Quad Doubles on a GPU

1. Floating-Point Arithmetic
   - floating-point numbers
   - quad double arithmetic
   - quad doubles for use in CUDA programs

2. Quad Double Square Roots
   - quad double arithmetic on a GPU
   - a kernel using `gqd_real`
   - performance considerations

MCS 572 Lecture 37
Introduction to Supercomputing
Jan Verschelde, 16 November 2016

Quad Doubles on a GPU

1. Floating-Point Arithmetic
   - floating-point numbers
   - quad double arithmetic
   - quad doubles for use in CUDA programs

2. Quad Double Square Roots
   - quad double arithmetic on a GPU
   - a kernel using `gqd_real`
   - performance considerations

floating-point numbers

A floating-point number consists of a sign bit, exponent, and a fraction (also known as the mantissa).

Almost all microprocessors follow the IEEE 754 standard.

GPU hardware supports 32-bit (single float)
and for compute capability $\geq$ 1.3 also double floats.

Numerical analysis studies algorithms for continuous problems, investigating

- problems for their sensitivity to errors in the input; and
- algorithms for their propagation of roundoff errors.

parallel numerical algorithms

The floating-point addition is *not* associative!

Parallel algorithms compute and accumulate the results in an order that is different from their sequential versions.

Example: Adding a sequence of numbers is more accurate if the numbers are sorted in increasing order.

Instead of speedup, we can ask questions about quality up:

- If we can afford to keep the total running time constant, does a faster computer give us more accurate results?
- How many more processors do we need to guarantee a result?

Quad Doubles on a GPU

1. Floating-Point Arithmetic
   - floating-point numbers
   - quad double arithmetic
   - quad doubles for use in CUDA programs

2. Quad Double Square Roots
   - quad double arithmetic on a GPU
   - a kernel using `gqd_real`
   - performance considerations

quadruple precision

A quad double is an unevaluated sum of 4 doubles,
improves working precision from 2.2×10^{-16} to 2.4×10^{-63}.

Y. Hida, X.S. Li, and D.H. Bailey: **Algorithms for quad-double precision floating point arithmetic.** In *15th IEEE Symposium on Computer Arithmetic* pages 155–162. IEEE, 2001. Software at `http://crd.lbl.gov/~dhbailey/mpdist`.

A quad double builds on `double double`, some features:

- The least significant part of a `double double` can be interpreted as a compensation for the roundoff error.
- Predictable overhead: working with `double double` is of the same cost as working with complex numbers.

Newton's method for $\sqrt{x}$

```
#include <iostream>
#include <iomanip>
#include <qd/qd_real.h>
using namespace std;

qd_real newton ( qd_real x )
{
   qd_real y = x;
   for(int i=0; i<10; i++)
      y -= (y*y - x)/(2.0*y);
   return y;
}
```

the main program

```
int main ( int argc, char *argv[] )
{
   cout << "give x : ";
   qd_real x; cin >> x;
   cout << setprecision(64);
   cout << "        x : " << x << endl;

   qd_real y = newton(x);
   cout << "  sqrt(x) : " << y << endl;

   qd_real z = y*y;
   cout << "sqrt(x)^2 : " << z << endl;

   return 0;
}
```

the makefile

If the program is on file `newton4sqrt.cpp`
and the makefile contains

```
QD_ROOT=/usr/local/qd-2.3.17
QD_LIB=/usr/local/lib

newton4sqrt:
        g++ -I$(QD_ROOT)/include newton4sqrt.cpp \
              $(QD_LIB)/libqd.a -o /tmp/newton4sqrt
```

then we can create the executable as

```
$ make newton4sqrt
g++ -I/usr/local/qd-2.3.17/include newton4sqrt.cpp \
      /usr/local/lib/libqd.a -o /tmp/newton4sqrt
$
```

Quad Doubles on a GPU

1. Floating-Point Arithmetic
   - floating-point numbers
   - quad double arithmetic
   - quad doubles for use in CUDA programs

2. Quad Double Square Roots
   - quad double arithmetic on a GPU
   - a kernel using `gqd_real`
   - performance considerations

extended precision on the GPU

Large problems often need extra precision.

The QD library has been ported to the GPU.

Mian Lu, Bingsheng He, and Qiong Luo: **Supporting extended precision on graphics processors.** In A. Ailamaki and P.A. Boncz, editors, *Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010), June 7, 2010, Indianapolis, Indiana*, pages 19–26, 2010.
Software at `http://code.google.com/p/gpuprec/`, and at `https://github.com/lumianph/gpuprec/tree/master/gqd`.

Installed on `kepler` and `pascal` in `/usr/local/gqd_1_2`.

For graphics cards of compute capability < 1.3, one could use the freely available Cg software of Andrew Thall to achieve double precision using float-float arithmetic.

gqd_reals are of double4 type

```
#include "gqd_type.h"
#include "vector_types.h"
#include <qd/qd_real.h>

void qd2gqd ( qd_real *a, gqd_real *b )
{
   b->x = a->x[0];
   b->y = a->x[1];
   b->z = a->x[2];
   b->w = a->x[3];
}

void gqd2qd ( gqd_real *a, qd_real *b )
{
   b->x[0] = a->x;
   b->x[1] = a->y;
   b->x[2] = a->z;
   b->x[3] = a->w;
}
```

a first kernel

```
#include "gqd.cu"

__global__ void testdiv2 ( gqd_real *x, gqd_real *y )
{
   *y = *x/2.0;
}

int divide_by_two ( gqd_real *x, gqd_real *y )
{
   gqd_real *xdevice;
   size_t s = sizeof(gqd_real);
   cudaMalloc((void**)&xdevice,s);
   cudaMemcpy(xdevice,x,s,cudaMemcpyHostToDevice);
   gqd_real *ydevice;
   cudaMalloc((void**)&ydevice,s);
   testdiv2<<<1,1>>>(xdevice,ydevice);
   cudaMemcpy(y,ydevice,s,cudaMemcpyDeviceToHost);
   return 0;
}
```

testing the first kernel

```
#include <iostream>
#include <iomanip>
#include "gqd_type.h"
#include "first_gqd_kernel.h"
#include "gqd_qd_util.h"
#include <qd/qd_real.h>
using namespace std;

int main ( int argc, char *argv[] )
{
   qd_real qd_x = qd_real::_pi;
   gqd_real x;
   qd2gqd(&qd_x,&x);
   gqd_real y;

   cout << " x : " << setprecision(64) << qd_x << endl;
```

test program continued

```
   int fail = divide_by_two(&x,&y);

   qd_real qd_y;
   gqd2qd(&y,&qd_y);

   if(fail == 0) cout << " y : " << qd_y << endl;

   cout << "2y : " << 2.0*qd_y << endl;

   return 0;
}
```

the makefile

```
QD_ROOT=/usr/local/qd-2.3.17
QD_LIB=/usr/local/lib
GQD_HOME=/usr/local/gqd_1_2
SDK_HOME=/usr/local/cuda/sdk

test_pi2_gqd_kernel:
	@-echo ">>> compiling kernel ..."
	nvcc -I$(GQD_HOME)/inc -I$(SDK_HOME)/C/common/inc \
	     -c first_gqd_kernel.cu
	@-echo ">>> compiling utilities ..."
	g++ -I/usr/local/cuda/include -I$(GQD_HOME)/inc \
	    -I$(QD_ROOT)/include -c gqd_qd_util.cpp
	@-echo ">>> compiling test program ..."
	g++ test_pi2_gqd_kernel.cpp -c \
	    -I/usr/local/cuda/include -I$(GQD_HOME)/inc \
	    -I$(QD_ROOT)/include
	@-echo ">>> linking ..."
	g++ -I$(GQD_HOME)/inc -I$(QD_ROOT)/include \
	     first_gqd_kernel.o test_pi2_gqd_kernel.o gqd_qd_util.o \
	     $(QD_LIB)/libqd.a \
	     -o /tmp/test_pi2_gqd_kernel \
	    -lcuda -lcutil_x86_64 -lcudart \
	    -L/usr/local/cuda/lib64 -L$(SDK_HOME)/C/lib
```

compiling and running

```
$ make test_pi2_gqd_kernel
>>> compiling kernel ...
nvcc -I/usr/local/gqd_1_2/inc -I/usr/local/cuda/sdk/C/common/inc \
             -c first_gqd_kernel.cu
>>> compiling utilities ...
g++ -I/usr/local/cuda/include -I/usr/local/gqd_1_2/inc \
            -I/usr/local/qd-2.3.17/include -c gqd_qd_util.cpp
>>> compiling test program ...
g++ test_pi2_gqd_kernel.cpp -c \
            -I/usr/local/cuda/include -I/usr/local/gqd_1_2/inc \
            -I/usr/local/qd-2.3.17/include
>>> linking ...
g++ -I/usr/local/gqd_1_2/inc -I/usr/local/qd-2.3.17/include \
             first_gqd_kernel.o test_pi2_gqd_kernel.o gqd_qd_util.o \
             /usr/local/lib/libqd.a \
             -o /tmp/test_pi2_gqd_kernel \
            -lcuda -lcutil_x86_64 -lcudart \
            -L/usr/local/cuda/lib64 -L/usr/local/cuda/sdk/C/lib
$ /tmp/test_pi2_gqd_kernel
 x : 3.1415926535897932384626433832795028841971693993751058209749445923e+00
 y : 1.5707963267948966192313216916397514420985846996875529104874722961e+00
2y : 3.1415926535897932384626433832795028841971693993751058209749445923e+00
$
```

Quad Doubles on a GPU

1. Floating-Point Arithmetic
   - floating-point numbers
   - quad double arithmetic
   - quad doubles for use in CUDA programs

2. Quad Double Square Roots
   - quad double arithmetic on a GPU
   - a kernel using `gqd_real`
   - performance considerations

quad double arithmetic on a GPU

Recall our first CUDA program to take the square root of complex numbers stored in a `double2` array.

In using quad doubles on a GPU, we have 3 stages:

1. The kernel in a file with extension `cu` is compiled with `nvcc -c` into an object file.
2. The application code is compiled with `g++ -c`.
3. The linker `g++` takes `.o` files and libraries on input to make an executable file.

Working without a makefile now becomes very tedious.

the makefile

```
QD_ROOT=/usr/local/qd-2.3.17
QD_LIB=/usr/local/lib
GQD_HOME=/usr/local/gqd_1_2
SDK_HOME=/usr/local/cuda/sdk

sqrt_gqd_kernel:
        @-echo ">>> compiling kernel ..."
        nvcc -I$(GQD_HOME)/inc -I$(SDK_HOME)/C/common/inc \
             -c sqrt_gqd_kernel.cu
        @-echo ">>> compiling utilities ..."
        g++ -I/usr/local/cuda/include -I$(GQD_HOME)/inc \
            -I$(QD_ROOT)/include -c gqd_qd_util.cpp
        @-echo ">>> compiling test program ..."
        g++ run_sqrt_gqd_kernel.cpp -c \
            -I/usr/local/cuda/include -I$(GQD_HOME)/inc \
            -I$(QD_ROOT)/include
        @-echo ">>> linking ..."
        g++ -I$(GQD_HOME)/inc -I$(QD_ROOT)/include \
             sqrt_gqd_kernel.o run_sqrt_gqd_kernel.o gqd_qd_util.o \
             $(QD_LIB)/libqd.a \
             -o /tmp/run_sqrt_gqd_kernel \
            -lcuda -lcutil_x86_64 -lcudart \
            -L/usr/local/cuda/lib64 -L$(SDK_HOME)/C/lib
```

running make

```
$ make sqrt_gqd_kernel
>>> compiling kernel ...
nvcc -I/usr/local/gqd_1_2/inc -I/usr/local/cuda/sdk/C/common/inc \
             -c sqrt_gqd_kernel.cu
>>> compiling utilities ...
g++ -I/usr/local/cuda/include -I/usr/local/gqd_1_2/inc \
            -I/usr/local/qd-2.3.17/include -c gqd_qd_util.cpp
>>> compiling test program ...
g++ run_sqrt_gqd_kernel.cpp -c \
            -I/usr/local/cuda/include -I/usr/local/gqd_1_2/inc \
            -I/usr/local/qd-2.3.17/include
>>> linking ...
g++ -I/usr/local/gqd_1_2/inc -I/usr/local/qd-2.3.17/include \
             sqrt_gqd_kernel.o run_sqrt_gqd_kernel.o gqd_qd_util.o \
             /usr/local/lib/libqd.a \
             -o /tmp/run_sqrt_gqd_kernel \
            -lcuda -lcutil_x86_64 -lcudart \
            -L/usr/local/cuda/lib64 -L/usr/local/cuda/sdk/C/lib
$
```

Quad Doubles on a GPU

1. Floating-Point Arithmetic
   - floating-point numbers
   - quad double arithmetic
   - quad doubles for use in CUDA programs

2. Quad Double Square Roots
   - quad double arithmetic on a GPU
   - a kernel using `gqd_real`
   - performance considerations

a kernel using gqd_real

```
#include "gqd.cu"

__global__ void sqrtNewton ( gqd_real *x, gqd_real *y )
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   gqd_real c = x[i];
   gqd_real r = c;
   for(int j=0; j<10; j++)
      r = r - (r*r - c)/(2.0*r);
   y[i] = r;
}
```

the file `sqrt_gqd_kernel.cu` continued

```
int sqrt_by_Newton ( int n, gqd_real *x, gqd_real *y )
{
   gqd_real *xdevice;
   size_t s = n*sizeof(gqd_real);
   cudaMalloc((void**)&xdevice,s);
   cudaMemcpy(xdevice,x,s,cudaMemcpyHostToDevice);

   gqd_real *ydevice;
   cudaMalloc((void**)&ydevice,s);

   sqrtNewton<<<n/32,32>>>(xdevice,ydevice);

   cudaMemcpy(y,ydevice,s,cudaMemcpyDeviceToHost);

   return 0;
}
```

the main program

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include "gqd_type.h"
#include "sqrt_gqd_kernel.h"
#include "gqd_qd_util.h"
#include <qd/qd_real.h>
using namespace std;

int main ( int argc, char *argv[] )
{
   const int n = 256;
   gqd_real *x = (gqd_real*)calloc(n,sizeof(gqd_real));
   gqd_real *y = (gqd_real*)calloc(n,sizeof(gqd_real));
```

run_sqrt_gqd_kernel.cpp continued

```
   for(int i = 0; i<n; i++)
   {
      x[i].x = (double) (i+2);
      x[i].y = 0.0; x[i].z = 0.0; x[i].w = 0.0;
   }
   int fail = sqrt_by_Newton(n,x,y);
   if(fail == 0)
   {
      const int k = 24;
      qd_real qd_x;
      gqd2qd(&x[k],&qd_x);
      qd_real qd_y;
      gqd2qd(&y[k],&qd_y);
      cout << "     x     : " << setprecision(64) << qd_x << endl;
      cout << "sqrt(x)    : " << setprecision(64) << qd_y << endl;
      cout << "sqrt(x)^2  : " << setprecision(64) << qd_y*qd_y
           << endl;
   }
   return 0;
}
```

Quad Doubles on a GPU

1. Floating-Point Arithmetic
   - floating-point numbers
   - quad double arithmetic
   - quad doubles for use in CUDA programs

2. Quad Double Square Roots
   - quad double arithmetic on a GPU
   - a kernel using `gqd_real`
   - performance considerations

sequential and interval memory layout

Consider four quad doubles a, b, c, and d.

Stored in a sequential memory layout:

a_0	a_1	a_2	a_3	b_0	b_1	b_2	b_3	c_0	c_1	c_2	c_3	d_0	d_1	d_2	d_3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Stored in an interval memory layout:

a_0	b_0	c_0	d_0	a_1	b_1	c_1	d_1	a_2	b_2	c_2	d_2	a_3	b_3	c_3	d_3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The implementation with an interval memory layout is reported to be three times faster over the sequential memory layout.

Bibliography

- A. Thall. **Extended-Precision Floating-Point Numbers for GPU Computation.** Software available at `andrewthall.org`.
- Y. Hida, X.S. Li, and D.H. Bailey. **Algorithms for quad-double precision floating point arithmetic.** In *15th IEEE Symposium on Computer Arithmetic*, pages 155–162. IEEE, 2001.
  Software at `http://crd.lbl.gov/~dhbailey/mpdist`.
- M. Lu, B. He, and Q. Luo. **Supporting extended precision on graphics processors.** In A. Ailamaki and P.A. Boncz, editors, *Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010), June 7, 2010, Indianapolis, Indiana*, pages 19–26, 2010.
  Software at `http://code.google.com/p/gpuprec/` and at `https://github.com/lumianph/gpuprec/tree/master/gqd`.

summary and exercises

Chapter 7 in the book of Kirk & Hwu provides some background. The application of quad double arithmetic is an illustration of combined usage of `nvcc` and `g++` to compile and link several libraries.

Some exercises:

1. Compare the performance of the CUDA program for Newton's method for square root with quad doubles to the code of lecture 29.
2. Extend the code so it works for complex quad double arithmetic.
3. Use quad doubles to implement the second parallel sum algorithm of lecture 33. Could the parallel implementation with quad doubles run as fast as sequential code with doubles?
4. Consider the program to approximate π of lecture 13. Write a version for the GPU and compare the performance with the multicore version of lecture 13.