Improving GPU Throughput through Parallel Execution Using Tensor Cores and CUDA Cores

Khoa Ho, Hui Zhao, Adwait Jog*, and Saraju Mohanty
Department of Computer Science and Engineering, University of North Texas
*Department of Computer Science, William & Mary
khoaho@my.unt.edu, hui.zhao@unt.edu, ajog@wm.edu, saraju.mohanty@unt.edu,

***Abstract*—To accelerate the execution of Machine Learning applications, recent GPUs use Tensor cores to speed up the general matrix multiplication (GEMM), which is the heart of deep learning. The Streaming Processors in such GPUs also contain CUDA cores to implement general computations. While the Tensor cores can significantly improve the performance of GEMM, the CUDA cores remain idle when Tensor cores are running. This leads to inefficient resource utilization. In this work, we propose to offload part of the GEMM operations from Tensor cores to CUDA cores to fully utilize GPU resources. We investigated the performance bottleneck in such offloading schemes and proposed architectural optimization to maximize the GPU throughput. Our technique is purely hardware-based and does not require a new compiler or other software support. Our evaluation results show that the proposed scheme can improve performance by 19% at the maximum.**

***Index Terms*—Accelerator, GPU, Machine Learning, Tensor core, GEMM, throughput, parallel scheduling**

I. Introduction

In recent years, GPUs have become one of the most widely used accelerators for deep learning, especially following NVIDIA's introduction of Tensor cores in the Volta GPU architecture [1], [16], [22] in 2017. Today, multiple NVIDIA's GPU architectures support Tensor cores, including Volta [1], Turing [3], and Ampere [4]. By trading off some precision, Tensor cores can achieve an order of magnitude of speed-up for general matrix multiplication (GEMM) operations. This leads to a significant acceleration in the overall performance of neural network applications.

A GPU consists of multiple Streaming Multiprocessors (SMs) that run CUDA kernels. For instance, there are 80 SMs in V100 and 108 SMs in A100 NVIDIA's GPUs. Each SM contains thousands of registers, several caches, warp schedulers, and execution cores. CUDA cores exist in all SMs and each CUDA core contains functional units to perform general integer and floating-point operations. Using the V100 GPU as an example, each SM is partitioned into four sub-cores with each sub-core having a single warp scheduler and dispatch unit. Each SM sub-core has its dedicated L0 instruction cache and a branch unit (BRU). In every clock cycle, a sub-core can process one warp instruction and feeds into the shared MIO unit which contains the Texture Cache, L1 Data Cache, and Shared Memory.

The CUDA programming model provides an abstraction of the GPU architecture, acting as a bridge between an application and its implementation on hardware. In a GPU, thousands of threads can run in parallel, and a function executed by different threads at the same time is called a kernel. A kernel launches an array of thread blocks and each thread block is a set of concurrently executing threads that reside in the same SM. Once a thread block is assigned to an SM, it will be further divided into a set of warps. Each group of 32 consecutive threads constitutes a warp which is the primary execution unit in an SM. Each SM contains warp schedulers that are responsible for scheduling the warps to the computing cores.

TABLE I
GPU System Configuration

GPU Type	Volta TitanV	Turing RTX2060
Number of SMs	80	30
Number of CUDA cores	64 per SM (64 FP32, 64 INT32, 32 FP64), 5120 total	64 per SM (64 FP32, 64 INT32, 32 FP64), 1920 total
Number of Tensor cores	8 per SM (work in pair of 2), 640 total	8 per SM (work in pair of 2), 240 total

Designed specifically for deep learning, Tensor cores are recently introduced to NVIDIA's GPUs to accelerate Machine Learning/AI applications. Tensor cores enable mixed-precision matrix multiplication and can greatly improve the performance for neural network training and inference. For instance, a Tesla V100 GPU has 640 Tensor cores in total, with 8 Tensor cores in each of its 80 SMs. Such a single Tensor core can perform 64 half-precision fused-multiply–add (FMA) operations per clock cycle. In total, the 8 Tensor cores in one V100 SM can perform 512 FMAs per clock cycle. Table I shows the numbers of different computing units (cores) in the Volta TitanV and the Turing RTX2060 GPUs.

Current approaches in enhancing Tensor core's performance include: ① faster next generations of Tensor cores, such as the second and third generations in Turing [3] and Ampere architectures [4]; ② multiple modes of operation precision such as F64, TF32, F16, INT8, and INT4 modes to allow for a flexible tradeoff between precision and speed; and ③ optimization in supporting sparse tensors [15], [17], [24]. However, these techniques focused on Tensor cores only. In the current NVIDIA CUDA execution model, one SM can only execute one kernel at a time. If this is a GEMM kernel for a Machine Learning application, the computation will be only allocated to Tensor cores. It has been demonstrated by Zhao et al. [25] that CUDA cores are mostly idle when Tensor cores are running,

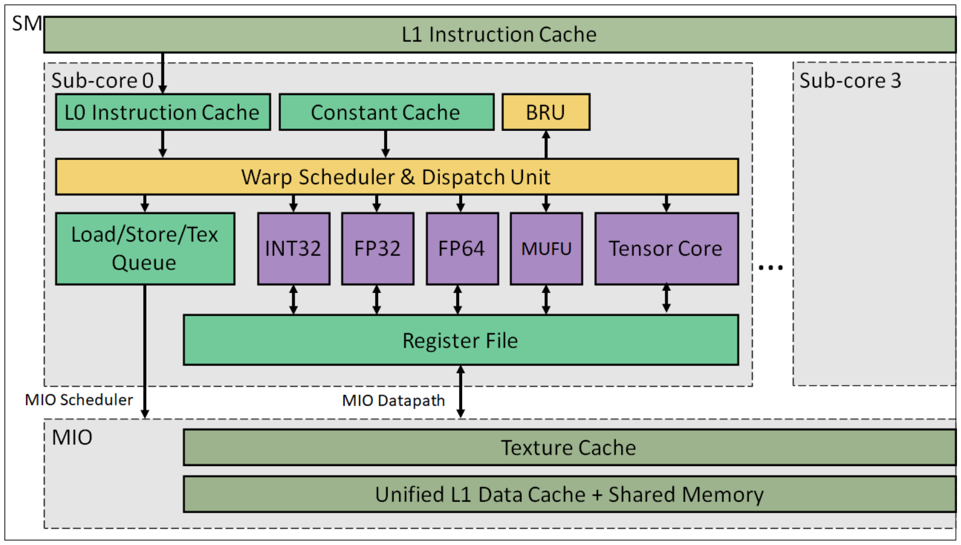

Fig. 1. Volta SM Sub-core Architecture [20].
*Turing uses a similar architecture with different numbers of functional units.

except for occasional light computation such as addressing. This leads to low utilization of the hardware resources in CUDA cores while also incurring power overheads since idling CUDA cores still consume energy.

To solve this problem, some researchers proposed a technique to exploit intra-SM parallelism by running other HPC kernels on CUDA cores while Tensor cores are executing GEMM kernels [25]. However, this technique has several limitations: firstly, compiler support is needed to modify the programming model; secondly, separate HPC applications running on CUDA cores need to be available to share the same SM. Therefore, it does not work if there is only a single machine learning application running on the GPU; thirdly, co-running GEMM and HPC kernels contend for shared SM resources, such as shared memory, which can cause performance degradation in both types of applications.

In this work, we propose to improve the intra-SM resource utilization by offloading part of GEMM kernels' workload from the Tensor cores to the CUDA cores. Our technique can increase CUDA cores' utilization and reduce the GEMM kernel execution time, effectively increasing the GPU's overall throughput. This technique is a hardware-based method and is transparent to the programmers. No compiler support or ISA modification is needed. This technique can also avoid the resource contention issue since only one kernel is running in one SM. We designed the micro-architecture to support the offloading, investigated bottlenecks in the offloading and performed experiments to evaluate its effectiveness.

II. Architectures for Parallel Execution of GEMM Using Tensor and CUDA cores

A. *WMMA API*

The warp-level matrix-multiply-and-accumulate (WMMA) API was introduced in CUDA 9 [2] to enable the programming of GPU Tensor cores [12]. The WMMA API allows GPU programmers to directly use Tensor cores to perform the computation $D = A \times B + C$, where A, B, C, and D are tiles of larger matrices. Threads in a warp cooperatively perform a matrix-multiply and accumulate operation. The size of the tile A, B, C, and D are denoted as $M \times N \times K$, where $M \times K$ is the dimension of tile A, $K \times N$ is the dimension of tile B, and $M \times N$ is the dimension of tile C and tile D. In CUDA 9 with PTX ISA 6.0, the fundamental tile size is 16×16×16. PTX ISA 6.1 introduces more tile size variants, 8x32x16 and 32x8x16 [2].

There are three functions related to WMMA in the CUDA API: load_matrix_sync, store_matrix_sync, and mma_sync. The load_matrix_sync and store_matrix_sync functions load and store part of the input matrices into the registers, so that each thread can access the data. The matrix multiply-accumulate operation is performed by the mma_sync function. The result is an $M \times N$ (e.g., 16×16) tile for the D matrix, which is then saved in the register file. Besides the WMMA API, NVIDIA also provides support in other high-level programming interfaces to program Tensor cores, including cuBLAS [6], cuDNN [7], and CUTLASS [9].

B. *Offloading from Tensor cores to CUDA cores*

As the CUDA cores are mostly idle during the Tensor cores' execution of matrix multiplication instructions within the GEMM kernels, we design an architecture to offload some parts of the workload from the Tensor cores by translating some of the WMMA instructions into multiple MAC instructions and sending them to be scheduled on CUDA cores – particularly, the FP32 compute units. The number of resulting MAC instructions will depend on the matrix-tile-size (m-n-k) of the respective MMA instruction.

For example, an MMA instance in our experiment has a tile size of m16n16k16, i.e., matrix multiplication of two 16x16 square tiles. The whole matrix multiplication operation contains a total of 4096 multiply-accumulate operations. Given that the CUDA cores execute in a SIMD model with 32 threads per warp, this MMA instruction can be translated into 4096/32 = 128 MAC warp instructions. Algorithm 1 shows

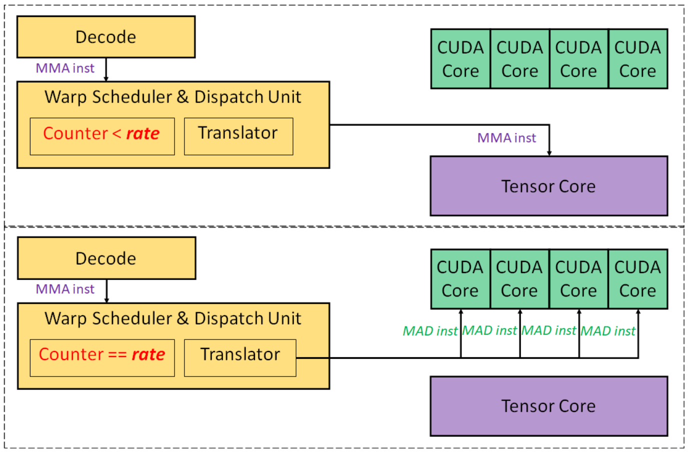

Fig. 2. Scheduling scenarios for Warp Scheduler with Offloading.

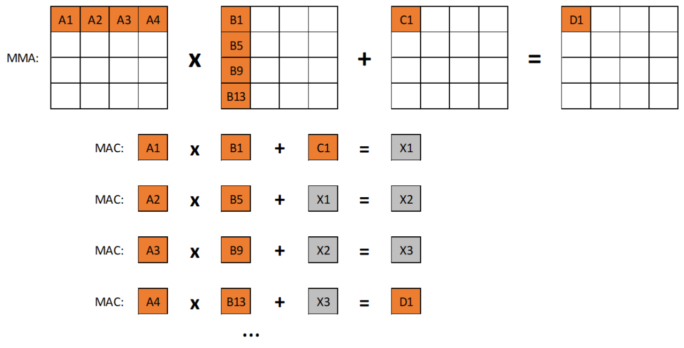

Fig. 3. One MMA instruction to multiple MAC instructions.

the scheduling procedure we designed to offload Tensor cores' work to CUDA cores.

The Warp Scheduler will need two functional sub-units to perform the offloading: a counter and an instruction translator to translate an MMA instruction into multiple MAC instructions working on the same set of registers. Fig. 2 shows two scenarios: one in which the Warp Scheduler schedules an MMA instruction to the Tensor cores in the normal way, and one in which it translates the MMA instruction and sends the resulting MAC instructions to CUDA cores. Fig. 3 illustrates the procedure of translating an MMA instruction into corresponding MAC instructions.

Most GEMM kernels running on Tensor cores use half-precision data type, i.e., FP16, which is also the case in our Cutlass benchmarks. Some newer applications, which arose after the release of Ampere architecture, may also use the smaller INT8 and INT4 data types that are supported by Ampere. CUDA cores use mainly F32 and INT32 operations, so it will need some conversion between the data types. The conversion function is already supported in most current GPUs. NVIDIA's GPUs from the Pascal generation and CUDA 8 already support inherent datatype conversion within the pipelined instruction execution, without the need for separate data conversion operations [11]. For instance, depending on the applications' requirement, Pascal GPUs, which do not have

Algorithm 1: Algorithm for Warp Scheduler to offload task from Tensor cores to CUDA cores

```
Input: mma_inst=decoded mma instruction;
       mma_counter=count of mma instructions since
       last offload
m, n, k = mma_inst.mnk
mma_latency = 64
mac_latency = 4
 // instructions' latencies can change
 depending on hardware architecture
 version
  // the used latencies are from Volta
 TitanV & Turing RTX2060
 configurations on GPGPU-Sim 4.0.1
threads_per_warp = 32
  // NVIDIA has always used 32 threads
 per warp
num_of_mac_inst = (m*n*k) / threads_per_warp
offload_rate = 1 + (num_of_mac_inst * mac_latency) /
 mma_latency
  // in our case, with m = n = k = 16,
 offload_rate = 9
if ((mma_counter+1) == offload_rate) &&
 cuda_cores_is_available() then
    translate_and_issue_to_cuda_cores( mma_inst )
    mma_counter = 0
                     // reset the counter
else
    issue_to_tensor_cores( mma_isnt )
    if mma_counter ¡ offload_rate then
        mma_counter++
          // only increase counter when
        "counter < rate"
          // if "counter==rate" while CUDA
        fp32 pipelines are occupied, then
        issue the MMA to Tensor cores but
        keep the counter
end
```

Tensor cores, already can accept input under FP32, FP16, or INT8 format to produce both FP32 and FP16 output without any impact on performance.

C. Additional load-store unit

Another bottleneck in the execution of the GEMM kernels is the long data path between global memory and shared memory, which causes significant stall time for all computation units, i.e., Tensor cores and CUDA cores. In the Volta and Turing architecture, when loading data from global memory to shared memory, the data must be first loaded into registers before being written into shared memory. Similarly, when writing data from shared memory back to global memory, the data must go through the registers again. Later architectures, starting from Ampere, resolved this issue by designing new direct asynchronous data paths between global and shared

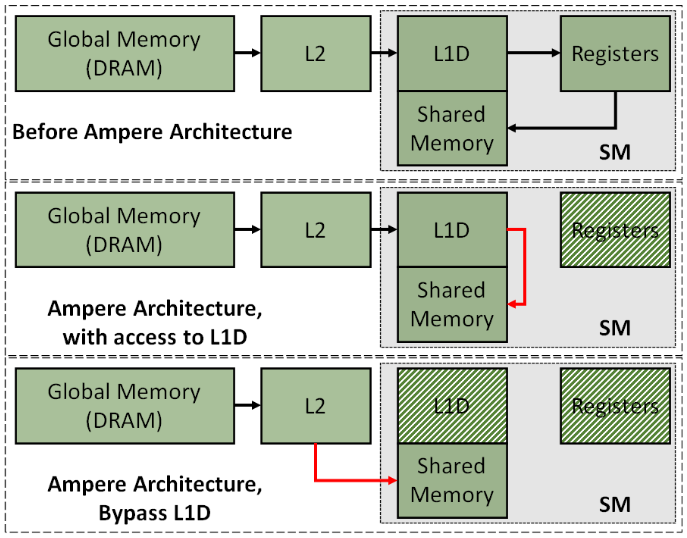
Global Memory (DRAM)
L2
L1D
Registers
Shared Memory
Before Ampere Architecture
SM
Ampere Architecture, with access to L1D
Ampere Architecture, Bypass L1D

Fig. 4. Datapath between Global and Shared Memory before and after Ampere Architecture.

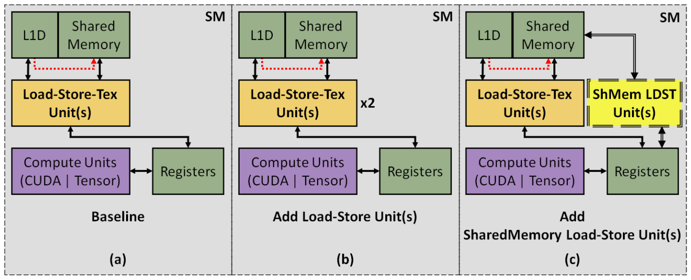
SM
L1D
Shared Memory
Load-Store-Tex Unit(s)
Compute Units (CUDA | Tensor)
Registers
Baseline
(a)
x2
Add Load-Store Unit(s)
(b)
ShMem LDST Unit(s)
Add SharedMemory Load-Store Unit(s)
(c)
*The red dash-link is Ampere architecture's new feature

Fig. 5. Load-Store Unit(s) controlling the Datapath within the SM in the baseline and proposed architectures.

memory [14]. Fig. 4 illustrates the global-shared memory data paths in architectures before and after Ampere.

In our experiments, we found the load-store units are the performance bottleneck in offloading tasks from Tensor cores to CUDA cores as in the baseline design shown in Fig. 5(a). We developed two design optimizations to relieve this bottleneck. In the first design, we added a common ldst-unit, which effectively increases the bandwidth for all memory operations as shown in Fig. 5(b). In terms of hardware, this means adding extra data pipelines parallel to the current ones. The added links between L1D and ldst-units will provide extra bandwidth when data needs to flow through L1D $\rightarrow$ LDST $\rightarrow$ registers to get to shared memory.

However, with Ampere's new direct link between L1D and shared memory, the extra bandwidth between L1D and LDST may not be fully utilized. Due to this reason, we also devised another alternative. We designed a special load-store unit that only handles data transactions between shared memory and registers, which is shown in Fig. 5(c). Only extra data pipelines are needed here between shared memory and registers. This design option is more applicable to Ampere and later architectures because redundant pipelines are removed in those architectures.

To implement our proposed offloading from Tensor cores, extra hardware is needed. This include counters, an MMA instruction converter, and extra links. Extra ldst units are also needed if higher performance is desired and larger performance improvement requires more complicated ldst units.

III. Evaluation

A. Experiment Setup

We used GPGPU-Sim 4.0.1 [18] with CUDA Toolkit and Cutlass 1.3 to simulate the proposed offloading architecture. We have two baseline configurations that are the TitanV, which represents Volta architecture, and the RTX2060 for Turing architecture. We evaluated the proposed technique against each of their baselines separately. Table II shows the configurations we used for our experiments for Volta and Turing architectures respectively. We used the cutlass performance test benchmark – mainly the WMMA-GEMM kernels from Cutlass 1.3 benchmark suite [8] to evaluate the baseline and our proposed architecture performance. GEMM kernels are the building-blocks that carry the most weight of computations in neural network applications. The performance improvement in GEMM kernels' can be used to represent that of the high-level neural network applications.

B. Evaluation Metrics

We evaluated GEMM kernels with square-matrices of variable size, from 128x128 to 2048x2048. The most important evaluation metric in our experiment is normalized performance, which is calculated as the inverse of execution time, normalized against the baseline. Another key metric we used to evaluate the offloading architecture's performance is the occupancy rates of the cores. There are two different occupancy measures that we evaluated: ① occupancy as the ratio of each core's occupied time over the kernel's total execution time. A core's "occupied time" is the time that it is executing an instruction; and ② the occupancy rate as the ratio of the core's occupied time over its "online" time. Here, "online time" is the difference between a core's first activated time and its last instruction completion time. Of those two occupancy rates, the first one is the overall evaluation of performance, whereas the second one can provide some insights into power-related issues. For both metrics, higher values indicate better utilization and less wasted resources.

C. Normalized Performance of GEMM Kernels

Our evaluation results in Fig. 6 show that the offloading architecture achieved up to 5.71% increase in normalized performance for the Volta TitanV GPU. When combined with an additional shared memory side ldst-unit, we can achieve up to 13.07% performance improvement. If we replace the shared-memory side ldst-unit with a general full ldst-unit, up to 29.03% performance improvement can be achieved.

For Turing RTX2060, the offloading design achieved up to 9.07% improvement alone as shown in Fig. 7. Up to 17.77% and 22.72% performance gain is observed when the shared memory ldst-unit and full ldst-unit are applied respectively.

TABLE II
GPU SYSTEM CONFIGURATION

GPU Type	Volta TitanV	Turing RTX2060
Device Limit	Kernel launch latency = 0	Kernel launch latency = 0
SM	80 SMs in 40 clusters, 1.2GHz, 4 sub-cores per SM	30 SMs in 30 clusters, 1.365 GHz, 4 sub-cores per SM
Warp Scheduler	4 W-Schedulers per SM (1 per sub-core), policy: Greedy-Then-Oldest	4 W-Schedulers per SM (1 per sub-core), policy: Greedy-Then-Oldest
Shared Memory	96 KB, limit 64 KB max per thread-block	64 KB, limit 64 KB max per thread-block
Cache	128 KB L1-I-Cache (64 sets/16 ways LRU) per SM, 32 KB L1-D-Cache (1 sets/256 ways LRU) per SM, 96 KB L2-Cache for each memory sub-partition (32 sets/24 ways LRU) (total 4.5 MB L2-Cache)	128 KB L1-I-Cache (64 sets/16 ways LRU) per SM, 64 KB L1-D-Cache (1 sets/512 ways LRU) per SM, 128 KB L2-Cache for each memory sub-partition (64 sets/16 ways LRU) (total 3 MB L2-Cache)
Memory Model	24 Memory Controllers with sub-partition=2, 850 MHz	12 Memory Controllers with sub-partition=2, 3.5GHz
NoC	topology: fly (k=88, n=1), subnets=2, 40-byte flits, dest-tag routing, num of VCs=1, VC buffer size=256	topology: fly (k=52, n=1), subnets=2, 40-byte flits, dest-tag routing, num of VCs=1, VC buffer size=64

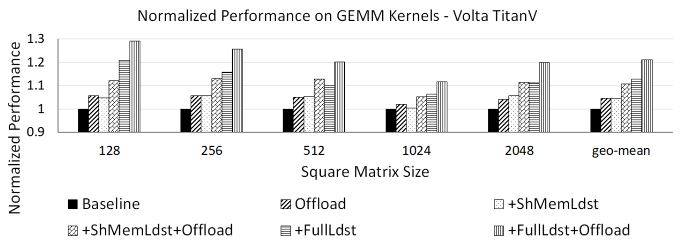

Fig. 6. Normalized Performance on GEMM Kernels – Volta TitanV.

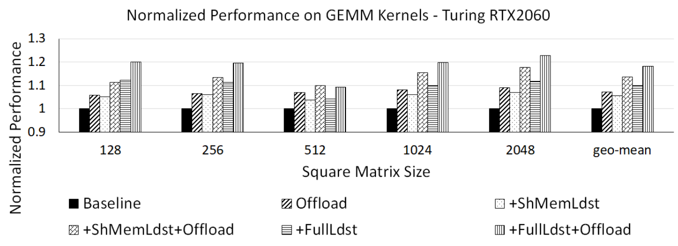

Fig. 7. Normalized Performance on GEMM Kernels – Turing RTX2060.

For the TitanV GPU, the proposed offloading architecture improved performance on average by 4.5%, 10.86%, and 21.16% respectively with no extra ldst-unit, with an extra shared memory ldst-unit, and with a full ldst-unit respectively. For the RTX2060 GPU, the corresponding performance gain is 7.27%, 13.53%, and 18.25%.

We also observed more performance gain when offloading is combined with each kind of the ldst-units as compared with the sum of performance gain from applying the techniques individually. For instance, on the TitanV configuration, the improvement is 12.75% for adding a full ldst-unit and 4.50% for offloading respectively. Therefore, the total improvement is 17.82%. However, combining those two techniques at the same time can improve the performance by 21.16%. This indicates the combination of two techniques can provide more opportunities to improve resource utilization.

D. *Utilization of CUDA cores during GEMM Execution*

Fig. 8 and 9 show the occupancy rates for CUDA cores under the baseline and under the proposed offloading schemes. We observed that the baseline systems' CUDA cores (FP32/SP-units and their pipelines) have a utilization rate close to zero. For larger matrices and longer kernels, this number gets even smaller. We also observe that the CUDA cores' occupancy over its online time can vary a lot depending on the number of "rounds" the selected SM needs to work during the GEMM kernel execution. For larger kernels, the SMs do not have enough resources to complete all tasks in one round, and they need to split the tasks into multiple rounds.

The SMs first activate their SP-units (CUDA cores) near the end of their first round of execution for a small workload – most likely to support the Tensor cores in finalizing their work. However, if the SM has a second round of tasks, the SP-units will stay online and remain idle through most of the second round. If the SM has multiple rounds to execute, the SP-units' idle time will keep increasing while the occupancy rate will keep dropping to near zero.

For our proposed offloading design and with the additional load-store unit, both occupancy measures for the CUDA cores increase significantly. In particular, the CUDA cores' utilization jumped from near-zero to as high as 83.06% for the TitanV, and 94.34% for the RTX2060, with an average of 73.44%, and 72.76% respectively. The CUDA cores' occupancy over their online time also increases significantly, to as high as 97.07% for TitanV, and 95.95% for RTX2060, with an average of 92.62% and 91.11%. That is because the SMs activate their CUDA cores earlier in our design, instead of near the end of their first work round. The CUDA cores are also able to load more instructions to keep executing throughout most of their online time.

IV. RELATED WORK

There have been plenty of works on improving GPU performance [10], [25]–[27]. To improve GPU throughput, Adriaens et al. explored spatial multitasking and proposed to partition GPU stream multiprocessors (SM) among different applications. Their technique works at the inter-SM level [26]. For intra-SM optimization, Zhao et al. explored the opportunity to take advantage of the idling and underutilized CUDA cores during Tensor cores' execution [25].

They proposed a method to improve the utilization of CUDA cores in parallel with Tensor cores by running a non-

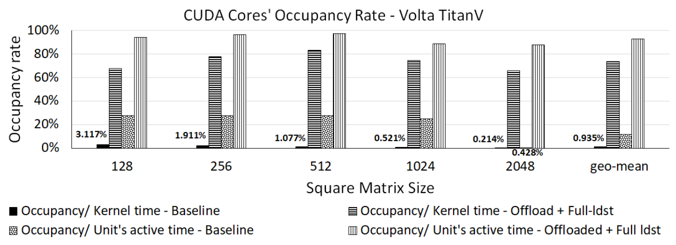

Fig. 8. CUDA cores' occupancy during GEMM Kernels – Volta TitanV.

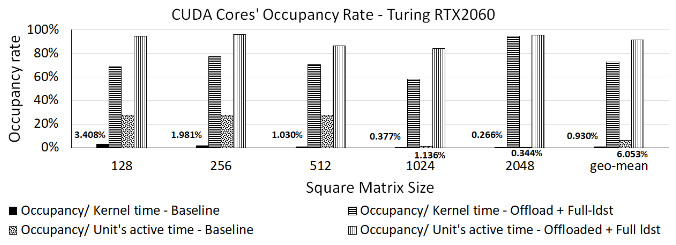

Fig. 9. CUDA cores' occupancy during GEMM Kernels – Turing RTX2060.

GEMM kernel from a different application in parallel with the GEMM kernel on Tensor cores.

Their technique can improve the GPU throughput. However, their technique has limitations in applicability because it requires different types of applications, i.e., both GEMM and non-GEMM types. Furthermore, the multiple applications involved need to have long kernels so that they can be scheduled to run over a long period of time to offset the profiling and scheduling overheads. This limits their techniques to be useful to only data centers. In contrast, our approach of offloading does not need parallel non-GEMM kernels while still improving the GPU throughput. In addition, we do not need compiler or software support for running multiple kernels in the same SM.

V. Conclusion

In this work, we explored schemes to improve the GPU throughput by running GEMM-based applications on Tensor cores and CUDA cores in parallel. We proposed architecture optimization for effective task offloading from Tensor cores to CUDA cores when executing a GEMM kernel. Without modifying software, our technique can achieve a performance improvement by as much as 19.69%.

VI. Acknowledgments

This work is supported in part by NSF Grants 1828105, 2046186, 2008911.

References

[1] NVIDIA, "NVIDIA Tesla V100 GPU Architecture," 2017.
[2] NVIDIA, "CUDA Toolkit Documentation v9.1," 2018.
[3] NVIDIA, "NVIDIA Turing GPU Architecture," 2018.
[4] NVIDIA, "NVIDIA A100 Tensor Core GPU Architecture," 2020.
[5] NVIDIA, "NVIDIA Ampere GA102 GPU Architecture," 2020.
[6] NVIDIA Corporation, "cuBLAS Developer Guide." https://docs.nvidia.com/cuda/cublas/index.html, Aug 2008.
[7] NVIDIA Corporation, "cuDNN Developer Guide." https://docs.nvidia.com/deeplearning/sdk/cudnn-developerguide/index.html, Aug 2014.
[8] NVIDIA, "CUTLASS: CUDA Templates for Linear Algebra Subroutines," v1.3, https://github.com/accel-sim/gpu-app-collection/tree/release/src/cuda/cutlass-bench, 2019.
[9] NVIDIA, "CUTLASS: CUDA Templates for Linear Algebra Subroutines," v2.8, https://github.com/NVIDIA/cutlass, 2022.
[10] X. Cheng, Y. Zhao, H. Zhao and Y. Xie, "Packet Pump: Overcoming Network Bottleneck in On-Chip Interconnects for GPGPUs," Proceedings of the 55th ACM/ESDA/IEEE Design Automation Conference (DAC), 2018.
[11] M. Harris, "Mixed-precision programming with CUDA 8," NVIDIA Developer Technical Blog, https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8, 2016.
[12] J. Appleyard and S. Yokim, "Programming Tensor cores in CUDA 9," NVIDIA Developer Technical Blog, https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9, 2017.
[13] A. Kerr, T. Liu, M. Hagog, J. Demouth, and J. Tran, "Programming Tensor cores: Native Volta Tensor cores with CUTLASS," NVIDIA GPU Tech Conference (GTC), 2019.
[14] M. Tardy and C. Edwards, "Controlling data movement to boost performance on the NVIDIA Ampere architecture," NVIDIA Developer Technical Blog, https://developer.nvidia.com/blog/controlling-data-movement-to-boost-performance-on-ampere-architecture, 2020.
[15] T. Yamaguchi and F. Busato, "Accelerating matrix multiplication with block sparse format and NVIDIA Tensor cores," NVIDIA Developer Technical Blog, https://developer.nvidia.com/blog/accelerating-matrix-multiplication-with-block-sparse-format-and-nvidia-tensor-cores, 2021.
[16] X. Cheng, H. Zhao, M. Kandemir, B. Jiang and G. Mehta, "AMOEBA: a coarse grained reconfigurable architecture for dynamic GPU scaling," Proceedings of the 34th ACM International Conference on Supercomputing (ICS), 2020.
[17] J. Pool, A. Sawarkar, and J. Rodge, "Accelerating inference with sparsity using the NVIDIA Ampere architecture and NVIDIA TensorRT," NVIDIA Developer Technical Blog, https://developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt, 2021.
[18] M. Raihan, N. Goli, and T. Aamodt, "Modeling Deep Learning Accelerator Enabled GPUs," IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2019.
[19] V. Kandiah et al., "AccelWattch: A Power Modeling Framework for Modern GPUs," IEEE/ACM International Symposium on Microarchitecture (MICRO), 2021.
[20] J. Choquette, O. Giroux, and D. Foley, "Volta: Performance and programmability," IEEE Micro, vol. 38, no. 2, pp. 42–52, 2018.
[21] Z. Jia, M. Maggioni, B. Staiger, and D. Scarpazza, "Dissecting the NVIDIA Volta GPU architecture via microbenchmarking," 2018.
[22] X. Cheng, H. Zhao, M. Kandemir, S. Mohanty and B. Jiang, "Alleviating Bottlenecks for DNN Execution on GPUs via Opportunistic Computing," Proceedings of the 21st International Symposium on Quality Electronic Design (ISQED), 2020.
[23] Z. Jia, M. Maggioni, J. Smith, and D. Scarpazza, "Dissecting the NVIDIA Turing T4 GPU via microbenchmarking," 2019.
[24] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse Tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern GPUs," IEEE/ACM International Symposium on Microarchitecture (MICRO), 2019.
[25] H. Zhao, W. Cui, Q. Chen, J. Zhao, J. Leng, and M. Guo, "Exploiting Intra-SM Parallelism in GPUs via Persistent and Elastic Blocks," IEEE International Conference on Computer Design (ICCD), 2021.
[26] J. Adriaens, K. Compton, N. Kim, and M. Schulte, "The case for GPGPU spatial multitasking," IEEE International Symposium on High-Performance Comp Architecture (HPCA), 2012.
[27] X. Cheng, Y. Zhao, M. Robaei, B. Jiang, H. Zhao and J. Fang, "A Low-Cost and Energy-Efficient NoC Architecture for GPGPUs," Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2019.