WHITE PAPER

Debugging CUDA-Accelerated Parallel Applications With TotalView

An Integrated Approach to Heterogeneous Multi-Tier Parallelism

Introduction

CUDA introduces developers to a number of new concepts (such as kernels, streams, warps, and explicit multi-level memory) that are not encountered in serial or other parallel programming paradigms. Visibility into these elements is critical for troubleshooting and tuning applications that make use of CUDA. This paper will highlight CUDA concepts, the impact of those concepts for troubleshooting CUDA, and how TotalView debugger helps users deal with these new CUDA-specific constructs. CUDA is frequently used alongside MPI parallelism and host-side multi-core and multi-thread parallelism. The TotalView parallel debugger provides developers with an integrated view of all three levels of parallelism within a single debugging session.

Contents

NVIDIA's CUDA language extension provides a very interesting opportunity for scientists and developers to accelerate the runtime performance of computationally intensive applications running on a single workstation or on a cluster of dedicated nodes augmented with GPU cards.

However, as developers begin adapting their applications for the GPU, they are finding out that harnessing the available performance requires mastering the concepts of data parallel programming and developing a fairly sophisticated understanding of the GPU architecture.

Parallel development and debugging tools are being adapted to help developers tackle both of these objectives. With TotalView, users can seamlessly debug apps that leverage CUDA in the context of multiple tiers of parallelism – device, host, and cluster.

HPC Debugging

High performance computing (HPC) applications, which typically run as distributed parallel applications on cluster type machines, require significant effort to write. Scientists and developers need to think about breaking tasks down into small units that can be distributed across that cluster. They need to pay special attention to the sequence of operations and the data and resource dependencies that different parts of the application have. They must also be conscious about what data is needed where, and the bandwidth and latency involved with moving data around the cluster.

Frameworks, libraries, and languages like MPI, UPC, Global Arrays, Co-Array FORTRAN, OpenMP, and pthreads provide powerful and flexible mechanisms that can be used to construct parallel applications and address the challenges of instantiating and managing parallel programs, providing synchronization and data movement between parallel tasks as required. Regardless of whether it is implicit or explicit, there is a tremendous degree of complexity that arises from parceling up the program and data while ensuring data movement and appropriate (but not too much) synchronization. This complexity becomes an issue when it comes to validating, troubleshooting, and debugging applications.

Debugging has always been challenging; there is an old adage that debugging a bit of code is at least twice as hard as writing that same bit of code — and that statement was primarily made in the context of serial applications running on a single core.

All of this adds to the complexity of validating, debugging, and tuning parallel applications. As if distributed architectures were not enough, recent trends have introduced heterogeneous computational elements into the mix.

Taking a given algorithm and making it parallel frequently means introducing 1) additional code and 2) more complicated data structures to facilitate distributed execution and data movement. Both cases provide additional opportunities for errors to creep in.

Parallel applications execute many different bits of code at the same time across a large set of distinct nodes. They frequently perform computations that are interconnected enough that an error or oversight anywhere may either cause a local failure or set in motion a series of events that causes incorrect data to be transmitted to another node. This can lead to a wrong answer or crash somewhere else in this distributed system. A failure to ensure proper synchronization can lead to failures that may happen only rarely — when natural perturbations introduced by the system or other factors external to the program lead to rare and unexpected sequencing of operations. Furthermore, synchronization-dependent errors may become visible only when the program is run at or above some specific scale or is sensitive to details about how parallel tasks are mapped to compute resources.

CUDA and Heterogeneous Acceleration Architectures

As general-purpose processors have grown more complicated, and physical limits having to do with thermal dissipation have served to limit raw clock speed increases, there has been a growing acceptance of heterogeneous architectures as a new way to move forward. Such architectures attempt to blend the benefits of general-purpose cores that excel at executing a wide range of computation with a set of cores specifically designed to execute more limited types of computations. These special-purpose cores typically focus on performing integer and floating-point calculations on vectors of data and may trade off some of the sophisticated predictive control flow capabilities that are featured on the general-purpose cores.

Pioneers in this area were IBM and Toshiba, with the introduction of the Cell processor, which blended a PowerPC-based general-purpose computing element in a processor package with a set of special vector processors called Synergistic Processing Elements (SPEs). Shortly after that, NVIDIA provided a solution that uses their extremely capable, mass-market Graphics Processing Units (GPUs) in the vector processor in conjunction with traditional mass-market Intel or AMD CPU.

There are several different techniques for taking advantage of GPUs as computational accelerators. In the 1990s, curious users found clever ways to repurpose the OpenGL image processing language to accomplish general-purpose computations. Later, NVIDIA introduced the proprietary CUDA for C and C++ language extensions. AMD and Apple sponsored work resulting in OpenCL, and independent companies like PGI and CAPS introduced techniques like the PGI- accelerated Fortran and CAPS HMPP. Currently, CUDA has a unique position in the market in terms of widespread adoption and a mature, though still rapidly evolving, tool chain and runtime environment.

CUDA: Powerful But Challenging

Programming for CUDA introduces a series of unique challenges that HCP developers need to overcome to successfully harness the power of the GPU and reduce the time that it takes their applications to run and generate results. The challenges include:

- Lack of Abstraction
- Data vs. Task Parallelism
- Data Movement and Multiple Address Spaces

LACK OF ABSTRACTION

CUDA generally exposes quite a bit about the way that the hardware works to the programmer. In this regard, it is often compared to assembly language for parallel programming. Explicit concepts such as thread arrays, discussed below, map directly to the way the hardware groups computational units and local memory.

The memory model requires explicit data movement between the host processor and the device and CUDA makes explicit use of a hierarchy of memory address spaces, each of which obeys different sharing rules. This is more low-level detail than the typical application or scientific programmer has to deal with when programming in C, C++, or Fortran and also raises significant concerns with regard to portability and maintainability of code.

DATA VS. TASK PARALLELISM

The second challenge is that the programming model for CUDA is one of data parallelism rather than task parallelism. When divvying up work across the nodes of a cluster, HPC programmers are used to looking for and exploiting parallelism at a certain level. The amount of data to assign to each node depends on a number of factors including computational speed of the node, the available memory, the bandwidth and latency of the interconnect, and the frequency with which results need to be shared with other processes.

Since processors are fast and network bandwidth is relatively scarce, the balance is typically to put quite a bit of data on each compute node and to move data as infrequently as possible. CUDA invites the programmer to think about parallelism of a completely different order, encouraging the developer to break the problem apart into units that are much smaller, often as small as the computation that might be required to assign a single variable into an array. This is often many orders of magnitude more parallelism than was previously expressed in the code and requires reasoning somewhat differently about the computations themselves.

DATA MOVEMENT AND MULTIPLE ADDRESS SPACES

NVIDIA GPUs are external accelerators attached to the host system via the PCI bus. Each GPU has both its own onboard memory and also features smaller bits of memory attached to each one of the compute elements (see the architecture review below for more detail). While there are now mechanisms to address regions of host memory from device kernels, such access is slower compared to accessing device memory.

CUDA programs therefore use a model in which code running on the host processor prepares and explicitly dispatches work to the GPU, pauses for the GPU to complete that work, then reads the resulting data back from the device. Both the units of code representing computational kernels and the associated data on which those computational kernels will execute are dispatched to the device. The data is moved, in whole or part, to the device over the PCI bus for execution. As results are produced, they need to be moved, just as explicitly, back from the device to the host computer's main memory.

The device has different kinds of instance memory: either local and reserved for specific sets of computing elements, or globally addressable from any of the computing elements on the CUDA device. CUDA makes this distinction explicit and requires that developers designate the location of variables at variable declaration time. A variable's location in this memory hierarchy determines how it can be used, forcing the developer to be cognizant of these issues.

The NVIDIA GPU Architecture and CUDA

The CUDA hardware model has some significant differences from the usual CPU model. NVIDIA GPU has an array of streaming multiprocessors (SMs), each of which provides an execution environment with a bit of local memory and a number of streaming processors (SP). Older hardware, such as the NVIDIA Tesla C2070 have 14 SMs with 32 SPs each and could execute up to 448 operations in parallel. By comparison, newer cards like A100 Tensor core consisted of 108 SMs with 6912 FP32 CUDA Cores per GPU and 432 Tensor Cores per GPU.

Each one of those streaming processors may be working on different data and produce different results but they are not all executing independently. The NVIDIA hardware introduces the concept of a warp, which is a set of 32 threads that execute the same instruction together. Branching within the warp is handled by stalling those threads that are not participating in a branch to be resumed when the shared program counter is again executing instructions that are part of the thread occupying that lane's execution path.

Threads are grouped together into arrays of up to 512 threads (16 warps) that execute on an SM and share a bit of memory. The geometry of the array is configurable based on the code and an individual thread is identified by a unique combination of three coordinate indices. Since each thread array executes on a single SM, CUDA programs will typically try to start up many more than one thread array at a time. CUDA provides a way to address sets of thread arrays using a second set of three coordinate indices that are referred to as the grid coordinates. Threads are grouped together into arrays. Arrays are grouped together in a grid.

The full combination of six-dimensional grid and array coordinates serve to fully specify a unit of work within a discrete CUDA computation.

The TotalView Debugger

TotalView is a highly interactive graphical source code debugger that provides developers working in C, C++, or Fortran with a way to explore and control their programs. Originally designed to debug one of the first distributed memory architectures, the BBN Butterfly, TotalView has been continually enhanced with a focus on multi-process and multithreaded applications.

TotalView is used to develop, troubleshoot, and maintain applications in a wide range of situations. One group uses it to develop Linux-based commercial embedded computing applications consisting of a single process with a hundred or more separate threads that simultaneously interact with a graphical user interface (GUI), sensors, online databases, and network services. Other users troubleshoot sophisticated mathematical models of complex physical systems in astrophysics, geophysics, climate modeling, and other areas. These models typically consist of thousands of separate processes that run on large-scale, high performance computing (HPC) clusters on the top 500 list.

In both use cases, TotalView provides the users with a debugging session in which they can examine in detail any one of the many threads or processes that make up the application, controlling each thread or process singly or as a part of various predefined and user-defined groups, and displaying multiple types of data and state information from across many processes and threads. With runtime-performance-conscious developers now exploring CUDA as a way to accelerate computation, TotalView is also supporting users who are developing and debugging CUDA C/C++ code.

TotalView For CUDA

TotalView has been extended to provide CUDA support in the same manner that it was extended to support others heterogeneous applications. The current version of TotalView builds on and extends the model of processes made up of threads to incorporate CUDA. There are two major differences. First, CUDA threads are heterogeneous. Second, TotalView shows a representative CUDA thread from among all those created as part of running a routine on the GPU.

The GPU thread is shown as part of the process alongside other pthreads or OpenMP threads. Threads created by OpenMP or pthreads in any single process share an executable image, a single memory address space, and other process level resources like file and network handles. This special CUDA thread, on the other hand, executes a different image, on a set of processors with a different instruction set, and in a completely distinct memory address space.

Because of the large number of CUDA threads, and the fact that only a small number of this large set of logical threads are likely to actually be instantiated on the hardware at any given moment in time, the UI has been adapted to display the thread specific data from a user-selectable representative CUDA device thread.

CUDA is heavily used in conjunction with distributed multi-process programming paradigms such as MPI. As such, the CUDA support in TotalView is complimentary and additive with the MPI support in TotalView. Users debugging an MPI+CUDA job see the same view of all their MPI processes running across the cluster and when they focus on any individual process they can see both what is happening on the host processor and what is happening in the CUDA kernels running on the GPU.

EXTENDING THE THREAD MODEL

TotalView has a very powerful model for multi-process and multithread applications. Each program is modeled as a set of processes running on one or more hosts and comprised of one or more threads. TotalView has features that allow the user to examine and control each thread separately. Most programs are comprised of processes that have anywhere from one to a handful of threads. A few ambitious programs may have a hundred or more threads.

The CUDA runtime allows each thread of a UNIX process to dispatch work to be done on the attached GPU device. This work is encapsulated in units of work referred to abstractly as "kernels." The parallel architecture of the GPU allows it to execute hundreds of instances of these kernels, called device threads, at a time. In order to hide latencies, the programmer is encouraged to request the creation of large numbers (many thousands) of such device threads.

The device cannot actually process all of these threads as the same time. Instead, it "streams" them, scheduling them onto the hardware in batches nad running each batch to completion while loading the preconditions for the next batch so that it can immediately run. This streaming technique hides much of the cost of data movement and context creation.

Ideally, each CUDA kernel thread is performing self-contained unit of work with clearly defined inputs and with no side effects or dependencies on anything that is meant to happen within any of the other kernel threads that are being executed at the same time. That allows the device and the device runtime environment a large degree of freedom in scheduling the kernels. If resources allowed, the runtime could create all the device threads ahead of time and run them all simultaneously.

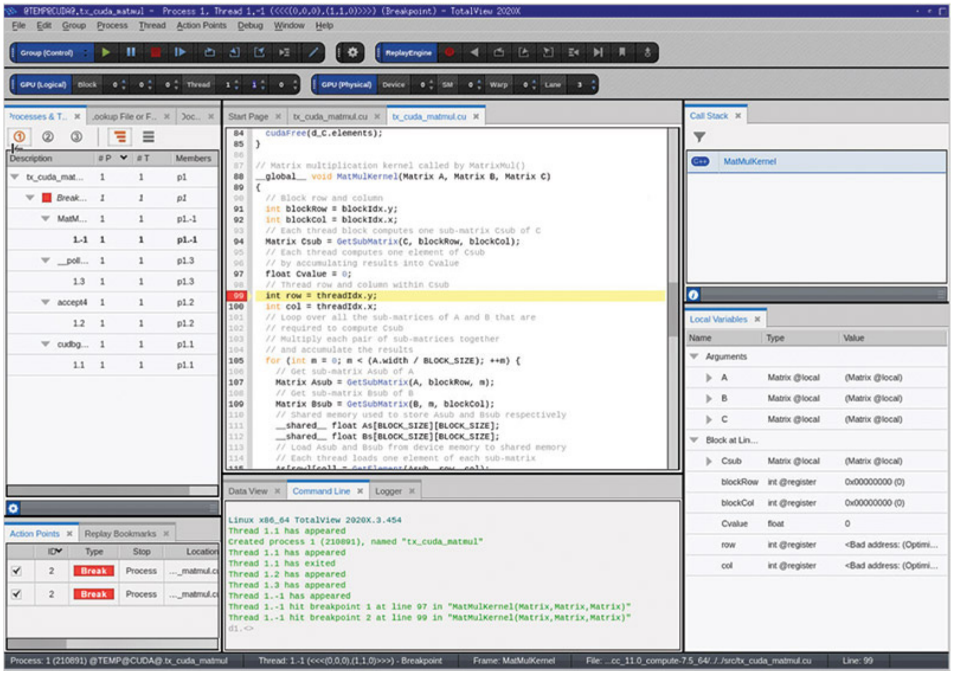
Alternately it could create one thread at a time, running it to completion and then eliminating it (leaving just its output data). Generally, the model allows the runtime to create and run device threads in any order or grouping that make sense based on hardware resources and the data being processed. This device kernel thread concept is very different from host processor threads.

The creation of a thread in pthreads or OpenMP is a significant and heavyweight operation. Best programming and thread runtime practices involve keeping those threads around for a relatively long period of time and using various techniques to have them operate on data structures that are shared across and between threads. The fact that multiple threads access the same data structures is the key feature and key challenge to multithreaded programming with pthreads or OpenMP.

While CUDA kernel threads are encouraged to be self-contained, they are not forced to be. They can access (both reading from and writing to) global and shared memory. This is important in that it avoids having the CUDA program duplicate certain input data structures, but also means that there is the possibility for race conditions and device threads reading incorrect or invalid data.

In order to give visibility into the dynamically changing set of CUDA kernel threads without overwhelming the user, TotalView creates a single thread object for each kernel invocation, called a GPU focus thread, which is displayed next to the host threads in the debugger's display. The user selects this thread to get a view into one of the device threads that are currently executing the kernel on the GPU device. TotalView gives host threads a positive debugger thread ID and CUDA threads a negative thread ID. The initial host thread in process "1" might be labeled "1.1" and the CUDA thread is labeled "1.-1". See Figure 1 for just such a pair of threads.

At any time, the user can control which of the currently active CUDA threads is displayed through the GPU focus thread. This selection is done using a new set of controls, called the GPU thread selector, on the TotalView Process Window. The GPU thread selector reports the device thread currently being displayed and can be used to change that selection. CUDA threads can be specified using one of two coordinate spaces. One is the logical coordinate space that is in CUDA terms of grid and block indices: <<<(Bx,By,Bz),(Tx,Ty,Tz)>>>. The other is the physical coordinate space that is in hardware terms of device number, streaming multiprocessor (SM) number on the device, warp (WP) number on the SM, and lane (LN) number on the warp. See the GPU thread selector in Figure 1.



```
@TEMP@CUDA@.tx_cuda_matmul - Process 1, Thread 1.-1 (<<<(0,0,0),(1,1,0)>>>) (Breakpoint) - TotalView 2020X
```

Description	# P	# T	Members
tx_cuda_mat...	1	1	p1
Break...	1	1	p1
MatM...	1	1	p1.-1
1.-1	1	1	p1.-1
__poll...	1	1	p1.3
1.3	1	1	p1.3
accept4	1	1	p1.2
1.2	1	1	p1.2
cudbg...	1	1	p1.1
1.1	1	1	p1.1

```
  cudaFree(d_C.elements);
}

// Matrix multiplication kernel called by MatrixMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
  // Block row and column
  int blockRow = blockIdx.y;
  int blockCol = blockIdx.x;
  // Each thread block computes one sub-matrix Csub of C
  Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
  // Each thread computes one element of Csub
  // by accumulating results into Cvalue
  float Cvalue = 0;
  // Thread row and column within Csub
  int row = threadIdx.y;
  int col = threadIdx.x;
  // Loop over all the sub-matrices of A and B that are
  // required to compute Csub
  // Multiply each pair of sub-matrices together
  // and accumulate the results
  for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
    // Get sub-matrix Asub of A
    Matrix Asub = GetSubMatrix(A, blockRow, m);
    // Get sub-matrix Bsub of B
    Matrix Bsub = GetSubMatrix(B, m, blockCol);
    // Shared memory used to store Asub and Bsub respectively
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    // Load Asub and Bsub from device memory to shared memory
    // Each thread loads one element of each sub-matrix
```

	ID	Type	Stop	Location
☑	2	Break	Process	..._matmul.cu
☑	2	Break	Process	..._matmul.cu

```
Linux x86_64 TotalView 2020X.3.454
Thread 1.1 has appeared
Created process 1 (210891), named "tx_cuda_matmul"
Thread 1.1 has appeared
Thread 1.1 has exited
Thread 1.2 has appeared
Thread 1.3 has appeared
Thread 1.-1 has appeared
Thread 1.-1 hit breakpoint 1 at line 97 in "MatMulKernel(Matrix,Matrix,Matrix)"
Thread 1.-1 hit breakpoint 2 at line 99 in "MatMulKernel(Matrix,Matrix,Matrix)"
d1.<>
```

Local Variables

Name	Type	Value
Arguments		
A	Matrix @local	(Matrix @local)
B	Matrix @local	(Matrix @local)
C	Matrix @local	(Matrix @local)
Block at Lin...		
Csub	Matrix @local	(Matrix @local)
blockRow	int @register	0x00000000 (0)
blockCol	int @register	0x00000000 (0)
Cvalue	float	0
row	int @register	<Bad address: (Optimi...
col	int @register	<Bad address: (Optimi...

Process: 1 (210891) @TEMP@CUDA@.tx_cuda_matmul | Thread: 1.-1 (<<<(0,0,0),(1,1,0)>>>) - Breakpoint | Frame: MatMulKernel | File: ...cc_11.0_compute-7.5_64/./../src/tx_cuda_matmul.cu | Line: 99

Figure 1. TotalView focused on a CUDA device thread at breakpoint. Note the CUDA thread selector between the toolbar with the stepping controls and the process status ribbon; the CUDA-specific information in the stack frame, including the qualifiers such as global; and the CUDA device thread 1.-1 and the host thread 1.1 listed in the Threads Pane on the right.

Any given thread has both a thread index in this 4D physical coordinate space, and a different thread index in the 6D logical coordinate space. These indices are shown in a series of spin boxes on the top.

DEALING WITH DIFFERENT MEMORY SPACES

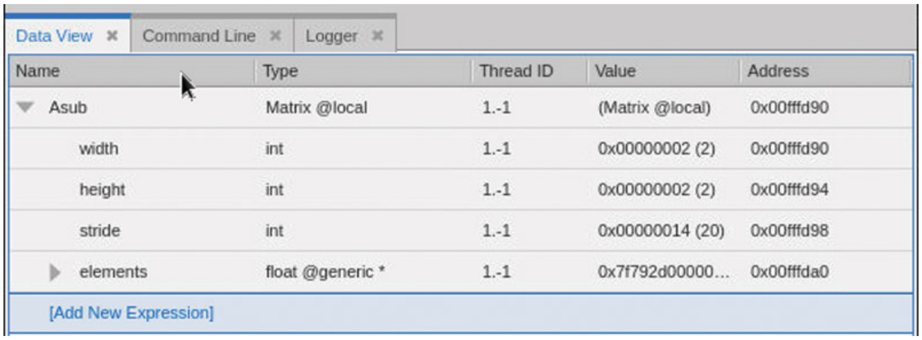
TotalView can display variables and data from a CUDA thread. CUDA variables for a device kernel function are displayed in the Local Variables Window in the same manner that variables are displayed in C, C++, or Fortran code on the host processor. The idioms of hovering over variables in Source Windows with the mouse or diving on variables to add it to Data View Window both works just like they do on the host thread. One thing to note is that when you use the CUDA thread selector to change the focus you will see the values of variables in the newly selected thread replace those from the previous thread in all the aforementioned places.

Naturally, all variables have addresses, but depending on the variable's type, those addresses may refer to any one of many different places. That is because CUDA uses an explicitly hierarchical memory. Each thread has its own local memory; so if variable X is a local variable and it has address 0x14 each thread can have its own instance of X, all with the same address but each referring to a private instance of that variable with a different value.

Similarly, there is a bit of memory associated with each Cooperative Thread Array (CTA) and the streaming multiprocessor that the CTA is running on. Variables that reference into that memory refer to data that is shared by all the threads in the CTA. Finally, there are variables that refer into the global memory that is addressable by and shared by all the threads running on the device.

TotalView expresses these different kinds of memory (register or local, shared, constant or global memory) through the notion of a storage qualifier in the type system. When looking at variables in CUDA these qualifiers appear next to familiar types such as "int" or "float" so that a local variable X expressing an integer would have type of "int @

local” in the TotalView Variable Window. Like types, these qualifiers are something that the user can modify. Casting a variable from “int @local” to “int @global” won’t change the pointer but it will use that pointer to look in a different place (the same address in a different memory address space) and will likely refer to a different value. These type qualifiers can be seen both in Figure 1 above and in Figure 2 below.



Name	Type	Thread ID	Value	Address
Asub	Matrix @local	1.-1	(Matrix @local)	0x00fffd90
width	int	1.-1	0x00000002 (2)	0x00fffd90
height	int	1.-1	0x00000002 (2)	0x00fffd94
stride	int	1.-1	0x00000014 (20)	0x00fffd98
elements	float @generic *	1.-1	0x7f792d000...	0x00fffda0
[Add New Expression]				

Figure 2. Data View Window displaying a variable called Asub that is at 0x00fffb90 in the thread local memory. It has a type of Matrix and it contains a pointer named elements that has a type of float* and points to an address in global memory.

DEALING WITH INLINED FUNCTIONS

The stack trace pane shows the stack backtrace and may include synthetic stack references for inlined functions. Each stack frame in the stack backtrace represents either the PC location of GPU kernel code, or the expansion of an inlined function. Inlined functions can be nested. The “return PC” of an inlined function is the address of the first instruction following the inline expansion, which is normally within the function containing the inlined-function expansion.

THREAD CONTROL AND SINGLE STEPPING THE GPU

TotalView allows you to single-step GPU code just like normal host code, but it is important to note that a single-step operation steps the entire warp associated with the GPU focus thread. That is because at the hardware level, all the threads in the warp share a single program counter. It is literally not possible for the GPU to understand the concept of having some of the threads in the warp at location A and the other threads at location B.

In some senses, this seems like a limitation on the freedom TotalView gives developers to arbitrarily control the execution of host threads. However, the purpose of that control is to allow the developer to explore alternate possible execution scenarios. Since it is not possible for the threads in the warp to get out of step with one another there is no need to exercise scenarios other than those with synchronized threads across the warp. Different warps, however, might examine or modify global memory and might run with different timing, so it is both valid and sometimes necessary to explore different sequences of execution between distinct warps.

CUDA MEMORY EXCEPTIONS

The NVIDIA hardware lacks some of the memory protection logic that developers are likely accustomed to when writing code for the host processor, so mistakes like de-referencing an uninitialized pointer in CUDA code won’t result in a segmentation violation, and the program will proceed with undefined consequences. However, NVIDIA provides a mechanism for emulating the memory protection in software; that mechanism can be engaged in TotalView Classic by activating the preference “Enable CUDA Memory Checking.”

With CUDA memory checking activated you can run the device kernel; when an invalid pointer operation occurs, the debugger will stop the thread with an error message as it does when a host thread encounters a segmentation error. Note that CUDA memory checking is not active when single stepping the CUDA thread.

Conclusion

This paper has focused on two main topics. First, it briefly reviewed CUDA and some of the challenges that CUDA introduces for HPC developers looking to harness GPU processing power to accelerate their applications. These challenges center on the conceptual transitions related to moving to (or more often layering in) fine-grained data parallelism from (or alongside) medium-grained task parallelism, but also extend to dealing with many low-level details about the NVIDIA GPU device architecture that are exposed to the CUDA programmer as additional language concepts. These features and challenges, while certainly not insurmountable, definitely complicate the picture. Increased complexity potentially hinders troubleshooting, validation, and ongoing software maintenance.

The second section of the paper focuses on changes that have been made to adapt the TotalView parallel debugger for CUDA. The paper shows how the TotalView process and thread model has been extended to allow users to easily switch back and forth between what is happening on the host processors, where MPI-level communication occurs and the program sets the stage for the data parallel work to happen on the GPU, and what is happening over on the GPU itself where lightweight threads are being created and destroyed at a furious pace, each one in existence only long enough to compute one data element. It also details how the TotalView variable display and type system have been modified to provide developers a clear understanding of how their data interacts with the various distinct memory address spaces, each with different characteristics and visibility.

Get a free trial of TotalView and see how you can dramatically simplify and accelerate HPC debugging.

TRY FREE

totalview.io/free-trial

About Perforce

Perforce powers innovation at unrivaled scale. With a portfolio of scalable DevOps solutions, we help modern enterprises overcome complex product development challenges by improving productivity, visibility, and security throughout the product lifecycle. Our portfolio includes solutions for Agile planning & ALM, API management, automated mobile & web testing, embeddable analytics, open source support, repository management, static & dynamic code analysis, version control, and more. With over 9,000 customers, Perforce is trusted by the world's leading brands, including NVIDIA, Pixar, Scania, Ubisoft, and VMware. For more information, visit www.perforce.com.