CUDA, Supercomputing for the Masses: Part 9

Extending High-level Languages with CUDA

November 01, 2008
URL:http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/211800683

In CUDA, Supercomputing for the Masses: Part 8 of this article series on CUDA (short for "Computer 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.

[Click image to view at full size]

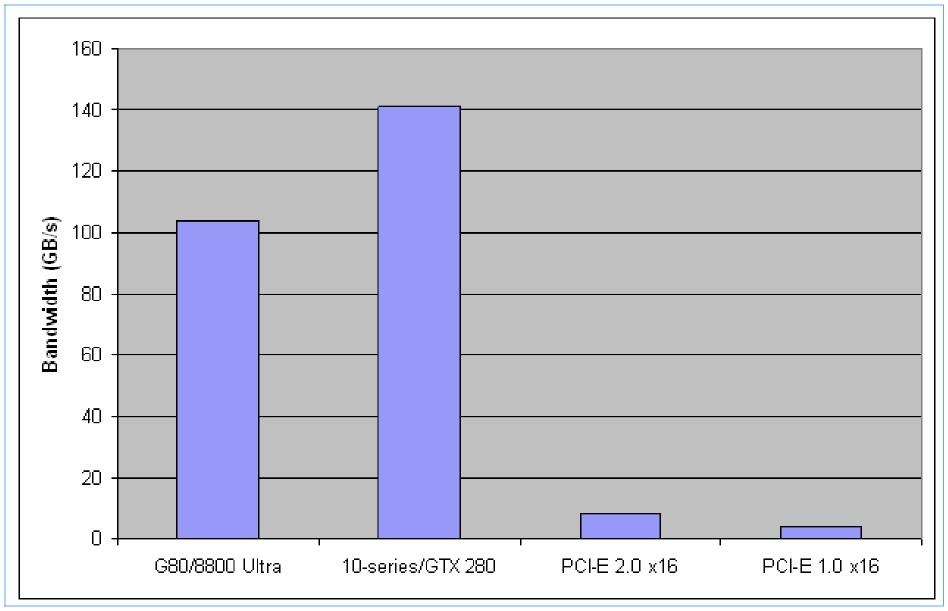

Figure 1: Bandwidth available to the thread-processors from various sources (e.g., global memory and the PCI-E bus) for several architectures.

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 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 (Java Native Interface). One example project to get you going is JCublas, 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, although CPAN is the canonical PERL repository.
- Python: An overview of extending Python with C (and hence CUDA) is a good place to start, as is SWIG. A working Python example (that operates on NumPy arrays) is pystream and the related Project GPUlib.

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 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, 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:

```
#! /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])

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

Within the exponentiationTest.py, a custom module is imported with the line:

```
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:

```
# 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:

```
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:

```
# 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 comparision as seen below. (Numpy is an excellent numerical Python package that has matrix operations.

```
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:

```
%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));
  $2 = (float *)(((PyArrayObject *)PyList_GetItem($input,1))->data);
  $3 = ((PyArrayObject *)PyList_GetItem($input,1))->dimensions[0];
}
```

```
extern void matrixMulLoop(int steps, float *u, int n);

%{
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();
%}
```

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);
}
%}
```

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 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

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.