Future Scaling of Memory Hierarchy for Tensor Cores and Eliminating Redundant Shared Memory Traffic Using Inter-Warp Multicasting

Sunjung Lee, Seunghwan Hwang, Michael Jaemin Kim, Jaewan Choi, and Jung Ho Ahn, *Senior Member, IEEE*

Abstract—The CUDA core of NVIDIA GPUs had been one of the most efficient computation units for parallel computing. However, recent rapid developments in deep neural networks demand an even higher level of computational performance. To meet this requirement, NVIDIA has introduced the Tensor core in recent generations. However, their impressive enhancements in computational performance have newly brought high pressure on the memory hierarchy. In this paper, first we identify the required memory bandwidth in the memory hierarchy as the computational performance increases in actual GPU hardware. Through a comparison of the CUDA core and the Tensor core in V100, we find that the tremendous performance increase of the Tensor core requires much higher memory bandwidth than that in the CUDA core. Moreover, we thoroughly investigate memory bandwidth requirement over Tensor core generations of V100, RTX TITAN, and A100. Lastly, we analyze a hypothetical next-generation Tensor core introduced by NVIDIA through a GPU simulation, through which we propose an inter-warp multicasting microarchitecture that reduces redundant shared memory (SMEM) traffic during the GEMM process. Our evaluation shows that inter-warp multicasting reduces the SMEM bandwidth pressure by 33% and improves the performance by 19% on average in all layers of ResNet-152 and BERT-Large.

Index Terms—GPU performance, deep neural network, tensor core, inter-warp multicasting

1 Introduction

Deep neural networks (DNNs) are currently the most successful algorithms for various tasks such as image recognition, natural language processing, translation, and Q&A tasks. To improve the model accuracy or to handle diverse problems, DNN models have been evolving over the past few years. The convolutional neural networks (CNNs) used for image recognition have increased the image resolution, channel size, and the number of layers [1], [2], [3], [4]. The transformers used for natural language processing, translation, and Q&A tasks have evolved by increasing the number of weight parameters or the sequence lengths [5], [6], [7], [8], [9]. These developments of DNN models have led to a massive increase in the required computational resources, thus demanding hardware with higher computational capabilities than ever before.

NVIDIA GPUs equipped with the CUDA core could provide better computational performance than a CPU but are still insufficient to manage the evolving DNN models [10]. Therefore, NVIDIA newly introduced the Tensor core starting with the Volta generation to increase its lower-precision computational performance dramatically (see Fig. 1). GPUs prior to the Pascal generation provided peak performance of up to 10 TFLOPS with the 32-bit floating point format (FP32). However, the Tensor core v1 [10] introduced during the Volta generation provides a much higher performance of 112 TFLOPS with the 16-bit floating point format (FP16). The Tensor core v2 [11] of the Turing generation provides 114 TFLOPS, and the Tensor core v3 [12] of the Ampere generation increases the number of streaming multiprocessors (SMs) while also improving the performance of the single Tensor core, reaching the level of 312 TFLOPS (FP16). Compared to the Kepler generation which was introduced in 2012, the Ampere generation has improved by 35.5× in terms of floating-point performance.

However, the rate of improvement in the memory bandwidth of the GPU has not kept up with such a large boost in the computational capabilities [15]. Modest increases in the L2 cache and DRAM bandwidth for each GPU generation resulted in 11.2× and 3.2× improvements, respectively, in the Ampere generation compared to the initial architecture. The aggregate shared memory (SMEM) bandwidth has improved by 4.5× due to the increase in the number of SMs

- *Sunjung Lee, Seunghwan Hwang, Michael Jaemin Kim, and Jaewan Choi are with the Department of Intelligence and Information, Seoul National University (SNU), Seoul 08826, South Korea. E-mail: {sjlee0407, seunghwan.hwang, michael604, jwchoi}@scale.snu.ac.kr.*
- *Jung Ho Ahn is with the Department of Intelligence and Information, Interdisciplinary Program in Artificial Intelligence, the Research Institute for Convergence Science, and Institute of Computer Technology, Seoul National University (SNU), Seoul 08826, South Korea. E-mail: gajh@snu.ac.kr.*

Manuscript received 14 January 2022; revised 20 June 2022; accepted 20 August 2022. Date of publication 15 September 2022; date of current version 11 November 2022.
This work was supported in part by the Samsung Advanced Institute of Technology, Samsung Electronics Co., Ltd., and in part by the Engineering Research Center Program through the NRF of Korea funded by the Korean Government, MSIT, under Grant NRF-2018R1A5A1059921. The EDA tool was supported by IC Design Education Center (IDEC), Korea.
(Corresponding author: Jung Ho Ahn.)
Recommended for acceptance by Hardware Acceleration of Machine Learnin Guest Editors.
Digital Object Identifier no. 10.1109/TC.2022.3207134

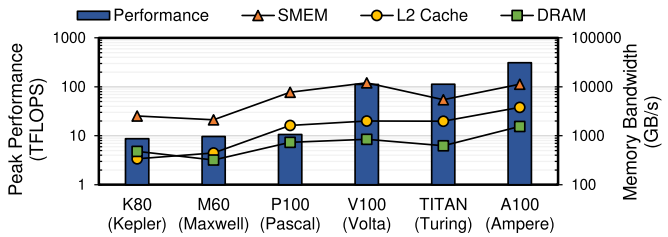


Fig. 1. Peak performance and memory bandwidths of multiple NVIDIA GPU generations. The graph is presented on a log scale. The peak performances of K80, M60, and P100 refer to the reported FP32 performance, while those of V100, TITAN, and A100 refer to the FP16 performance. Shared memory and L2 cache bandwidth of K80, M60, and P100 are referenced from [13], and that of V100, TITAN, and A100 are measured using a micro-benchmark [14].

from 13 to 108. However, the SMEM bandwidth per SM remained the same for each generation or has even been reduced by half for the Turing generation. In this paper, we identify the required memory bandwidth in the memory hierarchy as the computational performance increases on actual GPU hardware. First, we compare the required memory bandwidth between the CUDA and Tensor cores in V100 GPUs. We show that the CUDA core is almost fully utilized regardless of the DNN model, as its computational performance is relatively low. However, the Tensor core is often under-utilized because the hardware resources (e.g., bandwidth) are insufficient to support its high computational performance. We also compare the Tensor core performance of V100, RTX TITAN, and A100. The latest A100 requires up to three times more bandwidth in several layers compared to other generations. We analyze through simulation the required bandwidth in the memory hierarchy of a hypothetical next-generation Tensor core introduced by NVIDIA [16]. In the analysis, we discover the need for L1/SMEM bandwidth optimization, an issue previous studies [17], [18] did not examine. We propose an inter-warp multicasting microarchitecture to reduce the L1/SMEM bandwidth considering the arithmetic characteristics of the GPU when executing GEMM processes. Our proposal reduces the required L1/SMEM bandwidth by 33% and improves the performance by 19% on average in all layers of ResNet-152 and BERT-Large. In summary, we make the following key contributions:

- We demonstrate the required memory bandwidth of Tensor cores running DNN applications, especially compared to CUDA cores using V100.
- we thoroughly investigate the memory bandwidth requirement trends over various Tensor core generations, specifically V100, RTX TITAN, and A100.
- We propose an inter-warp multicasting microarchitecture that reduces the L1/SMEM bandwidth pressure, based on an analysis of the next-generation Tensor core.

2 Background

2.1 NVIDIA GPU Architecture

A modern NVIDIA GPU is composed of multiple streaming multiprocessors (SM), which are connected to a shared L2 cache and global memory (or DRAM) through an on-chip

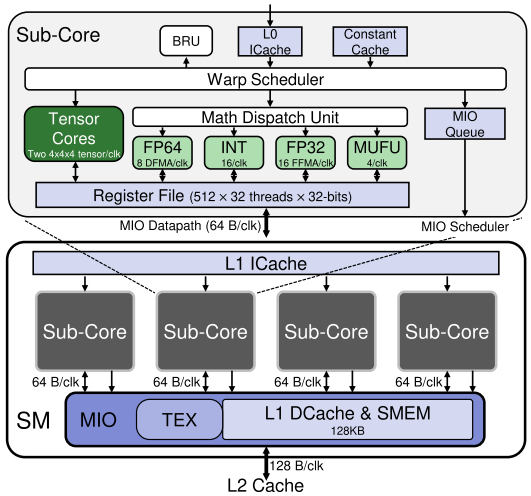


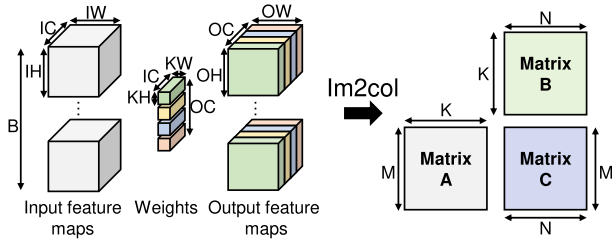
Fig. 2. SM and Sub-Core of the Volta GPU architecture [10].

interconnection network. Each SM consists of an L1 instruction cache (ICache), shared memory I/O (MIO), and four sub-cores (see Fig. 2). The MIO consists of texture memory, L1 data cache (DCache), and shared memory (SMEM). As opposed to the previous generations, the GPUs from the Volta generation combine the L1 DCache and SMEM to use both the bandwidth and capacity efficiently. The sub-core is composed of a memory subsystem that stores data and instructions, as well as arithmetic units which perform various types of operations. The memory subsystem consists of the L0 ICache, constant cache, and register file (RF). The arithmetic units include CUDA cores (supporting FP64, FP32, and INT data formats), a multi-function unit (MUFU), and two Tensor cores. Tensor cores have been newly adopted since the Volta generation to target machine learning workloads. These perform D=A $\times$ B+C operations every cycle using the 4×4 matrices A, B, C, and D; each sub-core includes two Tensor cores, performing 256 FP16 operations (4×4 $\times$ 4×2 multiply, 4×4 $\times$ 4×2 accumulate) every cycle.

GPU workloads handle multiple threads in the compute hierarchy consisting of threads, warps, and thread blocks (TBs). A single warp consists of 32 threads, while multiple warps form a single TB. Volta can process four warps simultaneously in a single SM with four sub-cores. Because the warp scheduler can schedule multiple warps in an interleaved manner, an entire TB with multiple warps can be allocated to a single SM. In fact, as long as the SMEM and RF capacities are large enough, multiple TBs can be allocated to a single SM. After the SM processes a TB, the TB scheduler allocates the next TB in a round-robin manner [19].

2.2 GEMM and Convolution

General matrix multiply (GEMM) and convolution (CONV) are two of the most commonly used operations for a DNN. GEMM multiplies a *matrix A* with a *matrix B* and accumulates the partial sum with a *matrix C*. The sizes of the three *matrices A, B,* and *C* are M $\times$ K, K $\times$ N, and M $\times$ N, respectively. CONV creates output feature maps by convolving input feature maps with weights. The sizes of the input feature maps, weights, and output feature maps are B $\times$ IH×IW $\times$ IC, KH



IH	IFmap Height	IW	IFmap Width	IC	# of IFmap Channels
OH	OFmap Height	OW	OFmap Width	OC	# of Ofmap Channels
KH	Weight Kernel height	KW	Weight Kernel width	B	# of Batch
M	B × OH × OW	N	OC	K	IC × KH × KW

Fig. 3. Conversion from convolution to the GEMM format (im2col) [20].

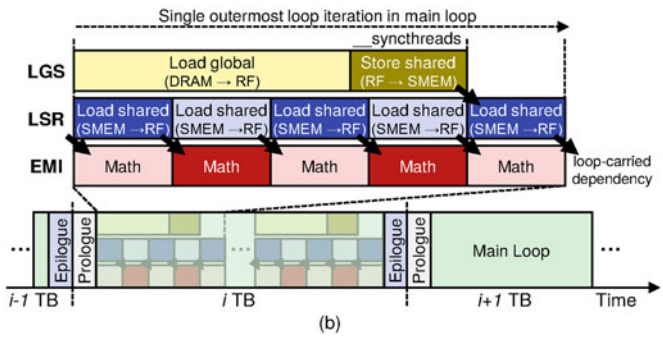
× KW×IC × OC, and B × OH×OW × OC, respectively. The GPU handles CONV with a GEMM format by converting it through im2col [20] (see Fig. 3a). The CUDA library [20], [21], [22] computes a single kernel by decomposing it into a TB, warp, and thread tile hierarchy to process DNNs efficiently. The TB tile and the warp tile sizes are expressed as (TB_M, TB_N, TB_K) and ($Warp_M$, $Warp_N$, $Warp_K$), respectively. The thread tile size differs depending on whether the GPU uses the CUDA core or the Tensor core. When the GPU uses the CUDA core, the thread tile size is defined by the number of cores. When the GPU uses the Tensor core, the thread tile size is defined by the matrix multiply-accumulate (MMA) instruction and the warp size.

2.3 GEMM of the CUDA Library

GEMM operations in the CUDA library consist of a *prologue*, *main loop*, and *epilogue* [22], [23]. Fig. 4a shows the pseudo code of CUTLASS. The prologue fetches the TB tile of matrices A and B from the global memory to SMEM, which will be used for the first iteration of the main loop. First, A $\langle TB_M, TB_K\rangle$ and B $\langle TB_K, TB_N\rangle$ are fetched from the global memory to RF. Then, the matrices are stored in SMEM after being swizzled (or permuted) to prevent bank conflicts. The prologue takes place only once per TB, whereas the other matrices are loaded inside the main loop. The main loop consists of three stages: loading from the global memory to SMEM (LGS), loading from SMEM to RF (LSR), and the execution of math instructions (EMI). To hide the data movement latency under the ALU execution time, the three stages are processed in a software pipelined manner (see Fig. 4b). Each pipeline hierarchy applies double buffering. The LGS stage fetches the TB tiles in advance from the global memory to SMEM that are to be used in the next iteration. LGS undergoes a process identical to the prologue stage. For the pre-Ampere GPU generations, if matrices A and B exist in the global memory, they are transferred through a global memory→ L2 cache → L1 cache → RF→SMEM pipeline. The execution time of the LGS stage (*LGS latency*) is the sum of the global memory access latency and the transfer latency across multiple memory levels. LSR prefetches the A$\langle Warp_M, Warp_K\rangle$ and B$\langle Warp_K, Warp_N\rangle$ warp tiles from SMEM to RF. The execution time of LSR can be calculated by dividing all data traffic of the L1 cache and SMEM by the L1/SMEM bandwidth because the L1 cache and SMEM have been combined since the Volta generation. EMI performs the matrix multiplication operation using CUDA or Tensor cores. The execution time of EMI can be

```
for(m=0; m<M; m += TB_M) {
 for(n=0; n<N; n += TB_N) {
  //Prologue (with swizzling)
  load_from_global A<TB_M,TB_K>;  store_to_shared_swizzle A<TB_M,TB_K>;
  load_from_global B<TB_K,TB_N>;  store_to_shared_swizzle B<TB_K,TB_N>;
  //Main Loop
  for(k=0; k<K; k += TB_K) {
   //Load from global memory to shared memory for next loop (LGS)
   load_from_global A<TB_M,TB_K>;  store_to_shared_swizzle A<TB_M,TB_K>;
   load_from_global B<TB_K,TB_N>;  store_to_shared_swizzle B<TB_K,TB_N>;
   for(tm=0; tm< TB_M; tm += Warp_M) {
    for(tn=0; tn< TB_N; tn += Warp_N) {
     //Load from shared memory to register file (LSR)
     load_from_shared  A<Warp_M,Warp_K>;
     load_from_shared  B<Warp_K,Warp_N>;
     for(wk=0; wk< Warp_K; wk++) {
      for(wm=0; wm< Warp_M; wm++) {
       for(wn=0; wn< Warp_N; wn++) {
        //Execute math instructions (EMI)
        C[m + tm + wm][n + tn + wn] +=
            A[m + tm + wm][k + wk] × B[k + wk][n + tn + wn]
 }}}}}}
  //Epilogue (with element-wise operations and reordering)
  element_wize(C);  store_to_global  C<TB_M,TB_N>;
}}
```

(a)



(b)

Fig. 4. GEMM of CUTLASS [22]. (a) CUTLASS GEMM pseudo code and (b) software pipeline of the outermost loop in the main loop.

calculated by dividing the number of operations of a single TB tile by the computational performance of a single SM. Because the design goal of the GPU is to hide the data transfer latency under the execution time of the computation unit, the LGS and the LSR latency can be hidden during the EMI stage in the CUDA core. The main loop is finished when the loop iterates all instances of K, as *matrix C* must be stored in RF until the partial sums are complete. The epilogue is a post-processing step for C$\langle TB_M, TB_N\rangle$, being calculated immediately after the end of the main loop. The epilogue initially permutes *matrix C* and then executes element-wise operations such as ReLU and scaling. Subsequently, *matrix C* is written back to the global memory.

2.4 L1/SMEM Bandwidth Reduction Techniques in Conventional NVIDIA GPUs

There have been multiple adjustments among different GPU generations that have eliminated inefficiencies in the memory hierarchy, in turn improving the provisioned memory bandwidth for the Tensor core. First, post-Volta generations (Turing and Ampere) remove redundant reads from SMEM to RF due to the improvements in MMA instructions. An MMA instruction is the finest granularity of control on the Tensor core that is available to the programmer. The MMA instructions are referred to as the m$\langle MMA_M\rangle$ n$\langle MMA_N\rangle$k$\langle MMA_K\rangle$ notation. It multiplies $MMA_M \times MMA_K$ fragment A and $MMA_K \times MMA_N$ fragment B and adds the result of the multiplication to

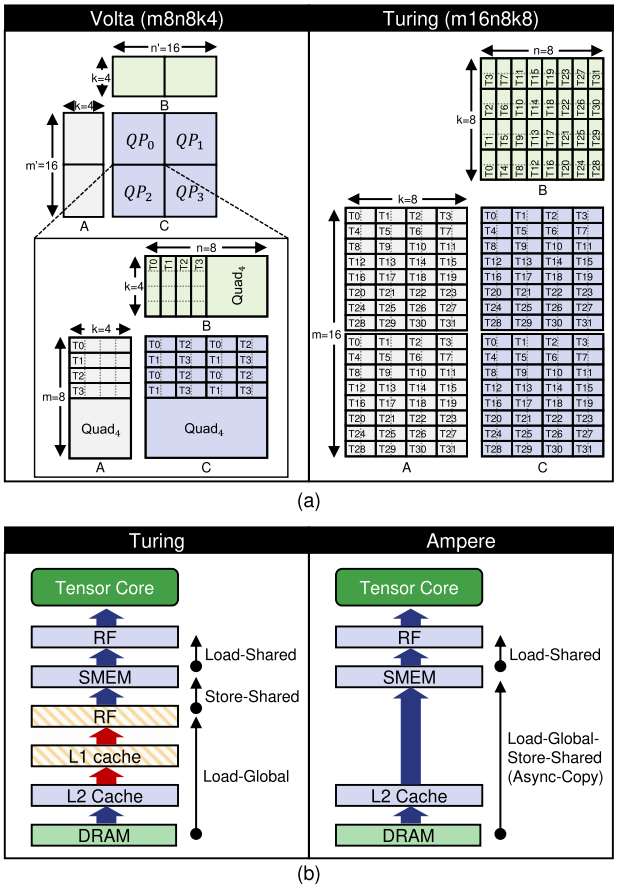


Fig. 5. L1/SMEM bandwidth optimization techniques applied in conventional GPUs: (a) Comparison of Volta and the post-Volta generations upon the execution of a single FP16 matrix multiply-accumulate (MMA) instruction. The m8n8k4 instruction used in the Tensor core v1 and the m16n8k8 instruction used in the Tensor core v2, where T indicates thread. (b) Comparison of pre-Ampere generations with Ampere on the data transfer path of matrices loaded from global memory to SMEM [12].

$MMA_M \times MMA_N$ fragment C. Tensor core v1 (Volta [10]) uses m8n8k4 instructions. As an exception to its naming convention, the actual values for the dimensions of the calculated matrices are 16, 16, and 4 (on the left in Fig. 5a). Each MMA operation is executed on eight different threads, composed of two different quads with four threads each. There exist eight different quads per warp, where each quad is allocated with threads in a sequential manner. For example, quad 0 includes threads 0-3 and quad 1 includes threads 4-7. Two of the quads that are allocated for a single MMA are referred to as a quad pair (QP). QP_0 consists of quads 0 and 4, QP_1 consists of quads 1 and 5, QP_2 has quads 2 and 6, and QP_3 has quads 3 and 7. Each QP requires its own fragment of A and B matrices in different RFs. (QP_0, QP_1) and (QP_2, QP_3) share fragment A. (QP_0, QP_2) and (QP_1, QP_3) share fragment B. Therefore, fragments A and B are redundantly loaded twice into the RF from SMEM. Tensor core v2 (Turing [11]) supports m16n8k8 instructions. Unlike the m8n8k4 instruction, the m16n8k8 instruction executes only one MMA operation per call. All 32 threads are utilized in a single MMA operation, and each thread is provided with fragments A, B, and C with the minimum dimensions of (8, 8, 8) (on the right in Fig. 5a). Because each thread manages its own independent fragment A and B, data are loaded only once from SMEM to RF, avoiding redundancies. Tensor core v3 (Ampere [12]) uses m16n8k16 instructions, which have a data shape and thread allocation method similar to those of the prior m16n8k8 instructions.

Second, a new instruction referred to as load-global-store-SMEM (LDGSTS) also eliminates inefficient data transfers of the L1 cache. Pre-Ampere generations cannot directly store data stored in the L2 cache into SMEM. Thus, they utilize simple load (global memory → L2 cache → L1 cache → RF) and store (RF → SMEM) instructions to transfer data from the global memory to SMEM, as shown on the left in Fig. 5b. However, the newly introduced LDGSTS allows a direct transfer of data from the L2 cache to SMEM, as shown on the right in Fig. 5b; it eliminates the L1 cache load/store access processes and reduces transfers along the data path shared by the L1 cache and SMEM.

2.5 Challenges and Opportunities for the Next Tensor Core Generations

The increasing number of operations needed for the latest DNNs exceeds the degree of improvement of the existing Tensor core performance. For example, modern DNNs such as GPT-3 [6], Big Bird [9], and Megatron-LM [24] use up to trillions of parameters, more than a hundred times higher than that used by BERT-large. Therefore, NVIDIA [16] has introduced a hypothetical next-generation Tensor core to cope with these emerging trends. The improvement of the computational performance of the next-generation Tensor core requires more bandwidth in the entire memory hierarchy. Fig. 6 compares the required memory bandwidth of Tensor core v1 and the next-generation Tensor core. In the figure, the next-generation Tensor core requires up to three times more memory bandwidth in all memory hierarchies compared to Tensor core v1. Various studies have been proposed to optimize the bandwidth of the GPU memory hierarchy.

DRAM Bandwidth. [16], [25], [26] reduce DRAM traffic by having several hundred MBs of L2 cache capacity. When accelerators have a huge L2 cache capacity, all inputs of one layer can be stored in the L2 cache, allowing accelerators to remove redundant DRAM accesses due to tiling. During inference, because all outputs of one layer are stored in the L2 cache, they are used immediately in the next layer without DRAM access.

L2 Cache Bandwidth. Because the L2 cache bandwidth bottleneck has been an important problem, several studies have attempted to address it. [18], [27], [28] remove redundant data traffic from the L2 cache by storing the data in the L1 cache. These methods utilize the inter-SM locality by sharing some or all of the L1 cache of individual SMs. [29], [30] reduce the L2 cache bandwidth by placing an L1.5 cache between the L2 cache and the L1 cache.

L1/SMEM Bandwidth. Because existing GPU studies have prioritized the solving of the L2 cache bandwidth bottleneck problems, there are few studies that focus on the L1/SMEM bandwidth. Recent studies [17], [18] mention that the L1/SMEM bandwidth is sufficiently abundant, and [18] increases the overall system performance using a trade-off between dropping the L1/SMEM bandwidth and utilizing inter-SM locality. However, with the continuous development of the Tensor core, the L1/SMEM bandwidth has been

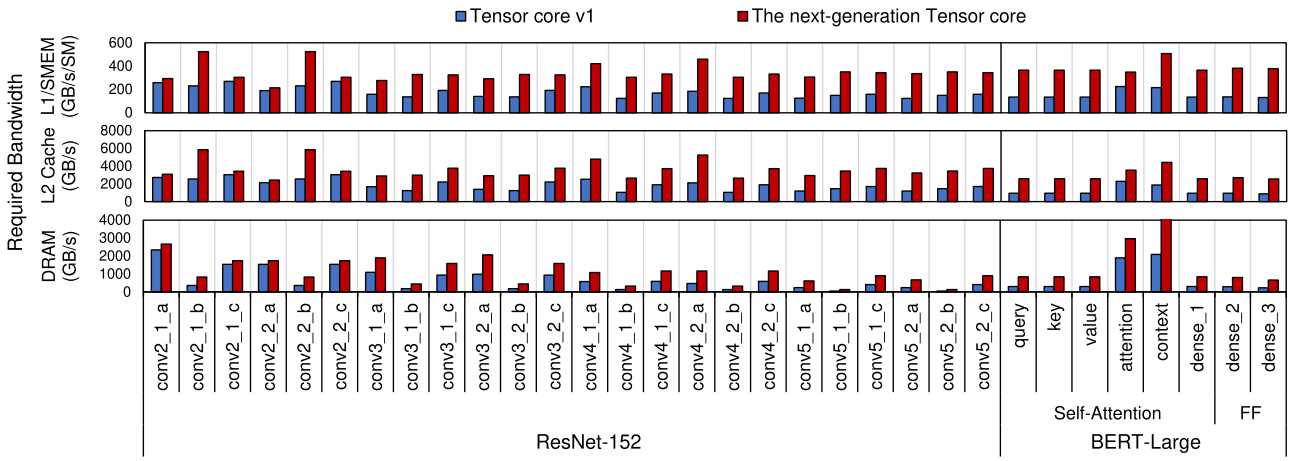


Fig. 6. Required memory bandwidth of each layer using Tensor core v1 and the next-generation Tensor core [16]. We assume that the GPUs provide an unlimited memory bandwidth of the memory hierarchy to find the bandwidth requirements at the best GPU performance. The results are obtained with the Accel-sim [17] simulator. According to [16], the next-generation Tensor core is capable of performance of 4,096 FLOPs/cycle per SM, a level is four times higher than that of Tensor core v1 and v2 and two times higher than that of Tensor core v3. Because Accel-sim only supports Tensor core v1, the next-generation Tensor core is simulated by quadrupling the Tensor core performance per SM in Tensor core v1.

responsible for one axis of the bottleneck. Therefore, it is essential to optimize the L1/SMEM bandwidth in the GPUs.

There is an opportunity to reduce the L1/SMEM bandwidth requirement by multicasting data into the sub-cores due to the arithmetic characteristics of the GPU when calculating GEMM. One SM performs GEMM operations using units of TB tiles (see Fig. 7a). Four sub-cores store C$\langle Warp_M, Warp_N\rangle$ in their own RF, as shown in Fig. 7b. A$\langle TB_M, TB_K\rangle$ and B$\langle TB_K, TB_N\rangle$ are stored in SMEM. During the operation, each sub-core reads A$\langle Warp_M, Warp_K\rangle$ and B$\langle Warp_K, Warp_N\rangle$ from the SMEM to the RF. A$\langle Warp_M, Warp_K\rangle$ and B$\langle Warp_K, Warp_N\rangle$ have reuse opportunities in multiple sub-cores. For example, both sub-core 0 and sub-core 1 require A$\langle Warp_M, Warp_K\rangle$ to be numbered to 0. However, because SMEM supports broadcasting only for threads within one warp (a.k.a. intra-warp broadcasting), multiple sub-cores read the same warp tile in SMEM redundantly. Fig. 7c is an example of reading warp tiles A and B from SMEM to RF of the sub-cores. At time 0 and time 1, sub-cores 0 and 1 require the same A$\langle Warp_M, Warp_K\rangle$ to be numbered to 0, but they read warp tiles repeatedly from SMEM. Moreover, sub-cores 2 and 3 require the same A$\langle Warp_M, Warp_K\rangle$ to be numbered to 1, but they also read repeatedly at time 2 and time 3. During times 4-7, multiple B warp tiles are also read unnecessarily from SMEM.

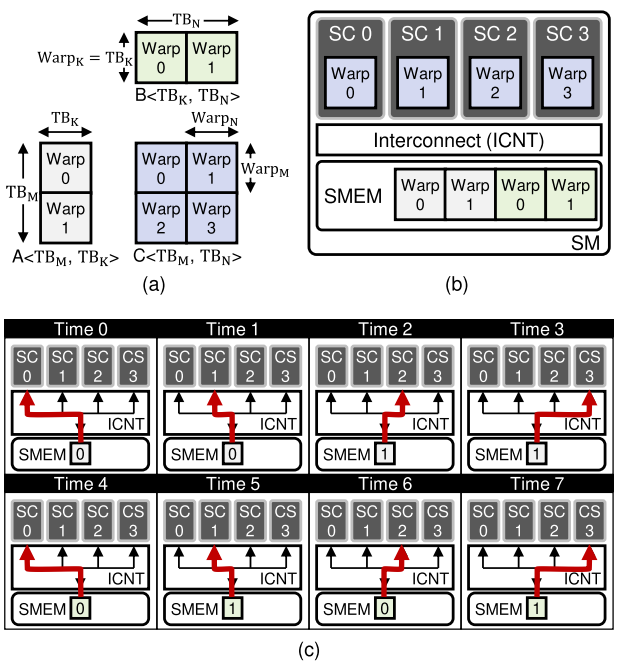


Fig. 7. (a) Unit of a TB tile used by one SM. A TB tile is divided by multiple warp tiles. (b) Method of a TB tile inside SM. (c) A redundant memory access example on a conventional SM.

3 Inter-Warp Multicasting

3.1 Microarchitectural Details

We propose an inter-warp multicasting microarchitecture (IWMA) that eliminates redundant L1/SMEM accesses when performing GEMM operations. IWMA supports inter-warp multicasting for multiple warps that require the same tiled matrix, while the warps are processed in different sub-cores. IWMA locates an inter-warp multicasting unit (IWMU) inside the shared MIO unit of SM (see Fig. 8a) to utilize the existing data/control path used for MIO instructions that are translated into L1/SMEM instructions [10]. To utilize the IWMU when needed, a new instruction, the multicasting SMEM load instruction (MLDS), is adopted and replaces the prior SMEM load instruction (LDS) when necessary. When multiple warps require the same tiled matrix as described in Section 2.5, they exhibit an identical LDS instruction in their NVIDIA Machine ISA (SASS) instruction traces. Fig. 8b shows an example trace during the GEMM operation. LDS redundancies can easily be found, as in the case where the warp 0 and 1 require the same matrix A$\langle Warp_M, Warp_K\rangle$ to be numbered to 0, or where warps 2 and 3 require the matrix A$\langle Warp_M, Warp_K\rangle$ to be numbered to 1. The redundant LDS instructions for these cases can be replaced by MLDS instructions and later trigger the IWMU for multicasting. The IWMU consists of a load request table (LRT), four comparators, a decision unit, and a request queue. First, each LRT entry holds four

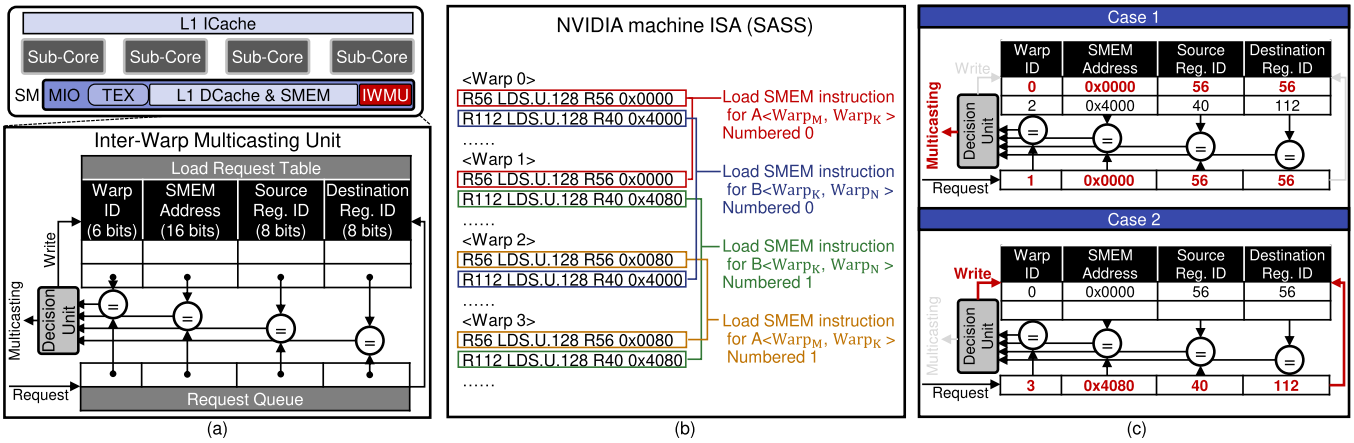


Fig. 8. (a) Inter-warp multicasting microarchitecture (b) Load SMEM (LDS) instruction traces among all NVIDIA Machine ISA (SASS) instructions when the GPU executes a GEMM operation. LDS instructions are expressed in the following order: destination register ID, opcode, source register ID, and SMEM address. (c) Examples of multicasting (case 1) and writing an instruction to the LRT (case 2).

values: the warp ID, SMEM address, source register ID, and destination register ID, which are written correspondingly to the requested MLDS instruction. Upon every MLDS request, the four comparators of the IWMU check every LRT entry and inform the decision unit. When there exists an entry with the same SMEM address, source register ID, and destination register ID, but a different warp ID, the decision unit triggers IWMU multicasting. If such a condition does not exist, the entry corresponding to the MLDS instruction is inserted into the LRT table.

When a decision unit decides to multicast, it sends requests to two sub-cores that will perform inter-warp multicasting. Before data are moved from SMEM to RF, the two sub-cores must check their dependencies to determine if their destination registers are available. Each sub-core checks for register collisions using a scoreboard. Afterward, when the target registers of the two sub-cores become available, enable requests are sent to the IWMU. Finally, the IWMU multicasts data to the two sub-cores.

3.2 Inter-Warp Multicasting Unit Operation

We illustrate the operation details of the IWMU using two example cases as shown in Fig. 8c. Case 1 describes a multicasting situation. The LRT already holds the previous MLDS instructions for warps 0 and 2. The (Warp ID, SMEM address, source register ID, and destination register ID) of warp 0 is (0, 0x0000, 56, 56), and that of warp 2 is (2, 0x4000, 40, 112). When the IWMU receives a new MLDS request, (1, 0x,0000, 56, 56), from the sub-score, the IWMU compares the request with every entry in the LRT. Because the first entry has the same SMEM address and source/destination register ID but has a different warp address, multicasting for the first entry and the new MLDS instruction is triggered. The matched first entry is then deleted from the LRT.

Case 2 describes a situation in which a new MLDS request is written to the LRT. The LRT already stores the previous MLDS instruction for warp 0, specifically (0, 0x0000, 56, 56). When a new MLDS instruction (3, 0x4080, 40, 112) is requested, the IWMU again compares it with the entries in LRT. Because there is no entry with the same SMEM address, source, and destination register ID, the IWMU newly writes the requested MLDS instruction to the LRT.

3.3 Load Request Table Size

We choose an LRT size large enough to support inter-warp multicasting fully for GEMM operations, but not an excessive size so as to minimize the table search overhead. For full support of inter-warp multicasting, the IWMA must store all MLDS instructions sent by the sub-cores, until they are deleted after multicasting. The maximum number of instructions can be inferred through the GEMM source code of the CUDA library. Because the CUDA library is executed as a software pipeline, the LRT only stores instructions included in one *__syncthreads* section as shown in Fig. 4b. Based on the widely used (128, 128, 32) TB tile size, the CUDA library needs 128 MLDS instructions for each TB. In fact, considering that two MLDS instructions from different warps form a pair, the LRT requires a maximum of 64 entries.

The average LRT search cost is even lower considering the average number of instructions stored, as the GPU fetches data by subdividing eight times, similar to the LSR in the software pipeline (see Fig. 4b). Thus, the instructions are multicasted before the LRT is full.

3.4 Compilation

To adopt the new techniques, prior works either 1) identified a certain pattern and undertook marking of the ordinary instructions [31], [32], or 2) replaced the prior instructions in the kernel code with new ones [33]. In the former case, the compiler uses marking to enable the instructions to execute the new techniques, in full consideration of $blockIdx$, $blockDim$, scalar constants, global kernel input parameters, and the patterns. This does not require modification to the kernel code because the ordinary instructions are utilized. However, it instead requires the compiler to investigate and mark the proper instructions. The latter case is necessary when implementing the new technique with the original instructions is not possible or when the previous instructions can be replaced with new instructions one by one. The LDSM instruction of NVIDIA, which replaces the LDS instruction, is one such example. While the prior LDS instruction causes unnecessary shared memory accesses due to the fixed RF mapping, LDSM newly enables different methods of data fetching. This method requires the explicit modification of the source code

TABLE 1
Hardware Specifications of the Latest GPU Architectures

	V100	RTX TITAN	A100
Architecture	Volta	Turing	Ampere
Boost core clock(GHz)	1.37	1.55	1.41
The number of SMs	80	72	108
FP32 CUDA perf. (TFLOPS)	14.0	14.2	19.5
FP16 Tensor perf. (TFLOPS)	112	114	312
L1/SMEM capacity (KB)	128	96	192
L2 cache capacity (MB)	6	6	40
***L1/SMEM BW (GB/s/SM)**	150	76	105
***L2 cache BW (GB/s)**	2,000	1,977	3,800
***DRAM BW (GB/s)**	850	624	1,555
***L1/SMEM latency (cycle)**	28	32	33
***L2 latency (cycle)**	198	230	205
***DRAM latency (cycle)**	397	489	568

**We measure the bandwidth and latency with reference to [14], [17], [36].*

TABLE 2
Hardware Specifications of the Simulated GPU Architectures

	Tensor core v1	Next-generation Tensor core
Tensor core perf. per SM (TFLOPS)	1.15	4.6
Total Tensor perf. (TFLOPS)	92	368
Boost core clock(GHz)	1.13	1.13
The number of SMs	80	80
L1/SMEM capacity (KB)	128	128
L2 cache capacity (MB)	6	6
L1/SMEM BW (GB/s/SM)	150	150
L2 cache BW (GB/s)	2,000	2,000
DRAM BW (GB/s)	850	850
L1/SMEM latency (cycle)	20	20
L2 latency (cycle)	180	180
DRAM latency (cycle)	300	300

in the CUDA library, using *asm volatile* as the LDSM instruction. To implement the IWMA, we modify the source code in a similar manner and change the LDS to LDSM instructions. When executing a GEMM operation in a Tensor core, there is some data redundancy when fetching matrices A and B from SMEM to RF, as explained in Section 2.5. In fact, this redundancy is static and exhibits a regular pattern. Therefore, it is possible to enable the IWMA technique simply by changing each of the LDS instructions to MLDS in the kernel code.

To support the MLDS instruction in the NVIDIA GPU, further modification to the NVCC compiler, which is provided by NVIDIA, is necessary. However, because the NVCC is a closed source, we instead exploit the fact that Accel-sim takes the SASS trace from NVBit [34] as the input. NVBit is a binary instrumentation tool that extracts the SASS trace from the kernel code executed on a GPU device. Based on this trace, we are able to identify the LDS instructions and replace them with MLDS instructions, enabling IWMA support on Accel-sim.

4 Experimental Setup

We measured and compared the required memory bandwidth and performance with V100 (Volta, tensor core v1), RTX TITAN (Turing, tensor core v2), and A100 (Ampere, tensor core v3) devices as shown in Table 1. To apply the same methodology used in [16], we used Accel-sim [17] to evaluate the proposed architecture. Table 2 shows the specifications of the simulated GPUs. Because Accel-sim only supports Tensor core v1, The next-generation Tensor core is simulated by quadrupling the Tensor core performance per SM in Tensor core v1. With regard to DNN workloads, we used ResNet-152 (ResNet) [2] and BERT-large (BERT) [5], two of the most widely used benchmarks [35]. Because ResNet and BERT have multiple repeated layer patterns, we do not present results for the duplicated layers. For example, ResNet uses the same layer shape for the layers after the conv2_2, conv3_2, conv4_2, and conv5_2. BERT consists of 24 self-attention and feed-forward (FF) modules. The batch sizes of ResNet and BERT were set to 256 and 128, respectively. We executed forward passes with a precision of FP16. Because data redundancy occurs in all types of layers using GEMM, our technique is applicable to both inference and training. Also, when we analyze the CUDA library, LDS instructions are used in the same way regardless of the degree of arithmetic precision. CUTLASS [22], an open-source library, was used to measure the effects of various components inside the library in detail. CUTLASS performs approximately 11% and 3% worse compared to cuDNN and cuBLAS, respectively, on ResNet and BERT (based on the V100 FP16 Tensor core). We utilized the CUDA 11.2 toolkit and the Nsight compute profiling tool provided by NVIDIA. We experimented with various types of TB and warp sizes and then selected the results that showed the peak performance.

5 Evaluation

We provide the evaluation results from the actual hardware and from a GPU simulator. In Section 5.1, we use real hardware to identify the required memory bandwidth and performance inside the GPU using the ResNet and BERT DNN models. In Section 5.2, we use the GPU simulator to demonstrate the impacts of the proposed architecture.

5.1 Identification of Resource Utilization in Real Hardware

5.1.1 Comparison of the CUDA Core and the Tensor Core

First, we compare the required memory bandwidth and performance of the CUDA and Tensor cores using ResNet and BERT. We calculate the required memory bandwidth by dividing the memory traffic by the ideal execution time which considers only the computation units to determine how much bandwidth is required at full utilization if there had not been a bandwidth bottleneck. We also demonstrate the performance of each layer with indicators that consider the impacts of under-utilized factors. We make the following key observations.

Most layers calculated on the Tensor core, as opposed to the CUDA core, require on-chip memory (L1/SMEM, L2 cache) bandwidth close to or exceeding what can be provisioned by the hardware. Fig. 9 shows the required memory bandwidth of each layer when the GPU executes DNN models using the V100 CUDA/Tensor cores. For the

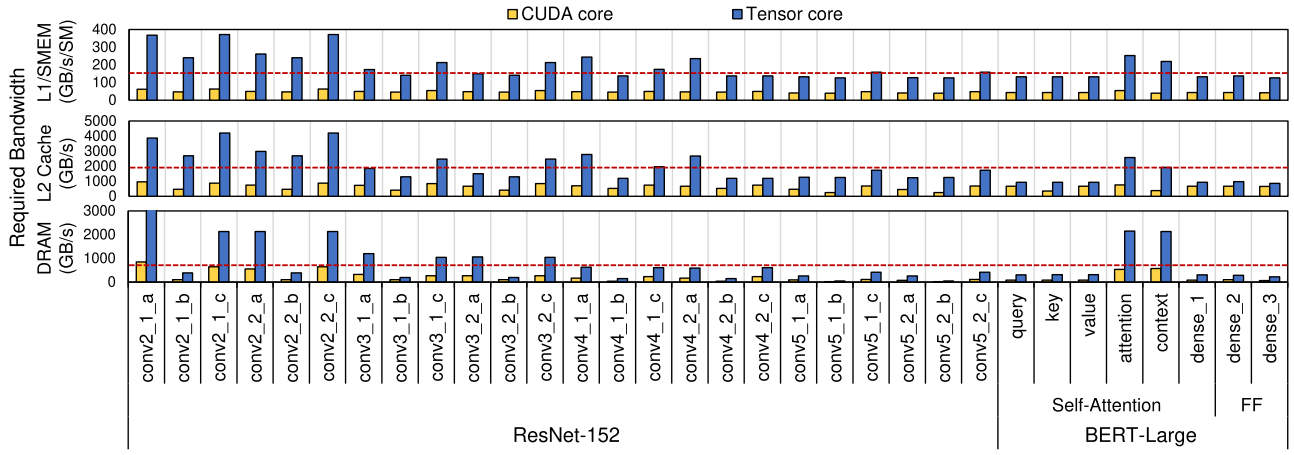


Fig. 9. Required memory bandwidth of each layer using V100 CUDA/Tensor cores. The required memory bandwidth is calculated by dividing the memory traffic by the ideal execution time which only considers the computation units. The L1/SMEM, L2 cache, and DRAM (global memory) bandwidth values provided by the hardware are 150 GB/s/SM, 2,000 GB/s, and 850 GB/s, respectively, as represented here by the dotted lines.

CUDA core, because the GPU is designed to provide memory bandwidth that matches the computational performance of the CUDA core, none of the layers undergo a bandwidth bottleneck. In contrast, as the overall data traffic in the memory hierarchy at a given time increases in the Tensor core, nearly all of the layers require memory bandwidth close to the limit of or even higher than the supplied on-chip memory bandwidth. The early layers in ResNet, which are located closer to the input feeding layer, have lower data reusability due to the small K sizes. Also, the conv_x_a and conv_x_c layers require relatively high memory bandwidth in all memory hierarchies because these layers use a 1×1 CONV filter which has a small K size compared to the conv_x_b layers and the corresponding 3×3 CONV filter. Therefore, they require high memory bandwidth at all levels compared to the other layers. BERT has a low required bandwidth because most layers are composed of matrices with large M, N, and K sizes. Although the L2 cache and the DRAM bandwidth are increased compared to the case with the CUDA core, it only accounts for 40% of the bandwidth provided by the hardware in most layers. In contrast, the required L1/SMEM bandwidth is relatively high because the m8n8k4 instruction inefficiently duplicates matrices from L1/SMEM to RF. Moreover, the attention and context layers of BERT, which perform the batched matrix multiply (BMM) operation, have a higher required bandwidth than the other layers because the nBMM matrices are grouped for each other and are not reused among the other matrix groups.

The Tensor core cannot reach its maximum performance in any of the layers. In particular, ResNet shows less than 70% of performance in all layers. The CUDA core shows high performance outcomes in all layers, except for the conv2_1_a, conv2_1_c, and conv2_2_c layers which have a small K size (see Fig. 10). First, the CUDA core does not encounter bandwidth bottlenecks in the memory hierarchy. Second, a longer EMI time can successfully hide the *LGS latency*. We can obtain the EMI time by dividing the total number of operations in a single TB tile by the number of CUDA cores in an SM. When we assume a TB tile size of (128, 128, 8), which is the maximum size using the CUDA core, the EMI time is $128\times128\times8\times2128$ (CUDA core performance per SM) = 2,048 cycles. The LGS takes about 1,100 cycles by adding the DRAM access latency (397 cycles) to the data transfer latency (calculated as 684 cycles) assuming the worst-case scenario of loading data from DRAM. Therefore, the LGS time is always less than the EMI time and can be fully pipelined. In contrast, the Tensor core performs worse (relative to the peak performance) in most layers due to various factors. First, the performances of layers with small K and N sizes given their low data reusability are degraded due to bandwidth bottlenecks. In Fig. 10, because the conv2 and context layers have N and K sizes equal to or smaller than 512, they have DRAM or L1/SMEM bandwidth bottlenecks. Second, if the layers have a small K or N size, the EMI time is not long enough to hide all of the LGS time. When we assume a TB tile size of (256, 128, 32), which is the maximum size using the Tensor core, the EMI time is $256\times128\times32\times21024$ (Tensor core performance per SM) = 2,048 cycles. The LGS, however, takes about 3,000 cycles by adding the DRAM access latency (397 cycles) to data transfer latency (calculated as 2,575 cycles) assuming the worst-case scenario in which data is loaded from DRAM.

5.1.2 Comparison of Three Tensor Core Generations

Although NVIDIA GPUs have been able to accelerate DNN models with the adoption of the Tensor core, various new problems have emerged that did not exist or were not noticeable in previous systems with only CUDA core. Consequently, the past three generations of the Tensor core and GPU have gradually been improving to assess these new problems. We compare the required memory bandwidth of the last three generations of V100, RTX TITAN, and A100 using ResNet and BERT.

The required bandwidth in the memory hierarchy tends to increase in proportion to the computational performance of the Tensor core. The L2 cache and DRAM bandwidth required by A100 are two to three times higher than those of other GPUs because the required bandwidth of A100 increases at a ratio identical to that of the performance improvement (see the green bars in Fig. 11). Regardless of the Tensor core generation, the L2 cache and DRAM traffic are

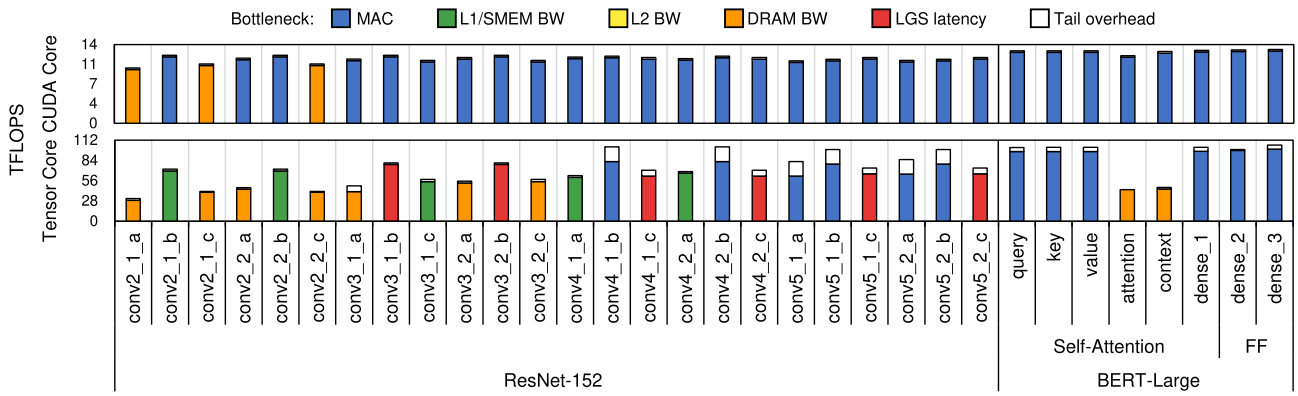


Fig. 10. Performance of each layer using V100 CUDA/Tensor cores. The color inside the bar indicates the cause of the performance degradation [37]. A white bar indicates overhead due to the tail effect.[1] The actual performance of the layer is solely represented by the color-marked part of the bar. The value at the top of the Y-axis presents the FP32/FP16 peak performance of the CUDA Tensor cores.

similar for the following reasons. First, all GPU generations read data from DRAM or the L2 cache in the same way. Second, they use similar (TB_M, TB_N) sizes, which determine the amount of data that must be read redundantly. Thus, the required bandwidth of the L2 cache and DRAM is proportional to the computational performance. L1/SMEM data traffic differs according to the Tensor core generations because the MMA and load instructions of each generation fetch data differently. When we compare V100 and RTX TITAN, which have similar computational performance capabilities, V100 requires the highest bandwidth in most layers, as the MMA instruction of V100 loads twice as much data from L1/SMEM to RF than the instruction after the Turing generation. Because the Ampere generation supports LDGSTS instructions that move data of the L2 cache directly to SMEM, A100 has less L1/SMEM traffic than RTX TITAN. However, the redundant traffic is less than the SMEM load traffic used by RF for MMA instructions because SMEM stores data only once. Meanwhile, because the computational performance is 2.2 times higher in A100, the L1/SMEM bandwidth is also higher in A100 with a similar proportion.

5.2 Evaluation of the Inter-Warp Multicasting Microarchitecture

5.2.1 Performance Evaluation

Because the inter-warp multicasting microarchitecture (IWMA) is applicable to all DNN models using GEMM, it reduces L1/SMEM for all layers of ResNet and BERT. Fig. 12 compares the required L1/SMEM bandwidths and performance of the next-generation Tensor core and IWMA. The figure confirms that a bandwidth reduction occurs at all layers. The IWMA can also be utilized in other CNNs, RNNs, MLPs, and transformers that use CONV and GEMM as the main operations to reduce their bandwidth pressures further. The bandwidth reduction is almost constant at all layers, at 33%, because the traffic is reduced at the same rate in all layers. The ratio of L1/SMEM load and store instructions is 5:1. When the IWMA performs inter-warp multicasting, it reduces the ratio of load instructions by 3. When we compare the before-and-after ratio of instructions, it changes from 6 to 4, i.e., approximately 33%. The IWMA improves the performance by 19% on average for all layers. The performance improvement of ResNet (17%) was lower than that of BERT (23%), because several layers of ResNet, which have less data reusability, are affected more by other memory hierarchies than L1/SMEM.

The IWMA can also be used in several applications with the CUDA core. Among applications that use CUDA core, there are those that perform matrix multiplications. For these applications, if multiple sub-cores require the same matrix existing in the SMEM, the IWMA can reduce SMEM accesses. In contrast, there is no performance degradation for CUDA core applications that do not require the IWMA. The IWMA operates only according to MLDS instructions. If an application using the CUDA core does not require the IWMA, the GPU uses the CUDA core as usual.

5.2.2 Comparing IWMA With DARSIE

Dimensionality-aware redundant SIMT instruction elimination (DARSIE) [32] aims to remove redundant SMEM accesses from different warps in a single TB, a goal that is similar to that of the IWMA. DARSIE initially detects whether multiple warps access the same data at the compilation level. If such a case is identified, the accessing instruction that is run later is guided to the RF instead of SMEM using a proper register renaming technique. While DARSIE prefers a larger number of warps to share the data in RF, recent changes in GPU as well as the warp configurations of popular NVIDIA libraries such as cuDNN, cuBLAS, and CUTLASS suggest otherwise. First, while DARSIE assumes the baseline GPU architecture with an integrated core and RF inside SM, recent GPUs instead have four sub-cores with separate RFs each inside SM (see Fig. 2). Because warps are allocated at sub-core granularity, fewer warps tend to share the data in RF, reducing the opportunity to utilize DARSIE. Second, the number of warps normally held by a single TB is small due to the limitations in the SMEM bandwidth and capacity. CUDA libraries for machine learning tend to make the TB and warp tile sizes large to minimize the redundant memory access. Because the SMEM capacity limits the warp tile size and the number of warps, only four to eight warps exist in a TB, as shown in Table 3. DARSIE can only eliminate redundant

1. The tail effect [38] means that SMs cannot be fully utilized because the number of thread blocks does not evenly fit into the SMs. conv4_1_b, conv4_2_b, conv5_1_a, conv5_1_b, conv5_2_a, and conv5_2_b layers can only use up to 80% of the Tensor core performance due to the tail effect.

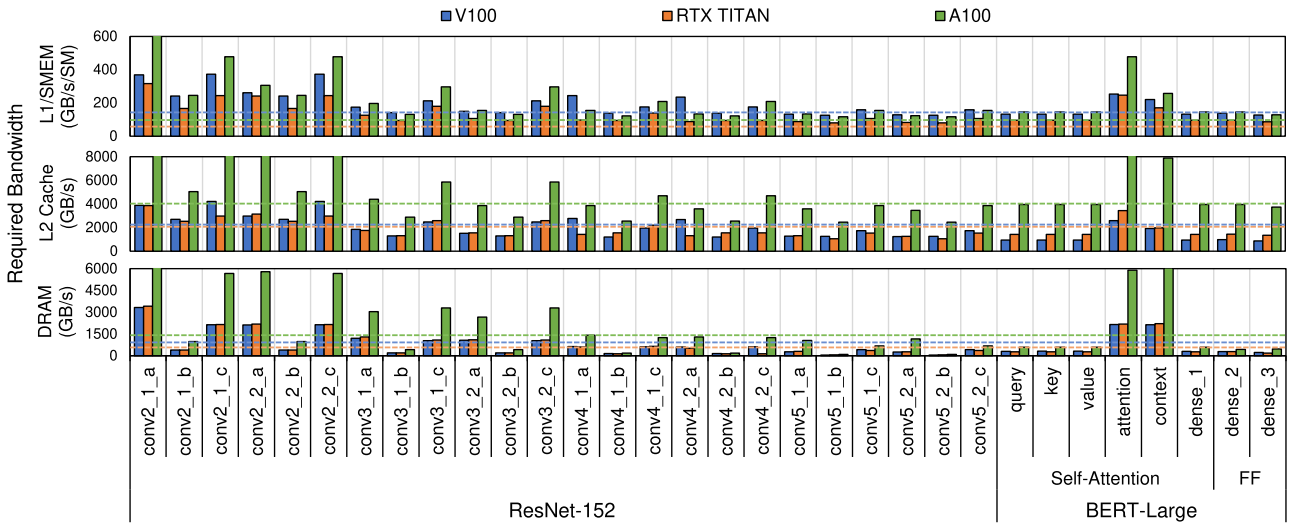


Fig. 11. Required memory bandwidth of each layer using the V100, RTX TITAN, and A100 Tensor cores. The required memory bandwidth is calculated by dividing the memory traffic by the ideal execution time which only considers the computation units. The L1/SMEM, L2 cache, and DRAM (global memory) bandwidths provided by the hardware are represented by the dotted lines.

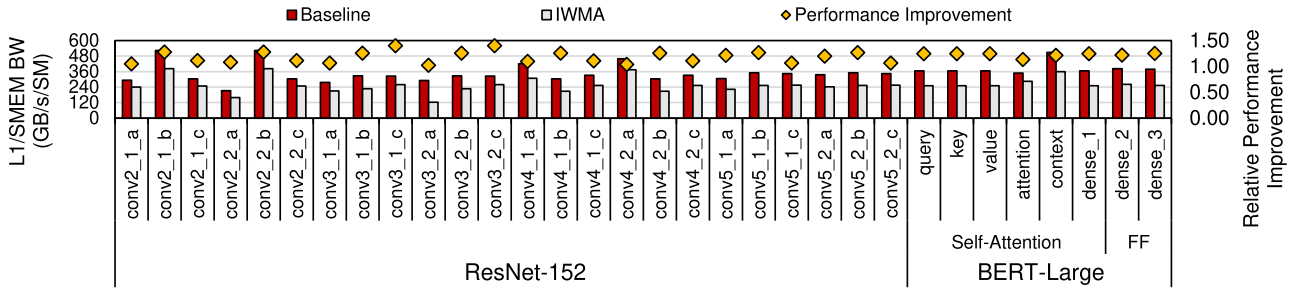


Fig. 12. Comparison of required L1/SMEM bandwidths and performance using the next-generation Tensor core (baseline) and the inter-warp multicasting microarchitecture. We assume that the GPUs provide an unlimited memory bandwidth of the memory hierarchy to find the bandwidth requirements at the best GPU performance.

SMEM accesses when there are eight warps. Thus, when the TB tiles of (256, 128, 32) or (128, 256, 32) are used, DARSIE can only be applied to matrix A. Even worse, when a TB tile with four warps is used, DARSIE is not applicable.

In contrast, the IWMA can reduce redundant SMEM accesses from matrices A and B regardless of the number of warps. Fig. 13 shows the differences in the L1/SMEM traffic of the IWMA and DARSIE when executing the query layer of BERT-Large. DARSIE shows traffic reductions of 24% and 12% compared to the baseline when (256, 128, 32) and (128, 256, 32) TB tiles are used, respectively. However, there is no traffic decrease with different TB tile configurations. In contrast, the IWMA shows an average traffic reduction of 33% in all TB tile cases. The decrease in traffic is always greater in the IWMA compared to DARSIE. Moreover, DARSIE and the IWMA can be used simultaneously. When both techniques are used on TB tiles of (256, 128, 32) and (128, 256, 32), traffic decreases of 47% and 41% are observed, respectively.

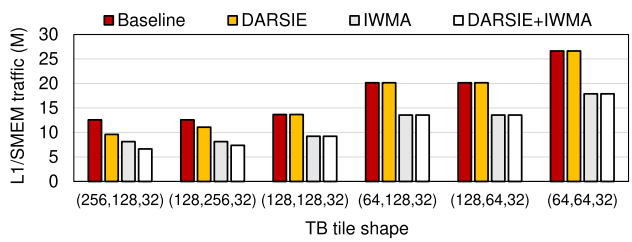


Fig. 13. Comparison of the amounts of L1/SMEM traffics between DARSIE and the IWMA when performing a query layer with BERT-Large.

5.2.3 Area Estimation

We evaluate the area overhead of the IWMA using a 7 nm process [39], [40]. The majority of the area overhead is caused by the load request table (LRT). A single LRT entry stores a warp ID, SMEM address, source register ID, and destination register ID. Because up to 64 warps can exist in an SM, six bits are used for the warp ID. Eight bits and 16 bits are required for the register ID and SMEM address, respectively. Therefore, in total, 38 bits are required per LRT entry. The number of LRT entries is determined by the

TABLE 3
Combinations of Warp Sizes in a Single TB Using the CUDA Library (Based on V100)

TB_M	TB_N	TB_K	$Warp_M$	$Warp_N$	$Warp_K$	# of warps in a single TB
256	128	32	64	64	32	8
128	256	32	64	64	32	8
128	128	32	64	64	32	4
128	64	32	64	64	32	4
64	128	32	64	64	32	4
64	64	32	64	64	32	4

CUDA library, as described in Fig. 4b. Because the maximum number of MLDS instructions used in a single $\_\_syncthreads$ instruction is 384, 192 LRT entries in total are necessary. Therefore, the LRT SRAM size required per SM is 912B, which constitutes 73 KB in total for the entire GPU. For the request queue, the size of a single entry is identical to that of the LRT. Because the request queue must store requests from four different MIOs of each sub-core, eight entries in total are required considering double buffering. This translates to 3 KB for the entire GPU. The size of the various comparators and the decision unit are relatively negligible. To sum up, the area overhead of the IWMA is 0.076 mm^2, which is less than 0.01% of the maximum die size of recent GPUs [16], at 826 mm^2.

6 Related Work

GPU Performance Analysis. Since the publication of important research related to this technology [41], the GPU has been the primary accelerator for DNN models. [42] and [43] analyze the execution time of multiple calculations such as GEMM, FFT, and Winograd on the GPU. [42] breaks down the execution times of CNN models to the level of layer granularity. [43] was the first study to evaluate the Tensor core in addition to the CUDA core. However, both works only report the execution time and do not scrutinize the root causes of the bottlenecks inside the GPU. [37] proposes a GPU performance model through the modeling of CNN memory traffic inside the GPU memory hierarchy. This work carefully investigates the underlying bottleneck of the observed GPU performance for each CNN layer. However, it only provides an analysis for the CUDA core and not for the Tensor core.

Memory Traffic Reduction Using Spatial Locality. There have been several memory access optimization solutions [44], [45] that use inter-warp spatial locality. [44] introduced a new memory coalescing method that detects instances of inter-warp spatial locality. This memory method merges DRAM read and write requests into the L1 cache to exploit spatial locality between multiple warps and increases the effective throughput of the L1 cache. [45] proposed a new memory access scheduling policy that exploits the inter-core locality at the MSHRs of the last level cache. This strategy decreases the DRAM request latency by prioritizing high inter-core locality requests. Although both works focus on redundant DRAM accesses, we eliminate redundant memory accesses between the L1/SMEM and the RF.

7 Conclusion

The adoption of the Tensor core in recent generations of NVIDIA GPUs enabled the impressive acceleration of DNN models in both training and inference. However, the exceptional improvement in the computation performance of the Tensor core newly intensified the memory bandwidth requirements in the existing GPU memory hierarchy. In this paper, we compare the required memory bandwidth of CUDA and Tensor cores when executing DNN models, through which we observed that most Tensor core generations are underutilized due to the bandwidth bottleneck of L1SMEM. Also, we proposed an inter-warp multicasting microarchitecture for the next-generation Tensor core. Inter-warp multicasting reduces the required L1/SMEM bandwidth by 33% and improves the performance by 19% on average in all layers of ResNet and BERT. Throughout detailed experiments, we demonstrated that not only the computational performance, but also the memory bandwidth and non-parallelizable stages of DNN models must be improved to accelerate DNN models further.

References

[1] C. Szegedy et al., "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1–9.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[3] M. Tan and Q. V. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 6105–6114.

[4] Q. Xie, M.-T. Luong, E. Hovy, and Q. V. Le, "Self-training with noisy student improves ImageNet classification," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 10684–10695.

[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.

[6] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018. [Online]. Available: https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf

[7] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," 2020, *arXiv:2004.05150*.

[8] K. Choromanski et al., "Rethinking attention with performers," 2020, *arXiv:2009.14794*.

[9] M. Zaheer et al., "Big bird: Transformers for longer sequences," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, pp. 17283–17297.

[10] J. Choquette, O. Giroux, and D. Foley, "Volta: Performance and programmability," *IEEE Micro*, vol. 38, no. 2, pp. 42–52, Mar./Apr. 2018.

[11] J. Burgess, "RTX on–The NVIDIA turing GPU," *IEEE Micro*, vol. 40, no. 2, pp. 36–44, Mar./Apr. 2020.

[12] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "NVIDIA A100 tensor core GPU: Performance and innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, Mar./Apr. 2021.

[13] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA volta GPU architecture via microbenchmarking," 2018, *arXiv:1804.06826*.

[14] "gpumembench," 2015. [Online]. Available: https://github.com/ekondis/gpumembench

[15] D. C. Burger, J. R. Goodman, and A. Kägi, "Memory bandwidth limitations of future microprocessors," in *Proc. 23rd ACM/IEEE Int. Symp. Comput. Archit.*, 1996, pp. 78–89.

[16] Y. Fu, E. Bolotin, N. Chatterjee, D. Nellans, and S. W. Keckler, "GPU domain specialization via composable on-package architecture," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 4, 2022, Art. no. 4.

[17] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated GPU modeling," in *Proc. 47th ACM/IEEE Int. Symp. Comput. Archit.*, 2020, pp. 473–486.

[18] M. A. Ibrahim, O. Kayiran, Y. E. G. H. Loh, and A. Jog, "Analyzing and leveraging decoupled L1 caches in GPUs," in *Proc. IEEE Int. Symp. High- Perform. Comput. Archit.*, 2021, pp. 467–478.

[19] M. Lee et al., "Improving GPGPU resource utilization through alternative thread block scheduling," in *Proc. IEEE 20th Int. Symp. High- Perform. Comput. Archit.*, 2014, pp. 260–271.

[20] S. Chetlur et al., "cuDNN: Efficient primitives for deep learning," 2014, *arXiv:1410.0759*.

[21] NVIDIA, "cuBLAS," 2007. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html

[22] NVIDIA, "CUTLASS," 2017. [Online]. Available: https://github.com/NVIDIA/cutlass

[23] S. G. Bhaskaracharya, J. Demouth, and V. Grover, "Automatic kernel generation for volta tensor cores," 2020, *arXiv:2006.12645*.
[24] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training multi-billion parameter language models using model parallelism," 2019, *arXiv:1909.08053*.
[25] N. P. Jouppi et al., "Ten lessons from three generations shaped google's TPUv4i : Industrial product," in *Proc. 48th ACM/IEEE Int. Symp. Comput. Archit.*, 2021, pp. 1–14.
[26] Z. Jia, B. Tillman, M. Maggioni, and D. P. Scarpazza, "Dissecting the graphcore IPU architecture via microbenchmarking," 2019, *arXiv:1912.03413*.
[27] S. Dublish, V. Nagarajan, and N. Topham, "Cooperative caching for GPUs," *ACM Trans. Archit. Code Optim.*, vol. 13, 2016, Art. no. 39.
[28] M. A. Ibrahim, O. Kayiran, Y. Eckert, G. H. Loh, and A. Jog, "Analyzing and leveraging shared L1 caches in GPUs," in *Proc. ACM Int. Conf. Parallel Archit. Compilation Techn.*, 2020, pp. 161–173.
[29] A. Arunkumar et al., "MCM-GPU: Multi-chip-module GPUs for continued performance scalability," in *Proc. 44th ACM/IEEE Int. Symp. Comput. Archit.*, 2017, pp. 320–332.
[30] J. Wang, L. Jiang, J. Ke, X. Liang, and N. Jing, "A sharing-aware L1.5D cache for data reuse in GPGPUs," in *Proc. 24th Asia South Pacific Des. Automat. Conf.*, 2019, pp. 388–393.
[31] K. Wang and C. Lin, "Decoupled affine computation for SIMT GPUs," in *Proc. 44th ACM/IEEE Int. Symp. Comput. Archit.*, 2017, pp. 295–306.
[32] T. T. Yeh, R. N. Green, and T. G. Rogers, "Dimensionality-aware redundant SIMT instruction elimination," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2020, pp. 1327–1340.
[33] NVIDIA, "DEVELOPING CUDA KERNELS TO PUSH TENSOR CORES TO THE ABSOLUTE LIMIT ON NVIDIA A100," 2020. [Online]. Available: https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21745-developing-cuda-kernels-to-push-tensor-cores-to-the-absolute-limit-on-nvidia-a100.pdf
[34] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 372–383.
[35] MLCommons, 2018. [Online]. Available: https://mlcommons.org/
[36] lmbench, 2018. [Online]. Available: https://github.com/intel/lmbench
[37] S. Lym, D. Lee, M. O'Connor, N. Chatterjee, and M. Erez, "DeLTA: GPU performance model for deep learning applications with in-depth memory system traffic analysis," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2019, pp. 293–303.
[38] NVIDIA, "GPU performance analysis and optimization," 2012. [Online]. Available: https://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf
[39] A. Shafaei, Y. Wang, X. Lin, and M. Pedram, "FinCACTI: Architectural analysis and modeling of caches with deeply-scaled FinFET devices," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2014, pp. 290–295.
[40] L. T. Clark, V. Vashishtha, D. M. Harris, S. Dietrich, and Z. Wang, "Design flows and collateral for the ASAP7 7 nm FinFET predictive process design kit," in *Proc. IEEE Int. Conf. Microelectronic Syst. Educ.*, 2017, pp. 1–4.
[41] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[42] H. Kim, H. Nam, W. Jung, and J. Lee, "Performance analysis of CNN frameworks for GPUs," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2017, pp. 55–64.
[43] M. Jordà, P. Valero-Lara, and A. J. Peña, "Performance evaluation of cuDNN convolution algorithms on NVIDIA volta GPUs," *IEEE Access*, vol. 7, pp. 70461–70473, 2019.
[44] J. Kloosterman et al., "WarpPool: Sharing requests with inter-warp coalescing for throughput processors," in *Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2015, pp. 433–444.
[45] D. Li and T. M. Aamodt, "Inter-core locality aware memory scheduling," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, pp. 25–28, Jan.–Jun. 2016.

Sunjung Lee received the BS degree in electrical engineering from Chungnam National University, and the MS degree from the Department of Transdisciplinary Studies, Seoul National University. He is currently working toward the PhD degree with the Department of Transdisciplinary Studies, Seoul National University, South Korea. He is interested in micro-architecture, and machine learning system.

Seunghwan Hwang received the BS degree in electronic and electrical engineering from the SungKyunKwan University, in 2020. He is currently working toward the MS degree with the Graduate School of Convergence Science and Technology, Seoul National University. He is interested in hardware architecture for accelerating machine learning system.

Michael Jaemin Kim received the BS degree in electrical and computer engineering from Seoul National University, in 2019. He is currently working toward the PhD degree with the Department of Transdisciplinary Studies, Seoul National University. His research interests include memory system optimization and computer architecture for accelerating emerging applications.

Jaewan Choi received the BS degree in electrical engineering from Seoul National University, Seoul, South Korea, in 2018. He is currently working toward the PhD degree from the Graduate School of Convergence Science and Technology, Seoul National University, Seoul, South Korea. His current research interests include high performance computing and machine learning accelerator.

Jung Ho Ahn (Senior Member, IEEE) received the BS degree in electrical engineering from Seoul National University, and the MS and PhD degrees in electrical engineering from Stanford University, Stanford, California. He is currently a professor with the Graduate School of Convergence Science and Technology, Seoul National University. He is interested in bridging the gap between the performance demand of emerging applications and the performance potential of modern and future massively parallel systems.