Migrating your code from Tesla Fermi to Tesla K20X, with examples from the QUDA Lattice QCD library

Microway, Inc.
Justin Foley, PhD

October 26, 2013

Introduction

Over the last seven years, general-purpose GPU computing has evolved from being something of a curiosity into an extremely popular and immensely powerful HPC platform. NVIDIA has been a driving force in this process through the development of GPU-based hardware for general computation and the parallel development of the CUDA programming model.

Tesla-architecture GPUs (compute capability 1.x) were the first NVIDIA processors to support general-purpose computing using CUDA. This first generation was superseded by the Fermi generation of GPUs (compute capability 2.x), which debuted in 2010. Notable differences between Fermi hardware and the previous generation of processors include much higher double-precision performance on Fermi (support for double precision was introduced in compute 1.3 devices), the addition of a configurable L1 cache on each SM and an L2 cache shared between SMs, and support for concurrent kernel execution on Fermi. The rollout of Kepler in 2012 marked another significant milestone in the evolution of GPGPU computing.

In this document, we discuss a number of new features introduced in Kepler, and highlight differences in performance and functionality between Kepler and Fermi. We focus on the Kepler GK110 architecture (compute capability 3.5) [1], which is designed for high-performance double-precision calculations. At the time of writing, Kepler GK110 represents the state of the art in GPGPU hardware, and it forms the basis for NVIDIA's Tesla K20 and K20X accelerators. Topics covered include

- The impact of changes to the number of registers
- Hyper-Q
- Dynamic Parallelism
- Bindless Textures
- Shared-Memory Bandwidth

Some of these topics, such as bindless textures and shared-memory bandwidth, are also relevant for the Kepler GK104 architecture (compute capability 3.0). However, two of the most significant new Kepler features - Hyper-Q and Dynamic Parallelism - are only supported on GK110.

A number of examples in this report are taken from the QUDA library [2], which is an open-source library for performing Lattice QCD[1] calculations on NVIDIA GPUs. QUDA, which is written in CUDA C++, is under ongoing development, and it takes advantage of a number of features introduced with Kepler, such as support for bindless textures.

Fermi to Kepler GK110 at a glance

Table 1 lists selected features of the Fermi GF110 and GF114 architectures (compute 2.0 and 2.1 respectively) and Kepler GK110. Major differences include an increase in the number of streaming processors per SM from 48 on Fermi GF114 (32 on Fermi GF110) to 192 on Kepler. This increase is coupled with a reduction in clock speed, a combination that helps to achieve both higher performance and greater power efficiencies in the new architecture. The significant differences between the Kepler streaming multiprocessor and previous SMs have prompted NVIDIA to relabel Kepler's multiprocessor an SMX, to denote a next-generation SM.

The number of threads per warp on Kepler remains unchanged from Fermi at 32. However, the maximum number of warps per SM has gone from 48 on Fermi to 64 on Kepler. Furthermore, the number of thread blocks that can execute concurrently on a single SM has doubled from 8 on Fermi to 16 on Kepler. The combination of 4 warp schedulers and 8 instruction dispatch

[1] Lattice QCD is a computational approach to the theory of the strong nuclear force and a significant HPC application

units on Kepler means that two independent instructions can be issued to each of 4 warps per clock cycle. On Fermi, either 16KB or 48KB of a 64KB segment of on-chip memory can be used as an L1 cache, with the remainder reserved for shared memory. On Kepler, in addition to these configurations, this on-chip memory can be divided fifty-fifty between L1 cache and shared memory.

Other important changes include a doubling of the number of 32-bit registers per SM on Kepler, and a significant increase in the maximum number of registers per thread from 63 on Fermi and Kepler GK104 to 255 on Kepler GK110. Performance implications of the increase in the number of registers are discussed below.

The Tesla K20 accelerator has 13 active SMX units, containing a total of 2,496 SPs, and it delivers a reported peak double-precision performance of 1.17 Tflops. The higher-end Tesla K20X consists of 14 functional SMX units, containing 2,688 SPs, and achieves 1.31 Tflops peak double-precision performance [3]. In contrast, the Tesla M2090, which uses the compute 2.0 GF110 GPU and has 512 SPs in total, gives a reported peak double-precision performance of 665 Gflops [4].

	Fermi GF114 (GF110)	Kepler GK110
SPs per SM	48 (32)	192
Threads per SM	1536	2048
Thread blocks per SM	8	16
Warp schedulers per SM	2	4
Dispatch Units per SM	4(2)	8
Shared Memory/L1 cache	16/48KB	16/32/48KB
32-bit Registers per SM	32K	64K
Registers per thread	63	255

Table 1: A comparison of some of the features of Fermi GF114 and GF110 architectures (compute capability 2.1 and 2.0 respectively) and Kepler GK110.

Changes to the number of registers

The increase in the number of registers per SM from 32,768 on Fermi to 65,536 on Kepler may enable applications to achieve high thread-level paral-

lelism on Kepler hardware. For example, an application using 48 registers per thread achieves up to 42% occupancy on Fermi, but up to 63% occupancy on Kepler. However, maximizing occupancy does not necessarily maximize GPU utilization, and high performance can be achieved at low occupancy through instruction-level parallelism (ILP); i.e., latencies can be hidden by having each thread execute multiple independent instructions. In fact, in some applications, maximum performance can only be achieved through ILP [5], and on Kepler, some degree of ILP is required in order to approach theoretical peak performance. Applications running on Kepler GK110 can take advantage of the larger number of registers available to each thread to increase instruction level parallelism.

The increase in the number of registers in GK110 has resulted in very substantial performance gains in many critical QUDA routines. Most Lattice QCD applications are limited by global-memory bandwidth. The low number of registers per thread on Fermi can result in significant register spilling, adding to global-memory traffic, and severely impacting the performance of double-precision floating-point applications in particular. NVIDIA's GK110 white paper cites one particular QUDA kernel (the wilson dslash kernel) that achieves a 5.3x speedup over Fermi on Kepler GK110, most of which can be attributed to a reduction in register spilling. Another core QUDA routine that suffers from heavy register spilling on Fermi, the double-precision HISQ fermion-force calculation, exhibits a 3x speedup over Fermi on GK110.

Hyper-Q

The CUDA programming model supports concurrency on multiple levels. In addition to thread-based parallelism, CUDA streams enable multiple kernels to execute concurrently on a single device and allow kernel execution to overlap with the transfer of data between host and device. On Fermi-architecture GPUs, up to 16 kernels can execute concurrently on a single device, and kernel execution can overlap with a load from and write to host memory. On Fermi, this concurrency is supported by three hardware queues. Two of these are copy engine queues, dedicated to the transfer of data between host and device. One of the two copy engines deals with data transfers from the host, and the other handles transfers to the host exclusively. The remaining queue, the compute engine queue, is used to dispatch kernels to hardware. Ideally, independent kernels in separate CUDA streams should execute concurrently

provided the required compute resources are available on the device. In practice, however, it may be difficult to achieve maximum concurrency on Fermi devices because there is just one compute engine queue. On Fermi, the rules governing kernel dispatch include the following:

- Kernels are dispatched to hardware in the order in which they are issued to the queue.
- A kernel in a particular stream is only dispatched after the preceding kernels in that stream have completed.
- A kernel can only begin execution after all kernels that precede it in the queue have been dispatched irrespective of CUDA streams.

Hence, if a CUDA program involving multiple kernels and utilizing two streams, with no explicit synchronization between streams, is structured such that all kernels in stream 0 are issued before the kernels in stream 1, the first kernel in stream 1 can only begin execution after the last kernel in stream 0 has been dispatched, and this can only happen after all preceding kernels in stream 0 have completed. This issue pattern, where all kernels in a stream are issued before moving on to the next stream, is known as depth-first issue, and it should be avoided on Fermi GPUs.

The dependence of the kernels in one stream on kernels performing independent computations in another stream is often referred to as a false dependency. In principle, code can be structured to eliminate false dependencies and maximize concurrency, but this becomes increasingly difficult for applications that utilize more streams involving kernels that run for different durations.

The Hyper-Q technology introduced in compute 3.5 devices eliminates such false dependencies. Kepler GK110 supports up to 32 concurrent streams compared to 16 on Fermi, and unlike Fermi, kernels in different streams are assigned to separate hardware queues. Thus, returning to the example above, with Hyper-Q the dispatch of kernels in stream 1 will not depend on the completion of kernels in stream 0 provided sufficient compute resources are available for kernels in both streams to execute concurrently.

Hyper-Q technology also enables multiple CPU processes, such as different MPI ranks, to simultaneously launch work onto a shared GPU. In the CUDA 5.5 release, this functionality is enabled through the Multi-Process Service (MPS) feature, which was previously referred to as CUDA Proxy in CUDA 5.0.

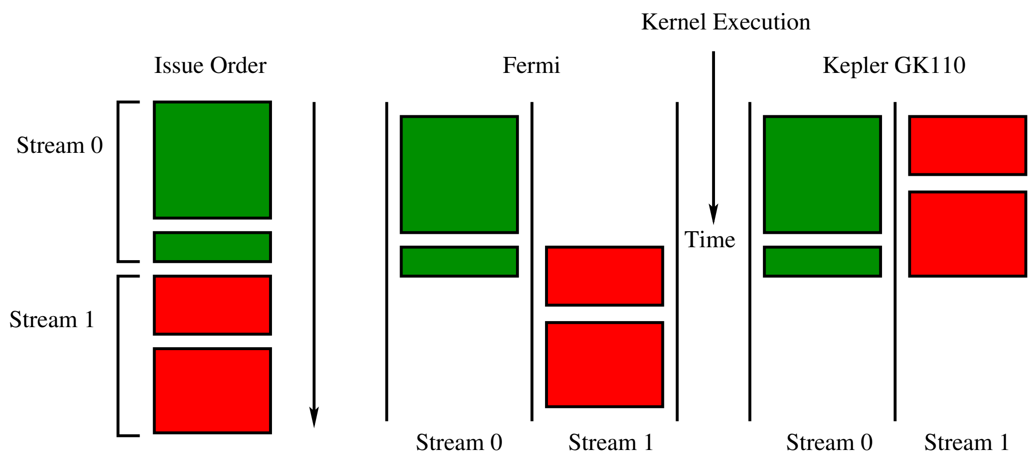

Figure 1: Kernel issue order and inter-stream dependencies. On the left, the kernels in stream 0 are issued before the kernels in stream 1. In Fermi and Kepler GK104, the use of a single hardware queue for all streams results in false dependencies. Thus, in this example, kernels in stream 1 can only execute after the kernels in stream 0 have been dispatched. The Hyper-Q feature in Kepler GK110 eliminates such dependencies by using a separate queue for each stream.

Dynamic parallelism

One of the best-known new features in Kepler GK110 is support for dynamic parallelism. This is the ability to launch kernels from within other kernels, whereas previously kernels could only be launched from the CPU. In certain applications, dynamic parallelism can eliminate much of the communication between host and device, resulting in an increase in concurrency and GPU utilization.

Overview

On compute 3.5 devices, any thread can create streams and launch kernels. A kernel launched from a parent thread executes on a child grid of threads. CUDA streams created on the device are accessible from the thread block of the parent thread and all threads within a block launch to the same default stream. However, streams created in one thread block are not accessible from other thread blocks in the parent grid. Similarly, kernels launched from a thread are only visible to other threads in the same thread block. As on the host, kernel launches on the device are asynchronous with respect to the parent thread.

To illustrate these ideas, consider the sample CUDA C++ code shown in Fig. 2. This code is similar to a saxpy routine to evaluate $y = ax + y$, where x and y are vectors and a is a scalar, except in this case the vector x is also modified: $x \to ax$. Using dynamic parallelism, this trivial example can be implemented using a nested kernel. In Fig. 2, the child kernel implements $x \to ax$, and the updated vector x is subsequently used in the parent kernel. In this example, thread 0 in each block launches a child kernel. Then, since kernel launches are asynchronous, thread 0 calls cudaDeviceSynchronize to force the child kernel to execute before it proceeds. The call to __syncthreads ensures that all threads in the block are synchronized with thread 0. The synchronization points in the parent kernel are essential, and the removal of either cudaDeviceSynchronize or __syncthreads would result in a race condition in this example.

Dynamic parallelism in action

Dynamic parallelism can greatly improve the performance and simplify the implementation of data-dependent and recursive algorithms on GPUs. The

```
__global__
void childKernel(float *x, float a, int offset){
  int id = offset + blockIdx.x*blockDim.x + threadIdx.x;
  x[id] = a*x[id];
}

__global__
void parentKernel(float *y, float *x, float a){
  int id = blockIdx.x*blockDim.x + threadIdx.x;
  if(threadIdx.x==0){
    childKernel<<<1,blockDim.x>>>(x, a, blockIdx.x*blockDim.x);
    cudaDeviceSynchronize(); // ensure childKernel has completed
  }
  __syncthreads(); // synchronize threads within block
  y[id] = x[id] + y[id];
}
```

Figure 2: Simple example of a nested kernel.

implementation of the quicksort algorithm using dynamic parallelism has been described in numerous NVIDIA presentations (see [6], for example). The CUDA 5.5 toolkit contains a number of sample codes that use dynamic parallelism, including quicksort and an LU decomposition code.

DiMarco and Taufer [7] have studied the impact of dynamic parallelism on clustering algorithms. These are heuristic algorithms used to partition data sets of N points into k ($k < N$) disjoint clusters based on some similarity criterion.The authors consider two different algorithms. One is the k-means algorithm, which is an iterative method that repeatedly reassigns data points to different clusters until a convergence criterion is satisfied. The second algorithm is a divisive clustering algorithm that initially assigns all the data points to a single cluster, which is divided into smaller clusters recursively. Clearly, the divisive algorithm is likely to benefit most from dynamic parallelism. In fact, the authors observe a slight decrease in performance when they apply dynamic parallelism to the k-means algorithm. On the other hand, dynamic parallelism results in performance gains ranging from 1.78x to 3.03x in tests of the divisive clustering algorithm performed on different data sets.

New texture features

Bindless textures

Designed for graphics applications specifically, textures are used more generally to maximize memory bandwidth in applications where global-memory reads do not satisfy coherency constraints but nonetheless exhibit a degree of spatial locality. Prior to compute capability 3.0, textures could only be accessed via the texture reference API. However, this API is subject to a number of limitations. For example, texture references have file scope and cannot be passed as function arguments. A texture reference declaration has the form:

texture⟨Type, Dim, ReadMode⟩ texRef;

where Type denotes the return type of the texture, Dim is the dimension of the texture, and ReadMode can take the values cudaReadModeElementType and cudaReadModeNormalizedFloat. The return type, dimension, and read mode attributes of a texture reference are thus fixed at compile time. At runtime, the texture reference must be bound to device memory with a call to cudaBindTexture or similar function before it can be used in a kernel. On Fermi, due to hardware restrictions, no more than 128 texture references can be bound to a kernel at any one time. Texture references can be unbound by calling cudaUnbindTexture.

The texture reference API is also supported on compute 3.x devices. However, on Kepler, the new texture object API offers all the functionality of texture references with better performance. Unlike texture references, texture objects are standard C++ objects that are created at runtime. The number of texture objects is not subject to the hardware restrictions that apply to texture references, and texture objects do not need to be bound and unbound. Hence, the texture-object feature is alternatively referred to as bindless textures.

Mike Clark of NVIDIA has described the performance benefits that bindless textures bring to QUDA in a ParallelForall blog post [9]. Clark highlights the fact that, in addition to the cost of binding and unbinding texture references (approximately 1 microsecond and 0.5 microseconds respectively), each texture reference that is bound to a kernel generates additional kernel overhead and that the additional overhead is incurred regardless of whether the

```
// kernel to copy an array of floating-point numbers
// using a texture reference.
// texRef is of type texture<float,1,cudaReadModeElementType>
__global__
void copyTexRef(float* y){
 int idx = blockIdx.x*blockDim.x + threadIdx.x;
 y[idx] = tex1Dfetch(texRef,idx);
}

// kernel to copy an array of floating-point numbers
// using a texture object.
__global__
void copyTexObject(float* y, cudaTextureObject_t texObj){
 int idx = blockIdx.x*blockDim.x + threadIdx.x;
 y[idx] = tex1Dfetch<float>(texObj,idx);
}
```

Figure 3: Sample copy kernels using texture references and texture objects.

texture is actually used in the kernel. Replacing texture references with texture objects eliminates this overhead. To demonstrate this, we implemented two simple copy kernels that use textures, one utilizing texture references and one with texture objects. These are shown in Fig. 3. Note that when using texture objects, the function tex1Dfetch is templated on the return type (in this case, float).

Fig. 4 shows the results of benchmark runs using these kernels on a Tesla K20c. Execution times are averaged over ten million kernel invocations. Cases 1 and 2 use the copyTexRef kernel from Fig. 3. In case 2, a second, redundant texture reference is bound before the kernel call. We stress that the times shown are kernel execution times only and that these figures do not include the time spent binding and unbinding textures. Note that the kernel that uses texture objects (case 3) is 0.4 microseconds faster than the kernel that uses a single texture reference (case 1), and that binding a second texture reference further increases the execution time for copyTexRef by 0.3 microseconds.

Although the differences in kernel overhead might appear small, switching to texture objects has substantially improved strong scaling in multi-GPU linear solvers in the QUDA library (Fig. 5).

```
# elements copied: 256
(gridSize,blockSize): (1,256)
Average Times :
Case 1 (Texture Reference)              : 4.85967 microseconds
Case 2 (Redundant Texture Reference)    : 5.16451 microseconds
Case 3 (Texture Object)                 : 4.44285 microseconds
```

Figure 4: Benchmark results for the copy kernels in Fig. 3 obtained on a Tesla K20c. The kernel that uses the texture object API achieves the best performance. Using a single texture reference (Case 1) incurs an additional 0.4 microsecond overhead, and binding a second, redundant reference to the copyTexRef kernel further increases the execution time by 0.3 microseconds.

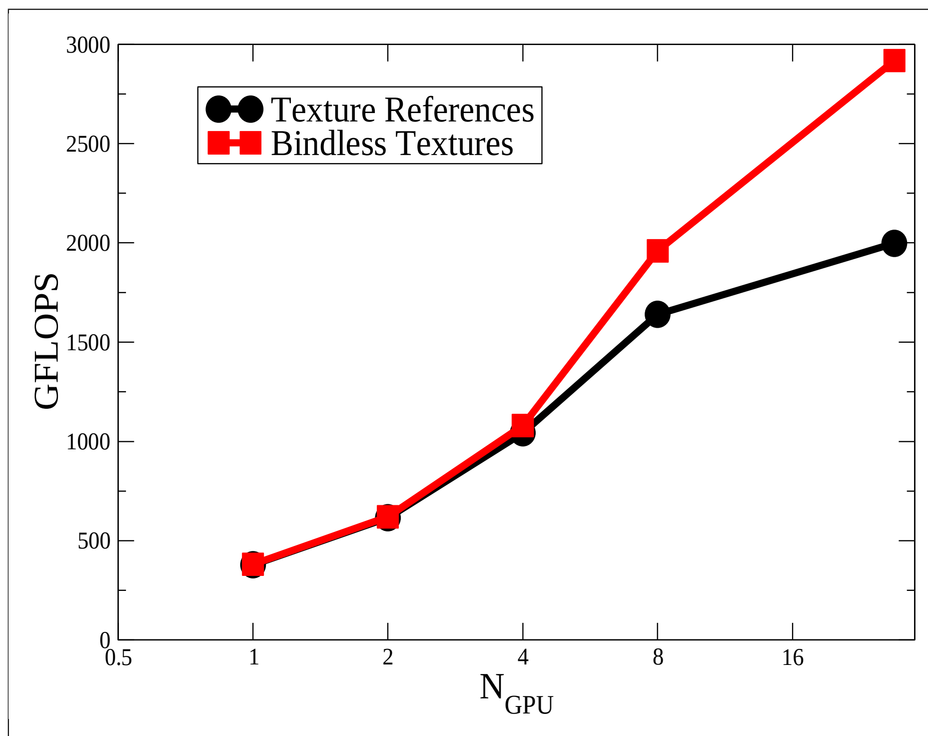

Figure 5: Strong scaling of a linear solver in the QUDA library using texture references and texture objects. In this example, switching to texture objects substantially improves performance on larger numbers of GPUs.

The __ldg intrinsic

Kepler GK110 also enables applications to utilize texture cache when reading from global memory without actually using the texture reference or texture object APIs. This is done using the __ldg function, which takes as an argument an address in global memory and returns the value contained at that address. The nvcc compiler will also attempt to use texture cache even when __ldg is not explicitly included in the source code. To allow this to happen, pointers to global memory must be marked with const __restrict__ qualifiers. Note that although extremely simple to use, the __ldg intrinsic lacks most of the functionality of the texture APIs. Furthermore, when using textures, address calculations are done in hardware, while the addresses of data that are read using the __ldg function are calculated in the kernel.

Shared-memory bandwidth

On Fermi-architecture GPUs, on-chip shared memory is divided into 32 4-byte banks, with successive 4-byte words assigned to successive banks. The bank index is given by the word address modulo 32, or the 5 least significant bits of the address. Bank conflicts occur when multiple threads in a warp access different words in the same bank, in which case, the accesses are serialized. Thus, for example, the kernel segment in Fig. 6 will result in a two-way bank conflict on Fermi, where threads in the first and second half-warps access the same banks

On Kepler, the number of shared-memory banks remains at 32, but the bank width has doubled to 8 bytes, with a corresponding 2x increase in shared-memory bandwidth. In addition, unlike Fermi, Kepler supports two shared-memory access modes, which we describe below. To avoid confusion, in the following discussion, we refer to 8-byte words and 4-byte words as words and half-words, respectively.

In 4-byte mode, successive half-words are assigned to successive shared-memory banks. Thus, when a warp of threads writes an array of doubles to shared memory, for example, the first half-warp of threads writes one half-word to each bank and the second half-warp writes a second half-word to each bank. Multiple accesses to the same shared-memory bank by different threads in a warp do not generate bank conflicts provided they are restricted to a single 256-byte aligned segment of shared memory (this implies that

```
__global__
void smemStrideKernel(float* gin, ...)
{
 extern __shared__ float sdata[];
 int tid = threadIdx.x;
 int id = tid + blockIdx.x*blockDim.x;

 sdata[2*tid] = gin[id];
 ...
```

Figure 6: A shared-memory access pattern that generates a two-way bank conflict on Fermi, but not on Kepler.

the strided access pattern in Fig. 6 is conflict free on Kepler). Conversely, multiple accesses to the same shared-memory bank involving different 256-byte segments will result in bank conflicts. For example, on Kepler, the shared-memory write in the code segment in Fig. 7 will suffer from a bank conflict whenever OFFSET assumes a non-zero value that is not an integer multiple of 32. Note that, in the absence of bank conflicts, the maximum shared-memory throughput of 8 bytes per bank per clock cycle can actually be achieved in four-byte mode.

As the name suggests, in 8-byte access mode, successive 8-byte words are assigned to successive banks. In this mode, the double-precision shared-memory write in Fig. 7 does not suffer from bank conflicts regardless of the value of OFFSET.

To ensure compatibility with applications developed on Fermi, codes built to run on Kepler default to 4-byte shared-memory mode. However, the access mode can be set explicitly in the code by calling cudaDeviceSharedMemConfig with cudaSharedMemBankSizeEightByte or cudaSharedMemBankSizeFourByte as an argument.

The increase in shared-memory bank width can result in immediate significant performance gains in applications that use 8-byte data types, such as codes performing double-precision floating-point arithmetic. NVIDIA's ParallelForall blog contains two posts describing the implementation of a finite-difference method in CUDA C++ that makes extensive use of shared memory [8]. The method is implemented on a three-dimensional grid with

```
__global__
void smemOffsetKernel(double* gin, ... )
{
 extern __shared__ double sdata[];
 int tid = threadIdx.x;
 int id = tid + blockIdx.x*blockDim.x;

 sdata[tid+OFFSET] = gin[id];
 ...
```

Figure 7: On Kepler, in 4-byte shared-memory access mode, the code sample above results in a two-way bank conflict for non-zero values of OFFSET that are not integer multiples of 32. Switching to 8-byte access mode eliminates this conflict.

directions labeled x,y, and z. The first blog post concerns the implementation of a derivative in the x direction using a 9-point stencil. The example is implemented in single-precision floating-point arithmetic, but modifying the code to use double-precision arithmetic is trivial. If we run the double-precision code with a 64×4 shared-memory tile using the default 4-byte access mode on a K20c, an average kernel execution time of 39 microseconds is reported. Switching to 8-byte mode reduces the measured kernel execution time to 37.2 microseconds, which corresponds to a performance gain of almost 5%.

As discussed above, applications that use 4-byte data types may also benefit from a reduction in bank conflicts on Kepler. However, it should be clear that shared-memory accesses that involve each thread in a warp reading or writing a single 4-byte word, such as an int or float, can achieve no more than half the available bandwidth on Kepler. In such cases, modifying the code to use int2 or float2 types instead can improve performance.

Additional features

GPUDirect RDMA

GPUDirect RDMA is the latest in the GPUDirect series of technologies that are designed to accelerate data transfer between GPUs. This latest technology enables direct data transfer between GPUs over Infiniband networks,

completely bypassing host memory. The alpha release of the Mellanox OFED driver supporting GPUDirect RDMA became available in Summer 2013, with the general release scheduled for the last quarter of 2013. Benchmarks of the new technology using MVAPICH2 show a 3x reduction in short-message latencies along with significant increases in MPI bandwidth. A series of presentations on GPUDirect RDMA is contained in the GTCExpress webinar 'Accelerating High Performance Computing with GPUDirect RDMA' [10].

Shuffle instructions

Another new feature of Kepler is a family of shuffle instructions. These allow threads within a warp to exchange data residing in registers without having to go through shared memory. At the present time, four shuffle functions are supported and these are overloaded for 4-byte integer and floating-point variables. Shuffle operations are faster than the corresponding shared-memory operations and the use of shuffle instructions may also indirectly improve performance by freeing up shared memory.

Atomic functions

The performance of atomic operations on global memory has improved significantly on Kepler GK110. This may allow atomics to be used more freely in software and facilitate the more rapid development of codes that achieve an acceptable base level of performance. GK110 also extends the set of native atomic functions that can operate on 64-bit integer values to include atomicMax and atomicMin, as well as the bitwise functions atomicAnd, atomicOr, and atomicXor. Previously, these functions were restricted to 32-bit integer types.

Conclusion

NVIDIA's Kepler architecture differs significantly from the previous-generation Fermi hardware, and Kepler supports a number of features not available on Fermi-class GPUs. The new features discussed here include Hyper-Q, dynamic parallelism, bindless textures, and changes to shared memory. Although applications developed on Fermi should perform well on Kepler hardware without modification, minor changes, such as switching from texture references to texture objects, or modifying code to fully utilize the greater

shared-memory bandwidth on Kepler, can result in noticable performance gains. Furthermore, support for dynamic parallelism drastically simplifies the implementation and increases the performance of many algorithms on Kepler. A firm understanding of the differences between Fermi and Kepler is therefore essential to ensure that applications achieve maximum performance on the latest generation of NVIDIA accelerators.

Acknowledgements

The author is grateful for many useful discussions with M. Clark.

References

[1] NVIDIA Kepler GK110 Architecture White Paper
`http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`

[2] QUDA: A library for QCD on GPUs
`http://lattice.github.io/quda/`

[3] `http://www.nvidia.com/object/tesla-servers.html`

[4] Tesla M-Class GPU Computing Modules - Accelerating Science
`http://www.nvidia.com/docs/IO/105880/DS-Tesla-M-Class-Aug11.pdf`

[5] V. Volkov. Better Performance at Lower Occupancy. GTC 2010.

[6] S. Jones. How Tesla K20 speeds quicksort, a familiar comp-sci code
`http://blogs.nvidia.com/blog/2012/09/12/how-tesla-k20-speeds-up-quicksort-a-familiar-comp-sci-code`

[7] J. DiMarco and M. Taufer. Performance Impact of Dynamic Parallelism on Different Clustering Algorithms and the New GPU Architecture. Proceedings of the DSS11 SPIE Defense, Security, and Sensing Symposium (2013).

Copyright Microway, Inc. 2013 – `www.microway.com`

[8] M. Harris. Finite Difference Methods in CUDA C/C++
`https://developer.nvidia.com/content/finite-difference-methods-cuda-cc-part-1`

[9] M. Clark. CUDA Pro Tip: Kepler Texture Objects Improve Performance and Flexibility
`https://developer.nvidia.com/content/cuda-pro-tip-kepler-texture-objects-improve-performance-and-flexibility`

[10] Accelerating High Performance Computing with GPUDirect RDMA
`http://on-demand.gputechconf.com/gtc/2013/webinar/gtc-express-gpudirect-rdma.pdf`