CHAPTER

More on CUDA and graphics processing unit computing

20

Mark Harris and Isaac Gelado

CHAPTER OUTLINE

Our main focus has been on scalable parallel programming. CUDA C and graphics processing unit (GPU) hardware have mostly played the role of programming platform for our examples and exercises. As we have demonstrated through the later chapters, parallel programming concepts and skills learned based on CUDA C can be easily adapted into other parallel programming platforms. In Chapter 18, Programming a heterogeneous computing cluster, for instance, most key concepts of MPI, such as processes, rank, and barriers have counterparts in CUDA C. Meanwhile, as was also discussed in Chapter 18, Programming a heterogeneous computing cluster, CUDA-enabled GPUs have become widely available in HPC systems. For many readers, CUDA C will likely be an important application development and deployment platform rather than a mere learning vehicle. In this case, the reader should understand advanced CUDA C features that have been designed to support high-performance programming at the application level. To illustrate, in Chapter 18, Programming a heterogeneous computing cluster, CUDA streams enable an MPI HPC application to overlap communication with computation. Such capability can achieve whole-application performance goals. With this considered, this chapter will provide an overview of the advanced features of CUDA C and GPU computing hardware that are essential in achieving high performance and maintainability of your applications. For each feature, we will present basic concepts as well as a brief history of its evolution through different generations of GPU computing. A sufficient understanding

Programming Massively Parallel Processors. DOI: http://dx.doi.org/10.1016/B978-0-12-811986-0.00020-0

of the concepts and history of each will help clear confusion surrounding them. The goal is to help you establish a conceptual framework for more detailed studies of these features.

20.1 MODEL OF HOST/DEVICE INTERACTION

We have thus far assumed a fairly simple model of interaction between a host and a device in a heterogeneous computing system. As presented in Chapter 2, Data parallel computing, each device in this model contains a device memory (CUDA global memory) that is separate from the host memory or the system memory. The data to be processed by a kernel running on a device must be transferred from the host memory to the device memory by calling the `cudaMemcpy()` function. The data produced by the device also need to be transferred from the device memory to the host by calling the `cudaMemcpy()` function before they can be utilized by the host. While the model exhibits simplicity and is easy to understand, it leads to several problems at the application level.

First, I/O devices such as disk controllers and network interface cards are designed to operate efficiently on the host memory. Since the device memory is separate from the host memory, input data have to be transferred from the host memory to the device memory, and output data need to be transferred from the device memory to the host memory. Such additional transfers increase the I/O latency and reduces the achievable throughput of the I/O operations. For a number of applications, the ability for I/O devices to operate directly on the device memory would improve the overall application performance and simplify the application code.

Second, the host memory is where the traditional programming systems place their application data structures. Some data structures are very large. The device memory in early generations of CUDA-enabled GPUs was small compared with the host memory, compelling application developers to partition their large data structures into chunks that fit into the device memory. To illustrate, in Chapter 15, Application case study—molecular visualization and analysis, the 3D electrostatic energy grid array was partitioned into 2D slices that are transferred between the host memory and the device memory. For many applications, having the entire data structures reside in the device memory would be preferable. For some, there may not be a satisfactory way to partition the data structure into smaller chunks. For these applications, it would be best if the GPU can directly access the data in the host memory or have the CUDA runtime system software migrate the data that are actually used during kernel execution.

These limitations of the host/device interaction model were rooted in the limitations of the memory architecture of early generations of CUDA-enabled GPUs. In these early devices, the only viable host/device interaction model for applications was the simple model that was assumed in the previous chapters. As more applications adopt GPU computing, CUDA system software developers, and GPU hardware designers have been motivated to provide better solutions. Researchers have been well aware of these needs and have proposed solutions since the early days of CUDA

[GSC 2010]. The remainder of this section will discuss a brief history of advancements that address these limitations.

Zero-copy memory and unified virtual address space (UVAS). In 2009, CUDA 2.2 introduced zero-copy access to system memory. This operation enables the host code to supply a special device data pointer to host the memory to a kernel. The code running on the device can use this pointer to directly access the host memory through the system interconnect, such as the PCIe bus without calling to `cudaMemcpy()`. Zero-copy memory is pinned host memory (see chapter: Programming a heterogeneous computing cluster) and is allocated by calling `cudaHostAlloc()`, with `cudaHostAllocMapped` as the value of the flag argument. The other values of the flag argument are for more advanced usage, such as zero-copy memory allocation. The data pointer returned by `cudaHostAlloc()` cannot be directly passed to the kernel; the host code has to obtain first a valid device data pointer using `cudaHostGetDevicePointer()`and then pass the device data pointer returned by this function to the kernel. This process shows that different data pointers for host and device codes are used to access the same physical memory.

As explained in Chapter 18, Programming a heterogeneous computing cluster, the host memory pages must be pinned to prevent the operating system from accidentally paging out the data while the GPU is accessing them. Obviously, the access will suffer from the long latency and limited bandwidth of the system interconnect. The bandwidth of the system interconnect is typically less than 10% of the global memory bandwidth. As we have learned in Chapter 5, Performance considerations, the performance of a kernel is typically limited by the global memory bandwidth unless tiling techniques are used to drastically reduce the number of global memory accesses per floating-point operation. If the majority of the memory accesses of a kennel are to zero-copy memory, the execution speed of the kernel can even be more severely limited by the bandwidth of the system interconnect. Therefore, zero-copy memory should be used for application data structures that are occasionally and sparsely accessed by a kernel running on a GPU.

In 2011, CUDA 4 introduced Unified Virtual Addressing. Until this CUDA release, the host and the device had their own virtual address spaces, with each of them mapping host or device data pointers to physical host or device memory locations. These disjoint virtual address spaces imply that the same physical memory location could be accessed by different data pointers in the host and the device, which effectively happens when zero-copy memory is used. The UVAS, first introduced by the GMAC library [GSC 2010] and adopted in CUDA 4, uses a single virtual address space shared by the host and the device. The UVAS ensures that each physical memory address is only mapped to one virtual memory location. This restriction in mapping enables the CUDA runtime to determine whether a data pointer is referencing the host or device memory by merely inspecting its virtual memory address on the host. This feature eliminates the need to specify the data copy direction on `cudaMemcpy()` calls.

Notably, the UVAS in CUDA 4 does not guarantee the accessibility of the data referenced by a pointer. To illustrate, the host code cannot use a device pointer returned by `cudaMalloc()` to directly access the device memory, and vice versa.

Zero-copy memory is the exception: the host code can directly pass a pointer to zero-copy memory as a kernel launch parameter to the device. When the kernel code dereferences this zero-copy pointer, the pointer value is translated to a physical host memory location and accessed directly through the PCIe. This approach does not necessarily allow the kernel code to dereference a pointer value read from a memory location, such as following a chain of pointers while traversing a linked data structure, unless all memory has been allocated using `cudaHostAlloc()`.

The limitations in both the types of data structures that can be supported and the bandwidth of data accesses of zero-copy memory motivate further improvements in the memory model of GPU architectures beyond UVAS.

Large virtual and physical address spaces. One fundamental limitation of early CUDA-enabled GPUs is the size of their virtual and physical addresses. These early devices support 32-bit virtual addresses and up to 32-bit physical addresses. For these devices, the size of the device memory is limited to 4 GB, the maximal amount of memory that can be addressed with 32 physical address bits. Furthermore, the CUDA kernels can only operate on data sets whose sizes are less than 4 GB, the maximal number of virtual memory locations that can be accessed through 32-bit pointers, regardless of whether the data set resides in the host memory or the device memory. Furthermore, modern CPUs are based on 64-bit virtual addresses with 48 bits actually utilized. These host virtual addresses cannot be accommodated by the 32-bit virtual addresses used by GPUs, which further restricts the types of data structures supported by zero-copy memory.

To remove this limitation, recent GPU generations, starting with the Kepler GPU architecture introduced in 2013, have adopted a modern virtual memory architecture with 64-bit virtual addresses and physical addresses of at least 40 bits. Among the obvious benefit are that that these GPUs can incorporate more than 4 GB of device memory and CUDA kernels can now operate on very large data sets. While the enlarged virtual and physical address spaces obviously enable the use of large device memories, they also allow for much improved host/device interaction models. The host and the device can now use exactly the same pointer value to access a piece of data, whether it is in the host memory or the device memory.

The large GPU physical address space allows the CUDA system software to place the device memory of different GPUs in the system into a unified physical address space. The benefit is that one GPU can directly access the memory of any other GPU attached to the same PCIe bus by simply dereferencing a data pointer mapped to the physical address of such GPU. Prior to the Kepler GPU, communication among different GPUs (e.g., halo exchange in the stencil example in see chapter: Programming a heterogeneous computing cluster) was only possible through device–to–device memory copies triggered by the host code. This resulted in extra memory consumed to store the data being copied from other GPUs and extra performance overheads because of the memory copy operations. Direct access to other device memories in the system enables merely passing the device pointer to the other GPU on the kernel launch and using it to load and/or store the data set that needs to be communicated.

Unified Memory. In 2013, CUDA 6 introduced Unified Memory, which creates a pool of managed memory shared between the CPU and GPU, thus bridging the CPU–GPU divide. Managed memory is accessible to both the CPU and GPU with the use

of a single pointer. Variables in the managed memory can reside in the CPU physical memory, the GPU physical memory, or even both. The CUDA runtime software and hardware implement data migration and coherence support such as the GMAC system [GSC 2010]. The net effect is that the managed memory resembles a CPU memory to code running on the CPU and GPU memory to code running on the GPU. The application must certainly perform appropriate synchronization operations such as barriers or atomic operations to coordinate any concurrent accesses to the managed memory locations. A shared global virtual address space allows all variables in an application to have unique addresses. Such memory architecture, when exposed by programming tools and runtime system to applications, can provide major benefits.

One such benefit is the reduced amount of effort required to port a CPU code to CUDA. In Fig. 20.1, we present a simple CPU code on the right side. With Unified Memory, the code can be ported to CUDA with two simple changes. The first change is to use `cudaMallocManaged()` and `cudaFree()` instead of `malloc()` and `free()`. The second change is to launch a kernel and perform device synchronization rather than call the `qsort_char()` function. Obviously, one still needs to write or have access to a parallel qsort_char kernel. What is shown here is that the change to the host code is straightforward and easy to maintain.

However, the performance of the CUDA 6 Unified Memory was limited by the hardware capabilities of Kepler and Maxwell GPU architectures: The contents of all managed memory locations modified by the CPU had to be flushed out to the GPU device memory before any kernel launch. The CPU and GPU could not simultaneously access a managed memory allocation and the Unified Memory address space was limited to the size of the GPU physical memory. These limitations exist because these GPU architectures lacked the ability to support coherence between the host and device memories, and data migration was mostly performed by software.

In 2016, the Pascal GPU architecture added features to further simplify programming and sharing of memory between CPU and GPU, and further reduce the effort

CPU code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);
  fread(data, 1, N, fp);
  qsort_char(data, N, 1);

  use_data(data);
  free(data);
}
```

CUDA 6 code with unified memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);
  fread(data, 1, N, fp);
  qsort_char<<<…>>>(data, N, 1);
  cudaDeviceSynchronize()
  use_data(data);
  cudaFree(data);
}
```

FIGURE 20.1

Unified Memory simplifies porting of CPU code (left) to CUDA code (right).

required to use GPUs for significant speedups. Two main hardware features enable these improvements: support for large address spaces and handling of page faults.

The Pascal GPU architecture extends GPU addressing capabilities to 49-bit virtual addressing. Such extension can sufficiently cover the 48-bit virtual address spaces of modern CPUs, as well as GPU memory. This enhancement allows Unified Memory programs to access the full address spaces of all CPUs and GPUs in the system as a single virtual address space rather than be limited by the amount of data that can be copied to the device memory. Consequently, the CPUs and GPUs can truly share the pointer values, enabling the GPUs to traverse linked data structures in the host memory.

Memory page fault handling support in the Pascal GPU architecture is a crucial feature that provides a more seamless Unified Memory functionality. Combined with the system-wide virtual address space, the ability to handle page faults eliminates the need for the CUDA system software to synchronize (flush) all managed memory contents to the GPU before each kernel launch. The CUDA runtime can implement a coherence mechanism by allowing the host and the device to invalidate each other's copy when they modify a variable in the managed memory. Invalidation can be done using the page mapping and protection mechanisms. When launching a kernel, the CUDA system software no longer has to bring all GPU copies of the managed memory data up to date. If the kernel accesses a piece of data whose copy in the device memory has been invalidated by the host, the GPU will handle a page fault to bring the data up to date and resume execution.

If a kernel running on the GPU accesses a page that does not reside in its device memory, it also will take a page fault, allowing the page to be automatically migrated to the GPU memory on-demand. Alternatively, the page may be mapped into the GPU address space for access over the system interconnects (mapping on access can sometimes be faster than migration) if the data are to be accessed only occasionally. Unified Memory is system-wide: GPUs (and CPUs) can fault and migrate memory pages either from the CPU memory or from the memory of other GPUs in the system. If a CPU function dereferences a pointer and accesses a variable mapped to the GPU physical memory, the data access would still be serviced, although perhaps at a longer latency. Such capability allows the CUDA programs to more easily call legacy libraries that have not been ported to GPUs. In the current CUDA memory architecture, the developer must manually transfer data from the device memory to the host memory in order to use legacy library functions to process them on the CPU.

The Unified Memory with a page fault handling capability enables a much more general CPU/GPU interaction mechanism compared with zero-copy memory. It allows the GPU to traverse large data structures in the host memory. Starting with the Pascal architecture, a GPU device can traverse a linked data structure even if the data structure does not reside in zero-copy memory. The reason is that the same pointer value is used in the host and device codes to refer to the same variable. Thus, the embedded pointer values of a linked data structure built by the host can be traversed by the device, and vice versa. In some application areas such as CAD, the host physical memory system may have hundreds of gigabytes of capacity. These physical memory systems are needed because the applications require the entire data set to

be "in core." With the ability to directly access very large CPU physical memories, GPUs can feasibly accelerate these applications.

20.2 KERNEL EXECUTION CONTROL

Function calls within kernel functions. Early CUDA versions did not allow function calls during kernel execution. Although the source code of kernel functions can appear to have function calls, the compiler must be able to inline all function bodies into the kernel object so that function calls are present in the kernel function at runtime. Although this function inlining model works reasonably well for performance-critical portions of various applications, it does not support the software engineering practices in more sophisticated applications. In particular, the model does not support system calls, dynamically linked library calls, recursive function calls, and virtual functions in object oriented languages such as C++.

More recent device architectures such as Kepler support function calls in kernel functions at runtime. This feature is supported in CUDA 5 and beyond. The compiler is no longer required to put inline the function bodies, but it can still do so as a performance optimization. This capability is partly enabled by cached, fast implementation of massively parallel call frame stacks for CUDA threads. It makes CUDA device code much more "composable" by allowing different authors to write different CUDA kernel components and assemble them all together without heavy re-design costs. It also allows software vendors to release device libraries without a source code for intellectual property protection.

Support for function calls at runtime allows recursion and will significantly ease the burden on programmers as they transition from legacy CPU-oriented algorithms toward GPU-tuned code for divide-and-conquer types of computation. The QuadTree example in Chapter 13, CUDA dynamic parallelism, demonstrates the benefit of recursively launching kernel functions in accordance with the data characteristics discovered at runtime. This also allows easier implementation of graph algorithms where data structure traversal often naturally involves recursion. In some cases, developers will be able to "cut and paste" a CPU code into a CUDA kernel and then obtain a reasonably performing kernel, although continued performance tuning would still be beneficial.

With the function call support, kernels can now call standard library functions such as `printf()` and `malloc()`. In our experience, the ability to call `printf()` in a kernel provides a subtle but important aid in debugging and supporting kernels in production software. Many end users are nontechnical and cannot be easily trained to run debuggers in order to provide developers with more details on what occurred before a crash. The ability to execute `printf()` in the kernel allows the developers to add a mode to the application to dump internal state so that the end users can submit meaningful bug reports.

Exception handling in kernel functions. Early CUDA systems did not support exception handling in kernel code. While not a significant limitation for

performance-critical portions of many high-performance applications, it often incurs software engineering costs in production quality applications that rely on exceptions to detect and handle rare conditions without executing a code to explicitly test for such conditions.

With the availability of limited exception handling support, CUDA debuggers allow a user to perform a step-by-step execution, set breakpoints, and/or run a kernel until an invalid memory access occurs. In each case, the user can inspect the values of kernel local and global variables when the execution is suspended. In our experience, the CUDA debugger is a very helpful tool for detecting out-of-bounds memory accesses and potential race conditions.

Simultaneous execution of multiple kernels. Early CUDA systems allow only one kernel to execute on each GPU device at any point in time. Multiple kernel functions can be submitted for execution. However, they are buffered in a queue that releases the next kernel after the current one completes execution. The Fermi GPU architecture and its successors allow the simultaneous execution of multiple kernels from the same application, which reduces the pressure for the application developer to "batch" multiple kernels into a larger kernel in order to more fully utilize a device. In addition, it is at times beneficial to partition work into chunks that can execute with different levels of priority.

A typical benefit is for parallel cluster applications that segment work into "local" and "remote" partitions, where remote work is involved in interactions with other nodes and reside on the critical path of global progress (see chapter: Programming a heterogeneous computing cluster). In previous CUDA systems, kernels needed to perform a lot of work to ensure that the device is utilized efficiently, and one had to be careful not to launch local work such that global work could be blocked. This limitation meant choosing between underutilizing the device while waiting for remote work to arrive, or eagerly starting on local work to keep the device productive at the cost of increased latency for completing remote work units [PS 2009]. With multiple kernel executions, the application can use much smaller kernel sizes for launching work. Consequently, when high-priority remote work arrives, the application can start running with low latency instead of being stuck behind a large kernel of local computation.

Hardware queues and dynamic parallelism. In Kepler and CUDA 5, the multiple kernel launch facility is extended by the addition of multiple hardware queues, which allow considerably more efficient scheduling of thread blocks from multiple kernels, including kernels in multiple streams. In addition, the CUDA dynamic parallelism feature (see Chapter 13: CUDA dynamic parallelism) allows GPU work creation: GPU kernels can launch child kernels, asynchronously, dynamically, and in a data-dependent or compute load-dependent fashion. This process reduces CPU–GPU interaction and synchronization because the GPU can now manage more complex workloads independently. The CPU is, in turn, free to perform other useful computations.

Interruptible kernels. The Fermi GPU architecture allows a running kernel to be "canceled," enabling the creation of CUDA-accelerated apps that allow the user to abort a long-running calculation at any time, without requiring significant design effort on the part of the programmer. This property enables the implementation of

user-level task scheduling systems that can more efficiently perform load balance between GPU nodes of a computing system and allows a smoother handling of cases where one GPU is heavily loaded and may be running more slowly than its peers [SHG 2009].

20.3 MEMORY BANDWIDTH AND COMPUTE THROUGHPUT

Double-precision speed. Early devices perform double-precision floating arithmetic with significant speed reduction (around eight times slower) compared with single precision. The floating-point arithmetic units of Fermi and its successors have been significantly strengthened to perform double-precision arithmetic at about half the speed of a single-precision arithmetic. Applications that are intensive in double-precision floating-point arithmetic benefit tremendously. Other applications that use double precision carefully and sparingly observe less performance impact.

In practice, the most significant benefit will likely be obtained by developers who are porting CPU-based numerical applications to GPUs. With improved double-precision speed, developers will have little incentive to spend the effort to evaluate whether their applications or portions of their applications can fit into single precision. The ability to use double-precision arithmetic without significant performance penalty can significantly reduce the development cost for porting CPU applications to GPUs and address a major criticism of GPUs by the high-performance computing community.

Some applications that are operating on smaller input data types (8-bit, 16-bit, or single-precision floating point) may continue to benefit from using single-precision arithmetic, because of the reduced bandwidth in using 32 vs 64-bit data. Applications such as medical imaging, remote sensing, radio astronomy, seismic analysis, and other natural data frequently fit into this category. The Pascal GPU architecture introduces a new hardware support for computing with 16-bit half-precision numbers to further improve the performance and energy efficiency of these applications.

Better control flow efficiency. Starting with the Fermi GPU architecture, CUDA systems have adopted a general compiler-driven predication technique [MHM 1995] that can more effectively handle control flow than previous CUDA systems. While this technique was moderately successful in VLIW systems, it can provide even more dramatic speed improvements in GPU warp-style SIMD execution systems. This capability broadens the range of applications that can take advantage of GPUs. In particular, major performance benefits can potentially be realized for applications that are highly data-driven, such as ray tracing, quantum chemistry visualization, and cellular automata simulation.

Configurable caching and scratchpad. The shared memory in early CUDA systems served as programmer-managed scratch memory and increased the speed of applications where key data structures have localized, predictable access patterns. Starting with the Fermi GPU architecture, the shared memory has been enhanced to a larger on-chip memory that can be configured to be partially cache memory and partially shared memory, which allows coverage of both predictable and less predictable

access patterns to benefit from on-chip memory. This configurability allows programmers to apportion the resources according to the best fit for their application.

Applications in an early-stage design ported directly from the CPU code will benefit greatly from caching as the dominant part of on-chip memory. This would further smooth performance tuning by increasing the level of "easy performance" when a developer ports a CPU application to GPU.

Existing CUDA applications and those with predictable access patterns will have the ability to increase their use of fast shared memory by a factor of three while retaining the same device "occupancy" they had on previous-generation devices. For CUDA applications whose performance or capabilities are limited by the size of shared memory, the three times increase in size will be a welcome improvement. For example, in stencil computation (see chapters: Parallel patterns: convolution and Programming a heterogeneous computing cluster) such as finite difference methods for computational fluid dynamics, the state loaded into the shared memory also includes "halo" elements from neighboring areas.

The relative portion of halo decreases as the size of the stencil increases. In 3D simulation models, the halo cells can be comparable in data size as the main data for currently shared memory sizes. This can significantly reduce the effectiveness of the shared memory because of the significant portion of the memory bandwidth spent on loading halo elements. To illustrate, if the shared memory allows a thread block to load an 8^3 (=512)-cell stencil into the shared memory, with one layer of halo elements on every surface, only 6^3 (=216), or less than half of the loaded cells are the main data. The bandwidth spent on loading the halo elements is larger than that spent on the main data. A threefold increase in shared memory size allows some of these applications to have a more favorable stencil size where the halo accounts for a much smaller portion of the data in shared memory. In our example, the increased size would allow an 11^3 (=1331) tile to be loaded by each thread block. With one layer of halo elements on each surface, a total of 9^3 (=729) cells, or more than half of the loaded elements, are main data. This enhancement improves the memory bandwidth efficiency and the performance of the application.

Enhanced atomic operations. The atomic operations in the Fermi GPU architecture are much faster than those in previous CUDA systems, and the atomic operations in Kepler are still faster. In addition, the Kepler atomic operations are more general. The atomic operations over shared memory variables in the Maxwell GPU architecture are further enhanced in their throughput. Atomic operations are frequently used in random scatter computation patterns such as histograms (see Chapter 9: Parallel patterns—parallel histogram computation). Faster atomic operations reduce the need for algorithm transformations such as prefix sum (see chapter: Parallel patterns: prefix sum) [SHZ 2007] and sorting [SH 2009] to implement such random scattering computations. These transformations tend to increase the number of kernel invocations and the total number of operations required to perform the target computation. Faster atomic operations can also reduce the need to involve the host CPU in algorithms that perform collective operations or where multiple thread blocks

update shared data structures, thereby reducing the data transfer pressure between CPU and GPU.

Enhanced global memory access. The speed of random memory access is much faster in Fermi and Kepler than in earlier GPU architectures. Programmers can be less concerned about memory coalescing. This improvement allows more CPU algorithms to be directly used in the GPU as an acceptable base, further smoothing the path of porting applications that access diverse data structures, such as ray tracing, and other applications that are heavily object-oriented and may be difficult to convert into perfectly tiled arrays.

The Pascal GPU architecture incorporates high-bandwidth memory version 2 3D-stacked DRAM, which provides up to thrice the memory bandwidth of previous-generation NVIDIA Maxwell architecture GPUs. Pascal is also the first architecture to support the new NVLink processor interconnect, which gives Tesla P100 up to five times the GPU–GPU and GPU–CPU communication performance of PCI Express 3.0. This new interconnect greatly improves the scalability of multi-GPU computation within a node, as well increases the efficiency of data sharing between GPUs and NVLink-capable CPUs.

20.4 PROGRAMMING ENVIRONMENT

Unified device memory space. In early CUDA devices, shared, memory, local memory, and global memory form their own separate address spaces. The developer can use pointers into the global memory but not others. Starting with the Fermi Architecture introduced in 2009, these memories are parts of a unified address space. This unified address space enables a single set of load/store instructions and pointer addresses to access any of the GPU memory spaces (global, local, or shared memory) rather than different instructions and pointers for each. This makes it easier to abstract which memory contains a particular operand, allowing the programmer to deal with this only during allocation, and making it simpler to pass CUDA data objects into other procedures and functions, irrespective of which memory area they come from.

It makes CUDA code modules much more "composable"; i.e., a CUDA device function can now accept a pointer that may point to any of these memories. To illustrate, without a unified GPU address space, a device function needs to have one implementation for each type of memory that one of its arguments can reside in. Unified GPU address space allows variables in all main types of GPU memories to be accessed similarly, thus allowing one device function to accept arguments that can reside in different types of GPU memory. The code would run faster if a function argument pointer points to a shared memory location and slower if it points to a global memory location. The programmer can still perform manual data placement and transfers as a performance optimization. This capability has significantly reduced the cost of building production-quality CUDA libraries. It also enabled full C and C++ pointer support, which was a significant advancement at the time.

Future CUDA compilers will include enhanced support for C++ templates and virtual function calls in kernel functions. Although hardware enhancements, such the runtime function calling capability, are in place, enhanced C++ language support in the compiler has been taking more time. With these enhancements, future CUDA compilers will support most mainstream C++ features. For instance, using C++ features such as new, delete, constructors, and destructors in kernel functions is already supported in recent compiler releases.

New and evolved programming interfaces will continue to improve the productivity of heterogeneous parallel programmers. As shown in Chapter 19, Parallel programming with OpenACC, OpenACC allows developers to annotate their sequential loops with compiler directives to enable a compiler to generate CUDA kernels. Appendix B shows that one can use the Thrust library of parallel type-generic functions, classes, and iterators to describe their computation and have the underlying mechanism generate and configure the kernels that implement the computation. In Appendix C, we presented CUDA FORTRAN that allows FORTRAN programmers to develop CUDA kernels in their familiar language. In particular, CUDA FORTRAN offers strong support for indexing into multidimensional arrays. Appendix D provides an overview of the C++AMP interface that allows developers to describe their kernels as parallel loops that operate on logical data structures such as multidimensional arrays in a C++ application. We fully expect that new innovations will continue to arise to further boost the productivity of developers in this exciting area.

Profiling with critical-path analysis. In heterogeneous applications that perform significant computations on both CPUs and GPUs, locating the best place to spend optimization effort presents a challenge. Ideally, when optimizing a code, one would like to target the locations in the application that will provide the highest speedup for the least effort. To this end, CUDA 7.5 introduced Program Counter (PC) sampling, providing instruction-level profiling so that the user could pinpoint specific lines of code that require the most time in his/her application.

However, a challenge facing the users of such profilers is that the longest-running kernel in an application is not always the most critical optimization target. As Fig. 20.2 shows, kernel X is the longer running kernel. However, its execution time is fully overlapped with the CPU execution activity A. Any further improvement in

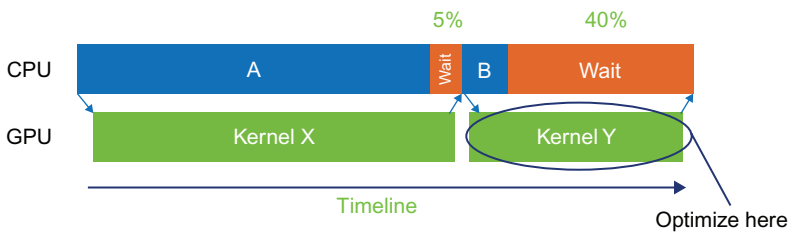

FIGURE 20.2

Importance of critical-path analysis for identifying the key kernels to optimize.

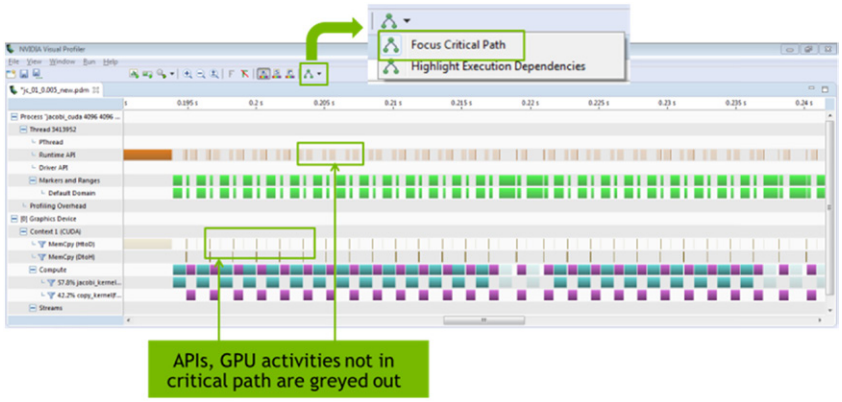

FIGURE 20.3

Application critical-path analysis in CUDA 8 Visual Profiler.

the execution time of kernel X will unlikely improve the application performance. While the execution time of kernel Y is not as long as kernel X, it is on the critical path of the application execution. The CPU is idling while waiting for the completion of kernel Y; speeding up kernel Y will reduce the time the CPU spends waiting. Thus, it is the best optimization target.

In 2016, the Visual Profiler in CUDA 8 provides critical-path analysis between GPU kernels and CPU CUDA API calls, enhancing the precise targeting of optimization efforts. Fig. 20.3 shows critical path analysis in the CUDA 8 Visual Profiler. GPU kernels, copies, and API calls that are not on the critical path are grayed out. Only the activities on the critical path of the application execution are highlighted in color. This allows the user to easily identify the kernels and other activities to target his/her optimization efforts.

20.5 FUTURE OUTLOOK

The evolution of CUDA continues to increase its support for developer productivity and modern software engineering practices. With the new capabilities, the range of applications that will satisfactorily perform at minimal development costs will expand significantly. Developers have experienced the reduction in application development, porting, and maintenance costs compared with previous CUDA systems. The existing applications developed with Thrust and similar high-level tools that automatically generate CUDA code will also likely get an immediate boost in their performance. The benefit of hardware enhancements in memory architecture, kernel execution control, and compute core performance will be visible in the associated Software Development Kit (SDK) releases; however, the true potential of these enhancements may take years to be fully exploited in the SDKs and runtimes.

For example, the true potential of the hardware virtual memory capability will likely be fully achieved only when a shared global address space runtime that supports direct GPU I/O and peer-to-peer data transfer for multi-GPU systems becomes widely available. We predict an exciting time for innovations from both industry and academia in programming tools and runtime environments for GPU computing in the next few years.

REFERENCES

Gelado, I., Stone, J. E., Cabezas, J., Patel, S., Navarro, N., & Hwu, W. W. (2010). An asymmetric distributed shared memory model for heterogeneous parallel systems. In: *The ACM/IEEE 15th international conference on architectural support for programming languages and operating systems (ASPLOS'10)*. Pittsburgh, PA. March 2010.

Mahlke, S. A., Hank, R. E., MCormick, J. E., August, D. I., & Hwu, W. W. (1995). A comparison of full and partial predicated execution support for ILP processors. In: *Proceedings of the 22nd annual international symposium on computer architecture* (pp. 138–150). Santa Margherita Ligure, Italy. June 1995.

Phillips, J., & Stone, J. (October 2009). Probing biomolecular machines using graphics processors. *Communications of ACM*, *Vol 52*(No. 10), 34–41.

Satish, N., Harris, M., & Garland, M. (2009). Designing efficient sorting algorithms for manycore GPUs. In: *Proceedings of the 23rd IEEE international parallel & distributed processing symposium*. May 2009.

Sengupta, S., Harris, M., Zhang, Y., & Owens, J. D. (2007). Scan primitives for GPU computing. In: *Proceedings of the graphics hardware* (pp. 97–106). August 2007.

Stone, J. E., & Hwu, W. W. (2009). *WorkForce: A lightweight framework for managing multi-GPU computations, technical report*. Champaign, IL: IMPACT Group, University of Illinois.