CUDA, Supercomputing for the Masses: Part 1

CUDA lets you work with familiar programming concepts while developing software that can run on a GPU

By Rob Farber, Dr. Dobb's Journal
Apr 15, 2008
URL: http://www.ddj.com/hpc-high-performance-computing/207200659

Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com.

Are you interested in getting orders-of-magnitude performance increases over standard multi-core processors, while programming with a high-level language such as C? And would you like that capability to scale across many devices as well?

Many people (myself included) have achieved this level of performance and scalability on non-trivial problems by using CUDA (short for "Compute Unified Device Architecture") from NVIDIA to program inexpensive multi-threaded GPUs. I purposefully stress "programming" because CUDA is an architecture designed to let you do your work, rather than forcing your work to fit within a limited set of performance libraries. With CUDA, you get to exploit your abilities to design software to achieve best performance on your multi-threaded hardware -- and have fun as well because figuring out the right mapping is captivating, plus the software development environment is both reasonable and straightforward.

This is the first of a series of articles to introduce you to the power of CUDA -- through working code -- and to the thought process to help you map applications onto multi-threaded hardware (such as GPUs) to get big performance increases. Of course, not all problems can be mapped efficiently onto multi-threaded hardware, so part of my thought process will be to distinguish what will and what won't work, plus provide a common-sense idea of what might work "well-enough".

"CUDA programming" and "GPGPU programming" are not the same (although CUDA runs on GPUs). Previously, writing software for a GPU meant programming in the language of the GPU. An acquaintance of mine once described this as a process similar to pulling data out of your elbow to get it to where you could look at it with your eyes. CUDA permits working with familiar programming concepts while developing software that can run on a GPU. It also avoids the performance overhead of graphics layer APIs by compiling your software directly to the hardware (GPU assembly language, for instance), thereby providing great performance.

The choice of CUDA device is up to you. Figures 1 and 2 show the CUDA N-body simulation program running on both a laptop and a discrete GPU based desktop PC.
Figure 1: nBody Astrophysics Simulation running on a Quadro FX 570M enabled laptop.
Can CUDA really increase application performance by one to two orders of magnitude -- or is all this hype rather than reality?

CUDA is a fairly new technology but there are already many examples in the literature and on the Internet highlighting significant performance boosts using current commodity GPU hardware. Tables 1 and 2 show summaries posted on the NVIDIA and Beckman Institute websites. At the heart of CUDA is the ability for programmers to keep thousands of threads busy. The current generation of NVIDIA GPUs can efficiently support a very large number of threads, and as a result they can deliver one to two orders of magnitude performance increase in application performance. These graphics processors are widely available to anyone at almost any price point. Newer boards will expand CUDA's capabilities by providing greater memory bandwidth, asynchronous data transfer, atomic operations, and double-precision floating point arithmetic among many hardware improvements. Look for the CUDA software environment to expand as the technology evolves and we eventually lose the distinction between GPUs and "many-core" processors. As developers, we have to anticipate that applications with many thousands of active threads will become common-place and look for CUDA to run on many platforms, including general-purpose processors.

<table>
<thead>
<tr>
<th>Example Applications</th>
<th>URL</th>
<th>Application Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seismic Database</td>
<td><a href="http://www.headwave.com">http://www.headwave.com</a></td>
<td>66x to 100x</td>
</tr>
<tr>
<td>Mobile Phone Antenna Simulation</td>
<td><a href="http://www.acceleware.com">http://www.acceleware.com</a></td>
<td>45x</td>
</tr>
<tr>
<td>Molecular Dynamics</td>
<td><a href="http://www.ks.uiuc.edu/Research/vmd">http://www.ks.uiuc.edu/Research/vmd</a></td>
<td>21x to 100x</td>
</tr>
<tr>
<td>Neuron Simulation</td>
<td><a href="http://www.evolvedmachines.com">http://www.evolvedmachines.com</a></td>
<td>100x</td>
</tr>
</tbody>
</table>
### Table 1: NVIDIA summary from www.nvidia.com/object/IO_43499.html

<table>
<thead>
<tr>
<th>Calculation / Algorithm</th>
<th>Algorithm class</th>
<th>Speedup vs. Intel QX6700 CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fluorescence microphotolysis</td>
<td>Iterative matrix / stencil</td>
<td>12x</td>
</tr>
<tr>
<td>Pairlist calculation</td>
<td>Particle pair distance test</td>
<td>10x to 11x</td>
</tr>
<tr>
<td>Pairlist update</td>
<td>Particle pair distance test</td>
<td>5x to 15x</td>
</tr>
<tr>
<td>Molecular dynamics nonbonded force calculation</td>
<td>N-body cutoff force calculations</td>
<td>10x to 20x</td>
</tr>
<tr>
<td>Cutoff electron density sum</td>
<td>Particle-grid w/ cutoff</td>
<td>15x to 23x</td>
</tr>
<tr>
<td>Cutoff potential summation</td>
<td>Particle-grid w/ cutoff</td>
<td>12x to 21x</td>
</tr>
<tr>
<td>Direct Coulomb summation</td>
<td>Particle-grid</td>
<td>44x</td>
</tr>
</tbody>
</table>

### Table 2: Beckman Institute table from www.ks.uiuc.edu/Research/vmd/publications/siam2008vmdcuda.pdf

As a scientist at Los Alamos National Laboratory in the 1980s, I had the pleasure of working with the massively parallel 65,536 processor Thinking Machines supercomputers. CUDA has proved to be a natural framework to again start working in a modern massively-parallel (i.e., highly-threaded) environment. Performance is clearly there. One of my production codes, now written in CUDA and running on NVIDIA GPUs, shows both linear scaling and a nearly two orders of magnitude speed increase over a 2.6-Ghz quad-core Opteron system.

CUDA-enabled graphics processors operate as co-processors within the host computer. This means that each GPU is considered to have its own memory and processing elements that are separate from the host computer. To perform useful work, data must be transferred between the memory space of the host computer and CUDA device(s). For this reason, performance results must include IO time to be informative. Colleagues have also referred to these as "Honest Flops" because they more accurately reflect the performance applications will deliver in production.

I claim that a one or two orders of magnitude performance increase over existing technology is a disruptive change that can dramatically alter some aspects of computing. For example, computational tasks that previously would have taken a year can now complete in a few days, hour long computations suddenly become interactive because they be completed in seconds with the new technology, and previously intractable real-time processing tasks now becomes tractable. Finally, lucrative opportunities can present themselves for consultants and engineers with the right skill set and capabilities to write highly-threaded (or massively parallel) software. What about you? How can this type of computing capability benefit your career, applications or real-time processing needs?

Getting started costs nothing and is as easy as downloading CUDA from the CUDA Zone homepage (look for "Get CUDA"). After that, follow the installation instructions for your particular operating system. You don't even need a graphics processor because you can start working right away by using the software emulator to run on your current laptop or workstation. Of course, much better performance will be achieved by running with a CUDA-enabled GPU. Perhaps your computer already has one. Check out the "CUDA-enabled GPUs" link on the CUDA Zone homepage to see. (A CUDA-enabled GPU includes shared on-chip memory and thread management.)
If purchasing a new graphics processor card, I suggest following this article series because I will discuss how various hardware characteristics (such as memory bandwidth, number of registers, atomic operations, and so on) will affect application performance, which will assist you in selecting the appropriate hardware for your application. Also, the CUDA Zone forums provide a wealth of information on all things CUDA, including discussions about what hardware to purchase.

Once installed, the CUDA Toolkit provides a reasonable set of tools for C language application development. This includes:

- The nvcc C compiler
- CUDA FFT and BLAS libraries for the GPU
- A profiler
- An alpha version (as of March 2008) of the gdb debugger for the GPU
- CUDA runtime driver (now also available in the standard NVIDIA GPU driver)
- CUDA programming manual

The nvcc C compiler does most of the work in converting C code into an executable that will run on a GPU or the emulator. Happily, assembly-language programming is not required to achieve high performance. Future articles will discuss working with CUDA from other high-level languages including C++, FORTRAN, and Python. I assume that you're familiar with C/C++. No previous parallel programming or CUDA experience is required. This is consistent with the existing CUDA documentation.

Creating and running a CUDA C language program follows the same workflow as other C programming environments. Explicit build and run instructions for Windows and Linux environments are in the CUDA documentation, but simply stated they are:

1. Create or edit the CUDA program with your favorite editor. Note: CUDA C language programs have the suffix ".cu".
2. Compile the program with nvcc to create the executable. (NVIDIA provides sane makefiles with the examples. Generally all you need to type is "make" to build for a CUDA device or "make emu=1" to build for the emulator.)
3. Run the executable.

Listing One is a simple CUDA program to get you started. It is nothing more than a program that calls the CUDA API to move data to and from the CUDA device. Nothing new is added that might cause confusion in learning how to use the tools to build and run a CUDA program. In the next article, I will discuss what is going on and start using the CUDA device to perform some work.

```
// moveArrays.cu
//
// demonstrates CUDA interface to data allocation on device (GPU)
// and data movement between host (CPU) and device.

#include <stdio.h>
#include <assert.h>
#include <cuda.h>

int main(void)
{
    float *a_h, *b_h;    // pointers to host memory
    float *a_d, *b_d;    // pointers to device memory
    int N = 14;
    int i;
    // allocate arrays on host
    a_h = (float *)malloc(sizeof(float)*N);
    b_h = (float *)malloc(sizeof(float)*N);
    // allocate arrays on device
    cudaMalloc((void **) &a_d, sizeof(float)*N);
    cudaMalloc((void **) &b_d, sizeof(float)*N);
    // initialize host data
    for (i=0; i<N; i++) {
        a_h[i] = 10.f+i;
        b_h[i] = 0.f;
    }
    // send data from host to device: a_h to a_d
    cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);
```
Listing One

Give it a try and play around with the development tools. A quick note to newbies: You can use printf statements to see what is happening on the GPU when running under the emulator (build the executable with make emu=1). Also, feel free to try out the alpha version of the debugger.

For More Information

- CUDA, Supercomputing for the Masses: Part 12
- CUDA, Supercomputing for the Masses: Part 11
- CUDA, Supercomputing for the Masses: Part 10
- CUDA, Supercomputing for the Masses: Part 9
- CUDA, Supercomputing for the Masses: Part 8
- CUDA, Supercomputing for the Masses: Part 7
- CUDA, Supercomputing for the Masses: Part 6
- CUDA, Supercomputing for the Masses: Part 5
- CUDA, Supercomputing for the Masses: Part 4
- CUDA, Supercomputing for the Masses: Part 3
- CUDA, Supercomputing for the Masses: Part 2
- CUDA, Supercomputing for the Masses: Part 1
CUDA, Supercomputing for the Masses: Part 2

A first kernel

By Rob Farber, Dr. Dobb’s Journal
Apr 29, 2008
URL: http://www.ddj.com/hpc-high-performance-computing/207402986

Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com.

In Part 1 of this article series, I presented a simple first CUDA (short for "Compute Unified Device Architecture") program -- moveArrays.cu -- to familiarize you with the CUDA tools for building and executing programs. For C programmers, this program did nothing more than call the CUDA API to allocate memory and move data to and from the CUDA device. Nothing new was added that might cause confusion in learning how to use the tools to build and run a CUDA program.

This article builds on that first example by adding a few additional lines of code to perform a simple calculation on the CUDA device -- specifically incrementing each element in a floating-point array by 1. Amazingly, this example already provides the basic framework ("move data to CUDA-enabled device(s), perform a calculation and retrieve result") for solving many problems with CUDA!

Before tackling more advanced topics, you first need to understand:

- What is a kernel? A kernel is a function callable from the host and executed on the CUDA device -- simultaneously by many threads in parallel.
- How does the host call a kernel? This involves specifying the name of the kernel plus an execution configuration. For the purposes of this column, an execution configuration just means defining the number of parallel threads in a group and the number of groups to use when running the kernel for the CUDA device. This is actually an important topic that will be discussed in greater depth in future columns.
- How to synchronize kernels and host code.

At the top of the Listing One (incrementArrays.cu), we see an example host routine, incrementArrayOnHost and our first kernel, incrementArraysOnDevice.

The host function incrementArrayOnHost is just a simple loop over the number of elements in an array to increment each array element by one. This function is used for comparison purposes at the end of this code to verify the kernel performed the correct calculation on the CUDA device.

Next in the Listing One is our first CUDA kernel, incrementArraysOnDevice. CUDA provides several extensions to the C-language. The function type qualifier __global__ declares a function as being an executable kernel on the CUDA device, which can only be called from the host. All kernels must be declared with a return type of void.

The kernel incrementArraysOnDevice performs the same calculation as incrementArrayOnHost. Looking within incrementArraysOnDevice, you see that there is no loop! This is because the function is
simultaneously executed by an array of threads on the CUDA device. However, each thread is provided with a unique ID that can be used to compute different array indices or make control decisions (such as not doing anything if the thread index exceeds the array size). This makes \texttt{incrementArrayOnDevice} as simple as calculating the unique ID in the register variable, \texttt{idx}, which is then used to uniquely reference each element in the array and increment it by one. Since the number of threads can be larger than the size of the array, \texttt{idx} is first checked against \texttt{N}, an argument passed to the kernel that specifies the number of elements in the array, to see if any work needs to be done.

So how is the kernel called and the execution configuration specified? Well, control flows sequentially through the source code starting at main until the line right after the comment containing the statement \textbf{Part 2 of 2} in Listing One.

```c
// incrementArray.cu
#include <stdio.h>
#include <assert.h>
#include <cuda.h>

void incrementArrayOnHost(float *a, int N)
{
    int i;
    for (i=0; i < N; i++) a[i] = a[i]+1.f;
}

__global__ void incrementArrayOnDevice(float *a, int N)
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx<N) a[idx] = a[idx]+1.f;
}

int main(void)
{
    float *a_h, *b_h;           // pointers to host memory
    float *a_d;                 // pointer to device memory
    int i, N = 10;
    size_t size = N*sizeof(float);
    // allocate arrays on host
    a_h = (float *)malloc(size);
    b_h = (float *)malloc(size);
    // allocate array on device
cudaMalloc((void **) &a_d, size);
    // initialization of host data
    for (i=0; i<N; i++) a_h[i] = (float)i;
    // copy data from host to device
cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);
    // do calculation on host
    incrementArrayOnHost(a_h, N);
    // do calculation on device:
    // Part 1 of 2. Compute execution configuration
    int blockSize = 4;
    int nBlocks = N/blockSize + (N%blockSize == 0?0:1);
    // Part 2 of 2. Call incrementArrayOnDevice kernel
    incrementArrayOnDevice <<< nBlocks, blockSize >>> (a_d, N);
    // Retrieve result from device and store in b_h
    cudaMemcpy(b_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
    // check results
    for (i=0; i<N; i++) assert(a_h[i] == b_h[i]);
    // cleanup
    free(a_h); free(b_h); cudaFree(a_d);
}
```

\textbf{Listing One}: incrementArrays.cu.

This queues the launch of \texttt{incrementArrayOnDevice} on the CUDA-enabled device and illustrates another CUDA addition to the C-language, an asynchronous call to a CUDA kernel. The call specifies the name of the kernel and the execution configuration enclosed between triple angle brackets "<<<" and ">>>>". Notice the two parameters that specify the execution configuration: \texttt{nBlocks} and \texttt{blockSize}, which will be discussed next. Any arguments to the kernel call are provided via a standard C-language argument list for a function delimited in the standard C-language fashion with "(" and ")". In this example, the pointer to the device global memory \texttt{a\_d} (which contains the array elements) and \texttt{N} (the number of array elements) are passed to the kernel.
Since the CUDA device is idle, the kernel immediately starts running based on the execution configuration and according to the function arguments. Meanwhile, the host continues to the next line of code after the kernel launch. At this point, both the CUDA device and host are simultaneously running their separate programs. In the case of incrementArrays.cu, the host immediately calls cudaMemcpy, which waits until all threads have finished on the device (e.g., returned from incrementArrayOnDevice) after which it pulls the modified array back to the host. The program completes after the host system performs a sequential comparison to verify we got the same result on the parallel CUDA device with incrementArrayOnDevice as on the host with the sequential version incrementArrayOnHost.

There are several variables determined at kernel startup through the execution configuration (in this example via the variables nBlocks and blockSize contained between the triple angle brackets "<<<" and ") that are available to any kernel. The thinking behind nBlocks and blockSize is actually quite elegant because it allows the developer to account for hardware limitations without requiring the recompilation of the application -- which is an essential feature for developing commercial software with CUDA.

As I'll examine in future columns, threads within a block have the ability to communicate and synchronize with each other. This is a marvelous software feature that unfortunately costs money from a hardware standpoint. Expect more expensive (and future) devices to support a greater number of threads per block than less expensive (and older) devices. The grid abstraction was created to let developers take into account -- without recompilation -- differing hardware capabilities regardless of price point and age. A grid, in effect, batches together calls to the same kernel for blocks with the same dimensionality and size, and effectively multiplies by a factor of nBlocks the number of threads that can be launched in a single kernel invocation. Less capable devices may only be able to run one or a couple of thread blocks simultaneously, while more capable (e.g., expensive and future) devices may be able to run many at once. Designing software with the grid abstraction requires balancing the trade-offs between simultaneously running many independent threads, and requiring a greater number of threads within a block that can cooperate with each other. Please be cognizant of the costs associated with the two types of threads. Of course, different algorithms will impose different requirements, but when possible try to use larger numbers of thread blocks.

In the kernel on the CUDA-enabled device, several built-in variables are available that were set by the execution configuration of the kernel invocation. They are:

- blockIdx which contains the block index within the grid.
- threadIdx contains the thread index within the block.
- blockDim contains the number of threads in a block.

These variables are structures that contain integer components of the variables. Blocks, for example, have x-, y-, and z- integer components because they are three-dimensional. Grids, on the other hand, only have x- and y-components because they are two-dimensional. This example only uses the x-component of these variables as the array we moved onto the CUDA device is one-dimensional. (Future columns will explore the power of this two-dimensional and three-dimensional configuration capability and how it can be exploited.)

Our example kernel used these built-in variables them to calculate the thread index, idx with the statement:

```c
int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

The variables nBlocks and blockSize are the number of blocks in the grid and the number of threads in each block, respectively. In this example, they are initialized just before the kernel call in the host code:

```c
int blockSize = 4;
int nBlocks = N/blockSize + (N%blockSize == 0?0:1);
```

In cases where N is not evenly divisible by blockSize, the last term in the nBlocks calculation adds an extra block, which for some cases implies some threads in the block will not perform any useful work.

This example is obviously contrived for simplicity as it assumes that the array size is smaller than the number of threads that can be contained within four (4) thread blocks. This is an obvious oversimplification but it let us, with simple code, explore the kernel call to incrementArrayOnDevice.

It is important to emphasize that each thread is capable of accessing the entire array a_d on the device.
There is no inherent data partitioning when a kernel is launched. It is up to the programmer to identify and exploit the data parallel aspects of the computation when writing kernels.

Figure 1 illustrates how \( \text{idx} \) is calculated and the array, \( \text{a} \_ \text{d} \) is referenced. (If any of the preceding text is unclear, I recommend adding a \texttt{printf} statement to \texttt{incrementArrayOnDevice} to print out \( \text{idx} \) and the associated variables used to calculate it. Compile the program for the emulator, "make emu=1", and run it to see what is going on. Be certain to specify the correct path to the emulator executable to see the \texttt{printf} output.)

![Diagram of kernel launch and data partitioning](image)

**Figure 1**

Again, kernel calls are asynchronous -- after a kernel launch, control immediately returns to the host CPU. The kernel will run on the CUDA device once all previous CUDA calls have finished. The asynchronous kernel call is a wonderful way to overlap computation on the host and device. In this example, the call to \texttt{incrementArrayOnHost} could be placed after the call to \texttt{incrementArrayOnDevice} to overlap computation on the host and device to get better performance. Depending on the amount of time the kernel takes to complete, it is possible for both host and device to compute simultaneously.

Until the next column, I recommend:

- Try changing the value of \( \text{N} \) and \( \text{nBlocks} \). See what happens when they exceed the device capabilities.
- Think about how to introduce a loop to handle arbitrary sized arrays.
- Distinguish between the types of CUDA-enabled device memory (e.g., global memory, registers, shared memory, and constant memory). Take a look at the CUDA occupancy calculator and either the \texttt{nvcc} options \texttt{-cubin} or \texttt{--ptxas-options=-v} to determine the number of registers used in a kernel.

**For More Information**

- CUDA, Supercomputing for the Masses: Part 8
- CUDA, Supercomputing for the Masses: Part 7
- CUDA, Supercomputing for the Masses: Part 6
- CUDA, Supercomputing for the Masses: Part 5
- CUDA, Supercomputing for the Masses: Part 4
- CUDA, Supercomputing for the Masses: Part 3
- CUDA, Supercomputing for the Masses: Part 2
- CUDA, Supercomputing for the Masses: Part 1
Click here for more information on CUDA and here for more information on NVIDIA.
CUDA, Supercomputing for the Masses: Part 3

Error handling and global memory performance limitations

By Rob Farber, Dr. Dobb's Journal
May 13, 2008
URL: http://www.ddj.com/hpc-high-performance-computing/207603131

Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com.

Congratulations! Thanks to Part 1 and Part 2 of this series on CUDA (short for "Compute Unified Device Architecture"), you are now a CUDA-enabled programmer with the ability to create and run programs that can use many hundreds of simultaneous threads on CUDA-enabled devices. In Part 2, incrementArrays.cu, I provided a working example of a pervasive CUDA application pattern -- move data to device, run one or more kernels to perform a calculation, and retrieve the result(s). Essentially, incrementArrays.cu can be morphed into whatever application you desire simply by substituting your own kernel and loading your own data (which is what I do for the example in this column). Subsequent columns will discuss CUDA asynchronous I/O and streams.

"You now know enough to be dangerous!" is a humorous and accurate way to summarize the previous paragraph. The good news about CUDA is that it provides a natural way to translate your thoughts as a programmer into massively-parallel programs. The bad news is that more understanding is required to make those programs both robust and efficient.

Don't be cautious. Start experimenting and go for it! CUDA provides both programming tools and constructs to create excellent software and the only way to really learn it is to experiment. In reality, these columns will supplement your experimentation and learning process by highlighting CUDA features through short examples and by bringing good sources of information on the Internet to your attention. Remember that the CUDA Zone is a great place for all things CUDA and the forums are a great place to look for answers to your questions, plus they have the advantage of being interactive so you can post questions and get answers.

This and the next few columns utilize a simple array reversal application to expand your knowledge and highlight the performance impact of shared memory. I discuss error checks and performance behavior, along with the CUDA profiling tool. I've also included the source listing for the next column so you can see how to implement array reversal with shared memory. The program reverseArray_multiblock.cu implements an obvious, yet low performance, way to reverse an array in global memory on a CUDA device. Do not use it as a model for your applications because global memory is not the best memory type to use for this application -- plus this version also performs uncoalesced memory accesses, which adversely affects global memory performance. The best global memory bandwidth is achieved when simultaneous memory accesses can be coalesced into a single memory transaction. In subsequent columns, I discuss the differences between global and shared memory as well as the various requirements for memory accesses to coalesce based on the compute capability of the device.

CUDA Error Handling

file:///Users/tkaiser/Desktop/dobbs/part3.html
Detecting and handling errors is essential to creating robust and usable software. Users tend to get very grumpy when their applications fail or produce incorrect results. For developers, adding error-handling code can be annoying, and tedious. It can clutter up the elegance of the code, and slow the development process in attempting to deal with every conceivable error. Yes, error handling is a thankless job but keep in mind you are not doing it for yourself (although I have been saved countless times through good error checking) - - rather it is being done for the people who are going to use the program. If something can fail, users need to know why it failed and, more importantly, what they can do to fix the problem. Enough said, good error-handling and recovery can really make your application a hit with the users. Commercial developers should especially take note.

The CUDA designers are aware of the importance of good error handling. To facilitate this, every CUDA call -- with the exception of kernel launches -- returns an error code of type `cudaError_t`. Upon successful completion, `cudaSuccess` is returned. Otherwise, an error code is returned.

A human-readable description of the error can be obtained from:

```c
char *cudaGetErrorString(cudaError_t code);
```

C-language programmers will recognize a similarity between this method and the C library, which uses the variable `errno` to indicate errors and the reporting of human-readable error messages with `perror` and `strerror`. The C library paradigm has worked well for many millions of lines of C-code, and there is no doubt it should work well in the future for CUDA software.

CUDA also provides a method, `cudaGetLastError`, which reports the last error for any previous runtime call in the host thread. This has multiple implications:

- The asynchronous nature of the kernel launches precludes explicitly checking for errors with `cudaGetLastError`. Instead, use `cudaThreadSynchronize` which blocks until the device has completed all previous calls, including kernel calls, and returns an error if one of the preceding tasks fails. Queuing multiple kernel launches unfortunately implies that error checking can only be done after all the kernels have completed -- unless explicit error checking and reporting to the host is performed by programmers within the kernel.
- Errors are reported to the correct host thread. If the host is running multiple threads, as might be the case when an application is using multiple CUDA devices, the error will be reported to the correct host thread.
- When multiple errors occur between calls to `cudaGetLastError`, only the last error will be reported. This means the programmer must take care to tie the error to the runtime call that generated the error or risk making an incorrect error report to users.

### Looking at the Source Code

Looking at the source code for `reverseArray_multiblock.cu`, you notice that the structure of the program is very, very similar to the structure of `moveArrays.cu` from Part 2. An error routine, `checkCUDAAError` is provided so the host can print out a human-readable message and exit when an error is reported by `cudaGetLastError`. As can be seen, `checkCUDAAError` is judiciously utilized throughout the program to check for errors.

The program `reverseArray_multiblock.cu` essentially creates a 1D array of integers, `h_a`, containing the integer values `[0 .. dimA-1]`. Array `h_a` is moved via `cudaMemcpy` to array `d_a`, which resides in global memory on the device. The host then launches the `reverseArrayBlock` kernel to copy the array contents in reverse order from `d_a` to `d_b`, which is another global memory array. Again, `cudaMemcpy` is used to transfer data -- this time from `d_b` to the host. A check is then performed on the host to verify that the device produced the correct result (e.g., `[dimA-1 .. 0]`).

```c
// includes, system
#include <stdio.h>
#include <assert.h>

// Simple utility function to check for CUDA runtime errors
void checkCUDAAError(const char* msg);``
// Part 3: implement the kernel
__global__ void reverseArrayBlock(int *d_out, int *d_in)
{
    int inOffset = blockDim.x * blockIdx.x;
    int outOffset = blockDim.x * (gridDim.x - 1 - blockIdx.x);
    int in = inOffset + threadIdx.x;
    int out = outOffset + (blockDim.x - 1 - threadIdx.x);
    d_out[out] = d_in[in];
}

/////////////////////////////////////////////////////////////////////
// Program main
/////////////////////////////////////////////////////////////////////
int main( int argc, char** argv)
{
    // pointer for host memory and size
    int *h_a;
    int dimA = 256 * 1024; // 256K elements (1MB total)

    // pointer for device memory
    int *d_b, *d_a;

    // define grid and block size
    int numThreadsPerBlock = 256;

    // Part 1: compute number of blocks needed based on
    // array size and desired block size
    int numBlocks = dimA / numThreadsPerBlock;

    // allocate host and device memory
    size_t memSize = numBlocks * numThreadsPerBlock * sizeof(int);
    h_a = (int *) malloc(memSize);
    cudaMalloc( (void **) &d_a, memSize );
    cudaMalloc( (void **) &d_b, memSize );

    // Initialize input array on host
    for (int i = 0; i < dimA; ++i)
    {
        h_a[i] = i;
    }

    // Copy host array to device array
    cudaMemcpy( d_a, h_a, memSize, cudaMemcpyHostToDevice );

    // launch kernel
    dim3 dimGrid(numBlocks);
    dim3 dimBlock(numThreadsPerBlock);
    reverseArrayBlock<<< dimGrid, dimBlock >>>( d_b, d_a );

    // block until the device has completed
    cudaThreadSynchronize();

    // check if kernel execution generated an error
    // Check for any CUDA errors
    checkCUDAError("kernel invocation");

    // device to host copy
    cudaMemcpy( h_a, d_b, memSize, cudaMemcpyDeviceToHost );

    // Check for any CUDA errors
    checkCUDAError("memcpy");
// verify the data returned to the host is correct
for (int i = 0; i < dimA; i++)
{
    assert(h_a[i] == dimA - 1 - i);
}

// free device memory
cudaFree(d_a);
cudaFree(d_b);

// free host memory
free(h_a);

// If the program makes it this far, then the results are
// correct and there are no run-time errors. Good work!
printf("Correct!\n");
return 0;
}

void checkCUDAError(const char *msg)
{
    cudaError_t err = cudaGetLastError();
    if( cudaSuccess != err)
    {
        fprintf(stderr, "Cuda error: %s: %s.\n", msg,
                    cudaGetErrorString( err ) );
        exit(EXIT_FAILURE);
    }
}

A key design feature of this program is that both arrays d_a and d_b reside in global memory on the device.
The CUDA SDK provides an example program, bandwidthTest, which provides some information about the
device characteristics. On my system, the global memory bandwidth is slightly over 60 GB/s. This is
excellent until you consider that this bandwidth must service 128 hardware threads -- each of which can
deliver a large number of floating-point operations. Since a 32-bit floating-point value occupies four (4)
bytes, global memory bandwidth limited applications on this hardware will only be able to deliver around 15
GF/s -- or only a small percentage of the available performance capability. (This assumes the application
only reads from global memory and does not write to it.) Obviously, higher performance applications must
reuse data in some fashion. This is the function of shared and register memory and it is our job as
programmers to gain the maximum benefit of these memory types. To gain a better understanding of
machine balance as floating-point capability relates to memory bandwidth (and other machine
characteristics), read my article [HPC Balance and Common Sense](#).

Shared Memory Version

The following source listing is for arrayReversal_multiblock_fast.cu, which I discuss in the next installment.
I provide it now for your convenience so you can see how to use shared memory on this problem right now.

// includes, system
#include <stdio.h>
#include <assert.h>

// Simple utility function to check for CUDA runtime errors
void checkCUDAError(const char *msg);

// Part 2 of 2: implement the fast kernel using shared memory
__global__ void reverseArrayBlock(int *d_out, int *d_in)
{
    extern __shared__ int s_data[];

    int inOffset = blockDim.x * blockIdx.x;

int in  = inOffset + threadIdx.x;

// Load one element per thread from device memory and store it
// *in reversed order* into temporary shared memory
s_data[blockDim.x - 1 - threadIdx.x] = d_in[in];

// Block until all threads in the block have
// written their data to shared mem
__syncthreads();

// write the data from shared memory in forward order,
// but to the reversed block offset as before

int outOffset = blockDim.x * (gridDim.x - 1 - blockIdx.x);

int out = outOffset + threadIdx.x;
d_out[out] = s_data[threadIdx.x];
}

/////////////////////////////////////////////////////////////////////
// Program main
/////////////////////////////////////////////////////////////////////
int main( int argc, char** argv)
{
    // pointer for host memory and size
    int *h_a;
    int dimA = 256 * 1024; // 256K elements (1MB total)

    // pointer for device memory
    int *d_b, *d_a;

    // define grid and block size
    int numThreadsPerBlock = 256;

    // Compute number of blocks needed based on array size
    // and desired block size
    int numBlocks = dimA / numThreadsPerBlock;

    // Part 1 of 2: Compute number of bytes of shared memory needed
    // This is used in the kernel invocation below
    int sharedMemSize = numThreadsPerBlock * sizeof(int);

    // allocate host and device memory
    size_t memSize = numBlocks * numThreadsPerBlock * sizeof(int);
    h_a = (int *) malloc(memSize);
    cudaMalloc( (void **) &d_a, memSize );
    cudaMalloc( (void **) &d_b, memSize );

    // Initialize input array on host
    for (int i = 0; i < dimA; ++i)
    {
        h_a[i] = i;
    }

    // Copy host array to device array
    cudaMemcpy( d_a, h_a, memSize, cudaMemcpyHostToDevice );

    // launch kernel
    dim3 dimGrid(numBlocks);
    dim3 dimBlock(numThreadsPerBlock);
    reverseArrayBlock<<< dimGrid, dimBlock,
sharedMemSize >>= (d_b, d_a);

// block until the device has completed
cudaThreadSynchronize();

// check if kernel execution generated an error
// Check for any CUDA errors
checkCUDAError("kernel invocation");

// device to host copy
cudaMemcpy( h_a, d_b, memSize, cudaMemcpyDeviceToHost );

// Check for any CUDA errors
checkCUDAError("memcpy");

// verify the data returned to the host is correct
for (int i = 0; i < dimA; i++)
{
   assert(h_a[i] == dimA - 1 - i);
}

// free device memory
cudaFree(d_a);
cudaFree(d_b);

// free host memory
free(h_a);

// If the program makes it this far, then results are correct and
// there are no run-time errors. Good work!
printf("Correct!\n");
return 0;

} 
void checkCUDAError(const char *msg)
{
 cudaError_t err = cudaGetLastError();
 if( cudaSuccess != err )
 {
    fprintf(stderr, "CUDA error: %s: %s.\n", msg,
             cudaGetErrorString( err ));
    exit(EXIT_FAILURE);
 }
}

In the next column, I begin looking at the use of shared memory to increase performance. Until then, I suggest looking into the CUDA memory types -- specifically __shared__, __constant__, and register memory.

For More Information

- CUDA, Supercomputing for the Masses: Part 1
- CUDA, Supercomputing for the Masses: Part 10
- CUDA, Supercomputing for the Masses: Part 9
- CUDA, Supercomputing for the Masses: Part 8
- CUDA, Supercomputing for the Masses: Part 7
- CUDA, Supercomputing for the Masses: Part 6
- CUDA, Supercomputing for the Masses: Part 5
- CUDA, Supercomputing for the Masses: Part 4
- CUDA, Supercomputing for the Masses: Part 3
- CUDA, Supercomputing for the Masses: Part 2
CUDA, Supercomputing for the Masses: Part 1

Click here for more information on CUDA and here for more information on NVIDIA.
CUDA, Supercomputing for the Masses: Part 4

Understanding and using shared memory (1)

By Rob Farber, Dr. Dobb's Journal
Jun 03, 2008
URL: http://www.ddj.com/architect/208401741

Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com.

One of the most important performance challenges facing CUDA (short for "Compute Unified Device Architecture") developers is the best use of local multiprocessor memory resources such as shared memory, constant memory, and registers. The reason discussed in Part 3 of this series is that while global memory can deliver over 60GB/s, this would translate to only 15GF/s for single-touch use of data -- getting higher performance requires local data reuse. The CUDA software and hardware designers have done some wonderful work to hide global memory latency and global memory bandwidth restrictions -- so long as there is some local data reuse.

Recall from Part 2 that a kernel launch requires the specification of an execution configuration to define the number of threads that compose a block and the number of blocks that are combined together to form a grid. It is important to note that threads within a block can communicate with each other through local multiprocessor resources because the CUDA execution model specifies that a block can only be processed on a single multi-processor. In other words, data written to shared memory within a block is accessible to all other threads within that block, but it is not accessible to a thread from any other block. Shared memory with these characteristics can be implemented very efficiently in hardware which translates to fast memory accesses (with some caveats discussed shortly) for CUDA developers.

Now we have a way for the CUDA-enabled hardware designers to balance price versus the needs of the CUDA software developers. As developers, we want large amounts of local multiprocessor resources such as registers and shared memory. It makes our jobs much easier and our software more efficient. The hardware designer, on the other hand, needs to deliver hardware at a low price-point and unfortunately fast local multi-processor memory is expensive. We all agree that inexpensive CUDA hardware is wonderful, so CUDA-enabled hardware is designed to be marketed at various price-points with different capabilities. The market then decides on the appropriate price versus capability trade-offs. This is actually a very good solution because the technology is evolving quickly -- each new generation of CUDA-enabled devices is more powerful than the previous generation and contains ever greater numbers of higher performance components at the same price points of the previous generation.

Wait! This sounds more like a software headache than a compromise because the CUDA developer needs to account for all these different hardware configurations and we are challenged with limited amounts of device resources. To help, several design aids have been created to help select the "best" high-performance execution configurations for different architectures. I highly recommend downloading and playing with the CUDA occupancy calculator, which is simply a nicely done spreadsheet. (The nvcc compiler will report information for each kernel that is needed for the spreadsheet when passed the --ptxas-options=-v option such as the number of registers as well as local, shared, and constant memory usage.) Still, a common piece
of advice in both the forums and documentation is, "try some different configurations and measure the effect on performance". This is easy to do since the execution configuration is specified by variables. In fact, many applications might be able to effectively auto configure themselves (e.g., determine the best execution configuration) when installed. Also, the CUDA runtime calls `cudaGetDeviceCount()` and `cudaGetDeviceProperties()` provide a way to enumerate the CUDA devices in a system and retrieve their properties. One possible way to use this information is to perform a table lookup for the best performing execution configurations or to jump start an auto tuner.

The CUDA Execution Model

To potentially increase performance, each hardware multiprocessor has the ability to actively process multiple blocks at one time. How many depends on the number of registers per thread and how much shared memory per block is required by a given kernel. The blocks that are processed by one multiprocessor at one time are referred to as active. Kernels with minimal resource requirements can better utilize (or occupy) each multiprocessor because the registers and shared memory of the multiprocessor are split among all the threads of the active blocks. Use the CUDA occupancy calculator to explore the trade-offs between number of threads and active blocks versus the number of registers and amount of shared memory. Finding the right combination can greatly increase the performance of your kernels. If there are not enough registers or shared memory available per multiprocessor to process at least one block, the kernel will fail to launch. (See the Part 3 discussion on `cudaGetLastError()` to find out how to catch these failures.)

Each active block is split into SIMD ("Single Instruction Multiple Data") groups of threads called "warps". Each warp contains the same number of threads, called the "warp size", which are executed by the multiprocessor in a SIMD fashion. This means each thread within a warp is broadcast the same instruction from the instruction store, which directs the thread to perform some operation or manipulation of local and/or global memory. The SIMD model is efficient and cost effective from a hardware standpoint, but from a software standpoint it unfortunately serializes conditional operations (e.g., both branches of the conditional must be evaluated). Be aware that conditional operations can have profound effects on the runtime of your kernels. With care this is generally a manageable problem but it can be problematic for some problems.

Active warps (that is, all the warps from all active blocks) are time-sliced: The thread scheduler periodically switches from one warp to another to maximize the use of the multiprocessor's computational resources. The order of execution of the warps within a block and of blocks themselves is undefined, which means they can occur in any order. However, threads can be synchronized with `__syncthreads()`. Be aware that only after the execution of `__syncthreads()` are writes to shared (and global) memory guaranteed to be visible. Unless the variable is declared as volatile, the compiler is free to optimize (that is, reorder or eliminate) memory reads and writes to increase performance. The `__syncthreads()` call is allowed inside the scope of a conditional, but only if the conditional evaluates identically across the entire thread block. If not, the code execution is likely to hang or produce unintended side effects. Happily, `__syncthreads()` has low overhead as it only takes four (4) clock cycles to issue for a warp so long as no other thread has to wait for any other thread. A half-warp is either the first or second half of a warp, which is an important concept for memory accesses including coalescing memory accesses as discussed later in this article.

There are several take away messages from the previous discussion:

- Multiprocessor resources such as shared memory are limited and valuable.
- Effectively managing limited multiprocessor resources, such as shared memory, for a range of CUDA-enabled device configurations is a fact of life for CUDA developers.
- Be aware that conditional operations (such as `if` statements) can have a profound effect on the runtime of your kernels.
- The CUDA occupancy calculator and nvcc compiler are important tools to learn and use -- especially when exploring execution configurations.

The CUDA Memory Model

Figure 1 schematically illustrates a thread that executes on the device has access to global memory and the on-chip memory through the memory types.
Each multiprocessor, illustrated as Block (0, 0) and Block (1, 0) above, contains the following four memory types:

- One set of local registers per thread.
- A parallel data cache or shared memory that is shared by all the threads and implements the shared memory space.
- A read-only constant cache that is shared by all the threads and speeds up reads from the constant memory space, which is implemented as a read-only region of device memory. (Constant memory will be discussed in a later column. Until then, please refer to section 5.1.2.2 of the CUDA Programming Guide for more information.)
- A read-only texture cache that is shared by all the processors and speeds up reads from the texture memory space, which is implemented as a read-only region of device memory. (Texture memory will be discussed in a subsequent article. Until then, refer to section 5.1.2.3 of the CUDA Programming Guide for more information.)

Don't be confused by the fact the illustration includes a block labeled "local memory" within the multiprocessor. Local memory implies "local in the scope of each thread". It is a memory abstraction, not an actual hardware component of the multi-processor. In actuality, local memory gets allocated in global memory by the compiler and delivers the same performance as any other global memory region. Local memory is basically used by the compiler to keep anything the programmer considers local to the thread but does not fit in faster memory for some reason. Normally, automatic variables declared in a kernel reside in registers, which provide very fast access. In some cases the compiler might choose to place these variables local memory, which might be the case when there are too many register variables, an array contains more than four elements, some structure or array would consume too much register space, or when the compiler cannot determine if an array is indexed with constant quantities.

Be careful because local memory can cause slow performance. Inspection of the ptx assembly code
(obtained by compiling with the \texttt{-ptx} or \texttt{-keep} option) will tell if a variable has been placed in local memory during the first compilation phases as it will be declared using the \texttt{.local} mnemonic and accessed using the \texttt{ld.local} and \texttt{st.local} mnemonics. If it has not, subsequent compilation phases might still decide otherwise though if they find it consumes too much register space for the targeted architecture.

Until the next column installment, I recommend using the occupancy calculator to get a solid understanding of how the execution model and kernel launch execution configuration affects the number of registers and amount of shared memory.

**For More Information**

- CUDA, Supercomputing for the Masses: Part 8
- CUDA, Supercomputing for the Masses: Part 7
- CUDA, Supercomputing for the Masses: Part 6
- CUDA, Supercomputing for the Masses: Part 5
- CUDA, Supercomputing for the Masses: Part 4
- CUDA, Supercomputing for the Masses: Part 3
- CUDA, Supercomputing for the Masses: Part 2
- CUDA, Supercomputing for the Masses: Part 1

Click here for more information on CUDA and here for more information on NVIDIA.
CUDA, Supercomputing for the Masses: Part 5

Understanding and using shared memory (2)

By Rob Farber, Dr. Dobb's Journal
Jun 30, 2008
URL: http://www.ddj.com/hpc-high-performance-computing/208801731

Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com.

In Part 4 of this article series on CUDA, I discussed how the execution model and kernel launch execution configuration affects the number of registers and amount of local multiprocessor resources such as shared memory. In this installment, I continue with a discussion of memory performance and the use of shared memory in reverseArray_multiblock_fast.cu.

CUDA Memory Performance

The local and global memory spaces are not cached which means each memory access to global memory (or local memory) generates an explicit memory access. So what does it cost to access (read or write, for example) each of the different memory types?

A multiprocessor takes four clock cycles to issue one memory instruction for a "warp". Accessing local or global memory incurs an additional 400 to 600 clock cycles of memory latency. As an example, the assignment operator in the code snippet below takes four clock cycles to issue a read from global memory, four clock cycles to issue a write to shared memory, and 400 to 600 clock cycles to read a float from global memory. Note: the __device__ variable type qualifier is used to denote a variable that resides in global memory (among other variable characteristics; see section 4.2.2.1 of the CUDA Programming Guide for more information). Variables of type __device__ cannot be accessed by host code.

```c
__shared__ float shared[32];
__device__ float device[32];
shared[threadIdx.x] = device[threadIdx.x];
```

With a factor of 100x-150x difference in access time, it is no surprise that developers need to minimize accesses to global memory and reuse data within the local multiprocessor memories. The CUDA designers have done a good job with the thread scheduler; so much of the global memory latency can be transparently hidden just by specifying large numbers of blocks in the execution configuration and working as much as possible with variables with register, __shared__, and __constant__ memory types in the kernel.

Since shared memory is on chip accesses are significantly faster than accesses to global memory and the main optimization is avoiding bank conflicts. Shared memory is fast (some documentation indicates it is as fast as register accesses). However, recent large improvements in CUBLAS and CUFFT performance were achieved by avoiding shared memory in favor of registers -- so try to use registers whenever possible. CUDA shared memory is divided into equally-sized memory modules that are called memory banks. Each memory bank holds a successive 32-bit value (like an int or float) so consecutive array accesses by consecutive threads are very fast. Bank conflicts occur when multiple requests are made for data from the same bank (either the same address or multiple addresses that map to the same bank). When this happens, the hardware effectively serializes the memory operations, which forces all the threads to wait until all the memory requests are satisfied. If all threads read from the same shared memory address then a broadcast mechanism is automatically invoked and serialization is avoided. Shared memory broadcasts are an excellent and high-performance way to get data to many threads simultaneously. It is worthwhile trying to
exploit this feature whenever you use shared memory.

I will discuss bank conflicts in greater detail in a future column. For the moment, suffice it to say that reverseArray_multiblock_fast.cu has no bank conflicts because consecutive threads access consecutive values.

A quick summary of local multiprocessor memory types with read/write capability follows:

- Registers:
  - The fastest form of memory on the multi-processor.
  - Is only accessible by the thread.
  - Has the lifetime of the thread.
- Shared Memory:
  - Can be as fast as a register when there are no bank conflicts or when reading from the same address.
  - Accessible by any thread of the block from which it was created.
  - Has the lifetime of the block.
- Global memory:
  - Potentially 150x slower than register or shared memory -- watch out for uncoalesced reads and writes which will be discussed in the next column.
  - Accessible from either the host or device.
  - Has the lifetime of the application.
- Local memory:
  - A potential performance gotcha, it resides in global memory and can be 150x slower than register or shared memory.
  - Is only accessible by the thread.
  - Has the lifetime of the thread.

**Shared Memory Cautions**

- Watch out for shared memory bank conflicts, which can slow performance.
- All dynamically allocated shared variables in a kernel start at the same memory address. Using more than one dynamically allocated shared memory array requires manually generating the offset. For example, if you want dynamically allocated shared memory to contain two arrays, \( a \) and \( b \), then you need to do something like:

```c
__global__ void kernel(int aSize)
{
    extern __shared__ float sData[];
    float *a, *b;
    a = sData;  
    b = &a[aSize];
}
```

**Register/Local Memory Cautions**

- Register memory can be transparently placed into local memory. This can potentially be a cause for poor performance. Check the ptx assembly code or look for lmem in the output from nvcc with the "-ptxas-options=-v".
- Arrays indexed by constants known at compile time typically reside in registers but if they are indexed by variables they cannot reside in registers. This creates a conundrum for the developer because loop unrolling may be required to keep array elements in register memory as opposed to slow global memory. However, unrolling loops can greatly increase register usage, which may result in variables being kept in local memory -- obviating any benefit of loop unrolling. It is possible to use the nvcc option, -maxrregcount=value to tell the compiler to use more registers. (Note: the maximum register count that can be specified is 128.) This is a tradeoff between using more registers and creating fewer threads, which may hinder the opportunities to hide memory latency. With some architectures, use of this option may also prevent kernels from starting due to insufficient resources.

**A Shared Memory Kernel**

Both programs reverseArray_multiblock.cu and reverseArray_multiblock_fast.cu perform the same tasks. They create a 1D array of integers, \( h_a \), containing the integer values \([0 .. \text{dimA}-1]\). The array is then moved via cudaMemcpy to the device and the host then launches the reverseArrayBlock kernel to reverse order the array contents in place. Again, cudaMemcpy is used to transfer data from the device to the host where a check is performed to verify that the device produced the correct result (for example, \([\text{dimA}-1 .. 0]\)).
The difference is that reverseArray_multiblock_fast.cu uses shared memory to improve the performance of the kernel, while reverseArray_multiblock.cu operates entirely in global memory. Try timing the two programs and verify for yourself the difference in performance. Also, reverseArray_multiblock.cu accesses global memory in an inefficient manner. We will use the CUDA profiler to help diagnose and fix this performance issue in a future column, and show how improvements in the new 10 series architecture eliminate the need for these types of optimizations in many cases.

```c
#include <stdio.h>
#include <assert.h>

// Simple utility function to check for CUDA runtime errors
void checkCUDAError(const char* msg);

// Part 2 of 2: implement the fast kernel using shared memory
__global__ void reverseArrayBlock(int *d_out, int *d_in)
{
    extern __shared__ int s_data[];

    int inOffset = blockDim.x * blockIdx.x;
    int in = inOffset + threadIdx.x;

    // Load one element per thread from device memory and store it
    // *in reversed order* into temporary shared memory
    s_data[blockDim.x - 1 - threadIdx.x] = d_in[in];

    __syncthreads();

    // write the data from shared memory in forward order,
    // but to the reversed block offset as before
    int outOffset = blockDim.x * (gridDim.x - 1 - blockIdx.x);
    int out = outOffset + threadIdx.x;
    d_out[out] = s_data[threadIdx.x];
}
```

```c
int main( int argc, char** argv)
{
    // Part 1 of 2: Compute the number of bytes of shared memory needed
    // This is used in the kernel invocation below
    int numBlocks = numBlocks * sizeof(int);

    // allocate host and device memory
    size_t memSize = numBlocks * numThreadsPerBlock * sizeof(int);
    h_a = (int *) malloc(memSize);
    cudaMalloc( (void **) &d_a, memSize );
}```
cudaMalloc( (void **) &d_b, memSize );

// Initialize input array on host
for (int i = 0; i < dimA; ++i)
{
    h_a[i] = i;
}

// Copy host array to device array
cudaMemcpy( d_a, h_a, memSize, cudaMemcpyHostToDevice );

// launch kernel
dim3 dimGrid(numBlocks);
dim3 dimBlock(numThreadsPerBlock);
reverseArrayBlock<<< dimGrid, dimBlock, sharedMemSize >>>( d_b, d_a );

// block until the device has completed
cudaThreadSynchronize();

// check if kernel execution generated an error
// Check for any CUDA errors
checkCUDAError("kernel invocation");

// device to host copy
cudaMemcpy( h_a, d_b, memSize, cudaMemcpyDeviceToHost );

// Check for any CUDA errors
checkCUDAError("memcpy");

// verify the data returned to the host is correct
for (int i = 0; i < dimA; i++)
{
    assert(h_a[i] == dimA - 1 - i);
}

// free device memory
cudaFree(d_a);
cudaFree(d_b);

// free host memory
free(h_a);

// If the program makes it this far, then the results are correct and
// there are no run-time errors.  Good work!
printf("Correct!
");

return 0;
}

void checkCUDAError(const char *msg)
{
    cudaError_t err = cudaGetLastError();
    if( cudaSuccess != err)
    {
        fprintf(stderr, "Cuda error: %s: %s.
", msg, cudaGetErrorString( err ));
        exit(EXIT_FAILURE);
    }
}

Deciding on the amount of shared memory at runtime requires some setup in both host and device code. In this example, the amount of shared memory (in bytes) for each block in a kernel is specified in the execution configuration on the host as an optional third parameter. (Setup on the host side is only required if the amount of shared memory is specified at kernel launch. If it's fixed at compile time no setup is required on the host side.) By default, the execution configuration assumes no shared memory is used. For example, in the host code of arrayReversal_multiblock_fast.cu, the following code snippet allocates shared memory for an array of integers containing a number of elements equal to the number of threads in a block:
// Part 1 of 2: Compute the number of bytes of share memory needed
// This is used in the kernel invocation below
int sharedMemSize = numThreadsPerBlock * sizeof(int);

Looking at the reverseArrayBlock kernel, the shared memory is declared with the following:

extern __shared__ int s_data[];

Note that the size is not indicated in the kernel -- rather it is obtained from the host through the execution configuration.

Until the next column on profiling, I recommend looking at the reverseArray_multiblock.cu. Do you think there is a performance problem in accessing global memory? If you think there is a problem, try to fix it.

For More Information

- CUDA, Supercomputing for the Masses: Part 9
- CUDA, Supercomputing for the Masses: Part 8
- CUDA, Supercomputing for the Masses: Part 7
- CUDA, Supercomputing for the Masses: Part 6
- CUDA, Supercomputing for the Masses: Part 5
- CUDA, Supercomputing for the Masses: Part 4
- CUDA, Supercomputing for the Masses: Part 3
- CUDA, Supercomputing for the Masses: Part 2
- CUDA, Supercomputing for the Masses: Part 1

Click here for more information on CUDA, here for the CUDA Occupancy Calculator, and here for more information on NVIDIA.
CUDA, Supercomputing for the Masses: Part 6

Global memory and the CUDA profiler

By Rob Farber, Dr. Dobb's Journal
Jul 25, 2008
URL: http://www.ddj.com/architect/209601096

Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com.

In Part 5 of this article series on CUDA (short for "Compute Unified Device Architecture"), I discussed memory performance and the use of shared memory in reverseArray_multiblock_fast.cu. In this installment, I examine global memory using the CUDA profiler.

Astute readers of this series timed the two versions of the reverse array example discussed in Part 4 and Part 5 and were puzzled about how the shared memory version is faster than the global memory version. Recall that the shared memory version, reverseArray_multiblock_fast.cu, kernel copies array data from the global memory to the shared memory, then back to global memory while the slower kernel, reverseArray_multiblock.cu, only copies data from global memory to global memory. Since global memory performance is between 100x-150x slower than shared memory, shouldn't the significantly slower global memory performance dominate the runtime of both examples? Why is the shared memory version faster?

Answering this question requires understanding more about global memory plus the use of additional tools from the CUDA development environment -- specifically the CUDA profiler. Profiling CUDA software is fast and easy, as both the text and visual versions of the profiler read hardware profile counters on CUDA-enabled devices. Enabling text profiling is as easy as setting the environmental variables that start and control the profiler. Using the visual profiler is equally easy: Just start cudaprof and start clicking in the GUI. Profiling provides valuable insight. The collection of profile events is handled entirely by hardware within CUDA enabled devices. However, profiled kernels are no longer asynchronous. Reporting of results to the host only occurs after each kernel completes, which minimizes any communications impact.

Global Memory

Understanding how to efficiently use global memory is an essential requirement to becoming an adept CUDA programmer. Following is a brief discussion about global memory that should be sufficient to understand the performance difference between reverseArray_multiblock.cu and reverseArray_multiblock_fast.cu. Future columns will, of necessity, continue to explore efficient uses of global memory. In the meantime, a detailed discussion on global memory, with illustrations, can be found in Section 5.1.2.1 of the CUDA Programming Guide.

Global memory delivers the highest memory bandwidth only when the global memory accesses can be coalesced within a half-warp so the hardware can then fetch (or store) the data in the fewest number of transactions. CUDA Compute Capability devices (1.0 and 1.1) can fetch data in a single 64-byte or 128-byte transaction. If the memory transaction cannot be coalesced, then a separate memory transaction will be issued for each thread in the half-warp, which is undesirable. The performance penalty for non-coalesced memory operations varies according to the size of the data type. The CUDA documentation provides some rough guidelines for the expected performance degradation to expect for various size data types:

- 32-bit data types will be roughly 10x slower
- 64-bit data types will be roughly 4x slower
- 128-bit data types will be roughly 2x slower

Global memory access by all threads in the half-warp of a block can be coalesced into efficient memory transactions on a G80 architecture when:

1. The threads access 32-, 64- or 128-bit data types.
2. All 16 words of the transaction must lie in the same segment of size equal to the memory transaction size (or twice the memory transaction size when accessing 128-bit words). This implies that the starting address and alignment are important.
3. Threads must access the words in sequence: the kth thread in the half-warp must access the kth word. Note: not all threads in a warp need to access memory for the thread accesses to coalesce. This is...
Newer architectures such as the GT200 family of devices have more relaxed coalescing requirements than those just discussed. I will discuss architectural differences more deeply in a future column. For purposes here, suffice to say that if you tune your code to coalesce well on a G80 CUDA-enabled device, it will coalesce well on a GT200 device.

Enabling and Controlling Textual Profiling

The environmental variables that control the text version of the CUDA profiler are:

- CUDA_PROFILE: Set to 1 or 0 to enable/disable the profiler
- CUDA_PROFILE_LOG: Set to the name of the log file (The default is ./cuda_profile.log)
- CUDA_PROFILE_CSV: Set to 1 or 0 to enable or disable a comma separated version of the log
- CUDA_PROFILE_CONFIG: Specify a configuration file with up to four signals

The last bullet is important because only four signals can be profiled at a time. The developer can have the profiler collect any of the following events by specifying their names on separate lines in the file named by CUDA_PROFILE_CONFIG:

- gld_incoherent: Number of non-coalesced global memory loads
- gld_coherent: Number of coalesced global memory loads
- gst_incoherent: Number of non-coalesced global memory stores
- gst_coherent: Number of coalesced global memory stores
- local_load: Number of local memory loads
- local_store: Number of local memory stores
- branch: Number of branch events taken by threads
- divergent_branch: Number of divergent branches within a warp
- instructions: instruction count
- warp_serialize: Number of threads in a warp that serialize based on address conflicts to shared or constant memory
- cta_launched: executed thread blocks

Notes on Profiler Counters

Note that the performance counter values do not correspond to individual thread activity. Instead, these values represent events within a thread warp. For example, an incoherent store within a thread warp will increment the gst_incoherent counter by 1. So the final counter value stores information for all incoherent stores in all warps.

In addition, the profiler can only target one of the multiprocessors in the GPU, so the counter values will not correspond to the total number of warps launched for a particular kernel. For this reason, when using the performance counter options in the profiler the user should always launch enough thread blocks to ensure that the target multiprocessor is given a consistent percentage of the total work. In practice, NVIDIA suggests it is best to launch at least 100 blocks or so for consistent results.

As a result, users should not expect the counter values to match the numbers one would determine through inspection of the kernel code. Counter values are best used to identify relative performance differences between unoptimized and optimized code. For example, if the profiler reports some number of non-coalesced global loads for an initial piece of software, then it is easy to see if a more refined version of the code utilizes a smaller number of non-coalesced loads. In most cases, the goal is to make the number of non-coalesced global loads zero, so the counter value is useful for tracking progress toward this goal.

Profiling Results

Let's look at reverseArray_multiblock.cu and reverseArray_multiblock_fast.cu with the profiler. In this example, we will set the environment variables and configuration file in the bash shell under Linux as follows:

```bash
export CUDA_PROFILE=1
export CUDA_PROFILE_CONFIG=$HOME/.cuda_profile_config
```

Profiler configuration via environment variables in Linux with bash

```bash
gld_coherent
gld_incoherent
gst_coherent
gst_incoherent
```

Contents of the CUDA_PROFILE_CONFIG file

Running the reverseArray_multiblock.cu executable generates the following profiler report in ./cuda_profile.log:

```bash
method, gputime, cputime, occupancy, gld_incoherent, gld_coherent, gst_incoherent, gst_coherent
method[ memcopy ] gputime=[ 438.432 ]
method[ memcopy ] gputime=[ 349.344 ]
```
Profile report for reverseArray_multiblock.cu

Similarly, running the reverseArray_multiblock_fast.cu executable produces the following output, which overwrites the previous output in .cuda_profile.log.

```
method,gputime,cputime,occupancy,gld_incoherent,gld_coherent,gst_incoherent,gst_coherent
method=[ memcopy ] gputime=[ 449.600 ]
method=[ memcopy ] gputime=[ 509.984 ]
```

Profile report for reverseArray_multiblock_fast.cu

Comparing these two profiler results shows that reverseArray_multiblock_fast.cu has zero incoherent stores as opposed to reverseArray_multiblock.cu, which has many. Look at the source of reverseArray_multiblock.cu and see if you can fix the performance problem with incoherent stores. Once fixed, measure how fast the two programs are relative to each other.

For convenience, Listing One presents reverseArray_multiblock.cu and Listing Two reverseArray_multiblock_fast.cu.

```
#include <stdio.h>
#include <assert.h>

void checkCUDAError(const char* msg);

__global__ void reverseArrayBlock(int *d_out, int *d_in)
{
    int inOffset  = blockDim.x * blockIdx.x;
    int outOffset = blockDim.x * (gridDim.x - 1 - blockIdx.x);
    int in  = inOffset + threadIdx.x;
    int out = outOffset + (blockDim.x - 1 - threadIdx.x);
    d_out[out] = d_in[in];
}
```

```
#include <stdio.h>
#include <assert.h>

void checkCUDAError(const char* msg);

__global__ void reverseArrayBlock(int *d_out, int *d_in)
{
    int inOffset = blockDim.x * blockIdx.x;
    int outOffset = blockDim.x * (gridDim.x - 1 - blockIdx.x);
    int in  = inOffset + threadIdx.x;
    int out = outOffset + (blockDim.x - 1 - threadIdx.x);
    d_out[out] = d_in[in];
}
```

```
main( int argc, char** argv)
{  
    int h_a[1] = 1;
}
```

```
main( int argc, char** argv)
{  
    int h_a[1] = 1;
}
```

```
// First, compute number of blocks needed based on array size and desired block size
int numBlocks = dimA / numThreadsPerBlock;
```

```
// First, compute number of blocks needed based on array size and desired block size
int numBlocks = dimA / numThreadsPerBlock;
```

```
// Allocate host and device memory
size_t memSize = numBlocks * numThreadsPerBlock * sizeof(int);
int h_a = (int *) malloc(memSize);
```

```
// Allocate host and device memory
size_t memSize = numBlocks * numThreadsPerBlock * sizeof(int);
int h_a = (int *) malloc(memSize);
```

```
// Initialize input array on host
for (int i = 0; i < dimA; ++i)
    h_a[i] = i;
```

```
// Initialize input array on host
for (int i = 0; i < dimA; ++i)
    h_a[i] = i;
```

```
// Copy host array to device array
cudaMemcpy( d_a, h_a, memSize, cudaMemcpyHostToDevice );
```

```
// Copy host array to device array
cudaMemcpy( d_a, h_a, memSize, cudaMemcpyHostToDevice );
```

```
// Launch kernel
dim3 dimGrid(numBlocks);
dim3 dimBlock(numThreadsPerBlock);
reverseArrayBlock<<< dimGrid, dimBlock >>>( d_b, d_a );
```

```
// Launch kernel
dim3 dimGrid(numBlocks);
dim3 dimBlock(numThreadsPerBlock);
reverseArrayBlock<<< dimGrid, dimBlock >>>( d_b, d_a );
```

```
// Check if kernel execution generated an error
// Check for any CUDA errors
```
checkCUDAError("kernel invocation");

  // device to host copy  
  cudaMemcpy( h_a, d_b, memSize, cudaMemcpyDeviceToHost );

  // Check for any CUDA errors  
  checkCUDAError("memcpy");

  // verify the data returned to the host is correct  
  for (int i = 0; i < dimA; i++)  
  {  
    assert(h_a[i] == dimA - 1 - i );
  }

  // free device memory
  cudaFree(d_a);
  cudaFree(d_b);

  // free host memory
  free(h_a);

  // If the program makes it this far, then the results are correct and  
  // there are no run-time errors. Good work!
  printf("Correct!\n");
  return 0;
}

void checkCUDAError(const char *msg)
{
  cudaError_t err = cudaGetLastError();
  if( cudaSuccess != err)
  {
    fprintf(stderr, "Cuda error: %s: %s.\n", msg, cudaGetErrorString( err ));
    exit(EXIT_FAILURE);
  }
}

reverseArray_multiblock.cu

  // includes, system
  #include <stdio.h>
  #include <assert.h>
  // Simple utility function to check for CUDA runtime errors
  void checkCUDAError(const char* msg);

  // Part 2 of 2: implement the fast kernel using shared memory  
  __global__ void reverseArrayBlock(int *d_out, int *d_in)
  {
    extern __shared__ int s_data[];

    int inOffset  = blockDim.x * blockIdx.x;
    int in  = inOffset + threadIdx.x;

    // Load one element per thread from device memory and store it  
    // *in reversed order* into temporary shared memory  
    s_data[blockDim.x - 1 - threadIdx.x] = d_in[in];

    // Block until all threads in the block have written their data to shared mem
    __syncthreads();

    // write the data from shared memory in forward order,  
    // but to the reversed block offset as before
    int outOffset = blockDim.x * (gridDim.x - 1 - blockIdx.x);
    int out = outOffset + threadIdx.x;
    d_out[out] = s_data[threadIdx.x];
  }

  // Program main
  int main( int argc, char** argv)
  {
    // pointer for host memory and size
    int *h_a;
    int dimA = 256 * 1024; // 256K elements (1MB total)
/* pointer for device memory */
int *d_b, *d_a;

/* define grid and block size */
int numThreadsPerBlock = 256;

/* Compute number of blocks needed based on array size and desired block size */
int numBlocks = dimA / numThreadsPerBlock;

/* Part 1 of 2: Compute the number of bytes of shared memory needed */
/* This is used in the kernel invocation below */
int sharedMemSize = numThreadsPerBlock * sizeof(int);

/* allocate host and device memory */
size_t memSize = numBlocks * numThreadsPerBlock * sizeof(int);
int *h_a = (int *) malloc(memSize);
cudaMalloc( (void **) &d_a, memSize );
cudaMalloc( (void **) &d_b, memSize );

/* Initialize input array on host */
for (int i = 0; i < dimA; ++i)
  h_a[i] = i;

/* Copy host array to device array */
cudaMemcpy( d_a, h_a, memSize, cudaMemcpyHostToDevice);

/* launch kernel */
reverseArrayBlock<<< dimGrid, dimBlock, sharedMemSize >>>( d_b, d_a );

/* block until the device has completed */
cudaThreadSynchronize();

/* check if kernel execution generated an error */
/* Check for any CUDA errors */
checkCUDAError( "kernel invocation" );

/* device to host copy */
cudaMemcpy( h_a, d_b, memSize, cudaMemcpyDeviceToHost );

/* Check for any CUDA errors */
checkCUDAError( "memcpy" );

/* verify the data returned to the host is correct */
for (int i = 0; i < dimA; i++)
  assert(h_a[i] == dimA - 1 - i );

/* free device memory */
cudaFree(d_a);
cudaFree(d_b);

/* free host memory */
free(h_a);

/* If the program makes it this far, then the results are correct and */
/* there are no run-time errors. Good work! */
printf("Correct!\n");

return 0;

void checkCUDAError(const char *msg)
{
  cudaError_t err = cudaGetLastError();
  if( cudaSuccess != err )
  {
    fprintf(stderr, "Cuda error: %s: %s.\n", msg, cudaGetErrorString(err) );
    exit(EXIT_FAILURE);
  }
}
reverseArray_multiblock_fast.cu

For More Information

- CUDA, Supercomputing for the Masses: Part 11
- CUDA, Supercomputing for the Masses: Part 10
- CUDA, Supercomputing for the Masses: Part 9
- CUDA, Supercomputing for the Masses: Part 8
- CUDA, Supercomputing for the Masses: Part 7
- CUDA, Supercomputing for the Masses: Part 6
- CUDA, Supercomputing for the Masses: Part 5
- CUDA, Supercomputing for the Masses: Part 4
- CUDA, Supercomputing for the Masses: Part 3
- CUDA, Supercomputing for the Masses: Part 2
- CUDA, Supercomputing for the Masses: Part 1

Click here for more information on CUDA and here for more information on NVIDIA.
CUDA, Supercomputing for the Masses: Part 7

Double the fun with next-generation CUDA hardware

By Rob Farber, Dr. Dobb's Journal
Aug 20, 2008
URL: http://www.ddj.com/hpc-high-performance-computing/210102115

In Part 6 of this article series on CUDA (short for "Compute Unified Device Architecture"), I examined global memory using the CUDA profiler. In this installment, I take a look at some next-generation CUDA hardware.

CUDA and CUDA-enabled devices are co-evolving to deliver more performance and capability with each new generation. NVIDIA's recent introduction of the GeForce 200-series and Tesla 10-series of products, shows the rapidity of this evolution as roughly twice the hardware capability is now available at the same price point of the previous line of products plus the 200-series includes the addition of some valuable (and potentially indispensable) new features.

The 1.4 billion transistors in the GTX280 -- as opposed to the 690 million transistors in the original G80 -- clearly illustrates the 2-for-1 approach NVIDIA has taken with this next generation of CUDA hardware. My experience shows that NVIDIA used these transistors very well indeed in the 200-series. Without changes, my single-precision codes run roughly 2x faster on the new hardware when compared against the previous generation G80 hardware!

The following list summarizes the important new features and capabilities of the 200-series architecture:

- The hardware now supports double precision arithmetic (there are 30 64-bit floating point units in the GTX280).
- Global memory is larger, faster, and easier to use. The coalescing rules have been relaxed -- which makes it easier to achieve high global memory performance -- and global memory bandwidth -- at over 100 GB/s -- is nearly double that of the G80 architecture.
- The number of single-precision registers has doubled per hardware thread (although the number of available registers has not changed relative to the previous architectures when using double-precision).
- The 200-series supports 32-bit atomic signed and unsigned integer functions in shared memory and 64-bit functions in global memory.
- The 200-series now includes warp vote functions.
- The number of thread processors has nearly doubled -- 240 as opposed to 128 -- and supports a greater number of active warps and active threads per multiprocessor.
- The 200-series has an enhanced hardware ability to perform a MAD and a MUL at the same time, which should help some applications better approach peak performance.

Let's take a closer look at what this all means for CUDA.
For many applications, the biggest new feature is hardware double-precision arithmetic. The speed and massive parallelism of the NVIDIA GPUs has brought supercomputing to the masses. A realization that quickly occurs when working with super-sized problems and data sets is that numerical noise can quickly accumulate (due to floating-point inaccuracies) and cause garbage results. Physical simulations, for example, can suddenly exhibit spectacular non-physical behavior and previously working simulations can become unstable and start generating infinity, NaN, or other nonsense values. Although not necessary for all computing problems, the use of higher precision floating-point representations (such as 64-bit double-precision floats) can really help. (In a future article, I'll look at the ways you can combine fast single precision with double-precision calculations to speed up your results.)

The 200-series architecture is the first to include hardware double precision. As might be expected with any first generation product, there is some room remaining for performance improvement (since thread-processors in a multiprocessors all share a single double-precision hardware unit). My experience indicates a minor couple of percent decrease in performance occurs when using double-precision so long as it is only used where required to preserve numerical accuracy. This is, of course, problem dependent and your mileage may vary.

Most programmers will find it much easier to attain high performance with the 10-series architecture.

By doubling the number of registers, NVIDIA has made it easier for the CUDA programmer to load sufficient single-precision data into the registers to reduce (or possibly) eliminate much of the bottleneck imposed by global memory. Since double-precision values require twice the storage as single-precision (8 bytes versus 4 bytes), the number of double-precision registers on the 200-series is identical to the number of single-precision registers on the G80 and G92 architectures.

The bandwidth of global memory has also nearly doubled.

The 10-series boards provide over 100 GB/s of global memory bandwidth. For 32-bit floats, global memory bandwidth lost ground in the 10-series relative to the G80 and G92 architectures, as seen in Table 1, because the number of thread processors nearly doubled.

<table>
<thead>
<tr>
<th>Architecture / card</th>
<th>Global Memory Bandwidth (GB/s)</th>
<th>Number of processing units</th>
<th>Million 32-bit operands available per processing unit per second from Global Memory (considering bandwidth only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>G80 / 8800 Ultra</td>
<td>104</td>
<td>128</td>
<td>203</td>
</tr>
<tr>
<td>10-series/ GTX 280</td>
<td>141</td>
<td>240</td>
<td>146</td>
</tr>
</tbody>
</table>

Table 1

Double-precision (64-bit floats) performance is complicated by the fact that there is one double-precision unit per multiprocessor (8 thread processors) and that doubles consume twice as much bandwidth to move around. CUDA developers on the 200-series may discover that an application that is (marginally) bandwidth limited when using single-precision floats could become compute limited after switching to double-precision.

This table does demonstrate that the NVIDIA hardware design team did an excellent job because they delivered a new product with nearly twice the global memory bandwidth plus double-precision capability for the same price-points as the previous generation products.

A big win for all CUDA applications is the fact that global memory is easier to access in a high performance manner due to the relaxation of the coalescing rules in the 200-series. Section 5.2.2 of the [CUDA Programming Guide](http://www.ddj.com/article/printableArticle.jhtml?articleID=210102115&dept_url=/hpc-high-performance-computing/) goes into more detail on the protocol used to issue memory transactions. Following are three characteristics that are well worth highlighting:

- Coalescing is achieved for any pattern of address requests including multiple requests to the same
address. Previous architectures needed to access words in sequence. In essence, this means that many more CUDA applications will get excellent global memory performance, and that CUDA programmers will no longer have to invent (if possible) workarounds to get their global memory access patterns "just right".

- If a half-warp addresses words in n different segments, then only n memory transactions are issued. If \( n=2 \), for example, only two memory transactions would be issued as opposed to the 16 transactions that would occur in previous architectures (e.g. 4x fewer transactions).
- Unfortunately, unused words in a memory are still read, so they waste bandwidth even though the hardware will issue the smallest memory transaction possible.

The 200-series supports atomic operations for signed and unsigned integer. The exception of \texttt{atomicExch()}\footnote{atomicExch()}, which is also supported for single-precision floating-point numbers. An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory. For example, \texttt{atomicAdd()} reads a 32-bit word at some address in global or shared memory, adds an integer to it, and writes the result back to the same address. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads -- no other thread can access this address until the operation is complete.

Warp vote functions available in the 200-series. If you need them, they provide an indispensable capability to perform fast predicate operations (e.g., checking of a condition is true or false) across all the threads of a warp:

\begin{verbatim}
int __all(int predicate);
\end{verbatim}

Evaluates the specified predicate for all threads of the warp and returns non-zero if and only if predicate evaluates to non-zero for all of them.

\begin{verbatim}
int __any(int predicate);
\end{verbatim}

Evaluates specified predicate for all threads of the warp and returns non-zero if and only if predicate evaluates to non-zero for any of them.

Finally, improvements to the ability of the hardware to perform simultaneous MAD and MUL operations should help some applications achieve FLOP (floating-point operations) rates closer to peak performance.

For further information, look at the CUDA Zone forums. I also recommend downloading the latest version of the \texttt{CUDA Programming Guide}\footnote{CUDA Programming Guide} on the NVIDIA website. The current version is 2.0b2, which includes discussion of the new features and API.

This may be an excellent time to upgrade to the 200-series of CUDA-enabled devices. Fierce competition (http://www.tgdaily.com/content/view/38243/135) has forced some significant pricing adjustments, which might make this the perfect time to shop for a deal!

**For More Information**

- CUDA, Supercomputing for the Masses: Part 9
- CUDA, Supercomputing for the Masses: Part 8
- CUDA, Supercomputing for the Masses: Part 7
- CUDA, Supercomputing for the Masses: Part 6
- CUDA, Supercomputing for the Masses: Part 5
- CUDA, Supercomputing for the Masses: Part 4
- CUDA, Supercomputing for the Masses: Part 3
- CUDA, Supercomputing for the Masses: Part 2
- CUDA, Supercomputing for the Masses: Part 1

Click here for more information on CUDA and here for more information on NVIDIA.
CUDA, Supercomputing for the Masses: Part 8

Using libraries with CUDA

By Rob Farber, Dr. Dobb's Journal
Sep 19, 2008
URL: http://www.ddj.com/hpc-high-performance-computing/210602684

Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com.

In CUDA, Supercomputing for the Masses: Part 7 of this article series on CUDA (short for "Compute Unified Device Architecture"), I took a look at some next-generation CUDA hardware. In this installment, I shift gears a bit, moving from hardware to software, focusing on using libraries with CUDA.

Optimized libraries often provide an easy way to improve performance of applications. When porting large legacy projects, libraries may be the only real way to optimize for a new platform because code changes would require extensive validation efforts. Essentially libraries are convenient and can greatly accelerate code development as well as application performance, but they cannot be blindly utilized. GPU-based libraries in particular require you to think carefully about data placement and how the library is used. Otherwise poor performance will be the result.

The Basic Linear Algebra Subprograms (BLAS) package is the de facto programming interface for basic linear algebra operations such as vector and matrix multiplication. NVIDIA supports this interface with its own library for the GPU called CUBLAS. CUFFT is another NVIDIA-supported library that provides a GPU-based implementation of the Fast Fourier Transform (FFT), a commonly used method in scientific and signal processing applications. It is modeled after FFTW, a very highly-optimized and popular FFT package for general-purpose processors.

BLAS is structured according to three different "Levels":

- Examples of level-1 algorithms include taking an inner product of two vectors, or scaling a vector by a constant multiplier.
- Level-2 algorithms are matrix-vector multiplication or a single right-hand-side triangular solve.
- Level-3 algorithms include dense matrix-matrix multiplication.

If we assume a vector is length N or a matrix is order N*N, then the number of floating-point operations for a level-1, Level-2, and level-3 algorithm are O(N), O(N^2), and O(N^3), respectively. (Big-O notation is a convenient way to describe how the size of an input affects an algorithm's consumption of a computational resource such as time or memory.)

The basic model by which applications use the CUBLAS library is to create matrix and vector objects in GPU memory space, fill them with data, call a sequence of CUBLAS functions, and, finally, move the results from GPU memory space back to the host. To accomplish this, CUBLAS provides helper functions for creating and destroying objects in GPU space, and for writing data to and retrieving data from these objects. CUBLAS uses column-major storage and 1-based indexing for maximum FORTRAN
compatibility. C and C++ applications need to use macros or inline functions to facilitate access to CUBLAS-created objects.

Data movement is a central consideration when using CUBLAS (and BLAS routines in general). BLAS data movement according to level is $O(N)$, $O(N^2)$, and $O(N^2)$, which makes the number of floating-point operations per data item moved $O(1)$, $O(1)$, and $O(N)$, respectively. This last fact indicates how important it is to locate the data on the GPU, as I describe in greater detail shortly.

Since the thread-processors of the GPU operate only on data local to the GPU, any BLAS operation on a vector or matrix located in the memory space of the host computer requires a data transfer operation. These data transfers are expensive relative to the floating-point capability of the graphics processors (GPUs can perform floating-point far faster than they can move data) and are to be avoided whenever possible or they will bottleneck performance.

To illustrate this behavior, consider the costs involved in moving a vector from the host system to the GPU, multiplying it by a constant, and then returning the result to the host. This "application" requires moving $4N$ bytes of data (where $N$ is the number of float in the vector) to and from the GPU to perform $N$ multiplications (the constant times the vector). The best possible performance this "application" could achieve is the transfer bandwidth divided by 8 (the number of bytes required to move a 32-bit float to and from the GPU). Such an application would be lucky to achieve $a = GFLOP$ (billion floating-point operations per second) floating-point performance assuming a 4 GB/s transfer rate between the host and GPU. This is well below the floating-performance capability of even the lowest-cost low-end NVIDIA GPU. Batching multiple operations could potentially this performance by exploiting the full-duplex capability of the PCIe bus.

Bottom line: Getting good performance for level-1 and level-2 BLAS applications requires keeping as much data as possible on the GPU. When that is not possible, at least try to batch as many library calls as possible.

Level-3 BLAS applications can achieve extremely efficient performance. Table 1 describes a SGEMM benchmark on various architectures (e.g., an HP xw8600 with a Xeon E5440 at 2.83 GHz, a C870 and a C1060 both running CUDA 2.0). It is worth noting that benchmark performs the same sequence of operations used by computational finance applications to perform fast matrix exponentiation -- so it does reflect a real-world performance capability. All results are reported in GFLOP. "Thunking" in the CUBLAS nomenclature is an interface that automatically allocates memory, copies data to the GPU, runs the computation, copies the results back, and deallocates memory. This makes it a drop-in replacement for BLAS but with obvious performance implications due to all of the data movement. The columns denoted "GPU only" are where data allocation/freeing and copies are managed by the application developer. In this case the memory is allocated once, the data is copied over, and a large sequence of SGEMM calls are made, and then the results are copied back. The resulting cost of moving data is trivially small. Included are results for several $N\times N$ matrix sizes on the 8- and 10-series Tesla cards and a quad-core CPU. Please notice that the CPU is faster for small matrix sizes than the thunking interface, but the GPU can perform much better if the application developer more closely manages data movement.

<table>
<thead>
<tr>
<th>N</th>
<th>CPU single threaded</th>
<th>CPU 2 cores</th>
<th>CPU 4 cores</th>
<th>C870 Thunking</th>
<th>C870 GPU only</th>
<th>C1060 Thunking</th>
<th>C1060 GPU only</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>14.92</td>
<td>29.84</td>
<td>51.14</td>
<td>44.75</td>
<td>171.29</td>
<td>67.13</td>
<td>245.12</td>
</tr>
<tr>
<td>768</td>
<td>14.62</td>
<td>28.32</td>
<td>53.31</td>
<td>75.51</td>
<td>172.86</td>
<td>90.62</td>
<td>332.18</td>
</tr>
<tr>
<td>1024</td>
<td>14.62</td>
<td>28.08</td>
<td>54.73</td>
<td>85.92</td>
<td>173.7</td>
<td>113.04</td>
<td>317.68</td>
</tr>
<tr>
<td>2048</td>
<td>20.48</td>
<td>40.06</td>
<td>76.71</td>
<td>121.01</td>
<td>166.72</td>
<td>197.5</td>
<td>354.96</td>
</tr>
</tbody>
</table>

Table 1

Improvements over the current generation of stock NVIDIA libraries can be achieved. For example, in their paper [LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs](http://www.ddj.com/article/printableArticle.jhtml?articleID=210602684&dept_url=/hpc-high-performance-computing/), Vasily Volkov and James Demmel claim to approach the peak performance of the G80 series GPUs with their matrix-matrix multiply
routine. Their paper includes a detailed benchmarking study of the GPU memory system, which they used to devise their particular improvements. They claim their method is approximately 60 percent faster than the CUBLAS 1.1 implementation. These improvements have already been incorporated into the CUBLAS 2.0 release which was used to generate the results in the previous table as NVIDIA is continually working to improve both the hardware and library code.

Excellent performance can be achieved with BLAS on the GPU. For example, FORTRAN programmers should look to FLAGON, an open source library project that implements a middle-ware call layer that adds GPU "device descriptors" to help optimize performance. In their Supercomputing 2007 poster, Nail A. Gumerov, Ramani Duraiswami, and William D. Dorland claim a 25x speedup over an Intel QX6700 serial CPU code using CUFFT and some exceptional performance results compared to serial code when using an iterative solver to fit radial basis functions to scattered data.

There are also many examples in the literature where GPU libraries fill a convenience role and do not provide any performance benefits (or even a performance disadvantage). As discussed previously, the cost of data movement can be the deciding factor. For example, see CUFFT vs. FFTW for an excellent characterization.

In summary, the NVIDIA CUFFT and CUBLAS libraries provide a convenient and generally accepted interface for linear algebra and the Fast Fourier Transform. These libraries can enable large projects to use graphics processors. Excellent performance can be achieved when care is taken to minimize or eliminate data movement. Increased performance can be achieved by batching multiple library calls. Significant improvements can also be achieved by paying close attention to the CUDA-enabled device characteristics. Since NVIDIA is continually improving both the hardware and library codes, expect continued performance improvements.

For More Information

- CUDA, Supercomputing for the Masses: Part 9
- CUDA, Supercomputing for the Masses: Part 8
- CUDA, Supercomputing for the Masses: Part 7
- CUDA, Supercomputing for the Masses: Part 6
- CUDA, Supercomputing for the Masses: Part 5
- CUDA, Supercomputing for the Masses: Part 4
- CUDA, Supercomputing for the Masses: Part 3
- CUDA, Supercomputing for the Masses: Part 2
- CUDA, Supercomputing for the Masses: Part 1

Click here for more information on CUDA and here for more information on NVIDIA.
CUDA, Supercomputing for the Masses: Part 9

Extending High-level Languages with CUDA

By Rob Farber, Dr. Dobb's Journal
Nov 01, 2008
URL: http://www.ddj.com/hpc-high-performance-computing/211800683

Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com.

In CUDA, Supercomputing for the Masses: Part 8 of this article series on CUDA (short for "Compute Unified Device Architecture"), I focused on using libraries with CUDA. In this installment, I look at how you can extend high-level languages (like Python) with CUDA.

CUDA lets programmers who develop in languages other than C and C++ harness the power of thousands of software threads simultaneously running on hundreds of thread-processors inside of today's graphics processors. Libraries (discussed in Part 8) provide some of this capability, as most languages can link with C-language libraries. A more flexible and powerful capability lies in the ability of many languages -- such as Python, Perl, and Java -- to be extended through modules written in C, or CUDA when programming for GPU environments. Much of the power in these extensions is a result of the freedom they offer developers to define classes and methods that can locate and operate on data within the GPU without being limited by a static library interface.

The possibilities of using CUDA as a language extension are huge and impact commercial, open-source, and academic developers alike. Instead of fighting for small, incremental percentage increases in performance, a simple check in the module (or library) to call CUDA code when running in a CUDA-enabled environment can yield orders of magnitude performance increases while preserving compatibility. Suddenly, those Apache servers become far more capable and Java client applets far more "fat", the open-source community is "wowed" by the extra power and capability of the open-source project, and scientists and engineers are able to use laptops and workstations for previously intractable or supercomputer-bound applications. Let's see what the future holds. (Personally I'd like to see a CUDA-enabled GPU running on a mobile phone that is accessible from a JME, Java Micro Edition, application!)

There are many tutorials on the web discussing how to interface your favorite language with C. The elegance within the CUDA design becomes apparent as all these tutorials also apply to CUDA because CUDA is C (with a few extensions to harness the massive parallelism of graphics processors). The beauty of this arrangement is that the best approach and design for a problem is left up to the developer. What can you do that will be most beneficial to your favorite project or make your favorite language even better?

Don't forget that CUDA-enabled devices can simultaneously run both compute and visualization tasks -- so don't limit your thinking to just computational extensions. Try it yourself and run a computational task while simultaneously running a heavy visualization task. (glxgears is a good graphics intensive visualization test for Linux X-windows users because it is generally available and dynamically reports a frames/second rate. However, this application only tests a small part of 3D graphics performance.)
High-throughput and real-time streaming extensions are also possible. Current CUDA-enabled devices use PCI-E 2.0 x16 buses to quickly move data around between system memory and amongst multiple graphics processors at GB/s (billion bytes per second) rates. Data intensive video games use this bandwidth to run smoothly -- even at very high frame rates. That same high-throughput can really enable some innovative CUDA applications. One example is the RAID software developed by researchers at the University of Alabama and Sandia National Laboratory (see Accelerating Reed-Solomon Coding in RAID Systems with GPUs by Matthew Curry, Lee Ward, Tony Skjellum, Ron Brightwell, IPDPS 2008), which I discuss in greater detail in "Massively Parallel Linux Laptops, Workstations and Clusters with CUDA" (Linux Journal, November 2008).

Remember that high-throughput is a relative term. The following graph is based on the table from Part 7 "Double the Fun with Next-generation Hardware". It shows that the PCI-E bus cannot provide even a tiny fraction of the bandwidth global memory can supply to the thread-processors, which spells performance disaster for applications that are PCI-E bandwidth limited. Recall from several earlier columns that even global memory on the GPU does not provide sufficient bandwidth to keep all the thread-processors busy. The PCI-E numbers take into account the upgrade to PCI-E 2.0 capability in the 10-series/GTX280 cards, which effectively doubled the PCI-E 1.0 bandwidth.

![Figure 1: Bandwidth available to the thread-processors from various sources (e.g., global memory and the PCI-E bus) for several architectures.](image-url)

For this reason, control over the location of the data in either host or GPU global memory is critical for an application to achieve high performance. As discussed in Part 8 "Using Libraries with CUDA", it does not make sense to perform something like a level-1 BLAS operation to transfer a vector to the GPU only to add a constant and then move the modified vector back into host memory. For low flop per data item operations, data location is likely to be the dominate factor affecting performance. It only makes sense to perform such operations when the data already exists on the GPU.

Be aware that a common pitfall in extending higher-level languages with C is the overhead incurred when converting variables and data structures between the two languages. CUDA programmers have the additional burden of minimizing the overhead caused by the necessity of transferring data between the separate memory spaces of the host and graphic processor(s).

What is a drawback can also be a benefit. In terms of data location and operators, GPUs can have a significant advantage over the current generation of commodity CPU processors with appropriate use of the on-card global memory. Creating a well-designed module for a high-level language (or C++ class) can
reduce to the barest minimum (or even eliminate) data transfer overhead because the programmer has the ability to control the location of the data and enforce the continued location of the data on the GPU through the methods they define. This can make even trivial operations like the addition of a constant to a vector very worthwhile on the GPU -- so long as they are used in combination with other operations to get a high ratio of flops to data items transferred to the GPU. Of course, how well a language extension performs and how generally applicable it is across a number of applications depends heavily on how well the extension (or class) is designed by the developer.

### Introducing SWIG

An excellent software development tool that connects modules written in C and C++ to a wide variety of high-level programming languages is SWIG which supports Perl, PHP, Python, Tcl, Java, C#, Common Lisp, Octave, R and many more (see [www.swig.org/compat.html#SupportedLanguages](http://www.swig.org/compat.html#SupportedLanguages) for more languages).

Here are some links to get you started for three common languages. Check out the web for your favorite if not listed below:

- Java: Use [SWIG or the JNI](http://www.swig.org/docs/SWIGJava.html) (Java Native Interface). One example project to get you going is [JCublas](http://www.nvidia.com/object/cudaerviewer.html), which makes the CUBLAS library discussed in Part 8 available to Java applications.
- Perl: SWIG is a good place to start as well as the [Wikipedia page](http://en.wikipedia.org/wiki/SWIG), although [CPAN](http://www.cpan.org) is the canonical PERL repository.
- Python: An overview of [extending Python with C (and hence CUDA)](http://www.nvidia.com/object/cudaerviewer.html) is a good place to start, as is [SWIG](http://www.swig.org/compat.html#SupportedLanguages). A working Python example (that operates on NumPy arrays) is [pystream](http://www.nvidia.com/object/cudaerviewer.html) and the related [Project GPUlib](http://www.nvidia.com/object/cudaerviewer.html).

The following is a simple Python example, contributed by a colleague at NVIDIA, which demonstrates the simplicity and speed of calling a CUDA kernel from Python. This example actually implements a useful method for financial applications -- namely matrix exponentiation. Unfortunately, the reasoning behind why such a method is useful is beyond the scope of this article. See the discussion starting on page 19 in the paper at [http://arxiv.org/pdf/0710.1606](http://arxiv.org/pdf/0710.1606) for more information. Be forewarned, this paper is quite dense.

In the spirit of this article, this example module makes efficient use of the GPU. The reason it performs so well is because this module lets Python programmers call [SGEMM](http://www.nvidia.com/object/cudaerviewer.html), a high flop per data item level-3 BLAS routine in the NVIDIA CUBLAS library. It also demonstrates that it is possible to map variables -- in this case an array -- very efficiently between Python and CUDA.

The full listing for the Python code exponentiationTest.py is:

```python
#!/usr/bin/env python

import copy
import numpy
import FastMatrixExp

# Read input matrix using a user defined function
a = myInputReader()
b = copy.copy(a)

steps = 100

# Matrix exponentiation using CPU SGEMM
for i in range(steps):
a = numpy.dot(a,a)

# Matrix exponentiation using CUBLAS SGEMM
FastMatrixExp.matrixMulLoop([steps,b])
```

file:///Users/tkaiser/Desktop/dobbs/part9.html
Within the exponentiationTest.py, a custom module is imported with the line:

```python
import FastMatrixExp
```

The reader is required to define its own Python method to input a matrix into variable `a`, which is then duplicated in variable `b` for purposes of comparing the speed and accuracy of the CPU and GPU:

```python
# Read input matrix using a user defined function
a = myInputReader()
b = copy.copy(a)
```

Matrix `a` is then raised to the power specified in the variable `steps` (specifically 100) on the host processor with this code snippet:

```python
steps = 100

# Matrix exponentiation using CPU SGEMM
for i in range(steps):
    a = numpy.dot(a, a)
```

After which the SGEMM routine from the CUBLAS library is called from Python and utilized on the GPU to perform the matrix exponentiation with the following:

```python
# Matrix exponentiation using CUBLAS SGEMM
FastMatrixExp.matrixMulLoop([steps, b])
```

Both the GPU and CPU generated results are then checked to see if they are equal within a reasonable tolerance via a numpy comparison as seen below. (Numpy is an excellent numerical Python package that has matrix operations.

```python
numpy.testing.assert_array_almost_equal(a, b, decimal = 6)
print 'Error = %f' % numpy.linalg.norm(a-b)
```

The following is the SWIG interface code:

```python
%module FastMatrixExp

%header
%
#include <oldnumeric.h>
#include <cublas.h>
%

#include exception.i

/* Matrix multiplication loop for fast matrix exponentiation. */
%typemap(python,in) (int steps, float *u, int n)
{
    $1 = PyInt_AsLong(PyList_GetItem($input, 0));
```
The module name, FastMatrixExp, is defined in the first line of CUBLAS.i:

```
#module FastMatrixExp
```

The iterated calls to cublasSgemm occur in the following C subroutine, which is defined between the %{ and %} for SWIG:

```
%{
void matrixMulLoop(int steps, float *u, int n)
{
    int i;
    float *ud;
    cublasStatus status;

    /* Allocate memory and copy u to the device. */
    status = cublasAlloc(n*n, sizeof(float), (void **)ud);
    status = cublasSetMatrix(n, n, sizeof(float), (void *)u,n, (void *)ud, n);

    /* Do "steps" updates. */
    for(i=0; i<steps; i++)
        cublasSgemm('n','n',n,n,n,1.0f,ud,n,ud,n,0.0f,ud,n);

    /* Copy u back to the host and free device memory. */
    status = cublasGetMatrix(n, n, sizeof(float), (void *)ud,n, (void *)u, n);
    status = cublasFree((void *)ud);
}
%

%init
%
{ import_array();
  cublasStatus status;
  status = cublasInit();
%
```
To gain a greater understanding of the remaining parts of the SWIG file, I recommend consulting the SWIG documentation. You can also find out more about SWIG in David Beazley's article SWIG and Automated C/C++ Scripting Extensions, and Daniel Blezek's article Rapid Prototyping with SWIG.

For more advanced numerical packages that combine Python and CUDA, checkout pystream or GPUlib (which can be downloaded after submitting an email request).
CUDA, Supercomputing for the Masses: Part 10

CUDPP, a powerful data-parallel CUDA library

By Rob Farber, Dr. Dobb's Journal
Jan 29, 2009
URL: http://www.ddj.com/architect/212903437

Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com.

In CUDA, Supercomputing for the Masses: Part 9 of this article series on CUDA (short for "Compute Unified Device Architecture"), I looked at how you extend high-level languages (like Python) with CUDA. In this installment, I examine CUDPP, the "CUDA Data Parallel Primitives Library." CUDPP is a quickly maturing package that implements some not-so-obvious algorithms to efficiently use the GPU for basic data-parallel operations such as sorting, stream compaction, and even building data structures like trees and summed-area tables. I discuss CUDPP here because it might provide some of the functionality needed to quickly speed the development of one of your projects.

I also introduce the concept of creating a "plan", a programming pattern used to provide an optimized execution configuration based on problem specification and destination hardware. Although not an optimizing compiler, the use of plans can greatly enhance the ability of programmers to create efficient software for multiple types of CUDA-enabled GPUs -- in addition to providing the ability to select problem-specific optimized code for specific problems within a general-purpose library framework. The NVIDIA cuFFT library, for example, can decide to use more efficient power-of-two FFT algorithms when appropriate. While the concept of a plan is not new to CUDA or this article series, it is a common design pattern that has stood the test of time.

Why Use CUDPP?

Most of us have a tool kit of libraries and methods that we use to do some of our work for us. In a nutshell, these libraries provide primitives that we can use to quickly and efficiently perform some of our computational tasks. Sorting is one example where it is just as easy and efficient to call something like the qsort() routine to return a data structure in sorted order. The NVIDIA cuBLAS and cuFFT libraries provide similar functionality for some not-so-easy tasks such as programming the FFT and optimized BLAS functionality.

CUDPP uses the same ideas to provide a library of optimized "best in class" methods to perform primitive operations such as parallel-prefix-sum ("scan"), parallel sort (of numbers), parallel reduction and other methods that permit the efficient implementation of sparse matrix-vector multiply, and other operations.

A parallel-prefix scan is a primitive that can help in implementing efficient solutions to parallel problems in which each output apparently requires global knowledge of the inputs. For example, the prefix sum (also known as the "scan", "prefix reduction", or "partial sum") is an operation on lists in which each element in the result list is obtained from the sum of the elements in the operand list up to its index. This appears to be a serial operation because each result depends on all the previous values as follows:
**Definition:** The *all-prefix-sums* operation takes a binary associative operator \( \odot \) and an array of \( n \) elements:

- given: \([a_0, a_1, ..., a_{n-1}]\),
- returns: \([a_0, (a_0 \odot a_1), ..., (a_0 \odot a_1 \odot ... \odot (a_{n-1})]\).

**Example:** If \( \odot \) is addition, then the *all-prefix-sums* operation on the array of \( n \) elements given \([3, 1, 7, 0, 4, 1, 6, 3]\) returns \([3, 4, 11, 11, 15, 16, 22, 25]\).

There are many uses for *all-prefix-sums* illustrated above, including, but not limited to sorting, lexical analysis, string comparison, polynomial evaluation, stream compaction, and building histograms and data structures (graphs, trees, etc.) in parallel. There are various survey papers that provide more extensive and detailed applications such as Guy Blelloch's *Prefix Sums and Their Applications*.

Obviously a sequential version of scan (that could be run in a single thread on a CPU, for example) is trivial. We simply loop over all the elements in the input array and add the value of the previous element of the input array to the sum computed for the previous element of the output array, and write the sum to the current element of the output array.

```c
void scan( float* output, float* input, int length)
{
    output[0] = 0; // since this is a prescan, not a scan
    for(int j = 1; j < length; ++j) {
        output[j] = input[j-1] + output[j-1];
    }
}
```

This code performs exactly \( n \) additions for an array of length \( n \) -- the minimum number of additions required to produce the scanned array. It would be wonderful if a parallel version of scan could be work-efficient, which means the parallel version performs no more addition operations (or work) than the sequential version. In other words the two implementations should have the same work complexity, \( O(n) \). CUDPP claims to achieve \( O(n) \) scan runtime, which should clarify the value of CUDPP because creating a parallel implementation is non-trivial. For more information, see *Scan Primitives for GPU Computing* by Shubhabrata Sengupta et al.

For version 1.0, CUDPP provides:

- **Segmented Scan:** an algorithm for performing multiple variable-length scans in parallel. Useful for algorithms such as parallel quicksort, parallel sparse matrix-vector multiplication, and more.
- **Sparse Matrix-Vector Multiplication (based on segmented scan):** Sparse matrix operations are important because they allow GPUs to work on matrices with many zeros (e.g., a sparse matrix) in both a space and computationally efficient way. Since most of the values are zero, most of the work can be avoided. Similarly, there is no need to waste space in storing the zeros.
- **An improved scan algorithm, called "warp scan", for higher performance and simpler code.**
- **Scans and segmented scans now support add, multiply, maximum, and minimum operators.**
- **Inclusive scans and segmented scans are now supported.**
- **Improved, more useful, cudppCompact() interface.**
- **Backward compact (reverse-and-compact) is now supported.**
- **CUDA 2.0 support.**
- **Added support for Mac OS X and Windows Vista.**

**Downloading and Installing**

CUDPP can be downloaded from gpgpu.org in the developer section at [http://www.gpgpu.org/developer/cudpp](http://www.gpgpu.org/developer/cudpp). To install under Linux, download the latest version (currently cudpp_1.0a.tar.gz), unpack, change to the cudpp_1.0a directory, then build and run a test case (or cases by removing the comment symbol, '#' from the last line) as follows:

```bash
Download and Installing
```

---

http://www.ddj.com/article/printableArticle.jhtml?articleID=212903437&dept_url=/architect/
To compile, change /usr/local/cuda to where CUDA is installed
The following assumes CUDA is in /usr/local/cuda
   echo "Please be patient ... the build process takes some time"
   (cd common ; make cuda-install=/usr/local/cuda)
   (cd cudpp ; make cuda-install=/usr/local/cuda)

# Build test program
   (cd apps/cudpp_testrig ; make cuda-install=/usr/local/cuda)

# Try some tests...
   cd bin/linux/release
   echo "test a single scan"
   ./cudpp_testrig --scan --iterations=100 --n=100000
   echo "Uncomment the following to test everything (takes a long time)"
   #./cudpp_testrig -all

Note that CUDPP is incompatible with CUDA 2.1, which is currently the default download for many operating systems. This will be addressed with the release of CUDA 2.2. Until CUDA 2.2 is released, please use CUDA 2.0 with CUDPP. For more detailed information, see the thread on this topic in the Google CUDPP group.

simpleCUDPP: A CUDPP "Hello World" Program

Now let's take a look at building and running a simple CUDPP program, simpleCUDPP, which is the CUDPP variant of a C-programmer's "Hello World" program. Since simpleCUDPP is a test program, it relies on CUDA_SAFE_CALL to check that the CUDA calls return without error. This means we need to build the debug version of CUDPP and the simpleCUDPP test, which requires adding dbg=1 to the make commands as highlighted in bold in the script below:

# Compile for debug mode
# If needed, change /usr/local/cuda to the CUDA installation directory
# This script assumes CUDA is installed in /usr/local/cuda
   echo "Please be patient ... the make process takes some time"
   (cd cudpp_1.0a/common ; make dbg=1 cuda-install=/usr/local/cuda)
   (cd cudpp_1.0a/cudpp ; make dbg=1 cuda-install=/usr/local/cuda)

# Build test program
   (cd cudpp_1.0a/apps/simpleCUDPP ;
      make dbg=1 cuda-install=/usr/local/cuda)

The simpleCUDPP executable is created in the debug executable directory. To run this example, use the command:

# Run it
   ./cudpp_1.0a/bin/linux/debug/simpleCUDPP

For better error handling -- especially for production codes -- I recommend using cudaError_t and cudaGetLastError as described in Part 3 of this series ("Error Handling and Global Memory Performance Limitations").

Sample Code Walkthrough

The main function in simpleCUDPP.cu is runTest(), which initializes the CUDA device and then declares the number of elements plus the array size for the arrays used in scan. It allocates the host-side (CPU-side) input array, h_idata, and initializes the data with random values between 0 and 15.
void runTest( int argc, char** argv)
{
    CUT_DEVICE_INIT();
    unsigned int numElements = 32768;
    unsigned int memSize = sizeof( float) * numElements;
    // allocate host memory
    float* h_idata = (float*) malloc( memSize);
    // initialize the memory
    for (unsigned int i = 0; i < numElements; ++i)
    {
        h_idata[i] = (float)(rand() & 0xf);
    }

    After the input data is created on the host, the device (GPU) array d_idata is allocated on the GPU and the
    input data from the host is copied to the device using cudaMemcpy(). A device array is allocated for the
    output results, d_odata. (A general rule of thumb, the function random() is preferred over rand() because
    random() will generate more "random" random numbers. For simpleCUDPP, the use of rand() will not
    affect the results.)

    // allocate device memory
    float* d_idata;
    CUDA_SAFE_CALL( cudaMalloc( (void**) &d_idata, memSize));
    // copy host memory to device
    CUDA_SAFE_CALL( cudaMemcpy( d_idata, h_idata, memSize,
    cudaMemcpyHostToDevice));
    // allocate device memory for result
    float* d_odata;
    CUDA_SAFE_CALL( cudaMalloc( (void**) &d_odata, memSize));

**CUDPP Plans**

Next, CUDPP has to be configured to run efficiently on the GPU. Configuration of algorithms in CUDPP
relies on the concept of the plan. A plan is a data structure that maintains intermediate storage for the
algorithm, as well as information that CUDPP may use to optimize execution of the destination hardware.
When invoked using cudppPlan(), the CUDPP planner uses the configuration details passed to it to
generate an internal plan object. A CUDPPHandle (an opaque pointer type that is used to refer to the plan
object) is returned that will be passed to other CUDPP functions to execute algorithms on (and optimized
for) the destination GPU for the specified problem characteristics.

The use of a configuration plan appears to be a useful and common pattern to configure general-purpose
CUDA codes (among others) for individual problems and destination hardware platforms.

A plan is a simple configuration mechanism that specifies the best "plan" of execution for a particular
algorithm given a specified problem size, data type and destination hardware platform. The advantage of
this approach is that once the user creates a plan, the plan object contains whatever state is needed to
execute the plan multiple times without recalculation of the configuration. The NVIDIA cuFFT library, for
example, uses this configuration model because different kinds of FFTs require different thread
configurations and GPU resources, plus plans are a simple way to store and reuse these configurations. In
addition, cuFFT optimizations can also be applied depending on if the requested FFT is a power-of-two.
The highly popular FFTW project also uses the concept of a plan. FFTW is extensively used on a variety of
platforms. For these and many other reasons, plans are a useful tool to consider when developing general-
purpose solutions that also need to run on a number of GPU architectures.

The simpleCUDPP example needs to create a plan for a forward exclusive float sum-scan of numElements
elements on the destination GPU. This is accomplished by filling out a CUDPPConfiguration struct and
passing it to the planner. In this case the planner is told about the algorithm (CUDPP_SCAN), datatype
(CUDPP_FLOAT), operation (CUDPP_ADD), and options (CUDPP_OPTION_FORWARD,
CUDPP_OPTION_EXCLUSIVE). The method cudppPlan is then called with this configuration along
with the maximum number of elements to scan, numElements. Finally, the planner is told that we only wish
to scan a one-dimensional array by passing 1 and 0 for the numRows and rowPitch parameters. The CUDPP
documentation provides more details on the parameters to `cudppPlan()`.

```
CUDPPConfiguration config;
config.op = CUDPP_ADD;
config.datatype = CUDPP_FLOAT;
config.algorithm = CUDPP_SCAN;
config.options = CUDPP_OPTION_FORWARD | CUDPP_OPTION_EXCLUSIVE;
```

```
CUDPPHandle scanplan = 0;
CUDPPResult result = cudppPlan(&scanplan, config, numElements, 1, 0);
```

A successful call to `cudppPlan` returns a handle (a pointer) to the plan object in `scanplan`. CUDPP is then put to work by invoking `cudppScan()`, which is passed the plan handle, the output and input device arrays, and the number of elements to scan.

```
// Run the scan
cudppScan (scanplan, d_odata, d_idata, numElements);
```

Next, `cudaMemcpy` is used to copy the results of the scan from `d_odata` back to the host. The GPU result is verified by computing a reference solution on the CPU (via `computeSumScanGold()`), and compare the CPU and GPU results for correctness.

```
// allocate mem for the result on host side
float* h_odata = (float*) malloc( memSize);
// copy result from device to host
CUDA_SAFE_CALL( cudaMemcpy( h_odata, d_odata, memSize,
                             cudaMemcpyDeviceToHost ) );
// compute reference solution
float* reference = (float*) malloc( memSize);
computeSumScanGold( reference, h_idata, numElements, config);
// check result
CUTBoolean res = cutComparef( reference, h_odata, numElements);
printf( "Test %s
", (1 == res) ? "PASSED" : "FAILED" );
```

Finally, `cudppDestroyPlan()` is called to clean up the memory used for our plan object. The host then frees local and device arrays using `free()` and `cudaFree`, respectively and exits the application because simpleCUDPP is finished.

```
result = cudppDestroyPlan (scanplan);
if (CUDPP_SUCCESS != result)
{
    printf("Error destroying CUDPPPlan\n");
    exit(-1);
}
```

### Sparse Matrix-Vector Multiply

CUDPP contains many other powerful capabilities not discussed in this article. For example, a simple test code to demonstrate using CUDPP for sparse matrix vector multiply is `sptest.cu`. Just download it at http://www.nada.kth.se/~tomaso/gpu08/sptest.cu. You can compile and run it with the following:
# nvcc -I cudpp_1.0a/cudpp/include -o sptest sptest.cu \
   -L cudpp_1.0a/lib -lcudpp
# ./sptest

For More Information

Check at the following locations for more examples and deeper discussions:

- [CUDPP homepage](#)
- [Google CUDPP group](#)
- [GPGPU.org](#).
- [CUDA, Supercomputing for the Masses: Part 9](#)
- [CUDA, Supercomputing for the Masses: Part 8](#)
- [CUDA, Supercomputing for the Masses: Part 7](#)
- [CUDA, Supercomputing for the Masses: Part 6](#)
- [CUDA, Supercomputing for the Masses: Part 5](#)
- [CUDA, Supercomputing for the Masses: Part 4](#)
- [CUDA, Supercomputing for the Masses: Part 3](#)
- [CUDA, Supercomputing for the Masses: Part 2](#)
- [CUDA, Supercomputing for the Masses: Part 1](#)
CUDA, Supercomputing for the Masses: Part 11

Revisiting CUDA memory spaces

By Rob Farber, Dr. Dobb's Journal
Mar 18, 2009
URL: http://www.ddj.com/hpc-high-performance-computing/215900921

Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com.

In CUDA, Supercomputing for the Masses: Part 10 of this article series on CUDA (short for "Compute Unified Device Architecture"), I examined CUDPP, the "CUDA Data Parallel Primitives Library." In this installment, I revisit local and constant memory and introduce the concept of "texture memory."

Optimizing the performance of CUDA applications most often involves optimizing data accesses which includes the appropriate use of the various CUDA memory spaces. Texture memory provides a surprising aggregation of capabilities including the ability to cache global memory (separate from register, global, and shared memory) and dedicated interpolation hardware separate from the thread processors. Texture memory also provides a way to interact with the display capabilities of the GPU. Texture memory is an extensive and evolving topic that will be introduced here and discussed more in a dedicated follow-on article. Part 4 of this series introduced the CUDA memory model and illustrated the various CUDA memory spaces with the schematic in Figure 1. Each of these memory spaces has certain performance characteristics and restrictions.
Figure 1: CUDA memory spaces.

Appropriate use of these memory spaces can have significant performance implications for CUDA applications. Table 1 summarizes characteristics of the various CUDA memory spaces. Part 5 of this series discusses CUDA memory spaces (with the exception of texture memory) in greater detail and includes performance cautions.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>On-chip</td>
<td>No</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Shared</td>
<td>Off-chip</td>
<td>N/A</td>
<td>Read/write</td>
<td>All threads in a block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read (CUDA 2.1 and previous)</td>
<td>All threads + host</td>
</tr>
</tbody>
</table>

Table 1: Characteristics of the various CUDA memory spaces.

Local Memory

Local memory is a memory abstraction that implies "local in the scope of each thread". It is not an actual hardware component of the multi-processor. In actuality, local memory resides in global memory allocated by the compiler and delivers the same performance as any other global memory region. Normally, automatic variables declared in a kernel reside in registers, which provide very fast access. Unfortunately, the relationship between automatic variables and local memory continues to be a source of confusion for CUDA programmers. The compiler might choose to place automatic variables in local memory when:

- There are too many register variables.
- A structure would consume too much register space.
- The compiler cannot determine if an array is indexed with constant quantities. Please note that registers are not addressable so an array has to go into local memory -- even if it is a two-element array -- when the addressing of the array is not known at compile time.
For additional discussion, please see "Register/Local Memory Cautions" in Part 5 of this series, "The CUDA Memory Model" in Part 4, and the CUDA programming guide on the NVIDIA CUDA Zone Documentation site.

**Constant Memory**

Constant memory is read only from kernels and is hardware optimized for the case when all threads read the same location. Amazingly, constant memory provides one cycle of latency when there is a cache hit even though constant memory resides in device memory (DRAM). If threads read from multiple locations, the accesses are serialized. The constant cache is written to only by the host (not the device because it is constant!) with `cudaMemcpyToSymbol` and is persistent across kernel calls within the same application. Up to 64KB of data can be placed in constant cache and there is 8 KB of cache for each multiprocessor. Access to data in constant memory can range from one cycle for in cache data to hundreds of cycles depending on cache locality. The first access to constant memory often does not generate a cache "miss" due to pre-fetching.

**Texture Memory in CUDA**

Graphics processors provide texture memory to accelerate frequently performed operations such as mapping, or deforming, a 2D "skin" onto a 3D polygonal model. Tricks like those in Figure 2 let low-resolution game objects appear as visually richer objects with greater complexity and detail. To remain competitive, graphics card manufacturers added the extra hardware (e.g., texture units) to make these graphics operations occur very fast. CUDA provides mechanisms so C-programmers can exploit some of the added capabilities of the texture unit hardware on CUDA-enabled devices. (For more information on the use of textures in graphics, please see the free online **GPU Gems** books. A good place to start is Chapter 37 of **GPU Gems 2**.)

**Figure 2:** Texture memory in action.

The easiest way to think of texture memory is as an alternative memory access path that the CUDA programmer can bind to regions of the GPU device memory (e.g., global memory). Texture references can be bound to the same, overlapping or different textures in memory. Each on-chip texture unit has some internal memory that buffers data from global memory. For this reason, texture memory can be used as a relaxed mechanism for the thread processors to access global memory because the coalescing requirements discussed in previous articles do not apply to texture memory accesses. This is particularly useful when targeting previous generation CUDA-capable GPUs but may not matter as much with the relaxed coalescing requirements in newer hardware.

Since optimized data access is very important to GPU performance, the use of texture memory can (in the right circumstances) provide a large performance increase. The best performance will be achieved when the threads of a warp read locations that are close together from a spatial locality perspective. CUDA provides 1D, 2D and 3D fetch capabilities using texture memory. Since a texture performs a read operation from global memory only when there is a cache miss it is possible to exceed the maximum theoretical memory bandwidth of the underlying global memory through the judicious use of the texture memory cache. This makes the texture fetch (`texfetch`) rate the more useful metric for analyzing kernel performance when using textures. For example, Patrick LeGresley notes in slide 32 of **High Performance Computing with CUDA** that...
the G80 architecture can provide approximately 18 billion fetches per second.

Especially consider using textures when:

- There is locality of reference so caching in texture memory can help.
- Use of the texture cache can reduce the penalty for nearly coalesced accesses (such as a misaligned starting address) that cannot be made fully coalesced.

Each of the previous is problem dependent so some testing is required. Some authors have reported certain texture memory access patterns are many times faster than others (For one example, see Kipton Barro's presentation CUDA Tricks and Computational Physics).

Other performance benefits of texture memory (over global and constant memory) include:

- Packed data may be broadcast to separate variables in a single operation.
- 8-bit and 16-bit integer input data may be optionally converted to 32-bit floating-point values in the range \([0.0, 1.0]\) or \([-1.0, 1.0]\) by the texture unit hardware.
- Linear, bilinear, and tri-linear interpolation using dedicated hardware separate from the thread processors.

Texture memory provides many more capabilities beyond those mentioned in this first introduction. A follow-on article will discuss texture memory further. Until the next article, please look to the NVIDIA_CUDA_SDK projects folder for examples. The Internet also contains many more useful examples as well that you can download and try. Following are two possibilities:

- A potentially helpful CUDA 3D texture example is located in the cookbook examples at the Google cudaiap2009 "cuda@mit" site.
- The CIRL fuzzy logic tutorial.

An excellent resource to learn more about the texture cache and other methods for data reuse on GPUs is Mark Silberstein's paper Efficient Computation of Sum-products on GPUs Through Software Managed Cache.

Summary

All of the above (and the previous articles in this series) indicate that CUDA developers must have a firm grasp of CUDA memory spaces. Specifically, be aware that local and global memory are not cached and their access latencies are high. Access latencies as reported by David Kirk and Wen-mei Hwu in The CUDA Memory Model with additional API and Tools info are:

- Register - dedicated HW - single cycle
- Shared Memory - dedicated HW - single cycle
- Local Memory - DRAM, no cache - *slow*
- Global Memory - DRAM, no cache - *slow*
- Constant Memory - DRAM, cached, one to hundreds of cycles depending on cache locality
- Texture Memory - DRAM, cached, hundreds of cycles
- Instruction Memory (invisible) - DRAM, cached

Consider a typical CUDA template as:

- Split a task into subtasks
- Divide input data into chunks that fit into registers and shared memory
- Load a data chunk from global memory into registers and shared memory
- Each data chunk is processed by a thread block
- Copy results back to global memory

However texture and constant memory are cached and can significantly increase application performance -- due to their access characteristics -- depending on the application access patterns:

- R/O no structure → constant memory
- R/O array structured → texture memory (CUDA 2.1 and previous)
• R/W shared within block → shared memory
• R/W registers spill to local memory and may provide a surprise slowdown
• R/W inputs/results → global memory

For More Information

• CUDA, Supercomputing for the Masses: Part 10
• CUDA, Supercomputing for the Masses: Part 9
• CUDA, Supercomputing for the Masses: Part 8
• CUDA, Supercomputing for the Masses: Part 7
• CUDA, Supercomputing for the Masses: Part 6
• CUDA, Supercomputing for the Masses: Part 5
• CUDA, Supercomputing for the Masses: Part 4
• CUDA, Supercomputing for the Masses: Part 3
• CUDA, Supercomputing for the Masses: Part 2
• CUDA, Supercomputing for the Masses: Part 1
CUDA, Supercomputing for the Masses: Part 12

CUDA 2.2 Changes the Data Movement Paradigm

By Rob Farber, Dr. Dobb's Journal
May 14, 2009
URL: http://www.ddj.com/hpc-high-performance-computing/217500110

Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com.

In CUDA, Supercomputing for the Masses: Part 11 of this article series on CUDA, I revisited CUDA memory spaces and introduced the concept of "texture memory". In this installment, I discuss some paradigm changing features of the just released CUDA version 2.2 -- namely the introduction of "mapped" pinned system memory that allows compute kernels to share host system memory and provides zero-copy support for direct access to host system memory when running on many newer CUDA-enabled graphics processors. The next article in this series will resume the discussion of texture memory and include information about new CUDA 2.2 features such as the ability to write to global memory on the GPU that has a texture bound to it. (Go here for more on CUDA 2.2.)

Prior to CUDA 2.2, CUDA kernels could not access host system memory directly. For that reason, CUDA programmers used the design pattern introduced in Part 1 and Part 2:

1. Move data to the GPU.
2. Perform calculation on GPU.
3. Move result(s) from the GPU to host.

This paradigm has now changed as CUDA 2.2 has introduced new APIs that allow host memory to be mapped into device memory via a new function called cudaHostAlloc (or cuMemHostAlloc in the CUDA driver API). This new memory type supports the following features:

- "Portable" pinned buffers that are available to all GPUs.
  - The use of multiple GPUs will be discussed in a future article.

- "Mapped" pinned buffers that map host memory into the CUDA address space and provide asynchronous transparent access to the data without requiring an explicit programmer initiated copy.
  - Integrated GPUs share physical memory with the host processor (as opposed to the on-board fast global memory of discrete GPUs). Mapped pinned buffers act as "zero-copy" buffers for many newer (especially integrated graphics processors) because they avoid superfluous copies. When developing code for integrated GPUs, using mapped pinned memory really makes sense.
  - For discrete GPUs, mapped pinned memory is only a performance win in certain cases. Since the memory is not cached by the GPU:
    - It should be read or written exactly once.
    - The global loads and stores that read or write the memory must be coalesced to avoid a
2x-7x PCIe performance penalty.
- At best, it will only deliver PCIe bandwidth performance, but this can be 2x faster than cudaMemcpy because mapped memory is able exploit the full duplex capability of the PCIe bus by reading and writing at the same time. A call to cudaMemcpy can only move data in one direction at a time (i.e., half duplex).

Further, a drawback of the current CUDA 2.2 release is that all pinned allocations are mapped into the GPU's 32-bit linear address space, regardless of whether the device pointer is needed or not. (NVIDIA indicates this will be changed to a per-allocation basis in a later release.)

- "WC" (write-combined) memory can provide higher performance:
  - Since WC memory is neither cached or cache coherent, greater PCIe performance can be achieved because the memory is not snooped during transfers across the PCI Express bus. NVIDIA notes in their "CUDA 2.2 Pinned Memory APIs" document that WC memory may perform as much as 40% faster on certain PCI Express 2.0 implementations.
  - It may increase the host processor(s) write performance to host memory because individual writes are first combined (via an internal processor write-buffer) so that only a single burst write containing many aggregated individual writes need be issued. (Intel claims they have observed actual performance increases of over 10x but this is not typical). For more information, please see the Intel publication Write Combining Memory Implementation Guidelines.
  - Host-side calculations and applications may run faster because write-combined memory does not pollute the internal processor caches such as the L1 and L2 caches. This happens because WC does not enforce cache coherency, which can increase host processor efficiency by reducing cache misses as well as avoiding the overhead incurred when enforcing cache coherency. Write-combining also avoids cache pollution by utilizing a separate dedicated internal write-buffer cache, which by-passes and leaves the other internal processor caches untouched.
  - WC memory does have drawbacks and CUDA programmers should not consider a WC memory region as general-purpose memory because it is "weakly-ordered". In other words, reading from a WC memory location may return unexpected -- and incorrect -- data because a previous write to that memory location might have been delayed in order to combine it with other writes. Without programmer enforced coherency though a "fence" operation, it is possible that a read of WC memory may actually "read" old or even initialized data.

  - Unfortunately, enforcing coherent reads from WC memory may incur a performance penalty on some host processor architectures. Happily, processors with the SSE4 instruction set provide a streaming load instruction (MOVNTDQA) that can efficiently read from WC memory. (Check if the CPUID instruction is executed with EAX==1, bit 19 of ECX, to see if SSE4.1 is available.) Please see the Intel publication, Increasing Memory Throughput With Intel Streaming SIMD Extensions 4 (Intel SSE4) Streaming Load.
  - It is unclear if and when a CUDA programmer needs to take any action (such as using a memory fence) to ensure that the WC memory is in-place and ready for use by the host or graphics processor(s). The Intel documentation states that "[a] 'memory fence' instruction should be used to properly ensure consistency between the data producer and data consumer." The CUDA driver does use WC memory internally and must issue a store fence instruction whenever it sends a command to the GPU. For this reason, the NVIDIA documentation notes, "the application may not have to use store fences at all!" (emphasis added). A rough rule of thumb that appears to work is to look to the CUDA commands prior to referencing WC memory and assume they issue a fence instruction. Otherwise, utilize your compiler intrinsic operations to issue a store fence instruction and guarantee that every preceding store is globally visible. This is compiler dependent. Linux compilers will probably understand the _mm_sfence intrinsic while Windows compilers will probably use _WriteBarrier.

Each of these memory features can be used individually or in any combination -- you can allocate a
portable, write-combined buffer, a portable pinned buffer, a write-combined buffer that is neither portable nor pinned, or any other permutation enabled by the flags.

In a nutshell, these new features add convenience and performance while conversely adding complexity and creating version dependencies on the CUDA driver, the CUDA hardware and the host processors. However, many types of applications can benefit from these new features.

<table>
<thead>
<tr>
<th>Type</th>
<th>Discrete GPU Benefit</th>
<th>Integrated GPU Benefit</th>
<th>Can</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portable</td>
<td>Multiple GPUs can access a single pinned region.</td>
<td>Same as Discrete</td>
<td>Adds software version dependency</td>
</tr>
<tr>
<td>Mapped</td>
<td>Adds transparent, asynchronous I/O capability.</td>
<td>Functions as zero-copy buffer</td>
<td>Add software version and GPU dependency</td>
</tr>
<tr>
<td></td>
<td>Extends GPU memory by adding host memory (PCIe bandwidth limited) to GPU address space</td>
<td></td>
<td>Must be fully coalesced or 2x-7x performance will result</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Mapped memory provides up to 2x the PCIe bandwidth at best</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Current GPUs use 32-bit pointers, so each thread can use a maximum of 4GB of memory (on-board DRAM plus mapped CPU memory)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Note: The CUDA 2.2 documentation indicates all pinned memory is mapped to CPU address space when cudaDeviceMapHost is specified</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Accessing CPU written data on the host may require synchronization</td>
</tr>
<tr>
<td>WC</td>
<td>Potential 40% increase in PCIe bandwidth.</td>
<td>Same as Discrete</td>
<td>Adds software version and GPU dependency. add host processor dependency (to use efficiently). Unclear if pinning is required</td>
</tr>
<tr>
<td></td>
<td>Potential host-side performance boost</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following source listing for `incrementMappedArrayInPlace.cu` is an adapted version of the `incrementArrays.cu` example from Part 2 to use the new mapped, pinned runtime API.

```c
// incrementMappedArrayInPlace.cu
#include <stdio.h>
#include <assert.h>
#include <cuda.h>

// define the problem and block size
#define NUMBER_OF_ARRAY_ELEMENTS 100000
#define N_THREADS_PER_BLOCK 256

void incrementArrayOnHost(float *a, int N)
{
    int i;
    for (i=0; i < N; i++) a[i] = a[i]+1.f;
}

__global__ void incrementArrayOnDevice(float *a, int N)
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx < N) a[idx] = a[idx]+1.f;
}

void checkCUDAError(const char *msg)
{
    cudaError_t err = cudaGetLastError();
    if( cudaSuccess != err )
```
int main(void)
{
    float *a_m; // pointer to host memory
    float *a_d; // pointer to mapped device memory
    float *check_h; // pointer to host memory used to check results
    int i, N = NUMBER_OF_ARRAY_ELEMENTS;
    size_t size = N*sizeof(float);
    cudaDeviceProp deviceProp;

    #if CUDART_VERSION < 2020
    #error "This CUDART version does not support mapped memory!\n"
    #endif

    // Get properties and verify device 0 supports mapped memory
    cudaGetDeviceProperties(&deviceProp, 0);
    checkCUDAError("cudaGetDeviceProperties");

    if(!deviceProp.canMapHostMemory) {
        fprintf(stderr, "Device %d cannot map host memory!\n", 0);
        exit(EXIT_FAILURE);
    }

    // set the device flags for mapping host memory
    cudaSetDeviceFlags(cudaDeviceMapHost);
    checkCUDAError("cudaSetDeviceFlags");

    // allocate mapped arrays
    cudaHostAlloc((void **)&a_m, size, cudaHostAllocMapped);
    checkCUDAError("cudaHostAllocMapped");

    // Get the device pointers to the mapped memory
    cudaHostGetDevicePointer((void **)&a_d, (void *)a_m, 0);
    checkCUDAError("cudaHostGetDevicePointer");

    // initialization of host data
    for (i=0; i<N; i++) a_m[i] = (float)i;

    // do calculation on device:
    // Part 1 of 2. Compute execution configuration
    int blockSize = N_THREADS_PER_BLOCK;
    int nBlocks = N/blockSize + (N%blockSize > 0?1:0);

    // Part 2 of 2. Call incrementArrayOnDevice kernel
    incrementArrayOnDevice <<< nBlocks, blockSize >>> (a_d, N);
    checkCUDAError("incrementArrayOnDevice");

    /* Note the allocation, initialization and call to incrementArrayOnHost
       occurs asynchronously to the GPU */
    check_h = (float *)malloc(size);
    for (i=0; i<N; i++) check_h[i] = (float)i;
    incrementArrayOnHost(check_h, N);

    // Make certain that all threads are idle before proceeding
    cudaThreadSynchronize();
    checkCUDAError("cudaThreadSynchronize");
// check results
for (i=0; i<N; i++) assert(check_h[i] == a_m[i]);

// cleanup
free(check_h); // free host memory
cudaFreeHost(a_m); // free mapped memory (and device pointers)
}

CUDA 2.2 added the following two device properties to the `cudaDeviceProp` structure that is retrieved by `cudaGetDeviceProperties` so you can determine if a device can support the new mapped memory API (as well as check if the GPU is an integrated graphics processor):

<table>
<thead>
<tr>
<th>int integrated;</th>
<th>Nonzero if the device is integrated with the host memory system. This structure member corresponds to the driver API's <code>CU_DEVICE_ATTRIBUTE_INTEGRATED</code> query.</th>
</tr>
</thead>
<tbody>
<tr>
<td>int canMapHostMemory;</td>
<td>Nonzero if the device can map host memory. This structure member corresponds to the driver API's <code>CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY</code> query.</td>
</tr>
</tbody>
</table>

The following code block utilizes a pre-processor check to make certain that a valid version of CUDA is being used to compile the mapped code plus the function `cudaGetDeviceProperties` is called so a runtime check can be made to ensure that the CUDA device supports mapped memory:

```c
#if CUDART_VERSION < 2020
#error "This CUDART version does not support mapped memory!"\n#endif

// Get properties and verify device 0 supports mapped memory
cudaGetDeviceProperties(&deviceProp, 0);
checkCUDAError("cudaGetDeviceProperties");

if(!deviceProp.canMapHostMemory) {
    fprintf(stderr, "Device %d cannot map host memory!\n", 0);
    exit(EXIT_FAILURE);
}
```

Host memory mapping is then enabled on the device:

```c
// set the device flags for mapping host memory
cudaSetDeviceFlags(cudaDeviceMapHost);
checkCUDAError("cudaSetDeviceFlags");
```

A mapped array, `a_m`, is then allocated on the host. (Note: The memory is mapped at this point but there is no device pointer. Getting the device pointer occurs in the following step.)

```c
// allocate host mapped arrays
cudaHostAlloc((void **)&a_m, size, cudaMemcpyHostMapped);
checkCUDAError("cudaHostAllocMapped");
```

Get the device pointer to the mapped memory:

```c
// Get the device pointers to the mapped memory
cudaHostGetDevicePointer((void **)&a_d, (void *)&a_m, 0);
checkCUDAError("cudaHostGetDevicePointer");
```
Data initialization occurs and the kernel is executed on the GPU. Unlike the original incrementArrays.cu example, no explicit programmer initiated data movement occurs with a cudaMemcpy. Note that the data movement and kernel execution occurs asynchronously to the host operations. As a result, the host creation and calculation of the the validation array, check_h, occurs while the GPU is simultaneously running the incrementArrayOnDevice kernel to update the host array a_m through the mapped device memory pointer a_d. 

Synchronization occurs via the call to cudaMemcpy after which the GPU results are validated against the host generated results.

Assuming the results from the host and GPU kernels agree, the program then cleans up after itself. The function cudaMemcpy is used to free up the mapped array on the host and pointer on the GPU.

Under Linux, the program can be compiled with the command-line:

```
nvcc "o incrementMappedArrayInPlace incrementMappedArrayInPlace.cu
```

The performance implications of performing in-place updates to mapped memory are not clear. To ensure the minimum number of PCIe operations occur, it seems prudent to stream data between separate arrays. In other words, use separate arrays where one is dedicated read operations and the other is dedicated to write operations.

### Demonstrating write-combining

The following program, incrementMappedArrayWC.cu, demonstrates the use of separate write-combined, mapped, pinned memory to increment the elements of an array by one. This required changing incrementArrayOnHost and incrementArrayOnDevice to read from array a and write to array b. In this way, coherency issues are avoided and streaming performance should be achieved. The cudaMemcpyWriteCombined flag was also added to the cudaMemcpy calls. We rely on the CUDA calls to the driver to issue the appropriate fence operation to ensure the writes become globally visible.

```c
#include <stdio.h>
#include <assert.h>
#include <cuda.h>

#define NUMBER_OF_ARRAY_ELEMENTS 100000
#define N_THREADS_PER_BLOCK 256

void incrementArrayOnHost(float *b, float *a, int N)
{
    int i;
    for (i=0; i < N; i++) b[i] = a[i]+1.f;
}

__global__ void incrementArrayOnDevice(float *b, float *a, int N)
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx < N) b[idx] = a[idx]+1.f;
}

void checkCUDAError(const char *msg)
{
    cudaError_t err = cudaGetLastError();
    if( cudaSuccess != err ) {
        printf("Error: %s\n", msg);
        exit(1);
    }
}
```

// incrementMappedArrayWC.cu
#include <stdio.h>
#include <assert.h>
#include <cuda.h>

// define the problem and block size
#define NUMBER_OF_ARRAY_ELEMENTS 100000
#define N_THREADS_PER_BLOCK 256

void incrementArrayOnHost(float *b, float *a, int N)
{
    int i;
    for (i=0; i < N; i++) b[i] = a[i]+1.f;
}

__global__ void incrementArrayOnDevice(float *b, float *a, int N)
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx < N) b[idx] = a[idx]+1.f;
}

void checkCUDAError(const char *msg)
{
    cudaError_t err = cudaGetLastError();
    if( cudaSuccess != err ) {
        printf("Error: %s\n", msg);
        exit(1);
    }
}
```c
int main(void)
{
  float *a_m, *b_m; // pointers to mapped host memory
  float *a_d, *b_d; // pointers to mapped device memory
  float *check_h;   // pointer to host memory used to check results
  int i, N = NUMBER_OF_ARRAY_ELEMENTS;
  size_t size = N*sizeof(float);
  cudaDeviceProp deviceProp;

#if CUDART_VERSION < 2020
#error "This CUDART version does not support mapped memory!"
#endif

  // Get properties and verify device 0 supports mapped memory
  cudaGetDeviceProperties(&deviceProp, 0);
  checkCUDAError("cudaGetDeviceProperties");

  if(!deviceProp.canMapHostMemory) {
    fprintf(stderr, "Device %d cannot map host memory!
", 0);
    exit(EXIT_FAILURE);
  }

  // set the device flags for mapping host memory
  cudaSetDeviceFlags(cudaDeviceMapHost);
  checkCUDAError("cudaSetDeviceFlags");

  // allocate host mapped arrays
  int flags = cudaHostAllocMapped|cudaHostAllocWriteCombined;
  cudaHostAlloc((void **)&a_m, size, flags);
  cudaHostAlloc((void **)&b_m, size, flags);
  checkCUDAError("cudaHostAllocMapped");

  // Get the device pointers to memory mapped
  cudaHostGetDevicePointer((void **)&a_d, (void *)a_m, 0);
  cudaHostGetDevicePointer((void **)&b_d, (void *)b_m, 0);
  checkCUDAError("cudaHostGetDevicePointer");

  /* initialization of the mapped data. Since a_m is write-combined,
   * it is not guaranteed to be initialized until a fence operation is
   * called. In this case that should happen when the kernel is
   * invoked on the GPU */
  for (i=0; i<N; i++) a_m[i] = (float)i;

  // do calculation on device:
  // Part 1 of 2. Compute execution configuration
  int blockSize = N_THREADS_PER_BLOCK;
  int nBlocks = N/blockSize + (N%blockSize > 0?1:0);

  // Part 2 of 2. Call incrementArrayOnDevice kernel
  incrementArrayOnDevice <<< nBlocks, blockSize >>> (b_d, a_d, N);
  checkCUDAError("incrementArrayOnDevice");

  // Note the allocation and call to incrementArrayOnHost occurs
  // asynchronously to the GPU
  check_h = (float *)malloc(size);
  incrementArrayOnHost(check_h, a_m, N);
}
```

// Make certain that all threads are idle before proceeding
cudaThreadSynchronize();
checkCUDAError("cudaThreadSynchronize");

// cudaThreadSynchronize() should have caused an sfence
// to be issued, which will guarantee that all writes are done

// check results. Note: the updated array is in b_m, not b_d
for (i=0; i<N; i++) assert(check_h[i] == b_m[i]);

// cleanup
free(check_h);

// free mapped memory (and device pointers)
cudaFreeHost(a_m);
cudaFreeHost(b_m);
}

Conclusion

CUDA 2.2 changes the data movement paradigm by providing APIs for mapped, transparent data transfers between the host and GPU(s). These APIs also allow the CUDA programmer to make data sharing between the host and graphics processor(s) more efficient by exploiting asynchronous operation, full-duplex PCIe data transfers, through the use of write combined memory, and by adding the ability for the programmer to share pinned memory with multiple GPUs.

Personally, I have used these APIs as a convenience when porting existing scientific codes onto the GPU because mapped memory allows me to keep the host and device data synchronized while I incrementally move as much of the calculation onto the GPU as possible. This allows me to verify my results after each change to ensure nothing has broken, which can be a real time and frustration saver when working with complex codes with many inter-dependencies. Additionally, I also use these APIs to increase efficiency by exploiting asynchronous host and multiple GPU calculations plus full-duplex PCIe transfers and other nice features of the CUDA 2.2 release.

I also see the new CUDA 2.2 APIs facilitating the development of entirely new classes of applications ranging from operating systems to real-time systems.

One example is the RAID research performed by scientists at the University of Alabama and Sandia National Laboratory that transformed CUDA-enabled GPUs into high-performance RAID accelerators that can calculate Reed-Solomon codes in real-time for high-throughput disk subsystems (see Accelerating Reed-Solomon Coding in RAID Systems with GPUs, by Matthew Curry, Lee Ward, Tony Skjellum, Ron Brightwell). From their abstract, "Performance results show that the GPU can outperform a modern CPU on this problem by an order of magnitude and also confirm that a GPU can be used to support a system with at least three parity disks with no performance penalty".

My guess is we will see a CUDA-enhanced Linux md (multiple device or software RAID) driver sometime in the near future. Imagine the freedom of not being locked into a proprietary RAID controller. If something breaks, just connect your RAID array to another Linux box to access the data. If that computer does not have an NVIDIA GPU then just use the standard Linux software md driver to access the data.

Don't forget that CUDA-enabled devices can accelerate and run multiple applications at the same time. An upcoming article demonstrating how to incorporate graphics and CUDA will exploit that capability. Until then, try running a separate graphics application while running one of your CUDA applications. I think you will be surprised at how well both applications will perform.

For More Information