Quadruple Precision BLAS Subroutines on GPUs

Feb. 3, 2012
Daichi MUKUNOKI
(Advisor: Prof. Daisuke Takahashi)
Graduate School of Systems and Information Engineering, University of Tsukuba, JAPAN
Outline

- Background
- Motivation & Goal
- Double-double type quadruple precision operations
- Implementation of quadruple precision BLAS on GPUs
- Performance on Tesla C2050
- Triple precision?
- Conclusion
Background

- **Demand for quadruple precision operations**
  - Need a quadruple precision to compute ill conditioned problems
  - In large-scale computing, an accumulation of round-off error may become more serious

- **Double-double (DD) type quadruple precision** [Dekker1971]
  - Use two double precision floating-point value to represent one quadruple precision value
  - Quadruple precision arithmetic library:
    - e.g. DDFUN90, QD ...
  - High precision BLAS using DD-type operations:
    - XBLAS, MBLAS

- **Quadruple precision BLAS is not implemented on GPUs**
Motivation & Goal

Motivation

- Quadruple precision operation is highly compute-intensive operation
- Quadruple precision linear algebra operations are suitable for GPU acceleration

Our goal

- To implement fast quadruple precision BLAS on GPUs using DD-type operations
- To evaluate the performance of three different levels of BLAS subroutines
  - Level1 BLAS: AXPY ($y=ax+y$)
  - Level2 BLAS: GEMV ($y=\alpha A x+\beta y$)
  - Level3 BLAS: GEMM ($C=\alpha AB+\beta C$)
Double-double (DD) type quadruple precision value

- One quadruple precision value $a$ is represented using two double precision value $a_{hi}$ and $a_{lo}$

$$a = (a_{hi}, a_{lo}) \quad (|a_{lo}| \leq 0.5\text{ulp}(a_{hi}))$$

† ulp: unit of least precision of floating-point value

DD-type quadruple precision number

- Exponent (11 bits)
- Significand (52+52=104 bits)

Double precision number

- Sign (1 bit)
- Exponent (11 bits)
- Significand (52 bits)

- Sign (1 bit)
- Exponent (11 bits)
- Significand (52 bits)
DD-type Operations (2/2)

- **DD-type quadruple precision arithmetic operations**
  - Can compute using only double precision floating-point arithmetic operations
  - Used the same algorithms as QD library [Hida et al.]

**DD-type multiplication**

\[(c_{hi} + c_{lo}) = (a_{hi} + a_{lo}) \times (b_{hi} + b_{lo})\]

\[c_{hi} = (a_{hi} \times b_{hi})\]

\[c_{lo} = e + (a_{hi} \times b_{lo}) + (a_{lo} \times b_{hi})\]

**DDMul (from QD [Hida et al.])**

```c
DDMul(ah, al, bh, bl){
    p = ah*bh;    // (1)
    e = fma(ah*bh-ch);  // (2)
    t = (ah*bl)+e;   // (3)
    e = (bh*al)+t;   // (4)
    ch = p+e;       // (5)
    cl = e-(ch-p);  // (6)
    return(ch, cl);
}
```

† **e**: round-off error of \((a_{hi} \times b_{hi})\) calculated by an error-free floating-point arithmetic algorithm [Dekker1971]

**DD-type Operations (2/2)**

- DD-type multiplication

**Example**

```c
chi = ah \times bh

e = err(ah \times bh)

t = (ah \times bl) + e

clo = (bh \times al) + t
```

**Normalize**

```
\[\text{DDMul(from QD [Hida et al.])}\]
```
Cost of DD-type Operations

- Number of instructions for DD-type operations
  - Algorithms from QD library [Hida et al.]
  - Can use one FMA instruction for DD-type multiplication

<table>
<thead>
<tr>
<th># of Double Precision Instructions</th>
<th>Add/Sub</th>
<th>Mul</th>
<th>FMA</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double Prec. MulAdd (a*b+c)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DD-type Add (a+b)</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>DD-type Mul (a*b)</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>DD-type MulAdd (a*b+c)</td>
<td>16</td>
<td>3</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

- On MulAdd operation, the computation cost of DD-type operation is 20x more than double precision operation
  - AXPY, GEMV, GEMM consist mainly of MulAdd
Theoretical Peak Performance on GPUs

Performance of MulAdd on Tesla C2050

- **DDFlops**: DD-type floating point operations per second
- 1 double prec. instruction requires 2 cycles on Tesla C2050
- **Double prec**: \(1.15 \text{ [GHz]} \times 14 \text{ [SMs]} \times 32 \text{ [CUDA Cores]} \times \left(\frac{2 \text{ [Flop]}}{(1 \text{ [instruction]} \times 2 \text{ [cycles]})}\right) = 515.2 \text{ [GFlops]}\)
- **Quadruple prec**: \(1.15 \text{ [GHz]} \times 14 \text{ [SMs]} \times 32 \text{ [CUDA Cores]} \times \left(\frac{2 \text{ [DDFlop]}}{(20 \text{ [instructions]} \times 2 \text{ [cycles]})}\right) = 25.76 \text{ [GDDFlops]}\)
  - \(25.76 \text{ [GDDFlops]} \times 21 \text{ [Flop] / 2 \text{ [DDFlop]} = 270.5 \text{ [GFlops]}}\)
  - *this is because 19/20 instructions are NOT FMA instruction…*
Implemented three BLAS subroutines
- AXPY (y=ax+y), GEMV (y=\(\alpha\)Ax+\(\beta\)y), GEMM (C=\(\alpha\)AB+\(\beta\)C)

Implementation is similar to “doublecomplex” kernel
- DD-type value is stored in “double2” type vector value
- DD-type operations are implemented as a device function with “\_\_forceinlined\_\_” (no function call overhead)
- Each thread performs one DD-type operation

```cpp
__device__ __forceinline__ void
CUDDMul (double2 &a, double2 &b, double2 &c) {
    double2 t;
    CUTwoProdFMA (a.x, b.x, t.x, t.y);
    t.y = __dadd_rn(t.y, __dadd_rn(__dmul_rn(a.x, b.y), __dmul_rn(a.y, b.x)));
    CUQuickTwoSum (t.x, t.y, c.x, c.y);
}
```
Implementation (2/2)

- Implementation of BLAS kernels
  - Used blocking for shared memory (GEMV & GEMM)
  - Experimentally determined the optimal blocking size and the number of threads
Performance Evaluation

- **Environment**
  - CPU: Xeon E5630*2 (2.53GHz, Quad-Cores, 2 sockets)
  - GPU: Tesla C2050 (ECC-enabled)
  - CentOS 6.0, CUDA ver. 4.0, gcc 4.4.4 (-O3)

- **Methodology**
  - Measured the performance using **DDFlops**
  - Not including the time of PCIe communications
  - Also implemented and evaluated the CPU version
    - Using QD library 2.3.11 for quadruple precision operations
    - Performed in multi-threads using OpenMP on 8 cores
    - Faster than MBLAS’s quadruple precision subroutines
**GEMM:** \( C = \alpha AB + \beta C \)

---

### Performance

- **N=2,048:**
  - Compute-bound on GPU (due to Byte/Flop ratio)
  - **22 GDDFlops**
    - \( = 231 \text{ GFlops of double} \)
  - **86% of theoretical peak**
  - **21x faster than CPUs**
    - \((4\text{cores} \times 2)\)
  - **13x slower than DGEMM**
    - This is because, DGEMM is only 58% of theoretical peak

- **Related work:** 23 GDDFlops on TeslaC2050 [Nakata2011]
**AXPY:** \( y = \alpha x + y \)

### Performance

<table>
<thead>
<tr>
<th>( N )</th>
<th>GDDFlops</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.4</td>
</tr>
<tr>
<td>10240</td>
<td>1.7</td>
</tr>
<tr>
<td>10240000</td>
<td>4.4</td>
</tr>
</tbody>
</table>

- Memory-bound on GPU (due to Byte/Flop ratio)
- 4.4 GDDFlops
- 17% of theoretical peak
- 6x faster than CPUs (4cores*2)
- 2x slower than DAXPY

### Execution Time (on Tesla C2050)

<table>
<thead>
<tr>
<th>( N )</th>
<th>Relative Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>1.0</td>
</tr>
<tr>
<td>10240</td>
<td>1.4</td>
</tr>
<tr>
<td>10240000</td>
<td>2.0</td>
</tr>
</tbody>
</table>

- Quadruple
- Double

---

**Note:** The graphs illustrate the performance and execution time of the AXPY operation on Xeon E5630*2 and C2050, highlighting the memory-bound nature of the operation and its comparison to CPUs and DAXPY on Tesla C2050.
GEMV: $y = \alpha Ax + \beta y$

**Performance**

- $N=8,192$: 
  - Memory-bound on GPU (due to Byte/Flop ratio)
  - 12 GDDFlops
  - 45% of theoretical peak
  - 15x faster than CPUs (4cores*2)
  - 2x slower than DGEMV

**Execution Time (on Tesla C2050)**

- Relative Execution Time
- Quadruple
- Double
- (CUBLAS4.0)
Conclusion

- Implemented and evaluated DD-type quadruple precision BLAS subroutines on Tesla C2050
- Computation cost of DD-type quadruple precision operation is $20 \times$ more than double precision in theory (on MulAdd)
- Actual execution time is only $2 \times$ more (AXPY & GEMV) $\sim 13 \times$ more (GEMM) than double precision subroutines of CUBLAS
- Quadruple precision BLAS can be accelerated using GPUs
  - On Tesla C2050, $6 \times$ faster (AXPY), $15 \times$ faster (GEMV), $21 \times$ faster (GEMM) than that on Xeon E5630*2 (4*2cores)
- Quadruple precision linear algebra operations are suitable for GPU acceleration
Triple Precision BLAS?
Why triple precision?

- Triple precision is clearly effective in cases where quadruple precision is not required but double precision is insufficient.
- In such cases, triple precision can save memory space & can save wasted bandwidth.
  - Quadruple precision AXPY & GEMV are memory-bound on GPUs: performance is close to 1/2 of double prec.
  - On memory-bound operations, triple precision can achieve 2/3 of double precision performance?

Triple precision has never been implemented on modern processors such as GPUs and x86 CPUs.
D+S-type Triple Precision

Double+Single (D+S) type triple precision value

- Stored one triple precision value in one double precision value & one single precision value (similar to DD-type)

D+S-type triple precision number

<table>
<thead>
<tr>
<th>Sign (1 bit)</th>
<th>Exponent (8 bits)</th>
<th>Significand (52+23=75 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

† Exponent is 8 bits: size of exponent depends on lower part’s exponent

Double+Single (D+S) type triple precision operations

- D+S-type operations require a lot of typecastings between single and double precision
- D+S-type is slower than DD-type on Tesla C2050 (also GeForce series) in theory and in practice
Implemented “triple precision interface” for quadruple precision BLAS

- Input/output data are D+S-type triple precision, but operations are DD-type quadruple precision
- Implementation techniques are almost the same as quadruple precision subroutines
- One triple precision value is 12-Bytes
  - To fulfill the 128-Bytes memory alignment on CUDA, we used structure-of-arrays (SOA) layout instead of array-of-structures (AOS)
  - SOA is up to 1.3x faster than AOS
Triple precision AXPY & GEMV are memory-bound, each execution time is approx. $1.6x$ and $1.7x$ more than double prec.

On memory-bound operations, execution time of triple precision is close to $3/4$ of quadruple precision subroutines.
No advantage to use triple precision interface on compute-bound operations

Triple precision is a little slower than quadruple
(but it is the same performance as quadruple in theory...)
We implemented triple precision interface for DD-type quadruple precision BLAS subroutines

Triple prec. AXPY & GEMV are memory-bound on Tesla C2050

- On memory-bound operations, execution time of triple precision subroutine is close to $\frac{3}{4}$ of quadruple precision

Triple precision interface is available for memory-bound operations, in cases where quadruple precision is not required, but double precision is not sufficient

**Future work:**

- To develop fast triple precision operation algorithm (also for quadruple precision)
- To apply triple & quadruple precision operations to actual scientific computation
**Performance of GESV (AX=B)**

- **TeslaC2050**
  - **SP**
  - **MP (SP+DP)**
  - **DP**
  - **MP(DP+QP)**
  - **QP (CPU)**

- **GeForceGTX580**
  - **SP**
  - **MP (SP+DP)**
  - **DP**
  - **MP(DP+QP)**
  - **QP (CPU)**

- **A=N*N, B=N*1 (NRHS=1)**
- QP is performed on CPU with 1 thread using MLAPACK (not optimized)
- Input matrices are initialized using “dlapackf77_dlarnv” (random)
- # of iteration: MP(SP+DP): 3~7, MP(DP+QD): 2
- Error (||b-Ax||/||A||): QP: order 1E-29~31, MP(DP+QD): order 1E-31~32