Pimacolaba: Collaborative Acceleration for FFT on Commercial Processing-In-Memory Architectures

Mohamed Assem Ibrahim
mohamed1.ibrahim@amd.com
Advanced Micro Devices, Inc.

Shaizeen Aga
shaizeen.aga@amd.com
Advanced Micro Devices, Inc.

ABSTRACT

This paper evaluates the efficacy of recent commercial processing-in-memory (PIM) solutions to accelerate fast Fourier transform (FFT), an important primitive across several domains. Specifically, we observe that efficient implementations of FFT on modern GPUs are memory bandwidth bound. As such, the memory bandwidth boost availed by commercial PIM solutions makes a case for PIM to accelerate FFT. To this end, we first deduce a mapping of FFT computation to a strawman PIM architecture representative of recent commercial designs. We observe that even with careful data mapping, PIM is not effective in accelerating FFT. To address this, we make a case for collaborative acceleration of FFT with PIM and GPU. Further, we propose software and hardware innovations which lower PIM operations necessary for a given FFT. Overall, our optimized PIM FFT mapping, termed **Pimacolaba**, delivers performance speedup and data movement savings of up to 1.38× and 64%, respectively, over a range of FFT sizes.

CCS CONCEPTS

• **Hardware → Memory and dense storage**; • **Computer systems organization → Single instruction, multiple data**.

KEYWORDS

Fast Fourier Transforms, GPU, Processing-in-Memory

1 INTRODUCTION

Discrete Fourier transform (DFT) is an important primitive across several domains of import (molecular dynamics, computational chemistry, data analysis and more) and it forms the key building block of important computations (e.g., solving partial differential equations). Consequently, efficient implementations of DFTs, specifically of fast Fourier transform (FFT) [14], have received considerable attention from widely deployed accelerators such as GPUs, which power seven out of ten fastest supercomputers [50].

We observe in this work that efficient implementations of FFTs on GPUs are memory bandwidth bound and as such could benefit from techniques which avail higher memory bandwidth than available at the GPU. As such, we evaluate the efficacy of recent commercially viable processing-in-memory (PIM) solutions [29, 43], which avail memory bandwidth boost (potentially up to 12× as projected in Section 3.2) over GPUs by pushing compute to near-memory compute units, to accelerate FFT. To this end, we begin with deducing compute orchestration and data mapping necessary to map FFT computation to in-memory compute units. We observe here that even with careful compute orchestration and appropriate data mapping, (except for small sizes) PIM leads to considerable slowdown vis-a-vis a GPU (average slowdown of about 52%).

To tackle this, we make a case for collaborative acceleration of FFT with PIM. That is, we propose that for a given FFT size, harnessing PIM for a judicious portion of the computation is a superior strategy than for the entire computation. To achieve this, we augment existing FFT decomposition mechanism [19, 34], which decomposes a given FFT into component computations all mapped to GPU, to also map some resultant components to PIM (using our FFT PIM routines). By carefully choosing the portion of computation that is mapped to PIM, we can harness PIM for FFT acceleration (maximum speedup of about 1.07×). As we will show, such a collaborative acceleration strategy, beyond performance, also has the effect of lowering data movement (up to 64%) which has the potential to translate to energy savings.

Next, we further observe that the majority of the operations are compute commands to in-memory compute units. As such, we propose two innovations which help lower PIM compute operations necessary for a given FFT. First, we observe that the butterfly computation [32] in FFT, which is the key building block of a FFT computation, can be decomposed to fewer PIM compute operations in certain scenarios (software optimization). Second, we identify a common computation pattern in the butterfly computation and propose simple extension to in-memory compute units which accelerates this pattern (hardware optimization). Our proposed software and hardware optimizations along with our collaborative acceleration strategy, all of which together we term **Pimacolaba**, deliver performance of up to 1.38× over a range of FFT sizes.

Finally, we observe that as accelerators and processors alike are coupled with memory, solutions like PIM can serve as augmentations over and above existing FFT acceleration solutions that only harness processor-side optimizations. As such, our work complements the rich spectrum of existing (and potentially future) efforts which aim to accelerate the important primitive of FFT.

We summarize the key contributions in this work below.

- We observe in this work that efficient implementations of FFT, an important scientific primitive, are memory bandwidth bound on modern GPUs. As such, this is the first work to evaluate the efficacy of the emerging commercially viable PIM solutions, which avail memory bandwidth boost, to accelerate FFT.
- To this end, we first deduce a mapping of FFT computation to a strawman PIM architecture representative of recent commercial PIM designs. Using the above FFT mapping, we show that even with careful data mapping and compute orchestration, (except for small sizes), PIM leads to considerable slowdown vis-a-vis a GPU (average slowdown of about 52%).
- To tackle the above challenge, we propose collaborative acceleration of FFT with PIM. That is, we augment existing

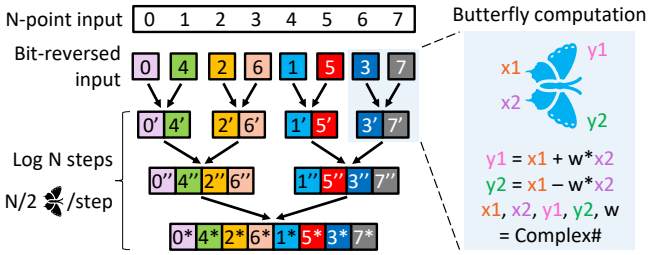

Figure 1: FFT algorithm for size N and butterfly computation.

FFT decomposition mechanism, which decomposes a given FFT into component computations all mapped to GPU, to also map some resultant components to PIM. We observe that such a strategy achieves maximum speedup of 1.07× and maximum data movement savings of 64%, a key challenge in today's systems.

- Next, we further analyze our collaborative FFT PIM mapping and propose augmentations to in-memory compute units and software optimizations to lower PIM operations necessary for a given FFT. Our resultant PIM FFT mapping, which we term **Pimacolaba**, delivers performance of up to 1.38× over a range of FFT sizes.
- We believe our work presents a complimentary acceleration strategy for an important primitive like FFT that plays well with spectrum of existing (and potentially future) acceleration solutions for FFT.

2 BACKGROUND

In this section, we provide a brief background of fast Fourier transform (FFT) algorithm and the key characteristics of its efficient implementations on GPUs. Additionally, we provide a background on commercial processing-in-memory (PIM) designs evaluated in this work.

2.1 Fast Fourier Transform (FFT)

The discrete Fourier transform (DFT) transforms a representation of a function in time-domain to its representation in frequency-domain. DFTs are an important primitive across several domains of import (e.g., molecular dynamics, computational chemistry and more). We focus in this work on an efficient method to calculate DFT, namely, fast Fourier transform (FFT) and more specifically on the Cooley-Tukey algorithm [14], a widely used and efficient algorithm for FFT. Further, we also focus on complex DFT, which transforms two N point time domain functions into two N point frequency domain functions. We discuss other forms of FFT in Section 7.

Figure 1 shows a simplified view of this efficient FFT algorithm. The algorithm takes as input an array of N samples (complex numbers) in a signal (termed as FFT size henceforth) and sorts them in bit-reversed order. The key building block of FFT algorithm is the butterfly computation. Each butterfly computation takes, as its inputs, two complex numbers x_1 and x_2 (which are two points in the input) and ω, which is another complex number called twiddle factor. As depicted in Figure 1 (right), the butterfly computation involves complex number multiplication and addition to produce as outputs two other complex numbers y_1 and y_2. The FFT algorithm comprises $log(N)$ steps each involving $N/2$ butterfly computations as depicted in Figure 1 (left). Note that the input to each step is the output of the previous step as depicted.

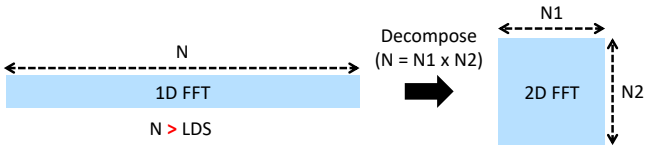

Figure 2: FFT decomposition.

2.2 Efficient FFTs on GPUs

We focus on efficient implementations of FFT on GPUs for the following reasons. First, GPUs are one of the widely used accelerators present today. In fact, GPUs power seven of ten fastest supercomputers in the world [50]. Second, starting with an accelerated baseline like that on a GPU allows us to assess the efficacy of new accelerator solutions like PIM beyond existing state-of-the-art solutions. Finally, emerging commercial PIM solutions are coupled with GPUs [43] allowing us a baseline architecture to assess.

A common feature of efficient implementations of FFT on GPUs [7, 19, 34, 37]) is that they anchor on effective use of local scratchpad (local data share/LDS on AMD GPUs or shared memory on other GPUs). That is, computing an FFT of size N such that the N input elements fit in LDS comprises loading the data in LDS and performing subsequent butterfly computations by accessing data from LDS and not main memory. By minimizing data movement and accessing data out of high-bandwidth on-chip scratchpad memory, high efficiency can be attained.

For FFT of size N such that the N input elements do not fit in LDS, existing FFT implementations decompose the problem into multi-dimensional (2D, 3D) space, processing each dimension sequentially. We depict a 2D decomposition in Figure 2. The decomposition is guided by multiple factors, and we discuss some of the key factors. First, the decomposed components together form the original computation (that is, $N = N_1 \times N_2$). Second, the decomposed components are chosen to fit in LDS (that is, while N elements do not fit in LDS, N_1 and N_2 elements, individually, fit in LDS). Note that, while a single GPU kernel is needed for FFT of size N when N input elements fit in LDS, for the depicted 2D decomposition, two GPU kernels are needed each representing a batched FFT computation. That is, N_1 (batch size) column FFTs of size N_2 followed by N_2 (batch size) row FFTs of size N_1. As such, decomposition leads to *batched* FFT computations. Finally, FFT decomposition is used recursively whenever one of the dimensions does not fit in the LDS.

2.3 Commercial PIM Solutions

Continued memory bandwidth demand, both from commercial and scientific workloads, has made it worthwhile for memory vendors to reassess, in a commercial context, processing-in-memory (PIM). PIM is a computing paradigm wherein portions of compute are offloaded to near-memory compute units to avail higher bandwidth (potentially an order of magnitude or more). As such, multiple

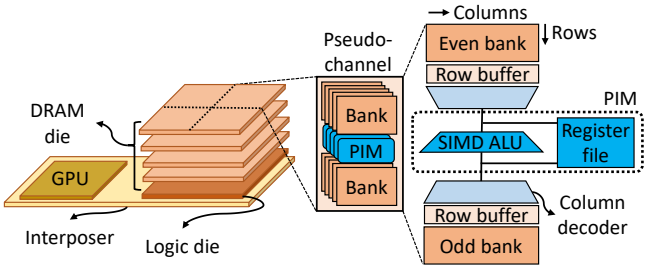

Figure 3: Strawman commercial HBM PIM architecture.

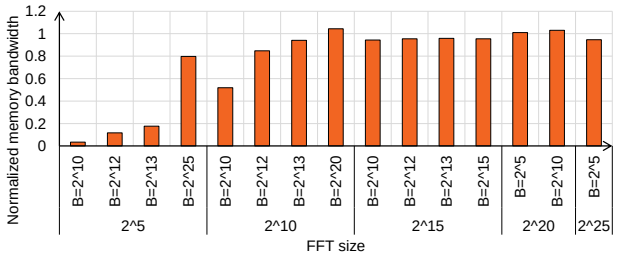

Figure 4: Efficient FFTs are memory bandwidth-bound. We measure the memory bandwidth consumption of the evaluated FFTs, then we normalize the measured bandwidth to memory bandwidth consumption of the copy kernel from BabelStream [5], which we denote as *normalized memory bandwidth*. BabelStream is broadly used as the benchmark to measure the highest achievable memory bandwidth consumption on GPUs and therefore represents a practical upper bound to compare against [9, 46, 48, 49].

memory vendors have recently demonstrated commercially viable working PIM prototypes more cognizant of industry constraints. In this work, we study a PIM architecture which is an exemplar of recent commercial PIM designs [28, 29].[1] As we study FFTs on GPUs, we focus on GPU-based PIM solutions which augment high bandwidth memory (HBM) [25].

Figure 3 depicts the strawman commercial PIM design we focus on in this work (henceforth referred to as PIM). To attain high bandwidth whilst incurring low energy/bit of data transfer, HBM is integrated with GPU within a single package, with communication amongst them via a silicon interposer. Each HBM module stacks multiples of four DRAM dies vertically, with a base logic die stacked below the memory dies. Each DRAM die comprises multiple pseudo channels, which are comprised of multiple banks. Banks within a pseudo channel share the data bus associated with the pseudo channel, while pairs of pseudo channels share a command bus. Read or write requests from the GPU typically access a single bank in a pseudo channel wherein, the address associated with the read/write request determines the pseudo channel, bank, row, and column address within the bank to be accessed. A read (or write) request first causes the specified row within a bank to be activated (*row activation*), which moves data in the row into a row-buffer structure associated with the bank. Next, data at specified column address is accessed within the row buffer via a *column access* command. A row once activated can process subsequent column accesses at lower overhead than accessing data in a separate row (necessitates another row activation).

In order to offload computations to HBM, compute units (depicted as PIM) are shared between two banks in pseudo channel. Such sharing limits area overheads and potential memory capacity costs of adding compute to memory module. A PIM unit is composed of an ALU and a register file. The ALU width and register input/output is matched to the output width of the DRAM bank (e.g., 256 bits). In addition, the ALU is capable of operating on narrow words within single DRAM word (e.g., eight 32bit operands within a 256bit DRAM word). The register file serves as a scratchpad for computation in PIM units. PIM units are controlled via read/write like instructions from GPU such as add, subtract, multiply, etc. Software enforced data consistency (e.g., cache flush) is employed to ensure data dependency ordering between GPU and PIM instructions. More details about the evaluated PIM architecture and programming model, which we carefully model in Section 4.1 and Section 4.4.1, are available in publications from memory vendors [28, 29].

As discussed, the key motivation for the above commercial interest in PIM designs is the memory bandwidth boost PIM avails. On one hand, GPUs can harness pseudo channel parallelism. That is, a GPU read/write only accesses a single bank in a pseudo channel at a time given the shared data bus. On the other hand, with PIM, in addition to pseudo channel parallelism, by computing on data without traversing the data bus, multiple banks within a pseudo channel can be configured to compute on data at the same time (via multi-bank broadcast commands). This makes it possible to have a potential memory bandwidth multiplier of $b/2$ with PIM over GPU, where b is the number of banks per pseudo channel, and the factor of 2 is because of sharing the PIM unit between two banks. Emerging commercial PIM designs, however, issue PIM operations at half the rate of regular reads/writes to accommodate multi-bank broadcast commands [28]. As such, this bandwidth multiplier is about $(b/2)/2 = b/4$ in practice. For HBM memory with 32 pseudo channels and 16 banks per pseudo channel (total #banks = 512), this bandwidth boost is about $16/4 = 4\times$. We discuss the bandwidth boost in more detail in Section 3.2. In addition to bandwidth amplification, PIM also avails considerable energy savings by not moving data (more than 50% [43]).

3 CASE FOR PIM ACCELERATION OF FFT

In this section, we motivate the consideration of PIM as a potential accelerator for FFTs based on two key observations. First, efficient FFT implementations on GPUs are memory bandwidth bound. Second, commercial PIM solutions avail memory bandwidth boost beyond that available at the GPU.

3.1 FFT is Memory Bandwidth-bound

To showcase the memory bandwidth boundedness of the efficient FFT implementations, we measure the memory bandwidth consumption of variety of FFT sizes and batch sizes (see Section 4.4.1

[1]While dissimilarities exist, these solutions have similar key architecture. In this work, we refer to these solutions as commercial PIM.

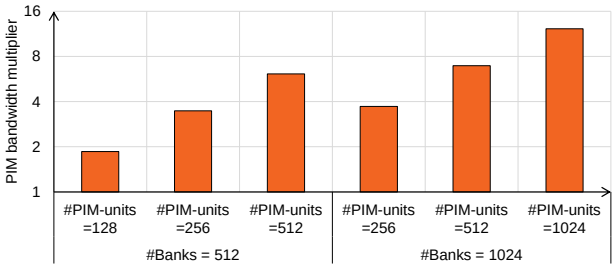

Figure 5: PIM bandwidth boost with GPU (optimistically) at 100% bandwidth utilization. Note that the bandwidth boost for #Banks = 512 and #PIM-units = 256 is < 4× (estimated in Section 2.3) as DRAM row overheads are considered.

for setup details) and normalize to memory bandwidth consumption of the copy kernel from the BabelStream benchmark [1, 5] as shown in Figure 4.[2] BabelStream is a synthetic GPU benchmark based on the STREAM benchmark for CPUs [35], which measures the sustainable memory bandwidth consumption to and from GPU memory. When running this benchmark with suitably large arrays, the data is streamed directly from GPU memory, and therefore memory bandwidth consumption of BabelStream is a strong anchor to compare memory bandwidth boundedness of other computations [9, 46, 48, 49]. We observe the following. First, as FFT size increases, memory bandwidth utilization increases and is high regardless of batch size (0.94× and 1.04× that of BabelStream for FFT size of 2^{10} with batch size 2^{13} and 2^{20}, respectively). This is expected for as the FFT size increases, on-chip scratchpad/caches are not enough to hold inputs and further, the likelihood of decomposition increases leading to inputs being read/written multiple times. Second, even for smaller FFTs, as the batch size increases, the memory bandwidth utilization increases (up to 80% that of BabelStream for FFT size of 2^5 with batch size of 2^{25}). Larger batch sizes are, in effect, similar to large FFT sizes and hence manifest similar behavior. Overall, as the data shows, efficient GPU implementations considerably push the memory bandwidth and as such can benefit from memory bandwidth boost.

3.2 PIM Avails Memory Bandwidth Boost

In Figure 5, we depict the memory bandwidth boost availed by emerging commercial PIM designs for forward-looking HBM memory for selected representative FFT sizes and batch sizes. Given the current available PIM designs showcased on HBM2 memory, we assume an upcoming HBM3 memory [26] in our study for both PIM and baseline GPU (see Section 4.4.1) as beside being forward-looking, it gives the best available bandwidth for GPU and presents a strong baseline for our work to improve upon. We consider different configurations such as baseline #banks (512) along with hypothetical exploration of large #banks (1024) due to increase in channels/stack or banks/channel. Additionally, we also vary #PIM units provisioned. Note that, when #PIM units are lower than #banks, a PIM unit is shared amongst the banks. Overall, we observe that PIM can avail considerable memory bandwidth boost over GPU (up to 12x) by having multiple banks in a channel compute on data at the same time vs. GPU accessing a single bank at a time. Also, we observe that when the PIM unit is shared between more banks, the memory bandwidth boost reduces. For example, with #PIM units = 128 under #Banks = 512), the memory bandwidth boost reduces to 1.86×. Further, more banks/more PIM units favor PIM (higher bandwidth available by PIM units). Overall, this memory bandwidth boost can be beneficial for memory bandwidth bound workloads like FFT.

[2]We are unable to report non-normalized bandwidth utilization for FFT to comply with publication guidelines of our host industry-research institution.

4 BASELINE PIM-FFT

Given the observations in Section 3, we believe that PIM is a worthy candidate for FFTs acceleration. To this end, in this section, we discuss how we orchestrate the FFT computation to PIM. We begin with the key considerations to be addressed when offloading any computation to PIM to harness acceleration, namely data mapping and compute orchestration. Subsequently, we discuss how we address these considerations specifically for FFT. We term the resultant FFT PIM routine we discuss in this section as *pim-base*.

4.1 PIM Offload Considerations

Data Mapping. As discussed in Section 2.3, our strawman PIM design, based on commercial PIM design [28], places a compute unit (e.g., ALU, register files) per two DRAM banks. Consequently, any interacting operands in the computation have to be placed in memory such that they are mapped to the same DRAM bank (or banks sharing a PIM compute unit). Further, PIM avails memory bandwidth boost by broadcasting the same command to multiple banks in the same pseudo-channel. If input/output data is interleaved appropriately across DRAM banks/channels, this broadcast feature can be harnessed. Note that avoiding inter-bank communication while harnessing PIM broadcast feature is an interesting balance. Finally, while commercial PIM designs place a SIMD ALU near DRAM banks to harness data parallelism, any cross-lane computations require shift operations which can be costly in DRAM technology due to the limited number of metal layers.

Compute Orchestration. PIM computations are kicked off by launching *pim kernels*, which are like existing GPU kernels except they issue *pim instructions*. A *pim instruction* has the effect of enqueuing a *pim command* at the memory controller which in turn instructs PIM unit to execute either computation (e.g., add, multiply, etc.) or data movement (read from row-buffer to register, write from register to row-buffer, etc.) along with necessary row activation. As discussed in Section 2.3, each PIM compute operation is a SIMD operation (e.g., eight 32bit operations over 256bit DRAM word). Finally, since memory channels are independently controlled in GPUs with multiple memory controllers, different groups of threads (e.g., workgroups or thread blocks) issue commands to different channels (commands broadcasted to banks within a channel).

Overall, programmer first decides data mapping for a computation to maximize harnessing of PIM strengths (e.g., command broadcasts) and avoid stressing PIM shortcomings (e.g., cross SIMD compute, inter-bank communication). Subsequently, a *pim kernel* which expresses the computation orchestration on in-memory compute units is launched.

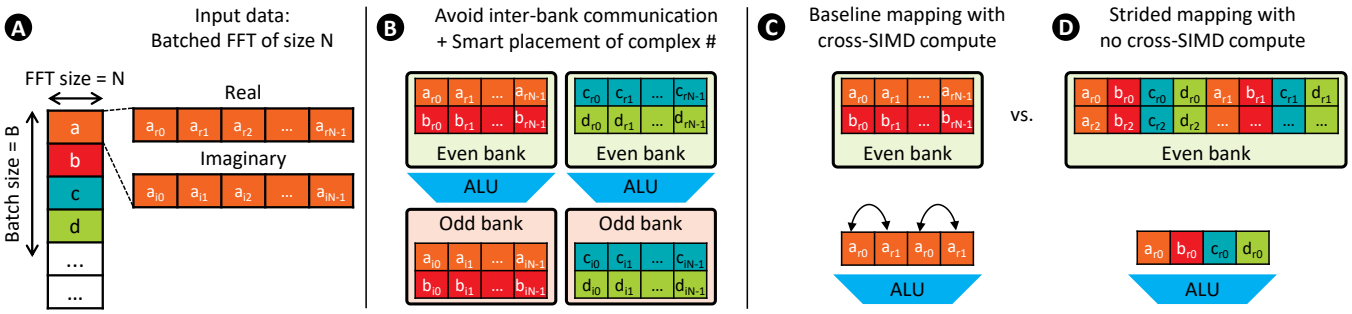

Figure 6: Proposed FFT data mapping for PIM.

4.2 PIM FFT Data Mapping

We discuss in this section the data mapping considerations specific to FFT and depict our choices with the help of Figure 6. Note that we focus on complex DFT and as such, input is comprised of complex numbers with both real and imaginary components Ⓐ. Consequently, data mapping consideration for FFT involves deciding how the N complex numbers which are the inputs to the FFT computation are placed in memory.

4.2.1 Avoiding Inter-bank Communication. As discussed in Section 2.1, for a FFT of size N, elements interact with each other at different strides in different steps. This makes mapping of these elements while avoiding inter-bank communication challenging. To tackle this, we consciously choose in our *pim-base* design to consider FFT sizes such that number of elements N fit in a pair of DRAM banks that share a PIM unit. This avoids any inter-bank communication as all interacting elements are mapped to banks with a shared ALU. This limits the maximum FFT size that we can tackle in our *pim-base* design to 2^{21} with single-precision elements (we will overcome PIM FFT size reach with alternate strategies in subsequent sections). Furthermore, we harness the fact that banks share ALU, to opportunistically place real and imaginary components of a given element in even and odd banks, respectively Ⓑ. This allows us to access both components in our computations without incurring costly row-opens.

4.2.2 Avoiding Cross-SIMD Compute. Inter-element interaction in FFT computation can also lead to cross-SIMD computation which is costly in PIM (baseline mapping) Ⓒ. To avoid these, we choose in our *pim-base* design to pack N elements belonging to a single FFT in single SIMD lane (termed strided mapping) Ⓓ. Note, this reduces the maximum FFT size that we can tackle in our *pim-base* design further to 2^{18} (driven by SIMD width and DRAM row buffer size). Furthermore, this can also lead to memory wastage if all SIMD lanes are not utilized. We discuss how we tackle this next.

4.2.3 Harnessing PIM Broadcasts and Avoiding Memory Wastage. As discussed in Section 2.2, FFTs decomposition leads to batched FFT computations. We employ such batching to both avoid memory wastage due to our data mapping design choices so far and harness PIM broadcast feature/memory bandwidth boost. That is, first, while we pack a single FFT in one SIMD lane, batching avails us of concurrent FFTs which can occupy the residual SIMD lanes and avoid memory wastage. Second, batching also allows us to spread

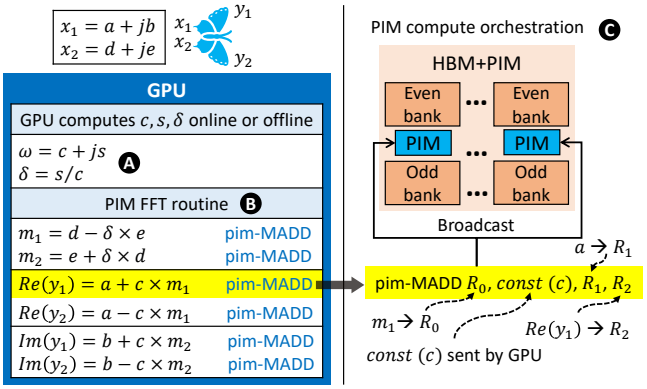

Figure 7: PIM FFT routine & orchestration.

available FFT batches across channels and banks, thus allowing us to broadcast same command across channels/banks and avail memory bandwidth boost of PIM. That is, we can compute multiple FFTs in different banks/channels concurrently by broadcasting the same PIM instructions/commands.

4.3 PIM FFT Routine

As discussed in Section 2.1, the building block of FFT is the butterfly computation. We depict in Figure 7 the orchestration of a single butterfly computation in our *pim-base* FFT routine. This computation comprises complex number multiplication and addition. Further, we depict how the required computation can be factored into six *pim-MADD* commands (multiply and add) Ⓑ along with an online or offline computation of twiddle factor components Ⓐ. As discussed in Section 4.2.3, using our *pim-kernel*, we broadcast these commands to banks in a channel (and to multiple channels) to concurrently compute multiple FFTs in a batch Ⓒ.

4.4 Performance Analysis of *pim-base*

In this section, we evaluate the performance of our proposed PIM FFT mapping, *pim-base* and further dive into how our design decisions play out. We start with motivating and discussing our performance models. Next, we discuss the effects of our data mapping choices and end with speedup analysis of our proposed *pim-base* routine vis-a-vis GPU.

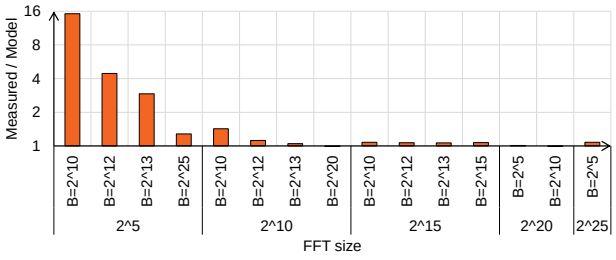

Figure 8: Fidelity of our GPU performance model.

4.4.1 Setup and Performance Model. We outline in this section our system setup and performance models for FFT computations on GPU and PIM. In our work, we choose to analyze performance using analytical models for several reasons. First, PIM is only yet available as part of functional prototypes (e.g., Lee et al. [28] couple a modified HBM2 with PIM capability with a off-the-shelf GPU as a functional prototype). Second, we aim to study performant, highly efficient FFT GPU implementations for a variety of sizes and whilst considering software optimizations (see below). This makes relying on GPU simulators difficult and lends well to analytical models. Third, as we discuss below, our setup provisions a much stronger baseline to show PIM benefits over. Finally, analytical models are currently the cornerstone to iterate over upcoming accelerator designs quickly and are increasingly employed by industry [24, 36].

System Setup. Our system setup consists of an AMD Instinct™ MI210 Accelerator comprising GPU with 104 compute units (or GPU cores) and four stacks of HBM2E memory for a total capacity of 64GB and a peak memory bandwidth of 1638.4 GB/s [3]. We profile behavior of FFT on GPU using rocFFT [7] library which is part of AMD's software ecosystem based on ROCm. We use Omniperf [6], a system performance profiling tool for machine learning/HPC workloads running on AMD MI GPUs, to study behavior of FFT kernels and gather performance statistics (e.g., reads/writes to HBM memory, memory bandwidth, etc.). Finally, we run the copy kernel from BabelStream using the maximum problem size to measure the memory bandwidth consumption as discussed in Section 3.1.

GPU Performance Model. Given the memory bandwidth boundedness of FFT computations (Section 3.1), for our GPU performance model, we assume that the GPU execution time is only limited by available memory bandwidth (i.e., we assume the compute to be free). Further, we observe that as FFT is decomposed, transpose kernels can be used in certain situations to improve the access patterns for computations. Increasingly as such kernels can be fused with FFT computation kernels [19, 34], we subtract out the effects of transpose kernels to assume an even stronger GPU baseline.

Figure 8 shows the fidelity of our GPU performance model for a variety of representative FFT sizes and batch sizes (same sizes shown in Figure 4). For our GPU performance model, we measure memory reads and writes only for FFT compute kernels (no transpose kernels) and assume the maximum memory bandwidth utilization reported by the copy kernel from BabelStream for the baseline GPU (Section 3.1). We compare this execution time to actual measured runtime. Figure 8 depicts that as FFT size or batch size increases, the computation gets increasing memory bandwidth bound and our performance model tracks well with measured execution time. For FFTs with small sizes or small batch sizes, where the computation is not memory bandwidth bound, our model projects a far more optimistic execution time than possible. In other words, for the small FFT sizes, our model reports lower FFT execution time compared to native execution, which results in a stronger GPU baseline for such sizes. We still use our proposed performance model to assess PIM speedups across all sizes to keep a unified model. Note, this means that, given our detailed performance model for PIM (see below), PIM benefits for small FFT sizes/batch sizes, will likely be higher than what we discuss below. Finally, as shown in Section 5.2, the combination of small FFT sizes with small batch sizes is uncommon in our proposed techniques.

In summary, our GPU performance model represents a strong GPU baseline with highly optimized FFT performance, which results in lower FFT execution time on GPUs compared to cycle-accurate simulators or native runs on hardware. This is because our model assumes free FFT compute operations on GPU, perfect cache reuse, optimized FFT execution with the least FFT kernels (based on the FFT decomposition algorithm and LDS size), and zero transpose kernels. Our memory-bound model assumes reading/writing input data only once per FFT kernel as any GPU implementation would need to read/write the FFT data at least once per FFT kernel. Relaxing any of these assumptions would increase the FFT execution time on GPU, resulting in higher speedups for Pimacolaba.

Table 1: Parameters for performance model.

#Banks per Stack (4-high)	512 [26]
Bandwidth per Pin	4.8 Gb/s [26]
GPU Memory Bandwidth per Stack	614.4 GB/s [26]
Row Buffer Size	1024 B [26]
DRAM Parameters	tRP = 15ns, tCCDL=3.33ns, tRAS=33ns [26]
PIM Parameters	#PIM Units per Stack = 256 #PIM Registers per ALU = 16

PIM Performance Model. We use an in-house PIM performance model. As discussed in Section 2.3, we assume a PIM architecture representative of recent commercial PIM designs [28, 29] in which the GPU issues PIM commands as special load/store accesses which bypass the caches and are issued in-order by the memory controller to multiple banks in parallel. We take a detailed DRAM command orchestration approach for our PIM performance model. That is, for a given FFT size, we deduce data mapping (Section 4.2) and orchestration (Section 4.3) necessary. Subsequently, we deduce the exact DRAM commands needed to orchestrate the computation. We augment a detailed DRAM model for modeling PIM instruction timing that incorporates the PIM DRAM timing restrictions for the deduced stream of PIM instructions, including row activation overheads. We assume the parameters listed in Table 1 for our model. Note that we assume a PIM-aware GPU which can issue *pim-instructions* and *pim-commands* at issue-rate. With the available thread parallelism at the GPU, we believe this to be a reasonable assumption.

4.4.2 Data Mapping Evaluation. Figure 9 depicts evaluation of our proposed strided data mapping (Section 4.2.2) to baseline data mapping. We depict along the y-axis the execution time normalized

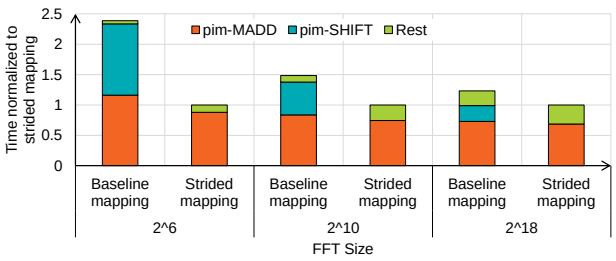

Figure 9: Strided mapping vs. baseline mapping.

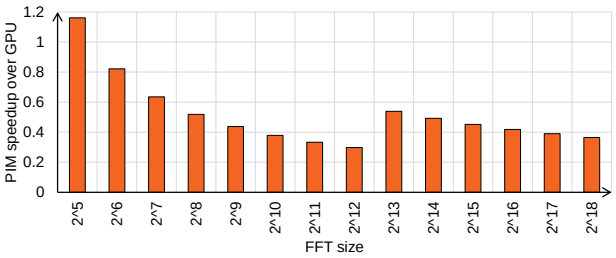

Figure 10: PIM speedup under *pim-base*.

to strided mapping for each FFT size. Further, we break down the execution time into that spent executing two key PIM instructions (*pim-MADD* and *pim-SHIFT*) and rest of the time as *Rest* (contains DRAM row-open overhead, data movement from register to row-buffer, etc.). Note that only baseline mapping scheme relies on costly shift instructions. As Figure 9 depicts, strided mapping, by avoiding *pim-SHIFT* instructions, is superior to baseline mapping. As FFT size increases, we do see these schemes get closer in performance albeit strided mapping still maintains an edge. This is so, as FFT size increases, portion of computation needing cross-lane interaction and hence *pim-SHIFT* drops. While this is so, note also that functionality like lane shifts is hard to provision in DRAM technology. Overall, our proposed strided mapping eliminates the need for costly *pim-SHIFT* commands which translates to significant reduction in FFT execution time on PIM, especially for the small FFT sizes.

4.4.3 Speedup with pim-base. Finally, we evaluate the performance of our *pim-base* FFT routine given the data mapping and orchestration decisions we have made thus far. We depict speedup of *pim-base* over GPU for various FFT sizes in Figure 10 up to the maximum size we can support in PIM (2^{18}). As the figure depicts, despite careful data mapping and orchestration, except for small sizes (2^5), *pim-base* FFT routine incurs considerable slowdown vis-a-vis GPU (average slowdown of 52%). We believe the primary reason for this is that while FFT is memory bandwidth bound on GPU, FFT manifests compute-boundedness in PIM. This is so as when FFT is mapped to GPU, only reads/writes consume memory bandwidth. In contrast, when FFT is mapped to PIM, every operation in FFT computation is now a PIM compute command (e.g., *pim-MADD*). Further, PIM compute throughput is typically lower than GPU. For example, the peak single-precision PIM throughput

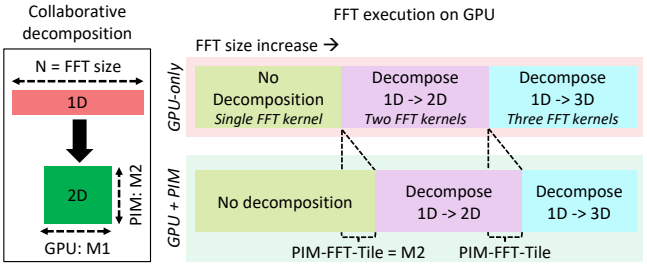

Figure 11: Collaborative decomposition in *pim-colab*.

is about 7× lower compared to our MI210 GPU with four HBM2e memory stacks, which considerably stresses PIM compute.[3] Overall, this analysis points to a more nuanced approach to harness PIM for FFT than a binary decision to use or not use PIM for the entire computation.

5 COLLABORATIVE PIM-FFT

In this section, we motivate, propose, and analyze an alternate strategy to offload FFT computations to PIM which harnesses *pim-base* but moves away from a binary offload decision (all or none of computation offloaded to PIM) to a more judicious offload mechanism where GPU and PIM collaborate to complete a FFT computation. We term the resultant FFT PIM mapping as *pim-colab*.

5.1 Collaborative Decomposition

Our proposed *pim-colab* is influenced by a confluence of several observations. First, our analysis in Section 4 showed that there are some FFT sizes (2^5) where *pim-base* does provide performance benefit. Further, in other cases (2^6), while *pim-base* is slower by a small amount, by offloading to PIM, we can harness data movement savings at a small performance cost. That is, when GPU performs the computation, data is read/written to HBM, and resultant energy expenditure is incurred. Instead, if the computation is offloaded to PIM, the said data movement and resultant energy can be saved at a small performance cost. As such, we can attain performance acceleration/data movement savings if we have an avenue to invoke PIM for certain sizes only. Second, existing efficient FFT implementations already decompose a problem into constituent components to better harness GPU scratchpad size. We propose augmenting this existing decomposition mechanism to invoke both GPU (existing kernels) and PIM component (*pim-base*). We term this strategy *collaborative decomposition*.

Figure 11 depicts collaborative decomposition (left). As depicted, a given FFT of size N is decomposed into GPU kernel (FFT of size $M1$, batch $M2$) and PIM kernel (FFT of size $M2$, termed *PIM-FFT-Tile*, batch $M1$). Our choice of *PIM-FFT-Tile* is driven by a simple algorithm: we pick the PIM FFT offload size such that we end up with same or less total #kernels invoked (PIM or GPU). In the presence of multiple choices, we pick the most efficient *PIM-FFT-Tile* (note, this can be analyzed once, offline).

Figure 11 also depicts FFT kernels invoked by GPU as FFT size increases from left-to-right for baseline GPU and for *pim-colab*. For

[3] As GPU compute throughput typically increases faster than memory bandwidth, this performance gap will exist or get wider in the future.

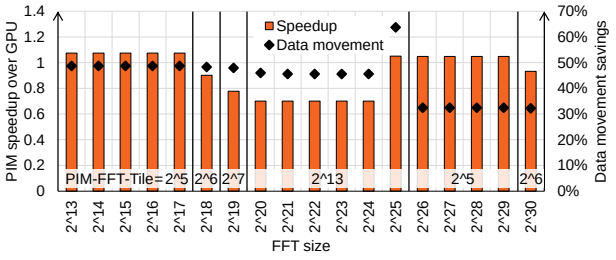

Figure 12: PIM speedup and data movement savings for *pim-colab* and *PIM-FFT-Tile* used.

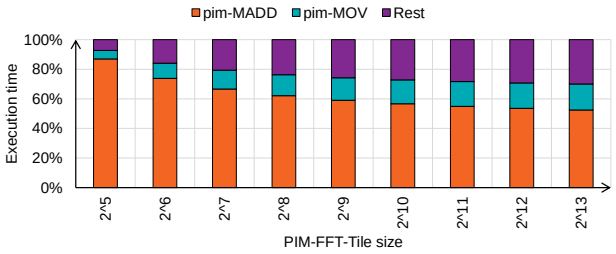

Figure 13: *pim-colab* is dominated with PIM compute.

baseline GPU, available scratchpad space (LDS) dictates #kernels to be invoked and as such, as we go from left-to-right, GPU invokes increasingly more kernels. With collaborative execution, the net effect is we shift the (size-range)-to-(kernel-count) association boundaries and effectively shrink the region where three GPU kernels are needed as depicted in Figure 11. Overall, our collaborative decomposition strategy has several benefits. First, it employs PIM judiciously and as we show in Section 5.2 stands to better harness PIM acceleration. Second, it does so while piggybacking on existing efficient mechanism of FFT decomposition. Finally, it avails considerable data movement savings.

5.2 Performance Analysis of *pim-colab*

5.2.1 Speedup with pim-colab. We evaluate the performance of *pim-colab* over GPU for various FFT sizes in Figure 12. Notice that the size range we depict here is different from *pim-base* speedup results (Figure 10). This is so, as first, the key tenet for *pim-colab* is to invoke PIM judiciously. As such, for FFT sizes, where GPU invokes single kernel (less than 2^{13} on our setup) and is already efficient, *pim-colab* does not harness PIM. As such, the performance is the same as baseline GPU. Second, *pim-colab* harnesses both PIM and GPU and as such, the maximum size we can harness PIM for increases to maximum FFT size the GPU memory can support (2^{30} for our setup). Overall, as the figure depicts, by judiciously using PIM, *pim-colab* outperforms *pim-base*. For several sizes, by harnessing PIM only where it makes sense, we attain speedup over GPU. For other cases, *pim-colab* presents a trade-off: data movement savings of up to 64% at some performance cost.

The data movement savings stem from the following. First, by using PIM to collaboratively process the FFT with PIM, we eliminate the data movement for the portion processed by PIM. Second, with *pim-colab*, we possibly reduce the total number of FFT kernels required by shrinking the region where three GPU kernels are needed as shown in Figure 11. The resultant speedup is a function of the total number of FFT kernels and the execution time of the FFT portion executing on PIM. For example, for FFT size of 2^{25} in Figure 12, the total number of FFT kernels under *GPU-only* is three kernels. However, with *pim-colab*, we use two FFT kernels achieving 64% data movement savings; GPU processes a single FFT of size 2^{12}, while PIM processes an FFT of size 2^{13}. However, we do not observe a commensurate speedup in the overall execution time as processing an FFT of size 2^{13} on PIM result in a 56% slowdown, compared to GPU baseline, as depicted in Figure 10.

5.2.2 Compute behavior of pim-colab. While *pim-colab* considerably improves over *pim-base*, we analyze *pim-colab* further to understand its behavior and deduce additional optimizations. Figure 13 depicts *pim-colab* execution time proportioning for FFT sizes we employ as *PIM-FFT-Tiles*. We break down the execution time into that spent executing PIM computation (*pim-MADD*), data movement local to PIM unit (*pim-MOV*, move data from register to row-buffer and vice versa) and rest of the time as *Rest* (contains DRAM row-open overhead). We observe that the majority of PIM execution time is spent on compute operations or *pim-MADD* commands, the building blocks of the butterfly computation. Specifically, *pim-MADD* commands represent an average of 76% of the PIM compute commands and an average of 54% of the total PIM execution time (average not shown for space reasons). The remaining time is spent moving the FFT data in and out of PIM registers. As such, any further improvements in PIM execution will have to lower resultant compute operations. As a limit study, if *pim-base* used one *pim-MADD* command instead of six we use now (Section 4.3), this can lead to a speedup of up to 4.22×. To this end, we next focus on software and hardware optimizations to lower PIM compute commands.

6 PIMACOLABA

In this section, we motivate and analyze both software optimization and hardware augmentation which aim to lower PIM compute commands. We also discuss how these can be combined to further lower PIM compute commands. We term the resultant FFT PIM mapping, which harnesses collaborative decomposition and our optimizations, as **Pimacolaba**.

6.1 Twiddle Factor Aware PIM Orchestration

We analyze the values of twiddle factors involved in FFT computation at different stages and notice that we can harness these differing values to lower PIM compute commands. Specifically, we notice that first, while twiddle factors needed is a function of FFT size N, they are deterministic based on FFT step being computed on. That is FFT size N subsumes the twiddle factors needed for size $N-1$. Second, twiddle factors 1 or $-j$ are used in initial FFT steps. For these specific values, as Figure 14 depicts, we can reduce the PIM compute commands needed for a single butterfly computation from six *pim-MADD* to four *pim-ADD operations*. As these values are used repeatedly, by enabling twiddle factor aware PIM

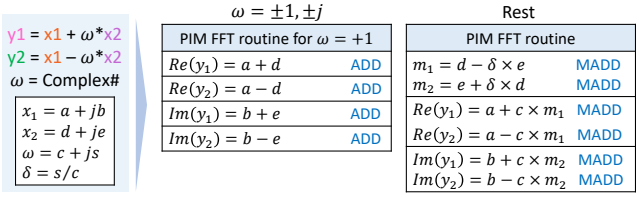

Figure 14: Twiddle factor aware PIM orchestration to reduce the number of PIM compute commands.

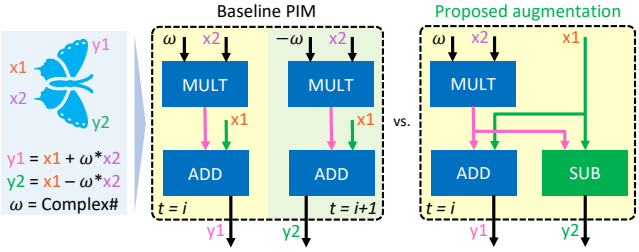

Figure 15: PIM ALU augmentations to reduce the number of PIM compute commands.

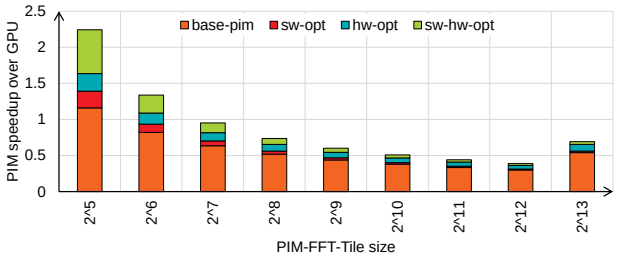

Figure 16: Optimized *PIM-FFT-Tile*.

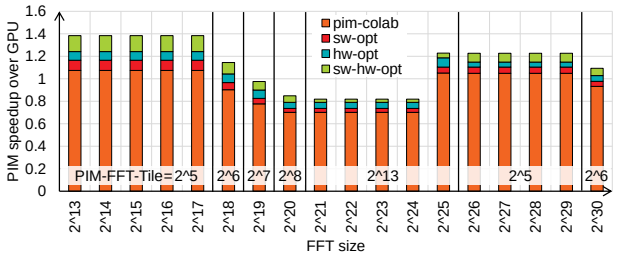

Figure 17: Pimacolaba speedup with optimized *PIM-FFT-Tile*

orchestration from GPU, as we will show below, average number of *pim-MADD* commands per butterfly can be lowered.

6.2 PIM Augmentations for FFT

Analyzing the operations in butterfly computation we observe that the result of the same multiply operation ($\omega \times x_2$) is reused in an addition and a subtraction. Using baseline PIM ALU design, while we reuse the result of the multiplication, addition and subtraction take two PIM commands to be orchestrated by the GPU. Instead, we propose to augment the PIM ALU unit, as depicted in Figure 15, such that a single PIM command realizes not just multiplication and addition (*pim-MADD*) as the baseline PIM design supports, but also an additional subtraction. This augmentation has the net effect of bringing down the number of *pim-MADD* commands per butterfly to four instead of six, independent of the used twiddle factor. Our proposed PIM command does not affect PIM orchestration; however, this necessitates additional write port to PIM ALU register file. That said, the proposed fused operation can accelerate other workloads which manifest similar patterns such as complex arithmetic, convolutions as FFTs, etc.

6.3 Combining Optimizations

Our PIM ALU augmentation helps us further enhance twiddle factor aware orchestration. First, in computations with twiddle factors of 1 and $-j$, each butterfly can now be computed using two PIM commands. Furthermore, for butterflies with $\pm 1/\sqrt{2}$ twiddle factor, the symmetry of the real and imaginary parts can be exploited to reduce the number of PIM commands to three.

6.4 Performance Analysis of Pimacolaba

6.4.1 Optimized PIM-FFT-Tile. Figure 16 depicts the speedups attained by the optimizations we discuss above for FFT sizes we employ as *PIM-FFT-Tiles*. We denote our twiddle factor aware orchestration as *sw-opt*, PIM ALU augmentation as *hw-opt*, combining the two as *sw-hw-opt*. We observe that *sw-opt* improves the performance of small *PIM-FFT-Tiles* ($\leq 2^6$) albeit with diminishing returns as *PIM-FFT-Tile* increases. This is because as the FFT size increases the proportion of twiddle factors we optimize for drops. Overall, *sw-opt* addresses the PIM compute bottleneck to some extent by reducing the average number of *pim-MADD* commands to range from 4.85 and 5.54 per butterfly (vs. 6). Compared to *sw-opt*, *hw-opt* leads to better speedups. This is because *hw-opt* benefits all butterfly computations regardless of twiddle factors. This enables *hw-opt* to tackle the PIM compute bottleneck by reducing the average number of *pim-MADD* commands to four per butterfly across all *PIM-FFT-Tiles*. Finally, *sw-hw-opt* gets us the best of both optimizations and leads to even lower *pim-MADD* commands per butterfly (2.67 to 3.46). Overall, compared to GPU, *sw-hw-opt* provides higher acceleration for a range of *PIM-FFT-Tiles* and therefore enables more options to be used when collaboratively decomposing a given FFT between GPU and PIM as shown next.

6.4.2 Speedup with Pimacolaba. We analyze how overall FFT performance looks (combining our optimized *PIM-FFT-Tile* with our collaborative decomposition approach) in Figure 17. As discussed, we term this Pimacolaba. Using our *sw-opt* and *hw-opt* approach we see speedups up to 1.16× and 1.24×, respectively. Combining the two, with Pimacolaba, we see a maximum speedup of 1.38×. An interesting benefit of our optimizations is that they increase the possible *PIM-FFT-Tile* options available as depicted in Figure 17. As discussed in Section 4.4.1, we intentionally subtract the transpose kernels time to have a stronger GPU baseline to compete against [19, 34]. If we do not subtract transposes time, Pimacolaba

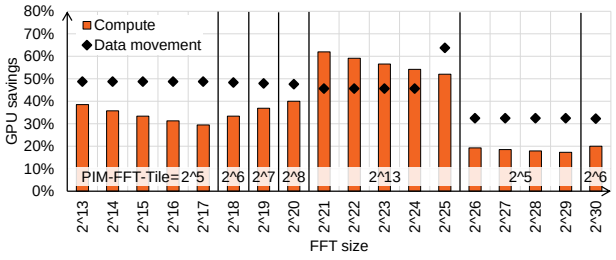

Figure 18: Savings in overall data movement and GPU compute.

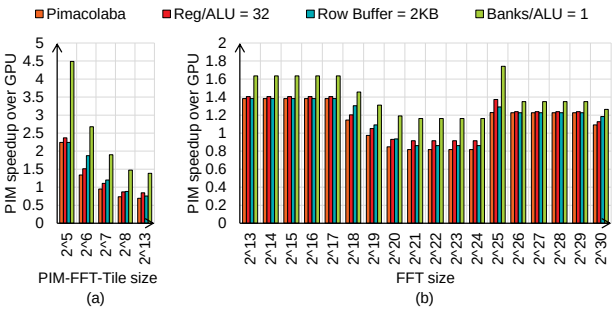

Figure 19: Pimacolaba sensitivity to potential PIM architecture optimizations for (a) *PIM-FFT-Tiles* and (b) overall FFT.

results in higher PIM speedups of up to 2.46× (instead of 1.38×). This is because, by offloading portion of FFT computation to PIM, transposes needed for FFT portion mapped to GPU can be lowered.

6.5 Data Movement Savings of Pimacolaba

Figure 18 depicts the data movement savings of Pimacolaba. Recall that by offloading portion of the computation to PIM, Pimacolaba avoids data movement (GPU reads/writes from/to HBM) and as such can lower energy expenditure.[4] We see this realization in Figure 18 as Pimacolaba leads to 32–64% data movement savings for all evaluated FFT sizes (43% on average), which in turn can result in energy savings and therefore improve the overall performance-per-watt. We also depict in Figure 18 the savings in FFT computation performed by GPU as a consequence of offloading work to PIM (in terms of butterfly computation count reduction for GPU). Overall, Pimacolaba leads to 17–62% compute savings for all evaluated FFT sizes (33% on average).

6.6 PIM Architecture Sensitivity Studies

We discuss in this section the sensitivity of Pimacolaba performance to PIM architecture variations. We depict speedups for optimized *PIM-FFT-Tiles* and overall FFT in Figure 19.

PIM Register File. Similar to register file (RF) usage in baseline GPU, RF size associated with PIM ALU allows reuse of data read from memory. As such, a larger RF size is beneficial as depicted in Figure 19. We see here that doubling the RF size, from 16 (baseline) to 32, leads to a speedup for *PIM-FFT-Tiles* ranging from 6-22% which translates to overall Pimacolaba maximum speedup of 1.41×.

Row-buffer Size. Similar to RF size, larger row-buffer (RB) size lowers row-open overheads associated with PIM computation. We study this effect in Figure 19 by doubling the RB size. While small *PIM-FFT-Tile* sizes like 2^5 already fit in baseline RB and do not benefit from larger RB size, other *PIM-FFT-Tiles* show speedups up to 40% for 2^6, which boosts the PIM speedup of FFT sizes 2^{18} and 2^{30} using this PIM-FFT-Tile.

PIM Units to Banks Ratio. Figure 19 depicts the effects of lowering PIM unit sharing between DRAM banks by provisioning a PIM unit per bank. As PIM FFT computation is bottlenecked by compute operations, doubling the number of PIM units accelerates all the *PIM-FFT-Tiles* by 2× translating to maximum Pimacolaba speedup of 1.64×.

7 DISCUSSION

7.1 Future PIM Designs

In this work, we focus on current prototypes from memory vendors [28, 29] which place a PIM unit per two banks. However, future PIM advancements may result in placing multiple PIM units within a bank (e.g., subarray-level), resulting in a higher bandwidth boost due to the higher parallelism degree unlocked by these designs. Our proposed Pimacolaba is applicable to these designs, resulting in even higher speedups for FFTs.

7.2 Real-world Applications

As in recent FFT prior work [32], we focus on 1D complex-to-complex FFT as this is the core operation of other transforms. We varied the FFT sizes to represent a range of possible transforms that are of interest to real-world HPC applications such as CHOLLA [44], GESTS [42], and GENE [18]. Many of these applications, which run on supercomputers such as Frontier as shown by ORNL [38, 39], employ 1D FFTs of sizes $2^{13} - 2^{16}$ and $2^4 - 2^5$ batches. Pimacolaba provides significant speedups for these sizes. 2D/3D FFT are also of interest and similarly decomposed to harness on-chip scratchpad space. We can harness our proposed PIM routines to accelerate each dimension separately.

7.3 FFT Variants

In our work, we focus on a radix-2 FFT for power-of-two sizes. We discuss here how we can tackle other FFT variants.

Non-2 radix. Higher radix FFTs (radix-3, radix-5, etc.) are also of interest and can improve compute intensity of FFT computation. While we deduce optimized PIM FFT routine for radix-2, routines for other radixes can be similarly deduced.

Non-power-of-two sizes. Non-power-of-two FFTs are often decomposed (e.g., as $2^a \times 3^b \times 5^c \times 7^d$ and beyond). While we discuss optimized routines for 2^a sizes, routines for other blocks can be deduced and judiciously employed.

Real FFTs. Real input FFTs are also of interest and are typically tackled using complex FFT routines [19, 34] we already discuss (e.g., by setting imaginary part of input to zero, packing real inputs into complex input with half the size, etc.).

Precision. Current PIM prototypes, being focused on machine learning, support 16bit arithmetic with 32bit accumulation. First

[4]Note, we account for data movement due to GPU transmitting commands/constants to orchestrate PIM computation

note that 16bit FFTs on GPUs are also of interest [13, 31, 33]. If higher-precision FFTs are desired, these can be supported in PIM with additional area expenditure for PIM units. Additionally, since our performance model assumes free compute for GPU, for higher precision (64bit), our speedups will stay intact (PIM compute throughput drop will be matched with GPU memory traffic increase).

Distributed FFT. We focus in our work on FFT sizes that fit in local memory attached to GPU. For larger sizes, FFT computation is distributed over multiple GPUs. In such cases, PIM can be harnessed for GPU-local portions of computations. However, resultant communication between GPUs can eat into the overall speedup that PIM can provide. That said, accelerating communication is orthogonal to this work.

Larger scratchpads. Given how central on-chip scratchpads are for efficient GPU execution, it is possible that scratchpad sizes increase for future GPUs. This will certainly help improve GPU efficiency. However, even in this scenario, for sizes where FFT does not fit in scratchpad, decomposition will be employed, and our proposal can be useful for performance and data movement savings.

7.4 PIM Software Implications

As discussed in Section 4.2, efficient computation offloading to PIM requires that data be mapped/packed appropriately in memory. Where standalone FFT computations are launched, this can be realized as a one-time cost. Further, for our collaborative decomposition scheme, by prioritizing the execution of GPU component of the decomposed FFT we achieve the following. First, the prior non-FFT kernel does not need to shuffle the data before calling the FFT library. Second, necessary data mapping for PIM can be realized by augmenting existing writes from GPU at the end of GPU FFT execution before launching *pim-kernel*. With the GPU FFT kernel processing multiple FFTs in parallel, we expect the efficiency of writing to memory to not drop considerably. Finally, such layouts can be achieved via existing semantics in FFT libraries, namely stride and distance [8].

8 RELATED WORK

Given the importance of DFT, FFT is a widely studied primitive and there exists vendor provided FFT libraries for CPUs [4, 10, 11, 23], GPUs [7, 37, 47], vendor-independent auto-tuning FFT frameworks such as Fastest Fourier Transform in the West (FFTW) [16, 17], and also for heterogeneous architectures as in HeFFTe [12]. We believe that our PIM FFT routines can be a good complement to these existing efficient FFT solutions. As an example, Pimacolaba can be integrated in those libraries as part of the auto-tuning and plan selection process. While this can add complexity to the FFT plan selection, it can be a one-time cost to be reused for a given FFT size.

Additionally, many prior works which optimize FFT implementation exist such as harnessing built-in generalized matrix multiplication (GEMM) accelerator [15, 31, 41, 45], exploiting symmetry and periodicity of the butterflies [32] and more. As accelerators and processors alike are coupled with memory, PIM can serve as augmentations over and above such existing FFT acceleration solutions that only harness optimizations targeted for the processors. As such, our work complements the rich spectrum of existing (and potentially future) efforts which aim to accelerate the important primitive of FFT.

To the same effect, FourierPIM [30] recently investigated the use of digital memristive PIM designs [51] to accelerate FFT algorithms, convolutions, and polynomial multiplication. In contrast, Pimacolaba investigates PIM designs such as those realized by memory vendors [28, 29]. Also, our work is the first to propose collaborative GPU/PIM execution which enables running larger FFTs (compared to FourierPIM being limited by crossbar size).

Overall, compared to PIM designs using speculative technology (e.g., memristor), incurring considerable area overheads due to significant changes to DRAM, or having PIM and non-PIM memory spaces (which requires memory copies) [21], Pimacolaba consciously focuses on commercially viable PIM designs getting wide traction as evident by multiple memory vendors converging to this design [28, 29]. Furthermore, HBM-based PIM solutions are suitable for GPUs which are bandwidth hungry and as such are coupled with HBM. Finally, other than FFT acceleration, many works exploit PIM's data movement reduction and performance boost to accelerate key ML and HPC workloads [2, 20, 22, 27, 40].

9 CONCLUSION

We observe in this work that high-performance implementations of discrete Fourier transforms, aka fast Fourier transform (FFT) are memory bandwidth bound on accelerators such as GPUs. As such, we evaluate in this work the efficacy of emerging commercial processing-in-memory (PIM) solutions, which have a memory bandwidth advantage over GPU by pushing compute to in-memory compute units, to accelerate FFT. By deducing a PIM FFT routine with specialized data mapping and compute orchestration, we see that PIM does not accelerate FFT. To overcome this, we propose collaborative acceleration, which augments existing FFT decomposition mechanism to use our PIM optimized FFT routines. Further, we also propose hardware augmentation and software optimization to lower PIM operations needed for a given FFT. Our proposed design, Pimacolaba, which efficiently harnesses PIM, delivers performance of up to 1.38× over a range of FFT sizes and further leads to data movement savings of up to 64%. Overall, our work introduces a complimentary FFT acceleration technique that can be combined with current (and potentially future) processor-side FFT acceleration efforts.

ACKNOWLEDGMENTS

The authors thank Nuwan Jayasena and the anonymous MEMSYS reviewers for helping improve the paper. AMD, the AMD Arrow logo, AMD Instinct, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] 2018. Evaluating Attainable Memory Bandwidth of Parallel Programming Models via BabelStream. *International Journal of Computational Science and Engineering* (2018).

[2] Shaizeen Aga, Nuwan Jayasena, and Mike Ignatowski. 2019. Co-ML: A Case for Collaborative ML Acceleration Using near-Data Processing. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*.

[3] AMD. 2023. AMD Instinct™ MI210 Accelerator. https://www.amd.com/en/products/server-accelerators/amd-instinct-mi210.

[4] AMD. 2023. AMD Optimizing CPU Libraries (AOCL) FFTW. https://www.amd.com/en/developer/aocl/fftw.html.

[5] AMD. 2023. BabelStream. https://www.amd.com/en/technologies/infinity-hub/babelstream.

[6] AMD. 2023. Omniperf. https://github.com/AMDResearch/omniperf.

[7] AMD. 2023. rocFFT Library. https://github.com/ROCmSoftwarePlatform/rocFFT.

[8] AMD. 2024. rocFFT Library Documentation. https://rocm.docs.amd.com/projects/rocFFT/en/latest/.

[9] Hartwig Anzt, Yuhsiang M. Tsai, Ahmad Abdelfattah, Terry Cojean, and Jack Dongarra. 2020. Evaluating the Performance of NVIDIA's A100 Ampere GPU for Sparse and Batched Computations. In *Proceedings of the IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*.

[10] Apple. 2023. Apple Accelerate Libraries. https://developer.apple.com/documentation/accelerate/vdsp.

[11] ARM. 2023. ARM Performance Libraries. https://developer.arm.com/downloads/-/arm-performance-libraries.

[12] Alan Ayala, Stanimire Tomov, Azzam Haidar, and Jack Dongarra. 2020. heFFTe: Highly Efficient FFT for Exascale. In *Proceedings of the International Conference on Computational Science (ICCS)*.

[13] Shenggan Cheng, Hao-Ran Yu, Derek Inman, Qiucheng Liao, Qiaoya Wu, and James Lin. 2020. CUBE – Towards an Optimal Scaling of Cosmological N-body Simulations. In *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*.

[14] James W. Cooley and John W. Tukey. 1965. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comp.* (1965).

[15] Sultan Durrani, Muhammad Saad Chughtai, Mert Hidayetoglu, Rashid Tahir, Abdul Dakkak, Lawrence Rauchwerger, Fareed Zaffar, and Wen-mei Hwu. 2021. Accelerating Fourier and Number Theoretic Transforms using Tensor Cores and Warp Shuffles. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[16] M. Frigo and S.G. Johnson. 1998. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*.

[17] M. Frigo and S.G. Johnson. 2005. The Design and Implementation of FFTW3. *Proc. IEEE* (2005).

[18] K. Germaschewski, B. Allen, T. Dannert, M. Hrywniak, J. Donaghy, G. Merlo, S. Ethier, E. D'Azevedo, F. Jenko, and A. Bhattacharjee. 2021. Toward Exascale Whole-device Modeling of Fusion Devices: Porting the GENE Gyrokinetic Microturbulence Code to GPU. *Physics of Plasmas* (2021).

[19] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. 2008. High Performance Discrete Fourier Transforms on Graphics Processors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[20] Juan Gómez-Luna, Yuxin Guo, Sylvan Brocard, Julien Legriel, Remy Cimadomo, Geraldo F. Oliveira, Gagandeep Singh, and Onur Mutlu. 2023. Evaluating Machine LearningWorkloads on Memory-Centric Computing Systems. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[21] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2022. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access* (2022).

[22] Mohamed Ibrahim, Shaizeen Aga, Ada Li, Suchita Pati, and Mahzabeen Islam. 2024. JIT-Q: Just-in-time Quantization with Processing-In-Memory for Efficient ML Training. In *Proceedings of Machine Learning and Systems*.

[23] Intel. 2023. Intel oneAPI Math Kernel Library. https://www.intel.com/content/www/us/en/docs/onemkl/get-started-guide/2023-0/overview.html.

[24] Mikhail Isaev, Nic Mcdonald, Larry Dennison, and Richard Vuduc. 2023. Calculon: A Methodology and Tool for High-Level Co-Design of Systems and Large Language Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[25] JEDEC. 2013. High Bandwidth Memory (HBM) DRAM. https://www.jedec.org/standards-documents/docs/jesd235a.

[26] JEDEC. 2023. High Bandwidth Memory (HBM3) DRAM. https://www.jedec.org/standards-documents/docs/jesd238a.

[27] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shin-Haeng Kang, Sukhan Lee, Songyi Han, YeonGon Cho, Jin Hyun Kim, Yongsuk Kwon, KyungSoo Kim, Jin Jung, Ilkwon Yun, Sung Joo Park, Hyunsun Park, Joonho Song, Jeonghyeon Cho, Kyomin Sohn, Nam Sung Kim, and Hsien-Hsin S. Lee. 2022. Near-Memory Processing in Action: Accelerating Personalized Recommendation With AxDIMM. *IEEE Micro* (2022).

[28] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[29] Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeongpil Kang, Jungyeon Kim, Junyeol Jeon, Nahsung Kim, Yongkee Kwon, Kornijcuk Vladimir, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Jaewook Lee, Donguc Ko, Younggun Jun, Keewon Cho, Ilwoong Kim, Choungki Song, Chunseok Jeong, Daehan Kwon, Jieun Jang, Il Park, Junhyun Chun, and Joohwan Cho. 2022. 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 *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*.

[30] Orian Leitersdorf, Yahav Boneh, Gonen Gazit, Ronny Ronen, and Shahar Kvatinsky. 2023. FourierPIM: High-Throughput In-Memory Fast Fourier Transform and Polynomial Multiplication. *Memories - Materials, Devices, Circuits and Systems* (2023).

[31] Binrui Li, Shenggan Cheng, and James Lin. 2021. tcFFT: A Fast Half-Precision FFT Library for NVIDIA Tensor Cores. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*.

[32] Zhihao Li, Haipeng Jia, Yunquan Zhang, Tun Chen, Liang Yuan, Luning Cao, and Xiao Wang. 2019. AutoFFT: A Template-Based FFT Codes Auto-Generation Framework for ARM and X86 CPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[33] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. 2020. Fourier Neural Operator for Parametric Partial Differential Equations. *arXiv* (2020).

[34] D. Brandon Lloyd, Chas Boyd, and Naga Govindaraju. 2008. Fast Computation of General Fourier Transforms on GPUS. In *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME)*.

[35] John D. McCalpin. 2023. STREAM. https://www.cs.virginia.edu/~mccalpin/papers/bandwidth/bandwidth.html.

[36] Diksha Moolchandani, Joyjit Kundu, Frederik Ruelens, Peter Vrancx, Timon Evenblij, and Manu Perumkunnil. 2023. AMPeD: An Analytical Model for Performance in Distributed Training of Transformers. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[37] NVIDIA. 2023. cuFFT Library. https://docs.nvidia.com/cuda/cufft/.

[38] Oak Ridge Leadership Computing Facility. 2023. Frontier. https://www.olcf.ornl.gov/frontier/.

[39] Oak Ridge Leadership Computing Facility. 2023. Update on Frontier and Early Science. https://indico.mit.edu/event/352/contributions/638/attachments/357/661/HEP-QCD.pdf.

[40] Suchita Pati, Shaizeen Aga, Nuwan Jayasena, and Matthew D. Sinclair. 2022. Demystifying BERT: System Design Implications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*.

[41] Louis Pisha and Lukasz Ligowski. 2021. Accelerating Non-power-of-2 Size Fourier Transforms with GPU Tensor Cores. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[42] Kiran Ravikumar, David Appelhans, and P. K. Yeung. 2019. GPU Acceleration of Extreme Scale Pseudo-Spectral Simulations of Turbulence Using Asynchronism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[43] Samsung. 2022. Samsung Electronics Semiconductor Unveils Cutting-edge Memory Technology to Accelerate Next-generation AI. https://semiconductor.samsung.com/newsroom/tech-blog/samsung-electronics-semiconductor-unveils-cutting-edge-memory-technology-to-accelerate-next-generation-ai/.

[44] Evan E. Schneider and Brant E. Robertson. 2015. CHOLLA: A New Massively Parallel Hydrodynamics Code for Astrophysical Simulation. *The Astrophysical Journal Supplement Series* (2015).

[45] Anumeena Sorna, Xiaohe Cheng, Eduardo D'Azevedo, Kwai Won, and Stanimire Tomov. 2018. Optimizing the Fast Fourier Transform Using Mixed Precision on Tensor Core Hardware. In *Proceedings of the IEEE International Conference on High Performance Computing Workshops (HiPCW)*.

[46] Stefan Seritan and Craig Ulmer. 2021. Benchmarking the NVIDIA A100 Graphics Processing Unit for High-Performance Computing and Data Analytics Workloads. https://www.craigulmer.com/data/2021/SAND2021-1220_uur.pdf.

[47] Dmitrii Tolmachev. 2023. VkFFT-A Performant, Cross-Platform and Open-Source GPU FFT Library. *IEEE Access* (2023).

[48] Tom Deakin. 2020. Performance Portability of OpenMP on CPUs and GPUs. https://www.openmp.org/wp-content/uploads/OpenMPBoothTalk-Deakin-SC20.pdf.

[49] Tom Deakin. 2024. BabelStream. https://hpc.tomdeakin.com/projects/babelstream.

[50] Top500. 2023. The 61st Edition of the TOP500. https://www.top500.org/lists/top500/2023/06/.

[51] Minh S. Q. Truong, Eric Chen, Deanyone Su, Liting Shen, Alexander Glass, L. Richard Carley, James A. Bain, and Saugata Ghose. 2021. RACER: Bit-Pipelined Processing Using Resistive Memory. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*.