CUDA, Supercomputing for the Masses: Part 10

CUDPP, a powerful data-parallel CUDA library

January 29, 2009
URL:http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/212903437

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* $\oplus$ *and an array of `n` elements*:

given: [a0, a1, ..., an-1],
returns:[a0, (a0$\oplus$ a1), ..., (a0 $\oplus$ a1 $\oplus$ ... $\oplus$ (an-1)].

`Example`: *If* $\oplus$ *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.

```
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. 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:

```
# 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 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);
    // initalize 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);

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

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\n", (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

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.