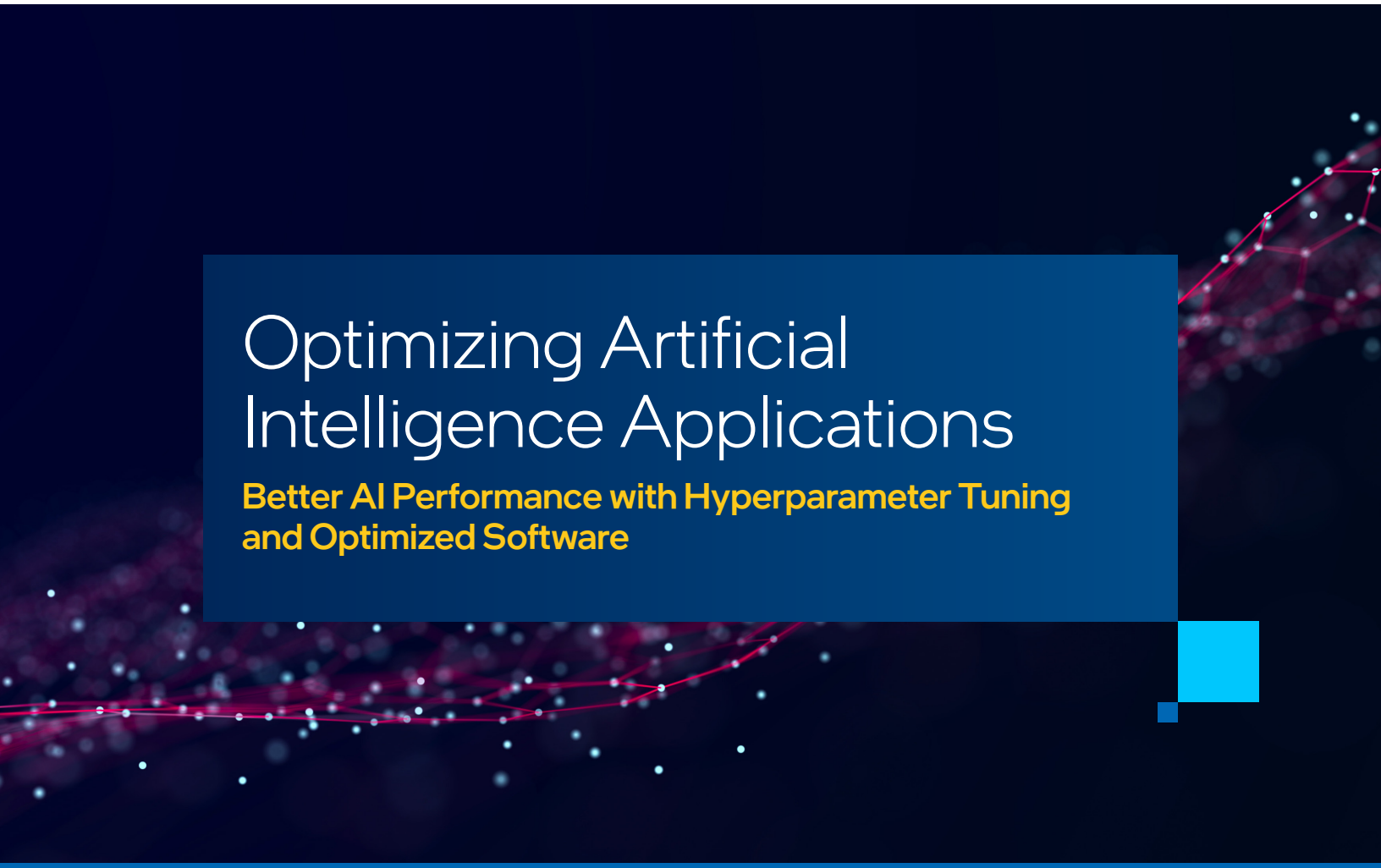
Configuration: Performance measured on a single-node, dual-socket 3rd Generation Intel Xeon Scalable 8380 processor (except DIEN and DLSA), 40 cores per socket. DIEN and DLSA were measured on 3rd Generation Intel Xeon Scalable 6348 processors, turbo mode enabled, hyperthreading disabled, BIOS: SE5C620.86B.01.01.0003.2104260124, kernel: 5.13.0-28-generic, OS: Ubuntu 21.10, 512GB memory (16 slots/32GB DIMMs/3200MHz), Intel 480GB SSD OS Drive.

Anomaly Detection: Python 3.7.11, torch 1.11.0, torchvision 0.11.3, PyTorch 1.10, numpy 1.22.1, pandas 1.3.5, scikit-learn-intelex 2021.4.0; Face Recognition: Python 3.7.9, TensorFlow 2.8.0, numpy 1.22.2, opencv-python 4.5.3.56, ffmpy 0.3.0; Video Streamer: Python 3.8.12, TensorFlow 2.8.0, opencv-python 4.5.2.54, pillow 8.3.1, gstreamer1.0, vdms 0.0.16; DLSA offline Inf: Python 3.7.11, PyTorch 1.10, HuggingFace Transformer:4.6.1; E2E DIEN: Python 3.8.10, Modin 0.12.0, TensorFlow 2.8.0, numpy 1.22.2; Census: Python 3.9.7, Modin 0.12.0, scikit-learn-intelex 2021.4.0; PLAsTiCC: Python 3.9.7, Modin 0.12.0, scikit-learn-intelex 2021.4.0, XGBoost 1.5.0; Predictive Analytics for Industrial IoT: Python 3.9.7, Modin 0.12.0, scikit-learn-intelex 2021.4.0.

In conclusion, as a result of cumulative optimization strategies across software, system, hardware, model-building, and hyperparameters, we achieve 1.8x to 81.7x speedup in E2E performance on Intel Xeon processors.

References

1. Census workload oneAPI sample code, Intel® oneAPI AI Analytics Toolkit: https://github.com/oneapi-src/oneAPI-samples/tree/master/AI-and-Analytics/End-to-end-Workloads/Census
2. Intel Distribution of Modin: https://www.intel.com/content/www/us/en/developer/tools/oneapi/distribution-of-modin.html
3. Getting Started with Intel Extension for Scikit-learn: https://www.intel.com/content/www/us/en/developer/articles/guide/intel-extension-for-scikit-learn-getting-started.html
4. Intel Extension for PyTorch: https://www.intel.com/content/www/us/en/developer/articles/guide/getting-started-with-intel-optimization-of-pytorch.html
5. Maximize TensorFlow Performance on CPU: https://www.intel.com/content/www/us/en/developer/articles/technical/maximize-tensorflow-performance-on-cpu-considerations-and-recommendations-for-inference.html
6. Optimizing Artificial Intelligence Applications: Better AI Performance with Hyperparameter Tuning and Optimized Software: https://medium.com/intel-analytics-software/optimizing-artificial-intelligence-applications-1bc22b5d707b



Optimizing Artificial Intelligence Applications

Better AI Performance with Hyperparameter Tuning and Optimized Software

Vrushabh Sanghavi, Senior Deep Learning Software Engineer, Intel Corporation

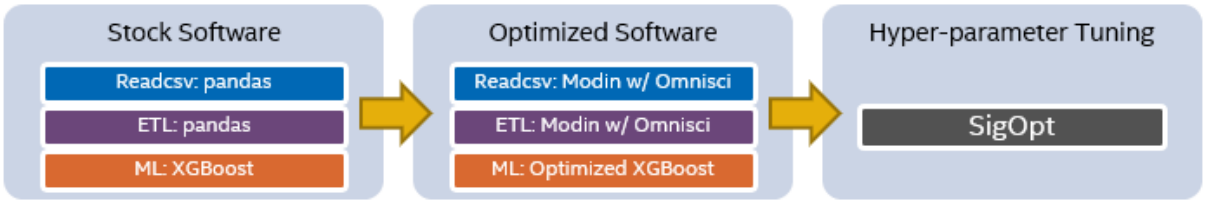
Data scientists are always looking for ways to boost their AI application performance. Using optimized machine learning software instead of stock packages is an easy way to do this. Tuning model hyperparameters using an AutoML-based platform like SigOpt is another. I will demonstrate the performance possibilities using the PLAsTiCC Classification Challenge from Kaggle.

PLAsTiCC is an open data challenge to classify objects in the sky that vary in brightness. It uses simulated astronomical time-series data in preparation for observations that will come from the Large Synoptic Survey Telescope being set up in northern Chile. The challenge is to determine the probability that each object belongs to one of 14 classes of astronomical filters, scaling from a small training set (1.4 million rows) to a very large test set (189 million rows).

The code can be divided into three distinct phases:

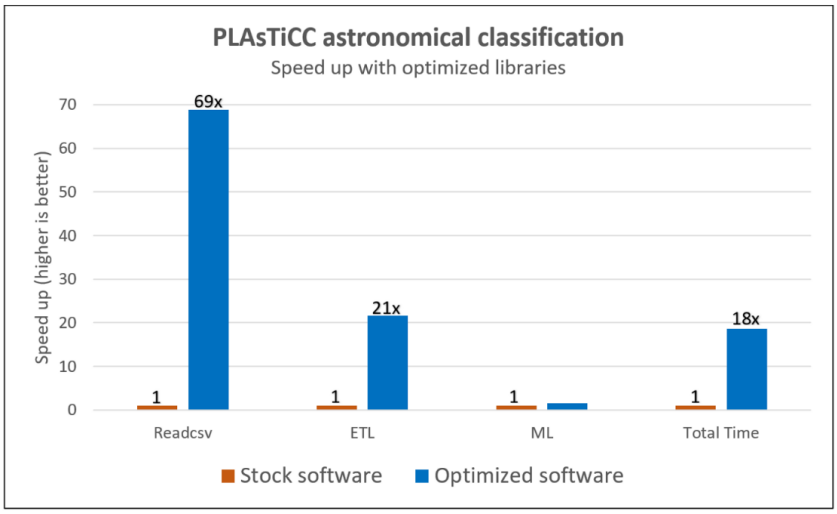
1. **Readcsv:** Loading the CSV-format training and testing data and their corresponding metadata into pandas dataframes.
2. **ETL:** Manipulating, transforming, and processing the dataframes for input to the training algorithm.
3. **ML:** Using the histogram tree method from the XGBoost library to train the classification model. The model is cross-validated, and the trained model is used to classify objects in the massive test set.

The chart below shows stock and optimized software that is used for each of these phases plus SigOpt for hyperparameter tuning:



The Intel® Distribution for Modin* is used to improve Readcsv and ETL performance. This parallel and distributed dataframe library uses the pandas API. It allows you to significantly improve the performance of dataframe operations just by changing a single line of code. To improve PLAsTiCC ML performance, the XGBoost Optimized for Intel® Architecture package is upstreamed to the main branch. This can be obtained by simply installing the latest version of XGBoost. (See Distributed XGBoost with Modin on Ray.)

The bar chart below shows the speed-ups obtained using the optimized software stack (shown in blue) over the stock software (shown in orange) in each PLAsTiCC phase. A massive 18x end-to-end speedup is achieved by using the optimized software. Intel Distribution for Modin performs lightweight, robust dataframe and Readcsv operations and scales efficiently with the number of cores, unlike pandas. The XGBoost kernels are optimized for cache efficiency, remote memory latency, memory access patterns on Intel® architectures, and optimally uses its higher processor frequencies, cache size, and cache bandwidth.



We can improve the end-to-end workload performance even further by tuning the hyperparameters in the machine learning model. SigOpt is a model-development platform that provides an easy way to do this. It tracks training experiments, provides tools to visualize the training, and scales hyperparameter optimization for any type of model.

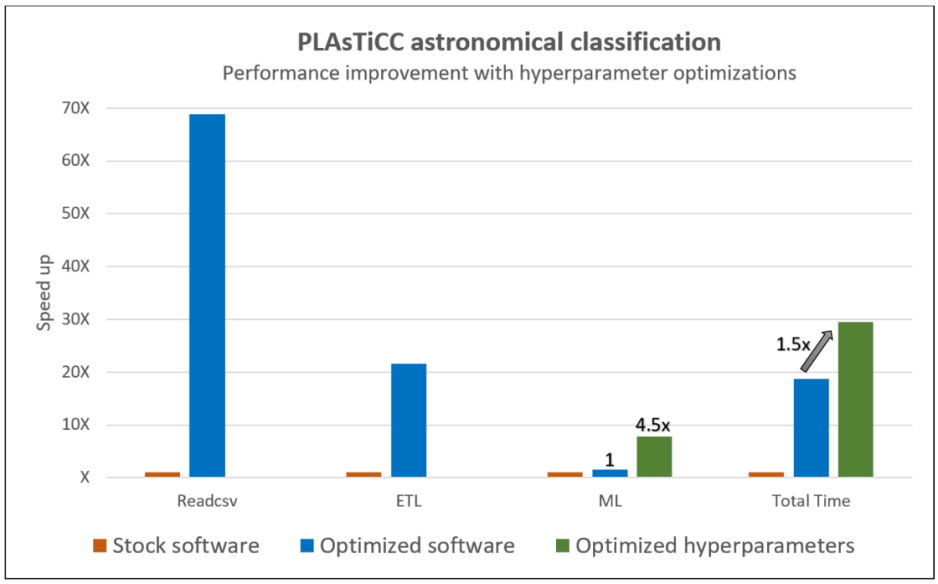
SigOpt finds the best parameter values for the model and provides the global optimum for the defined metric within the optimization loop. In the case of PLAsTiCC, **accuracy** and **timing** are the metrics to be optimized, while the model hyperparameters (like the number of parallel threads for XGBoost, number of trees, learning rate, max depth, L1/L2 normalization, etc.) are the parameters that need to be computed in order to achieve the objective. A minimum number of observations need to be run to find the global maximum or minimum of the objective function, and convergence mostly occurs when the number of experiments is set to 10–20 times the number of parameters in the experiment.

This following table shows the default model parameters and the tuned parameters as computed by the SigOpt autoML experiments:

Parameters	Default	Tuned
nthread	56	48
subsample	0.92	0.54
colsample_bytree	0.47	0.75
nround	43	18
max_depth	1	2
max_bin	38	60
min_child_weight	9.80	0.80
learning_rate (eta)	0.93	0.76
gamma	0.90	0.65
(reg_) alpha	0.38	0.39
(reg_) lambda	0.75	0.54
scale_pos_weight	7	6

It's easy to see that manually tuning and running through all these permutations would be almost impossible, whereas SigOpt can do it in a few hours. The log loss and validation loss for the model does not increase, which means that these improvements were achieved without compromising or affecting model accuracy.

The previous chart is replotted below to show the additional 5.4x ML performance improvement due to SigOpt hyperparameter tuning, which gives a 1.5x overall improvement.



These steps performed over a typical end-to-end pipeline show the significant performance improvement that can be achieved on an AI workload using a variety of Intel-optimized software packages, libraries, and optimization tools. (See Performance Optimizations for End-to-End AI Pipelines.)

Hardware and Software Configurations

Hardware: 2 Intel® Xeon® Platinum 8280L processors (28 cores), OS: Ubuntu 20.04.1 LTS Mitigated, 384 GB RAM (384 GB RAM: 12 x 32 GB 2933 MHz), kernel: 5.4.0–65-generic, microcode: 0x4003003, CPU governor: performance. **Software:** scikit-learn 0.24.1, pandas 1.2.2, XGBoost 1.3.3, Python 3.9.7, scikit-learn-intelex 2021.2, modin 0.8.3, omniscidbe v5.4.1.

Five Outstanding Additions Found in SYCL 2020

The SYCL Programming Language Is Evolving

James Brodman, Principal Engineer, and John Pennycook, Software Enabling and Optimization Architect, Intel Corporation

SYCL 2020 is an exciting update for C++ programmers looking to take advantage of accelerators. We have both had the pleasure of contributing to the SYCL specification, a book on SYCL, and the DPC++ open source project to implement SYCL into LLVM. We would like to share our pick for our favorite new features added to SYCL in the SYCL 2020 specification. We offer these as our opinions as Intel engineers, not on behalf of Khronos.

SYCL

SYCL is a Khronos standard that brings support for heterogeneous programming to C++. The SYCL 2020 specification was finalized in late 2020, and compiler support has been growing ever since. (See the Khronos website for information on implementations.)

The case for SYCL is articulated in many places, including *Considering a Heterogeneous Future for C++* and numerous other resources enumerated on sycl.tech. Put simply, SYCL addresses a key challenge: *How do we enable heterogeneous programming in C++, with portability across vendors and architectures?*

Thanks to strong community input, SYCL 2020 has exciting new features to serve the goal of being strongly multivendor and multiarchitecture. In this article, we discuss the functionality of and motivation for these new features.

The Outstanding Five

A key goal of SYCL 2020 is to align SYCL with ISO C++, which has two benefits. First, it ensures that SYCL feels natural to C++ programmers. Second, it allows SYCL to act as a proving ground for multivendor, multiarchitecture solutions to heterogeneous programming that may inform other C++ libraries (and perhaps ISO C++ itself).

Many of the syntactic changes in SYCL 2020 are a result of updating the base language from C++11 to C++17, enabling developers to take advantage of features such as class template argument deduction (CTAD) and deduction guides. But there are many new features, too! In this article, we choose to highlight five features new in SYCL 2020 and talk a little about why they matter.

1. **Backends** open the door for SYCL implementations built on other languages/frameworks besides OpenCL, enabling SYCL to target a wider variety of hardware.
2. **Unified shared memory (USM)** is a pointer-based access model, which serves as an alternative to the buffer/accessor model from SYCL 1.2.1.
3. **Reductions** are a common programming pattern, which SYCL 2020 accelerates via a "built-in" library.
4. The **group library** provides abstractions for cooperative work items, yielding additional application performance and programmer productivity through alignment with underlying hardware capabilities (regardless of vendor).
5. **Atomic references** aligned with the C++20 `std::atomic_ref` extend the C++ memory model to heterogeneous devices.

Together, these additions help to establish the SYCL ecosystem as one that is open, multivendor, and multiarchitecture, enabling C++ programmers to fully utilize the potential of heterogeneous computing now and into the future.

1. Backends

With the introduction of backends, SYCL 2020 opens the door to implementations built on other languages/frameworks besides OpenCL. Consequently, the namespace has shortened to just `sycl::`, rather than `cl::sycl::`, and the SYCL header file has moved from `<CL/sycl.hpp>` to `<sycl/sycl.hpp>`.

The changes here are not simply cosmetic and have profound implications for SYCL. Although implementations are still free to build atop OpenCL (and many do), support for generic backends has transformed SYCL into a programming model that can target a larger variety of heterogeneous APIs and hardware. SYCL is now able to act as the "glue" between C++ applications and vendor-specific libraries, allowing developers to target a range of platforms more easily — and without having to change their code.

SYCL 2020 delivers on being truly open, cross-architecture, and cross-vendor.

The open source DPC++ compiler project, which is implementing SYCL 2020 in LLVM (clang), takes advantage of this new flexibility to support NVIDIA, AMD, and Intel® GPUs. SYCL 2020 delivers on being truly open, cross-architecture, and cross-vendor (**Figure 1**).

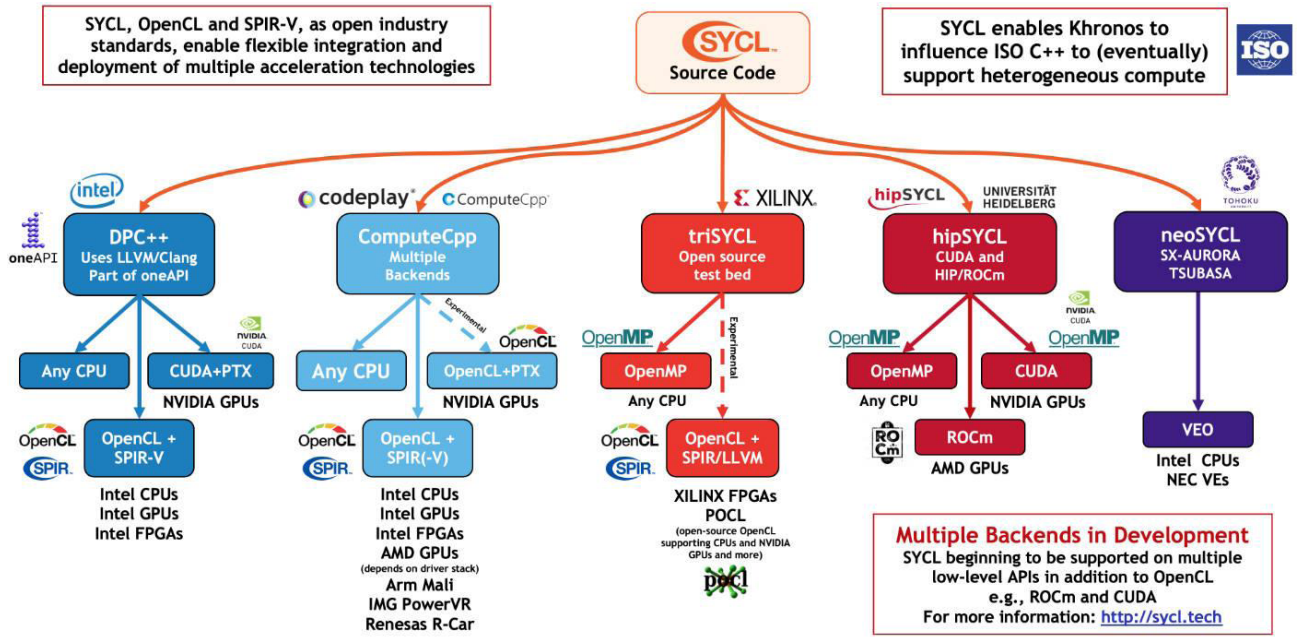


Figure 1. SYCL implementations targeting multiple backends, from https://www.khronos.org/sycl/.

2. Unified Shared Memory

Some devices can support a unified view of memory with the host (CPU). SYCL 2020 calls this **unified shared memory** (USM), and it enables a pointer-based access model that serves as an alternative to the buffer/accessor model from SYCL 1.2.1.

Programming with USM has two key advantages. First, USM supplies a single, unified address space across host and device; pointers to USM allocations are consistent across devices and can be directly passed to kernels as arguments. This greatly simplifies the porting of existing pointer-based C++ and

CUDA code to SYCL. Second, USM enables *shared* allocations that migrate automatically across devices, improving programmer productivity and providing compatibility with C++ containers (e.g., `std::vector`) and C++ algorithms (via oneDPL, **Figure 2**).

```
sycl::usm_allocator<int, sycl::usm::alloc::shared> alloc(q.get_context(),
                                                         q.get_device());
std::vector<int, decltype(alloc)> vec(n, alloc);

auto policy = oneapi::dpl::execution::make_device_policy(q);
std::fill(policy, vec.begin(), vec.end(), 0);
```

Figure 2. Using USM with C++ containers and algorithms, from our book examples.

The three different types of USM allocations provide programmers with as much or as little control over data movement as desired. Device allocations give programmers complete control over data movement in their applications. Host allocations are useful when data is used so infrequently that moving it is not worth the cost, or when the size of your data exceeds the memory of a device. Shared allocations are a happy medium that can automatically migrate to where they are being used, benefitting both performance and productivity.

3. Reductions

The SYCL 2020 approach to reductions was informed by other C++ reduction solutions, including the proposal in P0075 and the features implemented by the Kokkos and RAJA libraries.

Using the `reducer` class and the `reduction` function greatly simplifies the expression of variables with reduction semantics in SYCL kernels. It also gives implementations the freedom to employ compile-time specialization of reduction algorithms, providing high performance on a wide range of devices from many vendors.

For a real-life example of the improvements offered by SYCL 2020 reductions, we need look no further than the popular BabelStream benchmark, developed by the University of Bristol. BabelStream includes a simple dot product kernel that computes a floating-point summation across all work items in a kernel. The SYCL 1.2.1 version is 43 lines long, uses a specific algorithm (a tree reduction in work-group local memory), and requires the user to select the best work-group size for the device (**Figure 3**). Not only is the SYCL 2020 version shorter (at only 20 lines long), but it also has the potential to be more performance portable by leaving the selection of algorithm and work-group size to the implementation (**Figure 4**).

```
template <class T>
T SYCLStream<T>::dot()
{
  queue->submit([&](handler &cgh)
  {
    auto ka   = d_a->template get_access<access::mode::read>(cgh);
    auto kb   = d_b->template get_access<access::mode::read>(cgh);
    auto ksum = d_sum->template get_access<access::mode::write>(cgh);

    auto wg_sum = accessor<T, 1, access::mode::read_write,
access::target::local>(range<1>(dot_wgsize), cgh);

    size_t N = array_size;
    cgh.parallel_for<dot_kernel>(nd_range<1>(dot_num_groups*dot_wgsize, dot_wgsize),
[=](nd_item<1> item)
    {
      size_t i = item.get_global_id(0);
      size_t li = item.get_local_id(0);
      size_t global_size = item.get_global_range()[0];

      wg_sum[li] = 0.0;
      for (; i < N; i += global_size)
        wg_sum[li] += ka[i] * kb[i];

      size_t local_size = item.get_local_range()[0];
      for (int offset = local_size / 2; offset > 0; offset /= 2)
      {
        item.barrier(cl::sycl::access::fence_space::local_space);
        if (li < offset)
          wg_sum[li] += wg_sum[li + offset];
      }

      if (li == 0)
        ksum[item.get_group(0)] = wg_sum[0];
    });
  });

  T sum = 0.0;
  auto h_sum = d_sum->template get_access<access::mode::read>();
  for (int i = 0; i < dot_num_groups; i++)
  {
    sum += h_sum[i];
  }

  return sum;
}
```

Figure 3. SYCL 1.2.1 version of BabelStream's dot product kernel.

```
template <class T>
T SYCLStream<T>::dot()
{
  queue->submit([&](sycl::handler &cgh)
  {
    sycl::accessor ka {d_a, cgh, sycl::read_only};
    sycl::accessor kb {d_b, cgh, sycl::read_only};

    cgh.parallel_for(sycl::range<1>{array_size},
      sycl::reduction(d_sum, cgh, std::plus<T>(),
sycl::property::reduction::initialize_to_identity{}),
      [=](sycl::id<1> idx, auto& sum)
      {
        sum += ka[idx] * kb[idx];
      });
  });

  sycl::host_accessor result {d_sum, sycl::read_only};
  return result[0];
}
```

Figure 4. SYCL 2020 version of BabelStream's dot product kernel.

4. Group Library

SYCL 2020 expands on the work-group abstraction from SYCL 1.2.1 with a new sub-group abstraction and a library of group-based algorithms.

The `sub_group` class represents the set of cooperative work items within a kernel that are running "together," providing a portable abstraction for the underlying hardware capabilities of different vendors. In the DPC++ compiler, sub-groups always map to an important hardware concept — SIMD vectorization on Intel® architectures, "warps" on NVIDIA architectures, and "wavefronts" on AMD architectures — and enable low-level performance tuning for SYCL applications.

In another example of close alignment with ISO C++, SYCL 2020 introduces a selection of group-based algorithms based on the C++17 algorithms: `all_of`, `any_of`, `none_of`, `reduce`, `exclusive_scan` and `inclusive_scan`. Each algorithm is supported at different scopes, enabling SYCL implementations to provide highly tuned, cooperative versions of these functions using work-group and/or sub-group parallelism.

The group library in SYCL 2020 lays the groundwork for more group types and a wider range of group-based algorithms — watch this space!

5. Atomic References

C++20 took a big step forward with its atomics, introducing the ability to wrap types in an atomic reference (`std::atomic_ref`). SYCL 2020 adopts and extends this design (as `sycl::atomic_ref`) with support for address spaces and memory scopes, resulting in an atomic reference implementation fully prepared for the diverse world of heterogeneous computing.

SYCL does not deviate from ISO C++ lightly, and the concept of memory scopes was considered essential for enabling portable programming without sacrificing performance. Heterogeneous systems have complex memory hierarchies that shouldn't be ignored (**Figure 5**).

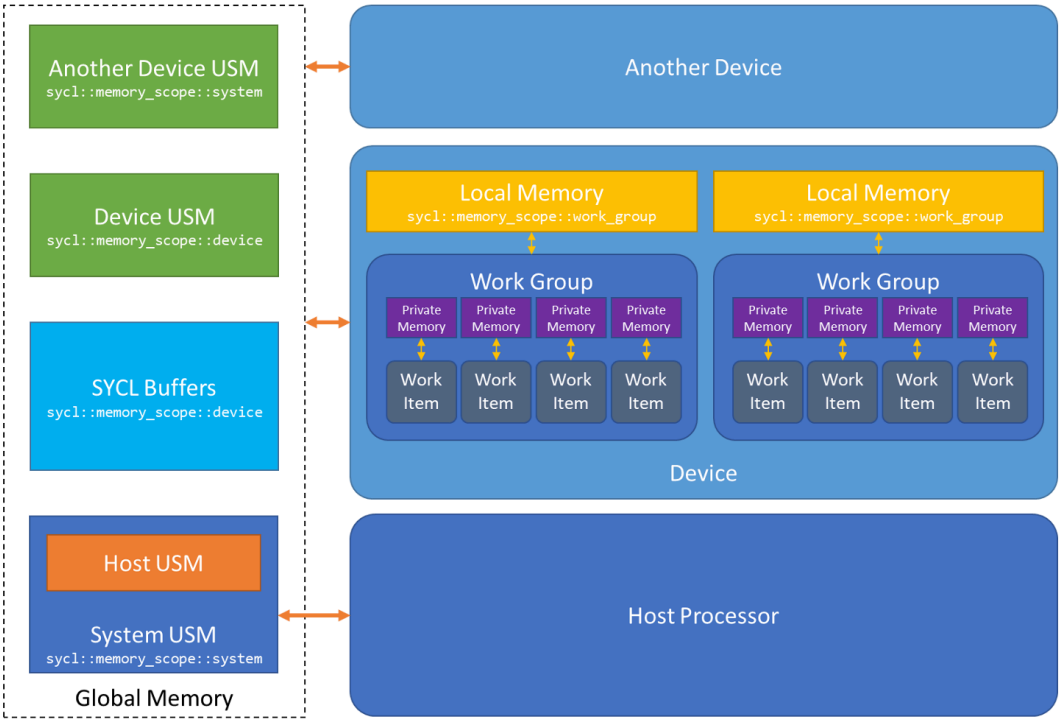


Figure 5. Using memory scopes enables atomic references to specify which memory must be made consistent, providing fine-grain control over which work-items and devices can "see" memory updates.

Memory models and atomics are complex beasts and so, in order to support as many devices as possible, SYCL does not require all devices to support the full C++ memory model. Rather, SYCL provides a rich array of different device capabilities — another great example of being open to devices from any vendor.

Beyond SYCL 2020: Vendor Extensions

SYCL 2020's expanded support for more backends and hardware has encouraged the development of more vendor extensions. These extensions enable innovation that offers practical solutions today for devices that need it and informs the direction of future SYCL standards. Extensions are an important part of the standardization process — several features highlighted in this article were informed in part by extensions explored by the DPC++ compiler project.

In this section, we'll briefly describe two new features supported in the DPC++ compiler project as SYCL 2020 vendor extensions.

Group-local Memory at Kernel Scope

SYCL 1.2.1 supports group-local memory via local accessors, which must be declared outside of a kernel and captured as a kernel argument. For programmers coming from languages like OpenCL or CUDA, this can feel unnatural, and so we have designed an extension that allows group-local memory to be declared inside of a kernel function. This change makes kernels more self-contained and can inform compiler optimizations (when the amount of local memory is known at compile-time).

FPGA-specific Extensions

We've enabled Intel® FPGAs in the DPC++ compiler project. We think our extensions, or something close to them, can prove portable to FPGAs from all vendors as well. FPGAs fill an important segment of the accelerator spectrum, and we hope our pioneering work will inform future SYCL standards with our experiences along with other extension projects from other vendors.

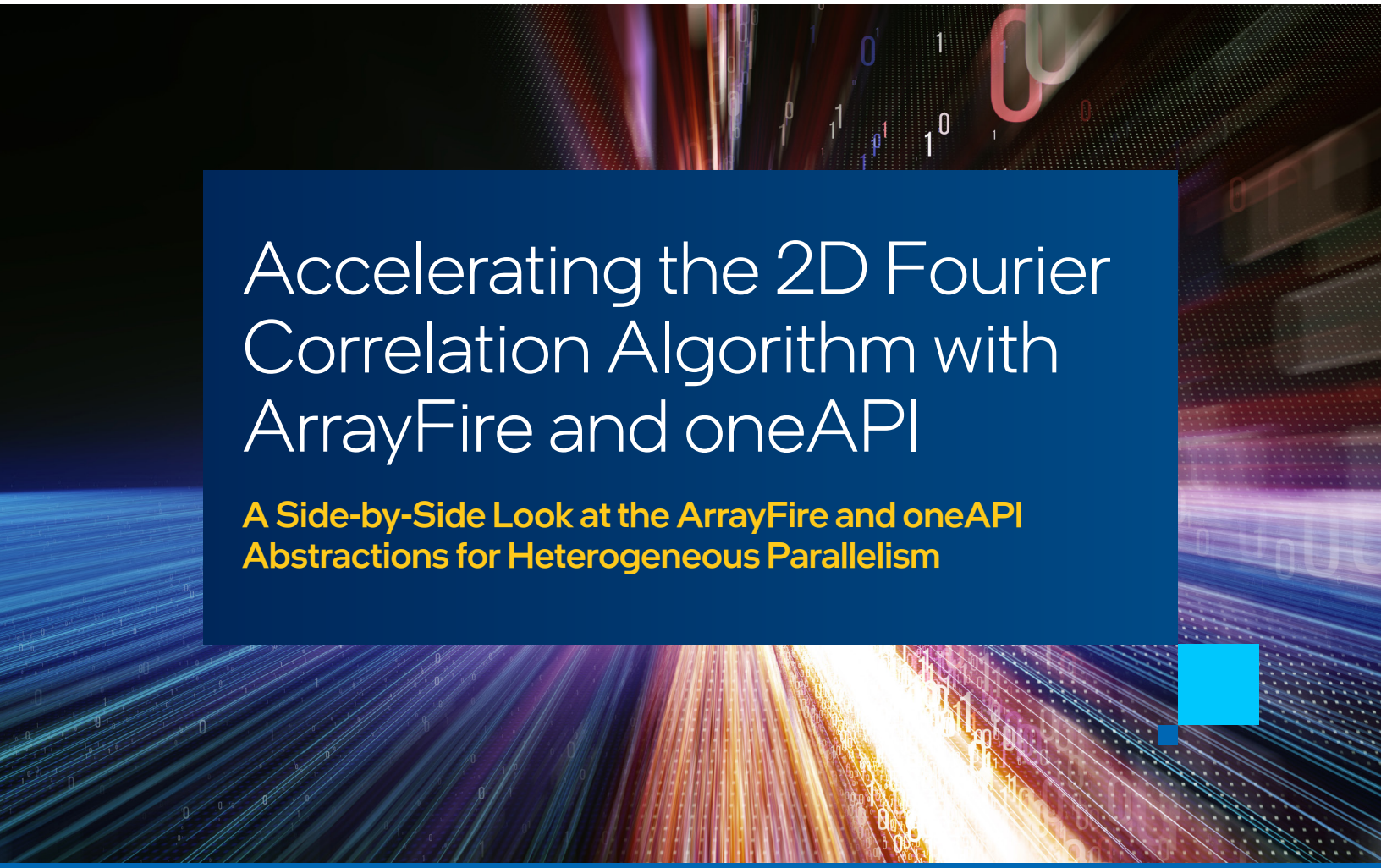
We added *FPGA selectors* that make it easy to specifically acquire an FPGA hardware or FPGA emulation device. The latter enables fast prototyping, a critical consideration for software developers when targeting FPGAs. *FPGA LSU controls* give us tuning controls for FPGA load/store operations — we can explicitly request that the implementation of a global memory access is configured in a certain way. We added placement controls for data with external memory banks (e.g., DDR channel) for tuning FPGA designs via *FPGA memory channel*. Key tuning controls for FPGA high performance pipelining are enabled with *FPGA register*.

Summary

Heterogeneity is here to stay. There is an increasing diversity of hardware options available, with many specializing in the pursuit of higher performance and performance-per-watt. This is a trend that will only increase the need for open, multivendor, and multiarchitecture programming models like SYCL.

We highlighted five new features in SYCL 2020 that help to fulfill its mission to enable portability and performance portability. With SYCL 2020, C++ programmers can fully use the potential of heterogeneous computing.

We invite you to sycl.tech to learn more. There you will find numerous online tutorials, a link for our SYCL book (available as a free PDF), and a link to the latest SYCL specification.



Accelerating the 2D Fourier Correlation Algorithm with ArrayFire and oneAPI

A Side-by-Side Look at the ArrayFire and oneAPI Abstractions for Heterogeneous Parallelism

Henry A Gabb, Senior Principal Engineer and Editor-in-Chief of The Parallel Universe, Intel Corporation
Umar Arshad, Software Engineer, ArrayFire

Implement the Fourier Correlation Algorithm Using oneAPI (*The Parallel Universe*, Issue 44) showed how to compute the 1D cross-correlation of two signals using a combination of SYCL and oneMKL functions. The present article looks at 2D cross-correlation to find the best overlap of two similar images (**Figure 1**). In addition to illustrating a 2D use case, this article expands on the previous work by comparing two approaches to write once, run anywhere heterogeneous parallelism: oneAPI and ArrayFire. [**Editor's note:** Readers may be interested in ArrayFire Interoperability with oneAPI, Libraries, and OpenCL (*The Parallel Universe*, Issue 47)].

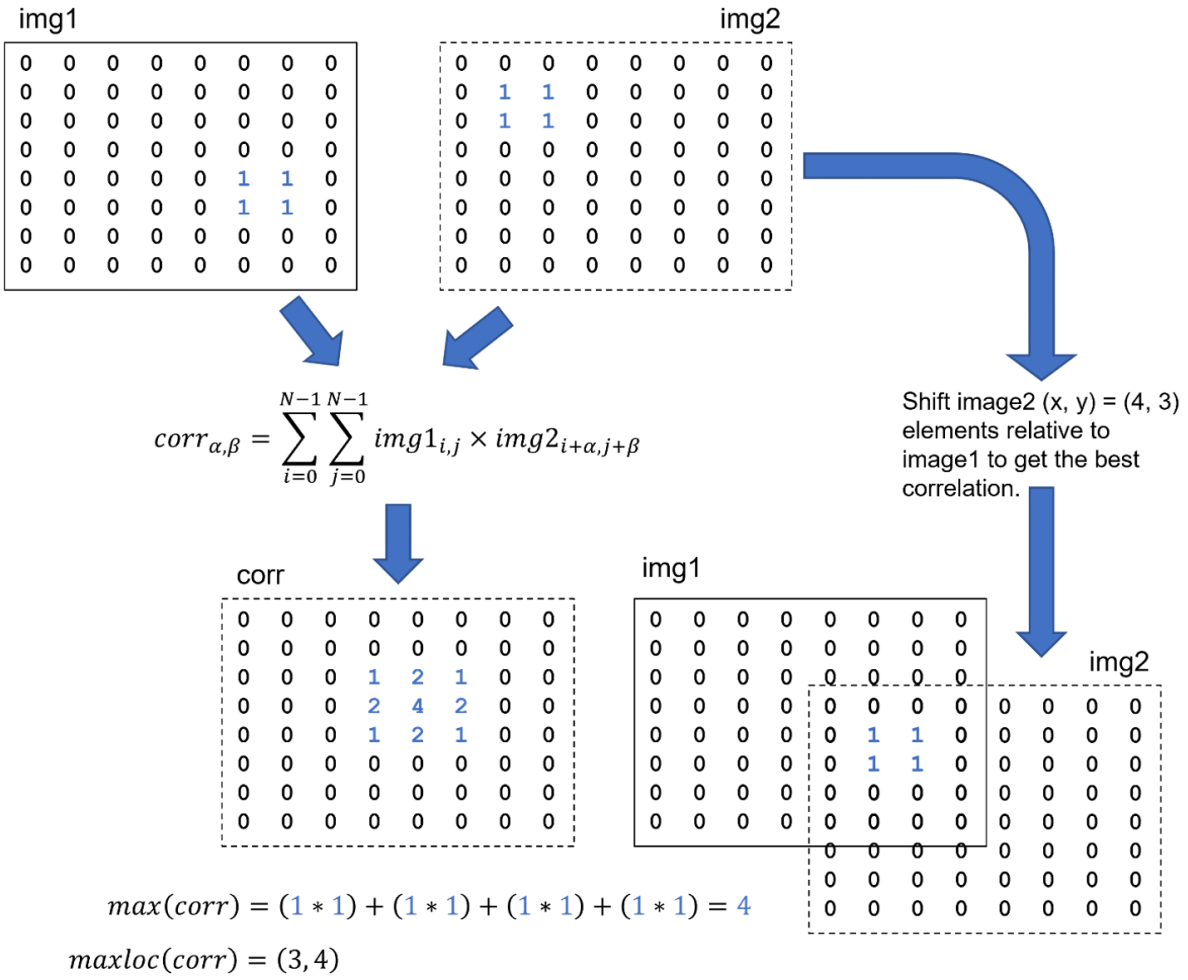


Figure 1. Finding the optimal alignment of two images (represented as binary, square matrices), where (α, β) is the displacement of img2 relative to img1. Note that the second image is circularly shifted when computing the correlation.

The brute force summation shown in **Figure 1** is inefficient and possibly infeasible for large problems, so like the previous 1D example, the 2D cross-correlation will also take advantage of the Fourier correlation algorithm to significantly reduce the computational complexity (**Figure 2**). The 2D Fourier correlation is implemented using both approaches, and the codes are shown side-by-side to illustrate their differences and relative strengths.

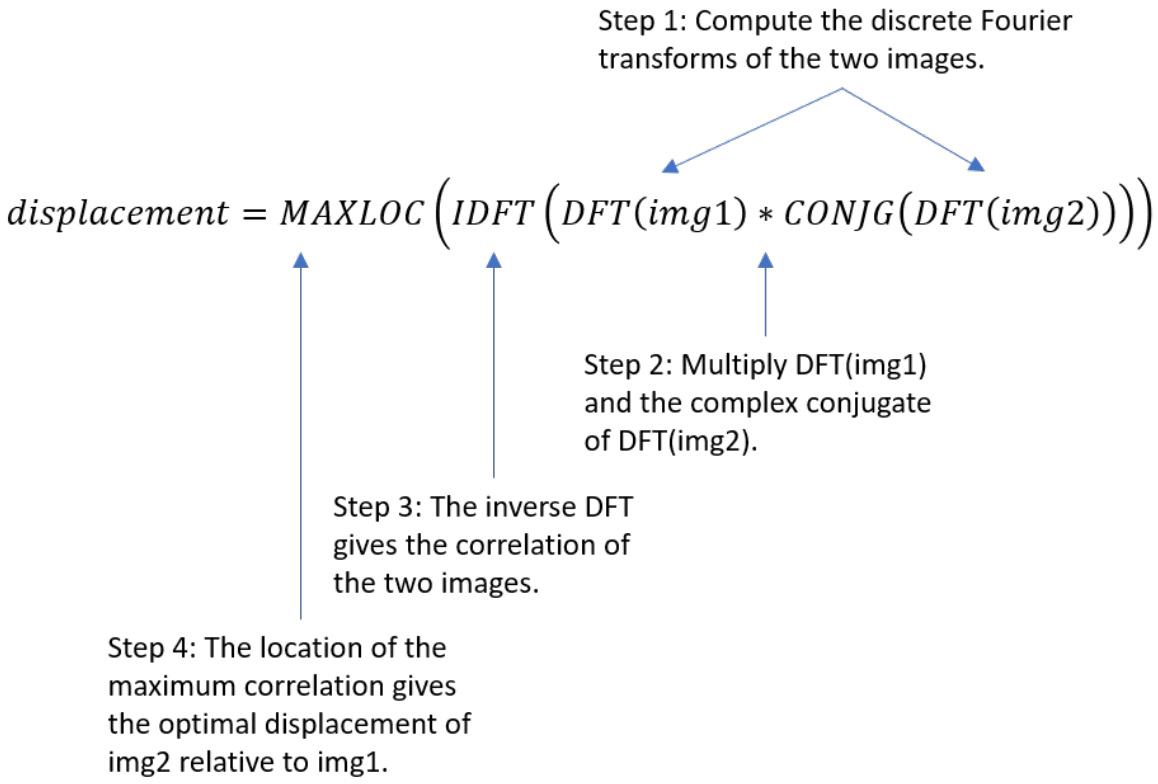


Figure 2. The Fourier correlation algorithm in four steps. DFT is the discrete Fourier transform, IDFT is the inverse DFT, CONJG is the complex conjugate, and MAXLOC is the location of the maximum correlation score.

The 2D oneAPI implementation also illustrates differences and improvements over the previous 1D implementation. First, multidimensional Fourier transforms in oneMKL must account for data layout in the real and complex domains. This is demonstrated in the oneAPI examples below. Second, the 1D implementation used a MAXLOC reduction operator written in SYCL. The 2D implementation replaces this code with standard functions from the oneAPI DPC++ Library (oneDPL). This makes the code clearer and much more succinct.

Implementing the Fourier Correlation Algorithm

Ideally, the entire computation should be performed on the device once the images are loaded into the device memory. Only the final displacement (two scalars in the 2D correlation) is needed by the host. Any other host-device data transfer is unnecessary, and will hurt performance, especially if the images are large.

Accelerator Offload with ArrayFire

It is easy to implement the Fourier correlation algorithm in ArrayFire because each step in **Figure 2** practically maps to a single statement (**Figure 3**). Even without the comments, this code should be readable by programmers familiar with C++ or MATLAB, Fortran, and NumPy array notation. There are no explicit loops in this code, and there is no explicit host-device data transfer or device offload. The coding abstraction is implicitly data parallel, so these details are handled by the ArrayFire runtime library.

```
#include <iostream>
#include <arrayfire.h>

int main(int argc, char **argv)
{
    // Select a device and display ArrayFire info
    af::setDevice(0);
    af::info();

    // Allocate and initialize 2D image array (Note: ArrayFire is column-major.)
    unsigned int n_rows = 8, n_cols = 8;

    auto img1 = af::constant(0.0, n_rows, n_cols, af::dtype::f32);
    auto img2 = af::constant(0.0, n_rows, n_cols, af::dtype::f32);
    auto corr = af::constant(0.0, n_rows, n_cols, af::dtype::f32);

    img1(af::seq(4, 5), af::seq(5, 6)) = 1.0f;  // Set elements in the lower right of the first image
    img2(af::seq(1, 2), af::seq(1, 2)) = 1.0f;  // Set elements in the upper left of the second image

    // Step 1: Compute DFT(img1) and DFT(img2)
    img1 = af::fftR2C<2>(img1, 0.0);
    img2 = af::fftR2C<2>(img2, 0.0);

    // Step 2: Compute DFT(img1) * CONJG(DFT(img2))
    corr = img1 * af::conjg(img2);

    // Step 3: Perform inverse DFT
    corr = af::fftC2R<2>(corr, 0.0);

    // Step 4: Find the optimal displacement of img2 relative to img1
    af::array max_score, shift;
    af::max(max_score, shift, af::flat(corr));

    auto max_corr = max_score.scalar<float>();
    auto s = shift.scalar<unsigned>();
    int x_shift = s / n_cols;
    int y_shift = s % n_rows;

    std::cout << std::endl << "Shift the second image (x, y) = (" << x_shift << ", " << y_shift
              << ") elements relative to the first image to get a maximum," << std::endl
              << "normalized correlation score of " << max_corr
              << ". Treat the images as circularly shifted versions of each other." << std::endl;
}
```

Figure 3. Fourier correlation algorithm implemented using ArrayFire.

Most ArrayFire functions are asynchronous, so the caller proceeds without waiting for the function to return. Function calls are added to an internal, in-order device queue. However, explicit synchronization can be used to wait until the queue is empty. For example:

```
// Step 1: Compute DFT(img1) and DFT(img2)
img1 = af::fftR2C<2>(img1, 0.0);
img2 = af::fftR2C<2>(img2, 0.0);
af::sync();
```

The sync statement forces the caller to wait until the two forward transforms are finished.

ArrayFire also does lazy evaluation of some computations. In this case, it stores the order of instructions, but it does not submit work to the device queue until the result is needed. Only then will a kernel be generated and added to the queue. The multiply-by-conjugate statement is an example of lazy evaluation:

```
// Step 2: Compute DFT(img1) * CONJG(DFT(img2))
corr = img1 * af::conjg(img2);
corr.eval();
af::sync();
```

The `eval` statement forces a kernel to be created and added to the queue. The `sync` statement forces the caller to wait until the multiply-by-conjugate operation is finished.

Accelerator Offload with oneAPI

Implementing a 1D Fourier correlation using oneAPI has been demonstrated in a previous article and webinar. The principles are the same for the 2D case, so rather than explain each step of the oneAPI implementation, this section will compare the oneAPI and ArrayFire code and highlight differences between these two approaches to heterogeneous parallelism.

Initializing the Images on the Device

Data movement between the host CPU and various accelerator devices is an important consideration in heterogeneous parallel programming. If some steps of an algorithm are performed on the host and others on the device, back-and-forth data transfer could limit the performance benefit of accelerator offload. Fortunately, each step of the Fourier correlation algorithm can be done on the device. Once the images are transferred to the device memory, they do not need to be transferred back to the host.

The ArrayFire and SYCL code to initialize the data on the device is shown in **Figure 4**. The same artificial images from **Figure 1** are used for the sake of simplicity. After setting the offload device, the ArrayFire code initializes the data using convenience functions and array syntax (**Figure 4**, left). Likewise, the oneAPI code initializes a SYCL queue for the default device, allocates sufficient space in the unified shared memory for an in-place, real-to-complex transform, and defines the data layout (**Figure 4**, right). (Describing the data layout for multidimensional DFTs is beyond the scope of this article, but

the FFTW documentation provides a good overview.) The SYCL code performs the initialization on the device by submitting work to the SYCL queue (that is, the `parallel_for` and `single_task` kernels). SYCL queues are asynchronous, so these kernels are explicitly told to wait for the work to finish before proceeding.

```
af::setDevice(0);

auto img1 = af::constant(0.0,
                         n_rows,
                         n_cols,
                         af::dtype::f32);
auto img2 = af::constant(0.0,
                         n_rows,
                         n_cols,
                         af::dtype::f32);
auto corr = af::constant(0.0,
                         n_rows,
                         n_cols,
                         af::dtype::f32);

img1(af::seq(4, 5), af::seq(5, 6)) = 1.0f;
img2(af::seq(1, 2), af::seq(1, 2)) = 1.0f;
```

```
// Initialize SYCL queue
sycl::queue Q(sycl::default_selector{});

// Allocate 2D image and correlation arrays
auto img1 = sycl::malloc_shared<float>(n_rows*n_cols*2+2, Q);
auto img2 = sycl::malloc_shared<float>(n_rows*n_cols*2+2, Q);
auto corr = sycl::malloc_shared<float>(n_rows*n_cols*2+2, Q);

// Set generalized strides for row-major addressing
int r_stride = 1;
int c_stride = (n_cols / 2 + 1) * 2;
int c_stride_h = (n_cols / 2 + 1);

// Initialize input images with artificial data.
// Do initialization on the device.
Q.parallel_for<>(sycl::range<2>{n_rows, n_cols},
                 [=](sycl::id<2> idx)
{
   unsigned int r = idx[0];
   unsigned int c = idx[1];
   img1[r * c_stride + c * r_stride] = 0.0;
   img2[r * c_stride + c * r_stride] = 0.0;
   corr[r * c_stride + c * r_stride] = 0.0;
}).wait();

Q.single_task<>([=]()
{
   // Set elements in lower right of the first image
   img1[4 * c_stride + 5 * r_stride] = 1.0;
   img1[4 * c_stride + 6 * r_stride] = 1.0;
   img1[5 * c_stride + 5 * r_stride] = 1.0;
   img1[5 * c_stride + 6 * r_stride] = 1.0;

   // Set elements in upper left of the second image
   img2[1 * c_stride + 1 * r_stride] = 1.0;
   img2[1 * c_stride + 2 * r_stride] = 1.0;
   img2[2 * c_stride + 1 * r_stride] = 1.0;
   img2[2 * c_stride + 2 * r_stride] = 1.0;
}).wait();
```

Figure 4. Initializing the data on the device using ArrayFire (left) and SYCL (right).

The host should not modify the data once it is in the device memory because this will trigger unnecessary host-device transfer. Notice that the image elements are set using a `single_task` kernel. Without this kernel, the elements would be set on the host, forcing the oneAPI runtime to make the host and device data consistent. This could hurt performance if the data is large (e.g., volumetric images from medical imaging applications).

The ArrayFire code is more compact and intuitive. The oneAPI code is more explicit about where data is allocated and where and when computation is performed. It is also more consistent with FFTW, the popular open-source fast Fourier transform package.

Step 1: Performing the Forward Transforms

Once again, the ArrayFire code is simple and intuitive (**Figure 5**, left). The function call specifies a 2D real-to-complex (R2C), unnormalized FFT with precision defined by the input data. The oneMKL DFT descriptor approach (**Figure 5**, right), while not as compact, is familiar to previous MKL DFTI and FFTW users. The oneMKL code initializes a descriptor for a single-precision, real-to-complex transform of the required size and dimensionality, commits this descriptor to the SYCL queue, and then computes the forward transforms. The `compute_forward` function returns a SYCL event that is used later for synchronization.

```
img1 = af::fftR2C<2>(img1, 0.0);
img2 = af::fftR2C<2>(img2, 0.0);
```

```
// Initialize FFT descriptor
oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,
                             oneapi::mkl::dft::domain::REAL>
                                       forward_plan({n_rows, n_cols});

// Data layout in real domain
std::int64_t real_layout[4] = {0, c_stride, 1};

// Data layout in conjugate-even domain
std::int64_t complex_layout[4] = {0, c_stride_h, 1};

forward_plan.set_value(oneapi::mkl::dft::config_param::INPUT_STRIDES,
                       real_layout);
forward_plan.set_value(oneapi::mkl::dft::config_param::OUTPUT_STRIDES,
                       complex_layout);
forward_plan.commit(Q);

auto evt1 = oneapi::mkl::dft::compute_forward(forward_plan, img1);
auto evt2 = oneapi::mkl::dft::compute_forward(forward_plan, img2);
```

Figure 5. Performing the forward transforms (real-to-complex) on the device using ArrayFire (left) and oneMKL (right).

Step 2: Complex Conjugate Multiplication

ArrayFire and oneMKL use different approaches to perform the multiply-by-conjugate operation. ArrayFire uses a straightforward array notation (**Figure 6**, left). The programmer does not have to specify the device or data layout. oneMKL provides a convenient mulbyconj function (**Figure 6**, right). The mulbyconj function is more complex, but it gives the programmer explicit control over the data layout and where and when the computation runs (i.e., the SYCL queue and events).

```
corr = img1 * af::conjg(img2);
```

```
oneapi::mkl::vm::mulbyconj(Q, n_rows * c_stride_h,
                           reinterpret_cast<std::complex<float>*>(img1),
                           reinterpret_cast<std::complex<float>*>(img2),
                           reinterpret_cast<std::complex<float>*>(corr),
                           {evt1, evt2}).wait();
```

Figure 6. Complex conjugate multiplication using ArrayFire (left) and oneMKL (right).

Step 3: Performing the Backward Transform

The code for steps 1 and 3 is similar except that only a single complex-to-real transform is performed. The ArrayFire code (**Figure 7**, left) calls `fftC2R` instead of the `fftR2C` function. The oneMKL code (**Figure 7**, right) initializes a new DFT descriptor specifying the complex-to-real data layout.

```
corr = af::fftC2R<2>(corr, 0.0);
```

```
oneapi::mkl::dft::descriptor<oneapi::mkl::dft::precision::SINGLE,
                             oneapi::mkl::dft::domain::REAL>
                                      backward_plan({n_rows, n_cols});

// Data layout in conjugate-even domain
backward_plan.set_value(oneapi::mkl::dft::config_param::INPUT_STRIDES,
                        complex_layout);

// Data layout in real domain
backward_plan.set_value(oneapi::mkl::dft::config_param::OUTPUT_STRIDES,
                        real_layout);
backward_plan.commit(Q);

auto bwd = oneapi::mkl::dft::compute_backward(backward_plan, corr);
bwd.wait();
```

Figure 7. Performing the backward transforms (complex-to-real) on the device using ArrayFire (left) and oneMKL (right).

Step 4: MAXLOC Reduction

The SYCL `MAXLOC` reduction operator used in previous experiments added complexity to the code (see `fcorr_1d_usm.cpp`). oneDPL provides the same functionality in two functions that will be familiar to C++ programmers: `max_element` and `distance`. The `MAXLOC` reduction shown in **Figure 8** (right) uses the oneDPL implementations of these functions. The SYCL queue tells `max_element` where to perform the computation. It is worth noting that unlike SYCL kernels, oneDPL algorithms are synchronous so there are no explicit wait statements.

The ArrayFire code (**Figure 8**, left) is not as straightforward as previous steps. The `af::flat` function flattens the 2D `corr` array, and then the `af::max` function finds the maximum correlation score and its location in the flattened array. This location is converted to the x- and y-shift that gives the optimal alignment of the two images, taking into account that ArrayFire is column-major (as noted in **Figure 3**). The oneDPL code (**Figure 8**, right) performs similar operations, but must take the oneMKL data layout into account.

```
af::array max_score, shift;
af::max(max_score, shift, af::flat(corr));

auto max_corr = max_score.scalar<float>();
auto s = shift.scalar<unsigned>();
int x_shift = s / n_cols;
int y_shift = s % n_rows;
```

```
auto policy = oneapi::dpl::execution::make_device_policy(Q);
auto maxloc = oneapi::dpl::max_element(policy,
                                       corr,
                                       corr + (n_rows * n_cols * 2 + 2));

auto s = oneapi::dpl::distance(corr, maxloc);
float max_corr = corr[s];
int x_shift = s % (n_cols + 2);
int y_shift = s / (n_rows + 2);
```

Figure 8. Performing the MAXLOC reduction in ArrayFire (left) and oneDPL (right).

Conclusions

The oneAPI and ArrayFire approaches both accomplish the goal of write once, run anywhere heterogeneous parallelism. Performance is not discussed because the entire 2D Fourier correlation computation is done in oneAPI or ArrayFire libraries. The separation of concerns between applications developers and compiler/library developers is a recurring theme in *The Parallel Universe*. The latter group is primarily concerned with performance and computing efficiency. The former group is primarily concerned with solving a problem as productively as possible. If you're in this group, you probably prefer high-level programming abstractions that still deliver performance. It's a reasonable expectation that the libraries will give good performance.

ArrayFire provides a higher level of abstraction, so the ArrayFire Fourier correlation implementation is more concise. Its array notation will be familiar to Fortran, MATLAB, and Python NumPy programmers.

ArrayFire also has a Python API. Lazy evaluation means the runtime controls when computations are launched, but as noted above, programmers can take control if they want to.

The oneAPI implementation is more verbose because it gives the programmer more control over host-device data transfer and where and when computations are performed. The oneMKL DFT descriptors and data layout will be familiar to previous MKL DFTI and FFTW users. In fact, Intel® oneAPI Math Kernel Library supports the FFTW interface. Finally, oneDPL functions will be familiar to C++ programmers.

Ultimately, project requirements and personal preference will guide the choice between oneAPI and ArrayFire.

intel software

Break Free of Code Boundaries

Experience the power of cross-architecture programming in the Intel® DevCloud for oneAPI.

Demo

Run our Mandelbrot demo on different architectures to see cross-architecture performance for yourself.

Learn

Get hands-on experience with Data Parallel C++ with 25 Jupyter notebooks loaded with code samples.

Develop

Plan and test future-ready applications on the latest Intel CPUs, GPUs, and FPGAs.

GET STARTED NOW >

The Maxloc Reduction in oneAPI

Implementing This Common Parallel Pattern in SYCL and oneDPL

Henry A Gabb, Senior Principal Engineer and Editor-in-Chief of The Parallel Universe; Alexey Kukanov, Principal Middleware Engineer; and John Pennycook, Software Enabling and Optimization Architect, Intel Corporation

What Is the Maxloc Operation?

Finding the location of the maximum value (maxloc) is a common search operation performed on arrays. It's such a common operation that many programming languages and libraries provide intrinsic maxloc functions: the NumPy argmax, Fortran maxloc, BLAS amax, and C++ max_element functions. The recent article, Optimize the Maxloc Operation Using Intel® AVX-512 Instructions (*The Parallel Universe*, Issue 46), explained how to vectorize maxloc searches for best performance. Obviously, it's an important operation in many algorithms, including cross-correlation (**Figure 1**).

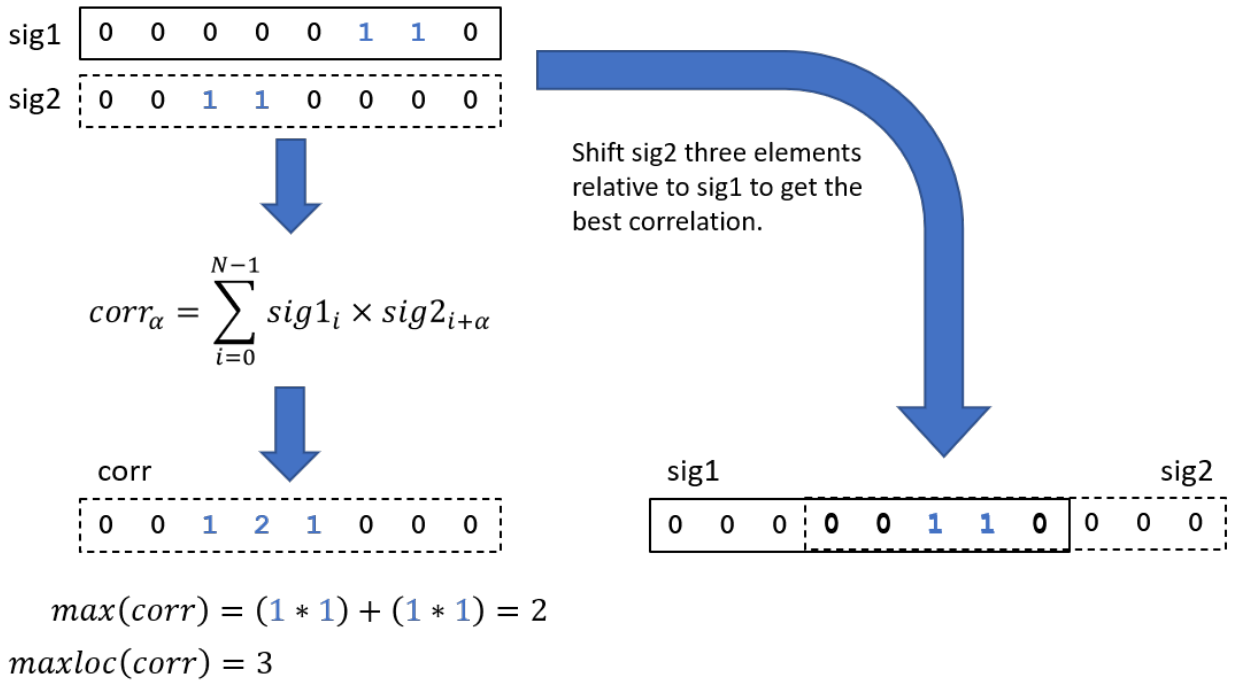


Figure 1. Finding the displacement that gives the maximum overlap of two discrete signals (represented as binary arrays), where α is the number of elements by which sig2 is shifted relative to sig1. Note that the second signal is circularly shifted when computing the correlation.

The previous article, Implement the Fourier Correlation Algorithm Using oneAPI (*The Parallel Universe*, Issue 44), showed how to compute the 1D cross-correlation of two signals using a combination of SYCL and oneMKL functions, but the final maxloc step was ommitted (**Figure 2**). Steps 1 through 3 were offloaded to the accelerator device, but step 4 was computed on the host CPU. This means that the final correlation array had to be transferred back to the host. Ideally, the entire computation should be performed on the device once the signals are loaded into the device memory. Only the final displacement (a single scalar in the 1D correlation) is needed by the host. Any other host-device data transfer is unnecessary and could hurt performance. Therefore, we've been experimenting with different ways to perform maxloc on the device, which is the subject of the present article.

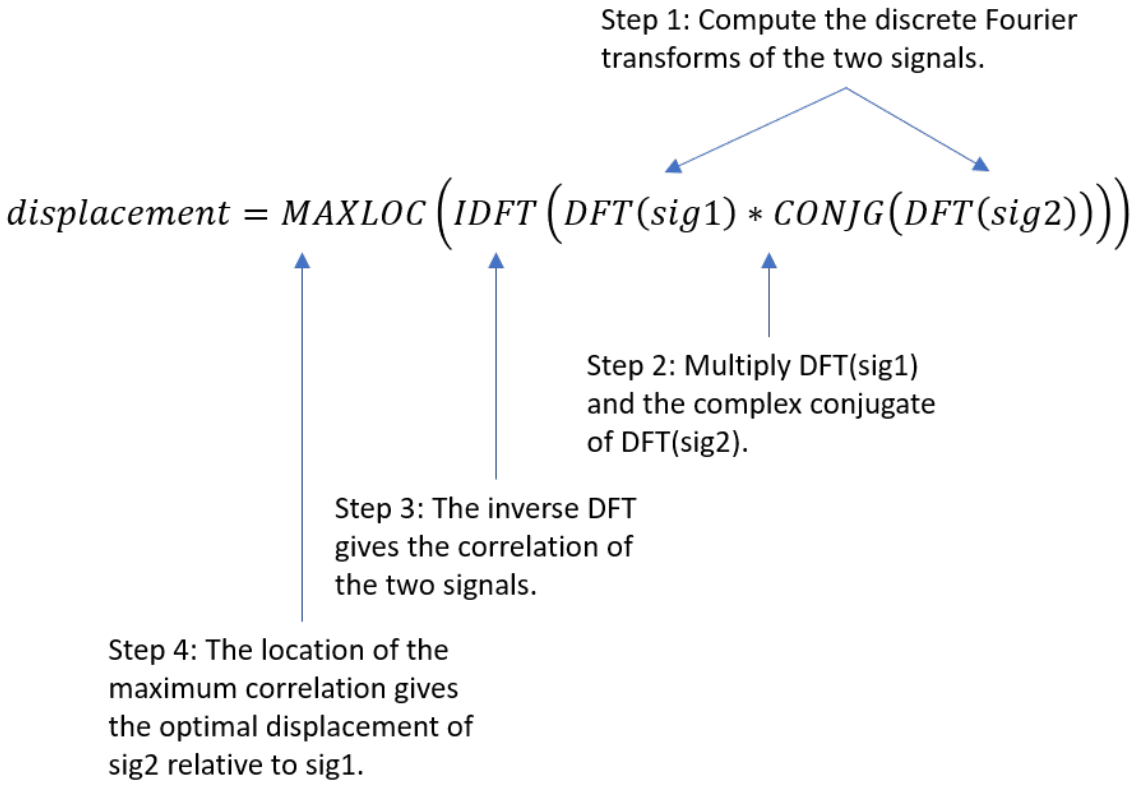


Figure 2. The maxloc reduction is the last step of the Fourier correlation algorithm. DFT is the discrete Fourier transform, IDFT is the inverse DFT, CONJG is the complex conjugate, and MAXLOC is the location of the maximum correlation score.

Reduction Operators in SYCL*

Reduction is a common parallel pattern that reduces several values to a single value. For example, a summation reduction adds the values in an array to get a single sum. Finding the minimum or maximum value in an array, or the locations of those values, are also reduction operations. SYCL* provides a built-in reduction operator that can be used in parallel kernels (**Figure 3**).

```
#include <CL/sycl.hpp>
#include <iostream>

int main() {

    sycl::queue Q;
    std::cout << "Running on: " << Q.get_device().get_info<sycl::info::device::name>() << std::endl;

    int sum;
    std::vector<int> data{1, 1, 1, 1, 1, 1, 1, 1};

    sycl::buffer<int> sum_buf(&sum, 1);
    sycl::buffer<int> data_buf(data);

    Q.submit([&](sycl::handler& h)
    {
        sycl::accessor buf_acc{data_buf, h, read_only};

        h.parallel_for(sycl::range<1>{8},
                       sycl::reduction(sum_buf, h, std::plus<>()),
                       [=](sycl::id<1> idx, auto& sum)
        {
            sum += buf_acc[idx];
        });
    });
    sycl::host_accessor result{sum_buf, read_only};
    std::cout << "Sum equals " << result[0] << std::endl;

    return 0;
}
```

Figure 3. Example of summation reduction in SYCL.

It is possible to implement other operators, like maxloc, using the SYCL reduction class (**Figure 4**). This code was adapted from the minloc example in Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL (Chapter 14, Common Reduction Patterns, pp. 334–339). Note that SYCL queues are asynchronous, so the wait statement ensures that the computation is finished before proceeding.

```
#include <iostream>
#include <CL/sycl.hpp>

template <typename T, typename I>
using maxloc = sycl::maximum<std::pair<T, I>>;

constexpr size_t L = 1;

int main(int argc, char **argv)
{
   sycl::queue Q;
   const size_t n = 7;
   float* data = sycl::malloc_shared<float>(n, Q);
   data[0] = 1; data[1] = 1; data[2] = 1; data[3] = 2; data[4] = 1; data[5] = 1; data[6] = 1;

   std::pair<float, int>* max_res = sycl::malloc_shared<std::pair<float, int>>(1, Q);
   std::pair<float, int> max_identity = {
       std::numeric_limits<float>::min(), std::numeric_limits<int>::min()
   };
   *max_res = max_identity;
   auto red_max = sycl::reduction(max_res, max_identity, maxloc<float, int>());

   Q.parallel_for(sycl::nd_range<1>{n, L}, red_max, [=](sycl::nd_item<1> item, auto& max_res) {
       int i = item.get_global_id(0);
       std::pair<float, int> partial = {data[i], i};
       max_res.combine(partial);
   }).wait();

   std::cout << "Maximum value = " << max_res->first << " at element " << max_res->second << std::endl;

   sycl::free(data, Q.get_context());
   sycl::free(max_res, Q.get_context());

   return 0;
}
```

Figure 4. Implementing maxloc as a SYCL reduction operator.

This example is a straightforward use of the SYCL reduction class, but there are ways to tune for the underlying architecture. The previous articles, Reduction Operations in Data Parallel C++ (*The Parallel Universe*, Issue 44) and Analyzing the Performance of Reduction Operations in DPC++ (*The Parallel Universe*, Issue 45), give examples of different implementations and their performance characteristics. With the built-in reduction operator, however, compiler and library developers bear the optimization burden. Application developers can reasonably expect that the built-in implementation will deliver good performance on the target CPU or accelerator.

Doing the Same Reduction with oneDPL

The Intel® oneAPI Data Parallel C++ Library (oneDPL) provides an alternative for programmers who would rather call a function than use the SYCL reduction class. From Data Parallel C++ (p. 339):

> *"The C++ Standard Template Library (STL) contains several algorithms which correspond to the parallel patterns... The algorithms in the STL typically apply to sequences specified by pairs of iterators and — starting with C++17 — support an execution policy argument denoting whether they should be executed sequentially or in parallel. [oneDPL] leverages this execution policy argument to provide a high-productivity approach to parallel programming that leverages kernels written in DPC++ under the hood. If an application can be expressed solely using functionality of the STL algorithms, oneDPL makes it possible to make use of the accelerators in our systems without writing a single line of DPC++ kernel code!"*

There's a lot to like in this description, but let's highlight two points:

1. **C++ STL:** The functions will be familiar to C++ programmers.
2. **High productivity:** The coding and performance burden is on the STL developers, where it belongs. The application developer can access an accelerator without writing lower-level SYCL kernels.

The advantages become apparent when you compare the kernel-based maxloc code (**Figure 4**) to the oneDPL implementation (**Figure 5**). The latter uses the familiar `max_element` function to perform the maxloc reduction. SYCL kernels are leveraged "under the hood," as noted in the previous quote. The oneDPL default execution policy places the computation on an accelerator if one is available. Otherwise, the computation runs on the host CPU.

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <iostream>

int main()
{
   std::vector<int> data{1, 1, 1, 2, 1, 1, 1};

   auto policy = oneapi::dpl::execution::dpcpp_default;
   auto maxloc = oneapi::dpl::max_element(policy, data.cbegin(), data.cend());

   std::cout << "Run on "
             << policy.queue().get_device().template get_info<sycl::info::device::name>()
             << std::endl;
   std::cout << "Maximum value is at element " << oneapi::dpl::distance(data.cbegin(), maxloc) << std::endl;

   return 0;
}
```

Figure 5. Basic maxloc reduction using the oneDPL max_element and distance functions and implicit host-device data transfer.

Notice that host-device data transfer is being handled implicitly in **Figure 5**. The oneDPL runtime automatically wraps the data in a temporary buffer if the computation is being offloaded to an accelerator. In a larger oneAPI program, it's possible that the data are already in SYCL buffers, so oneDPL functions accept buffers and can iterate over them (**Figure 6**). Whether the buffered data are transferred to the device implicitly or explicitly (i.e., via SYCL buffers), the buffers don't need to be transferred back to the host unless they are modified, thus avoiding unnecessary overhead.

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <iostream>

int main()
{
   std::vector<int> data{1, 1, 1, 2, 1, 1, 1};
   sycl::buffer<int> data_buf(data);

   auto policy = oneapi::dpl::execution::dpcpp_default;
   auto maxloc = oneapi::dpl::max_element(policy, oneapi::dpl::begin(data_buf), oneapi::dpl::end(data_buf));

   std::cout << "Run on "
             << policy.queue().get_device().template get_info<sycl::info::device::name>()
             << std::endl;
   std::cout << "Maximum value is at element "
         << oneapi::dpl::distance(oneapi::dpl::begin(data_buf), maxloc) << std::endl;

   return 0;
}
```

Figure 6. Maxloc reduction using oneDPL and SYCL buffering for explicit host-device data transfer.

The oneDPL functions also accept pointers to unified shared memory (USM) to handle host-device data transfer (**Figure 7**). In this example, space is allocated in the appropriate USM, using the SYCL `malloc_shared` function and a SYCL queue. The same queue is used to set the oneDPL execution policy. It is worth noting that oneDPL algorithms are synchronous, so there are no explicit wait statements in **Figures 5–7**. SYCL kernels, on the other hand, are asynchronous.

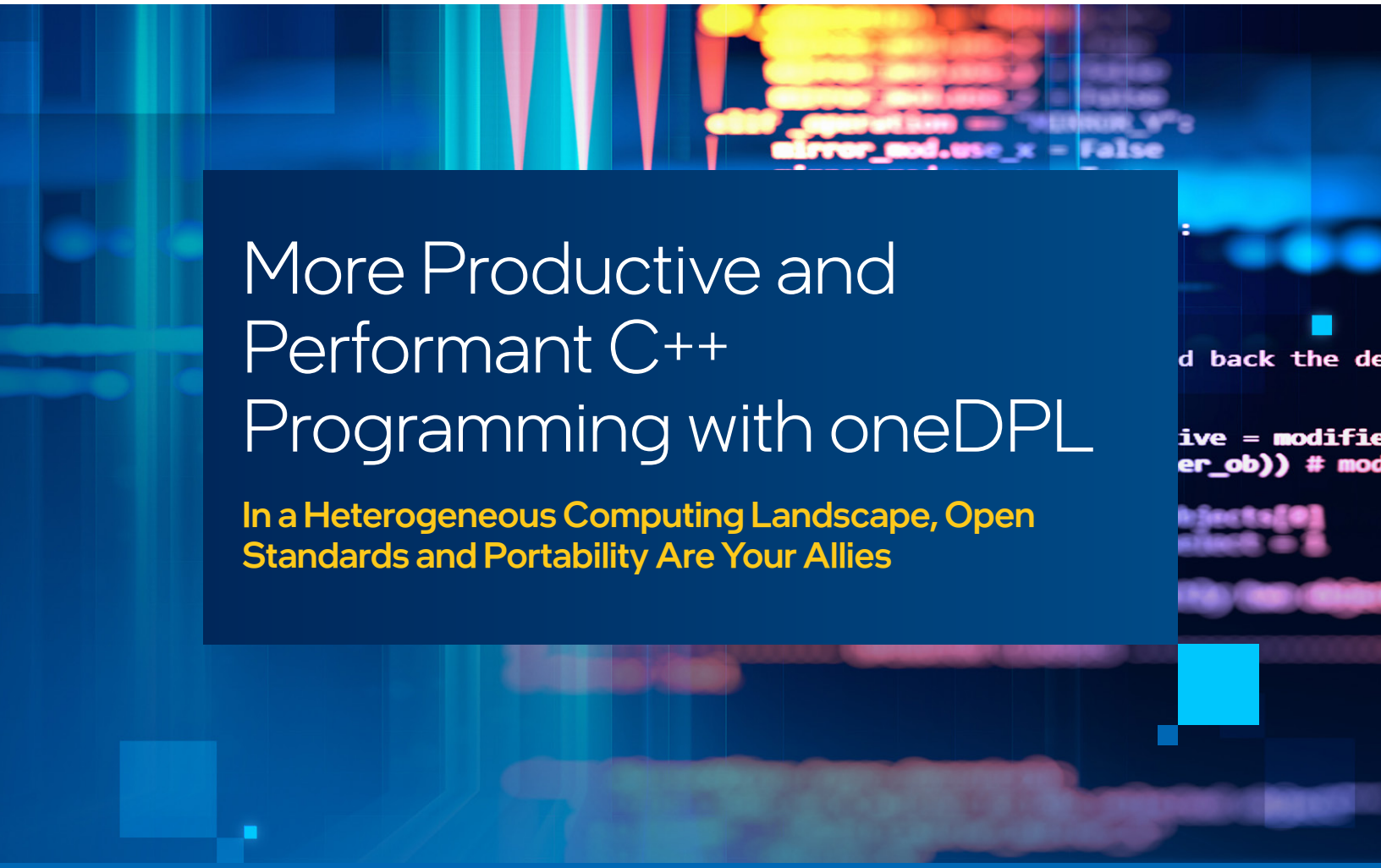
```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <iostream>

int main()
{
   sycl::queue Q(sycl::default_selector{});
   auto policy = oneapi::dpl::execution::make_device_policy(Q);

   const size_t n = 7;
   auto data = sycl::malloc_shared <int>(n, Q);

   data[0] = 1; data[1] = 1; data[2] = 1; data[3] = 2; data[4] = 1; data[5] = 1; data[6] = 1;

   auto maxloc = oneapi::dpl::max_element(policy, data, data + n);

   std::cout << "Run on "
             << policy.queue().get_device().template get_info<sycl::info::device::name>()
             << std::endl;
   std::cout << "Maximum value is at element " << oneapi::dpl::distance(data, maxloc) << std::endl;

   sycl::free(data, Q);
   return 0;
}
```

Figure 7. Maxloc reduction using oneDPL and USM for host-device data transfer.

When Should I Use a SYCL Reduction or oneDPL?

Like many programming questions, there's no definitive answer. There are valid reasons for using one or the other approach. It depends on your requirements. If oneDPL provides an algorithm that matches your requirements, calling a standard function is simpler than writing a SYCL kernel. For example, if there's already a big array in device memory and only the scalar output from the reduction is required by the host, calling the oneDPL function is probably best. However, the function call is synchronous, so the program blocks until the function returns, which is not always desirable. The SYCL reduction is more complicated, but it is asynchronous, more flexible, and provides more tuning opportunities. For example, if you're transforming the data on which the reduction is being performed or performing multiple reductions simultaneously, writing a SYCL kernel might be preferable.

The separation of concerns between applications developers and compiler/library developers is a recurring theme in *The Parallel Universe*. The latter group is primarily concerned with performance and computing efficiency. The former group is primarily concerned with solving a problem as productively as possible. If you're in this group, you probably prefer high-level programming abstractions that still deliver performance. The SYCL `maxloc` reduction operator shown in **Figure 4** is complicated, low level, and may require some architecture-specific tuning (e.g., of work-group sizes) to achieve best performance. The oneDPL examples shown in **Figures 5–7** are simpler, familiar to C++ STL users, and versatile in terms of host-device data transfer. More importantly, they shift the tuning burden to oneDPL product developers who are mainly concerned about performance. Consequently, oneDPL functions should deliver good performance regardless of the underlying architecture. This is the promise of the oneAPI software abstraction for heterogeneous computing.



More Productive and Performant C++ Programming with oneDPL

In a Heterogeneous Computing Landscape, Open Standards and Portability Are Your Allies

Pablo Reble, Software Engineer, Intel Corporation

We have come a long way since 2005 when Herb Sutter declared that "the free lunch is over," referring to challenges that programmers face with emerging multicore processors. Today's portfolio of computing architectures and accelerators is even richer and constantly growing: a development driven by fundamental limitations of semiconductors and the desire for more powerful, energy-efficient computing. The computing world is becoming more and more heterogeneous, which creates challenges for programmers.

C++ is still among the five most popular programming languages (TIOBE ranks it #4 as of January 2022). Attributes like full control over memory management and support for generic programming make it a great language to tackle heterogeneous programming challenges. Developer productivity and the cost of code maintenance are common concerns when choosing a programming language. Fortunately, previous studies show that we can expect a productivity boost by combining parallel building blocks with C++ algorithms. For example, optimized, built-in implementations of common functions and patterns (e.g., reduction) for specific architectures improve both performance and developer productivity.[1, 2]

Our industry-leading implementation of the Intel® oneAPI Data Parallel C++ Library (oneDPL) was contributed to the open-source LLVM project. As a result, developer effort can be significantly reduced in a multithreaded world.[1, 6]

Supercharged Classic STL Algorithms

Boost your code with something old and something new.

The C++ language itself is evolving, and so is its standard template library (STL). For example, five years ago execution policies were added to the algorithms library so that even existing C++ codes can benefit from the ubiquitous parallelism of modern processors. You can think of oneDPL as a supercharged C++ STL that allows different vendors to implement accelerated versions of classic algorithms in a portable way.

oneDPL implements the C++ algorithms library using SYCL*:

> *"SYCL (pronounced 'sickle') is a royalty-free, cross-platform abstraction layer that enables code for heterogeneous processors to be written using standard ISO C++ with the host and kernel code for an application contained in the same source file."*[3]

There is a learning curve for direct accelerator programming in SYCL. While C++ gives programmers full control over memory management, it has no concept of separate host and device memories. oneDPL relies on SYCL's memory abstraction as a portable way to share data between host and device(s). oneDPL algorithm functions are ready to use, familiar to C++ programmers, and optimized for a variety of accelerators. This flattens the learning curve and improves code performance and developer productivity.

[1] "The oneDPL library is built on top of SYCL and so it is particularly interesting to see that it outperforms native SYCL code." Deakin et al. 2021 (p. 40)[2]

Here's a simple example to illustrate the power of oneDPL:

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <iostream>

int main()
{
   std::vector<int> data{1, 1, 1, 2, 1, 1, 1};

   auto policy = oneapi::dpl::execution::dpcpp_default;
   auto maxloc = oneapi::dpl::max_element(policy, data.cbegin(), data.cend());

   std::cout << "Run on "
             << policy.queue().get_device().template get_info<sycl::info::device::name>()
             << std::endl;
   std::cout << "Maximum value is at element " << oneapi::dpl::distance(data.cbegin(), maxloc) << std::endl;

   return 0;
}
```

This example offloads the common maxloc reduction (i.e., finding the element in the data set with the maximum value) to the accelerator specified in the execution policy. The included headers are conformant with ISO C++, and so is the blocking behavior of `max_element`. Data movement is handled implicitly in this example. In other words, the runtime automatically handles host-device data transfer by wrapping the data in a SYCL buffer if the computation is offloaded to an accelerator. Other modes exist that allow the programmer to explicitly control host-device data transfer.

In addition to parallel algorithm implementations in SYCL, oneDPL is supporting essential extensions for device programming such as custom iterators. To ensure interoperability across different platforms such extensions were added to the oneDPL specification.[4]

What's Next?

A Look into the Crystal Ball

Let's focus on some powerful, experimental oneDPL features that are currently under development but have not been fully baked into ISO C++, and how to get access to them:

- C++20 introduces Ranges that can greatly improve expressiveness when using C++ STL algorithms. They extend the utility of algorithms by supporting more complex data access patterns with Views. All this with fewer lines of code. As of today, ISO C++ Ranges algorithms are not supporting execution policies, which means it lacks accelerator support. oneDPL enables Ranges for selected algorithms and provides extensions, such as custom SYCL views, to enable device programming.[7]

- Classic C++ algorithms are well defined, including the blocking behavior of their function calls. However, blocking the host processor is not always desirable when offloading computation to an accelerator. To allow interleaving of host-device execution and data transfer, a set of asynchronous algorithms have been added to oneDPL. Their functionality is similar to C++ algorithms, but without the blocking behavior. To control nonblocking behavior, a C++ future-like object is returned instead of the result directly.[8]

There's more to come. Other exciting features like automatic device selection[4] are planned for future release, so stay tuned and follow us on GitHub.

Final Thoughts

oneDPL provides C++ building blocks that combine high performance with high productivity across CPUs, GPUs, FPGAs, and other accelerators. It is based on open standards, and its specification ensures interoperability across different platforms. Intel's reference implementation is a permissibly licensed open-source project.[5]

References

1. Parallel Research Kernels
2. Analyzing Reduction Abstraction Capabilities
3. SYCL Programming Language
4. oneDPL Specification
5. oneAPI DPC++ Library
6. How to Boost Performance with Intel Parallel STL and C++17 Parallel Algorithms
7. oneDPL Range-based API Algorithms
8. oneDPL Asynchronous API Algorithms

Learn more about programming with oneAPI and oneDPL:

- Reduce Cross-Platform Programming Efforts and Achieve High-Performance Parallel Code with oneDPL
- Intel oneAPI Base Training Modules

intel software

THE PARALLEL UNIVERSE

Printed in USA 707/IH Please Recycle.