OpenNUMA: A Custom Shared System-Memory Allocator for Heterogeneous Neural-Network Computing Across CPU and Integrated GPU

Md. Muminul Islam
Independent Researcher
Dhaka, Bangladesh
ORCID: 0009-0003-5463-8771

August 2026

Abstract

Neural network and machine-learning workloads commonly rely on dedicated GPUs with tensor cores, While existing systems with integrated GPUs (iGPUs) may have unused computing power. This study investigates whether a custom memory allocator can enable existing or older systems to utilize integrated GPUs for such workloads without requiring hardware-level modification. The proposed method does not require unified virtual memory mappings.

OpenNUMA (Open Source Neural Unified Memory Allocator) is a custom allocator developed for these tasks. Although OpenNUMA proposed to platform-independent, in this study it relies on PyTorch XPU backend, Intel oneAPI SYCL and Shared Unified Shared Memory (USM), and Microsoft Windows. OpenNUMA provides a memory-management layer that allows CPU and iGPU workloads to access shared system-memory allocations through the parent application interface. This prototype implements memory pooling, allocation tracking, deallocation tracking, and thread safety using mutex locking.

Feasibility testing successfully shows OpenNUMA initialization, iGPU tensor allocation, tensor operations, and tensor aliasing through PyTorch. CPU-to-iGPU transfer testing achieved a 34.076% reduction in transfer time, while iGPU-to-CPU transfer time increased by 5.258%. General workload testing showed minimal performance differences for iGPU and neural-network workloads compared to default allocator.

The performance of OpenNUMA is nearly identical and for some cases improved compared to default allocator. Although the results can vary due to different systems, OpenNUMA shows such allocator can exist and perform effectively. However, CPU-iGPU shared tensor access could not be verified through the current PyTorch allocator interface.

Future work will investigate memory-release mechanisms, improved synchronization, and queue tracking.

Keywords: Memory Management; Custom Memory Allocator; Integrated GPU; Heterogeneous Computing; Neural Networks

1 Introduction

In computing, CPU and integrated GPU (iGPU) typically share the same system memory, allow both processors to access the available system memory. But CPU and iGPU workloads may use separate memory-management paths and potentially require data movement or additional memory-management overheads.

Although previous studies have investigated CPU-GPU shared-memory management, shared-buffer allocation, neural-network GPU memory management, and unified memory architectures, limited attention found in custom memory allocators specifically designed to enable neural-network frameworks to efficiently utilize shared system memory on older iGPU architectures without requiring a unified address space or specialized tensor-core hardware.

Older iGPU might have computing power for machine-learning tasks but generally lacks proper memory allocator to utilize high speed system-memory effectively. Most older system iGPUs intended for graphics processing rather neural network thus may contain less video memory than system memory if the motherboard provide video memory. Some iGPUs shared its video memory with system memory but not utilize unified address scheme. This may also reduce iGPU effective performance compared to unified architecture in sense of memory management overhead, memory translation overhead and other memory related overhead. In both cases, those iGPUs lack efficient memory-management mechanism for exposing and utilizing that memory for neural-network workloads.

Existing or older systems may have capable iGPUs that are underutilized for neural-network workloads. Enabling these systems to use their available iGPU computing power effectively could provide additional acceleration and performance improvement without hardware modification. This study is guided by four research questions:

- Can a custom memory allocator provide CPU and integrated-GPU access to the same system-memory without requiring explicit unified virtual address scheme and ensuring no unnecessary data copy or migration at application level, while preserving the requirements of the target neural-network framework?
- To what extent this approach reduce unnecessary CPU-iGPU data movement operations?
- What performance trade-offs will result from the proposed allocation method?
- Can existing or older systems utilize the computing power of an integrated GPU for neural-network and machine learning workloads when the system

is not primarily designed for such tasks?

To investigate these questions, this study develops a custom allocator for neural-network frameworks named OpenNUMA (Open Source Neural Unified Memory Allocator). This allocator provides shared system-memory access for CPU and iGPU neural workloads without requiring system-level unified virtual addressing. It provides a way for both the CPU and integrated-GPU to access the system-memory without explicit data copy or migration at application level.

Although similar CPU-iGPU shared-memory behavior can also be found in systems that support unified virtual addressing, OpenNUMA does not require a unified virtual address space.

The term "Open Source" implies that the source code is openly available for future modification, improvement, and adaptation. "Neural" implies the primary goal of this study, which is to enable and improve neural-network performance on older iGPU architectures, including GPUs that do not support hardware tensor cores. The term "Unified" in OpenNUMA refers to the unified access to the same system-memory allocation by CPU and iGPU.

OpenNUMA allows the CPU and iGPU to access the same underlying system-memory regardless of their virtual mappings. The CPU and iGPU may use different virtual address mappings while accessing the system-memory allocation without explicit data copy or migration at application level. The focus of this study is to test feasibility and performance impact of such custom allocator, rather than eliminate all memory-related overhead.

This study investigates the proposed allocator using PyTorch, Intel oneAPI, and Microsoft Windows on an Intel Integrated GPU as the experimental platform. Although the current implementation targets this specific platform, the allocator design is intended to be adaptable to other platforms with appropriate modifications.

2 Architecture

2.1 Conventional System Overview

A tensor is an array that represents numerical data in machine-learning frameworks. Tensors are needed because neural networks process large amounts of numerical data. They provide an efficient way to store and perform operations on those data. A CPU-iGPU workload is a type of workload where CPU and integrated GPU both perform computations on the same application data or tensors.

In most conventional CPU-iGPU workload, memory is typically allocated by the framework and underlying memory system. Although CPU and iGPU may share the same physical memory, the framework may use different virtual mappings and memory regions for CPU and iGPU. As a result, when CPU and iGPU operations need to access the same data, the framework may require data copying, migration, mapping, or synchronization between them.

CPU tensor data is stored in system RAM that can be accessed by the CPU through its virtual memory address space. Similarly, iGPU tensor data is also stored in the same RAM in a conventional computing environment. It can be accessed by the iGPU through its own virtual address space. The CPU and iGPU virtual addresses are different in general, while both can refer to the same physical pages depending on the architecture. Specialized systems may use unified virtual address space where both CPU and iGPU can have the same virtual address space.

When the iGPU needs a CPU tensor or data, the underlying framework may use copy, migration, or memory mapping to make the data accessible to the iGPU. Similarly, when the CPU needs a tensor or data from the iGPU it may use the same process of copy, migration or memory map depending on the architecture of the system.

Repeated CPU-iGPU switching can cause additional memory overhead in systems where both CPU and GPU use different virtual mappings due to address translations and mapping operations. This can happens even when the data is not changing or moving. Therefore, the conventional approach can create unnecessary data movement and memory-management overhead when CPU and iGPU repeatedly switch the workload.

2.2 Frameworks Overview

PyTorch is an open-source machine-learning framework used to develop and run neural networks. It provides support for tensor creation and manipulation, CPU and GPU computation, automatic differentiation, neural network building and training, and memory management for tensor data.

PyTorch is widely used for developing and executing neural-network workloads. Therefore, improving its memory-management efficiency can reduce unnecessary memory overhead and potentially improve the performance of CPU-iGPU workloads.

When a tensor is created, PyTorch determines its required memory. Then it requests the memory from its memory allocator and stores the tensor's data into the allocated memory. Later it uses the resulting memory for tensor operations on the CPU or GPU.

A custom or pluggable allocator provides memory allocation and deallocation for frameworks with specific requirements. PyTorch allows custom allocators to support different hardware, memory-management strategies, and specialized workloads that may not be efficiently handled by its default allocator.

Although PyTorch has a default allocator, it also provides allocator interfaces or APIs that support custom allocators for different devices. The custom allocator OpenNUMA can act as an allocator for the memory requested by PyTorch. XPU is PyTorch's interface for Intel GPUs.

PyTorch's default XPU allocator uses Device USM, where memory is allocated in GPU/device memory and intended for GPU/device access, and the CPU cannot normally access the pointer directly. Also CPU and GPU data movement may cause explicit data migration or copy.

Shared USM generally allocates memory in system RAM, which the CPU can access directly, while the GPU can also access it without an explicit copy.

OpenNUMA implements the required allocation and deallocation operations. When PyTorch requests memory, OpenNUMA allocates it and returns the pointer to PyTorch. When the tensor is released, PyTorch calls the allocator and OpenNUMA frees the memory.

The allocator acts as a bridge between PyTorch and the underlying memory system. It receives memory requests from PyTorch, obtains and manages the required memory. The underlying memory system provides and manages the actual memory resources used by the application, including allocation, deallocation, virtual-to-physical mapping, and memory access. It provides the system RAM pages that OpenNUMA manages for CPU and iGPU access.

Intel oneAPI provides the software tools and runtime support for developing and executing workloads on Intel CPUs and GPUs. Intel provides a SYCL implementation in its oneAPI software stack. OpneNUMA use the SYCL model for GPU programming and management.

SYCL provides the API model through which applications can access GPU resources. It provides a standard C++ programming model that allows GPUs to be programmed in standard C++, then the SYCL runtime translates requests into lower-level operations needed for accessing and managing GPU resources, including device execution and memory operations.

Overall, PyTorch requests memory through its XPU backend in Intel Systems. In this study, OpenNUMA used as an XPU pluggable allocator to handles these requests and manages the required memory. OpenNUMA uses Intel oneAPI/SYCL APIs to perform the required GPU-related memory operations, which are ultimately handled by the Intel GPU runtime and driver to access the iGPU. OpenNUMA does not directly manage the system RAM pages or virtual-to-physical mappings. Windows and the Intel runtime/driver handle those operations.

OpenNUMA C++ codes containing SYCL APIs is compliled by the Intel DPC++ compiler, which is part of Intel oneAPI software stack. Then the compiled code uses SYCL runtime which also implemented in the oneAPI. The SYCL runtime manages SYCL-level operations, while Intel GPU runtime handles the platform-specific GPU operations. Finally, the Intel GPU driver provides the interface between Windows and physical GPU hardware.

2.3 Proposed Design

For a conventional system, CPU and iGPU may have different underlying virtual memory mappings which depends on the OS and driver, the proposed custom allocator introduced approach so that the same allocation can access to CPU and GPU without explicit data copy or migration at the application/framework level.

Overall, OpenNUMA does not directly create the page tables itself. The OS and Intel GPU runtime/driver handle the actual mappings because direct page-table manipulation would be complex, unsafe, and highly platform-dependent.

Synchronization is handled using the PyTorch XPU and SYCL synchronization mechanisms at their respective abstraction levels. Intel runtime/driver handle synchronization at lower level. Before the CPU access data produced by the iGPU, the corresponding GPU operations are completed. Similarly, GPU operations are submitted only after required CPU-side operation is completed. This ensure correct data visibility and access ordering without requiring a separate copy of the shared memory.

After the required synchronization, the CPU or iGPU should observe the updated value in the shared memory allocation, which is one of the requirements of the proposed design. It reduces unnecessary copying or migration by allowing the CPU and iGPU to access the same memory allocation, so a separate copy of the same tensor is not required when switching between CPU and iGPU operations.

Because OpenNUMA replaces PyTorch's normal memory allocation, it must preserve PyTorch's memory requirements, so that tensors do not fail or produce incorrect results. It must preserve valid memory pointer, requested allocation size, proper lifetime, correct deallocation, CPU/iGPU accessibility, required synchronization, and other requirements.

The custom allocator needs to track allocated pointer and its size so it can safely manage the memory throughout its lifetime. It can be a table or data structure where the allocated pointer is used as the key and its corresponding size is stored as the associated value or any of the combination of these.

When memory is freed, OpenNUMA receives the pointer and size from PyTorch. Then it validates the pointer, locates the pointer and its information, release the allocated memory, and removes the pointer and size from the tracking structure. The underlying driver perform any additional memory-management operations.

The limitation of this study are that the current implementation is designed for Windows OS and Intel integrated GPUs. It depends on Intel oneAPI/SYCL, PyTorch XPU, and the Intel graphics driver. Also, OpenNUMA does not directly control OS page tables or GPU mappings. The performance may potentially be improved through more direct control of memory mappings, which is left as a subject for future research. The current implementation is limited to the PyTorch XPU allowed tensor types and memory operations evaluated in this study.

Shared memory does not eliminate synchronization overhead, which is necessary for correct access order for CPU and iGPU.

The proposed design assumes that the OS and Intel runtime/driver correctly establish and manage the required mappings for allocator run time. It also assumes that the allocation remains valid until PyTorch releases it. Required CPU-iGPU synchronization is handled through the PyTorch XPU and SYCL Synchronization mechanisms. Therefore, this study assume that the required synchronization is maintained throughout the lifetime of the allocation through these frameworks.

The aim of OpenNUMA is to reduce unnecessary CPU-iGPU data movement rather than eliminate data movement entirely.

2.4 Generic Pseudocode

Algorithm 1 Custom Allocator's Primary Operations

Input:
Operation
Size
Pointer
if Operation = Allocate AND Size is valid **then**
    Input: Size
    Validate the requested size.
    Check into allocator cache memory for any proper sized block available.
    Return if such memory block found.
    Change returned memory block record from free to acquired.
    Allocate new memory if cache miss occur.
    Establish CPU and iGPU access to the same allocation.
    Record pointer and allocation size in a table.
    Return pointer to main application.
else
    Return failure.
end if
if Operation = Free AND Pointer is valid **then**
    Input: Pointer
    Validate pointer.
    Locate allocation in allocation table using the pointer.
    Change allocation state from acquired to free block.
    Release some allocation if cache memory consumption become large in size.
    Return success to main application.
else
    Return failure.
end if

Algorithm 2 Custom Allocator's Supporting Operations

Initialize allocation tracking table.
Validate allocation requests and reject zero, invalid, or excessively large allocation requests.
Track each allocated pointer and its corresponding size.
Handle failures and exceptions.
Perform required synchronization when necessary.
Release remaining allocations and allocator-owned resources during the allocator destruction.

3 Implementation

3.1 Development Environment

Component	Software/Hardware	Version
Operating System	Microsoft Windows	10
Build Environment	Visual Studio Build Tools	2026
C++ Compiler	ICX-CL	2026.1.0
iGPU Development	Intel oneAPI Base Toolkit	2025
Graphics Driver	Intel Graphics Driver	32.0.101.7088
Programming Language	Python	3.14.5
Machine-Learning Framework	PyTorch	2.13.0+xpu
CPU	Intel Core i3-12100	12th Gen
GPU	Intel Integrated GPU	UHD Graphics 730

Table 1: Development Environment

PyTorch was tested within a Python virtual environment (venv).

3.2 Custom Allocator Implementation

The custom allocator is implemented in C++ using the Intel oneAPI SYCL programming model. On Windows, the allocator is compiled using the Intel oneAPI icx-cl compiler driver with SYCL support enabled through the -fsycl option. The Microsoft MSVC toolchain provides the supporting build environment, Windows SDK, system libraries, and linking components required to build the application.

PyTorch uses the custom allocator by calling the provided allocation and deallocation functions. The allocator does not validate pointers or allocation sizes for most cases. It trusts the values provided by PyTorch. For stability, OpenNUMA checks for null pointer or queue, zero size in function.

After installation, the allocator is owned and managed by PyTorch. Therefore, manual destruction of the allocator is not preferred.

Memory allocation does not require explicit waiting or synchronization. The allocator preforms the allocation when requested by PyTorch. Read/write ordering and operation are handled by PyTorch. The custom allocator does not explicitly manage the ordering.

Deallocation may require device-wide synchronization because memory must not be freed while any queue or stream is still accessing it. A stream is an independent queue, so independent work submitted to different streams may execute on same tensor independently. PyTorch does not provide the custom allocator sufficient information from its API to determine when all streams have

finished using a particular allocation. Thus this design trusts PyTorch about memory deallocation decision as PyTorch allocate/deallocate all tensors.

Although the design implements a mutex lock for thread safety, this can reduce performance because multiple threads can no longer execute the mutex-protected section of the custom allocator simultaneously.

Both allocation and deallocation in OpenNUMA maintain a memory pool or cache. Two lists are implemented for this purpose. When an allocation is requested, the allocator checks whether it have valid size and parameters, and then tries to find an available suitable block in memory pool. If a suitable block is available, OpenNUMA returns the pointer from free-memory list. If no suitable block available, It allocates memory using Intel Shared USM and records the pointer in the in-use map for future tracking with its size.

When deallocation is requested, OpenNUMA check whether the pointer exists in the in-use list. If it is found, the pointer with its size is moved to the free pool for reuse. If the pointer is not found in the in-use pool, OpenNUMA treats the pointer as invalid for this allocator.

OpenNUMA implements exception handling and pointer, queue and allocation-size checks to improve the stability. However, there is currently a limitation, memory is never freed. Instead, freed allocations remain in the memory pool for reuse. This improves allocator performance by enabling memory reuse, but it can also increase memory consumption because cached memory remains allocated. A memory-release mechanism is planned for future version.

Some PyTorch libraries, such as `c10_xpu`, can provide synchronization mechanism. However, using them make the allocator dependent on PyTorch libraries. Therefore, it will remain study area for future work.

For a standalone custom allocator, this implementation may use oneAPIs `wait_and_throw()` before freeing the memory. This provides synchronization and ensures that all work on that queue has completed before the memory is released.

However, calling this function every free operation can introduce significant overhead because it waits for all outstanding work on the queue, including works that are not related to the memory being freed. This cause an overhead each time and may cause performance drop. This can be improved by using an allocation tracking mechanism which the current implementation of OpenNUMA have. When PyTorch call free function, the pointer is intended to be safe for the allocator to release. For this reason, OpenNUMA only ensure the pointer to be freed in its list of allocation. Current implementation does not verify GPU level operation track on memory being freed.

Allocation tracking within custom allocator also introduces performance overhead. However, this overhead may be lower than performing device-wide synchronization on every free call. But XPUPluggableAllocator does not provide callback to custom allocator when a tensor begins using a different stream. Therefore, a simple allocation-tracking table cannot reliably track stream usage. However, this implementation trusts PyTorch's deallocaion decisions.

3.3 Framework Integration

PyTorch provides an XPUPluggableAllocator class for loading a custom XPU memory allocator and `torch.xpu.memory.change_current_allocator()` function for changing default allocator to a custom allocator as the active XPU allocator. The current allocator must not already have been initialized or used when `change_current_allocator()` is called [1].

After importing PyTorch, An instace of the XPUPluggableAllocator class must be defined with OpenNUMA specified as pluggable allocator. Then `change_current_allocator()` function is called to change the active allocator.

For the basic pluggable allocator interface, two callback functions, malloc and free, are called by PyTorch. The required parameters depend on the operation. The malloc callback receives the requested size, device, and queue, while the free callback receives the pointer, size, device, and queue.

PyTorch calls the implemented malloc() and free() callback functions. Then the callback functions receive a pointer to SYCL queue named as sycl::queue*, from PyTorch. A SYCL queue is an object used to submit work and memory operations to the Intel GPU. It associated with information such as target device, context, queue properties and other data.

OpenNUMA uses SYCL/oneAPI memory functions with the provided queue to allocate shared USM memory that can be accessed by the CPU and GPU. The Intel driver handles the required GPU-side operations.

There are two possible way of implementing this idea. For simplification, the first approach is for OpenNUMA to allocate CPU-accessible memory and gets a CPU virtual pointer. OpenNUMA then passes that pointer in a compatible format to SYCL/oneAPI using appropriate functions to make the memory usable by iGPU. Then the Intel runtime and GPU driver establish the iGPU-side mappings. After this, both the CPU and iGPU should be able to access the same allocation.

The second approach for OpenNUMA is to use Intel oneAPI SYCL and shared USM (Unified Shared Memory) to provide CPU and iGPU access to the same memory allocation.

shared USM is a SYCL memory type in which the memory is allocated by runtime and can be accessed by both the CPU and a supported SYCL device.

Function `sycl::aligned_alloc_shared()` allocated memory in current OpenNUMA implementation. Intel Implementation may migrate between the host and the device for faster access. It is done by the Level-Zero driver. Intel expresses no explicit copy is necessary for synchronizing between the host and the device. The allocation can be accessed by the host and the specified device only [2].

This method allows the compatible devices like iGPU to access a memory allocation without requiring an explicit copy or migration to device/iGPU memory. A SYCL queue is still used here to submit and synchronize iGPU operations.

A SYCL device is a computing device on which SYCL can submit and execute work. A SYCL device is not limited to an Intel iGPU and can be any

supported CPU, GPU, or other accelerator that implements SYCL. Therefore, this implementation approach makes OpenNUMA flexible enough to be ported to other systems with appropriate modifications.

Because of these advantages, in this current implementation OpenNUMA uses the second approach based on shared USM.

PyTorch has device-specific allocator support. This study only focus over Intel iGPUs and therefore it uses PyTorch XPU backend. For this reason, the design of OpenNUMA can be portable and potentially be adapted to other GPU platforms in PyTorch; however, the current implementation is device specific to Intel integrated GPUs.

The allocator is not responsible for supporting tensor types. It provides the underlying memory allocation or deallocation interface used by the XPU backend, while the PyTorch's XPU backend determines whether a particular tensor types or operations are supported or not. PyTorch may not use XPU for a particular tensor or operation if it is not supported by the XPU backend, or if the tensor is explicitly placed on another device. OpenNUMA is responsible for memory allocation, deallocation. While tensor-type, lifetime, synchronization and operation support is determined by the PyTorch XPU backend and the target Intel GPU platform.

Therefore, current implementation of OpenNUMA only affects allocations that go through the XPU allocator. It does not control CPU-only tensors or operations that PyTorch handles outside the XPU backend.

XPUPluggableAllocator takes the dynamic library name, allocator function name, and deallocator function name as parameters. PyTorch loads the specified library and calls those functions whenever it needs memory allocation or deallocation.

3.4 Implementation Limitation

Queue tracking could also be investigated to reduce the need for device-wide synchronization during every free operation if implementation demand for it. However, the current XPUPluggableAllocator API does not provide sufficient information to implement a simple and reliable queue tracking mechanism. Thus a workaround version is implemented using map from C++ Standard Library.

The current implementation also does not explicitly free allocated memory during free operation, instead it removes the allocation from the in-use list and moves to free list. As a result, high memory consumption can occur in current prototype version.

4 Results and Discussion

4.1 Feasibility Tests

Test	Proposed Allocator (Pass/Fail)
XPU Allocator Initialization	Pass
iGPU Tensor Allocation	Pass
Tensor Operations on iGPU	Pass
Support for CPU-iGPU Shared Allocation	Not verified
Tensor Aliasing Support	Pass

Table 2: Feasibility of Proposed Method

In this study, XPU is the iGPU of the testing system and all tests including above are carried out in Python virtual environment (venv).

In simplified sense, PyTorch maintains CPU tensor to CPU memory and XPU tensor to XPU memory. Although for Shared USM the memory is the same allocation that can be accessed by both CPU and GPU but PyTorch maintains separate device rules for CPU and XPU tensors. Testing for zero copy/migration in either direction through normal PyTorch tensors requires a CPU/XPU transfer or copy.

CPU-iGPU shared allocation and bidirectional access can be verified at the SYCL/Shared USM level. However, none of these tests prove that PyTorch uses the same allocation or allow bidirectional CPU/iGPU access through its tensor APIs. These SYCL level tests only prove that Shared USM use same memory allocation and that is already specified from Intel.

OpenNUMA only maintains tensors that are allocated through XPU allocator interface. Therefore, such criteria indicates an API/integration limitation, not a failure of OpenNUMA or Shared USM.

4.2 Memory Tests

Workload	Baseline Time	Proposed Time	Reduction (%)
CPU to iGPU Transfer	174.015 ms	114.717 ms	34.076%
iGPU to CPU Transfer	14.720 ms	15.494 ms	-5.258%

Table 3: Memory Migration Test Results

Reduction(%) = [(Baseline - Proposed)/Baseline] x 100

The proposed allocator reduces CPU-to-iGPU transfer time by 34.076%, indicating improved performance for this workload. However, iGPU-to-CPU

transfer time increases by 5.258%, resulting in a negative reduction. This behavior can caused by Shared USM implementation, where Intel might handle data differently by depending which side is accessing the data. Also, the custom allocator have mutex lock and bookkeeping, which can add some overhead.

4.3 Performance Tests

Test	Baseline Time	Proposed Time	Improvement (%)
CPU Workload	25.308 ms	31.920 ms	-26.126%
iGPU Workload	92.365 ms	91.615 ms	0.811%
CPU to iGPU Alternating Workload	212.409 ms	209.374 ms	1.428%
Neural-Network Workload	500.973 ms	500.559 ms	0.082%

Table 4: Performance Test Results

Reduction(%) = [(Baseline - Proposed)/Baseline] x 100

The results shows that the proposed allocator has minimal impact on most workloads. The CPU workload becomes 26.126% slowe, likely due to mutex locking and bookkeeping overhead. In contrast, iGPU, alternating CPU-iGPU, and neural-network workloads show only small improvements of 0.811%, 1.428%, and 0.082% respectively.

5 Conclusion

This study developed OpenNUMA as an extension to provide a practical way for neural networks and machine-learning applications to utilize an integrated GPU on systems that are not primarily designed for such workloads. The current implementation of the proposed approach uses PyTorch's XPU backend with Intel oneAPI SYCL and Shared USM to provide both CPU and GPU accessible memory and integrate the iGPU into the application's memory-management path.

The prototype successfully demonstrated XPU allocator initialization, iGPU tensor allocation, tensor operations, and tensor aliasing through PyTorch. The Memory-transfer testing showed a 34.076% reduction in CPU-to-iGPU transfer time, while iGPU-to-CPU transfer time increased by 5.258%. Testing showed minimal performance differences for iGPU and neural-network workloads, while CPU-only workloads experienced additional overhead from allocator bookkeeping and mutex locking. The alternating workload benefited slight improvement. The test results shows difference between PyTorch's default allocator and OpenNUMA. Shared CPU-iGPU tensor access could not be fully verified due to current PyTorch allocator Interface.

Although test results can vary due to operating-system background processes, iGPU driver versions, or differences in PyTorch and oneAPI implementations, the results demonstrate that the proposed custom allocator can perform effectively on older iGPU architectures and GPUs without specialized tensor-core hardware.

The study demonstrates feasibility of using an integrated GPU as an accelerator for machine-learning workloads through a custom memory-management extension, including systems where the iGPU was not primarily intended for such application. Thus this study suggests that existing or older systems with compatible integrated GPUs may be able to leverage their available iGPU computing power for neural-network workloads.

Future work will focus on memory-release mechanisms, improved synchronization, and queue tracking for further improve reliability and performance.

6 Acknowledgment

The author has no acknowledgments to declare.

7 Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

8 Conflict of Interest Statement

The author declares that there are no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

9 Data Availability Statement

The data supporting the findings of this study are available from the corresponding author upon reasonable request.

The source code developed in this study will be made publicly available through popular open-source repository under the project name and the author's name. The repository will include the custom extension implementation and supporting code required to reproduce the experiments.

10 CRediT authorship contribution statement

Md. Muminul Islam: Conceptualization, Methodology, Software, Investigation, Data curation, Formal analysis, Visualization, Validation, Writing — orginal draft, Writing — review & editing.

References

[1] PyTorch Foundation. torch.xpu, 2026. PyTorch 2.13 Documentation.

[2] Intel Corporation. Unified shared memory allocations, 2025. oneAPI GPU Optimization Guide.