THEME ARTICLE: Hot Chips

Volta: Performance and Programmability

Jack Choquette, Olivier Giroux, and Denis Foley
NVIDIA

GV100 is NVIDIA’s latest flagship GPU. It has been designed with many features to improve performance and programmability. It features enhancements to NVLink, a redesigned streaming microprocessor (SM), and independent thread scheduling enhancements to the single instruction, multiple threads (SIMT) model. GV100 also adds new tensor cores for an order of magnitude throughput improvement for deep-learning kernels.

A comparison of NVIDIA's previous-generation GP100 with the new GV100 shows a 1.5x to 2x throughput improvement on math, memory, and NVLink interconnect, as well as an order of magnitude improvement in deep-learning throughput—meeting a major design goal (see Figure 1). Designers also had the goal of making the Volta architecture in GV100 performance accessible to programmers by improving programmability and adding many improvements. This article covers a few of those improvements: an enhanced NVLink, a redesigned SM core, independent thread scheduling enhancements to the SIMT model, and tensor cores for deep-learning acceleration.

NVLINK ON VOLTA

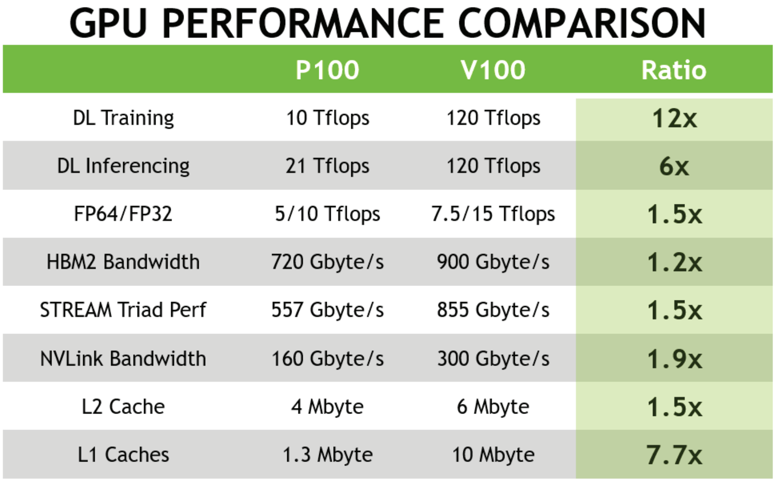
NVLink, as introduced on GP100 in 2016, allowed the Pascal GPU to directly read and write the high-bandwidth memory (HBM) attached to other GPUs (graphics memory, or GMEM) or the large memory pool attached to the IBM POWER 8+ processor (system memory, or SYSMEM). On top of that, Pascal could issue atomic operations to NVLink-connected GPUs where the atomics could be completed at the destination. The relationship between the GPU and the CPU was asymmetric. The CPU could not initiate commands on the NVLink interface and could not access GMEM except over PCI Express (PCIe). If high-bandwidth movement of data between GMEM and SYSMEM were required, the processor would program the GPU’s high-speed copy engines to move data between the memory pools using NVLink.

Shared Address Space

NVLink on Volta and on POWER9 allows both the GPU and the CPU to initiate transactions over the link. GMEM for each GPU and SYSMEM for each CPU socket in a node are combined

Published by the IEEE Computer Society
0272-1732/18/$33.00 ©2018 IEEE

in a single flat address space with equal access and capabilities across each of the processing units. Volta can now issue loads, stores, and atomics to SYSMEM, and POWER9 can issue loads, stores, and atomics to any GMEM in the node. The value of this to a programmer is that all memory is directly accessible. There is still a non-uniform-memory aspect (NUMA) that must be considered, but now that becomes a matter of performance, not correctness. GPU access to its local GMEM is substantially higher bandwidth than to SYSMEM or to other GPUs. CPU latency to GMEM is substantially longer than to SYSMEM. The CUDA runtime driver dynamically determines the best data placement by monitoring CPU and GPU accesses to remote memories (using performance counters). Programmers are also able to take control and explicitly bind and move pages between device memories.



GPU PERFORMANCE COMPARISON

	P100	V100	Ratio
DL Training	10 Tflops	120 Tflops	12x
DL Inferencing	21 Tflops	120 Tflops	6x
FP64/FP32	5/10 Tflops	7.5/15 Tflops	1.5x
HBM2 Bandwidth	720 Gbyte/s	900 Gbyte/s	1.2x
STREAM Triad Perf	557 Gbyte/s	855 Gbyte/s	1.5x
NVLink Bandwidth	160 Gbyte/s	300 Gbyte/s	1.9x
L2 Cache	4 Mbyte	6 Mbyte	1.5x
L1 Caches	1.3 Mbyte	10 Mbyte	7.7x

Figure 1: P100 vs. V100 performance.

Atomics

GPUs and CPUs support various atomic operations. A GPU can issue any of its atomics to a peer GPU where it will be completed at the destination. When a GPU wants to issue an atomic operation to SYSMEM that is not supported by the CPU, it can issue a read-modify-write operation on the link. The CPU provides data to the GPU and defends the block against access by other clients until the GPU has completed the atomic operation locally and returns ownership of the block to the CPU.

Hardware Coherence

Volta NVLink supports a data race free hardware coherence protocol between the GPU and CPU. The CPU can cache GMEM data in its local caches, and the GPU will snoop the CPU as needed to maintain coherence. The GPU caches SYSMEM data in its L1 cache and is guaranteed to be coherent at synchronization points without requiring the CPU to probe the GPU caches. The L1 is write-through, and synchronizing events use a hardware mechanism to invalidate cache entries. This asymmetric approach is a good tradeoff considering the different natures of the CPU and GPU. The CPU is generally running serial workloads that are highly dependent on effective use of its cache hierarchy to maintain good performance, and Volta's NVLink capability allows the caching hierarchy to be used even for GMEM data. On the other hand, the GPU is a throughput-optimized processor, and coherency at synchronization points is a good match to the CUDA programming model.

The GPU uses a simple checkout scheme to track GMEM blocks in the CPU's cache hierarchy. Blocks can be in one of two states—either the block might be in the CPU caches or it is definitely not in the CPU cache. The ambiguity around whether the CPU has the block or not exists because the CPU can drop cache lines that it no longer requires without notifying the GPU.

POWER9's Level-3 (L3) cache is 120 Mbytes. Tracking the state for all the CPU cache in the node would require a very large tag structure. The GPU implements a probe filter that tracks the state of a subset of the cache capacity. All requests—whether generated locally, by a peer GPU, or by the CPU—are checked against the probe filter. Coherence operations are sent over NVLink only for blocks flagged as being potentially in use by the CPU. The probe filter will tend to fill up over time as entries get consumed by cache lines that have been dropped silently by the CPU. If the probe filter were to fill up entirely, each new access by the CPU would result in a probe to the CPU to free up space in the probe filter for the new request. This would add to the latency of the CPU request. Instead, a watermark scheme is implemented so that when the probe filter fills up beyond a programmable threshold, the GPU proactively pulls cache lines back from the CPU, guaranteeing space for new requests by the CPU.

NVLink Address Translation Services

Pascal uses a graphics memory management unit (GMMU) for virtual-to-physical address translation and access controls. The GMMU generates what is referred to as a guest physical address (GPA). When this GPA is passed to the CPU, further translation and access control checks are performed using the CPU's translation control entry (TCE). Volta adds address translation services (ATSs) on the NVLink path to a CPU, where the GPU can issue a request directly to the CPU for a translation using the CPU's page tables. When the GPU misses in its translation lookaside buffers (TLBs), it issues an address translation request (ATR) to the NVLink-attached CPU. The CPU responds with a translation (and pre-fetches a number of related pages), which can then be used by the GPU to generate the appropriate physical address. This system physical address (SPA) can be used directly to access system memory without further translation or checks by the CPU. When the CPU wants to remap a page in use by the GPU, it issues an address translation shoot-down request (ATSD), which is acknowledged by the GPU once all operations in flight that could have been using the translation have resolved. At that point, the TLB entry is invalidated.

Link Bandwidth

NVLink on Volta increases the peak signaling rate from 20 Gbytes/s to 25.78125 Gbytes/s. Each NVLink has eight lanes in each direction. The number of NVLinks is increased from four on GP100 to six on Volta. Consequently, bidirectional bandwidth climbs from 160 Gbytes/s to 309 Gbytes/s, or greater than 1.9x. POWER9 also increased its link count to six compared to the four supported on POWER8+.

Power

NVLink is a free-running interface and consumes power even when there is little or no data to be moved. A power-saving feature was added to GV100 NVLink, allowing for the link width to be reduced from eight lanes in one direction to a single lane. This reduces the power consumed significantly (approximately 25 percent of the free-running power) at the cost of reduced NVLink bandwidth. This power-saving feature can be applied independently in each direction and per gang (the combination of NVLinks connecting two chips) based on programmable activity counters.

SM CORE

Volta's redesigned SM core has made many improvements over the previous generation's Pascal architecture. Volta's SM has twice the instruction schedulers and simplified issue rules. This design makes it easier for a compiler to target and get more utilization and performance from the SM's data paths. Volta's SM has added a large, fast Level-1 (L1) cache for greater application performance, while also allowing the programmer to achieve more performance with less effort. The SM has also improved energy efficiency by 50 percent, which translates to more performance in power-limited workloads.

SM Microarchitecture

The Volta SM core (see Figure 2) is composed of four independently scheduled sub-cores. As with previous generations, the SM performs SIMT scheduling of a Warp (term used in the SM to describe a SIMT group of 32 threads). Each sub-core scheduler can schedule one Warp instruction per clock. Splitting the data path and schedulers into sub-cores maximizes locality and minimizes the power consumption of data paths, register files, and scheduling control. This local independent scheduler also enables a compiler to produce more performance-optimized code by providing it a simple single issue model that doesn't require interleave of multiple Warps for short latencies.

Global memory load and store-cached operations, shared memory (SMEM) scratch pad operations, and Texture (TEX) operations are sent to a shared memory and I/O (MIO) unit. A shared MIO unit provides high and uniform inter-thread communication between sub-cores, enabling efficient, cooperative thread execution. The four sub-cores share an L1 instruction cache that can deliver instructions at four Warp instructions per clock.

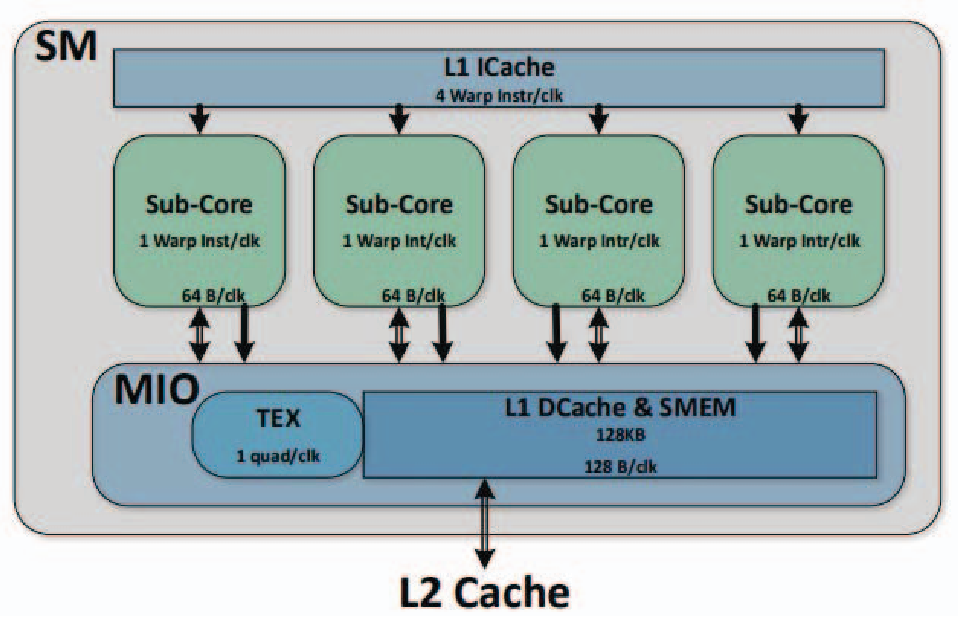


Figure 2: Volta SM core.

SM Sub-Core

The SM Sub-Core (see Figure 3) can issue one Warp instruction per clock from a dedicated Level-0 (L0) instruction cache. Instructions are issued to either the local branch unit (BRU), math dispatch unit, tensor cores, or shared MIO unit.

The math dispatch unit can dispatch instructions at one Warp instruction per clock to four data paths. There are separate data paths for integer instructions (INT), 32-bit floating-point instructions (FP32), 64-bit floating-point instructions (FP64), and miscellaneous transcendental instructions (MUFU). The math dispatch unit is able to keep two or more data paths fully utilized, depending on the mix of instructions in the program being executed. Having different data paths for different common instruction types allows for the data paths to be more power-optimized for the operations they perform. While having separate data paths increases area, having a scheduler design that can keep multiple math data paths busy improves execution efficiency and overall performance per mm^2.

Each SM sub-core also contains two 4x4x4 tensor cores. The Warp scheduler issues matrix multiply operations to the tensor cores for execution. The tensor cores receive input matrices from the register file, perform multiple 4x4x4 matrix multiplies until the full matrix multiply is completed, and write the resulting matrix back into the register file.

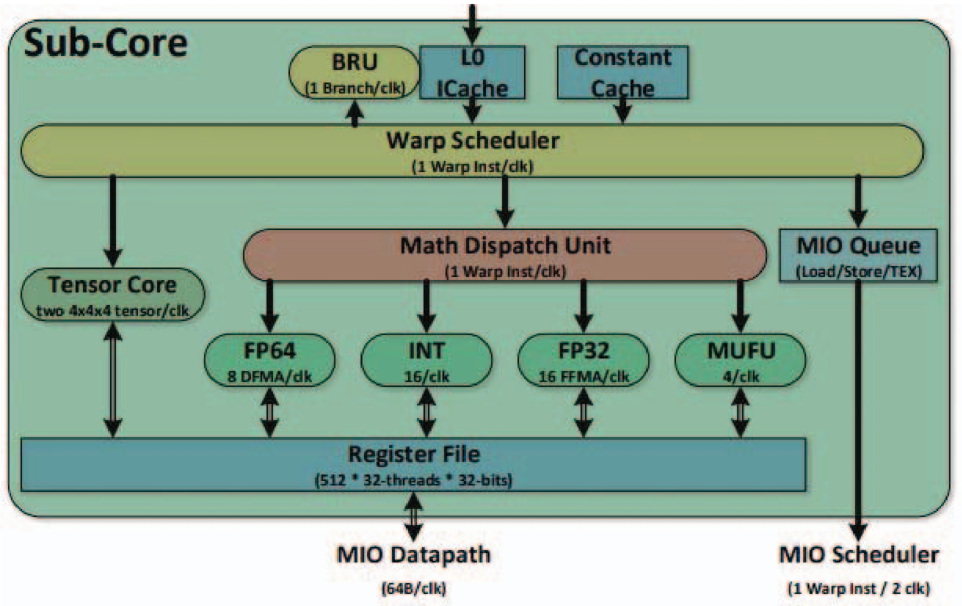


Figure 3: SM sub-core.

L1 CACHE AND SHARED MEMORY

The four SM sub-cores send their MIO instructions to the shared MIO unit for scheduling execution. The MIO scheduler schedules execution on either the texture unit or the unified shared memory and L1 data cache. The TEX unit can execute texture instructions at up to one pixel quad per clock. Data needed for the texture operations is pulled from the L1 streaming data cache.

The L1 data cache is 128 Kbytes and can execute loads and stores for up to 32 threads per clock and provide up to 128 bytes per clock. This is four times the bandwidth and four times the capacity of NVIDIA's previous-generation GP100. The cache is designed as a streaming cache, which enables it to process cache misses and cache hits on outstanding cache misses without stalling, regardless of the address request pattern. For example, the number of misses to a single cache set is not limited by the number of ways. The cache can support the same number of outstanding requests regardless of whether all of the outstanding cache misses and hits on outstanding cache misses are to a single cache set or they are spread across all the sets.

The L1 cache data storage is unified with SMEM data storage, and up to 96 Kbytes of the 128 Kbytes of L1 data storage can be dynamically configured to be used as SMEM. The L1 data cache load and store take the same execution path as SMEM load and store. This provides L1 loads and stores cache hits with the same bandwidth and latency as SMEM load and stores. In GP100 streaming cache design, all requests were processed in order, which caused cache hits to have the same latency as cache misses. This unification with SMEM reduces the latency of cache hits by two orders of magnitude.

In previous generations of GPUs, programs using the SMEM scratchpad provide much better latency and throughput performance than using the L1 cache at the cost of increased programmer effort. With the unification of L1 cache and SMEM scratch pad, the L1 cache has performance characteristics that are much closer to SMEM scratch pad. To estimate the performance benefit of SMEM, a set of applications designed to use SMEM was taken and their SMEM loads and stores were remapped to L1 cache loads and stores. The change of performance was measured (see Figure 4) on both our previous-generation Pascal architecture and on Volta. While the amount of slowdown varied widely by application, Pascal saw an average slowdown of 30 percent. On Volta, with its improved L1 cache design, the application slowdown was dramatically reduced to 7 percent, on average.

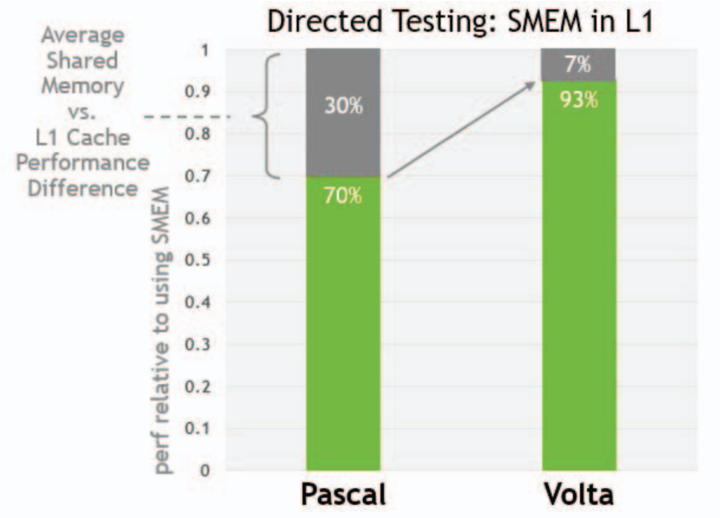


Figure 4: SMEM vs. L1 cache performance.

Using SMEM can still be a good option for some programmers. Their application might have characteristics where SMEM can still provide them a large performance benefit. Examples include making use of SMEM's faster atomics, or SMEM's much better bank conflict avoidance. Programmers might desire the more predictable performance of SMEM's guaranteed low latency and want to avoid the possibility of a cache miss using the L1 cache.

However, the improved L1 cache in Volta greatly reduces the performance difference between using SMEM and the L1 cache, as well as reduces the need to write programs using SMEM for high performance. Many programmers will be able to get reasonably high performance without the additional effort of programming to shared memory.

SUPPORT FOR CONCURRENT PROGRAMMING

The CUDA programming model has its origins in the bulk-synchronous parallel models described by Valiant.[1,2] In these models, computation and communication are independent within each of a series of phases, separated by barrier synchronization. In a practical programming platform with atomic operations, however, this structure defines one part of a larger approach.

Concurrent programming has been a part of GPU programming ever since CUDA 1.1. The Volta architecture advances the practice significantly, by expanding and formalizing the semantics of concurrent algorithms on the CUDA platform.

Independent Thread Scheduling

In a SIMT processor, [3] program threads execute instructions synchronously in a group called a Warp. Execution divergence among the threads of a Warp is handled by executing each path serially, with the observable effect of suspending the execution of some threads, while others execute. The lack of forward-progress for the suspended threads means that SIMT processors have only been able to support lock-free[4] concurrent algorithms, up until now.

The Volta architecture expands the capabilities of SIMT processors to support starvation-free algorithms, including for mutual-exclusion. This is achieved with a completely new mechanism that ensures each thread eventually makes progress, when the compiler identifies visible machine steps as specified by the programming language (for example, in C++ [5]).

Forward-Progress Guarantee Example

A simple spinlock that guards the entry into a critical section provides a good example of the new capabilities of the Volta architecture. Multiple threads can concurrently invoke the function demo_A in the following example, if, and only if, the scheduler ensures each thread is guaranteed to make independent forward-progress. This spinlock is syntactically legal in CUDA C++ 1.1, but is not fully supported until CUDA C++ 9.0 on the Volta architecture:

```
enum { unlocked = 0, locked = 1 };

volatile int mutex = unlocked;

void demo_A() {
  while(atomicCAS(&mutex,unlocked,locked)==locked)
    ;
  /* Among throughput processors, only Volta is
     guaranteed to run this critical section. */
  mutex = unlocked;
}
```

In previous designs, execution divergence optimizations could delay the execution of the critical section by any thread, until the termination of the loop by all threads, causing the program to enter a live-lock state. The execution divergence optimization in the Volta architecture solves this issue by ensuring that no thread is suspended indefinitely. When threads that remain in the loop perform a visible machine step, such as an atomic operation, the scheduler gives other threads an opportunity to make equivalent forward-progress.

Although the example spinlock program is not optimized for scalability, it serves as a litmus test for robustness of a programming system to naïve or optimistic synchronization in user programs. Figure 5 shows a typical result for a CPU architecture next to the Volta GPU architecture's. Notably, in comparison to the CPU, the GPU shows robustness over a wider operational range.

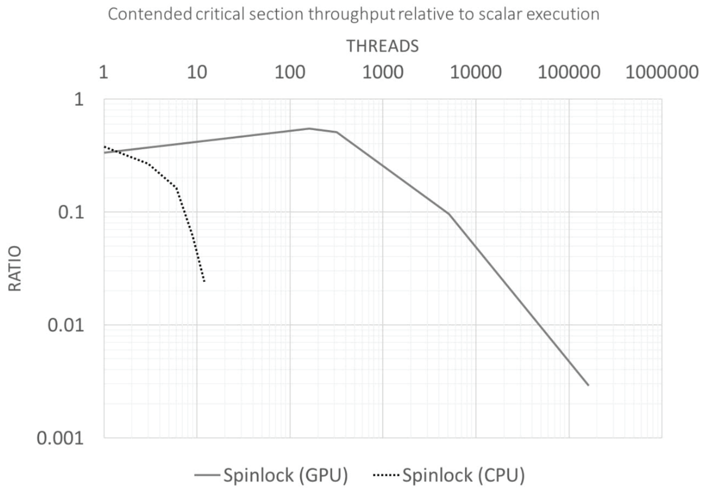


Figure 5: Spinlock executed over page-able memory on x86-64 Linux 4.8.0, i7-5820K, and CUDA 9.0 RC with V100 pre-production hardware.

Flexible Barrier Usage

In CUDA, the most common synchronization is a barrier among all the threads in a block. As a general rule, this barrier implies that if all of the threads in a block invoke it the same number of times, corresponding invocations in each thread synchronize with each other. This is the original idea behind the CUDA C++ 1.0 barrier primitive, dubbed __syncthreads(), as used in the demo_B function below:

```
extern bool is_odd(int);
extern bool is_even(int);

volatile bool is_sound = true;

void demo_B() {
  const int x = threadIdx.x;
  if(is_odd(x) || is_even(x)) {
    assert(is_sound);
  /* The assertion does not fire on Volta, but
     may fire on other throughput processors. */
    __syncthreads();
    is_sound = false;
  }
}
```

The question of this function's soundness relates to an additional restriction placed on synchronization in CUDA, and similar parallel systems, specifically the prohibition on thread invocations of __syncthreads() when Warp execution is diverged. Guidelines on this subject are usually incomplete and often missing. For example, demo_B is potentially made unsound by the short-circuit control-flow of the logical operators ||. The introduction of independent thread scheduling in Volta allows us to eliminate this prohibition and the need to reason about Warp execution divergence to determine if a program is sound.

Improved Execution Convergence

Some programs have the potential for more convergent execution than any implementation is able to exploit. Consider this program with statically indeterminate execution divergence:

```
extern bool is_equal(int, int);

void demo_C() {
  for(int i = 0;i < 1024; ++i)
      if(is_equal(threadIdx.x, i)) {
  /* Other throughput processors run this block with
     execution divergence, but Volta may avoid it. */
      }
}
```

The function demo_C might execute slowly on a SIMT processor, because threads serialize on the execution of the condition's body. A prediction that the innermost block is likely to be executed by all threads is not sufficient to avoid this serialization, and in many cases, a proof is required. The Volta architecture enables convergence optimization to be used speculatively for cases like this.

A final class of programs needs firm guarantees on convergent execution, however. The program shown after this paragraph efficiently exchanges values between threads using a shuffle operation. The function demo_D aggregates data from an entire Warp using the Cooperative Groups[6] facility introduced in CUDA 9.0. The explicit group abstraction conveys the high-level program information, which the implementation (hardware and software) needs to provide the guarantees the program expects. This eliminates the guesswork required to apply Warp optimizations and makes the high-performance programmer more productive. Execution path re-convergence can be a seen as performance optimization, while explicit synchronization within a Warp remains both supported and fast.

```
#include <cooperative_groups.h>

int demo_D(int *ptr) {
    cg::coalesced_group g = cg::coalesced_threads();
    int prev;
  /* Elect the first active thread to perform atomic add. */
    if(g.thread_rank() == 0)
        prev = atomicAdd(ptr, g.size());
  /* Broadcast previous value within the Warp
     and add each active thread's rank to it. */
    prev = g.thread_rank() + g.shfl(prev, 0);
    return prev;
}
```

TENSOR CORES FOR DEEP LEARNING

The Pascal architecture delivered considerably higher performance for training neural networks compared to the prior-generation NVIDIA Maxwell and Kepler architectures, but the complexity and size of neural networks have continued to grow. New networks with thousands of layers and millions of neurons demand even higher performance and faster training times. New tensor cores are a key capability enabling the Volta GV100 GPU architecture to deliver the performance required to train large neural networks.

Tensor Cores

The GV100 GPU contains 640 tensor cores: eight per SM. In Volta GV100, each tensor core performs 64 floating-point operations per clock, and eight tensor cores in an SM perform a total of 1,024 floating-point operations per clock. GV100's tensor cores deliver up to 120 tensor Tflops for training and inference applications. Tensor cores provide up to 12x higher peak Tflops on GV100 that can be applied to deep-learning training compared to using standard FP32 operations on GP100. For deep-learning inference, Volta's tensor cores provide up to 6x higher peak Tflops compared to standard FP16 operations on Pascal.

Tensor cores and their associated data paths are custom-designed to dramatically increase floating-point compute throughput with high energy-efficiency. Each tensor core operates on a 4x4 matrix and performs the following operation:

$D = A \times B + C$

where A, B, C, and D are 4x4 matrices. The matrix multiply inputs A and B are FP16 matrices, while the accumulation matrices C and D may be FP16 or FP32 matrices. The computational density of this operation is very high; with three relatively small 4x4 matrices of multiply and accumulator data, 64 multiply-add operations can be performed. This enables the tensor core to perform matrix multiplications very efficiently in terms of area and power. Tensor cores operate on FP16 input data with FP32 accumulation. The FP16 multiply results in a full precision product that is then accumulated using FP32 addition with the other intermediate products for a 4x4x4 matrix multiply. In practice, tensor cores are used to perform much larger 2D or higher-dimensional matrix operations, built up from these smaller elements.

DEEP-LEARNING PERFORMANCE AND PROGRAMMABILITY

General matrix multiplication (GEMM) operations are at the core of neural-network training and inferencing, and are used to multiply large matrices of input data and weights in the connected layers of the network. Figure 6 shows that for the case of matrix operations with FP16 inputs and FP32 accumulation, Volta's mixed-precision tensor cores with CUDA 9 help boost V100's performance by more than 9x over P100.

CUDA basic linear algebra subroutine (CuBLAS) and CUDA DNN (cuDNN) libraries have been updated to provide new library interfaces to make use of tensor cores for deep-learning applications and frameworks. NVIDIA has worked with many popular deep-learning frameworks such as Caffe2 and Apache MXNet to enable use of tensor cores for deep-learning research on Volta GPU-based systems. NVIDIA is working to add support for tensor cores in other frameworks, as well. Integration of tensor-core library routines into the frameworks provides easy out-of-the-box, high-performance deep learning on Volta.

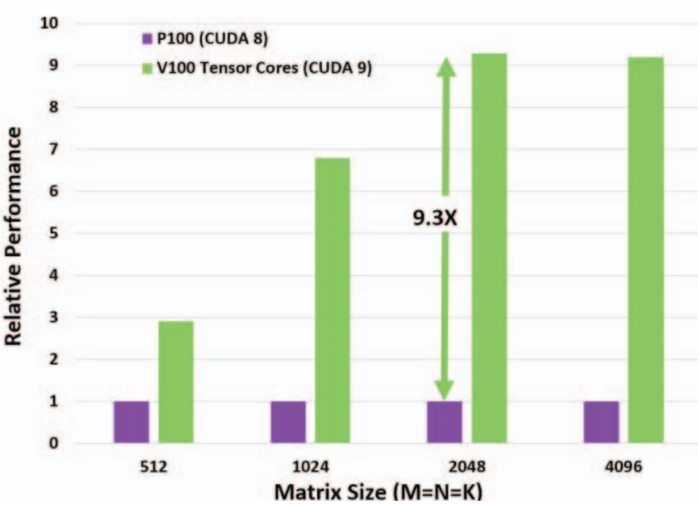


Figure 6: cuBLAS mixed precision (FP16 input, FP32 compute).

In addition to libraries and frameworks, the Volta tensor cores are accessible and exposed as Warp-level matrix operations in the CUDA 9 C++ API. [7] The API exposes specialized matrix load, matrix multiply and accumulate, and matrix store operations to efficiently use tensor cores from a CUDA-C++ program. [8] At the CUDA level, the Warp-level interface assumes 16x16 size matrices spanning all 32 threads of the Warp. Programmability and discovery of new machine-learning techniques are critical for pushing the boundaries of what can be learned by neural networks. The CUDA C++ interfaces for tensor cores enable researchers and developers to invent new high-performance techniques that are not suited for standard matrix library routines. As highlighted by Figure 7, the state of the art for deep learning is evolving rapidly, enabled by custom development in CUDA.

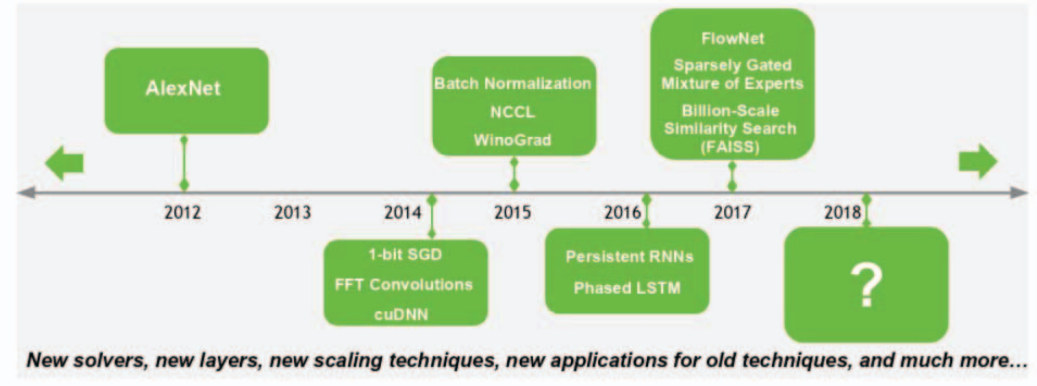

AlexNet
Batch Normalization
NCCL
WinoGrad
FlowNet
Sparsely Gated Mixture of Experts
Billion-Scale Similarity Search (FAISS)
2012
2013
2014
2015
2016
2017
2018
1-bit SGD
FFT Convolutions
cuDNN
Persistent RNNs
Phased LSTM
?
New solvers, new layers, new scaling techniques, new applications for old techniques, and much more...

Figure 7: Deep-learning methods developed using CUDA.

CONCLUSION

GV100 is NVIDIA's highest-performance processor to date. The improved NVLink, the redesigned SM core, and independent thread scheduling have dramatically improved the performance, efficiency, and programmability for HPC and deep-learning applications. We have dramatically expanded the space of concurrent GPU programs and eliminated long-standing confusion between convergence optimizations and synchronization between SIMT threads. The addition of the tensor core also provides an order of magnitude throughput improvement for deep-learning kernels.

REFERENCES

1. L.G. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM*, vol. 33, no. 8, 1990; https://dl.acm.org/citation.cfm?id=79181.
2. L.G. Valiant, "A Bridging Model for Multi-Core Computing," *Journal of Computer and System Sciences*, vol. 77, no. 1, 2011; https://dl.acm.org/citation.cfm?id=1889509.
3. E. Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, 2008, pp. 39–55; www.computer.org/csdl/mags/mi/2008/02/mmi2008020039-abs.html.
4. M. Herlihy and N. Shavit, "On the Nature of Progress," *Principles of Distributed Systems*, Springer Berlin Heidelberg, 2012.
5. *Forward progress (4.7.2)*, ISO/IEC DIS 14882: Programming Languages — C++, International Organization for Standardization, Geneva, Switzerland; http://eel.is/c++draft/intro.progress.
6. *Parallel Forall*, Cooperative Groups: Flexible CUDA Thread Programming; https://devblogs.nvidia.com/parallelforall/cooperative-groups/.
7. "Warp matrix functions (B.16)," *CUDA C Programming Guide*; http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma.
8. *CUTLASS: Fast Linear Algebra in CUDA C++*; https://devblogs.nvidia.com/parallelforall/cutlass-linear-algebra-cuda/.

ABOUT THE AUTHORS

Jack Choquette is a Senior Distinguished Engineer at NVIDIA, where he has led the architecture development of NVIDIA's GPGPU streaming microprocessors for multiple generations. He has been doing CPU and system designs for more than 20 years and has held senior architecture positions for some of the highest-volume MIPS embedded processors, as well as the innovative Azul 768-core symmetric multiprocessing (SMP) Vega network-attached Java processor and system. Choquette has a master's degree in computer engineering from the University of Illinois at Urbana-Champaign. Contact him at jchoquette@NVIDIA.com.

Olivier Giroux is a principal engineer in the GPU architecture group at NVIDIA. He has worked on eight GPU and four SM architecture generations, where he focused on clarifying the programming model of GPU threads. Giroux has a master's degree in computer science from McGill University. Contact him at ogiroux@NVIDIA.com.

Denis Foley is a senior director in NVIDIA's GPU architecture team, where he leads the architectural development of NVLink. Previously, he was a senior fellow at AMD (and the lead engineer on AMD's initial APUs), a system architect for ATI's Imageon applications processors, and the implementation lead for a 128-processor SMP system at Hewlett Packard. Foley has a bachelor's degree in engineering from University College Cork, Ireland. Contact him at dfoley@NVIDIA.com.