**Parallelism in NVidia Kepler Architecture**

**ECE 332 Computer Architecture**

**Xiwen Li, Peicheng Tang**

**May 11, 2017**

Kepler architecture is a microarchitecture employed in GTX 600 series, 700 series and 800 series. Developed by NVidia, Kepler architecture is used to replace Fermi microarchitecture. Fermi emphasizes the performance of GPU but ignores the development and the need of power efficiency. Under such circumstance, NVidia architects introduce its successor, Kepler architecture, and aim to make Kepler not only the fastest but also most efficient architecture at that time [7]. As the first microarchitecture to focus on energy efficiency, Kepler fully takes advantage of the idea of parallelism to boost the efficiency using their unique techniques and improved memories structure.

**New Techniques**

To make Kepler the most powerful and efficient architecture ever, Kepler architects come up with the following three new techniques, which are SMX architecture, Dynamic Parallelism and Hyper-Q.

* **SMX Architecture**

SMX architecture is one of three techniques employed by Kepler architecture. It inherits some features from Fermi architecture, an architecture adapted by GTX 400 series and 500 series. SMX architecture stands for streaming multiprocessor architecture. It can make processor powerful, programmable and power-efficient. The trick hidden behind is the combination of parallelism and data share. Some of its features can reflect this idea. This section will discuss the following five features of SMX architecture, which improve processor’s performance:

1. SMX Processing Core Architecture
2. Quad Warp Scheduler
3. Shuffle instruction
4. Atomic Operation
5. Texture Improvement
6. **SMX Processing Core Architecture**

SMX processing core architecture can reflect the idea of power-saving and high programmability. The basic components of SMX consist of 32 load/store units, 32 special function units, 64 double-precision units and 192 single-precision CUDA cores. Load/store units fetch and save data to memory, while special function units handle transcendental and graphics interpolation instructions. CUDA cores, in other words, parallel processors, perform texture filtering [2].

NVidia creates CUDA, a parallel computing platform and application programming interface model to let programmers directly get access to GPU’s instruction set. CUDA can work with C, C++ and FORTRAN, and allows programmers to do parallel programming using GPI resources. Each CUDA core consists of fully pipelined floating-point and integer arithmetic logic units. In addition, Kepler architecture adopts primary GPU clock, which is slower than Fermi’s 2x shader clock. Since the slower clock rate can consume less power, Kepler’s designers decide to sacrifice a little bit performance to achieve high power efficiency.

1. **Quad Warp Scheduler**

Quad Warp Scheduler enables GPU to fetch instructions in groups. It can schedule groups of 32 parallel threads called warps. There are four warp schedulers for each SMX, while each warp scheduler has two instruction dispatch units as shown in Figure 1. When the processor is about to encode, each instruction dispatch unit will select one instruction and send that to register file in each cycle.

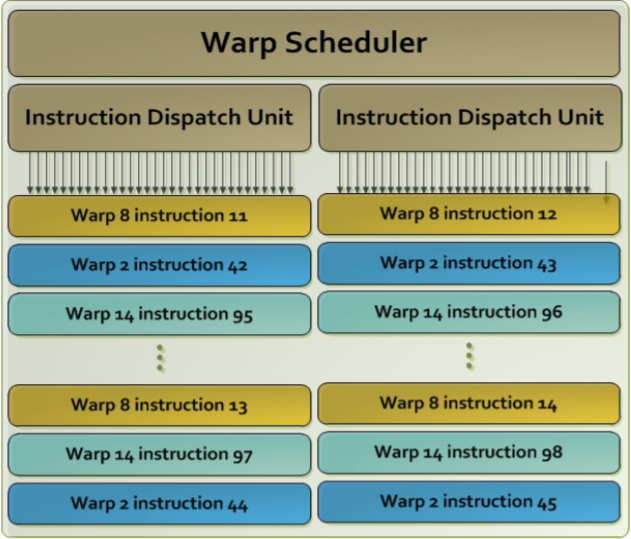


Figure 1: Each warp scheduler has two dispatch units [2]

As mentioned above, the SMX can schedule 32 warps. For each thread, it can access at most 225 registers [2]. Due to such large number of accessible registers, processor can process those instructions passed in at the same time. As a result, SMX processor can run faster than other architectures.

1. **Shuffle Instruction**

Since instructions can be executed in parallel, SMX can also share data between different threads. Inside one warp, a warp can read data from any other thread. In this way, each warp can obtain data from other warps without accessing memory. There are four ways to shuffle the data, including “indexed any-to-any”, shuffle right, shuffle left and “XOR”. “Indexed any-to-any” means each thread can read from any other thread. Shuffle right and shuffle left mean shifting all the data to nth neighbors on the right or left, while “XOR” allows data interchanges with each other from different threads as Figure 2 shows [2]. Since getting data from memory needs to use load/store word, data shuffling improves the performance in the way that the processor does 006Eot waste time on frequent memory access.

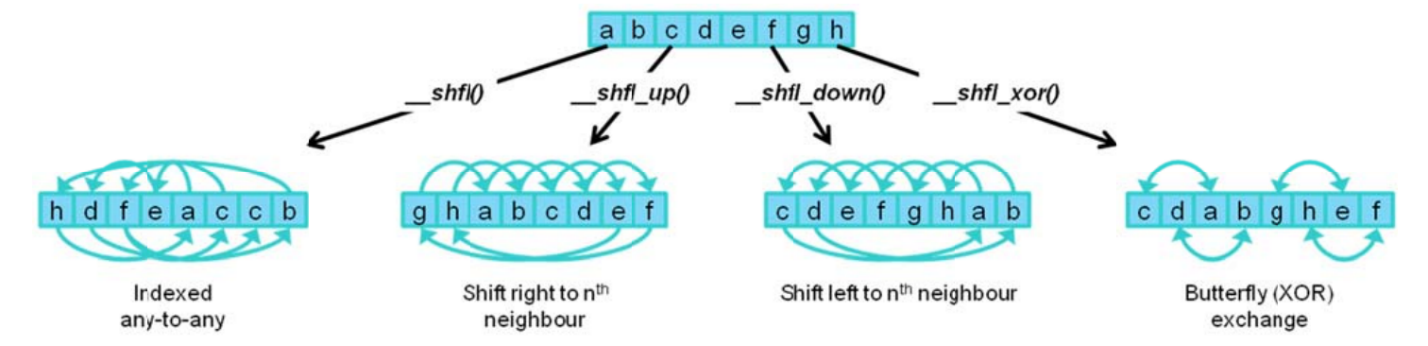


Figure 2: Four shuffle ways [2]

1. **Atomic Operations**

SMX architecture also has a special operation called atomic operation that helps parallel programming. SMX processor is a concurrent system where different processes can access a shared data structure at the same time [2]. If a process is accessing a piece of data while another process is modifying that data, it may cause problems. Atomic operations, acting like a “lock” cope with the problem by only allowing one operation at a time. When multiple threads want to access the same data, atomic operation keeps a certain sequence so that they will not interrupt with each other. In this way, this technique allows simultaneous threads read, modify or write operations on shared data structures.

1. **Texture Improvements**

The last feature is texture unit improvements. Texture unit can sample and filter image data. SMX architecture processor has 16 texture filtering units, which are more than other processors. In this way, Kepler has more texture throughput than other architecture. Moreover, Kepler improves the way to managing texture state [2]. It can give each program unlimited texture states to access. Those states are packed up as an object in the memory. A hardware can fetch the object when it needs.

* **Dynamic Parallelism**

In simple words, Dynamic Parallelism allows GPU to generate work by itself without going through the CPU, which enables GPU to run complex tasks on its own, and frees CPU to focus on other tasks.

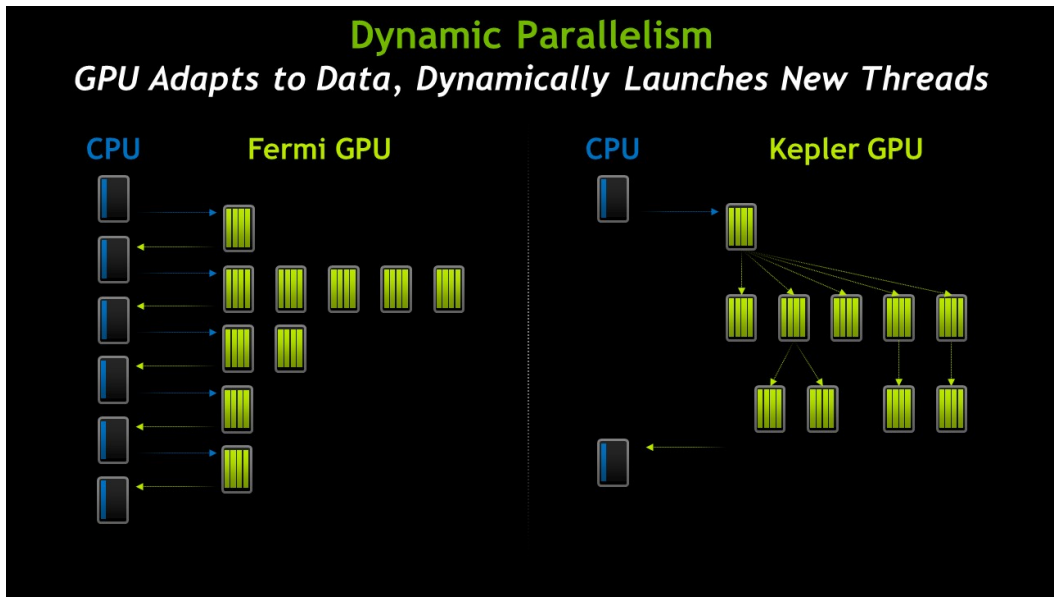
In Fermi architecture, the previous architecture NVDIA used in their GPU, programs can only run as a sequence of kernel launches. In other words, even though CPU can ask GPU to run multiple tasks in parallel in a single kernel, GPU must return the results and ask for new quests from CPU after finishing each kernel launch, as shown in the left side of Figure 3 [2]. In this case, it takes more power and time to finish all the tasks due to the intervention of CPU. However, as we can see from the right side of Figure 3, GPU now launches kernels and schedules the works on its own. Such technique only involves CPU twice during the entire tasks: one is to get the request; the other is to return the result.

Figure 3: Comparison between two architectures with and without Dynamic Parallelism [2]

Figure 3 also illustrates how Dynamic Parallelism works. On the right part of the figure, the top block is called a parent grid, where a grid is “a group of blocks of threads that are running a kernel”, and the subsequent five grids are child grids. As shown in the graph, each child grid creates their own child grids, while all the child grids in the same level can run simultaneously [4]. After all the operations finish, the result goes back to CPU.

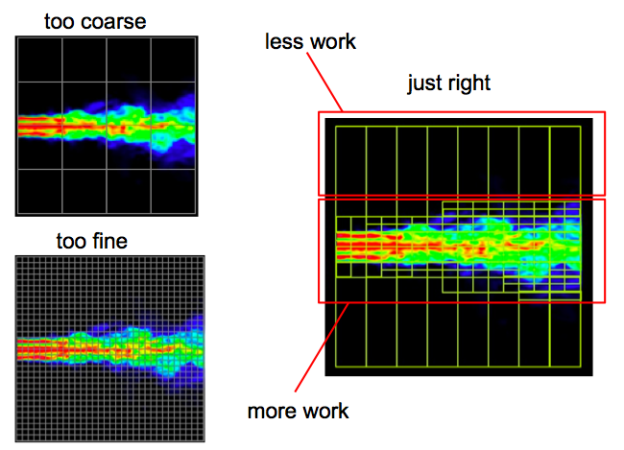
The benefits of Dynamic Parallelism in the application are significant. One example is the numerical simulation, as shown in the figure below.

Figure 4: A fluid simulation that uses adaptive mesh refinement performs work only where needed. [5]

Normally, to be precise, the simulation must have certain resolution. A fixed resolution simulation, in other words, without Dynamic Parallelism will have excessively fine resolution grid, but there is no need to refine the “black” part. Thus, the waste of power and time becomes noticeable. Of course, if we do not have high resolution, the part we care will be too coarse to analyze. On the other hand, with Dynamic Parallelism, each grid can decide whether to obtain a fine resolution. In this way, the system does not waste power on the unimportant part, but concentrates on the essential part.

* **Hyper-Q**

The idea of Hyper-Q is straightforward. What can we do if it is inefficient to do only one task at a time using GPU while the rest of GPU is idle? Yes, do more tasks simultaneously in GPU. This is the vital idea of Hyper-Q.

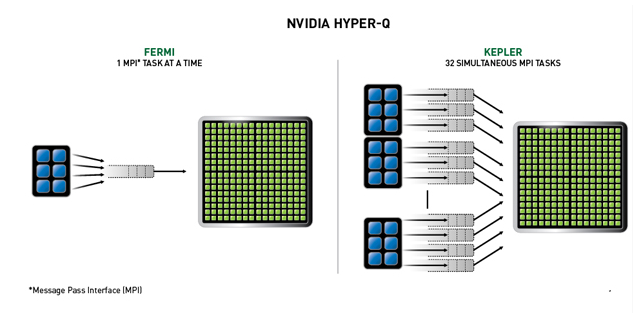


Figure 5: Comparison between two architectures with and without Hyper-Q [6]

As Figure 5 shows, in the old architecture, we can launch only one task at a time from CPU to GPU. It is inefficient and wasteful in the way that GPU only utilizes a small portion to implement the task while the rest of GPU is left without work to do. On the other hand, with Hyper-Q, multiple CPUs can launch work (32 tasks) on a single GPU, and GPU can process all the work at the same time. In this way, GPU uses its maximum capability and thus the technique pushes the GPU performance to its peak value [6].

As a result, without Hyper-Q, all the tasks form a single hardware work queue. These tasks are pipelined in the way that only the end of one task and the beginning of the next one can be processed concurrently. In contrast, with Hyper-Q, multiple independent tasks can be performed at the same time, and hence it improves the efficiency and performance of this architecture.

We can see the improvements visually with the example of CP2K, a widely used program that performs atomistic and molecular simulations of various systems.

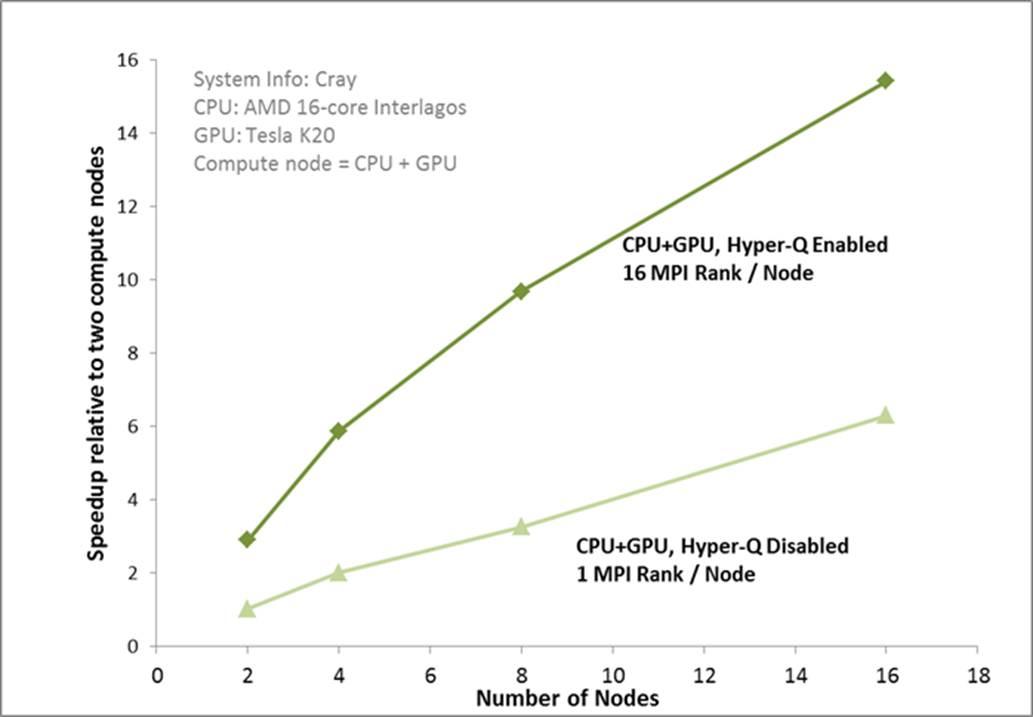


Figure 6: Comparison between performances with and without Hyper-Q [6]

Figure 6 is obtained by simulating data set of 864 water molecules. The data points represent how much it improves when using both CPU and GPU with respect to using CPU only. The performance is clearly not much better with only one task for GPU, while the performance is significantly better off with sixteen tasks launching at the same time, which in this case, fully utilize the GPU’s capability. It turns out that with Hyper-Q, we can achieve about 2.5x speedup [6].

**Memory**

There are not many unique techniques created for Kepler’s memory system. Like other memory structure, Kepler supports a unified memory with L1 cache, L2 cache and DRAM memory, while it also has a shared memory and a read-only cache set, as shown in Figure 7.

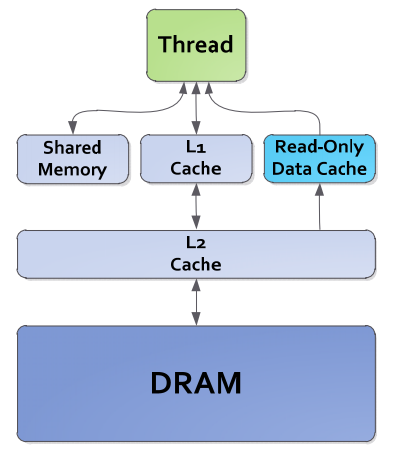


Figure 7: Kepler Memory Hierarchy [2]

Each SMX multiprocessor will have a L1 Cache. Since each SMX has 64 KB on-chip memory, the size of L1 Cache can be different considering the size of Shared Memory. The common practice is 16 KB for L1 and 48 KB for Shared Memory, or the reversed case. In addition to L1 cache and Shared Memory, Read-only cache is also directly accessible by SMX for general load operation [2].

All the memory units are protected by a Single‐Error Correct Double-Error Detect (SECDED) ECC code. As the name of the rule says, if the code differs by 1 bit, the system can correct the error; while if the code differs by 2 bits, the system can detect and report the error, but may not be able to correct it [2].

After all, it is the L2 cache that improves the performance. The size of L2 cache for Kepler architecture is 1536 KB, double the amount for the previous architecture. It also has twice the bandwidth per clock [2]. In this case, for codes that we do not know the data address beforehand, L2 cache can increase the performance. In addition, algorithms that needs same data also benefit from this.

**Kepler Architecture on GTX 680**

This section will focus on NVidia GeForce GTX 680, a typical Kepler architecture adaption example, including techniques like Graphic Processing Clusters, next generation SMX architecture, PolyMorph Engine, L2 cache, bindless textures.

GTX 680 consists of four Graphics Processing Clusters (GPC), eight Stream Multiprocessors and four memory controllers. Kepler architecture always contains high level GPC, and GTX 680 keeps this tradition. These four GPCs can deliver 32 pixels per clock.

SMX is very important to GTX 680. The success of GTX 680 depends on its next generation SMX. GTX 680 scheduling unit concentrates on power efficiency. For Kepler, since the math pipeline latencies are not variable, it is impossible for the compiler to determine up front when instructions is ready to issue, and to provide the information in the instruction itself. In this way, several complex and power-expensive blocks with hardware blocks that extract the predetermined latency information can be reduced [3].

GTX 680 also has GPU boost. It can automatically and dynamically optimize clock speed according to how power hungry the application is. By reading GPU temperature and hardware utilization, it will raise the clock and voltage [7].

In GTX 680, PolyMorph has been improved significantly. The PolyMorph is designed to ensure that even as tessellation is increased to very high expansion factors, the impact on rendering performance is minimized. GTX 680 has less PolyMorph Engines than previous generation GPU. However, each Kepler PolyMorph can deliver double performance per clock.

Here is the performance comparison between GTX 680 and GTX580. From Figure 8, it is obvious that GTX 680 has better performance than other GPUs [3].

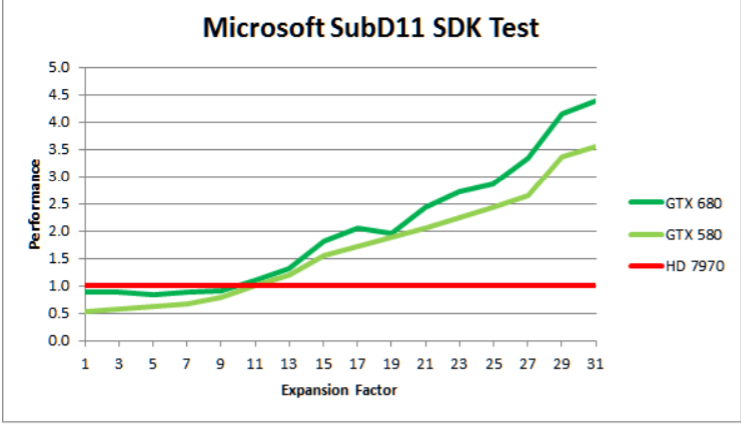


Figure 8: Performance Comparison between GTX 680 and GTX 580 [3]

Just like normal processor, GTX 680 has a unified 512KB L2 cache that provides buffer storing frequently used data and is shared across the GPU. Its hit rate has been improved by 73%, and the number of atomic operations has also been increased.

**Conclusion**

In conclusion, with the three unique techniques: SMX architecture, Dynamic Parallelism, and Hyper-Q, Kepler architecture becomes one of the most efficient and powerful microarchitecture at that time. The practice use of this architecture in GTX series has also proved that these innovative ideas are successful. Nowadays, more powerful and efficient architectures such as Maxwell and Pascal have been developed. It is believed that there is still a long way to go to keep improving those architectures.

**Work Cited**

[1] Ryan Smith. (2012, March 22). *NVIDIA GeForce GTX 680 Review: Retaking the Performance Crown* [Online]. Available: http://www.anandtech.com/show/5699/nvidia-geforce-gtx-680-review/2

[2] NVIDIA. (2012). *NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110* [Online]. Available FTP: www.nvidia.com/ Directory: content/PDF/kepler/. File: NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[3] NVIDIA (2012). *GTX 680 Whitepaper* [Online]. Available: http://la.nvidia.com/content/PDF/product-specifications/GeForce\_GTX\_680\_Whitepaper\_FINAL.pdf

[4] Andrew Adinetz. (2014, May 20). *CUDA Dynamic Parallelism API and Principles* [Online]. Available: https://devblogs.nvidia.com/parallelforall/cuda-dynamic-parallelism-api-principles/

[5] Andrew Adinetz. (2014, May 6). *Adaptive Parallel Computation with CUDA Dynamic Parallelism* [Online]. Available: https://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/

[6] Peter Mseemer. (2012, August 23). *UNLEASH LEGACY MPI CODES WITH KEPLER’S HYPER-Q* [Online]. Available: <https://blogs.nvidia.com/blog/2012/08/23/unleash-legacy-mpi-codes-with-keplers-hyper-q/>

[7] James Wang. (2012, March 22). *Introducing the GeForce GTX 680 GPU* [Online]. Available: http://www.geforce.com/whats-new/articles/introducing-the-geforce-gtx-680-gpu