Whitepaper

NVIDIA's Next Generation CUDA™ Compute Architecture: Kepler™ GK110

The Fastest, Most Efficient HPC Architecture Ever Built

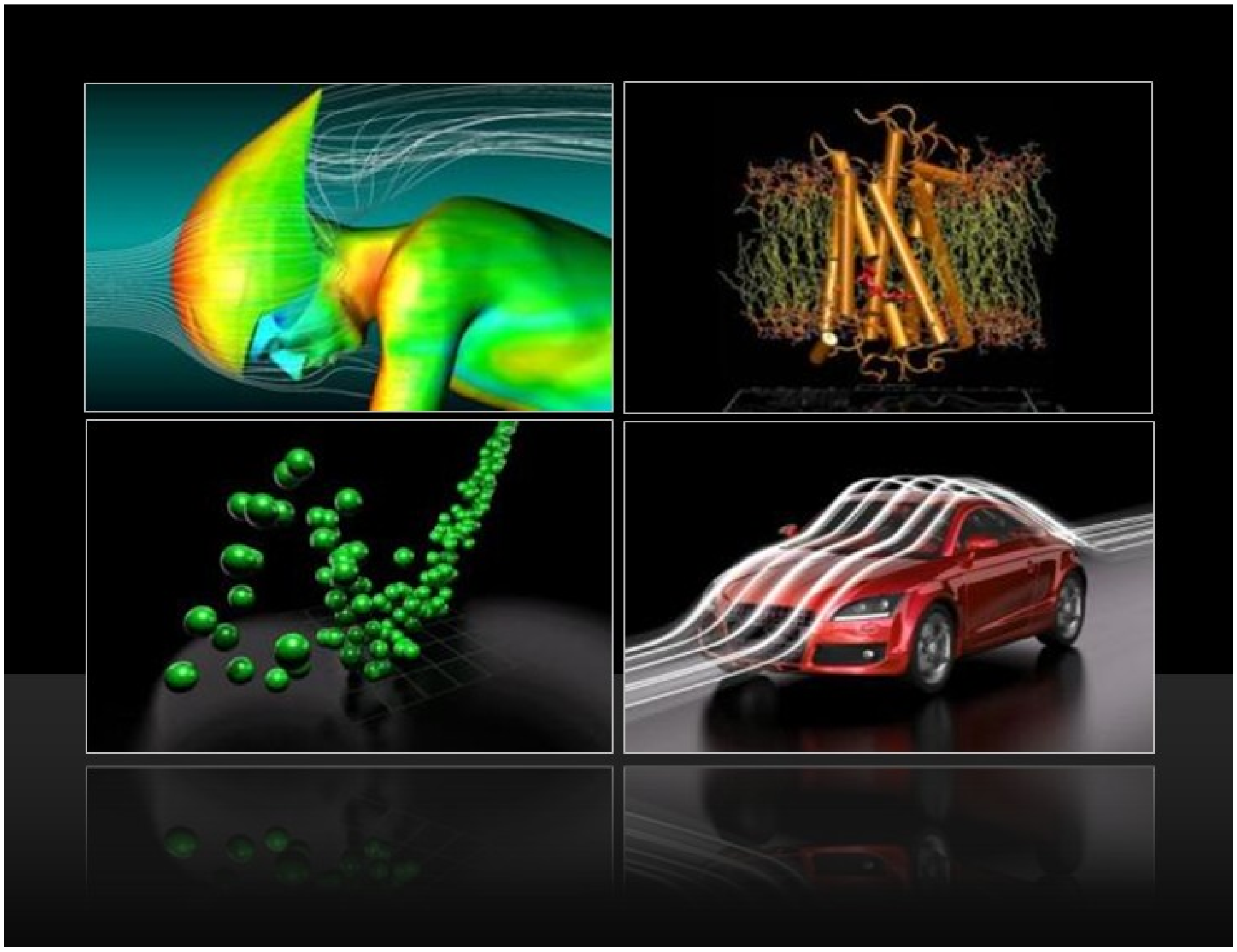
V1.0

Table of Contents

Kepler GK110 – The Next Generation GPU Computing Architecture

As the demand for high performance parallel computing increases across many areas of science, medicine, engineering, and finance, NVIDIA continues to innovate and meet that demand with extraordinarily powerful GPU computing architectures. NVIDIA's existing Fermi GPUs have already redefined and accelerated High Performance Computing (HPC) capabilities in areas such as seismic processing, biochemistry simulations, weather and climate modeling, signal processing, computational finance, computer aided engineering, computational fluid dynamics, and data analysis. NVIDIA's new Kepler GK110 GPU raises the parallel computing bar considerably and will help solve the world's most difficult computing problems.

By offering much higher processing power than the prior GPU generation and by providing new methods to optimize and increase parallel workload execution on the GPU, Kepler GK110 simplifies creation of parallel programs and will further revolutionize high performance computing.

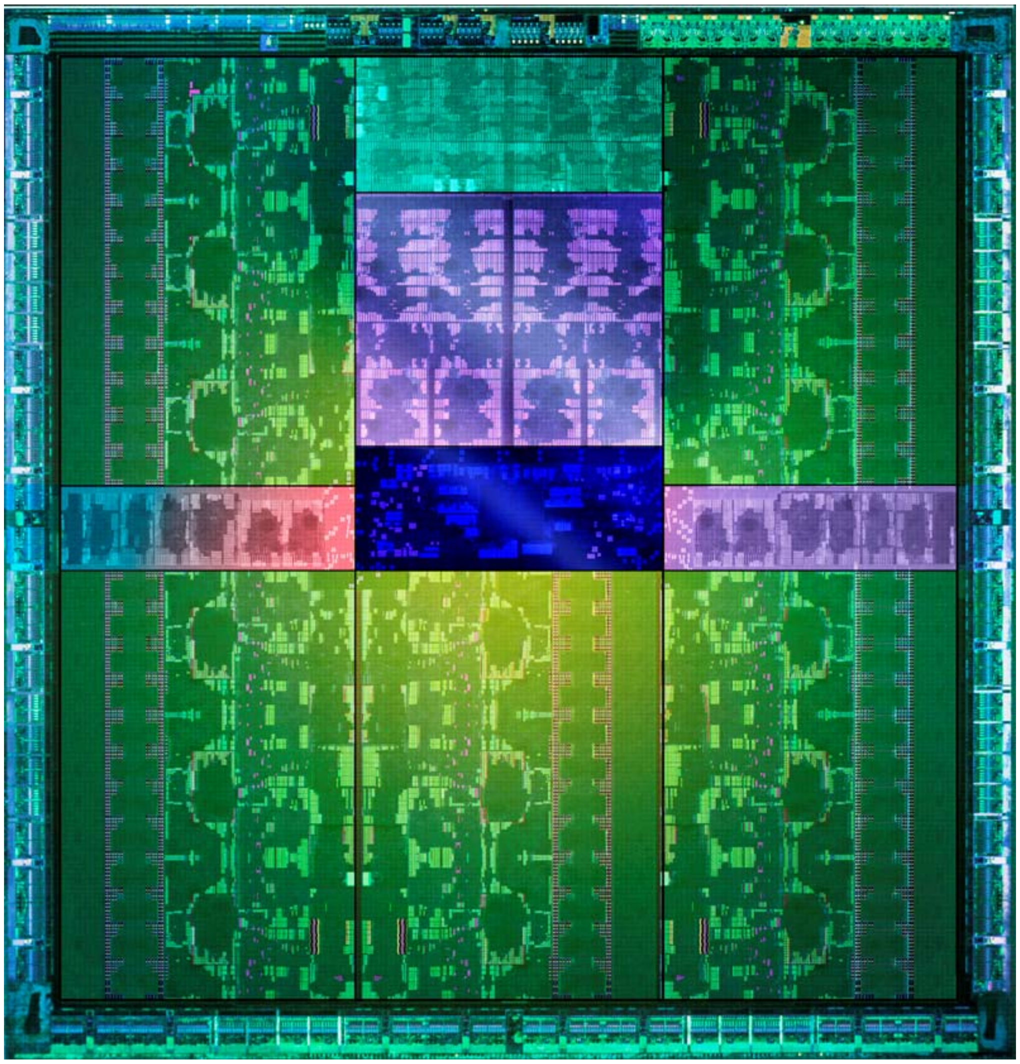


Kepler GK110 - Extreme Performance, Extreme Efficiency

Comprising 7.1 billion transistors, Kepler GK110 is not only the fastest, but also the most architecturally complex microprocessor ever built. Adding many new innovative features focused on compute performance, GK110 was designed to be a parallel processing powerhouse for Tesla® and the HPC market.

Kepler GK110 will provide over 1 TFlop of double precision throughput with greater than 80% DGEMM efficiency versus 60-65% on the prior Fermi architecture.

In addition to greatly improved performance, the Kepler architecture offers a huge leap forward in power efficiency, delivering up to 3x the performance per watt of Fermi.



Kepler GK110 Die Photo

The following new features in Kepler GK110 enable increased GPU utilization, simplify parallel program design, and aid in the deployment of GPUs across the spectrum of compute environments ranging from personal workstations to supercomputers:

- **Dynamic Parallelism** – adds the capability for the GPU to generate new work for itself, synchronize on results, and control the scheduling of that work via dedicated, accelerated hardware paths, all without involving the CPU. By providing the flexibility to adapt to the amount and form of parallelism through the course of a program's execution, programmers can expose more varied kinds of parallel work and make the most efficient use the GPU as a computation evolves. This capability allows less-structured, more complex tasks to run easily and effectively, enabling larger portions of an application to run entirely on the GPU. In addition, programs are easier to create, and the CPU is freed for other tasks.

- **Hyper-Q** – Hyper-Q enables multiple CPU cores to launch work on a single GPU simultaneously, thereby dramatically increasing GPU utilization and significantly reducing CPU idle times. Hyper-Q increases the total number of connections (work queues) between the host and the GK110 GPU by allowing 32 simultaneous, hardware-managed connections (compared to the single connection available with Fermi). Hyper-Q is a flexible solution that allows separate connections from multiple CUDA streams, from multiple Message Passing Interface (MPI) processes, or even from multiple threads within a process. Applications that previously encountered false serialization across tasks, thereby limiting achieved GPU utilization, can see up to dramatic performance increase without changing any existing code.

- **Grid Management Unit** – Enabling Dynamic Parallelism requires an advanced, flexible grid management and dispatch control system. The new GK110 Grid Management Unit (GMU) manages and prioritizes grids to be executed on the GPU. The GMU can pause the dispatch of new grids and queue pending and suspended grids until they are ready to execute, providing the flexibility to enable powerful runtimes, such as Dynamic Parallelism. The GMU ensures both CPU- and GPU-generated workloads are properly managed and dispatched.

- **NVIDIA GPUDirect™**– NVIDIA GPUDirect™ is a capability that enables GPUs within a single computer, or GPUs in different servers located across a network, to directly exchange data without needing to go to CPU/system memory. The RDMA feature in GPUDirect allows third party devices such as SSDs, NICs, and IB adapters to directly access memory on multiple GPUs within the same system, significantly decreasing the latency of MPI send and receive messages to/from GPU memory. It also reduces demands on system memory bandwidth and frees the GPU DMA engines for use by other CUDA tasks. Kepler GK110 also supports other GPUDirect features including Peer-to-Peer and GPUDirect for Video.

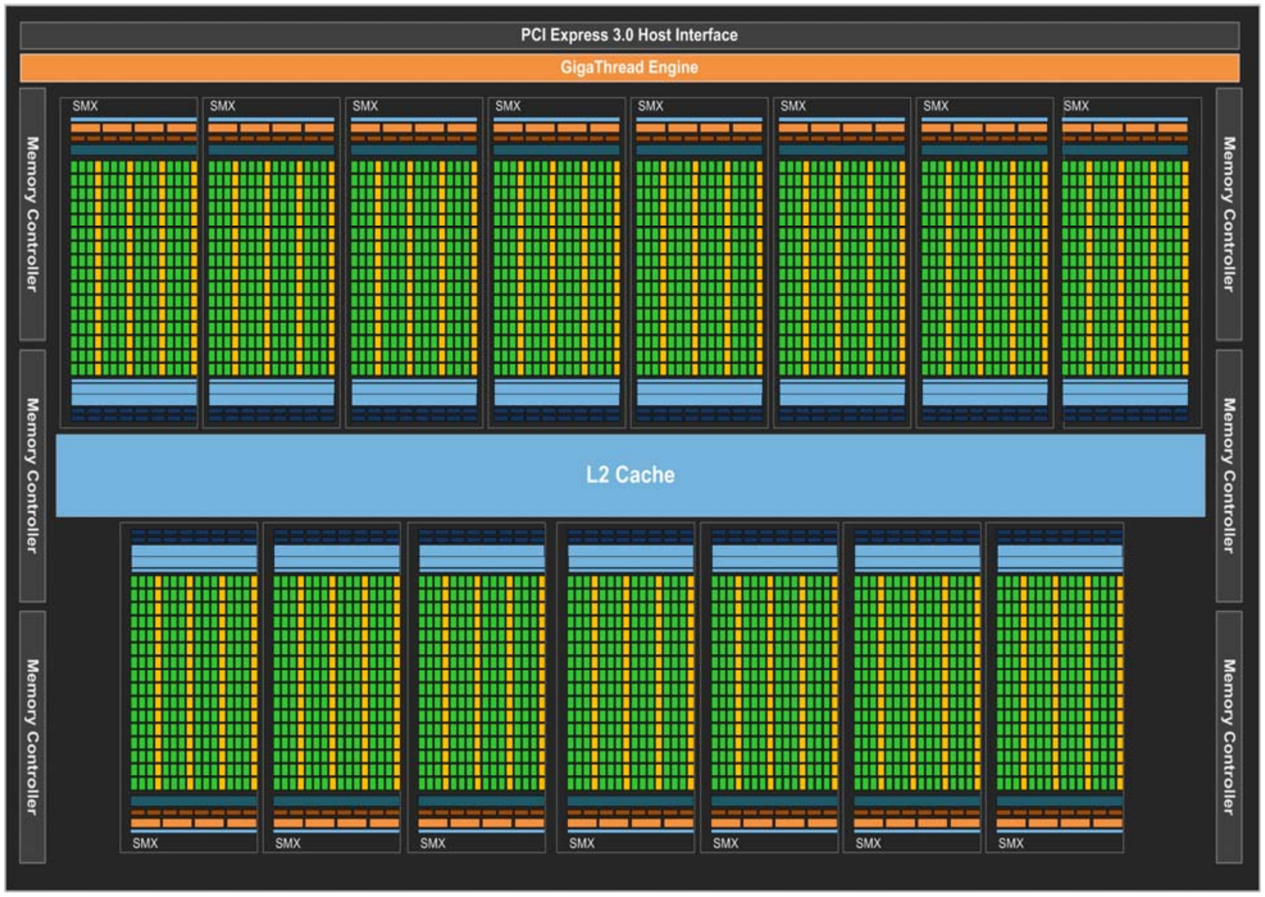
An Overview of the GK110 Kepler Architecture

Kepler GK110 was built first and foremost for Tesla, and its goal was to be the highest performing parallel computing microprocessor in the world. GK110 not only greatly exceeds the raw compute horsepower delivered by Fermi, but it does so efficiently, consuming significantly less power and generating much less heat output.

A full Kepler GK110 implementation includes 15 SMX units and six 64-bit memory controllers. Different products will use different configurations of GK110. For example, some products may deploy 13 or 14 SMXs.

Key features of the architecture that will be discussed below in more depth include:

- The new SMX processor architecture
- An enhanced memory subsystem, offering additional caching capabilities, more bandwidth at each level of the hierarchy, and a fully redesigned and substantially faster DRAM I/O implementation.
- Hardware support throughout the design to enable new programming model capabilities



Kepler GK110 Full chip block diagram

Kepler GK110 supports the new CUDA Compute Capability 3.5. (For a brief overview of CUDA see ***Appendix A - Quick Refresher on CUDA***). The following table compares parameters of different Compute Capabilities for Fermi and Kepler GPU architectures:

	FERMI GF100	**FERMI GF104**	**KEPLER GK104**	**KEPLER GK110**
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

Compute Capability of Fermi and Kepler GPUs

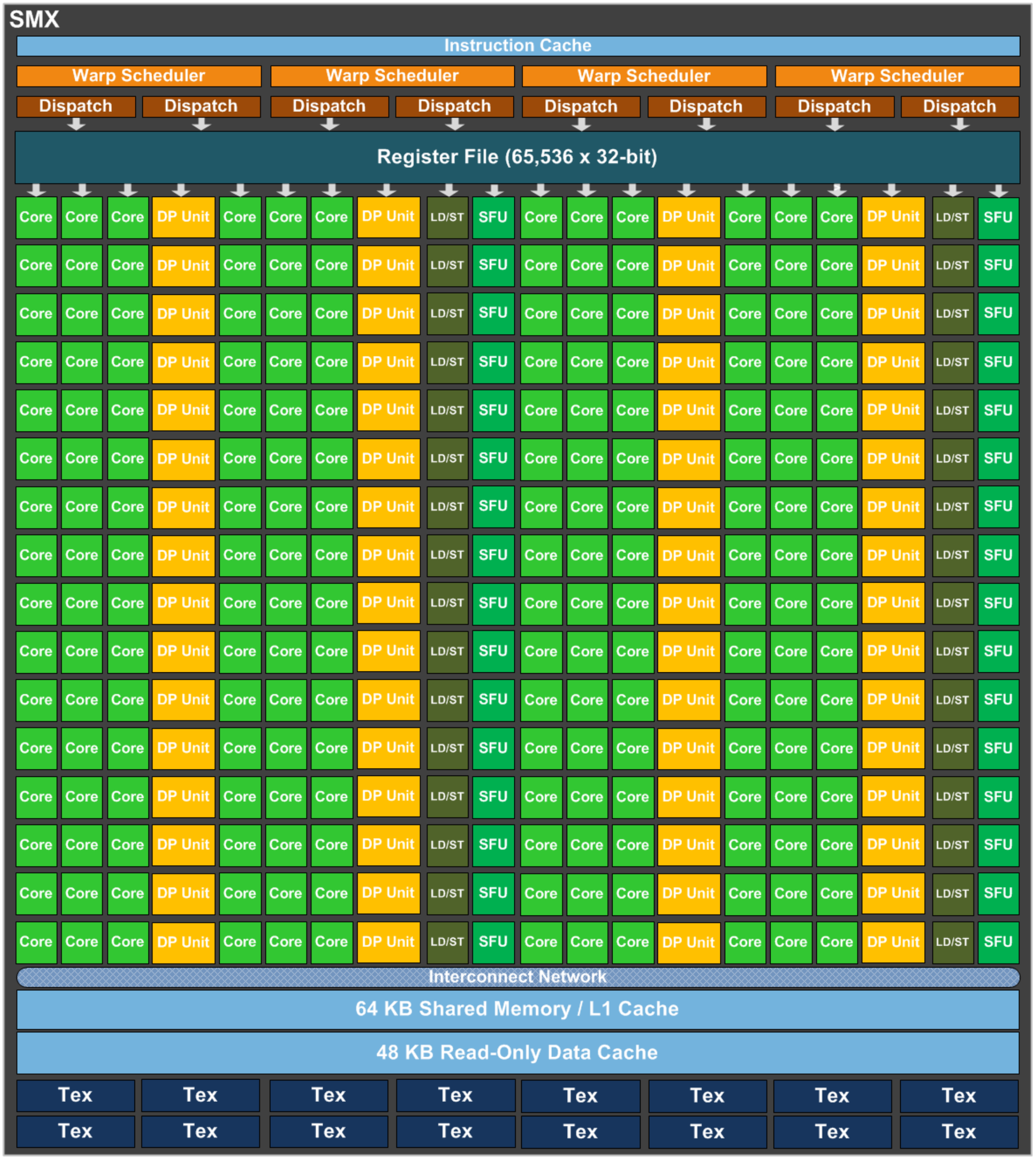
Performance per Watt

A principal design goal for the Kepler architecture was improving power efficiency. When designing Kepler, NVIDIA engineers applied everything learned from Fermi to better optimize the Kepler architecture for highly efficient operation. TSMC's 28nm manufacturing process plays an important role in lowering power consumption, but many GPU architecture modifications were required to further reduce power consumption while maintaining great performance.

Every hardware unit in Kepler was designed and scrubbed to provide outstanding performance per watt. The best example of great perf/watt is seen in the design of Kepler GK110's new Streaming Multiprocessor (SMX), which is similar in many respects to the SMX unit recently introduced in Kepler GK104, but includes substantially more double precision units for compute algorithms.

Streaming Multiprocessor (SMX) Architecture

Kepler GK110’s new SMX introduces several architectural innovations that make it not only the most powerful multiprocessor we’ve built, but also the most programmable and power-efficient.



SMX: 192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU), and 32 load/store units (LD/ST).

SMX Processing Core Architecture

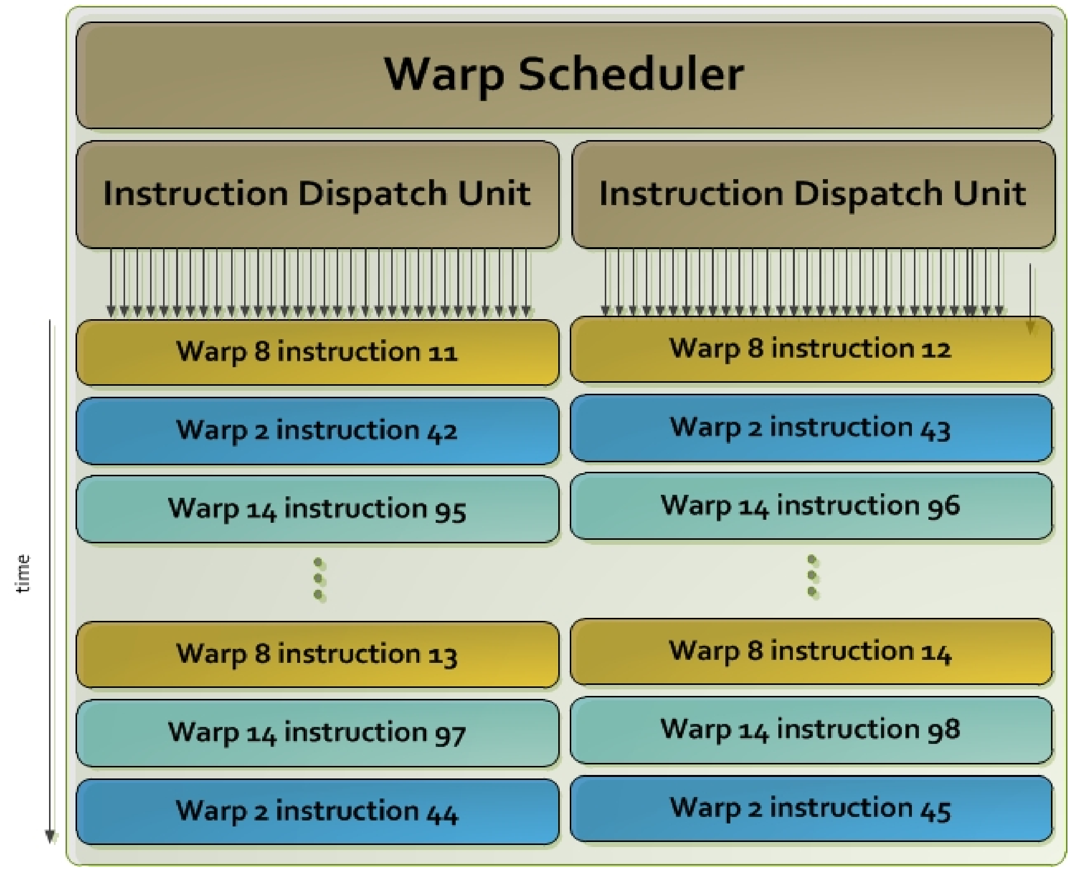
Each of the Kepler GK110 SMX units feature 192 single-precision CUDA cores, and each core has fully pipelined floating-point and integer arithmetic logic units. Kepler retains the full IEEE 754-2008 compliant single- and double-precision arithmetic introduced in Fermi, including the fused multiply-add (FMA) operation.

One of the design goals for the Kepler GK110 SMX was to significantly increase the GPU's delivered double precision performance, since double precision arithmetic is at the heart of many HPC applications. Kepler GK110's SMX also retains the special function units (SFUs) for fast approximate transcendental operations as in previous-generation GPUs, providing 8x the number of SFUs of the Fermi GF110 SM.

Similar to GK104 SMX units, the cores within the new GK110 SMX units use the primary GPU clock rather than the 2x shader clock. Recall the 2x shader clock was introduced in the G80 Tesla-architecture GPU and used in all subsequent Tesla- and Fermi-architecture GPUs. Running execution units at a higher clock rate allows a chip to achieve a given target throughput with fewer copies of the execution units, which is essentially an area optimization, but the clocking logic for the faster cores is more power-hungry. For Kepler, our priority was performance per watt. While we made many optimizations that benefitted both area and power, we chose to optimize for power even at the expense of some added area cost, with a larger number of processing cores running at the lower, less power-hungry GPU clock.

Quad Warp Scheduler

The SMX schedules threads in groups of 32 parallel threads called warps. Each SMX features four warp schedulers and eight instruction dispatch units, allowing four warps to be issued and executed concurrently. Kepler's quad warp scheduler selects four warps, and two independent instructions per warp can be dispatched each cycle. Unlike Fermi, which did not permit double precision instructions to be paired with other instructions, Kepler GK110 allows double precision instructions to be paired with other instructions.



Each Kepler SMX contains 4 Warp Schedulers, each with dual Instruction Dispatch Units. A single Warp Scheduler Unit is shown above.

We also looked for opportunities to optimize the power in the SMX warp scheduler logic. For example, both Kepler and Fermi schedulers contain similar hardware units to handle the scheduling function, including:

a) Register scoreboarding for long latency operations (texture and load)
b) Inter-warp scheduling decisions (e.g., pick the best warp to go next among eligible candidates)
c) Thread block level scheduling (e.g., the GigaThread engine)

However, Fermi's scheduler also contains a complex hardware stage to prevent data hazards in the math datapath itself. A multi-port register scoreboard keeps track of any registers that are not yet ready with valid data, and a dependency checker block analyzes register usage across a multitude of fully decoded warp instructions against the scoreboard, to determine which are eligible to issue.

For Kepler, we recognized that this information is deterministic (the math pipeline latencies are not variable), and therefore it is possible for the compiler to determine up front when instructions will be ready to issue, and provide this information in the instruction itself. This allowed us to replace several complex and power-expensive blocks with a simple hardware block that extracts the pre-determined latency information and uses it to mask out warps from eligibility at the inter-warp scheduler stage.

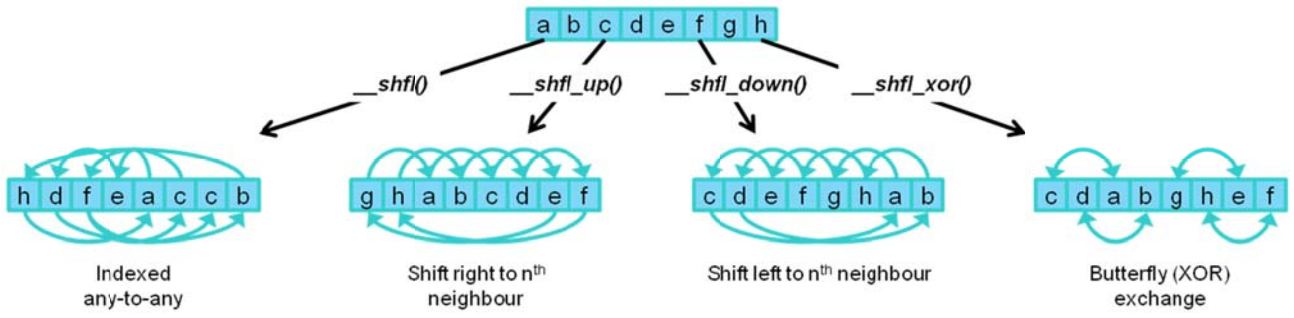
New ISA Encoding: 255 Registers per Thread

The number of registers that can be accessed by a thread has been quadrupled in GK110, allowing each thread access to up to 255 registers. Codes that exhibit high register pressure or spilling behavior in Fermi may see substantial speedups as a result of the increased available per-thread register count. A compelling example can be seen in the QUDA library for performing lattice QCD (quantum chromodynamics) calculations using CUDA. QUDA fp64-based algorithms see performance increases up to 5.3x due to the ability to use many more registers per thread and experiencing fewer spills to local memory.

Shuffle Instruction

To further improve performance, Kepler implements a new Shuffle instruction, which allows threads within a warp to share data. Previously, sharing data between threads within a warp required separate store and load operations to pass the data through shared memory. With the Shuffle instruction, threads within a warp can read values from other threads in the warp in just about any imaginable permutation. Shuffle supports arbitrary indexed references – i.e. any thread reads from any other thread. Useful shuffle subsets including next-thread (offset up or down by a fixed amount) and XOR "butterfly" style permutations among the threads in a warp, are also available as CUDA intrinsics.

Shuffle offers a performance advantage over shared memory, in that a store-and-load operation is carried out in a single step. Shuffle also can reduce the amount of shared memory needed per thread block, since data exchanged at the warp level never needs to be placed in shared memory. In the case of FFT, which requires data sharing within a warp, a 6% performance gain can be seen just by using Shuffle.



This example shows some of the variations possible using the new Shuffle instruction in Kepler.

Atomic Operations

Atomic memory operations are important in parallel programming, allowing concurrent threads to correctly perform read-modify-write operations on shared data structures. Atomic operations such as add, min, max, and compare-and-swap are atomic in the sense that the read, modify, and write operations are performed without interruption by other threads. Atomic memory operations are widely used for parallel sorting, reduction operations, and building data structures in parallel without locks that serialize thread execution.

Throughput of global memory atomic operations on Kepler GK110 is substantially improved compared to the Fermi generation. Atomic operation throughput to a common global memory address is improved by 9x to one operation per clock. Atomic operation throughput to independent global addresses is also significantly accelerated, and logic to handle address conflicts has been made more efficient. Atomic operations can often be processed at rates similar to global load operations. This speed increase makes atomics fast enough to use frequently within kernel inner loops, eliminating the separate reduction passes that were previously required by some algorithms to consolidate results. Kepler GK110 also expands the native support for 64-bit atomic operations in global memory. In addition to atomicAdd, atomicCAS, and atomicExch (which were also supported by Fermi and Kepler GK104), GK110 supports the following:

- atomicMin
- atomicMax
- atomicAnd
- atomicOr
- atomicXor

Other atomic operations which are not supported natively (for example 64-bit floating point atomics) may be emulated using the compare-and-swap (CAS) instruction.

Texture Improvements

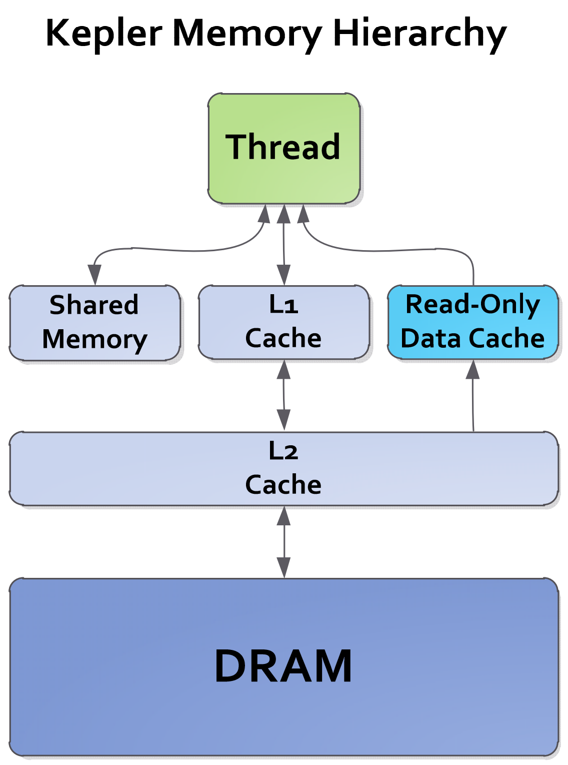
The GPU's dedicated hardware Texture units are a valuable resource for compute programs with a need to sample or filter image data. The texture throughput in Kepler is significantly increased compared to Fermi – each SMX unit contains 16 texture filtering units, a 4x increase vs the Fermi GF110 SM.

In addition, Kepler changes the way texture state is managed. In the Fermi generation, for the GPU to reference a texture, it had to be assigned a "slot" in a fixed-size binding table prior to grid launch. The number of slots in that table ultimately limits how many unique textures a program can read from at run time. Ultimately, a program was limited to accessing only 128 simultaneous textures in Fermi.

With bindless textures in Kepler, the additional step of using slots isn't necessary: texture state is now saved as an object in memory and the hardware fetches these state objects on demand, making binding tables obsolete. This effectively eliminates any limits on the number of unique textures that can be referenced by a compute program. Instead, programs can map textures at any time and pass texture handles around as they would any other pointer.

Kepler Memory Subsystem – L1, L2, ECC

Kepler's memory hierarchy is organized similarly to Fermi. The Kepler architecture supports a unified memory request path for loads and stores, with an L1 cache per SMX multiprocessor. Kepler GK110 also enables compiler-directed use of an additional new cache for read-only data, as described below.



64 KB Configurable Shared Memory and L1 Cache

In the Kepler GK110 architecture, as in the previous generation Fermi architecture, each SMX has 64 KB of on-chip memory that can be configured as 48 KB of Shared memory with 16 KB of L1 cache, or as 16 KB of shared memory with 48 KB of L1 cache. Kepler now allows for additional flexibility in configuring the allocation of shared memory and L1 cache by permitting a 32KB / 32KB split between shared memory and L1 cache. To support the increased throughput of each SMX unit, the shared memory bandwidth for 64b and larger load operations is also doubled compared to the Fermi SM, to 256B per core clock.

48KB Read-Only Data Cache

In addition to the L1 cache, Kepler introduces a 48KB cache for data that is known to be read-only for the duration of the function. In the Fermi generation, this cache was accessible only by the Texture unit. Expert programmers often found it advantageous to load data through this path explicitly by mapping their data as textures, but this approach had many limitations.

In Kepler, in addition to significantly increasing the capacity of this cache along with the texture horsepower increase, we decided to make the cache directly accessible to the SM for general load operations. Use of the read-only path is beneficial because it takes both load and working set footprint off of the Shared/L1 cache path. In addition, the Read-Only Data Cache's higher tag bandwidth supports full speed unaligned memory access patterns among other scenarios.

Use of the read-only path can be managed automatically by the compiler or explicitly by the programmer. Access to any variable or data structure that is known to be constant through programmer use of the C99-standard "const __restrict" keyword may be tagged by the compiler to be loaded through the Read-Only Data Cache. The programmer can also explicitly use this path with the __ldg() intrinsic.

Improved L2 Cache

The Kepler GK110 GPU features 1536KB of dedicated L2 cache memory, double the amount of L2 available in the Fermi architecture. The L2 cache is the primary point of data unification between the SMX units, servicing all load, store, and texture requests and providing efficient, high speed data sharing across the GPU. The L2 cache on Kepler offers up to 2x of the bandwidth per clock available in Fermi. Algorithms for which data addresses are not known beforehand, such as physics solvers, ray tracing, and sparse matrix multiplication especially benefit from the cache hierarchy. Filter and convolution kernels that require multiple SMs to read the same data also benefit.

Memory Protection Support

Like Fermi, Kepler's register files, shared memories, L1 cache, L2 cache and DRAM memory are protected by a Single-Error Correct Double-Error Detect (SECDED) ECC code. In addition, the Read-Only Data Cache supports single-error correction through a parity check; in the event of a parity error, the cache unit automatically invalidates the failed line, forcing a read of the correct data from L2.

ECC checkbit fetches from DRAM necessarily consume some amount of DRAM bandwidth, which results in a performance difference between ECC-enabled and ECC-disabled operation, especially on memory bandwidth-sensitive applications. Kepler GK110 implements several optimizations to ECC checkbit fetch handling based on Fermi experience. As a result, the ECC on-vs-off performance delta has been reduced by an average of 66%, as measured across our internal compute application test suite.

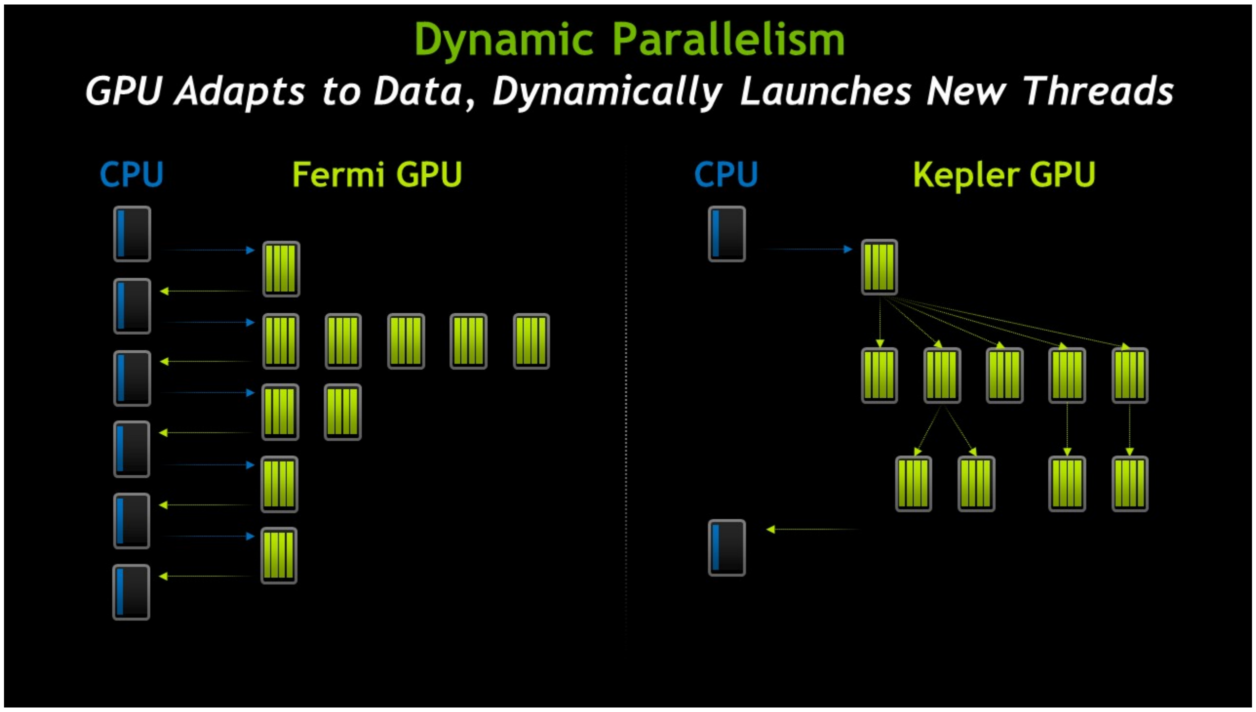
Dynamic Parallelism

In a hybrid CPU-GPU system, enabling a larger amount of parallel code in an application to run efficiently and entirely within the GPU improves scalability and performance as GPUs increase in perf/watt. To accelerate these additional parallel portions of the application, GPUs must support more varied types of parallel workloads.

Dynamic Parallelism is a new feature introduced with Kepler GK110 that allows the GPU to generate new work for itself, synchronize on results, and control the scheduling of that work via dedicated, accelerated hardware paths, all without involving the CPU.

Fermi was very good at processing large parallel data structures when the scale and parameters of the problem were known at kernel launch time. All work was launched from the host CPU, would run to completion, and return a result back to the CPU. The result would then be used as part of the final solution, or would be analyzed by the CPU which would then send additional requests back to the GPU for additional processing.

In Kepler GK110 any kernel can launch another kernel, and can create the necessary streams, events and manage the dependencies needed to process additional work without the need for host CPU interaction. This architectural innovation makes it easier for developers to create and optimize recursive and data-dependent execution patterns, and allows more of a program to be run directly on GPU. The system CPU can then be freed up for additional tasks, or the system could be configured with a less powerful CPU to carry out the same workload.



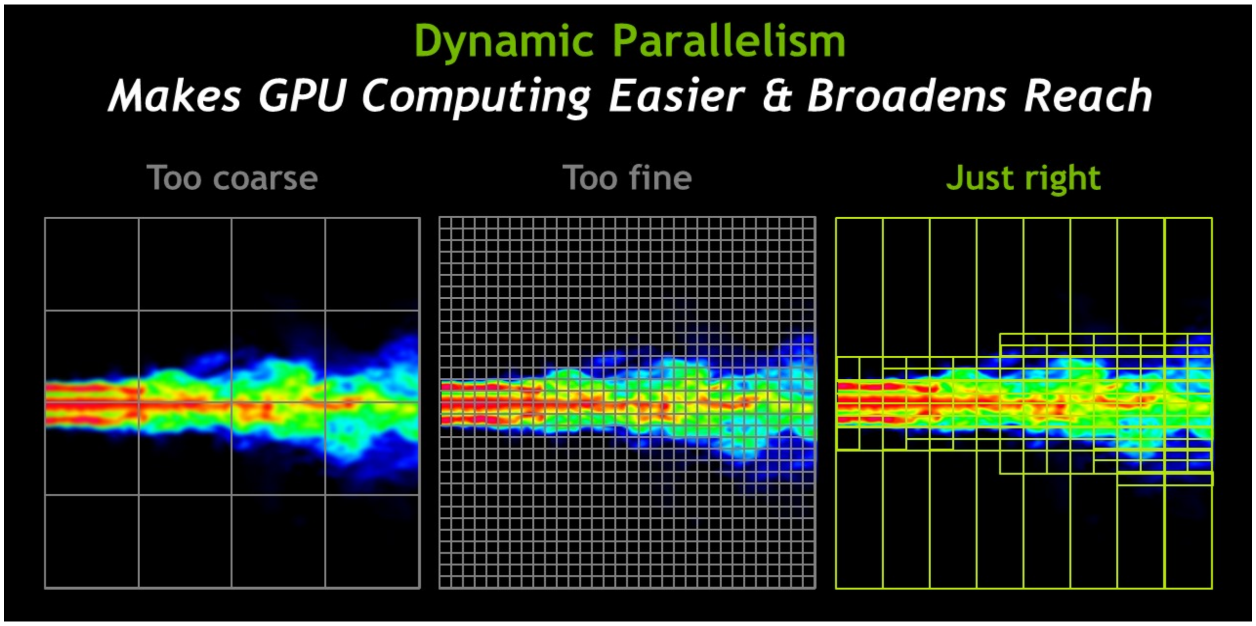
Dynamic Parallelism allows more parallel code in an application to be launched directly by the GPU onto itself (right side of image) rather than requiring CPU intervention (left side of image).

Dynamic Parallelism allows more varieties of parallel algorithms to be implemented on the GPU, including nested loops with differing amounts of parallelism, parallel teams of serial control-task threads, or simple serial control code offloaded to the GPU in order to promote data-locality with the parallel portion of the application.

Because a kernel has the ability to launch additional workloads based on intermediate, on-GPU results, programmers can now intelligently load-balance work to focus the bulk of their resources on the areas of the problem that either require the most processing power or are most relevant to the solution.

One example would be dynamically setting up a grid for a numerical simulation – typically grid cells are focused in regions of greatest change, requiring an expensive pre-processing pass through the data. Alternatively, a uniformly coarse grid could be used to prevent wasted GPU resources, or a uniformly fine grid could be used to ensure all the features are captured, but these options risk missing simulation features or "over-spending" compute resources on regions of less interest.

With Dynamic Parallelism, the grid resolution can be determined dynamically at runtime in a data-dependent manner. Starting with a coarse grid, the simulation can "zoom in" on areas of interest while avoiding unnecessary calculation in areas with little change. Though this could be accomplished using a sequence of CPU-launched kernels, it would be far simpler to allow the GPU to refine the grid itself by analyzing the data and launching additional work as part of a single simulation kernel, eliminating interruption of the CPU and data transfers between the CPU and GPU.



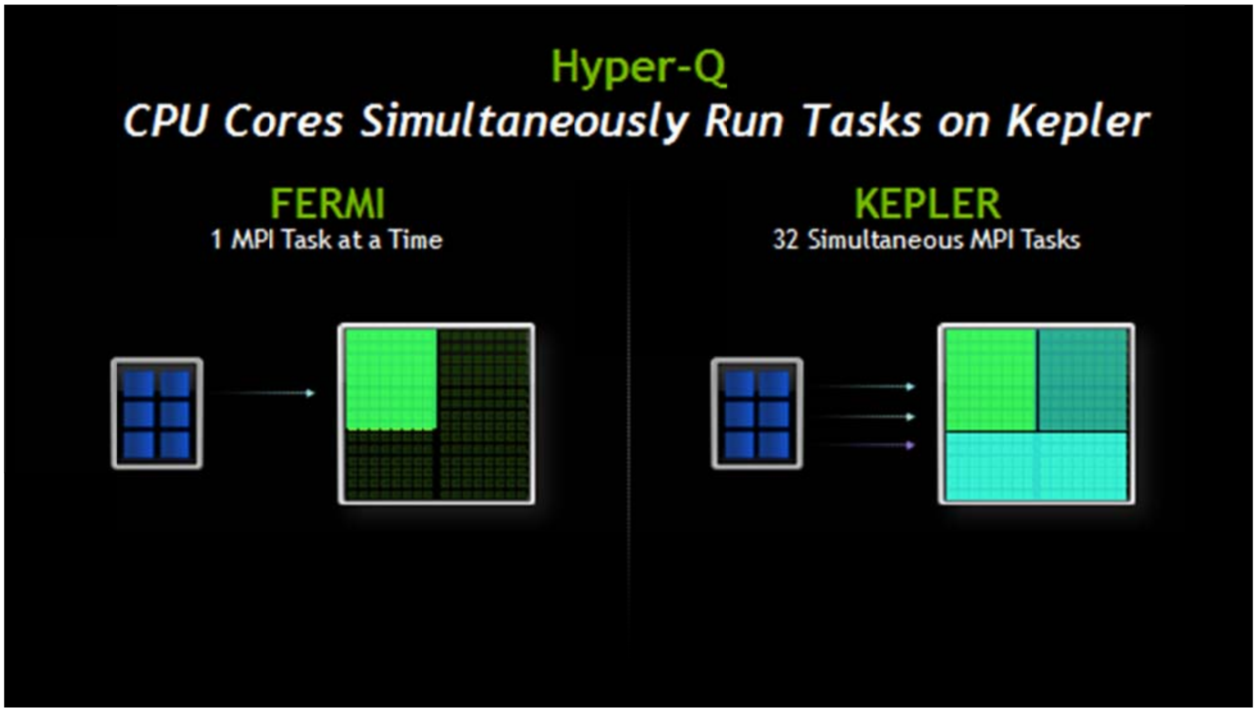
–Image attribution Charles Reid

The above example illustrates the benefits of using a dynamically sized grid in a numerical simulation. To meet peak precision requirements, a fixed resolution simulation must run at an excessively fine resolution across the entire simulation domain, whereas a multi-resolution grid applies the correct simulation resolution to each area based on local variation.

Hyper-Q

One of the challenges in the past has been keeping the GPU supplied with an optimally scheduled load of work from multiple streams. The Fermi architecture supported 16-way concurrency of kernel launches from separate streams, but ultimately the streams were all multiplexed into the same hardware work queue. This allowed for false intra-stream dependencies, requiring dependent kernels within one stream to complete before additional kernels in a separate stream could be executed. While this could be alleviated to some extent through the use of a breadth-first launch order, as program complexity increases, this can become more and more difficult to manage efficiently.

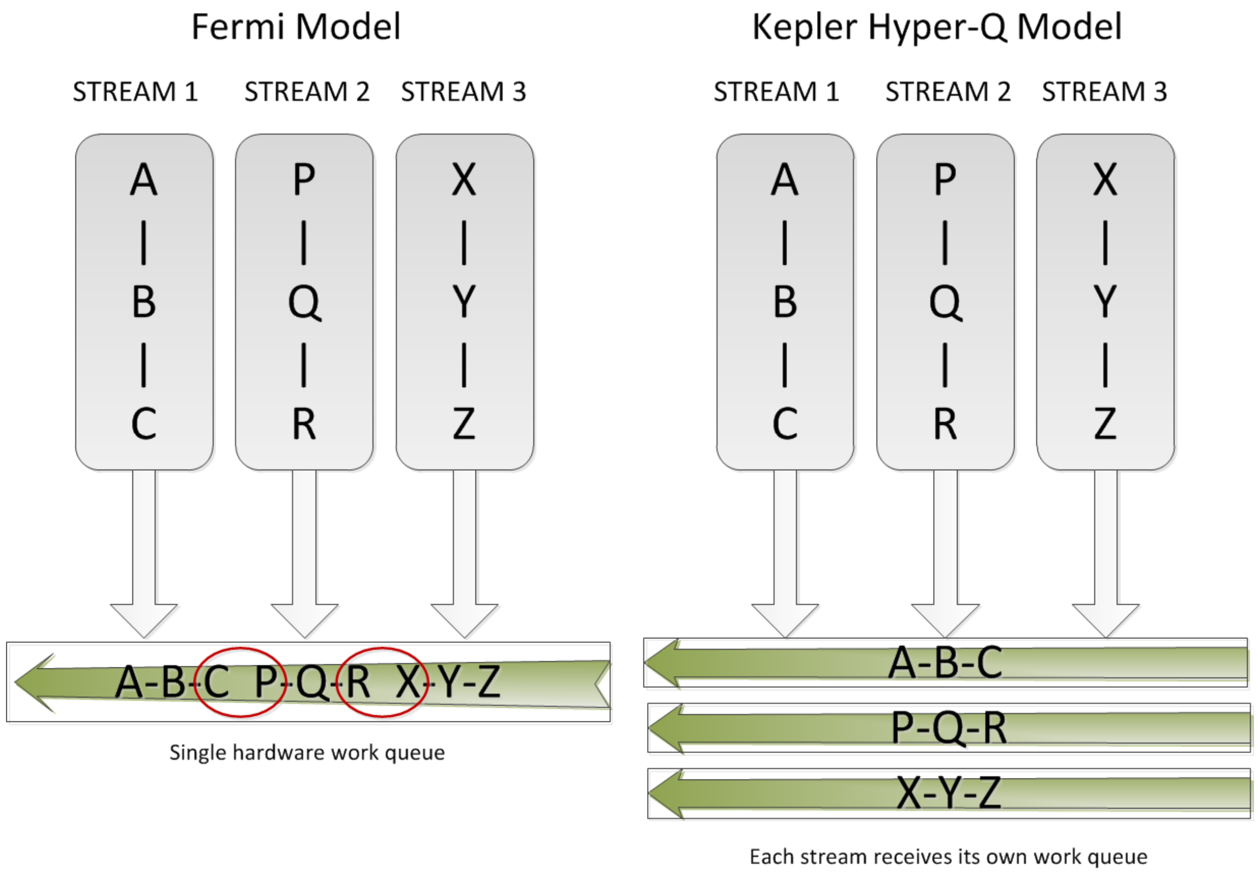
Kepler GK110 improves on this functionality with the new Hyper-Q feature. Hyper-Q increases the total number of connections (work queues) between the host and the CUDA Work Distributor (CWD) logic in the GPU by allowing 32 simultaneous, hardware-managed connections (compared to the single connection available with Fermi). Hyper-Q is a flexible solution that allows connections from multiple CUDA streams, from multiple Message Passing Interface (MPI) processes, or even from multiple threads within a process. Applications that previously encountered false serialization across tasks, thereby limiting GPU utilization, can see up to a 32x performance increase without changing any existing code.


Hyper-Q
CPU Cores Simultaneously Run Tasks on Kepler
FERMI
1 MPI Task at a Time
KEPLER
32 Simultaneous MPI Tasks

Hyper-Q permits more simultaneous connections between CPU and GPU.

Each CUDA stream is managed within its own hardware work queue, inter-stream dependencies are optimized, and operations in one stream will no longer block other streams, enabling streams to execute concurrently without needing to specifically tailor the launch order to eliminate possible false dependencies.

Hyper-Q offers significant benefits for use in MPI-based parallel computer systems. Legacy MPI-based algorithms were often created to run on multi-core CPU systems, with the amount of work assigned to each MPI process scaled accordingly. This can lead to a single MPI process having insufficient work to fully occupy the GPU. While it has always been possible for multiple MPI processes to share a GPU, these processes could become bottlenecked by false dependencies. Hyper-Q removes those false dependencies, dramatically increasing the efficiency of GPU sharing across MPI processes.



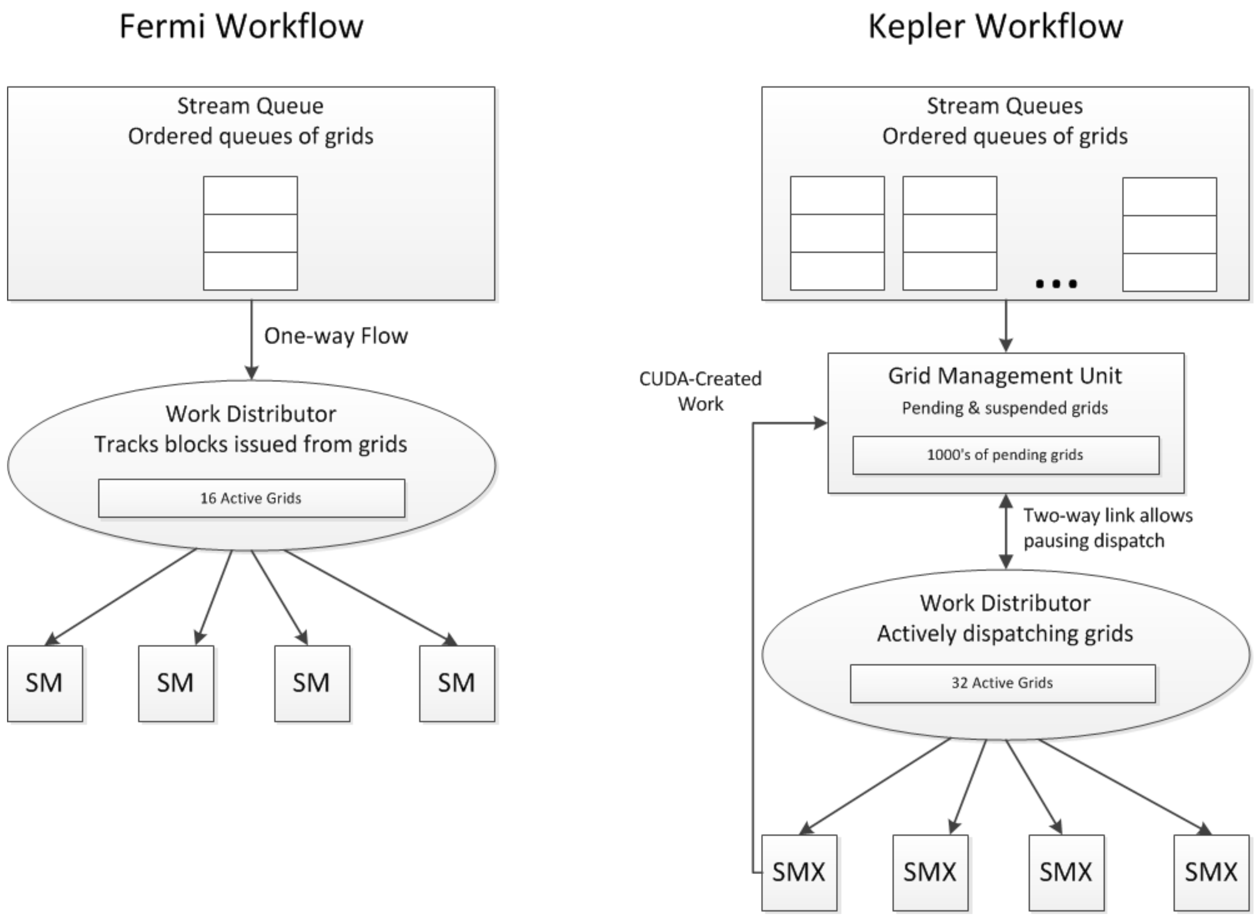
Hyper-Q working with CUDA Streams: In the Fermi model shown on the left, only (C,P) & (R,X) can run concurrently due to intra-stream dependencies caused by the single hardware work queue. The Kepler Hyper-Q model allows all streams to run concurrently using separate work queues.

Grid Management Unit - Efficiently Keeping the GPU Utilized

New features in Kepler GK110, such as the ability for CUDA kernels to launch work directly on the GPU with Dynamic Parallelism, required that the CPU-to-GPU workflow in Kepler offer increased functionality over the Fermi design. On Fermi, a grid of thread blocks would be launched by the CPU and would always run to completion, creating a simple unidirectional flow of work from the host to the SMs via the CUDA Work Distributor (CWD) unit. Kepler GK110 was designed to improve the CPU-to-GPU workflow by allowing the GPU to efficiently manage both CPU- and CUDA-created workloads.

We discussed the ability of the Kepler GK110 GPU to allow kernels to launch work directly on the GPU, and it's important to understand the changes made in the Kepler GK110 architecture to facilitate these new functions. In Kepler, a grid can be launched from the CPU just as was the case with Fermi, however new grids can also be created programmatically by CUDA within the Kepler SMX unit. To manage both CUDA-created and host-originated grids, a new Grid Management Unit (GMU) was introduced in Kepler GK110. This control unit manages and prioritizes grids that are passed into the CWD to be sent to the SMX units for execution.

The CWD in Kepler holds grids that are ready to dispatch, and it is able to dispatch 32 active grids, which is double the capacity of the Fermi CWD. The Kepler CWD communicates with the GMU via a bi-directional link that allows the GMU to pause the dispatch of new grids and to hold pending and suspended grids until needed. The GMU also has a direct connection to the Kepler SMX units to permit grids that launch additional work on the GPU via Dynamic Parallelism to send the new work back to GMU to be prioritized and dispatched. If the kernel that dispatched the additional workload pauses, the GMU will hold it inactive until the dependent work has completed.



The redesigned Kepler HOST to GPU workflow shows the new Grid Management Unit, which allows it to manage the actively dispatching grids, pause dispatch, and hold pending and suspended grids.

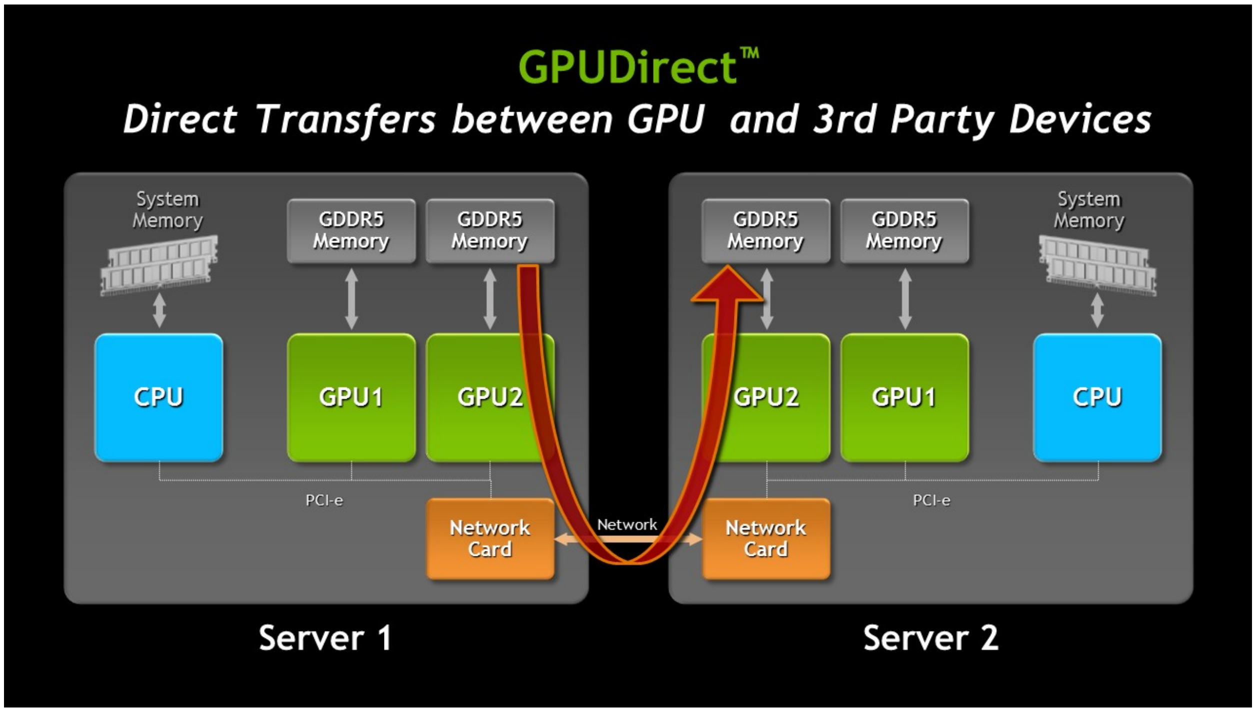
NVIDIA GPUDirect™

When working with a large amount of data, increasing the data throughput and reducing latency is vital to increasing compute performance. Kepler GK110 supports the RDMA feature in NVIDIA GPUDirect, which is designed to improve performance by allowing direct access to GPU memory by third-party devices such as IB adapters, NICs, and SSDs. When using CUDA 5.0, GPUDirect provides the following important features:

- Direct memory access (DMA) between NIC and GPU without the need for CPU-side data buffering.
- Significantly improved MPISend/MPIRecv efficiency between GPU and other nodes in a network.
- Eliminates CPU bandwidth and latency bottlenecks
- Works with variety of 3rd-party network, capture, and storage devices

Applications like reverse time migration (used in seismic imaging for oil & gas exploration) distribute the large imaging data across several GPUs. Hundreds of GPUs must collaborate to crunch the data, often communicating intermediate results. GPUDirect enables much higher aggregate bandwidth for this GPU-to-GPU communication scenario within a server and across servers with the P2P and RDMA features.

Kepler GK110 also supports other GPUDirect features such as Peer-to-Peer and GPUDirect for Video.



GPUDirect RDMA allows direct access to GPU memory from 3rd-party devices such as network adapters, which translates into direct transfers between GPUs *across* nodes as well.

Conclusion

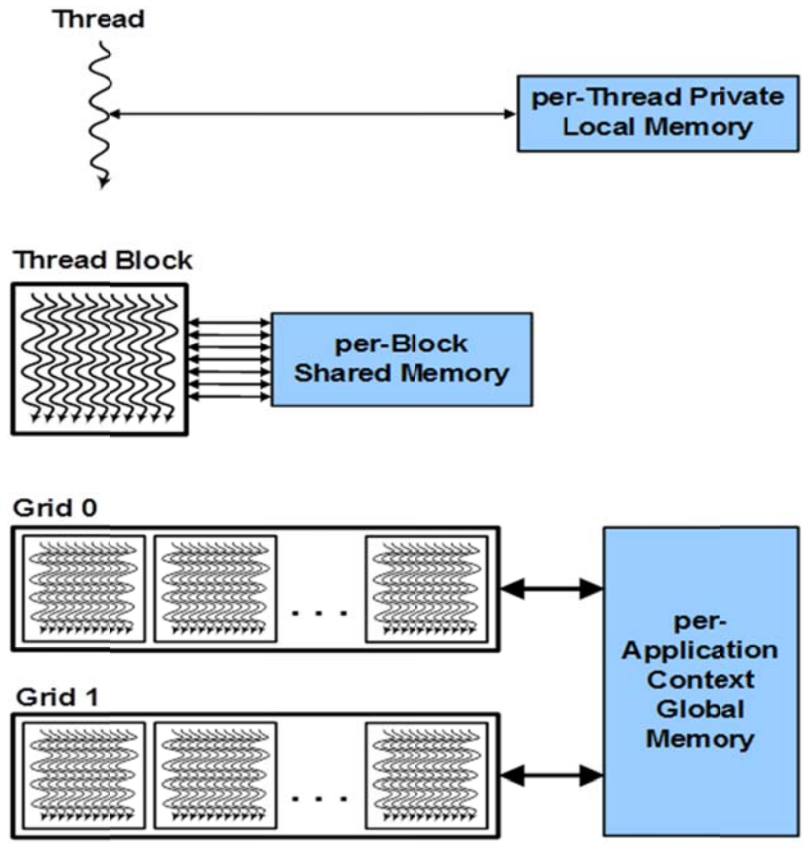
With the launch of Fermi in 2010, NVIDIA ushered in a new era in the high performance computing (HPC) industry based on a hybrid computing model where CPUs and GPUs work together to solve computationally-intensive workloads. Now, with the new Kepler GK110 GPU, NVIDIA again raises the bar for the HPC industry.

Kepler GK110 was designed from the ground up to maximize computational performance and throughput computing with outstanding power efficiency. The architecture has many new innovations such as SMX, Dynamic Parallelism, and Hyper-Q that make hybrid computing dramatically faster, easier to program, and applicable to a broader set of applications. Kepler GK110 GPUs will be used in numerous systems ranging from workstations to supercomputers to address the most daunting challenges in HPC.

Appendix A - Quick Refresher on CUDA

CUDA is a combination hardware/software platform that enables NVIDIA GPUs to execute programs written with C, C++, Fortran, and other languages. A CUDA program invokes parallel functions called kernels that execute across many parallel threads. The programmer or compiler organizes these threads into thread blocks and grids of thread blocks, as shown in Figure 1. Each thread within a thread block executes an instance of the kernel. Each thread also has thread and block IDs within its thread block and grid, a program counter, registers, per-thread private memory, inputs, and output results.

A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. A thread block has a block ID within its grid. A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls. In the CUDA parallel programming model, each thread has a per-thread private memory space used for register spills, function calls, and C automatic array variables. Each thread block has a per-block shared memory space used for inter-thread communication, data sharing, and result sharing in parallel algorithms. Grids of thread blocks share results in Global Memory space after kernel-wide global synchronization.



Thread
per-Thread Private Local Memory
Thread Block
per-Block Shared Memory
Grid 0
Grid 1
per-Application Context Global Memory

Figure 1: CUDA Hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces.

CUDA Hardware Execution

CUDA's hierarchy of threads maps to a hierarchy of processors on the GPU; a GPU executes one or more kernel grids; a streaming multiprocessor (SM on Fermi / SMX on Kepler) executes one or more thread blocks; and CUDA cores and other execution units in the SMX execute thread instructions. The SMX executes threads in groups of 32 threads called warps. While programmers can generally ignore warp execution for functional correctness and focus on programming individual scalar threads, they can greatly improve performance by having threads in a warp execute the same code path and access memory with nearby addresses.