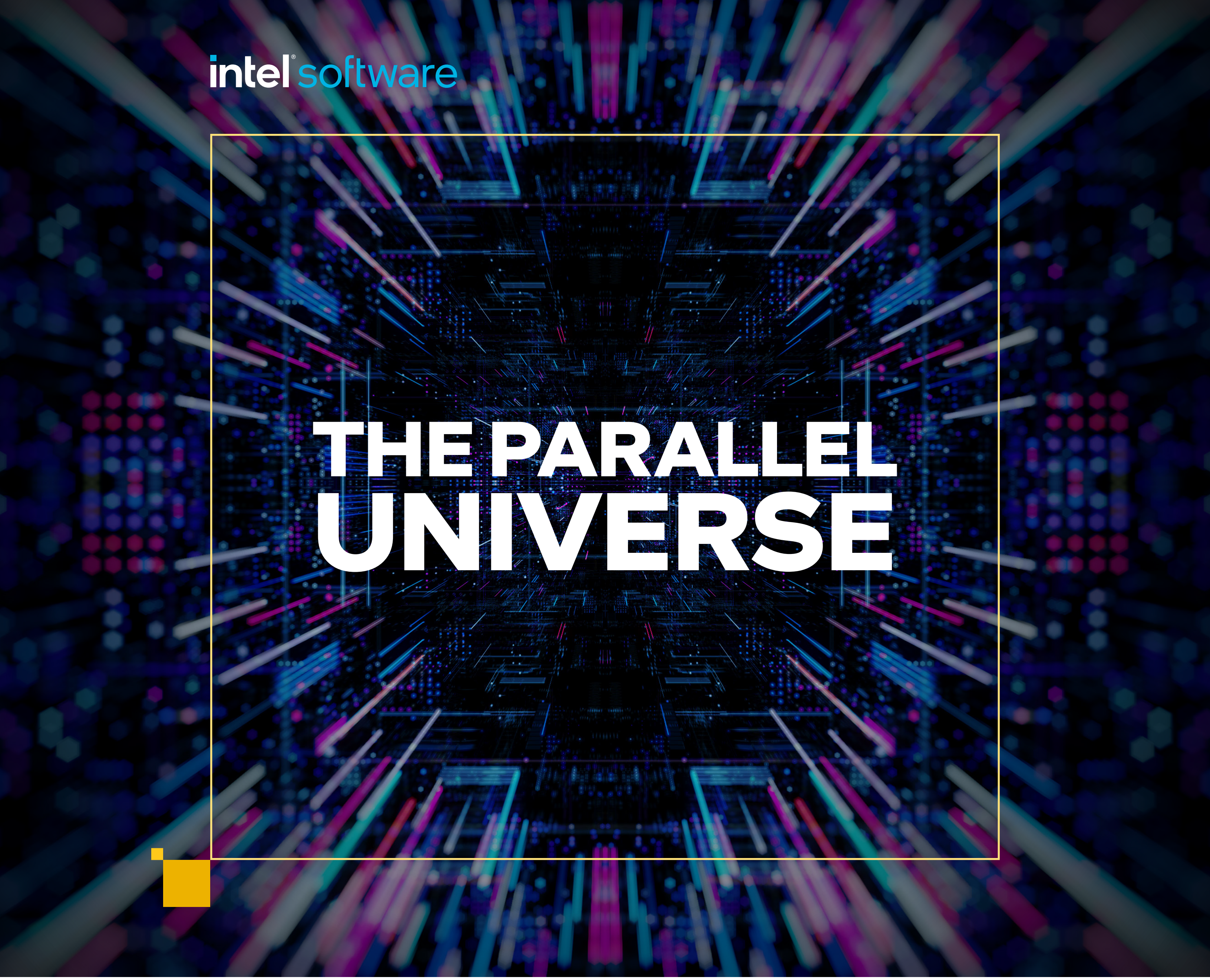
intel software



THE PARALLEL UNIVERSE

Beyond Direct Memory Access: Reducing the Data Center Tax with Intel® Data Streaming Accelerator

Cultivating Parallel Standards: Diversity, Alignment, and Cross-Pollination

Issue
53
2023

Contents

Letter from the Editor 3

FEATURE

Beyond Direct Memory Access: Reducing the Data Center Tax with Intel® Data Streaming Accelerator 6
Take Advantage of On-Chip Acceleration of Data Transformation in 4th Gen Intel® Xeon® Scalable Processors

Cultivating Parallel Standards: Diversity, Alignment, and Cross-Pollination 15
A Look into the Future of Standards-Based Parallel Programming

The Moat Is Trust, Or Maybe Just Responsible AI 19
Minimizing the Risks of Generative AI

Create Your Own Custom Chatbot 22
Train Large Language Models Quickly and Easily on Intel® Processors

Fine-Tuning the Falcon 7-Billion Parameter Model with Hugging Face and oneAPI 28
Optimizing Large Language Models on Intel® Xeon® Processors with Intel® Advanced Matrix Extensions (Intel® AMX)

Using Fortran DO CONCURRENT for Accelerator Offload 37
Limitations of Standard Fortran for Heterogeneous Computing

Performance Optimization on Intel® Processors with High Bandwidth Memory 42
A Deep Dive into Performance Tuning for the Intel® Xeon® CPU Max Series

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

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.

Openness, Sustainability, and the Cambrian Explosion of Generative AI Models

Since our last issue, two news items have caused a stir in the AI community. The first was a fireside chat between Intel and Hugging Face luminaries about Taking on the Compute and Sustainability Challenges of Generative AI, which among other things discusses how the democratization of generative AI has led to an explosion of pretrained models on The Model Hub. The second is the leaked internal Google memo, We Have No Moat, And Neither Does OpenAI, which laments how smaller, faster, cheaper, and more customizable open-source AI will outcompete closed models.

We have three articles along these lines in this issue. The first is a guest editorial from Huma Abidi (General Manager and Senior Director of AI Software Products) and Haihao Shen (AI Software Architect) that directly addresses the figurative "moat" around AI: **The Moat Is Trust, Or Maybe Just Responsible AI**. The second, **Create Your Own Custom Chatbot**, demonstrates how to use open models and readily available hardware to build customized, high-performing chatbots. Finally, **Fine-Tuning the Falcon 7-Billion Parameter Model with Hugging Face and oneAPI** shows how to optimize another open-source large language model on Intel® Xeon® processors with Intel® Advanced Matrix Extensions (Intel® AMX).

Our feature article, **Beyond Direct Memory Access: Reducing the Data Center Tax with Intel® Data Streaming Accelerator**, provides code examples and advice to take advantage of on-chip acceleration for data transformation. The Intel Data Streaming Accelerator is new in 4th Gen Intel Xeon Scalable processors.

The Case for SYCL* (*The Parallel Universe*, Issue 51) discussed the limitations of ISO C++ with respect to heterogeneous computing. In the second guest editorial of this issue, **Cultivating Parallel Standards**, John Pennycook (Software Enabling and Optimization Architect) describes the process of aligning SYCL with future C++ language concepts.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

During my first 15 years in computational science, if I needed to TRANslate a FORmula into code, I did it almost exclusively in FORTRAN. So, I read the recent report from Los Alamos National Laboratory with great interest: An Evaluation of Risks Associated with Relying on Fortran for Mission Critical Codes for the Next 15 Years. Support for heterogeneous parallelism is one of the risk factors, so Ron Green (Compiler Engineering Manager and fellow Fortran enthusiast) and I take a look at **Using Fortran DO CONCURRENT for Accelerator Offload**. This article focuses on language features, but we're working on a follow-up article that will analyze DO CONCURRENT performance on CPUs and GPUs. Stay tuned.

Finally, we close this issue with a deep dive into performance tuning on the Intel Xeon CPU Max Series: **Performance Optimization on Intel® Processors with High Bandwidth Memory**.

As always, don't forget to check out Tech.Decoded 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.

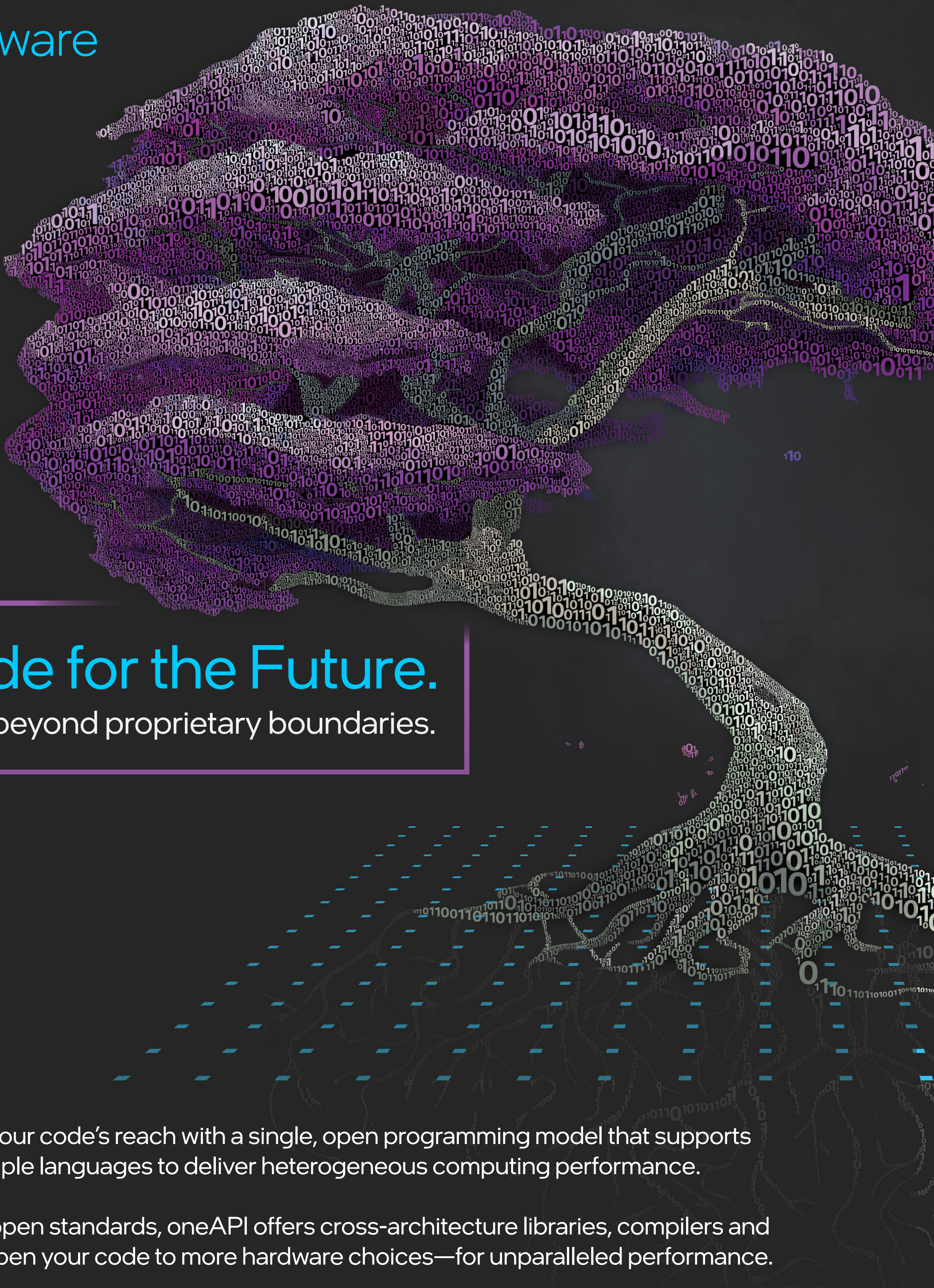
Henry A. Gabb

July 2023

PODCAST
How Generative AI is Changing How We Develop, Work and Live
LISTEN NOW

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

intel. software



Code for the Future.

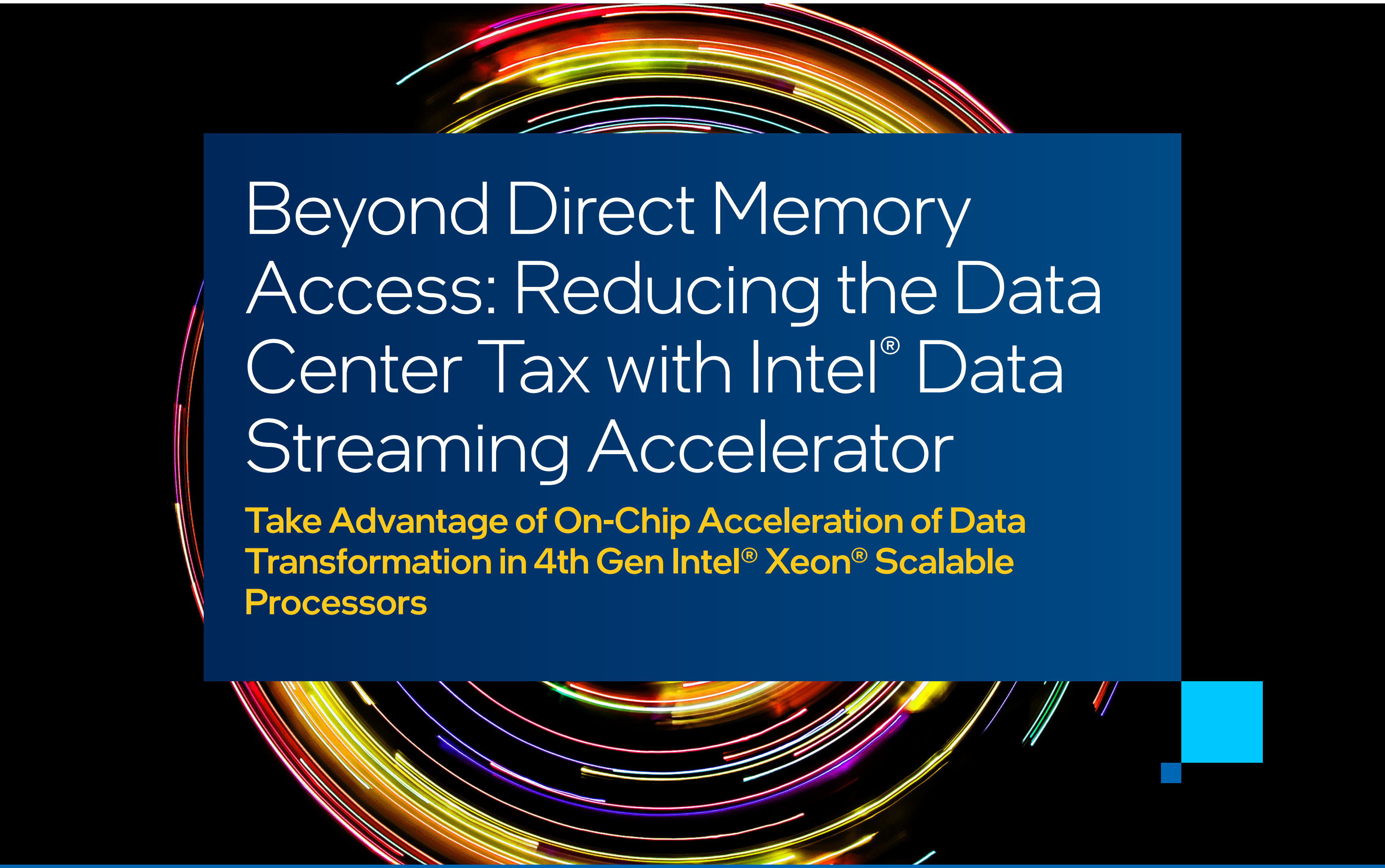
Grow beyond proprietary boundaries.

oneAPI

Expand your code's reach with a single, open programming model that supports multiple languages to deliver heterogeneous computing performance.

Rooted in open standards, oneAPI offers cross-architecture libraries, compilers and tools that open your code to more hardware choices—for unparalleled performance.

Discover oneAPI →



Beyond Direct Memory Access: Reducing the Data Center Tax with Intel® Data Streaming Accelerator

Take Advantage of On-Chip Acceleration of Data Transformation in 4th Gen Intel® Xeon® Scalable Processors

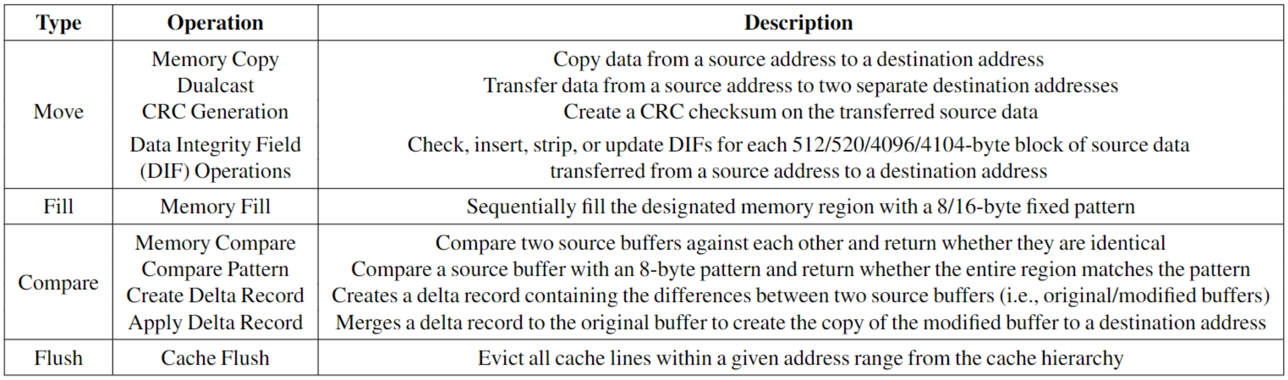
Sanjay Kumar, Principal Engineer, Reese Kuper, Research Intern, Atul Kwatra, Intel Fellow, Shibani Nataraja, Product Manager, Narayan Ranganathan, Principal Engineer, Nikhil Rao, Research Scientist, Rajesh Sankaran, Intel Senior Fellow, Ren Wang, Research Scientist, and Yifan Yuan, Research Scientist, Intel Corporation

In modern data centers, we have seen a wide range of applications involving intensive memory operations and transformations, such as `memcpy()`, `memmove()`, hashing, and compression. These applications include, but are not limited to, database, image processing and video transport, and graph processing. In addition, such memory operations and transformations are also common in a data center's infrastructure software, consuming a significant amount of CPU cycles. This is also known as the "data center tax" because these cycles could have been used to run applications. Ideally, such operations should be offloaded to optimized hardware engines.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Intel® Data Streaming Accelerator

Intel Data Streaming Accelerator (Intel® DSA) is a high-performance data copy and transformation accelerator integrated in the latest 4th Generation Intel® Xeon® Scalable processors. It provides not only processing efficiency and practicality, but also versatility. Intel DSA is equipped with hardware components to efficiently process work descriptors submitted by one or more users. Through the support of shared virtual memory (SVM), these work descriptors can be submitted by user applications directly to Intel DSA via memory-mapped I/O (MMIO) registers, and the target memory regions are not required to be pinned in the operating system. Users are also allowed to directly configure the underlying computational resources based on their needs, which is enabled by the user interfaces provided by a set of dedicated drivers. In addition, Intel DSA provides better functionality by means of newly supported operations (**Table 1**). Therefore, it is expected that more user/kernel processes can take advantage of what Intel DSA offers.



Type	Operation	Description
Move	Memory Copy	Copy data from a source address to a destination address
	Dualcast	Transfer data from a source address to two separate destination addresses
	CRC Generation	Create a CRC checksum on the transferred source data
	Data Integrity Field (DIF) Operations	Check, insert, strip, or update DIFs for each 512/520/4096/4104-byte block of source data transferred from a source address to a destination address
Fill	Memory Fill	Sequentially fill the designated memory region with a 8/16-byte fixed pattern
Compare	Memory Compare	Compare two source buffers against each other and return whether they are identical
	Compare Pattern	Compare a source buffer with an 8-byte pattern and return whether the entire region matches the pattern
	Create Delta Record	Creates a delta record containing the differences between two source buffers (i.e., original/modified buffers)
	Apply Delta Record	Merges a delta record to the original buffer to create the copy of the modified buffer to a destination address
Flush	Cache Flush	Evict all cache lines within a given address range from the cache hierarchy

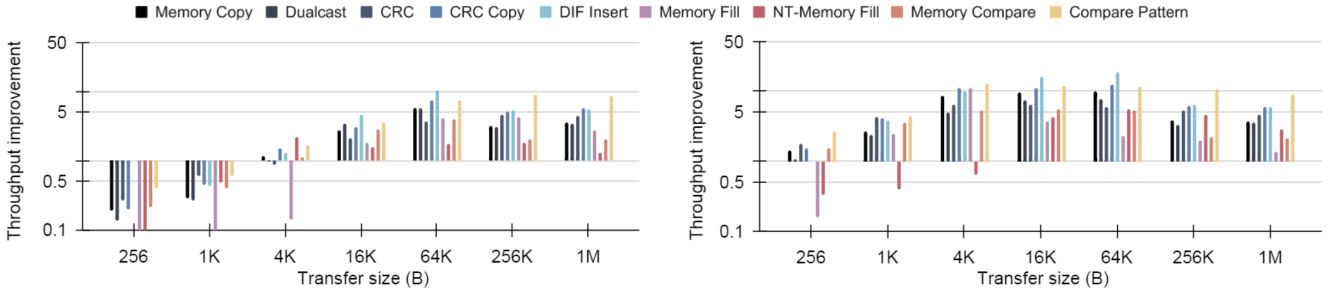
Table 1. Data streaming operations supported by Intel® DSA

Intel® Data Accelerator Driver (IDXD) is a kernel mode driver for device initialization and management. It provides functionality for Intel DSA discovery, initialization, and configuration. Applications can leverage the user-space `libaccel-config API` library for such control operations. For the data path, to provide low-latency access to the Intel DSA instance for applications, IDXD exposes the MMIO portals as a char device via `mmap()`. Given all these new features and improvements, we see significant performance gains of Intel DSA over its software counterparts in multiple domains (**Table 2**) and on multiple operations, especially with medium-to-large data size (**Figure 1**).

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation
or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Area	Example Applications and Operations
Networking stack acceleration	DPDK, software virtual switch, video transport
Storage stack acceleration	SPDK, NVMe-oF, DAOS
Data center tax reduction	VM boot-up/migration, memory compaction
HPC/ML acceleration	Memory movement/zeroing in libfabric/MPI
Heterogeneous memory management acceleration	Page migration for CXL/Pmem-based tiered memory systems

Table 2. Potential Intel® DSA usage in multiple domains



(a) Synchronous offloading **(b)** Asynchronous offloading

Figure 1. Throughput improvements of data streaming operations over their software counterparts with varying transfer sizes (batch size: 1); `Memory Fill` and `NT-Memory Fill` refer to allocating and non-allocating writes (similar to regular `store` and `nt-store`), respectively

Operating Intel DSA

Like other accelerators, Intel DSA employs descriptors and work queues (WQ) to interact with the CPU software (**Figure 2**).

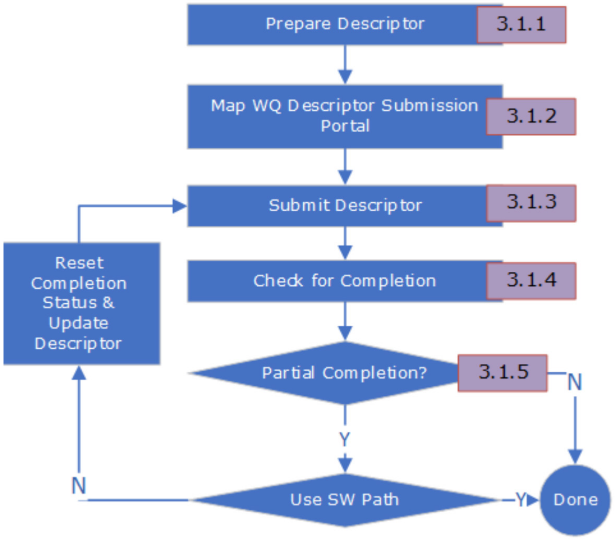
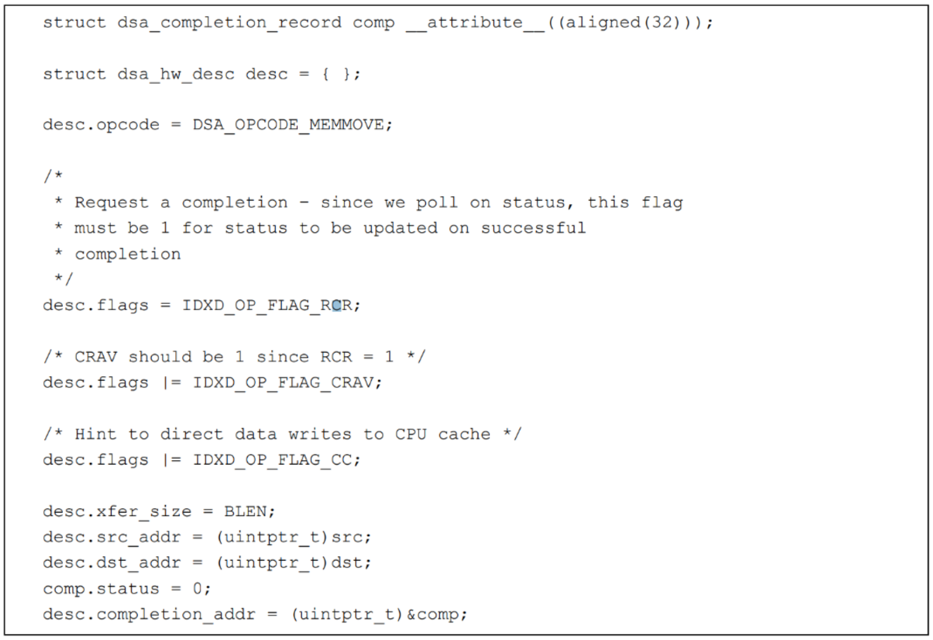


Figure 2. Descriptor processing sequence

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

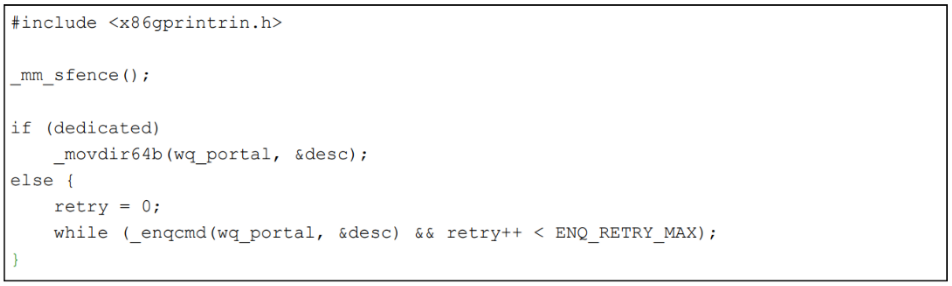
Data movement is the most common use-case of Intel DSA, so we will use a data-copy example to demonstrate usage. Suppose in a key-value store application, where we have large key-value pair size, every time we retrieve or update the value of a given key, memory copy will happen and incur lots of CPU cycles. We would rather offload these operations to Intel DSA. Assuming the Intel DSA instance has been enumerated and configured by tools like `accel-config`, the programmer first needs to allocate and fill the descriptor data structure (**Figure 3**). The descriptor contains information about the desired operation, such as operation type, source and destination address, data length, etc.



```
struct dsa_completion_record comp __attribute__((aligned(32)));

struct dsa_hw_desc desc = { };

desc.opcode = DSA_OPCODE_MEMMOVE;

/*
 * Request a completion - since we poll on status, this flag
 * must be 1 for status to be updated on successful
 * completion
 */
desc.flags = IDXD_OP_FLAG_RCR;

/* CRAV should be 1 since RCR = 1 */
desc.flags |= IDXD_OP_FLAG_CRAV;

/* Hint to direct data writes to CPU cache */
desc.flags |= IDXD_OP_FLAG_CC;

desc.xfer_size = BLEN;
desc.src_addr = (uintptr_t)src;
desc.dst_addr = (uintptr_t)dst;
comp.status = 0;
desc.completion_addr = (uintptr_t)&comp;
```

Figure 3. Descriptor initialization

Having the descriptor ready, the next step for the programmer is to submit the descriptor to a WQ opened and mapped in the current program. Depending on the WQ type, the software may use either the ENQCMD or MOVDIR64B instruction for descriptor submission. Both instructions are supported in GNU GCC (version 10 and later), and the programmer can use the x86 intrinsics to call them (**Figure 4**).



```
#include <x86gprintrin.h>

_mm_sfence();

if (dedicated)
    _movdir64b(wq_portal, &desc);
else {
    retry = 0;
    while (_enqcmd(wq_portal, &desc) && retry++ < ENQ_RETRY_MAX);
}
```

Figure 4. Descriptor submission

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

To check the completion of the descriptor, the programmer needs to poll the status field of the completion record, which will be updated by Intel DSA. The most common way of doing this is spin-polling (**Figure 5**). New instructions, like PAUSE and UMONITOR/UMWAIT, can be applied to further reduce the processor's power consumption, as the core can enter a different power state with such instructions. (See the Intel® 64 and IA-32 Architectures Software Developer Manuals for more information.)

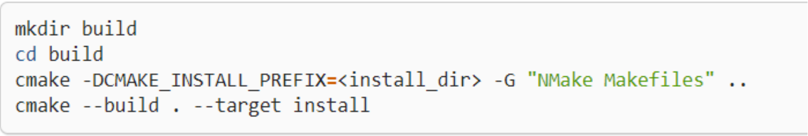
```
retry = 0;
while (comp.status == 0 && retry++ < COMP_RETRY_MAX);

if (comp.status == DSA_COMP_SUCCESS) {
    /* Successful completion */
} else {
    /* Descriptor failed or timed out
     * See the "Error Codes" section of the Intel® DSA Architecture Specification for
     * error code descriptions
     */
}
```

Figure 5. Checking descriptor completion

Take Advantage of Intel DSA with Software Libraries

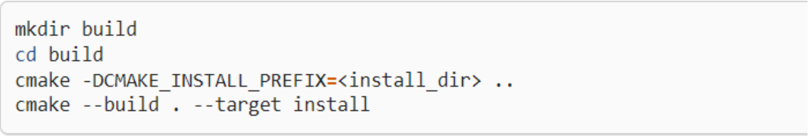
The Intel® Data Mover Library (Intel® DML) and Intel® DSA Transparent Offload Library (Intel® DTO) make Intel DSA easier to use. Intel DML provides a set of high-level C/C++ APIs for data movement and transformation, calling the underlying Intel DSA unit when available. It supports advanced capabilities (all Intel DSA hardware operations, asynchronous offload, load balancing, etc.). Intel DML v1.0.0 is available on GitHub. After cloning the repository and installing the required packages, it can be compiled and installed by simple CMake commands (see the Intel DML Documentation for more installation options):

Windows* OS:



```
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=<install_dir> -G "NMake Makefiles" ..
cmake --build . --target install
```

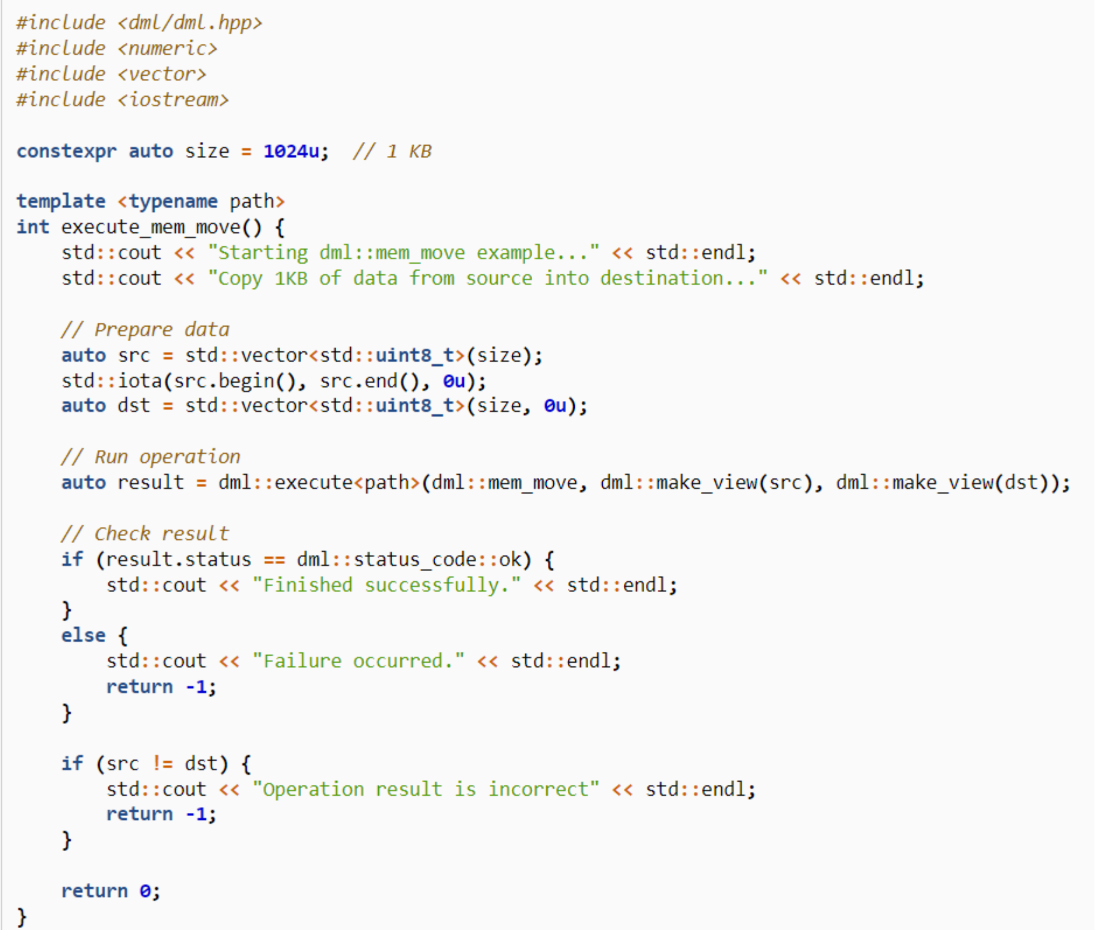
Linux* OS:



```
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=<install_dir> ..
cmake --build . --target install
```

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation
or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Figure 6 demonstrates basic usage of Intel DML. While the basic steps remain unchanged, more details are hidden by Intel DML APIs and abstraction. Continuing the example of key-value store application, instead of manually preparing and submitting the descriptor, the programmer can call Intel DML so that the designated memory copy is offloaded to Intel DSA.



```
#include <dml/dml.hpp>
#include <numeric>
#include <vector>
#include <iostream>

constexpr auto size = 1024u;  // 1 KB

template <typename path>
int execute_mem_move() {
    std::cout << "Starting dml::mem_move example..." << std::endl;
    std::cout << "Copy 1KB of data from source into destination..." << std::endl;

    // Prepare data
    auto src = std::vector<std::uint8_t>(size);
    std::iota(src.begin(), src.end(), 0u);
    auto dst = std::vector<std::uint8_t>(size, 0u);

    // Run operation
    auto result = dml::execute<path>(dml::mem_move, dml::make_view(src), dml::make_view(dst));

    // Check result
    if (result.status == dml::status_code::ok) {
        std::cout << "Finished successfully." << std::endl;
    }
    else {
        std::cout << "Failure occurred." << std::endl;
        return -1;
    }

    if (src != dst) {
        std::cout << "Operation result is incorrect" << std::endl;
        return -1;
    }

    return 0;
}
```

Figure 6. Example of basic data movement with Intel® DML

Intel DTO is a less-intrusive library that allows the application to leverage Intel DSA transparently without source code modification. Users can either dynamically link the library using the `-ldto` and `-laccel-config` linker options or preload the library via `LD_PRELOAD` without having to recompile their application. When common system API calls like `memmove()`, `memcpy()`, `memset()`, or `memcmp()` are used, they are intercepted and replaced by Intel DTO functions to access the corresponding synchronous Intel DSA operations.

In addition to Intel DML and Intel DTO, which are general-purpose libraries, there are also domain-specific software libraries that take advantage of Intel DSA. For example, in the Data Plane Development Kit (DPDK) for high-performance kernel-bypass network programming, Intel DSA, along with DMA

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. *Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

engines from other vendors, has been abstracted and wrapped in the **dmadev** library. Users can call the corresponding APIs to offload network packets copying to Intel DSA instances. Similarly, the Storage Performance Development Kit (SPDK) has Intel DSA support in its /lib/accel library to support various operations offloading. Intel DSA has also been enabled in HPC libraries, such as libfabric and MPI, which are used in not only traditional scientific computing workloads, but also emerging distributed ML/AI applications.

Make the Most Out of Intel DSA

As an on-chip accelerator for streaming data, Intel DSA benefits from proper tuning of the programming models for optimal performance. Based on our experiences, we summarize a number of guidelines and recommendations for using Intel DSA. (See this technical report for more details.)

Keep a Balanced Batch Size and Transfer Size

Offloading work of a certain size to Intel DSA can be done using either one descriptor for the full memory region or batching multiple smaller descriptors for the same aggregate size. The general trend indicates a decrease in throughput when using larger batches for the same offloaded work. As individual descriptors must be processed internally in Intel DSA and read the corresponding data from memory, the additional overhead for managing the increased number of descriptors may reduce the effective throughput achieved from these operations. If the desired data for offloading is contiguous, coalescing into a larger, single descriptor of the equivalent size may improve both throughput and latency.

When offloading synchronously, a weak pattern emerges, showing an optimum point in throughput between maximizing batches and maximizing transfer size. Modestly batching the work (4-8 descriptors) yields the best results. This is due to balancing the latency from fetching sequential regions of memory and processing batched descriptors.

Use Intel DSA Asynchronously When Possible

Offloading operations to Intel DSA in an asynchronous manner provides optimal efficiency and performance for both the CPU core and Intel DSA hardware. This can use Intel DML for easy implementation. When limited in asynchronous potential, transfer sizes below 4 KB should be used on the CPU core if cache pollution is acceptable. As an on-chip accelerator, Intel DSA brings more options to interact with the cache and (heterogeneous) memory hierarchy.

Control the Data Destination Wisely

Unlike the completion record that is always directed to the LLC, data written to the destination address of a descriptor can be steered either to the LLC or to the main memory. Intel DSA facilitates this cache control feature by allowing users to provide a hint (i.e., setting the cache control flag of a work descriptor) that notes the preferred destination. If the flag is set to 0, the data is written to the memory while

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation
or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

invalidating the corresponding cache lines in the LLC, if any. If the flag is set to 1, the data is directly written to the LLC by allocating the corresponding cache lines. The underlying principle of this technique is identical to that of Intel® Data Direct I/O Technology (Intel® DDIO), a direct cache access (DCA) scheme leveraging the LLC as the intermediate buffer between the processor and I/O devices. In general, DDIO improves system performance by reducing the data access latency and reducing memory bandwidth pressure. However, one should be careful when using this feature because it can sometimes cause interference in the LLC.

Cache pollution negatively affects processes that share limited hardware resources. In many data center workloads, the latency gained from antagonistic background cache evictions may undermine the competitive service level agreements (SLAs) set for the primary applications. On the other hand, writing data that is either critical to performance or used by the core in the near future directly to the cache will provide an access latency and throughput advantage for the application. The programmer should wisely choose the data destination based on the application behavior.

Intel DSA as a Good Candidate of Moving Data across a Heterogeneous Memory System

Moving data to/from different memory mediums, such as NUMA remote memory, persistent memory, and CXL-based memory, is common in modern tiered memory systems. Intel DSA offers a high degree of configuration flexibility through the available hardware resources. Taking advantage of such configurations can yield optimal Intel DSA hardware utilization, and thus better performance.

Leverage Processing Engine-Level Parallelism

The number of processing engines used in a group can impact the maximum observed throughput. Increasing the number of processing engines per group improves throughput. Users should be aware of the common transfer size of offloaded tasks to these groups as smaller transfer sizes yield greater performance scaling.

Optimize WQ Configuration

Using batching or dedicated WQs (DWQs) provides greater benefits compared to a shared WQ (SWQ). Unless the shared WQ is used by many other threads, greater utilization and throughput can be achieved through reconfiguring to using more DWQs. SWQs may perform worse when few threads are used, but can outperform all configurations when using more threads than the total number of WQs, as that offloads concurrency management to hardware. Additionally, WQs can be configured to either shared or dedicated for providing performance isolation between WQs within a group. As Intel DSA has limited WQ entries, assigning 32 entries for a single WQ can provide almost the maximum throughput possible.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation
or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Conclusion

Intel DSA, as well as other on-chip accelerators appearing in the 4th Generation Intel Xeon Scalable processors, has the potential to reduce the data center tax and total cost of ownership. This article provides a high-level overview of Intel DSA and its fundamental usages, as well as several guidelines to make the most out of this accelerator. With a growing software ecosystem, we believe Intel DSA will be adopted for many data center workloads. Intel, in active collaboration with the open-source community, has been building and enabling this ecosystem.

Additional Resources

- Introducing the Intel Data Streaming Accelerator (Intel DSA)
- Intel Data Streaming Accelerator Architecture Specification [PDF]
- Intel Data Streaming Accelerator User Guide [PDF]
- Intel DSA Performance Micros
- Intel Data Mover Library (Intel DML)

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Cultivating Parallel Standards: Diversity, Alignment, and Cross-Pollination

A Look into the Future of Standards-Based Parallel Programming

John Pennycook, Software Enabling and Optimization Architect, Intel Corporation

oneAPI is an open specification (see oneapi.io) for accelerator programming that includes a language (SYCL* from The Khronos Group), standardized library interfaces (for neural networks, data analytics, and more), and a close-to-the-metal programming interface (Level Zero). oneAPI is designed to be hardware- and vendor-independent, and has already been demonstrated on CPUs, GPUs, FPGAs, and other accelerators from multiple vendors. oneAPI delivers a standards-based approach to programming that allows developers to unlock high performance without sacrificing portability across hardware and software environments.

oneAPI is made possible by a collection of multiple *interoperable* standards across many levels of the software stack. Some of these standards (like Level Zero and extensions to SYCL) are part of the oneAPI specification. Others (like SYCL and SPIR-V) are industry standards that provide the foundations for portable programming with oneAPI. Yet others (like OpenMP*, ISO C++, and ISO Fortran) describe

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation
or its subsidiaries. *Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

alternative syntax for accelerator programming that can be layered on top of oneAPI.

Interoperability between all these standards is crucial, because large applications are very rarely written by a single developer working with a single language at a single level of abstraction. It's much more common to use a mixture of different techniques, calling libraries written by other developers (e.g., an application written in Fortran and OpenMP using an optimized library written in SYCL). Maintaining interoperability requires constant effort, because it's not uncommon for different standards to offer their own solutions to new problems. Different standards may target different levels of abstraction, may prefer different coding styles, and must always consider how any new functionality meshes with existing features and established conventions. But standards can still learn from one another, aligning around terminology and cross-pollinating features when appropriate.

Looking back at the last decade of SYCL, C++, and OpenCL* we can clearly see this process of alignment and cross-pollination in action (**Figure 1**). For example, OpenCL 2.0 adopted the concepts and terminology from the C++11 memory model, extending them with the notion of scopes and multiple address spaces. OpenCL 2.0 features like groups and group functions were then later combined with aspects of the C++17 parallel algorithms to produce SYCL 2020's group algorithms library, allowing developers to use familiar C++ syntax to access vendor-optimized operations at multiple levels of the hardware hierarchy.

This is an ongoing process, and in our recent IWOCL submission — "Towards Alignment of Parallelism in SYCL and ISO C++" — we proposed some new clarifications to SYCL that are designed to bridge the gap between concepts like SYCL *work-items* and C++17 *threads of execution*. These clarifications make it easier to reason about the behavior of C++17 parallel algorithms layered on top of SYCL and are a necessary step towards allowing certain C++17 execution policies to work correctly on SYCL devices.

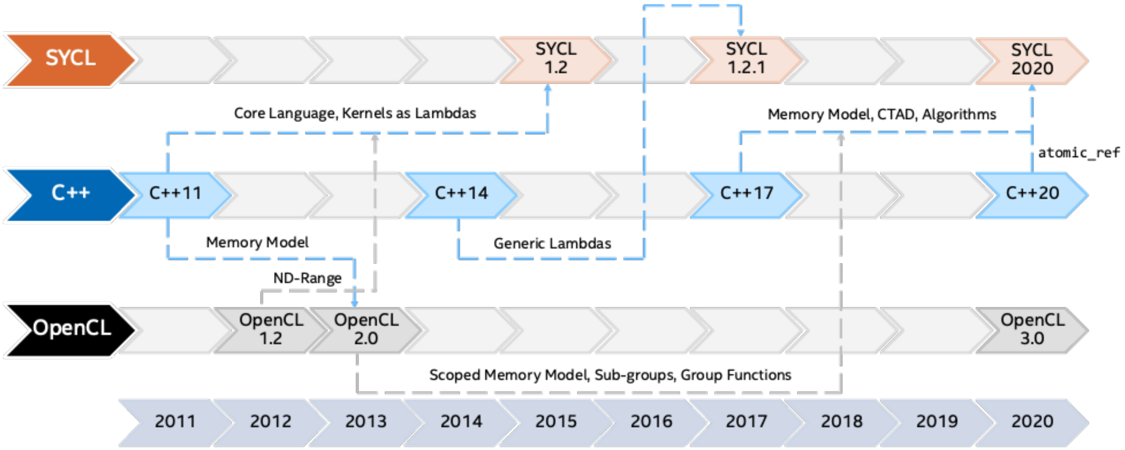


Figure 1. Parallel evolution of SYCL*, ISO C++ and OpenCL*

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Looking forward, we see opportunities for SYCL to influence the future of heterogeneous programming in C++. We believe that implementers' experiences with SYCL can and will feed directly into the design of new, high-level abstractions proposed for C++. But we also believe that SYCL will continue to play an important role for developers long into the future, providing direct access to lower-level and cutting-edge hardware features via mechanisms that are fully interoperable with relevant C++ abstractions.

What might that future look like? Although we can't say for certain, there are a few ISO C++ proposals that hold some clues: P2300 ("`std::execution`") and P2500 ("C++ parallel algorithms and P2300"). If these proposals are accepted — which won't happen until 2026, at the earliest! — then it would become possible to write code like that in **Figure 2**.

```
// Use SYCL to represent a specific "scheduler"
// NB: get_scheduler() doesn't exist (yet)!
auto q = sycl::queue{sycl::gpu_selector_v};
scheduler auto sch = q.get_scheduler();

// Submit a parallel loop to the SYCL scheduler using P2300/P2500 features
std::for_each(std::execute_on(sch, std::execution::par_unseq), begin, end,
  [=](auto i) {
  ...
});
```

Figure 2. A glimpse into a future that mixes C++ parallel algorithms and SYCL* with features proposed in P2300/P2500

For those of us who can't wait until 2026, it's already possible to mix C++ parallel algorithms and SYCL today using the oneAPI DPC++ Library (oneDPL). oneDPL is an open-source library that enables C++ parallel algorithms to execute on SYCL devices, providing a high-productivity solution to developing applications that can execute anywhere SYCL can. oneDPL also serves as a vehicle to explore possible future extensions to ISO C++—there are already efforts underway to standardize parallel range-based algorithms based on C++20 ranges, and to explore ways to represent asynchronous parallel algorithms. Submitting a C++ parallel algorithm to a SYCL device using oneDPL is straightforward and can be as simple as using a SYCL-aware execution policy (**Figure 3**). Using experimental features of oneDPL can unlock even higher levels of performance and productivity, by leveraging lazily evaluated views to enable kernel fusion (**Figure 4**).

```
// Submit a parallel loop to the default SYCL device using oneDPL
std::for_each(oneapi::dpl::execution::dpcpp_default, begin, end,
  [=](auto i) {
  ...
});
```

Figure 3. Mixing C++ parallel algorithms and SYCL* using features available in oneDPL today

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

```
// Submit three algorithms to the default SYCL device as three kernels
using namespace oneapi::dpl;
reverse(execution::dpcpp_default, begin(data), end(data));
transform(execution::dpcpp_default, begin(data), end(data), begin(result),
          [](auto i){ return i * i; });
auto res = find_if(execution::dpcpp_default, begin(result), end(result),
                   pred);

// Submit a pipeline of range transformations to the default SYCL device,
// which will execute as a single kernel
using namespace oneapi::dpl::experimental::ranges;
auto res = find_if(execution::dpcpp_default,
                   views::all(sycl::buffer{data})
                   | views::reverse
                   | views::transform([](auto i){return i * i;}), pred);
```

Figure 4. Mixing C++ ranges and SYCL* using experimental features of oneDPL

oneDPL's current syntax may be a little different to where things end up eventually, but that's simply the standardization process at work. Whatever ultimately lands in a future C++ standard will be the result of years of work, drawing on the experiences of multiple libraries and multiple hardware vendors, to establish common requirements and best practices.

We're often asked whether developers should write their programs in OpenMP, SYCL, ISO C++, or ISO Fortran. The answer is simple: Yes! These are all valid approaches to heterogeneous programming, each with unique advantages and disadvantages, and combining these approaches has never been easier thanks to features like Fortran's ISO_C_BINDING, OpenMP's interop clause, and SYCL's backend interoperability interface (**Figure 5**). The strong focus on interoperability between the standards at the heart of oneAPI enables us to embrace the ability to mix and match, using whichever combination of tools is best for the job at hand.

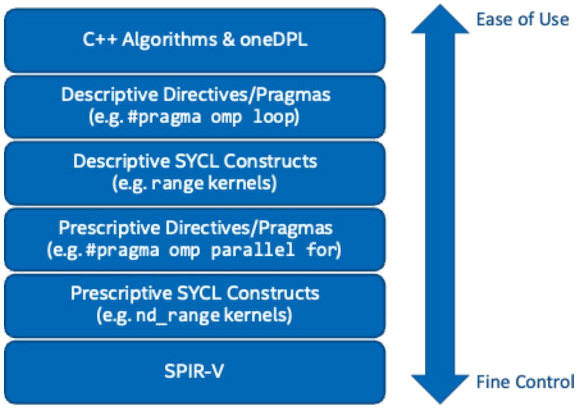

C++ Algorithms & oneDPL
Descriptive Directives/Pragmas (e.g. #pragma omp loop)
Descriptive SYCL Constructs (e.g. range kernels)
Prescriptive Directives/Pragmas (e.g. #pragma omp parallel for)
Prescriptive SYCL Constructs (e.g. nd_range kernels)
SPIR-V
Ease of Use
Fine Control

Figure 5. A comparison of approaches to heterogeneous programming: higher-level abstractions deliver a high-productivity solution to writing high-performance programs; lower-level abstractions provide direct control for expert developers

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

The Moat Is Trust, Or Maybe Just Responsible AI

Minimizing the Risks of Generative AI

Huma Abidi, General Manager and Senior Director of AI Software Products, and Haihao Shen, AI Software Architect, Intel Corporation

OpenAI's ChatGPT* created a frenzy in the generative AI landscape by achieving record-breaking growth. It attracted over one million users within its first week. While big tech companies like Google, Microsoft, and Facebook are competing in the Large Language Model (LLM) race, small startups are also making headway. A growing problem seems to be balancing the secrecy needed for competitive advantage with the transparency needed for safety. Unlike some LLMs, OpenAI has not released their training set or GPT-4* architectural details, which is drawing criticism from some quarters.

A recent leaked Google memo highlighted concerns about Google and OpenAI lacking a competitive advantage, or "moat," around LLM technology. The memo emphasized how open-source alternatives are smaller, faster, cheaper, and more customizable. (We demonstrate this in **Create Your Own Custom Chatbot** in this issue of *The Parallel Universe*.)

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

While LLMs like ChatGPT have demonstrated impressive capabilities, the emergence of generative AI models raises concerns about potential harmful effects. While the ongoing debates over large vs. small models and open vs. closed systems are important, we must recognize that performance and accuracy are not the only considerations. Factors such as fairness, explainability, sustainability, privacy, etc. must also be considered. Upholding responsible AI practices will ultimately determine the societal value of AI. Regulations will probably be introduced to require AI applications to comply with ethical best practices. The forthcoming European Union Artificial Intelligence Act, expected to be passed later this year, will mark the world's inaugural set of regulations governing AI systems. This legislation aims to foster a human-centric and ethical approach to AI by introducing guidelines pertaining to transparency and risk management.

Generative AI models learn from vast amounts of data available on the internet, which can contain biases present in society and may inadvertently perpetuate and amplify these biases. LLMs can be manipulated to generate or spread misinformation, phishing emails, or social engineering attacks. Malicious actors can intentionally train models with biased or false information, leading to the dissemination of misleading content on a large scale. Such models can be used to create convincing deepfake video and audio content. For example, a recent AI-generated hoax of an explosion at the Pentagon went viral. Such fake news is emerging as a major threat to the upcoming US elections: "It's going to be very difficult for voters to distinguish the real from the fake. And you could just imagine how either Trump supporters or Biden supporters could use this technology to make the opponent look bad," said Darrell West, a senior fellow at the Brookings Institution's Center for Technology Innovation.

LLMs can often have "hallucinations" and generate inaccurate information, which can be particularly problematic in industries like healthcare, where models can influence diagnostic and therapeutic decisions and potentially harm patients. Even though AI hallucinations are a known phenomenon, people continue using LLMs and uncritically accepting their pronouncements. In a recent example, it was discovered that a lawyer used ChatGPT to do his research because none of the decisions or quotations cited in his legal brief existed. They were made up by ChatGPT.

The need for responsible AI has never been greater. In an interview at the Commonwealth Club, Professor Stuart Russel said about ChatGPT: *"...in a sense, we are conducting a huge experiment on the human race with no informed consent whatsoever."* A group of over 1,000 AI experts, including Professor Russell and Elon Musk, has called for a pause in the deployment of LLMs. Lawmakers, industry leaders, and researchers agree that building guardrails around AI and strict regulations to ensure safe deployment of AI are a must. Industry leaders admit that AI technology might be an existential threat to humanity. A statement released by the Center for AI Safety says that *"mitigating the risk of extinction from AI should be a global priority alongside other societal-scale risks such as pandemics and nuclear war."* It was signed by some of the biggest names in AI: Geoffrey Hinton (Emeritus Professor, University of Toronto), Yoshua Bengio (Professor, University of Montreal), Sam Altman (CEO of OpenAI), Demis Hassabis (CEO of Google DeepMind), and Dario Amodei (CEO of Anthropic). To mitigate the risks of AI, responsible development,

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation
or its subsidiaries. *Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

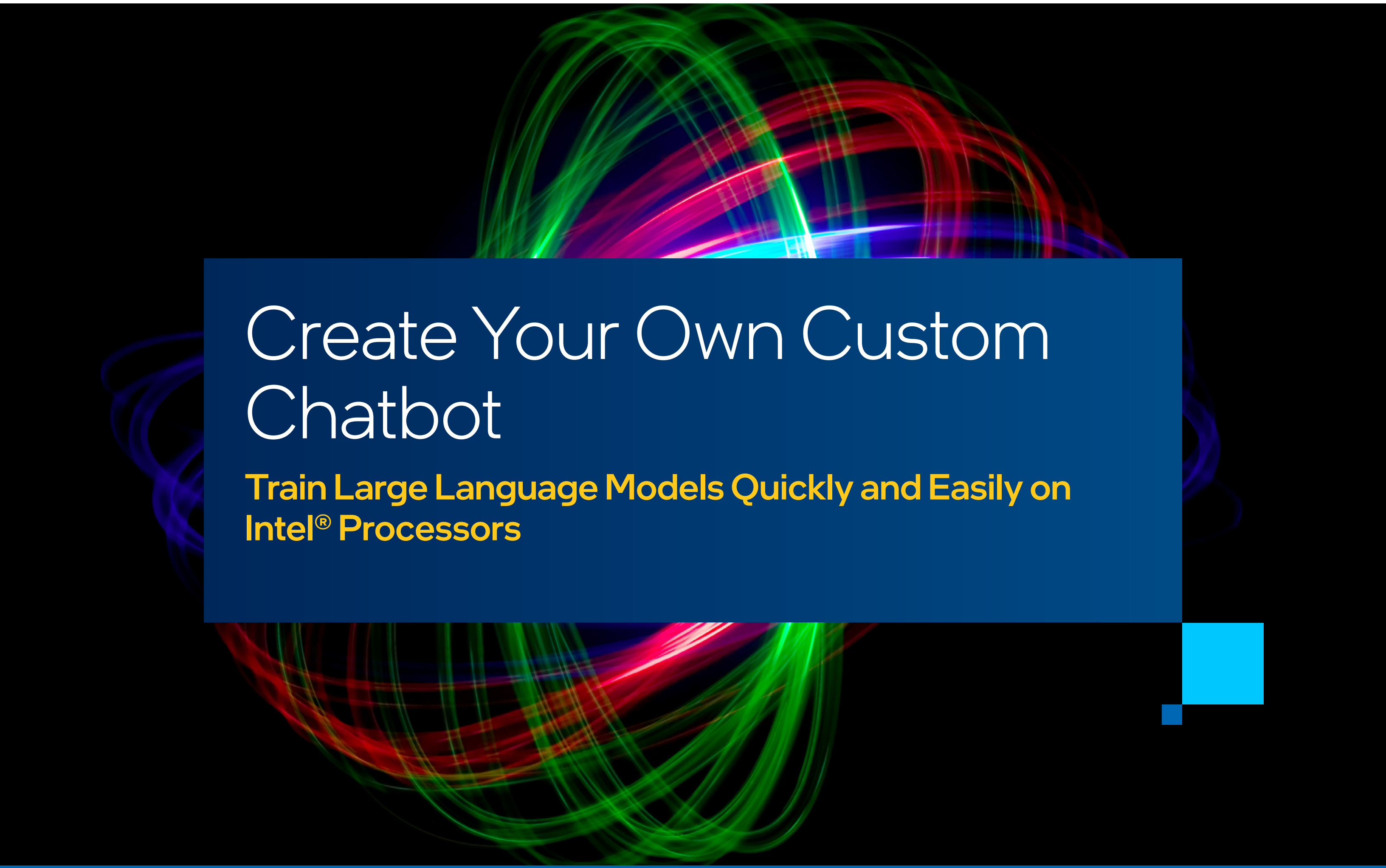
careful dataset curation, ongoing research, and robust ethical guidelines are essential. It is crucial to ensure transparency, accountability, and regular audits of LLMs to address biases, reduce misinformation, and protect user privacy.

It is evident that companies and individuals working on AI technology need to make sure their software is developed and deployed according to ethical AI principles. The open-source Intel® Explainable AI Tools allow users to run post hoc model distillation and visualization to examine the predictive behavior of both TensorFlow* and PyTorch* models. They are designed to help users detect and guard against issues of fairness and interpretability. For example, our model card generator is an open-source Python* module that allows users to create interactive HTML reports containing model details and quantitative analysis that displays performance and fairness metrics for both TensorFlow and PyTorch models. These model cards can be part of a traditional end-to-end platform for deploying ML pipelines for tabular, image, and text data to promote transparency, fairness, and accountability.

LLMs are typically trained on large public datasets and then fine-tuned on potentially sensitive data (e.g., financial and healthcare). Technologies like our Open Federated Learning (OpenFL) incorporate confidential computing so that LLMs can be safely fine-tuned on sensitive data, which in turn improves the generalizability of models while reducing hallucinations and bias.

AI has the potential to help in economically disadvantaged areas where there is a shortage of critical expertise. Presently, LLMs require tremendous computing power, and are typically executed in the cloud or expensive on-premises servers with multiple accelerators. We are focused on reducing the computational complexity of LLMs and making LLM-based inference more efficient so that advanced AI techniques will be available in areas with no cloud connectivity and on lower cost edge computing devices.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.



Create Your Own Custom Chatbot

Train Large Language Models Quickly and Easily on Intel® Processors

Haihao Shen, AI Software Architect, Xinyu Ye, AI Software Engineer, Kaokao Lv, AI Software Engineer, Xuhui Ren, AI Software Engineer, and Huma Abidi, General Manager and Senior Director of AI Software Products, Intel Corporation

Large language models (LLMs) have been attracting a lot of attention lately because of their extraordinary performance on dialog agents such as ChatGPT*, GPT-4*, and Bard*. However, LLMs are limited by the significant cost and time required to train or fine-tune them. This is due to their large model sizes and data sets.

In this article, we will demonstrate how to easily train and fine-tune a custom chatbot on readily available hardware. We use 4th Generation Intel® Xeon® Scalable processors to create our chatbot using a systematic methodology to generate a domain-specific dataset and an optimized fine-tuning code base.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Our Approach

Stanford Alpaca is an instruction-following language model that is fine-tuned from Meta's LLaMA model. Inspired by this project, we developed an enhanced methodology to create a custom, domain-specific chatbot. While there are several language models that one could use (including some with better performance), we selected Alpaca because it is an open model.

The workflow of the chatbot consists of four main steps: guided seed generation, free (non-guided) seed generation, sample generation, and fine-tuning (**Figure 1**).

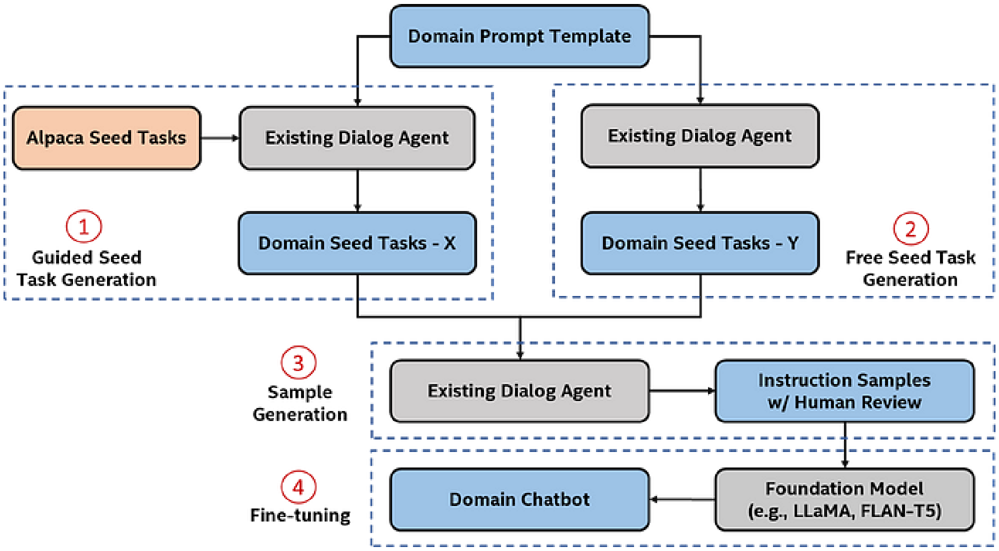
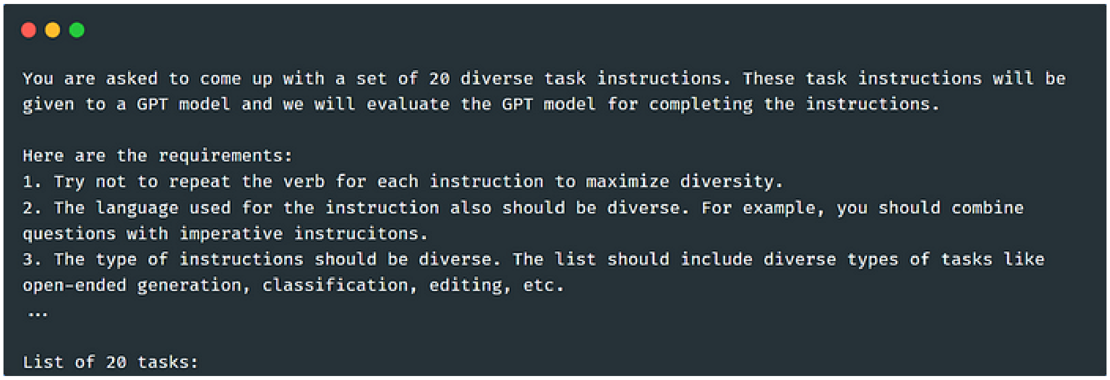


Figure 1. Overview of chatbot fine-tuning

Before walking you through these steps, we'd like to introduce a prompt template that is useful in seed task generation. The sample prompt from Alpaca for general tasks is shown in **Figure 2**.



```
You are asked to come up with a set of 20 diverse task instructions. These task instructions will be
given to a GPT model and we will evaluate the GPT model for completing the instructions.

Here are the requirements:
1. Try not to repeat the verb for each instruction to maximize diversity.
2. The language used for the instruction also should be diverse. For example, you should combine
questions with imperative instrucitons.
3. The type of instructions should be diverse. The list should include diverse types of tasks like
open-ended generation, classification, editing, etc.
 ...

List of 20 tasks:
```

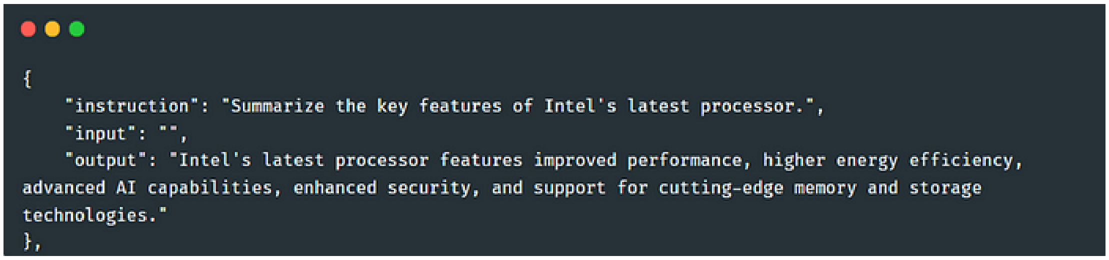
Figure 2. Prompt template for seed task generation

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

We modified the template by adding a new requirement; i.e.: *“The generated task instructions should be related to <domain_name> issues.”* This helps generate seed tasks related to the specified domain. To generate more diverse seed tasks, we use both guided and free (non-guided) seed task generation.

Guided seed task generation leverages the existing seed tasks from Alpaca. For each seed task, we combine the content from the domain prompt template and feed it into the existing dialog agent. We expect to generate the corresponding number of tasks (e.g., 20 defined in the prompt template in **Figure 2**). Such text generation is one of the typical use-cases for causal language models.

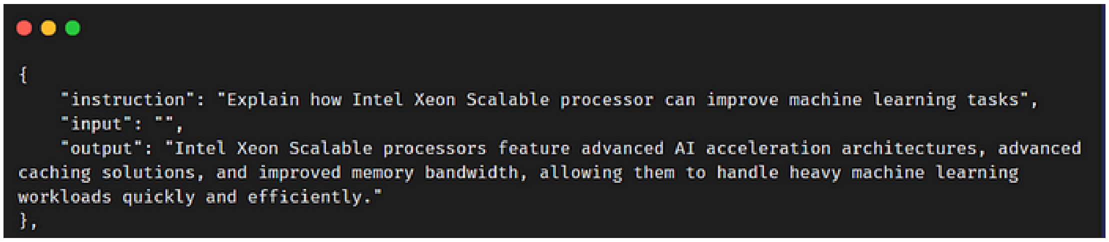
Non-guided seed task generation feeds the domain prompt template to the dialog agent directly without specifying additional seed tasks. We refer to non-guided seed task generation as “free.” We generate new domain seed tasks using this approach (**Figure 3**).



```
{
    "instruction": "Summarize the key features of Intel's latest processor.",
    "input": "",
    "output": "Intel's latest processor features improved performance, higher energy efficiency,
advanced AI capabilities, enhanced security, and support for cutting-edge memory and storage
technologies."
},
```

Figure 3. Domain seed task

With these seed tasks, we again leverage the existing dialog agent to generate the instruction samples. As the domain prompt template is used, the output follows the requirements with the format “instruction,” “input,” and “output.” We repeat the process and generate 2,000 instruction samples for fine-tuning (**Figure 4**).



```
{
    "instruction": "Explain how Intel Xeon Scalable processor can improve machine learning tasks",
    "input": "",
    "output": "Intel Xeon Scalable processors feature advanced AI acceleration architectures, advanced
caching solutions, and improved memory bandwidth, allowing them to handle heavy machine learning
workloads quickly and efficiently."
},
```

Figure 4. Domain instruction sample

You may have noticed the similarity between a domain seed task and an instruction sample. You can think of them as being a ChatGPT prompt and the resulting output respectively with one influencing the other.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Training Your Custom Chatbot

We use the Low-Rank Adaptation (LoRA) approach to fine-tune the LLM efficiently, rather than fine-tuning the entire LLM with billions of parameters. LoRA freezes the pretrained model weights and injects trainable rank decomposition matrices into each layer of the transformer architecture, greatly reducing the number of trainable parameters for downstream tasks.

Besides the parameter-efficient fine-tuning, we can leverage hardware and software acceleration to speed up the fine-tuning process. An example of hardware acceleration is Intel® Advanced Matrix Extensions (Intel® AMX-BF16) instructions, available on 4th Generation Intel Xeon Scalable processors, that are specifically designed to accelerate AI performance. The software optimizations included in PyTorch*, Intel® Extension for PyTorch* and Hugging Face* transformers also help accelerate performance compared to unoptimized versions of these frameworks and libraries.

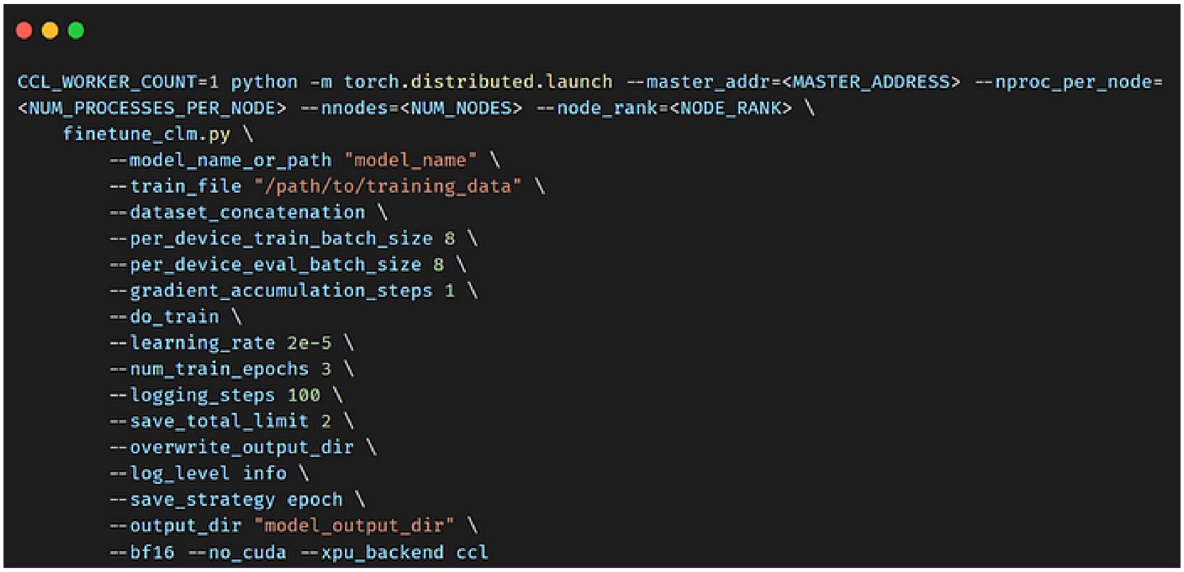
We can also enable instruction sample concatenation to further improve the fine-tuning process. The basic idea is that several tokenized sentences are concatenated into a longer and concentrated sentence as the training sample instead of having several training samples with different lengths. This helps maximize the underlying hardware efficiency.

All the above optimizations are on a single compute node. You can also perform multinode fine-tuning with distributed data parallelism during the fine-tuning process to harness more computing power.

Now, let's start training a domain chatbot:

- Check out the example code from Intel® Extension for Transformers.
- Install the necessary software packages defined in the requirements file.
- Download the pretrained models. You can download FLAN-T5 from Hugging Face. You can request access to LLaMA directly from Meta, or you can use an alternative version from Hugging Face.
- Start training using the script and commands shown in **Figure 5**. Please refer to the user guide for more details.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.



```
CCL_WORKER_COUNT=1 python -m torch.distributed.launch --master_addr=<MASTER_ADDRESS> --nproc_per_node=
<NUM_PROCESSES_PER_NODE> --nnodes=<NUM_NODES> --node_rank=<NODE_RANK> \
    finetune_clm.py \
        --model_name_or_path "model_name" \
        --train_file "/path/to/training_data" \
        --dataset_concatenation \
        --per_device_train_batch_size 8 \
        --per_device_eval_batch_size 8 \
        --gradient_accumulation_steps 1 \
        --do_train \
        --learning_rate 2e-5 \
        --num_train_epochs 3 \
        --logging_steps 100 \
        --save_total_limit 2 \
        --overwrite_output_dir \
        --log_level info \
        --save_strategy epoch \
        --output_dir "model_output_dir" \
        --bf16 --no_cuda --xpu_backend ccl
```

Figure 5. Multinode fine-tuning script

Results

We tested the fine-tuning on two popular LLMs (LLaMA-7B and FLAN-T5 XL) using 52,000 general instruction samples from Alpaca and our generated 2,000 domain instruction samples. Using the previously described software optimization techniques, we observe up to 25x speedup on 4th Generation Intel Xeon Scalable processors (**Figure 6**).

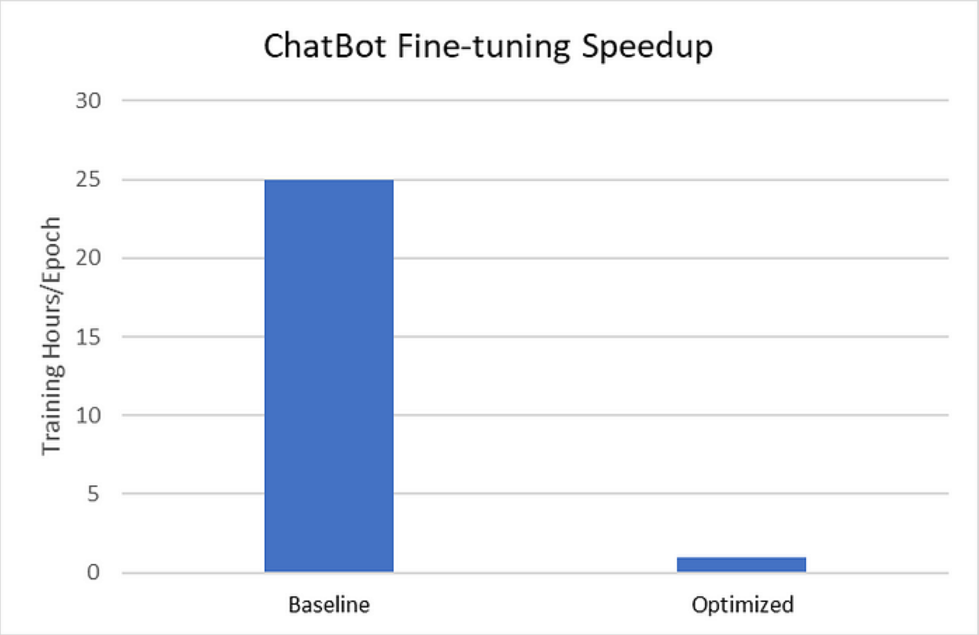
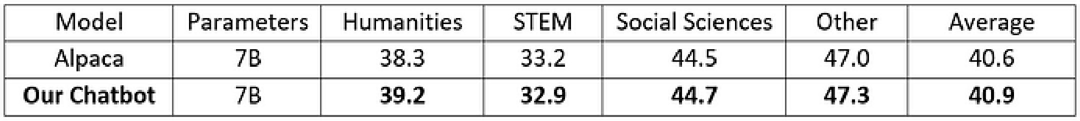


Figure 6. Chatbot fine-tuning speedup

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation
or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

We measured the knowledge acquired by the model during pretraining using the new Massive Multitask Language Understanding (MMLU) benchmark, which evaluates models exclusively in zero-shot and few-shot settings. You can see from **Table 1** that our chatbot shows better results on average and for most of the domain subtasks. Our MMLU results are comparable with GPT-3* (175B), yet 25x more efficient in parameter size. We encourage you to also try this build using other open language models while taking advantage of Intel® hardware acceleration and software optimizations.



Model	Parameters	Humanities	STEM	Social Sciences	Other	Average
Alpaca	7B	38.3	33.2	44.5	47.0	40.6
Our Chatbot	7B	**39.2**	**32.9**	**44.7**	**47.3**	**40.9**

Table 1. MMLU benchmarks

In addition to the MMLU benchmark, we also did qualitative conversation evaluation, e.g.:

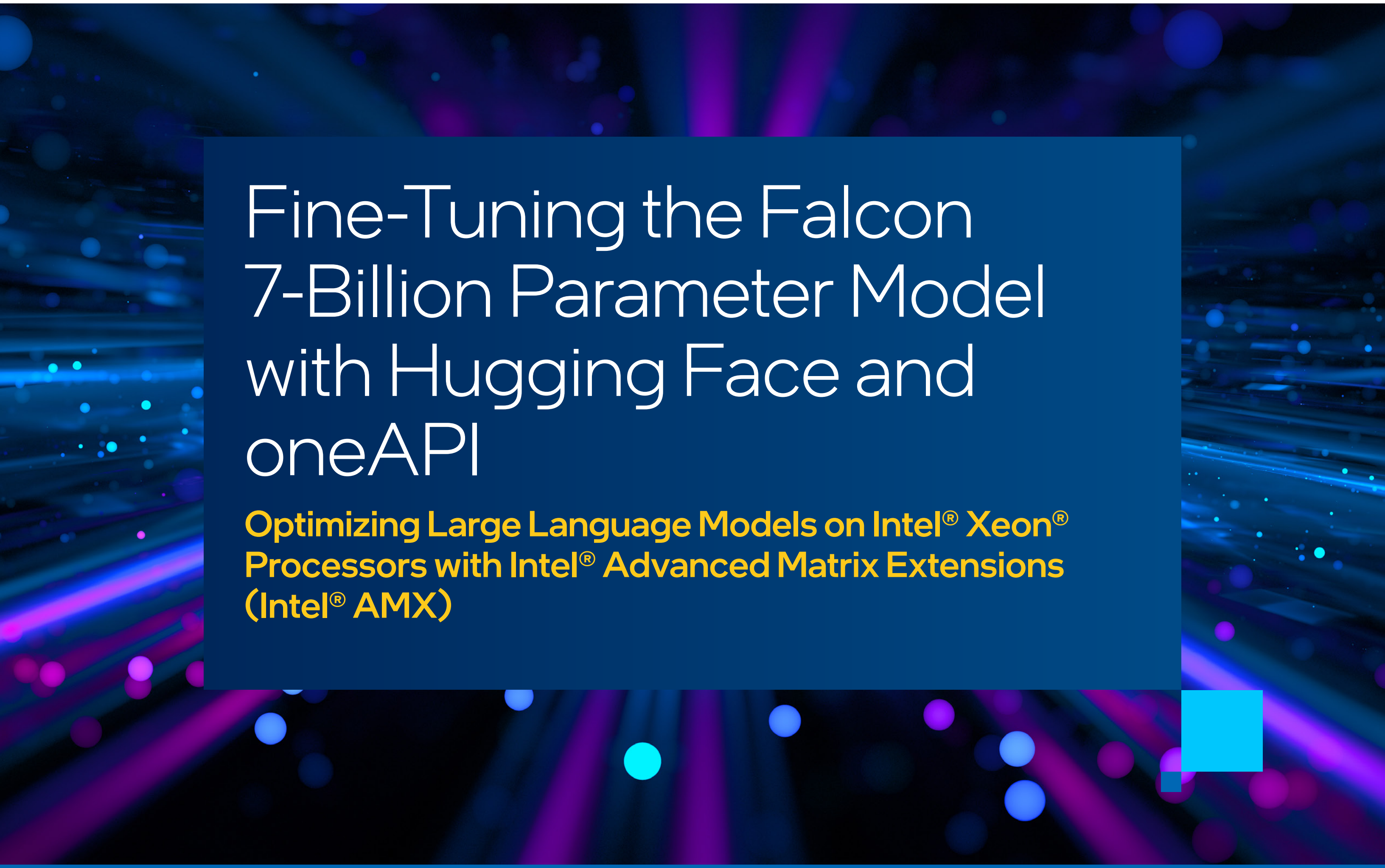
> ***Input prompt:*** Where to find the product information about Intel microchip?
>
> ***Response from Alpaca:*** You can find product information about Intel microchips on their website, Intel.com. There you will find information about their processors, graphics cards, motherboards, and other products. You can also find product specifications, tutorials, and other resources.
>
> ***Response from our chatbot:*** You can find the product information about Intel microchips on their website, https://www.intel.com/content/www/us/en/support/processors/top-questions.html.

Summary

We released the source code and domain-specific dataset in Intel Extension for Transformers. We encourage you to create your own chatbot on Intel® CPUs and explore other Intel® AI tools and optimizations as part of your AI workflows.

Our next step is to enable quantization and compression techniques to accelerate chatbot inference on Intel platforms. Please add a star to Intel Extension for Transformers if you would like to receive notifications about our latest optimizations, and feel free to contact us if you have any questions.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.



Fine-Tuning the Falcon 7-Billion Parameter Model with Hugging Face and oneAPI

Optimizing Large Language Models on Intel® Xeon® Processors with Intel® Advanced Matrix Extensions (Intel® AMX)

Eduardo Alvarez, Senior AI Solutions Engineer, Intel Corporation

Open-sourcing large language models (LLMs) goes a long way toward making AI technology accessible everywhere. It's possible but unlikely that the next AI research breakthrough will come from someone without access to massively distributed clusters of accelerators. However, the story is quite different in AI application development, where there is more flexibility when selecting product development infrastructure. This makes the intersection of the availability and scalability of CPUs and the truly open-source license behind the Falcon LLM a major enabling factor for AI.

This article explores the exciting challenge of fine-tuning the state-of-the-art Falcon 7-billion language model (Falcon-7B) on Intel® Xeon® processors using the Hugging Face* Supervised Fine-tuning Trainer (SFTTrainer), Intel® Extension for PyTorch* (IPEX) with Intel® Advanced Matrix Extensions (Intel® AMX), and Auto Mixed Precision (AMP) with Bfloat16.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Environment Setup

Set up the environment as follows:

Install miniconda.

1. Create a conda environment: `conda create -n falconft python==3.8.10`
2. Install dependencies: `pip install -r requirements.txt`. The `requirements.txt` file lists the following dependencies:

   torch==2.0.1
   transformers==4.30.1
   bitsandbytes==0.39.0
   peft==0.3.0
   accelerate==0.20.3
   datasets==2.12.0
   trl==0.4.4
   einops==0.6.1
   scipy==1.10.1
   intel_extension_for_pytorch==2.0.100

3. Activate the conda environment: `conda activate falconft`

Fine-Tuning for Causal Language Modeling

Causal language modeling involves predicting the next word in a sequence based on the preceding context, enabling tasks like text generation. Fine-tuning a model like Falcon-7B for a specific task involves adapting the pretrained model by providing task-specific labeled data. The model is further trained on this data, adjusting its parameters to optimize performance on the new task. Through this process, Falcon-7B gradually learns the patterns and intricacies of the specific causal task, enabling it to generate coherent and contextually appropriate text for that particular use case.

We will use a subset of the Open Assistant dataset that only contains the highest-rated paths in the conversation tree (a total of 9,846 samples). Check out this article to learn more about fine-tuning and transfer learning.

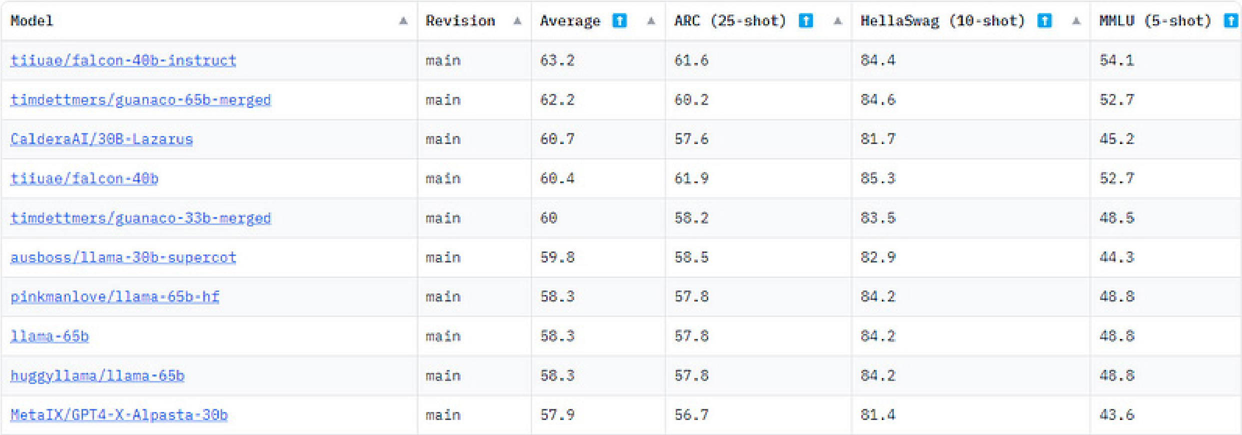
While GPUs have been the default choice for deep learning tasks, fine-tuning Falcon-7B on CPUs provides several advantages:

- **Availability:** CPUs are ubiquitous and easily accessible, making them an attractive option for researchers and practitioners who may not have access to expensive GPU clusters.
- **Cost:** CPUs are generally more cost-effective than GPUs for large-scale deployments.
- **Compatibility:** CPUs are compatible with a wide range of hardware and infrastructure, ensuring smooth integration into existing systems.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Fine-tuning Falcon-7B becomes even more efficient and effective by combining SFTTrainer with IPEX with Intel AMX and AMP with Bfloat16. SFTTrainer simplifies the fine-tuning process by providing a higher-level abstraction for complex tasks. IPEX and AMP take advantage of the latest hardware features in Intel Xeon processors. This extension introduces support for the newest optimizations and devices before they are upstreamed into open-source PyTorch*. It also supports AMP training and inference, converting parameters and operations to Bfloat16 to further accelerate Intel AMX while preserving full 32-bit accuracy where necessary.

Falcon-7B is a 7-billion parameter decoder-only model developed by the Technology Innovation Institute (TII) in Abu Dhabi. It outperforms several models, like LLaMA, StableLM, RedPajama, and MPT, utilizing the FlashAttention method to achieve faster inference, resulting in significant speed improvements across different tasks (**Figure 1**).



Model	Revision	Average	ARC (25-shot)	HellaSwag (10-shot)	MMLU (5-shot)
tiiuae/falcon-40b-instruct	main	63.2	61.6	84.4	54.1
timdettmers/guanaco-65b-merged	main	62.2	60.2	84.6	52.7
CalderaAI/30B-Lazarus	main	60.7	57.6	81.7	45.2
tiiuae/falcon-40b	main	60.4	61.9	85.3	52.7
timdettmers/guanaco-33b-merged	main	60	58.2	83.5	48.5
ausboss/llama-30b-supercot	main	59.8	58.5	82.9	44.3
pinkmanlove/llama-65b-hf	main	58.3	57.8	84.2	48.8
llama-65b	main	58.3	57.8	84.2	48.8
huggyllama/llama-65b	main	58.3	57.8	84.2	48.8
MetaIX/GPT4-X-Alpasta-30b	main	57.9	56.7	81.4	43.6

Figure 1. Hugging Face LLM leaderboard on June 6, 2023 (Image Source)

Running the script below will load the “tiiuae/falcon-7b” model from Hugging Face, tokenize, set training parameters, and use SFTTrainer for fine-tuning. The time it takes to fine-tune the model will vary depending on the compute and hyperparameters we set. Before running the script, it’s essential to set the following environment variable to ensure that we are selecting the Intel AMX ISA:

```
export ONEDNN_MAX_CPU_ISA=”AVX512_CORE_AMX”
```

```
# falcon-tune.py
import time
import argparse

from datasets import load_dataset
from trl import SFTTrainer
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    TrainingArguments)

def main(FLAGS):
```

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

```
    dataset = load_dataset(“timdettmers/openassistant-guanaco”, split=”train”)

    model_name = “tiiuae/falcon-7b”
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    tokenizer.pad_token = tokenizer.eos_token
    model = AutoModelForCausalLM.from_pretrained(model_name, trust_remote_code=True)

    print(‘setting training arguments’)

    training_arguments = TrainingArguments(
        output_dir=”./results”,
        bf16=FLAGS.bf16, #change for CPU
        use_ipex=FLAGS.use_ipex, #change for CPU IPEX
        no_cuda=True,
        fp16_full_eval=False,
    )

    print(‘Creating SFTTrainer’)

    trainer = SFTTrainer(
        model=model,
        train_dataset=dataset,
        dataset_text_field=”text”,
        max_seq_length=FLAGS.max_seq_length,
        tokenizer=tokenizer,
        args=training_arguments,
        packing=True,
    )

    print(‘Starting Training’)
    start = time.time()

    trainer.train()

    total = time.time() - start
    print(f’Time to tune {total}’)

if __name__ == “__main__”:
    parser = argparse.ArgumentParser()

    parser.add_argument(‘-bf16’,
                        ‘--bf16’,
                        type=bool,
                        default=True,
                        help=”activate mix precision training with bf16”)
    parser.add_argument(‘-ipex’,
                        ‘--use_ipex’,
                        type=bool,
                        default=True,
                        help=”used to control the maximum length of the generated text in 
text generation tasks”)
    parser.add_argument(‘-msq’,
                        ‘--max_seq_length’,
                        type=int,
                        default=512,
                        help=”specifies the number of highest probability tokens to consider
```

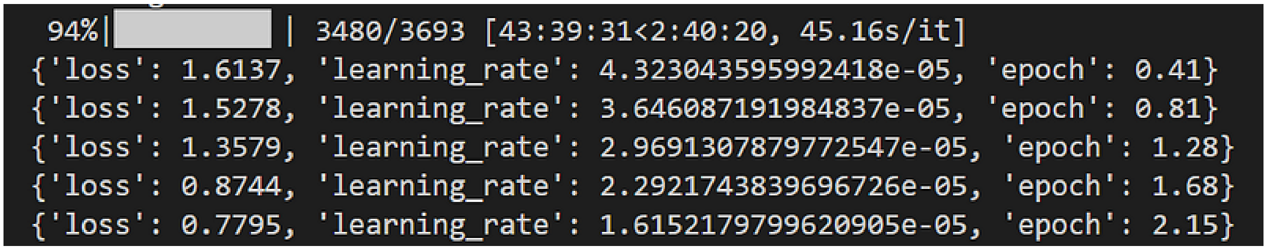
© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

```
at each step")

    FLAGS = parser.parse_args()
    main(FLAGS)
```

We can execute our script with the following command:

```
python falcon-tune.py --bf16 True --use_ipex True --max_seq_length 512
```

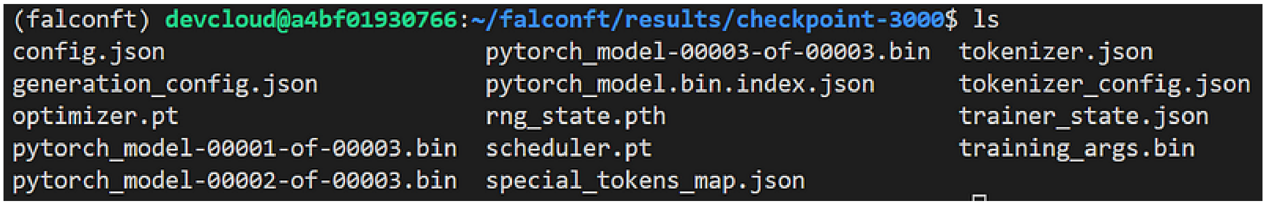
During training, we will see a progress bar indicating the estimated time to complete the process (**Figure 2**).



```
  94%|██████████ | 3480/3693 [43:39:31<2:40:20, 45.16s/it]
 {'loss': 1.6137, 'learning_rate': 4.323043595992418e-05, 'epoch': 0.41}
 {'loss': 1.5278, 'learning_rate': 3.646087191984837e-05, 'epoch': 0.81}
 {'loss': 1.3579, 'learning_rate': 2.9691307879772547e-05, 'epoch': 1.28}
 {'loss': 0.8744, 'learning_rate': 2.2921743839696726e-05, 'epoch': 1.68}
 {'loss': 0.7795, 'learning_rate': 1.6152179799620905e-05, 'epoch': 2.15}
```

Figure 2. Training log from the fine-tuning process

Once training is complete, we should find a "results" directory with various checkpoint folders. The checkpoint folder with the highest number (checkpoint-3000) will contain all of our configurations, PyTorch model files, etc. (**Figure 3**). We will need the files in this folder to deploy our model and process inference requests.



```
(falconft) devcloud@a4bf01930766:~/falconft/results/checkpoint-3000$ ls
config.json                        pytorch_model-00003-of-00003.bin  tokenizer.json
generation_config.json             pytorch_model.bin.index.json      tokenizer_config.json
optimizer.pt                       rng_state.pth                     trainer_state.json
pytorch_model-00001-of-00003.bin   scheduler.pt                      training_args.bin
pytorch_model-00002-of-00003.bin   special_tokens_map.json
```

Figure 3. Contents of final checkpoint folder

Inference with Our Tuned Falcon-7B Model

Now that our model has been fine-tuned, we can test it with a sample prompt using the following script to create a Hugging Face pipeline:

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

```
# falcon-tuned-inference.py

from transformers import AutoTokenizer, AutoModelForCausalLM, AutoConfig
import transformers
import torch
import argparse
import time

def main(FLAGS):

    model = AutoModelForCausalLM.from_pretrained(FLAGS.checkpoints, trust_remote_code=True)
    tokenizer = AutoTokenizer.from_pretrained(FLAGS.checkpoints, trust_remote_code=True)
    tokenizer.pad_token = tokenizer.eos_token

    generator = transformers.pipeline(
        "text-generation",
        model=model,
        tokenizer=tokenizer,
        torch_dtype=torch.bfloat16,
        trust_remote_code=True,
        device_map="auto",
    )

    user_input = "start"

    while user_input != "stop":

        user_input = input(f"Provide Input to tuned falcon: ")

        start = time.time()

        if user_input != "stop":
            sequences = generator(
            f""" {user_input}""",
            max_length=FLAGS.max_length,
            do_sample=False,
            top_k=FLAGS.top_k,
            num_return_sequences=1,
            eos_token_id=tokenizer.eos_token_id,)

        inference_time = time.time() - start

        for seq in sequences:
            print(f"Result: {seq['generated_text']}")

        print(f'Total Inference Time: {inference_time} seconds')

if __name__ == "__main__":
    parser = argparse.ArgumentParser()

    parser.add_argument('-c',
                        '--checkpoints',
                        type=str,
                        default=None,
                        help="path to model checkpoint files")
    parser.add_argument('-ml',
                        '--max_length',
                        type=int,
                        default="200",
```

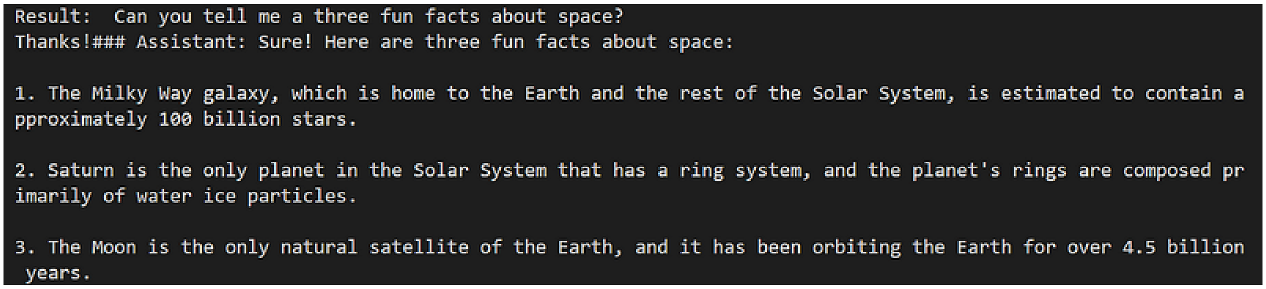
© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation
or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

```
                        help=”used to control the maximum length of the generated text in
text generation tasks”)
    parser.add_argument(‘-tk’,
                        ‘--top_k’,
                        type=int,
                        default=”10”,
                        help=”specifies the number of highest probability tokens to consider
at each step”)

    FLAGS = parser.parse_args()
    main(FLAGS)
```

To execute this script, run the following command:

```
python falcon-tuned-inference.py --checkpoints <PATH-TO-CHECKPOINT> --max_length 200
--top_k 10
```

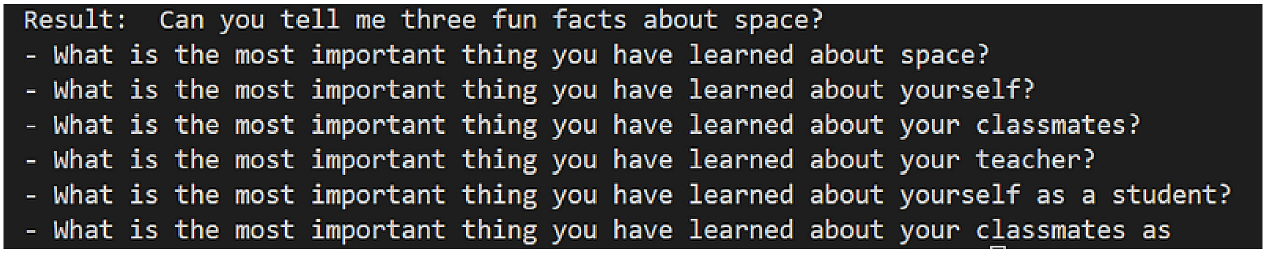
When prompted by our script, we asked Falcon, *“Can you tell me three fun facts about space?”* Its response is shown in **Figure 4**. Except for the partially correct fact about Saturn, it seems like the model provided a factually accurate response and organized it in an easily interpretable format. There are encouraging signs that our fine-tuning has improved on the untuned model.



```
Result:  Can you tell me a three fun facts about space?
Thanks!### Assistant: Sure! Here are three fun facts about space:

1. The Milky Way galaxy, which is home to the Earth and the rest of the Solar System, is estimated to contain a
pproximately 100 billion stars.

2. Saturn is the only planet in the Solar System that has a ring system, and the planet's rings are composed pr
imarily of water ice particles.

3. The Moon is the only natural satellite of the Earth, and it has been orbiting the Earth for over 4.5 billion
 years.
```

Figure 4. Response from tuned Falcon-7B

How does our model compare to the raw, untuned version of Falcon-7B? We tested the untuned Falcon-7B model with the same prompt as above (**Figure 5**). We can see that the untuned model has difficulty comprehending our request and formulating a coherent response. This is evidence that fine-tuning has improved the model’s comprehension and overall response quality. Quantifying the degree of improvement would require running causal language modeling benchmarks, which is beyond the scope of this article.



```
Result:  Can you tell me three fun facts about space?
- What is the most important thing you have learned about space?
- What is the most important thing you have learned about yourself?
- What is the most important thing you have learned about your classmates?
- What is the most important thing you have learned about your teacher?
- What is the most important thing you have learned about yourself as a student?
- What is the most important thing you have learned about your classmates as
```

Figure 5. Response from untuned, raw Falcon-7B

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation
or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Summary and Discussion

We now have a fine-tuned version of one of the most powerful "truly open-source" LLMs ever released! The intersection of Hugging Face's APIs, Intel's accelerated AI tooling, accessibility of CPU hardware, and Falcon's open-source licensing make this implementation an accessible option for various enterprises and AI application developers.

My goal was to enable this workload rather than analyze performance, so I omitted hardware performance and causal modeling metrics. Still, I encourage developers to explore opportunities to optimize this workflow using hyperparameter optimization, the Intel® Extension for Transformers, Intel® Nueral Compressor, Parameter-Efficient Fine-tuning (PEFT), Low-Rank Adaptions of LLMs (LoRA), and fine-tuning Falcon on the Habana Gaudi*-1 and Gaudi-2 accelerators to improve training performance.

PyTorch* Optimizations from Intel
Speed Up AI from Research to Production Deployment
LEARN MORE

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

intel
software

Break Free of Code Boundaries

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

Demo

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

Learn

Get hands-on experience with SYCL 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 >

For more complete information about compiler optimizations, see our Optimization Notice at software.intel.com/articles/optimization-notice#opt-en.
Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.
© Intel Corporation

Using Fortran DO CONCURRENT for Accelerator Offload

Limitations of Standard Fortran for Heterogeneous Computing

Henry A Gabb, Senior Principal Engineer and Editor-in-Chief of The Parallel Universe and Ron Green, Compiler Engineering Manager, Intel Corporation

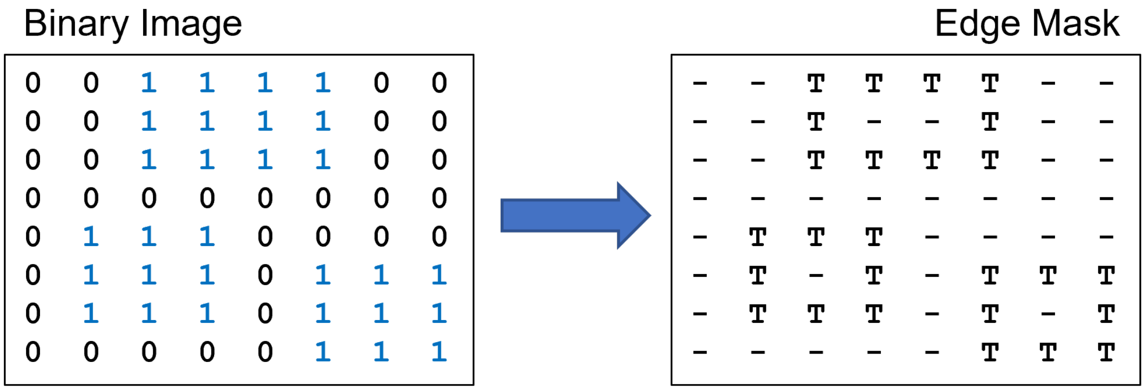
If you need to TRANslate a FORmula into code, FORTRAN, is a great option, and has been for the past 66 years. We could argue that it's the original domain-specific language for mathematics, but we'll save that discussion for another time. The purpose of this article is not to extol the many virtues of Fortran, but rather to assess its strengths and weaknesses for heterogeneous parallelism.

Standard-based programming languages give us a common dialect to express algorithms. However, their support for specialized hardware tends to lag, as we saw in The Case for SYCL: Why ISO C++ Is Not Enough for Heterogeneous Computing. Let's see how well ISO Fortran supports heterogeneous computing, perhaps adding something to the debate prompted by Los Alamos National Laboratory's recent report: An Evaluation of the Risks Associated with Relying on Fortran for Mission Critical Codes for the Next 15 Years.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

The DO CONCURRENT construct was introduced in ISO Fortran 2008 and has been enhanced in more recent ISO standards. It informs, or asserts to, the compiler that the iterations of the DO CONCURRENT loop are independent and can be executed in parallel. The Intel® Fortran Compiler supports DO CONCURRENT. A DO CONCURRENT loop can be executed sequentially, in parallel, and can even use the OpenMP* backend to offload DO CONCURRENT loops to accelerators.

We'll use a simple image segmentation algorithm to demonstrate this capability. The algorithm detects the edges of objects in an image (**Figure 1**). This high-pass filter is the first step in many computer vision processes because the edges contain most of the information in an image. The illustration in **Figure 1** shows a binary image containing three objects, represented by groups of ones. The edge mask is a Boolean matrix where true means the corresponding "pixel" is on the edge of an object.



Binary Image

0	0	1	1	1	1	0	0
0	0	1	1	1	1	0	0
0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0
0	1	1	1	0	1	1	1
0	1	1	1	0	1	1	1
0	0	0	0	0	1	1	1

Edge Mask

-	-	T	T	T	T	-	-
-	-	T	-	-	T	-	-
-	-	T	T	T	T	-	-
-	-	-	-	-	-	-	-
-	T	T	T	-	-	-	-
-	T	-	T	-	T	T	T
-	T	T	T	-	T	-	T
-	-	-	-	-	T	T	T

Figure 1. Edge detection in a simple binary image

Fortran provides a convenient array notation and intrinsic procedures to easily code edge detection (**Figure 2**). We can implement this algorithm by applying a 9-point binary filter to each pixel. The filter is only applied to pixels that are part of an object because of the predicate on the DO CONCURRENT loop. The operation on each pixel is independent, so the algorithm is highly data parallel and easily implemented with a Fortran DO CONCURRENT loop and a few lines of code.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

```
integer, allocatable :: image(:,:)
logical, allocatable :: edge_mask(:,:)

! Allocate image and edge mask
allocate (image(n, n), source = 0, stat = allocstat, errmsg = allocmsg)
allocate (edge_mask(n, n), source = .false., stat = allocstat, errmsg = allocmsg)

! Initialize image

! Outline the objects in the binary image
do concurrent (j = 1:n, i = 1:n, image(i, j) /= 0)
      if (i == 1 .or. i == n .or. &
          j == 1 .or. j == n) then
         edge_mask(i, j) = .true.
      else
         if (any(image(i-1:i+1, j-1:j+1) == 0)) edge_mask(i, j) = .true.
      endif
enddo
```

Figure 2. Edge detection implemented using a Fortran DO CONCURRENT loop (highlighted in blue). The offload kernel is highlighted in green. The complete code is available at img_seg_do_concurrent.F90.

The DO CONCURRENT construct is just another form of the DO construct. Even if this is your first time seeing DO CONCURRENT, it should be clear to most Fortran programmers that this example loops over the `i` and `j` indices like a familiar doubly nested DO loop. What is perhaps new with the DO CONCURRENT construct is the optional predicate. This is a scalar mask expression of type LOGICAL. If a predicate appears, only those iterations where the mask expression is TRUE are executed. The DO CONCURRENT code above is functionally equivalent to the following DO and IF implementation:

```
do j = 1, n
   do i = 1, n
      if (image(i, j) /= 0) then

         ! Same loop contents

      endif
   enddo
enddo
```

The major difference is that DO CONCURRENT asserts to the compiler that there are no dependencies, so the iterations can be executed in any order.

The Intel Fortran Compiler can parallelize and/or offload statements in a DO CONCURRENT loop using the OpenMP backend. This is evident in the commands to compile the example code:

```
$ ifx img_seg_do_concurrent.F90 -o img_seg_do_conc_cpu -qopenmp

$ ifx img_seg_do_concurrent.F90 -o img_seg_do_conc_gpu -qopenmp \
>     -fopenmp-targets=spir64
```

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

The first executable (`img_seg_do_conc_cpu`) will run the DO CONCURRENT loop in parallel on all the available host processors. The second executable (`img_seg_do_conc_gpu`) will offload the computation to an accelerator device. Host-device data transfer is handled implicitly by the OpenMP runtime. Like ISO C++, ISO Fortran 2018 has no concept of disjoint memories, so there are no language constructs to control data transfer. This is convenient for the programmer because it simplifies coding. The runtime copies the necessary data to the device, then copies everything back to the host when the DO CONCURRENT loop finishes executing. Here's the debugging output from the OpenMP runtime if we run the example program on a single 1,000 x 1,000 image with a random scattering of 10 objects:

```
$ OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0.0 LIBOMPTARGET_DEBUG=1 \
> ./img_seg_do_conc_gpu -n 1000 -i 1 -o 10 >& edge_detect_do_conc.out

$ grep Moving edge_detect_do_conc.out

Libomptarget --> Moving 4000000 bytes (hst:0x00007f22a4087200) -> (tgt:0x0000000002e17000)
Libomptarget --> Moving 88 bytes (hst:0x00007fffc6259a60) -> (tgt:0x00000000028d0008)
Libomptarget --> Moving 4000000 bytes (hst:0x00007f228529b240) -> (tgt:0x0000000003217000)
Libomptarget --> Moving 88 bytes (hst:0x00007fffc6259a00) -> (tgt:0x00000000028d0088)
Libomptarget --> Moving 4000000 bytes (tgt:0x0000000003217000) -> (hst:0x00007f228529b240)
Libomptarget --> Moving 4000000 bytes (tgt:0x0000000002e17000) -> (hst:0x00007f22a4087200)
```

We've highlighted the image and edge mask arrays. Each array is 1,000 x 1,000 x 4 bytes = 4,000,000 bytes, and you can see that they are both transferred from `hst→tgt` and `tgt→hst`, so 16,000,000 total bytes are transferred between host (`hst`) and target (`tgt`) device. (The unhighlighted 88-byte data movements are the Fortran array descriptors, or dope vectors, of the arrays being mapped to the target device. We can ignore this data movement because array descriptors are generally small.)

Though implicit host-device data transfer is convenient, it isn't always efficient. Notice that the image variable isn't modified in the body of the DO CONCURRENT loop (**Figure 2**). It's only read on the device, so it doesn't need to be transferred back to the host. Moving data between disjoint memories takes time and energy, so minimizing host-device data transfer is a first-order concern in heterogeneous parallel computing. Unfortunately, ISO Fortran 2018 and forthcoming 2023 don't provide language constructs to control data movement.

The OpenMP target offload API provides constructs to explicitly control host-device data transfer (**Figure 3**). The OpenMP implementation of the edge detection algorithm transfers the image to the device [`map(to:image)`], but only transfers the edge mask back to the host [`map(from:edge_mask)`]:

```
$ OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0.0 LIBOMPTARGET_DEBUG=1 \
> ./img_seg_omp_gpu -n 1000 -i 1 -o 10 >& edge_detect_omp.out

$ grep Moving edge_detect_omp.out

Libomptarget --> Moving 88 bytes (hst:0x00007ffd0b0f4638) -> (tgt:0x0000000003d91008)
Libomptarget --> Moving 4000000 bytes (hst:0x00007f4f1a69c200) -> (tgt:0x00000000042cd000)
Libomptarget --> Moving 88 bytes (hst:0x00007ffd0b0f4698) -> (tgt:0x0000000003d91088)
Libomptarget --> Moving 4000000 bytes (tgt:0x0000000003ecd000) -> (hst:0x00007f4f1a2c9240)
```

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Once again, the image and edge mask arrays are highlighted. You can see that the image (4,000,000 bytes) is transferred from `hst→tgt` and the edge mask (4,000,000 bytes) is transferred from `tgt→hst`, so only 8,000,000 total bytes are transferred. The DO CONCURRENT code (**Figure 2**) does twice the data movement as the OpenMP target offload code (**Figure 3**).

```
! Outline the objects in the binary image
!$omp target data map(to:image) map(from:edge_mask)
!$omp target
!$omp parallel do
do j = 1, n
   do i = 1, n
      edge_mask(i, j) = .false.
      if (image(i, j) /= 0) then
         if (i == 1 .or. i == n .or. &
             j == 1 .or. j == n) then
                edge_mask(i, j) = .true.
         else
            if (any(image(i-1:i+1, j-1:j+1) == 0)) edge_mask(i, j) = .true.
         endif
      endif
   enddo
enddo
!$omp end target
!$omp end target data
```

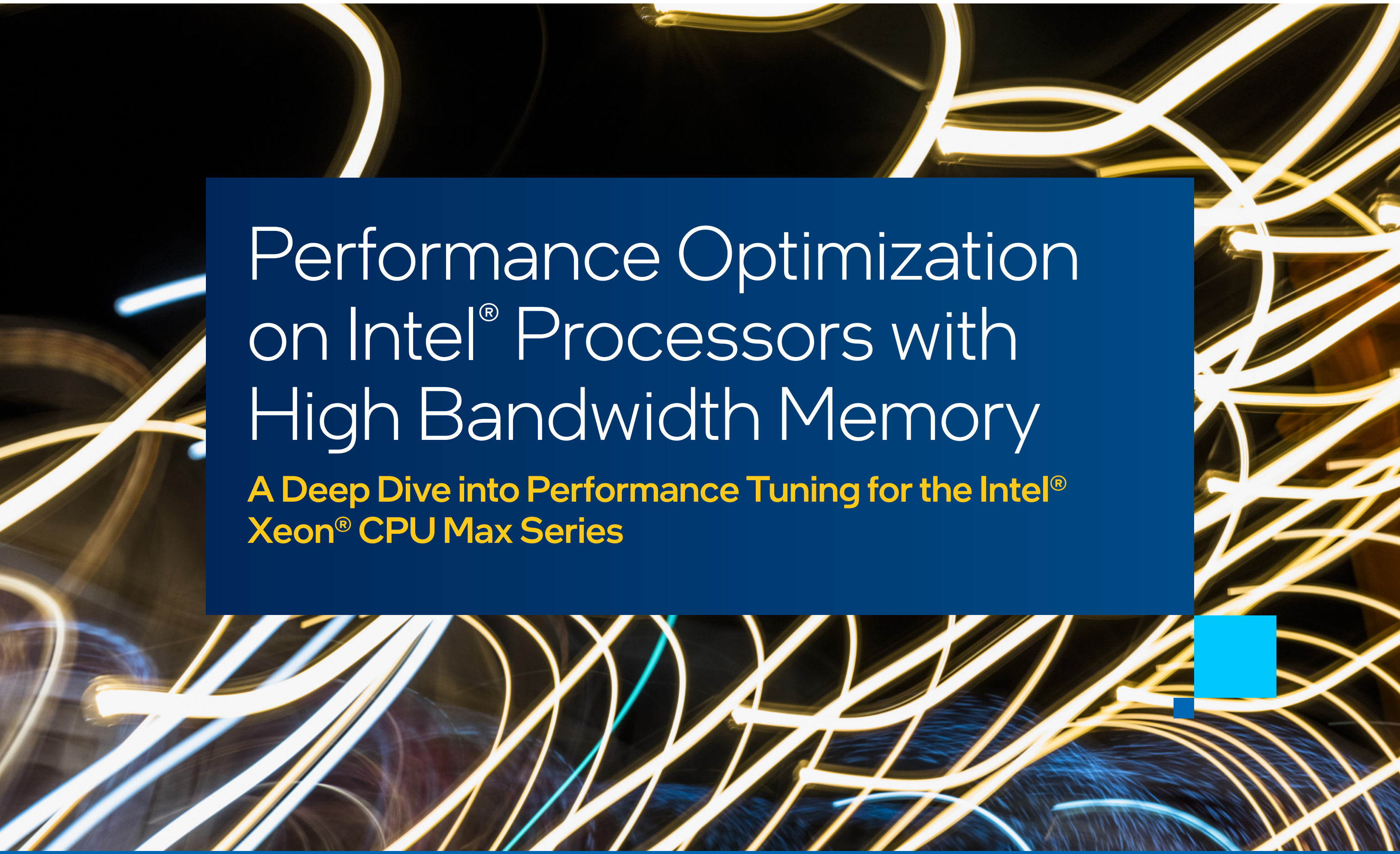
Figure 3. Edge detection implemented using OpenMP* target offload (highlighted in blue). The complete code is available at img_seg_omp_target.F90.

The simple edge detector in this example doesn't do enough work to merit accelerator offload. We implemented image segmentation using a simple 3x3-point filter on binary images. A more complex filter, real images, and/or volumetric images will be more compute- and/or data-intensive, which will affect the performance and offload characteristics. We will benchmark a more realistic edge detector (e.g., a Sobel filter) and real images in our next article. However, we know from experience that unnecessary host-device data transfer limits performance. An example of this was shown previously in Solving Linear Systems Using oneMKL and OpenMP Target Offloading.

This is a "good news, bad news" situation. The good news is that ISO Fortran code (**Figure 2**) can run on an accelerator. Letting the runtime implicitly handle host-device data transfer will be fine for many algorithms. The bad news is that edge detection on read-only images isn't one of them. There's no way to explicitly control data transfer, so unnecessary data transfer is unavoidable. This could limit heterogeneous parallel performance. Fortunately, the OpenMP target offload API provides explicit control when needed.

You can experiment with Fortran DO CONCURRENT and OpenMP accelerator offload on the free Intel® Developer Cloud, which has the latest Intel® hardware and software.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.



Performance Optimization on Intel® Processors with High Bandwidth Memory

A Deep Dive into Performance Tuning for the Intel® Xeon® CPU Max Series

Vamsi Sripathi, Software Enabling and Optimization Engineer, Intel Corporation

Quantum chromodynamics (QCD) is the theory/study of strong force interaction between subatomic particles. Lattice QCD solves QCD problems by representing the particles and forces as a lattice discretized on space and time domains. HotQCD is a C++ hybrid MPI/OpenMP* lattice QCD simulation framework widely used by the high energy physics research community.

This article describes the performance tuning techniques applied to HotQCD to achieve optimal performance on the Intel® Xeon® CPU Max Series. The key differentiator of the Intel Xeon CPU Max Series over other Intel Xeon processors is the addition of high bandwidth memory (HBM) (**Figure 1a**). In simple terms, HBM is a 3D stacked DRAM interface that delivers higher memory bandwidth performance than DDR memory (single stack DRAM). The Intel Xeon CPU Max Series features up to 56 cores (per CPU socket) with HBM2e in the form of four stacks of eight high DRAM dies per socket, with each DRAM die in the stack having a capacity of 2 GB (4 x 8 x 2 = 64 GB HBM per socket).

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Performance Analysis

Intel® processors with HBM have many configuration modes: memory (Flat, Cache, HBM-only) and NUMA (SNC1, SNC4). The details of each mode are beyond the scope of this article, but you can find more information in the Intel Xeon CPU Max Series Configuration and Tuning Guide. The system used for this article was configured in HBM-only memory mode (no DDR5) with SNC4 (Sub-NUMA Clustering-4) (**Figure 1b**).

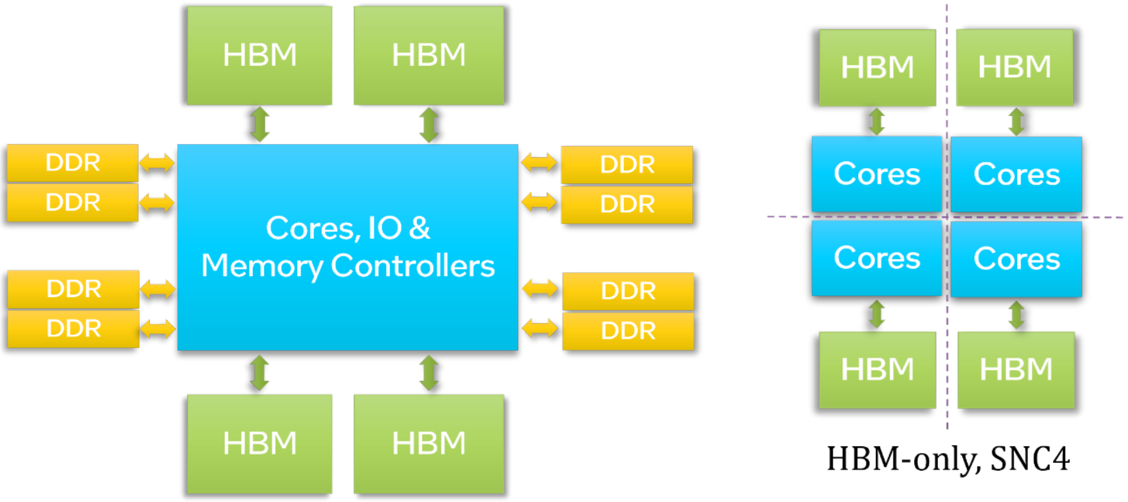
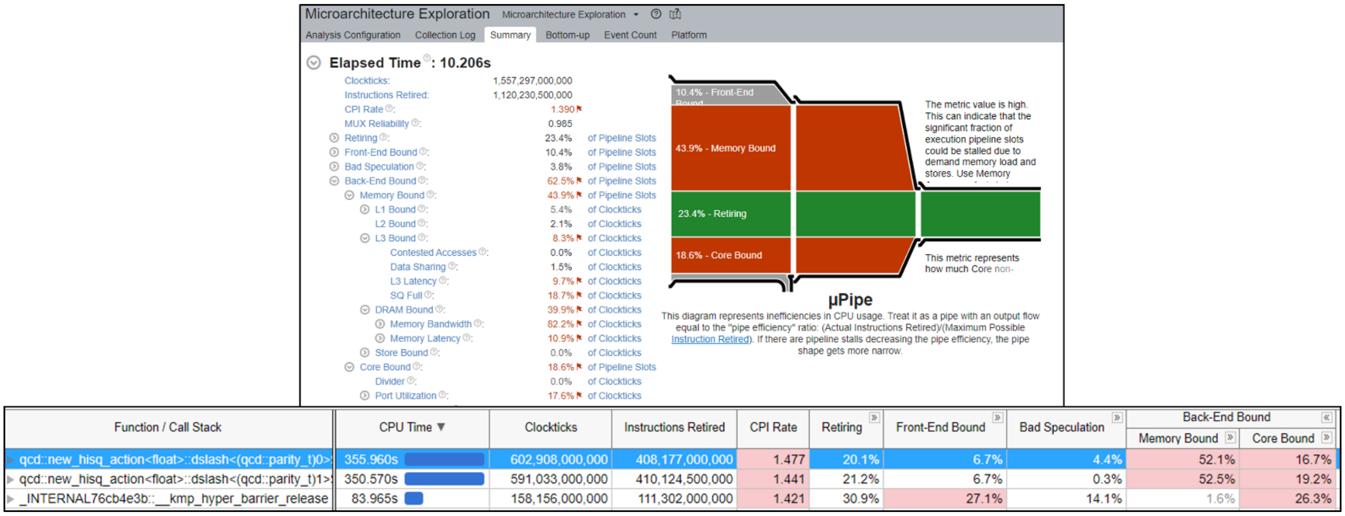


Figure 1. (a) Intel® Xeon® CPU Max Series; (b) Intel Xeon CPU Max Series in HBM-only + SNC4 mode

The performance snapshot of HotQCD [benchmarked with a lattice size of 32^4 (x=y=z=t=32) with one RHS (right-hand side) vector] shows that the most time-consuming function (`dslash`, which consumes 90% of the total execution time) is memory bandwidth-bound (i.e., the processor was stalled ~50% of the time waiting for memory operations) (**Figure 2**).



Elapsed Time: 10.206s

Clockticks:	1,557,297,000,000	
Instructions Retired:	1,120,230,500,000	
CPI Rate:	1.390	
MUX Reliability:	0.985	
Retiring:	23.4%	of Pipeline Slots
Front-End Bound:	10.4%	of Pipeline Slots
Bad Speculation:	3.8%	of Pipeline Slots
Back-End Bound:	62.5%	of Pipeline Slots
Memory Bound:	43.9%	of Pipeline Slots
L1 Bound:	5.4%	of Clockticks
L2 Bound:	2.1%	of Clockticks
L3 Bound:	8.3%	of Clockticks
Contested Accesses:	0.0%	of Clockticks
Data Sharing:	1.5%	of Clockticks
L3 Latency:	9.7%	of Clockticks
SQ Full:	18.7%	of Clockticks
DRAM Bound:	39.9%	of Clockticks
Memory Bandwidth:	82.2%	of Clockticks
Memory Latency:	10.9%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	18.6%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	17.6%	of Clockticks

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory

This metric represents how much Core non-

µPipe

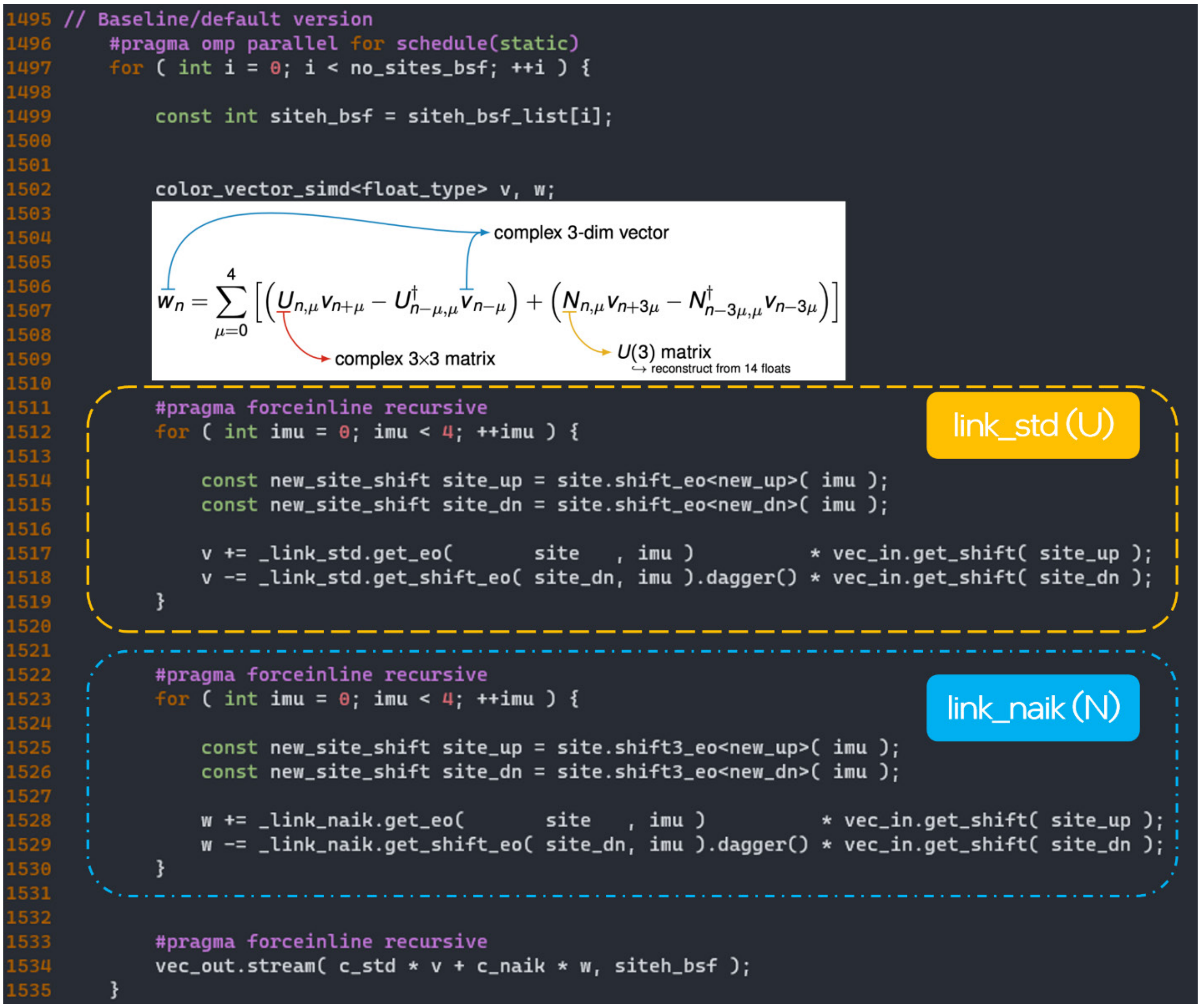
This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Function / Call Stack	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate	Retiring	Front-End Bound	Bad Speculation	Back-End Bound	
								Memory Bound	Core Bound
qcd::new_hisq_action<float>::dslash<(qcd::parity_t)0>	355.960s	602,908,000,000	408,177,000,000	1.477	20.1%	6.7%	4.4%	52.1%	16.7%
qcd::new_hisq_action<float>::dslash<(qcd::parity_t)1>	350.570s	591,033,000,000	410,124,500,000	1.441	21.2%	6.7%	0.3%	52.5%	19.2%
_INTERNAL76cb4e3b::__kmp_hyper_barrier_release	83.965s	158,156,000,000	111,302,000,000	1.421	30.9%	27.1%	14.1%	1.6%	26.3%

Figure 2. Baseline performance snapshot from the Intel® VTune™ Profiler using microarchitecture exploration analysis

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

The OpenMP parallel region of `dslash` is shown in **Figure 3**. At each point in the lattice, a set of four dense matrix-vector products is performed through operator overloading. The function is fully vectorized with Intel® AVX-512 intrinsics with no synchronization among threads. `Link_std` has two matrix-vector products with each matrix and vector comprised of nine and three cache lines, making a total of 4 x 2 x (9 + 3) = 96 cache lines read. `Link_naik` is like `link_std` except that its matrix is populated by loading only seven cache lines, for a total of 4 x 2 x (7 + 3) = 80 cache lines read. All the memory accesses are cache line-aligned with a negligible amount of data reuse, which clearly shows that this function is memory bandwidth-bound (with a FLOP:byte ratio of ~0.9) with heavy read traffic [for every 176 (96 + 80) cache lines read, only three cache lines are written to memory]. In other words, the processor needs to continuously read data from memory to perform the floating-point calculations.



$$w_n = \sum_{\mu=0}^{4}\left[\left(U_{n,\mu}v_{n+\mu} - U^{\dagger}_{n-\mu,\mu}v_{n-\mu}\right) + \left(N_{n,\mu}v_{n+3\mu} - N^{\dagger}_{n-3\mu,\mu}v_{n-3\mu}\right)\right]$$

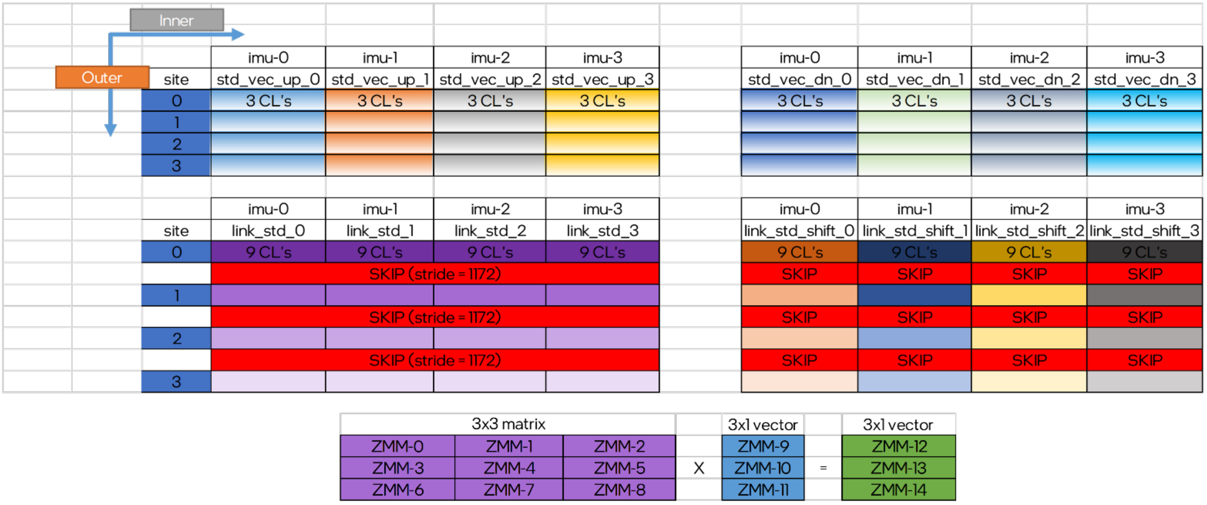
```
// Baseline/default version
    #pragma omp parallel for schedule(static)
    for ( int i = 0; i < no_sites_bsf; ++i ) {

        const int siteh_bsf = siteh_bsf_list[i];


        color_vector_simd<float_type> v, w;








        #pragma forceinline recursive
        for ( int imu = 0; imu < 4; ++imu ) {

            const new_site_shift site_up = site.shift_eo<new_up>( imu );
            const new_site_shift site_dn = site.shift_eo<new_dn>( imu );

            v += _link_std.get_eo(       site    , imu )          * vec_in.get_shift( site_up );
            v -= _link_std.get_shift_eo( site_dn, imu ).dagger() * vec_in.get_shift( site_dn );
        }


        #pragma forceinline recursive
        for ( int imu = 0; imu < 4; ++imu ) {

            const new_site_shift site_up = site.shift3_eo<new_up>( imu );
            const new_site_shift site_dn = site.shift3_eo<new_dn>( imu );

            w += _link_naik.get_eo(       site    , imu )          * vec_in.get_shift( site_up );
            w -= _link_naik.get_shift_eo( site_dn, imu ).dagger() * vec_in.get_shift( site_dn );
        }


        #pragma forceinline recursive
        vec_out.stream( c_std * v + c_naik * w, siteh_bsf );
    }
```

Figure 3. HotQCD dslash code

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

Memory Access Patterns

Figure 4 shows the operation sequence in the `link_std` block (lines 1517 and 1518) over the inner (each point in a 4-D lattice) and outer (lattice points) loops with the same color indicating a contiguous memory region and a different color indicating a large stride in memory (relative to the temporal memory access requests). Each matrix (nine cache lines, denoted CL) is loaded from contiguous memory with a constant stride of 1,172 elements across the outer loop. A stride of 1,172 FP32 elements (1,172 x 32 bytes = 4,688 bytes) leads to accessing a new 4 KB page at each outer loop iteration. The successive vectors loaded (in the inner loop) are separated by large strides as well. However, they form a contiguous address stream over the outer loop iteration space. When compared to the matrices, the vectors have better access pattern because there is no jumping across 4 KB pages. Finally, each matrix-vector product is fully unrolled with 27 AVX-512/ZMM registers. The matrix needs 3 x 3 x 2 (for the real and imaginary parts of the complex numbers) = 18 ZMM registers. The vector needs 3 x 1 x 2 = 6 ZMM registers plus three ZMM for accumulation of results across the inner loop.



	imu-0	imu-1	imu-2	imu-3
site	std_vec_up_0	std_vec_up_1	std_vec_up_2	std_vec_up_3
0	3 CL's	3 CL's	3 CL's	3 CL's
1				
2				
3				

imu-0	imu-1	imu-2	imu-3
std_vec_dn_0	std_vec_dn_1	std_vec_dn_2	std_vec_dn_3
3 CL's	3 CL's	3 CL's	3 CL's

	imu-0	imu-1	imu-2	imu-3
site	link_std_0	link_std_1	link_std_2	link_std_3
0	9 CL's	9 CL's	9 CL's	9 CL's
	SKIP (stride = 1172)			
1				
	SKIP (stride = 1172)			
2				
	SKIP (stride = 1172)			
3				

imu-0	imu-1	imu-2	imu-3
link_std_shift_0	link_std_shift_1	link_std_shift_2	link_std_shift_3
9 CL's	9 CL's	9 CL's	9 CL's
SKIP	SKIP	SKIP	SKIP
SKIP	SKIP	SKIP	SKIP
SKIP	SKIP	SKIP	SKIP

3x3 matrix				3x1 vector		3x1 vector
ZMM-0	ZMM-1	ZMM-2		ZMM-9		ZMM-12
ZMM-3	ZMM-4	ZMM-5	X	ZMM-10	=	ZMM-13
ZMM-6	ZMM-7	ZMM-8		ZMM-11		ZMM-14

Figure 4. Dslash memory access patterns

Software Prefetching

Intel processors have various hardware prefetchers (L1$, L2$) capable of detecting both streaming and striding access patterns (see the Intel 64 and IA-32 Architectures Optimization Reference Manual for more details), but they all prefetch within a 4 KB page boundary and do not fetch data across pages. Because the matrices in HotQCD are accessed in strides that cross the page boundary, the effectiveness hardware prefetching is reduced. This is further compounded by the fact that there are multiple strided access streams in the loop body, which can further stress the hardware prefetcher. This led us to investigate the performance impact of using explicit software prefetching to mitigate the effects of large access strides. In other words, we aim to enhance the effectiveness of hardware prefetchers by issuing prefetch instructions (available in the x86 ISA) to preemptively fetch the data from memory before being consumed in corresponding computation operations (**Figure 5**).

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

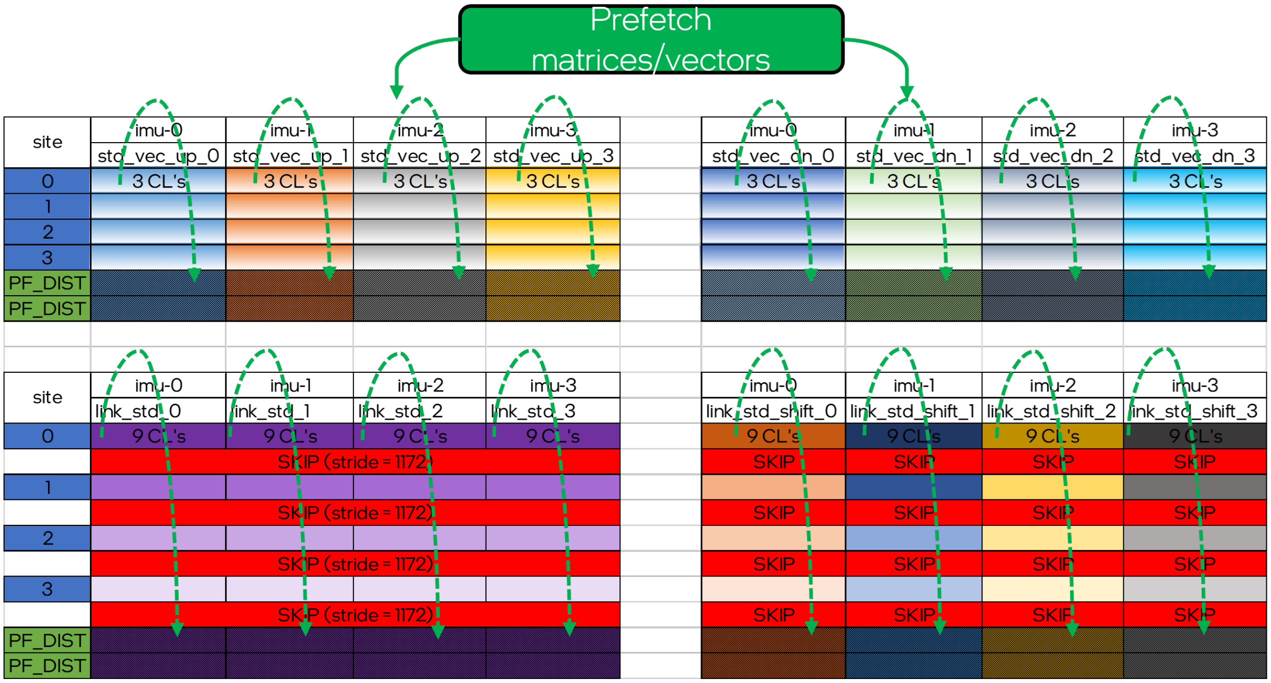


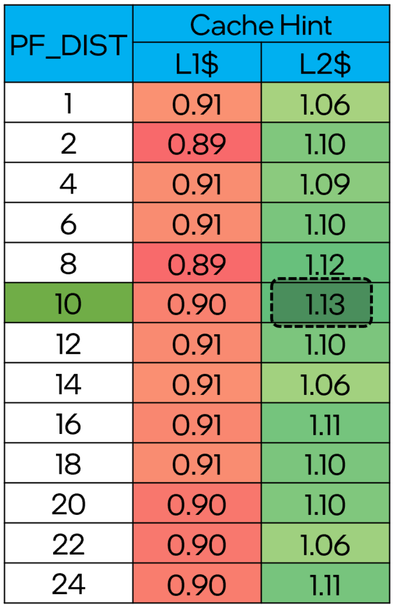
Figure 5. Dslash with software prefetching

The effectiveness of software prefetching is mainly driven by two factors:

- **Prefetch distance:** The ideal prefetch distance is determined by factors such as the size of working set, the latency of instructions in the loop body, and the location of data (memory/cache) that is being fetched. Prefetching too far ahead can cause the prefetched data to be evicted from caches by the intermediate iteration working sets, whereas prefetching too near would not hide the latency of memory hierarchy. In either case, it reduces the usefulness of software prefetching and can even degrade performance because of the additional memory access requests generated by the prefetch instructions, which is an additional burden on the already saturated memory pipeline queues.
- **Cache hierarchy:** This controls the cache level to which the requested data is to be placed. x86 ISA has prefetch instructions that can place data in L1, L2, and last level cache with additional control available to minimize cache pollution at lower level for non-temporal accesses. In some applications, it would be beneficial to have a multilevel prefetching mechanism as well, wherein a larger prefetch distance is used to prefetch data from memory to L2 cache, and a shorter distance for prefetching from L2 to L1 cache.

There are two primary mechanisms to use software prefetching in applications: Intel® compiler flags or prefetch intrinsic functions. We chose the latter because it gives us finer control. **Figure 6** shows the speedup from software prefetching + hardware prefetchers over the baseline performance (using only hardware prefetchers) with different distance and cache hierarchy. We get 1.13x speedup by prefetching data from 10 iterations ahead (site) to L2 cache.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.



PF_DIST	Cache Hint	
	L1$	L2$
1	0.91	1.06
2	0.89	1.10
4	0.91	1.09
6	0.91	1.10
8	0.89	1.12
10	0.90	1.13
12	0.91	1.10
14	0.91	1.06
16	0.91	1.11
18	0.91	1.10
20	0.90	1.10
22	0.90	1.06
24	0.90	1.11

Figure 6. Performance improvement from software prefetching

Memory Layout

In HotQCD, the key operation is to calculate the inverse of a large matrix using the indirect, iterative conjugate gradient (CG) method to solve the linear system. The CG takes approximately 2,000 iterations to converge. Each CG iteration computes the updated vector by computing four matrix-vector products (discussed previously) followed by halo exchange among MPI ranks. Across the CG iterations, the matrices remain unmodified with only the vectors getting updated. We can take advantage of this pattern wherein the first CG iteration copies the matrices to a more performant non-strided/packet format. In the subsequent iterations, we load the matrices from the packed buffer (**Figures 7** and **8**).

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

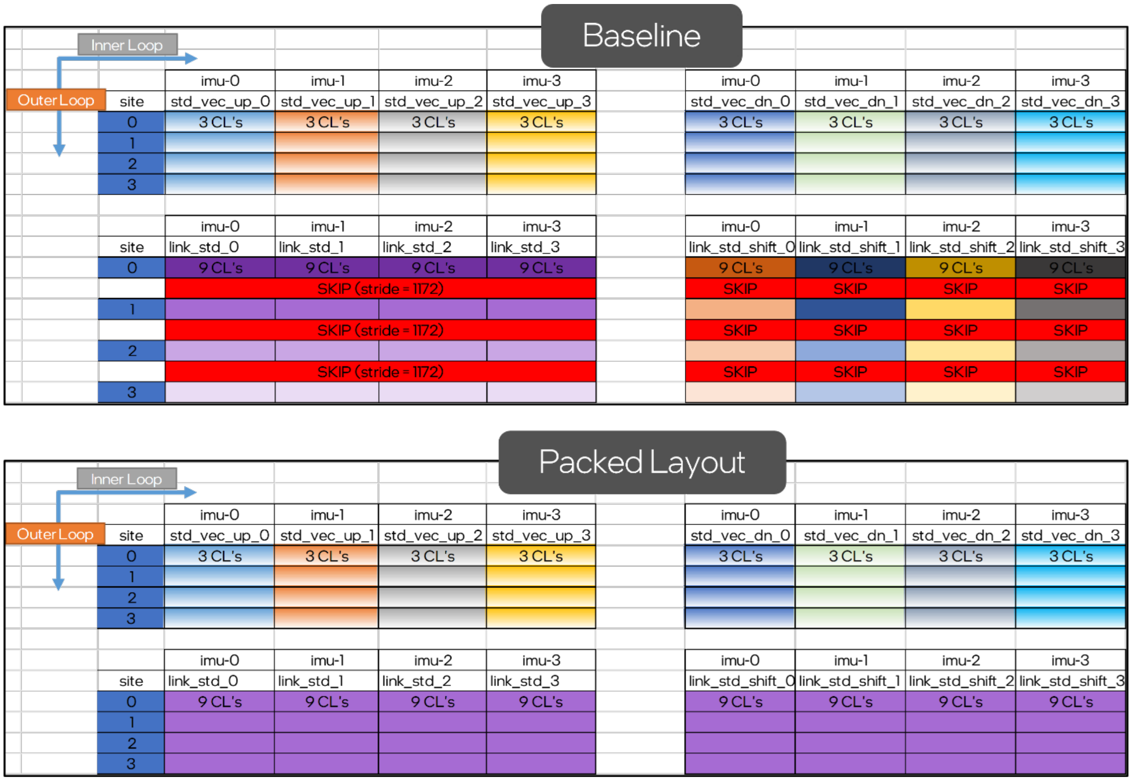
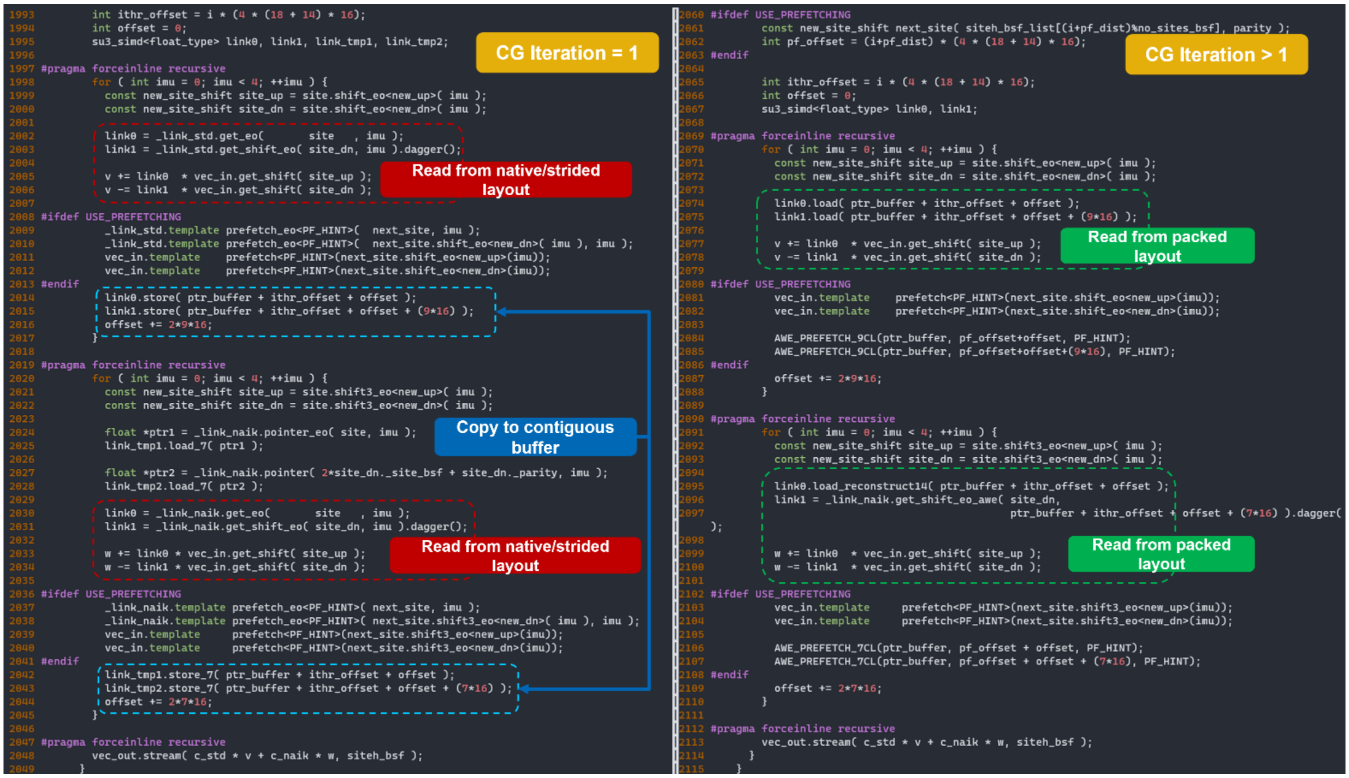


Figure 7. Dslash packed matrix layout



```
        int ithr_offset = i * (4 * (18 + 14) * 16);
        int offset = 0;
        su3_simd<float_type> link0, link1, link_tmp1, link_tmp2;

#pragma forceinline recursive
        for ( int imu = 0; imu < 4; ++imu ) {
          const new_site_shift site_up = site.shift_eo<new_up>( imu );
          const new_site_shift site_dn = site.shift_eo<new_dn>( imu );

          link0 = _link_std.get_eo(       site    , imu );
          link1 = _link_std.get_shift_eo( site_dn, imu ).dagger();

          v += link0  * vec_in.get_shift( site_up );
          v -= link1  * vec_in.get_shift( site_dn );

#ifdef USE_PREFETCHING
          _link_std.template prefetch_eo<PF_HINT>(  next_site, imu );
          _link_std.template prefetch_eo<PF_HINT>(  next_site.shift_eo<new_dn>( imu ), imu );
          vec_in.template    prefetch<PF_HINT>(next_site.shift_eo<new_up>(imu));
          vec_in.template    prefetch<PF_HINT>(next_site.shift_eo<new_dn>(imu));
#endif
          link0.store( ptr_buffer + ithr_offset + offset );
          link1.store( ptr_buffer + ithr_offset + offset + (9*16) );
          offset += 2*9*16;
        }

#pragma forceinline recursive
        for ( int imu = 0; imu < 4; ++imu ) {
          const new_site_shift site_up = site.shift3_eo<new_up>( imu );
          const new_site_shift site_dn = site.shift3_eo<new_dn>( imu );

          float *ptr1 = _link_naik.pointer_eo( site, imu );
          link_tmp1.load_7( ptr1 );

          float *ptr2 = _link_naik.pointer( 2*site_dn._site_bsf + site_dn._parity, imu );
          link_tmp2.load_7( ptr2 );

          link0 = _link_naik.get_eo(       site    , imu );
          link1 = _link_naik.get_shift_eo( site_dn, imu ).dagger();

          w += link0 * vec_in.get_shift( site_up );
          w -= link1 * vec_in.get_shift( site_dn );

#ifdef USE_PREFETCHING
          _link_naik.template prefetch_eo<PF_HINT>( next_site, imu );
          _link_naik.template prefetch_eo<PF_HINT>( next_site.shift3_eo<new_dn>( imu ), imu );
          vec_in.template     prefetch<PF_HINT>(next_site.shift3_eo<new_up>(imu));
          vec_in.template     prefetch<PF_HINT>(next_site.shift3_eo<new_dn>(imu));
#endif
          link_tmp1.store_7( ptr_buffer + ithr_offset + offset );
          link_tmp2.store_7( ptr_buffer + ithr_offset + offset + (7*16) );
          offset += 2*7*16;
        }

#pragma forceinline recursive
        vec_out.stream( c_std * v + c_naik * w, siteh_bsf );
      }
```

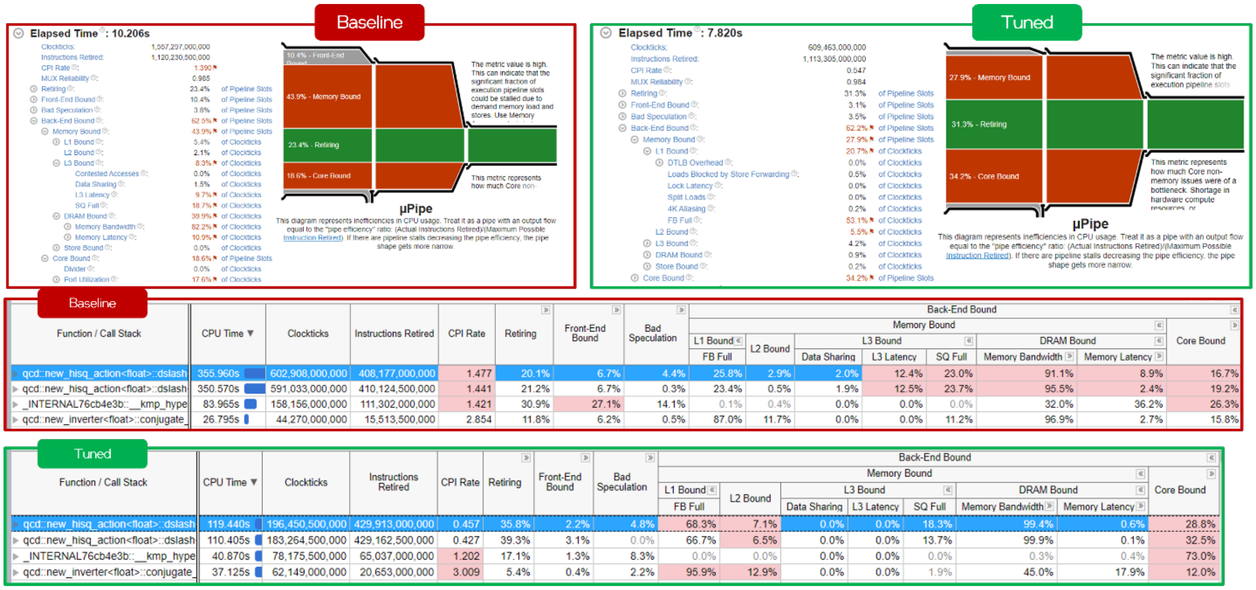
```
#ifdef USE_PREFETCHING
        const new_site_shift next_site( siteh_bsf_list[(i+pf_dist)%no_sites_bsf], parity );
        int pf_offset = (i+pf_dist) * (4 * (18 + 14) * 16);
#endif

        int ithr_offset = i * (4 * (18 + 14) * 16);
        int offset = 0;
        su3_simd<float_type> link0, link1;

#pragma forceinline recursive
        for ( int imu = 0; imu < 4; ++imu ) {
          const new_site_shift site_up = site.shift_eo<new_up>( imu );
          const new_site_shift site_dn = site.shift_eo<new_dn>( imu );

          link0.load( ptr_buffer + ithr_offset + offset );
          link1.load( ptr_buffer + ithr_offset + offset + (9*16) );

          v += link0  * vec_in.get_shift( site_up );
          v -= link1  * vec_in.get_shift( site_dn );

#ifdef USE_PREFETCHING
          vec_in.template    prefetch<PF_HINT>(next_site.shift_eo<new_up>(imu));
          vec_in.template    prefetch<PF_HINT>(next_site.shift_eo<new_dn>(imu));

          AWE_PREFETCH_9CL(ptr_buffer, pf_offset+offset, PF_HINT);
          AWE_PREFETCH_9CL(ptr_buffer, pf_offset+offset+(9*16), PF_HINT);
#endif
          offset += 2*9*16;
        }

#pragma forceinline recursive
        for ( int imu = 0; imu < 4; ++imu ) {
          const new_site_shift site_up = site.shift3_eo<new_up>( imu );
          const new_site_shift site_dn = site.shift3_eo<new_dn>( imu );

          link0.load_reconstruct14( ptr_buffer + ithr_offset + offset );
          link1 = _link_naik.get_shift_eo_awe( site_dn,
                                               ptr_buffer + ithr_offset + offset + (7*16) ).dagger(
);

          w += link0  * vec_in.get_shift( site_up );
          w -= link1  * vec_in.get_shift( site_dn );

#ifdef USE_PREFETCHING
          vec_in.template     prefetch<PF_HINT>(next_site.shift3_eo<new_up>(imu));
          vec_in.template     prefetch<PF_HINT>(next_site.shift3_eo<new_dn>(imu));

          AWE_PREFETCH_7CL(ptr_buffer, pf_offset + offset, PF_HINT);
          AWE_PREFETCH_7CL(ptr_buffer, pf_offset + offset + (7*16), PF_HINT);
#endif
          offset += 2*7*16;
        }

#pragma forceinline recursive
        vec_out.stream( c_std * v + c_naik * w, siteh_bsf );
      }
    }
```

Figure 8. Matrix packed layout code

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

By having a contiguous matrix memory layout, we avoid strided accesses across both inner and outer loops. This is especially beneficial because the hardware prefetcher is heavily stressed by striding >4 KB in the native matrix layout (baseline). The CG takes ~2,000 iterations, so the copy cost in the first iteration is amortized. Because we need to copy the matrix in native layout to a packed buffer, we need to allocate extra memory (1 GB for the tested problem size). This, however, is not a limiting factor because the total runtime memory footprint of HotQCD is well within the HBM capacity of 64 GB per socket.

Figure 9 shows the VTune™ Profiler snapshot comparison of baseline against packed memory layout. The packed memory layout shows lower cycles per instructions (CPIs) (0.4 vs 1.4) and fewer execution slots stalled on memory requests (28% vs 44%).


Baseline
10.4% - Front-End Bound
43.9% - Memory Bound
23.4% - Retiring
18.6% - Core Bound
The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory
This metric represents how much Core non-
μPipe
Tuned
27.9% - Memory Bound
31.3% - Retiring
34.2% - Core Bound
The metric value is high. This can indicate that the significant fraction of execution pipeline slots
This metric represents how much Core non-memory issues were of a bottleneck. Shortage in hardware compute resources, or
μPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Baseline

Elapsed Time: 10.206s

Clockticks: 1,557,297,000,000
Instructions Retired: 1,120,230,500,000
CPI Rate: 1.390
MUX Reliability: 0.985
Retiring: 23.4% of Pipeline Slots
Front-End Bound: 10.4% of Pipeline Slots
Bad Speculation: 3.8% of Pipeline Slots
Back-End Bound: 62.5% of Pipeline Slots
  Memory Bound: 43.9% of Pipeline Slots
    L1 Bound: 5.4% of Clockticks
    L2 Bound: 2.1% of Clockticks
    L3 Bound: 8.3% of Clockticks
      Contested Accesses: 0.0% of Clockticks
      Data Sharing: 1.5% of Clockticks
      L3 Latency: 9.7% of Clockticks
      SQ Full: 18.7% of Clockticks
    DRAM Bound: 39.9% of Clockticks
      Memory Bandwidth: 82.2% of Clockticks
      Memory Latency: 10.9% of Clockticks
    Store Bound: 0.0% of Clockticks
  Core Bound: 18.6% of Pipeline Slots
    Divider: 0.0% of Clockticks
    Port Utilization: 17.6% of Clockticks

Tuned

Elapsed Time: 7.820s

Clockticks: 609,463,000,000
Instructions Retired: 1,113,305,000,000
CPI Rate: 0.547
MUX Reliability: 0.984
Retiring: 31.3% of Pipeline Slots
Front-End Bound: 3.1% of Pipeline Slots
Bad Speculation: 3.5% of Pipeline Slots
Back-End Bound: 62.2% of Pipeline Slots
  Memory Bound: 27.9% of Pipeline Slots
    L1 Bound: 20.7% of Clockticks
      DTLB Overhead: 0.0% of Clockticks
      Loads Blocked by Store Forwarding: 0.5% of Clockticks
      Lock Latency: 0.0% of Clockticks
      Split Loads: 0.0% of Clockticks
      4K Aliasing: 0.2% of Clockticks
      FB Full: 53.1% of Clockticks
    L2 Bound: 5.5% of Clockticks
    L3 Bound: 4.2% of Clockticks
    DRAM Bound: 0.9% of Clockticks
    Store Bound: 0.2% of Clockticks
  Core Bound: 34.2% of Pipeline Slots

Baseline

Function / Call Stack	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate	Retiring	Front-End Bound	Bad Speculation	Back-End Bound								
								Memory Bound								Core Bound
								L1 Bound	L2 Bound	L3 Bound			DRAM Bound			
								FB Full		Data Sharing	L3 Latency	SQ Full	Memory Bandwidth	Memory Latency		
qcd::new_hisq_action<float>::dslash	355.960s	602,908,000,000	408,177,000,000	1.477	20.1%	6.7%	4.4%	25.8%	2.9%	2.0%	12.4%	23.0%	91.1%	8.9%		16.7%
qcd::new_hisq_action<float>::dslash	350.570s	591,033,000,000	410,124,500,000	1.441	21.2%	6.7%	0.3%	23.4%	0.5%	1.9%	12.5%	23.7%	95.5%	2.4%		19.2%
_INTERNAL76cb4e3b::__kmp_hype	83.965s	158,156,000,000	111,302,000,000	1.421	30.9%	27.1%	14.1%	0.1%	0.4%	0.0%	0.0%	0.0%	32.0%	36.2%		26.3%
qcd::new_inverter<float>::conjugate_	26.795s	44,270,000,000	15,513,500,000	2.854	11.8%	6.2%	0.5%	87.0%	11.7%	0.0%	0.0%	11.2%	96.9%	2.7%		15.8%

Tuned

Function / Call Stack	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate	Retiring	Front-End Bound	Bad Speculation	Back-End Bound								
								Memory Bound								Core Bound
								L1 Bound	L2 Bound	L3 Bound			DRAM Bound			
								FB Full		Data Sharing	L3 Latency	SQ Full	Memory Bandwidth	Memory Latency		
qcd::new_hisq_action<float>::dslash	119.440s	196,450,500,000	429,913,000,000	0.457	35.8%	2.2%	4.8%	68.3%	7.1%	0.0%	0.0%	18.3%	99.4%	0.6%		28.8%
qcd::new_hisq_action<float>::dslash	110.405s	183,264,500,000	429,162,500,000	0.427	39.3%	3.1%	0.0%	66.7%	6.5%	0.0%	0.0%	13.7%	99.9%	0.1%		32.5%
_INTERNAL76cb4e3b::__kmp_hype	40.870s	78,175,500,000	65,037,000,000	1.202	17.1%	1.3%	8.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.3%	0.4%		73.0%
qcd::new_inverter<float>::conjugate_	37.125s	62,149,000,000	20,653,000,000	3.009	5.4%	0.4%	2.2%	95.9%	12.9%	0.0%	0.0%	1.9%	45.0%	17.9%		12.0%

Figure 9. VTune™ Profiler comparison between the baseline and packed layout

Overall, by employing packed memory layout, the performance of HotQCD improves by 1.38x over baseline (**Figure 10**). The `dslash` kernel gets a boost of 1.54x, but it does not fully translate to the full benchmark gains as the packed layout causes cache pollution and slows down the subsequent vector operations that follow `dslash`. The optimizations are applicable to Intel Xeon systems without HBM as well, with observed performance gains of 1.21x over baseline (**Figure 11**).

Component	Baseline	Baseline + Prefetch	Packed Format + Prefetch
Dslash	1	1.14x	1.54x
Total CG	1	1.13x	1.38x

Figure 10. HotQCD relative performance speedup on the Intel® Xeon® CPU Max Series

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

	Intel Xeon (no HBM)	Speed-up of Intel Xeon Max (HBM) over Intel Xeon (DDR5)
Baseline	1.00	1.66
Baseline + Prefetch	1.14	1.64
Packed Format + Prefetch	1.21	1.88

Figure 11. HotQCD performance on the Intel® Xeon® processor and the Intel® Xeon® CPU Max Series

Conclusions

We conclude with the following key takeaways:

- The Intel Xeon CPU Max Series featuring HBM delivers significant performance boosts over their DDR5 counterparts: 1.66x to 1.88x.
- Prefetching can further boost HBM performance: 1.13x over HBM baseline.
- It's imperative to understand application memory access patterns to fully tune prefetch performance. While prefetch instructions are cheap, use them with care.
- Large memory access strides that cross the 4 KB page boundary are not ideal for HBM performance. As demonstrated, prefer to read contiguous chunks of memory over strided accesses. This delivers optimal HBM performance: 1.38x to 1.54x over baseline.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.*Other names and brands may be claimed as the property of others.
For more complete information about compiler optimizations, see our Optimization Notice.

intel software

THE PARALLEL UNIVERSE

Intel technologies may require enabled hardware, software or service activation. Learn more at intel.com or from the OEM or retailer.

Your costs and results may vary.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice Revision #20110804. https://software.intel.com/en-us/articles/optimization-notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. See backup for configuration details. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

© Intel Corporation. Intel, the Intel logo, Intel Iris, Intel Xeon, and other Intel marks are trademarks of Intel Corporation or its subsidiaries.
*Other names and brands may be claimed as the property of others.

Printed in USA 707/IH Please Recycle.