# Programming Model Extensions for General-Purpose Processing-In-Memory

Hyesun Hong*, Lukas Sommer†, Bongjun Kim*, Mikhail Kashkarov‡, Kumudha Narasimhan†
Ilya Veselov‡, Mehdi Goli†, Jaeyeon Kim*, Ruyman Reyes Castro§, and Hanwoong Jung*

***Abstract*—The performance of many applications is limited by the available memory bandwidth. One approach to improve the performance of such memory-bound applications is to move the computation closer to the required data. Processing In Memory (PIM) integrates computational units directly with the memory. To enable PIM technology in widely used programming models, we propose extensions to OpenMP and OpenACC, two examples of directive-based programming models, as well as SYCL. The extensions are designed to be portable across many existing and future parallel computing devices and platforms, making PIM technology widely available.**

**For the extensions, we propose an end-to-end compilation framework based on several steps of abstraction and progressive lowering. To achieve this goal, we formulate a new PIM IR and conduct optimizations tailored to hardware characteristics. By using AMD MI100 GPU with PIM-enabled HBM2 memory, we observe a performance improvement of 1.2-2.1 times for representative examples and a real high-performance computing (HPC) application compared to the same GPU without PIM-enabled memory.**

***Index Terms*—Processing in memory, Programming model, MLIR, Compiler directive**

## I. Introduction

The performance of processor-oriented systems such as CPUs, GPUs, and accelerators is still increasing thanks to the parallelization and specialization of applications, and architectural improvements such as an increasing number of cores that are positioned closer to one another. Yet, for the last decades, memory bandwidth has not been growing at the same rate and is severely limited by physical constraints. This makes it even more important to not only consider compute latency and throughput but also memory bandwidth and latency.

The typical architecture of most CPUs or GPUs requires reading data from memory, performing calculations on it, and then writing the results back to memory. A new approach, however, where computation occurs where data resides, has arisen to address memory bottlenecks. Processing In Memory (PIM) integrates computational units directly with the memory to improve the performance of bandwidth-intensive workloads and to improve energy efficiency by reducing data movement between computing and memory units. Therefore, it can ease the burden of constrained memory bandwidth.

For this work, we are going to focus on architectures that couple PIM with an existing accelerator. As an example, the coupling of PIM-enabled high-bandwidth memory (HBM) with GPUs allows to compute kernels executing on the GPU to leverage PIM.

The productivity of developers is a key factor to consider when designing new hardware architectures, and researchers have presented many approaches, focusing mostly on two areas (1) domain-specific libraries, such as BLAS [1], and (2) general programming models for new architectures, such as OpenACC [2], CUDA [3], and SYCL [4]. Hardware vendors must offer a variety of options to developers, to address the various requirements that they have for their domains if they want to gain widespread adoption of their hardware. Furthermore, open standard programming models help the adoption of new architectures given they are known to the developers already, and do not bind them to a specific vendor.

We identify the key requirements for a PIM programming model based on the current state of the art in programming models and developer ecosystem trends as follows:

- R1) PIM operations should be integrated seamlessly into the existing programming models to make them available to users in an easy-to-use and comprehensible manner.
- R2) The design of the programming model should be vendor-agnostic rather than specific to one accelerator or software development kit (SDK).
- R3) A user without hardware expertise should be able to run PIM operations and non-PIM kernels within the same application. In addition, its programming model allows a combination of regular non-PIM and PIM operations in the same kernel to apply incremental development.
- R4) PIM operations should be integrated into the existing runtime, e.g., into SYCL's dependency tracking.
- R5) The underlying intermediate representation (IR) should be designed to support existing optimization methods and to facilitate the integration of new optimization approaches.

The first three requirements are related to the extended programming model, while the remaining two focus on its implementation. To the best of our knowledge, the new PIM technology does not yet have a standard programming model. For instance, the user of UPMEM [5], which is one of the well-known in-memory processing devices in academics, needs to specify the application with UPMEM-specific features at the task level. Moreover, the software stack of Samsung's FIMDRAM [6] is proprietary closed-source.

In this paper, we extend the existing heterogeneous programming models to support PIM operations, to fulfill the need for general workload support. In addition, we propose

*Samsung Advanced Institute of Technology, Korea, †Codeplay Software Ltd, United Kingdom, ‡Samsung Electronics, Russia, §Intel, United Kingdom
Corresponding author: Hyesun Hong (hyesun.hong11280@gmail.com)

```
1   subroutine saxpy(n, a, x, y)
2        real :: x(:), y(:), a
3        integer :: n, i
4        do i=1,n
5             y(i) = a*x(i)+y(i)
6        enddo
7   end subroutine saxpy
8
9   integer::N = 1 << 20
10
11  ! Perform SAXPY
12  call saxpy(N, 2.0, x_d, y_d)
13
14
```

(a)

```
subroutine saxpy(n, a, x, y)
     real :: x(:), y(:), a
     integer :: n, i
     !$acc kernels loop
     do i=1,n
          y(i) = a*x(i)+y(i)
     enddo
     $!acc end kernels loop
end subroutine saxpy

integer::N = 1 << 20

! Perform SAXPY
call saxpy(N, 2.0, x_d, y_d)
```

(b)

```
1   std::vector<float> h_X(len, xval);
2   std::vector<float> h_Y(len, yval);
3
4   queue q{cpu_selector_v};
5
6   const float A{aval};
7   {
8      buffer<float> d_X{h_X};
9      buffer<float> d_Y{h_Y};
10
11     q.submit([&](handler& cgh){
12        auto X = d_X.get_access(cgh, read_only);
13        auto Y = d_Y.get_access(cgh);
14
15        cgh.parallel_for<class axpy>(len, [=](id<1> i) {
16          Y[i] = A * X[i] + Y[i];
17        });
18     });
19  }
```

(c)

Fig. 1. SAXPY example coded in (a) Fortran, (b) Fortran with OpenACC, and (c) SYCL

an end-to-end compilation framework based on multi-level intermediate representation (MLIR) [7].

By extending multiple existing high-level programming models, we enable even non-expert users to leverage PIM operations, allowing a wide range of workloads to benefit from PIM. The design of the extensions as well as the architecture of the underlying compiler infrastructure fulfill the above-mentioned requirements.

The viability of the proposed methodology is validated with experiments with two linear algebra and high-performance computing (HPC) workloads as case studies. In particular, we focus on how to specify the operation and the resultant intermediate representation and have an effect on the performance through optimization. To validate the productivity of software development, we compare the number of lines to be modified to utilize PIM hardware.

## II. Background

### A. Programming Model

There are four approaches to specify a parallel program by offloading the most computationally intensive part onto accelerator devices. The first one is to use an internally optimized library such as BLAS [1], MAGMA [8], or SLATE [9]. While BLAS [1] and LAPACK [10] are a basic collection of basic linear algebra operations, ScaLAPACK [11] is a variant of LAPACK designed for distributed memory parallel systems using domain decomposition techniques. More recent developments such as MAGMA [8], or SLATE [9] aim to extract the full performance potential and maximum scalability from modern, many-node machines with large numbers of cores and multiple hardware accelerators per node.

For the second approach, the developers identify which areas of code to accelerate by giving simple hints, known as directives, to the compiler. Typical examples include OpenMP [12] and OpenACC [2]. OpenMP started as a way to program multi-core CPUs, the accelerator offload model was later introduced in versions 4.0 and later. OpenACC started as an alternative to OpenMP 3.0 and first introduced GPU support. Both directives share the common goal of providing programmers with a high-level approach to heterogeneous programming [13]. To demonstrate the different programming models, SAXPY (Single-precision $A \times X + Y$), a combination of scalar multiplication and vector addition, is used as a common example. The complete SAXPY code in Fortran is given in Fig. 1(a). To perform on the GPU, the user needs to add compiler directives for OpenACC in Fig. 1(b).

Third, the most commonly used approach for performance optimization is developing custom parallel algorithms with extended languages such as CUDA [3] and OpenCL [14]. Especially, state-of-the-art in high-performance artificial intelligence is driven by NVIDIA GPUs [15], so there are translators available that can convert handwritten CUDA code into HIP [16] or other languages.

Finally, developers have the option to utilize C++ template-based high-level standard programming models such as SYCL [4] and Kokkos [17]. Leveraging the expressive power of modern C++, SYCL provides an easy-to-use single-source programming model with a higher level of abstraction than e.g. OpenCL, and can target a wide range of heterogeneous platforms. Thanks to these features, an increasing number of companies are starting to provide SYCL implementations, some of which support a variety of acceleration API back-ends. Fig. 1(c) shows a SAXPY code in SYCL to explain SYCL features.

Similar to OpenCL [18], SYCL also allows vendors to extend the programming model. While vendor extensions such as [19] and [20] offer a lightweight process to create an extension and are often focused on a vendor's platform, official Khronos (KHR) extensions follow a more formalized process and require the participation of multiple vendors.

### B. Multi-level Intermediate Representation (MLIR)

Multi-Level Intermediate Representation (MLIR) [7] is a flexible and extensible compiler infrastructure, ranging from backend code generation and orchestration of heterogeneous systems to high-level language semantics of programming languages and domain-specific frameworks. It serves as an intermediate representation (IR) framework for optimizing and transforming programs across various programming languages and hardware targets. MLIR enables efficient and portable code generation by providing a unified representation for different dialects and allowing seamless integration with existing compiler tools. It offers a modular and hierarchical approach, supporting high-level abstractions as well as low-level hardware-specific optimizations.

Some MLIR dialects that we have utilized in our work are provided below:

1) Affine dialect [21]: provides a set of operations and transformations for working with loop nests and affine

functions. It enables loop optimizations and transformations such as loop tiling and loop fusion.

2) Linalg dialect [22]: is designed to represent and optimize linear algebra operations. It provides a powerful tool for advanced optimizations for loop vectorization, mapping to parallel and reduction loops, and lowering to loops / library calls / intrinsic.
3) GPU dialect [23]: provides middle-level abstractions for launching GPU kernels following a programming model similar to that of CUDA or OpenCL. Its goal is to abstract away device- and driver-specific manipulations to launch a GPU kernel and provide a simple path toward GPU execution from MLIR. And it can be lowered to specific target-specific dialects such as “amdgpu” for AMD GPUs or “nvgpu” for NVIDIA GPUs.
4) ACC dialect [24] / OMP dialect [25]: model the construct from the OpenACC 3.1 and OpenMP directive language, respectively.
5) Transform dialect [26]: provides operations that can be used to control transformations of the payload IR using a different portion of the transform IR. Therefore, it plays a crucial role in enabling efficient and scalable code transformations within a compiler framework.

### C. Processing-In-Memory (PIM)

Computer architects are adopting alternative system models, such as processing near memory (PNM) and processing in memory (PIM), to address challenges related to data movement. This shift away from the traditional Von Neumann architecture aims to improve overall system efficiency by bringing processing capabilities closer to the memory, reducing data transfer bottlenecks, and enabling more efficient computation within the memory itself. While PIM integrates computational units into the memory itself, PNM focuses on placing processing units near the memory modules to improve performance.

Extensive research has been conducted in the field of PIM. UPMEM [27] provides products that enable researchers to run real-world applications, allowing them to experiment and evaluate the benefits of PIM. Additionally, Samsung [6] and SK Hynix [28] have introduced specialized PIM designed significantly for machine learning tasks. These systems utilize HBM2 and GDDR6 DRAM standards, offering support for high-performance operations in the range of TFLOPS [29]. Especially, in [6], the I/O boundary of a bank is equipped with a PIM execution unit comprising SIMD floating-point units, command register files (CRF), general register files (GRF), and scalar register files (SRF). In addition, it supports both standard DRAM and PIM modes for versatility. In DRAM mode, it is the same as usual DRAM. However, in PIM mode, PIM execution units across all the banks concurrently respond to a standard DRAM column command such as read and write command. Therefore, it executes one wide-SIMD operation with deterministic latency in a lock-step manner.

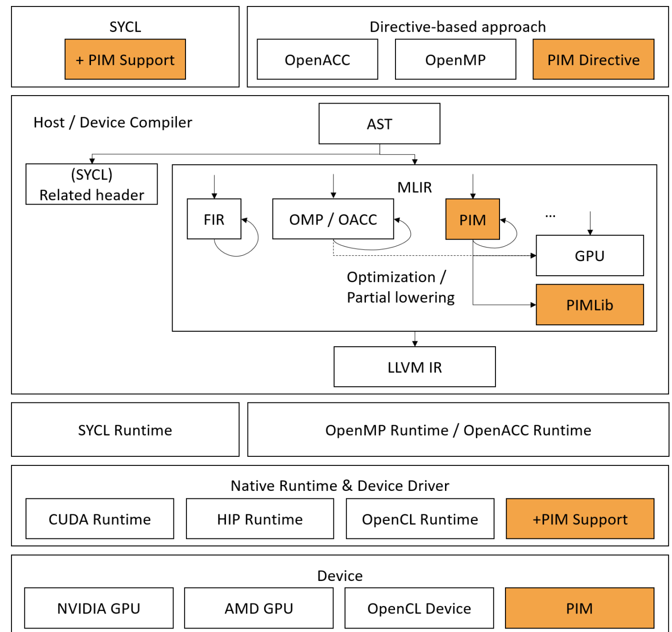

Fig. 2. Overview of the our proposed software stack for PIM

## III. Overview

The overall flow of the proposed software structure shown in Fig. 2 can be understood as the framework from a directive-based approach and a template-based approach to executable binary. The application is written using SYCL extensions for PIM support or PIM directive in Fortran/C. In this section, we explain our extended models and proposed MLIR with a simple element-wise addition example.

As shown in Fig. 3(a), the user selects a SYCL device supporting PIM operations via `pim_selector`. To facilitate standardization as a vendor-neutral design, we extend the transform function from standard C++. It applies the binary operation (plus) to each pair of elements in `in1` and `in2`, and stores the results (`out`), preserving the original order of elements. To express the same behavior, there are several options, such as explicit vector operation or automatic inference of PIM operations from scalar user code. However, explicit vectorization with a fixed vector width would hinder the portability of applications and automatic inference would place a huge burden on compiler implementation. Fig. 3(b) illustrates the Fortran code with the PIM directive. It allows the user to maintain existing sequential code and to annotate to expose parallelism.

The applications for PIM go through various levels of optimization by generating MLIR or related header files inside the compiler. For SYCL, while implementation details may diverge, our approach allows the SYCL runtime to directly invoke the PIM library for host-initiated PIM operations. For Fortran code with PIM directive, Flang parses and produces a series of MLIR, which is the mixture of FIR, PIM IR, and ACC IR [30]. Our Flang is based on Flacc to support the existing directive together to satisfy requirement R3.

Fig. 3(c) depicts PIM IR which is aware of the set of operators supported by PIM. It is a bridge between the programming

```
1  queue q{pim_selector{}};
2  {
3     buffer<half> b_a{a, range{dataSize}};
4     buffer<half> b_b{b, range{dataSize}};
5     buffer<half> b_c{c, range{dataSize}};
6     q.submit([&](handler &cgh) {
7        auto acc_a = b_a.get_access<access_mode::read>(cgh);
8        auto acc_b = b_b.get_access<access_mode::read>(cgh);
9        auto acc_c = b_c.get_access<access_mode::write>(cgh);
10       cgh.parallel_for<class Kernel>(nd_range<1>
11          {range<1>{10}, range<1>{2}}, [=](nd_item<1> item) {
12            auto groupID = item.get_group(0);
13            auto* a = &acc_a[groupID * 128];
14            auto* b = &acc_b[groupID * 128];
15            auto* c = &acc_c[groupID * 128];
16            joint_transform<64>(item.get_group(), a, b, c,
17                                sycl::plus<>(), 2);
18       });
19    });
20 }
```

(a)

```
1  subroutine add(a, b, c, n)
2     real :: a(:), b(:), c(:)
3     integer :: n, i
4     i = 0
5     !$pim parallel loop copyin(a, b) copyout(c)
6     do i = 1, n
7        c(i) = a(i) + b(i)
8     end do
9  end subroutine
```

(b)

```
1  pim.parallel copyin(%a, %b) copyout(%c) {
2     %1 = pim.create_buffer %a
3     %2 = pim.create_buffer %b
4     %3 = pim.create_buffer %c
5     %p1 = pim.load %1
6     %p2 = pim.load %2
7     %p3 = pim.add %p1, %p2
8     %3 = pim.update %p3
9     pim.delete_buffer %1
10    pim.delete_buffer %2
11    pim.delete_buffer %3
12    pim.yield
13 }
```

(c)

Fig. 3. Code snippet of (a) SYCL extensions, (b) PIM-directive and (c) PIM dialect for element-wise addition example

model and actual PIM instruction set architecture (ISA). Multiple levels of IR for each operation are provided, which can be adjusted based on different optimizations provided by the hardware backend. For instance, PIM IR can be transformed into the corresponding IR for the PIM SDK library based on the specific backend. Alternatively, it can undergo partial lowering into PIM IR for generating PIM kernels and the dialect of the underlying device like a GPU. Furthermore, by incorporating hardware characteristics as parameters, the IR can be regenerated, enabling compatibility with various backend devices. This approach also fulfills requirement R5. After additional analysis and rewriting passes, it emits LLVM IR dialect which declares operations related to the existing runtime functions. Finally, the LLVM IR is compiled and linked with the runtime libraries.

### A. Related Work

In the field of compiler research, there have been studies focusing on MLIR that aim to expand its functionality for effortless integration of new devices. Additionally, there are research efforts aimed at enhancing the expressiveness of MLIR, enabling a broader range of optimizations to be explored.

To enhance the expressiveness of new devices, CINM [29] proposed a general end-to-end compilation infrastructure for heterogeneous computing-in-memory (CIM) and compute-near-memory (CNM) devices. Similar to our design, it uses MLIR rewriting and introduces reusable abstractions and components. However, it begins with the Linalg dialect as a starting point and then represents the common aspects of CIM and CNM as separate dialects. It also further abstracts the specific characteristics of CIM and CNM into CIM dialect and CNM dialect, respectively. Subsequently, it establishes the connection between these dialects and the vendor-specific intrinsics. This enables users to effectively integrate the intrinsic operations provided by different vendors, allowing for more flexible and efficient code generation in the context of CIM and CNM. We have comparable goals for these reasons, however CINM pipeline differs from ours in that CINM pursues the lowering process from only AI applications. In addition, we assume PIM needs to be linked and operated with other device. For example, a PIM coupled with a GPU requires operating the PIM at the GPU's warp level.

To achieve high-performance GPU code generation, [31] extended the existing GPU dialect to support the utilization of tensor cores and performed optimization techniques to maximize the reuse at different levels of the memory hierarchy. It includes considerations such as creating and placing fast memory buffers, applying loop tiling, enabling vectorization, detecting parallel loops, and determining the placement of synchronization barriers.

Graphene [32] also introduced an IR specifically designed for efficient tensor computations on GPUs. It serves as a low-level target language for tensor compilers. In Graphene, tensors and threads can be decomposed hierarchically into tiles, enabling optimized tensor computations to be expressed as mappings between data and thread tiles. In particular, tensor computations are represented using specs, which are self-contained blocks of computation. The gradual lowering of operations in the PIM dialect is analogous to this feature.

SYCLops [33] is a converter that can translate SYCL-specific LLVM IR to MLIR. It provides the capability to perform both target and application-specific optimizations within the same compilation pipeline. This study differs from our proposed approach where the goal is to directly go from the programming model to MLIR.

SYCL-MLIR [34] recently introduced an overview of the architecture for a SYCL compiler based on MLIR. It demonstrates how MLIR can be leveraged for better optimization of SYCL code during compilation.

In summary, there has been a lack of research that integrates general-purpose programming models to compilers for PIM.

## IV. Proposed Programming Model

To fulfill the need for general workload support and for accelerating PIM, two approaches, one for a directive-based approach and the other for a template-based approach, are devised to make it easy for application or library developers to use. In this section, we investigate both approaches which extend the existing heterogeneous programming models to support PIM operation.

```
1  queue q{ext::pim_selector{}};
2  half a = 2.0;
3  half x[dataSize];
4  half y[dataSize];
5
6  // y[i] = a + x[i]
7  {
8     buffer<sycl::half> bX{x, range{dataSize}};
9     buffer<sycl::half> bY{y, range{dataSize}};
10    q.submit([&](ext::pim_handler &ph) {
11       ext::pim_accessor pAccX{bX, ph, sycl::read_only};
12       ext::pim_accessor pAccY{bY, ph, sycl::read_write};
13       ph.elementwise_add(pAccY, a, pAccX);
14    });
15 }
```

Fig. 4. SYCL extension for specific PIM hardware

### *A. SYCL Extension*

For the first attempt at a SYCL extension for PIM, we follow the rather lightweight process for vendor extensions. Its design is currently bound to a specific device and SDK to enable experimentation with actual hardware. To overcome these limitations in the future, we propose the extensions in two stages. First, the extension bound to specific hardware is elaborated in Fig. 4, and then a more general method is proposed in Fig. 3(a).

The user should be able to select a SYCL device that supports PIM operation, analog to general SYCL applications. As the memory on the device is partitioned between regular memory and PIM-enabled memory, PIM is modeled as a separate SYCL device. In this case, modeling PIM and the PIM-connected GPU as a single device would violate the SYCL platform model, as part of the memory is not accessible to the PIM computation units. Line 1 creates an SYCL queue for a PIM device selected by the specialized device selector that uses the `pim` device aspect to select a PIM device.

For PIM execution, we provide two options: A vendor-specific extension for host-initiated PIM, extending SYCL handlers and accessors, and a more generic approach that proposes new group functions.

For the first approach, to make existing SYCL memory objects such as buffers available for PIM execution, the dedicated `pim_accessor` is introduced. Similar to the regular SYCL accessor, the `pim_accessor` fulfills two roles: It requests access to the buffer for device execution, in this case on the PIM device. It also declares a data dependency on that buffer, integrating into the existing automatic directed acyclic dependency graph execution of SYCL. As PIM operations in this design are host-initiated and the `pim_accessor` is therefore only used in host code, its implementation is more lightweight than the regular SYCL accessor. The API for host-initiated PIM operations is inspired by existing functions for explicit memory operations (e.g., `fill` or `copy`) in the SYCL handler.

In addition, we propose a separate `pim_handler` to provide additional PIM operations and to highlight the semantic difference in sending commands to the attached PIM blocks. The highlighted section (line 10-14) in Fig. 4 represents the part that we have newly extended. It can be observed that its handler can directly perform element-wise addition operations as memory request operations.

However, the first approach has four limitations; 1) when different manufacturers provide numerous different operations, it becomes challenging to extend in a consistent manner. 2) If the entire memory of the device is composed of PIM, it is not natural to create a separate queue for execution. 3) Because GPU and PIM are treated as separate devices, mixing PIM and regular GPU computation in the same device kernel is currently not possible. As a result, this approach leads to the problem of forming too many command groups for complicated applications. 4) The concept of unified shared memory supported by SYCL has not been taken into consideration. Therefore, this simple approach is only valuable as a vendor extension.

As a more general extension approach, we propose to specify PIM operations using new group functions, as shown in Fig. 3(a). Since the SYCL specification already defines several group functions, adding additional group functions only requires minimal learning effort for users. Additionally, they can easily be combined with other device codes in the same kernel, matching R3 in the above-mentioned key requirements. For example, when GPU contains PIM, this approach allows to integrate PIM operations directly into GPU kernels. Therefore, no host-GPU round-trip would be required to switch between GPU and PIM execution.

To enable simple addition and multiplication operations in memory blocks, a proposal can be devised using a group function called `joint_transform`, which is similar to utilizing the transform function in C++ syntax. Additionally, similar to the `inner_product` function in C++, we introduce `joint_inner_product` to express inner products intuitively. For reduction operations, the existing `joint_reduce` in SYCL can be utilized. Although the example in Fig. 3(a) (line 16-17) use only a group function in a command group, these group functions can be expressed alongside other computation operations within a command group, allowing for more efficient specifications. Lastly, group functions can include accessors as arguments, but they can also accommodate unified shared memory as an argument. It makes them suitable for a broader memory model.

Vector operations included in the SYCL specification seemed like a natural fit because the PIM block internally uses SIMD execution. However, the SYCL specification defines horizontal operations for vectors, which are challenging to implement efficiently in distributed compute units in PIM. Also, the vector size explicitly stated in the code would need to match the alignment requirement, restricting the portability of code between different PIM implementations. In addition, vector operations do not provide a convergence guarantee for threads, which is likely to be necessary for PIM. Therefore, group functions are more suitable than intuitive vectors for PIM execution.

Moreover, group functions allow users to focus on expressing functionality without bothering about specific details of the PIM hardware. On the other hand, group functions might make it difficult to express if a user wants to use special features supported by a particular PIM hardware. In such

cases, additionally offering access to lower-level, specialized functions might be required.

For portability of the SYCL extension, execution of PIM operations could fall back to GPU (or even host) execution, if no PIM capabilities are available in the system. Note that all operations in the SYCL extensions have clearly defined semantics and could therefore also be computed on non-PIM hardware. The group functions in particular aid with the portability of the code using this extension, as other mappings on different hardware are also conceivable. For example, the group functions could also be mapped to high-performance vector instructions instead of PIM if the hardware has such capabilities.

### B. PIM Directive

The PIM directive allows developers to maintain existing sequential code while gradually adding directives to expose parallelism for verification. To take advantage of its benefit, we introduce a PIM directive that is similar to the existing directives. It is directive agnostic to be compatible with existing directives such as OpenMP and OpenACC.

As can be seen in Fig. 3(b), the directive starting with `pim` has compute constructs and loop constructs similar to other directives. In addition, the data construct allows for copying data to be used in PIM memory and transferring data back to the host after computation. Furthermore, a `requires` directive is added to ensure support for unified shared memory.

## V. Proposed Compiler Framework

### A. PIM Dialect

The use of a common intermediate representation (IR) within the compiler, which can accommodate inputs from diverse programming models, offers great reusability in terms of employing identical optimizations and lowering. Consequently, we aim to present an MLIR-based PIM dialect to address this aspect.

There are three main difficulties in presenting the PIM dialect; 1) if it remains too high-level for PIM, it would lack the necessary level of detailed information. Most of the high-level dialects are similar to programming models. For example, dialects such as ACC dialect [24] and OMP dialect [25] closely align with their corresponding programming models, almost on a one-to-one basis. 2) Unlike accelerators such as GPUs, PIM integrates processing units within the memory, which requires rearrangement of memory layout to align with PIM's characteristics. Incorrect layouts can lead to performance degradation or non-execution. Consequently, additional layout conversion becomes imperative. 3) The hardware configuration of PIM may frequently change in the future. Therefore, relying on fixed optimizations can result in the need to modify the compiler whenever new hardware is introduced.

To address these issues, the proposed dialect has three distinctive features. Firstly, like other dialects lower from the programming model, our dialect may exhibit similarities to the input programming model. However, there is a significant gap between the programming model and actual PIM instruction

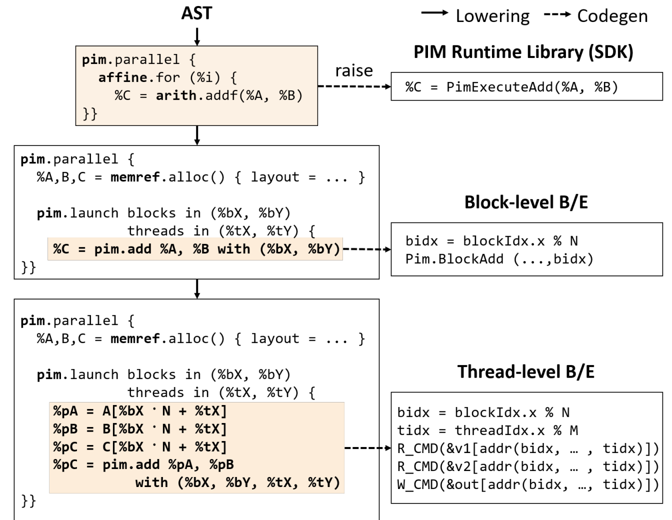

Fig. 5. Hierarchical operation lowering in PIM dialect

set architecture (ISA). Therefore, we introduce hierarchical PIM operation support. That means while PIM SDK defines the operation as a basic unit, we introduce multiple levels of software-defined operations to express the defined instructions in an optimized manner. In Fig. 5, the dialect derived from the programming model is represented in a simplified manner, often exhibiting a one-to-one mapping with constructs such as compute constructs, data-related constructs, loop constructs, and executable directives. Fig. 3(c) is an illustrative example of the PIM dialect, which closely aligns with the PIM programming model. If the level provided by the PIM SDK is at the library level, the compiler can generate code to invoke the corresponding libraries by raising the appropriate calls. If the intrinsics provided by the PIM backend operate at a lower level, such as group or thread, the PIM dialect transforms into low-level operations. In addition, this can be consistently utilized even as the hardware configuration changes. Secondly, through the lowering phase, we can exploit diverse optimization possibilities. For instance, when certain parts of an application are executed on the GPU initially and then transition to PIM in a recurring pattern, we can incorporate necessary layout conversions and perform kernel fusion to combine them with existing kernels. A more detailed explanation will be provided in the subsequent section. Lastly, we aim to apply transform IR to adapt to configurations. For example, if the number of channels, PIM blocks, or bank groups change, or if specific channels or banks are selectively used, we can modify the arguments in the transform IR rather than rewriting the entire pass each time. This approach allows us to apply the existing IR as a payload IR, providing substantial benefits in terms of scalability.

### B. Lowering Passes

To generate executable binaries for the device, the dialect undergoes a lowering process, establishing a connection with the PIM intrinsic, ISA, or SDK libraries in the PIM backend. The lowering stage can also include common optimization processes used in the existing MLIR framework, such as

```
1  %p1 = pim.load %1
2  %p2 = pim.load %2
3  %p3 = pim.gemv %p1, %p2
4  %3  = pim.update %p3
```

```
1  pim.launch blocks  (%bIdX, %bIdY) in (8, 8)
2              threads (%tIdX, %tIdY) in (32, 32){
3     %p1 = pim.reorder_load %1 with (%bIdX, %tIdX)
4                                 {map = pim_aligned}
5     pim.memcpy %p2, %2
6     %p3 = pim.gemv_wo_reorder %p1, %p2 with (%bIdX, %tIdX)
7     pim.memcpy %c, %p3
8  }
```

Fig. 6. Lowering example for GEMV operation

loop tiling, memory padding, and loop conversion, to target accelerators. This section presents four major optimizations in the lowering stage and explains the process of integrating them with the specific device back-ends.

Among the four key optimizations, the first one is alleviating redundancy. When using PIM, there is a need to synchronize the cache data from the underlying device to memory, which can be a costly operation. To improve performance, reducing the frequency of synchronization barriers is crucial. In addition, for general matrix-vector multiplication (GEMV) operations in PIM, it is necessary to rearrange the matrix data to facilitate efficient bank access. By employing techniques such as affine index transformation or kernel fusion, it is possible to eliminate the need for this rearrangement, thereby minimizing unnecessary computational overhead.

Fig. 6 illustrates an instance of lowering within the context of performing a GEMV operation. It shows the preloading rearrangement of matrix data, enabling immediate execution. Consequently, GEMV operations can be performed without requiring additional rearrangement adjustments. This facilitates the potential for kernel fusion opportunities based on mapping.

The second optimization involves hierarchical PIM operations for lowering. It entails transforming operations like data copy into lower-level operations such as vector copy and scalar copy. It can map them to the intrinsic functions supported by the backend. In Fig. 6, the ”gemv” operation is separated into two lower-level operations: matrix reorder and GEMV computation. The matrix reorder operation, in conjunction with the ”load” operation, undergoes fusion to become ”reorder_load”. And, both the ”load” and ”update” operations are lowered into ”memcpy” operations. The ”pim.launch” operation will undergo lowering based on the underlying device. If GPU contains PIM, ”pim.launch” is transformed into ”gpu.launch”. This lowering involves performing cache flush and necessary mode conversions from DRAM mode to PIM mode before pim computations. Our hierarchical lowering closely resembles the existing notion of specs in Graphene [32].

Besides, a reduction operation might incur extra operations based on the PIM implementation. If the PIM supports horizontal operations, the reduction can be performed completely in PIM blocks. However, if the PIM doesn't support horizontal operations, the reduction can partially be performed in PIM blocks, and then the host/device should finalize the reduction.

The third optimization strategy aims to maximize locality and parallelism, primarily through the exploitation of data reuse. Similar to traditional GPUs that leverage global memory and shared memory for memory hierarchy, PIM also benefits from such utilization. In PIM, it is crucial to map different thread blocks to channels and PIM blocks at the 1st level. Also, exploiting reuse by efficiently utilizing general register files, scalar register files, and banks is essential. It involves careful mapping and utilization of shared memory banks to fully leverage their capabilities and enhance overall performance.

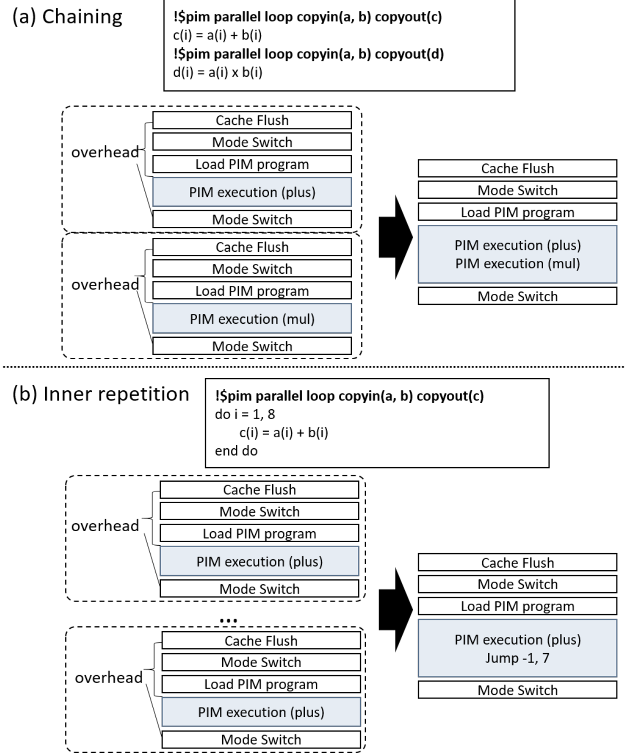


Fig. 7. Kernel fusion when mapped consecutively to PIM devices

Lastly, there is the concept of kernel fusion. In PIM execution, there can be non-negligible overhead associated with switching between different modes, such as cache flush or loading PIM programs. However, if a sequence of operations needs to be executed consecutively in the PIM, it is possible to reduce conversion overhead by employing techniques like chaining or inner repetition, as shown in Fig. 7. This approach allows for processing multiple commands within the same mode, minimizing mode switching and enhancing overall efficiency. In cases where PIM and underlying device, like a GPU, are mapped consecutively, the performance greatly relies on data placement and alignment. Moreover, as the dimensions of the threads may differ, it might be necessary to introduce kernel modifications to achieve coalesced memory access and facilitate fusion.

PIM dialect determines the dialect lowering stage based on the level of the backend. In this paper, we provide an example where the backend is specified at the library level. Its functionality is often at a similar or higher level than the dialect, which may require raising when necessary. In our implemented environment, the dialect is raised to a level where it can invoke

```
define void @test_parallel(half %0) !dbg !3 {
      br label %pim.init
pim.init:                                    ; preds = %1
      %upper.bound = alloca i32, align 4
      store i32 1, ptr %upper.bound, align 4
      %2 = call i32 @PimInitialize(i32 0, i32 0)
      %3 = icmp ne i32 %2, 0
      br i1 %3, label %pim.init.success, label %pim.init.fail

pim.init.success:                            ; preds = %pim.init
      br label %pim.body
                                    …
pim.body:                                    ; preds = %pim.init.success
      %input1 = alloca ptr, align 8
      %input2 = alloca ptr, align 8
      %5 = call i32 @PimExecuteAdd(ptr %input1, ptr %input2)
      br label %pim.deinit
                                    …
```

Fig. 8. LLVM IR with PIM library call

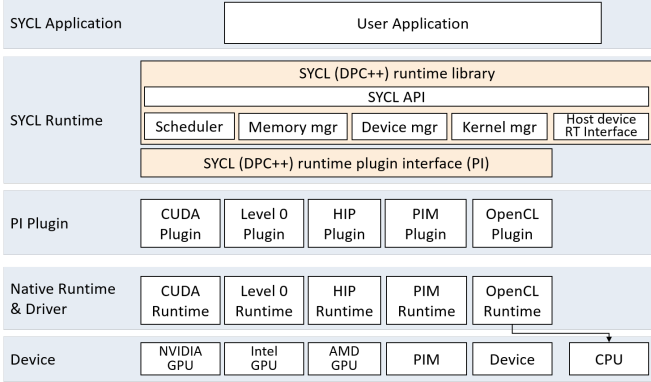


Fig. 9. SYCL Execution Flow

```
#include "pim_runtime_api.h"
int main(){
   PimInitialize(…);
   desc = PimCreateGemvDesc(…);
   h_i = PimCreateBo(desc, MEM_TYPE_HOST, GEMV_INPUT);
   … // Create weight (h_w) and output(h_o)
   d_i = PimCreateBo(desc, MEM_TYPE_DEVICE, GEMV_INPUT);
   d_w = PimCreateBo(desc, MEM_TYPE_DEVICE, GEMV_WEIGHT);
   d_o = PimCreateBo(desc, MEM_TYPE_DEVICE, GEMV_OUTPUT);

   // Set initial value (h_i, h_w)
   PimCopyMemory(d_i, h_i, HOST_TO_DEVICE);
   PimCopyMemory(d_w, h_w, HOST_TO_DEVICE);

   auto* w_to_reorder = use_device_weight ? d_w : h_w;
   auto* reordered_pim_w = PimConvertGemvWeight(w_to_reorder, gemv_order);
   PimExecuteGemv(d_o, d_i, reordered_pim_w, …);
   if(!block) PimSynchronize();
   PimDestroyBo (reordered_pim_w);

   PimCopyMemory (h_o, d_o, DEVICE_TO_HOST);

   // Destroy h_i, h_w, h_o, d_i, d_w, d_o
   PimDeinitialize();
}
```

Fig. 10. Code snippets of the PIM runtime library and tools [35]

the PIM library, followed by the generation of LLVM IR using LLVM IR builder. This process allows for seamless integration of the dialect with the library and facilitates the generations of LLVM IR, which can be further optimized and compiled using the LLVM compiler backend tools. Fig. 8 presents the LLVM IR code that includes the initialization and invocation of the element-wise addition function from the PIM library. This code demonstrates how the PIM library is integrated into the LLVM IR, enabling the utilization of PIM-specific operations and computations in the program.

In the case of SYCL, it has a software stack depicted in Fig. 9. We add support for PIM through the plugin interface. While PIM may have its standalone runtime, it is often extended to be supported by the underlying device's runtime as well.

## VI. Experiment

To demonstrate the viability of the proposed programming model and compiler framework, experiments are conducted with three scenarios. We compare the execution time between SYCL extensions and the portBLAS library with GEMV operation. Furthermore, we conduct performance comparisons for element-wise addition and general matrix-matrix multiplication (GEMM) using Fortran code annotated with PIM directives, considering various optimization techniques. Moreover, we apply our approach to the spin hall conductivity (SHC) code, an in-house code used in the HPC field, to evaluate its effectiveness. We have implemented these optimizations manually instead of relying on automatic compiler handling, leaving the potential for automatic as future work.

### A. Experimental Setup

We evaluate the performance of PIM applications which are written in SYCL and PIM directive on a GPU computing platform that contains dual AMD EPYC 7002 Series processors and AMD Instinct MI100 GPUs with infinity fabric. The operating system is Ubuntu 18.04.6 LTS. For functional verification, we compare the results obtained from the PIM simulator with the results calculated directly on the CPU to verify the correctness of the actual results. On the other hand, for performance evaluation, we employ the MI100 with PIM [36] installed on the high bandwidth memory (HBM2 DRAM – peak throughput is 1.2 TFlops for FP16 operations, 307 GB/s bandwidth per cube), as a test version. We utilize the PIM runtime library and tools [35] to execute computations on PIM hardware, providing functions for basic operations such as element-wise addition, multiplication, and GEMV. Additionally, it offers functions for buffer memory allocation and deallocation. Fig. 10 shows the GEMV code in [35] requires explicit data rearrangement.

We extend our compiler based on the SYCL compiler with HIP support built from the most recent release at the time of the study, which is the oneAPI DPC++ Compiler [37]. We also compare the results of the SYCL PIM application and portBLAS application [38]. In the performance measurement, we exclude the first iteration to warm up the driver.

To support the PIM directive, we extend the Flang compiler [39] by branching from the master and continuously rebasing to incorporate upstream changes. The integration of the PIM directive with the Clang compiler [40] opens up opportunities for its utilization in a wider range of applications, which is considered future work to be explored.

### B. SYCL Experiment: SYCL PIM vs portBLAS

PortBLAS [38] is a SYCL-based open-source implementation of BLAS co-routines, designed for performance portability across a wide range of accelerators. To optimize for the memory-bound routines in BLAS level 1 and level 2, portBLAS uses an expression tree design. Through the expression tree, multiple operations can be fused into a single kernel, which increases the computational intensity and significantly improves performance.

Fig. 11 shows the two code snippets to execute the GEMV operation. Fig. 11(a) calls the GEMV function from the portBLAS library, while Fig. 11(b) calls it from the PIM

```
1  queue q{gpu_selector{}};
2
3  // Create portBLAS executor
4  blas::Executor<blas::PolicyHandler<blas::codeplay_policy>> executor(q);
5  auto policy_handler = executor.get_policy_handler();
6
7  {
8    auto a_gpu = blas::make_sycl_iterator_buffer<half>(
9       reinterpret_cast<half *>(matrix), lda * n);
10   auto x_gpu = blas::make_sycl_iterator_buffer<half>(
11      reinterpret_cast<half *>(vector), n);
12   auto y_gpu = blas::make_sycl_iterator_buffer<half>(
13      reinterpret_cast<half *>(output.data()), m);
14
15   blas::internal::_gemv_impl<64, 256, blas::gemv_memory_t::local,
16                              blas::transpose_type::Transposed>(
17      executor, m, n, alpha, a_gpu, lda, x_gpu, incx, beta, y_gpu, incy);
18 }
```

(a) portBLAS

```
1  queue q{ext::pim_selector{}};
2  {
3    buffer<half, 2> bufMat{ reinterpret_cast<half *>(matrix),
4          range<2>{config.matrix_width, config.vector_width}};
5    buffer<half> bufVec{ reinterpret_cast<half *>(vector),
6          range<1>{config.vector_width}};
7    buffer<half> bufOut{ reinterpret_cast<half *>(output.data()),
8          range<1>{config.matrix_width}};
9
10   q.submit([&](ext::pim_handler &ph) {
11        ext::pim_accessor accMat{bufMat, ph, read_only};
12        ext::pim_accessor accVec{bufVec, ph, read_only};
13        ext::pim_accessor accOut{bufOut, ph, write_only};
14        ph.gemv(accOut, accVec, accMat);
15   });
16 }
```

(b) SYCL PIM

Fig. 11. Code snippets of GEMV operation

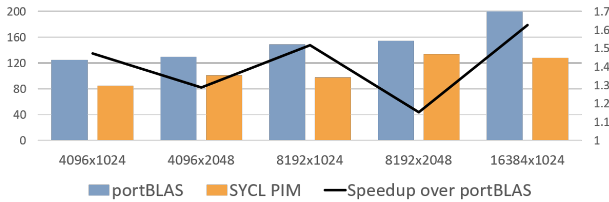


Fig. 12. Execution time comparison of the GEMV operation itself between portBLAS and SYCL PIM

extension. When using portBLAS [38], the queue is created using the GPU selector, and the GEMV function in the blas library is called by passing several arguments. However, in the SYCL PIM extension, the queue is generated using `pim_selector`. And through the `pim_accessor`, the user can access the data in a buffer for a PIM operation before finally calling the GEMV operation.

We also compared the performance between GPU and PIM using both portBLAS and PIM extension. This serves two purposes; First, it allows one to evaluate the speedup provided by PIM operations against execution on the GPU device. Second, portBLAS is also implemented in SYCL and therefore both configurations in this comparison use the same SYCL runtime for memory and execution management. This allows one to mostly eliminate the impact of the SYCL runtime on performance [41][42] and allows for a more direct comparison between GPU execution and PIM acceleration. Consequently, we did not compare it with the widely used library such as OpenBLAS and cuBLAS.

The graph depicted in Fig. 12 shows the comparison of the execution time of the GEMV operation itself, disregarding the time needed for data transfer. The left Y axis is time in microseconds, and the right Y axis is speedup over portBLAS. And the X axis indicates the size of the matrix for that particular experiment. The figure shows the raw execution time in microseconds, increasing proportionally to the number of

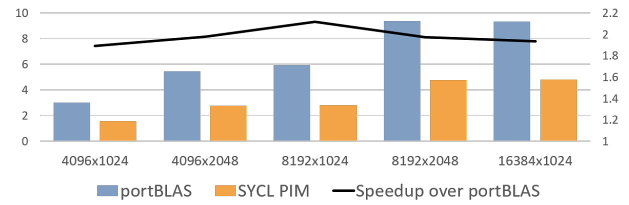


Fig. 13. Execution time comparison of the GEMV application between portBLAS and SYCL PIM

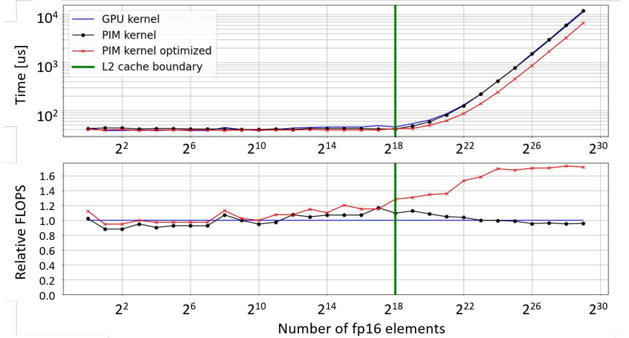


Fig. 14. Performance comparison of element-wise addition example between the base version and the optimized version

elements in the input matrix for both implementations. The absolute difference ranges between 20 and 80 microseconds. This means that the acceleration with PIM can provide a speedup between 1.2x and 1.7x over the highly optimized portBLAS library executing on the GPU.

On the other hand, the graph shown in Fig. 13 demonstrates the overall application performance of GEMV. It includes the creation of necessary buffer objects and data transfer between host and device. The left Y axis is time in milliseconds, and the right Y axis is speedup over portBLAS. And the X axis indicates the size of the matrix for that particular experiment. This result confirms the potential of PIM, delivering speedup up to 2.1 times for a memory-bound core BLAS operation like GEMV. Therefore, these two graphs provide evidence that the integration of PIM operations into the SYCL programming model does not lead to excessive overheads, since SYCL PIM achieves 89-95% of the theoretical peak performance on PIM. With larger application sizes as often encountered in real-life applications, this overhead can be better amortized.

### C. PIM Directive Experiment 1: Basic examples

The compiler performed lowering on the code in Fig. 3(b) to enable PIM library calls. Additionally, we applied an optimization technique of synchronization elimination, and compared differences in execution time. PIM operation is managed through read and write commands, and synchronization is typically achieved using barriers to guarantee the calculation. However, for one-dimensional tensors, synchronization may not be necessary and can be safely omitted.

In Fig. 14, the performance of the HIP kernel running element-wise addition on an AMD GPU is represented by a thick black line. A block dotted line represents the PIM kernel on an AMD GPU with PIM, and a red line is the optimized

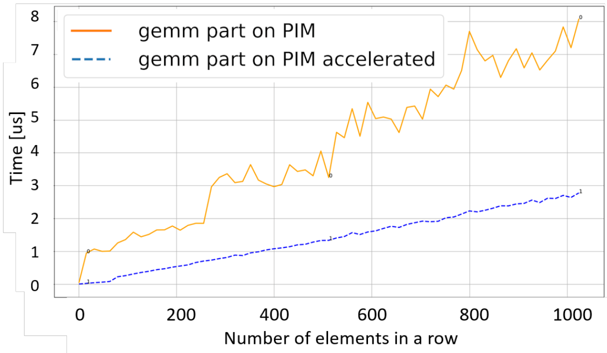

Fig. 15. Execution time comparison of GEMM operation between the base version and the optimized version

version with eliminated barriers. The below graph shows speedup over the GPU kernel. The optimized kernel demonstrates a significant speedup, achieving up to 1.7 times faster performance compared to equivalent GPU implementations when processing large tensors. This remarkable improvement indicates that our compiler surpasses the GPU in terms of efficiency for element-wise kernels.

Another optimization technique that can be applied is data rearrangement through fusion. In the case of GEMV operations in PIM, it is necessary to reorder the matrix data to meet the bank access limitations. However, by employing techniques such as affine index transformation and kernel fusion, it is possible to eliminate the need for data rearrangement and consequently reduce the execution time, as shown in Fig. 6.

Fig. 15 illustrates the result of performing a GEMM operation using the GEMV operation. The non-optimized version is represented in orange, while the optimized version is shown in blue. The pointed shape in the non-optimized version indicates a higher number of memory movement instructions. Therefore, it can be observed that the optimized GEMV operation achieves an average speedup of over 2x compared to the non-optimized version.

As an indirect measure of effectiveness and productivity improvement of the proposed programming model, we compare the lines of code for the simple GEMV application. If a user writes GEMV using the PIM runtime library as shown in Fig. 10, about 23 lines of code are required. However, if the PIM directive is used, it is possible to add just one line of directive to the existing GEMV code of eight lines, written for CPU. In the case of SYCL, only modifying six lines out of the 14-line code can enable the usage of PIM. Considering that the average development period is proportional to the number of lines [43], the result indicates that the productivity is higher than when implementing using the PIM SDK API directly.

### *D. PIM Directive Experiment 2: Real example in HPC*

We experimented with applying the PIM directive to the SHC code, an in-house HPC code, by reducing the precision due to the constraints of the PIM hardware. The most time-consuming part of the SHC code involves complex operations in the form of $Ax + By + z$. To support this operation, we extended the PIM library to handle complex numbers

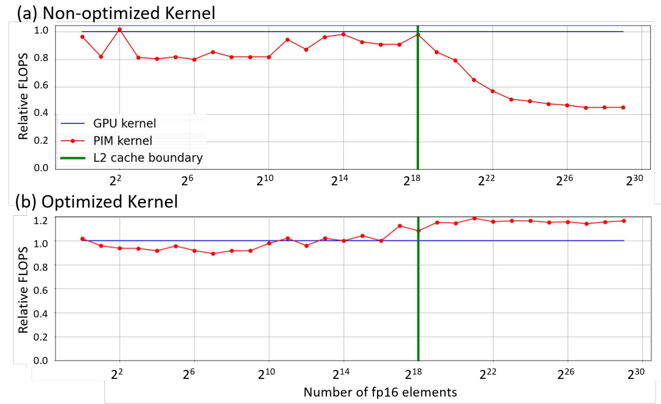

Fig. 16. Performance comparison on real example in HPC, where the x-axis is the size of the vector length, and y-axis is the speedup over GPU kernel

and evaluated its performance. However, we observed that the performance was not as expected, as the synchronization overhead between memory access instructions was too high, resulting in slower execution, illustrated in Fig. 16(a). To address this issue, we optimized the synchronizations, resulting in approximately 1.2x performance improvement for larger examples exceeding the cache size, as shown in Fig. 16(b).

The utilization of the PIM directive in the Fortran code enables to achieve performance acceleration. The performance improvements are limited to specific applications because the PIM hardware we used supports only fp16 and performs better performance in specific operations. However, it is expected that the range of HBM PIM applications can be expanded as a result of applying multiple optimization techniques in the compiler and the potential relaxation of hardware constraints.

## VII. CONCLUSION

In this paper, we have presented an end-to-end compilation infrastructure that enables targeting Processor In Memory technology from standard programming models with minimal user intervention. We demonstrate its feasibility by implementing said extensions onto open-source production toolchains and provide experiments to highlight the performance benefits of using PIM on real HPC workloads. Since the compilation pipeline is based on MLIR, we leverage the effort from the community and expand it to novel architectures. The progressive lowering of MLIR enables us to optimize high-level programming models using different techniques at different levels of abstraction.

We released our PIM SYCL extension as a vendor extension[1] in SYCL, and we are currently preparing to promote it to the Khronos extension. Furthermore, it is expected that combining multiple optimization techniques will lead to further performance improvements. Future work will focus on the automatic optimization in PIM IR and applying our proposal to a wider range of applications.

[1] Vendor extension in SYCL: https://github.com/SAITPublic/SYCL-Extension-Document.git (master branch)

## REFERENCES


[1] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[2] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openacc—first experiences with real-world applications," in *Euro-Par 2012 Parallel Processing: 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings 18*. Springer, 2012, pp. 859–870.

[3] NVIDIA. (2007) Parallel programming and computing platform. [Online]. Available: https://developer.nvidia.com/cuda-zone

[4] R. Reyes and V. Lomüller, "Sycl: Single-source c++ accelerator programming," in *Parallel Computing: On the Road to Exascale*. IOS Press, 2016, pp. 673–682.

[5] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture," *arXiv preprint arXiv:2105.03814*, 2021.

[6] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin *et al.*, "Hardware architecture and software stack for pim based on commercial dram technology: Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 43–56.

[7] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.

[8] A. Fortenberry and S. Tomov, "Extending magma portability with oneapi," in *2022 Workshop on Accelerator Programming Using Directives (WACCPD)*, 2022, pp. 22–31.

[9] A. Abdelfattah, H. Anzt, A. Bouteiller, A. Danalis, J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov *et al.*, "Roadmap for the development of a linear algebra library for exascale computing: Slate: Software for linear algebra targeting exascale," SLATE Working Note 1, Innovative Computing Laboratory, University of Tennessee, Tech. Rep., 2017.

[10] LAPACK community. (1992) Lapack — linear algebra package. [Online]. Available: https://www.netlib.org/lapack/

[11] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "Scalapack: A scalable linear algebra library for distributed memory concurrent computers," in *The Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society, 1992, pp. 120–121.

[12] R. Chandra, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[13] J. Lambert, S. Lee, J. S. Vetter, and A. D. Malony, "Ccamp: an integrated translation and optimization framework for openacc and openmp," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.

[14] Khronos Group. (2010) The open standard for parallel programming of heterogeneous systems. [Online]. Available: https://www.khronos.org/opencl/

[15] R. Han, B. Tine, J. Lee, J. Sim, and H. Kim, "Supporting cuda for an extended risc-v gpu architecture," *arXiv preprint arXiv:2109.00673*, 2021.

[16] AMD. (2016) Hipify; tools to translate cuda into portable hip c++. [Online]. Available: https://github.com/ROCm-Developer-Tools/HIPIFY

[17] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez *et al.*, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2021.

[18] H. Wang and B. Calidas, "An overview of opencl vendor extensions supported in qualcomm adreno gpus," in *International Workshop on OpenCL*, 2022, pp. 1–1.

[19] Intel. Sycl joint matrix extension. [Online]. Available: https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/experimental

[20] OpenSYCL Researchers. Sycl extensions in open sycl. [Online]. Available: https://github.com/OpenSYCL/OpenSYCL/blob/develop/doc/extensions.md

[21] MLIR Community. Affine dialect in mlir. [Online]. Available: https://mlir.llvm.org/docs/Dialects/Affine/

[22] ——. Linalg dialect in mlir. [Online]. Available: https://mlir.llvm.org/docs/Dialects/Linalg/

[23] ——. Gpu dialect in mlir. [Online]. Available: https://mlir.llvm.org/docs/Dialects/GPU/

[24] ——. acc dialect in mlir. [Online]. Available: https://mlir.llvm.org/docs/Dialects/OpenACCDialect/

[25] ——. omp dialect in mlir. [Online]. Available: https://mlir.llvm.org/docs/Dialects/OpenMPDialect/

[26] ——. Transform dialect in mlir. [Online]. Available: https://mlir.llvm.org/docs/Dialects/Transform/

[27] F. Devaux, "The true processing in memory accelerator," in *IEEE HCS*, 2019.

[28] S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim, J. Jeon, N. Kim, Y. Kwon, K. Vladimir, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, J. Lee, D. Ko, Y. Jun, K. Cho, I. Kim, C. Song, C. Jeong, D. Kwon, J. Jang, I. Park, J. Chun, and J. Cho, "A 1ynm 1.25v 8gb, 16gb/s/pin gddr6-based accelerator-in-memory supporting 1tflops mac operation and various activation functions for deep-learning applications," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65, 2022, pp. 1–3.

[29] A. A. Khan, H. Farzaneh, K. F. Friebel, C. Fournier, L. Chelini, and J. Castrillon, "Cinm (cinnamon): A compilation infrastructure for heterogeneous compute in-memory and compute near-memory paradigms," *arXiv preprint arXiv:2301.07486*, 2022.

[30] F. Hedman and A. Laaksonen, "Large scale parallel molecular dynamics simulations," in *Theoretical and Computational Chemistry*. Elsevier, 1999, vol. 7, pp. 231–280.

[31] N. Katel, V. Khandelwal, and U. Bondhugula, "High performance gpu code generation for matrix-matrix multiplication using mlir: some early results," *arXiv preprint arXiv:2108.13191*, 2021.

[32] B. Hagedorn, B. Fan, H. Chen, C. Cecka, M. Garland, and V. Grover, "Graphene: An ir for optimized tensor computations on gpus," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 302–313.

[33] A. Singer, F. Gao, and K.-T. A. Wang, "Syclops: A sycl specific llvm to mlir converter," in *International Workshop on OpenCL*, 2022, pp. 1–8.

[34] E. Tiotto, V. Pérez, W. Tsang, L. Sommer, J. Oppermann, V. Lomüller, M. Goli, and J. Brodman, "Experiences building an mlir-based sycl compiler," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2024, pp. 399–410.

[35] Samsung Electronics. Pim runtime library and tools. [Online]. Available: https://github.com/SAITPublic/PIMLibrary

[36] ——. Hbm-pim: Cutting-edge memory technology to accelerate next-generation ai. [Online]. Available: https://semiconductor.samsung.com/news-events/tech-blog/hbm-pim-cutting-edge-memory-technology-to-accelerate-next-generation-ai/

[37] Intel. oneapi dpc++ compiler. [Online]. Available: https://github.com/intel/llvm (commit:8edb62e)

[38] Codeplay Software LTD. portblas implementation. [Online]. Available: https://github.com/codeplaysoftware/sycl-blas

[39] Flang Community. Flang: Fortran compiler targeting llvm. [Online]. Available: https://github.com/flang-compiler/flang

[40] LLVM Community. Clang: C-like language compiler targeting llvm. [Online]. Available: https://github.com/llvm/llvm-project

[41] J. Lawson, M. Goli, D. McBain, D. Soutar, and L. Sugy, "Cross-platform performance portability using highly parametrized sycl kernels," *arXiv preprint arXiv:1904.05347*, 2019.

[42] T. Sabino and M. Goli, "Toward performance portability of highly parametrizable trsm algorithm using sycl," in *International Workshop on OpenCL*, 2021, pp. 1–10.

[43] R. E. Kmetovicz, *New product development: design and analysis*. John Wiley & Sons, 1992.