High-Performance Implementation of Reproducible and Accurate Matrix-Multiplication
(Implementation and evaluation of Ozaki-scheme on GPUs)

June 27, 2018, PMAA 18 @ Zurich
Daichi Mukunoki (Tokyo Woman’s Christian University)
Roman Iakymchuk (KTH Royal Institute of Technology)
Stef Graillat (Sorbonne University)
Takeshi Ogita (Tokyo Woman’s Christian University)
Background
- Numerical computations with floating-point operations suffer from round-off errors, which impact the accuracy & reproducibility of the final result.
- This can be observed not only for ill-conditioned but also for regular problems.

Accuracy
- The accumulation of round-off errors becomes non-negligible in large-scale and long-time computations.
- 64-bit floating-point (double-prec.) may become insufficient in near future.

Reproducibility
- Round-off errors impact reproducibility as well as accuracy because the result of floating-point operations varies depending on the order of computations.
  - Factors which may change the order of computations: algorithm, number of threads/processes, use of atomic operations, and so on.
- Loss of reproducibility makes it more difficult to debug a program (e.g. when you port a program to another environment).
- Scientific experiments should be reproducible by other people.
Examples of linear algebra libraries supporting accurate and/or reproducible computations

- **Accurate:**
  - XBLAS [Li et al.]: for CPU, quadruple-precision (double-double)
  - MBLAS & MLAPACK (MPACK) [Nakata]: for CPU, based on some existing high-precision arithmetic libraries such as MPFR

- **Reproducible:**
  - Conditional Numerical Reproducible (CNR) mode in Intel MKL: for CPU

- **Accurate & Reproducible:**
  - ReproBLAS [Demmel et al.]: for CPU, accuracy is tunable
  - RARE-BLAS [Chohra et al.]: for CPU, correct rounding
  - ExBLAS [Iakymchuk et al.]: for CPU & GPU (OpenCL), correct rounding

**Next generation BLAS (BLAS G2)**

- Accurate and reproducible computations have been discussed as new features of next generation BLAS (at “Workshop on Batched, Reproducible, and Reduced Precision BLAS” in 2016 & 2017)
The goal of this study

- To develop a high-performance implementation of accurate & reproducible matrix-multiplication on GPUs and to analyze the performance
- Accurate & reproducible methods: (1) Ozaki scheme and (2) ExBLAS scheme
- Ozaki-scheme: an accurate (and reproducible) matrix-multiplication method based on DGEMM [Ozaki et al. 2012]

Motivations

- **GEMM:**
  - Level-3 BLAS, compute-intensive
  - A key kernel in scientific computations
  - Highly-optimized implementation is needed
- **GPU:**
  - A major architecture widely used in HPC systems
  - Huge computational power: suitable for compute-intensive tasks
- **Implementation of Ozaki-scheme:**
  - The first full GPU implementation
  - With some optimization techniques for GPUs
Overview

- An accurate result is computed as a summation of several multiplications of matrices which are split not to occur rounding-errors during the multiplications:

$$C = AB = (A^{(1)} + A^{(2)} + \ldots + A^{(s)}) \cdot (B^{(1)} + B^{(2)} + \ldots + B^{(t)})$$

- Matrices are split to be $|A^{(p)}(i,j)| \geq |A^{(q)}(i,j)|$, $|B^{(p)}(i,j)| \geq |B^{(q)}(i,j)|$, $p < q$
- and $\text{fl}(A^{(i)}B^{(j)}) = A^{(i)}B^{(j)}$ for any $i$ and $j$
  * $\text{fl}(\ldots)$: result of floating-point operations

- If the summation of the multiplications of the split matrices is performed with correct rounding, the final result achieves correct rounding
  - In this study, we used the NearSum algorithm [Rump et al. 2005]

- The multiplications of split matrices can be performed using DGEMM
  - This is the most time-consuming portion in this method, and highly optimized DGEMM is available on most processors including GPUs

- It consumes a lot of memory
  - It can be solved using blocking (discussed later)
Ozaki scheme (cont’d)

\[ C = AB = (A^{(1)} + A^{(2)} + \ldots + A^{(s)}) \cdot (B^{(1)} + B^{(2)} + \ldots + B^{(t)}) \]

- Matrices are split to be \(|A^{(p)}(i,j)| \geq |A^{(q)}(i,j)|, \, |B^{(p)}(i,j)| \geq |B^{(q)}(i,j)|, \, p < q \)
- and \( \text{fl}(A^{(i)}B^{(i)}) = A^{(i)}B^{(i)} \) for any \( i \) and \( j \)

1. **Split**
   - Splitting of input matrices A and B

2. **Matrix Multiplications**
   - Multiplications of the split matrices by DGEMM

3. **Summation**
   - Summation of the computed matrices by NearSum

\[ C = \text{Sum}(C^{(0)} + C^{(1)} + C^{(2)} + C^{(3)}) \]
Ozaki scheme – Matrix splitting

- Matrices are split not to occur rounding-errors during the multiplications of the split matrices
- This algorithm is the error-free transformation for vector
- Matrix can be split by applying the algorithm to a matrix towards inner-product direction
- # of splits depends on both the length of the vector and the max value

\[ C = AB = (A^{(1)} + A^{(2)} + \ldots + A^{(s)}) \cdot (B^{(1)} + B^{(2)} + \ldots + B^{(t)}) \]

\[ \text{Matrices are split to be } |A^{(p)}(i,j)| \geq |A^{(q)}(i,j)|, |B^{(p)}(i,j)| \geq |B^{(q)}(i,j)|, p < q \]

\[ \text{and } \text{fl}(A^{(i)}B^{(j)}) = A^{(i)}B^{(j)} \text{ for any } i \text{ and } j \]

\[ \begin{align*}
\text{Algorithm 1 Split vector } x &= [x_1, x_2, \ldots, x_n] \\
1: & \quad \text{function Split}(x) \\
2: & \quad \quad i = 1 \\
3: & \quad \quad \text{while } (\text{norm}(x(i),\inf) != 0) \text{ do} \\
4: & \quad \quad \quad c_1 = \text{ceil}((\log_2 u^{-1} + \log_2(n + 1))/2) \\
5: & \quad \quad \quad c_2 = \text{ceil}(\log_2 \text{max}(|x_i|)) \\
6: & \quad \quad \quad t = 2^{c_1} \cdot 2^{c_2} \\
7: & \quad \quad \quad \sigma = [t, t, \ldots, t] \\
8: & \quad \quad \quad x^{(i)} = x' = \text{fl}((x^{(i)} + \sigma) - \sigma) \\
9: & \quad \quad \quad x^{(i+1)} = \text{fl}(x' - x^{(i)}) \\
10: & \quad \quad \quad i = i + 1 \\
11: & \quad \quad \text{end while} \\
12: & \quad \text{end function}
\end{align*} \]
Ozaki scheme – GPU implementation

**Overview**
- Full CUDA implementation
- Interface compatible with cublasDgemm
  - Computes double-precision matrices located on GPU memory
  - But currently $\alpha$ & $\beta$ are not supported (just compute $C=AB$)

**Challenges**
- To improve performance
- To reduce memory consumption
  - GPUs have limited memory space

**Techniques we applied**
- Blocking (for memory and performance)
- Utilizing batched BLAS (for performance)

```c
__host__ int exblasExdgemm (  
cublasHandle_t ch,  
char tranA, char tranB,  
int m, int n, int k,  
double *alpha,  
double *devA, int lda,  
double *devB, int ldb,  
double *beta,  
double *devC, int ldc) {  
...
  Split (devA, devAsplit);  
  Split (devB, devBsplit);  
...
  Compute (devAsplit,  
    devBsplit,  
    devCsplit);  
...
  NearSum (devCsplit, devC);  
...  
}
```
Ozaki scheme – Blocking

Memory consumption:
When matrices A and B are \(n \times n\) square matrices, the naïve implementation consumes \((s+t+st)n^2\) extra memory space

\[
C = \text{Sum}(C^{(0)} + C^{(1)} + C^{(2)} + C^{(3)})
\]
**Ozaki scheme – Blocking (cont’d)**

### Memory consumption (with blocking):

When block-size=$b$, # of splits of mat-$A$ =$s_i=s$ $(1< i \leq m/b)$, and # of splits of mat-$B$ =$t_j=t$ $(1< j \leq n/b)$, the required memory space decreases from $(s+t+st)n^2$ to $(s+t)nb+stb^2$

*Note: $s_i$ and $t_j$ may vary depending on the max value in each block, thus there is a possibility to reduce # of splits compared to the non-blocking case*
A negative side-effect:
If the block-size is quite small, the throughput of GEMM may decrease as problem size becomes too small to utilize all the cores on a GPU.
A negative side-effect:
If the block-size is quite small, the throughput of GEMM may decrease as problem size becomes too small to utilize all the cores on a GPU

A solution: Batched BLAS
- A new BLAS interface that computes multiple independent BLAS operations as a single task
- cuBLAS (intel MKL as well) provides a batched DGEMM routine, which computes multiple small DGEMMs concurrently at a high speed

```
cublasDgemmBatched (cublasHandle_t handle, 
cublasOperation_t transa, cublasOperation_t transb, 
int m, int n, int k, const double *alpha, 
const double *Aarray[], int lda, 
const double *Barray[], int ldb, 
const double *beta, double *Carray[], int ldc, 
int batchCount)
```

\[
C = AB = (A^{(1)} + A^{(2)} + \ldots + A^{(s)}) \cdot (B^{(1)} + B^{(2)} + \ldots + B^{(t)})
\]
Evaluation

- Evaluation
  - Performance (effectiveness of blocking & batched BLAS)
  - Demonstrations of accuracy

- Environment
  - GPU: NVIDIA Titan V (Volta architecture)
    - 7449.6 GFlops on double-precision, 16GB memory
  - CUDA 9.1, driver version: 390.67

- Problem setting
  - Matrices are square & initialized as \((\text{rand}(n)-0.5) \times \exp(\phi \times \text{randn}(n))\)
    - The larger \(\phi\) is, the wider the range of the absolute values in the matrix becomes
      (and thus, # of splits and DGEMMs also increase)

- Theoretical / expected performance
  - Cost: the computation by DGEMM is \(O(n^3)\), the other parts are \(O(n^2)\)
  - We assume that the expected performance is the performance based on the cost for \(st\) times of non-block & non-batched DGEMM (where # of splits for mat A & B are \(s\) & \(t\), respectively)
Performance ($\phi=0$)

**Input:** $(\text{rand}(n)-0.5) \times \exp(\phi \times \text{randn}(n))$

"Expected" performance: The performance based on the cost for $st$ times of non-block & non-batched DGEMM

- $s$: # of splits of mat A
- $t$: # of splits of mat B

In this case, $s=2$ & $t=2$: the expected overhead is 4x of DGEMM

- In this case ($\phi=0$, # of splits = 2), the effectiveness of batched BLAS is a little (# of batched tasks is too few)
Performance ($\phi=0$)

Input: $(\text{rand}(n)-0.5) \times \exp(\phi \times \text{rand}(n))$

- Problem Size (m=n=k)
- GFlops (exact−operations)
- Memory Consumption
- MBytes
- Out of memory

Performance (phi=0, TitanV)

- b=1024
- b=2048
- b=4096
- b=no

- batch,b=1024
- batch,b=2048
- batch,b=4096
- batch,b=no
• “Other” is mainly cost for allocating the memory for storing split matrices
• This cost can be ignored as you can use the allocated memory repeatedly in the case you use the routine multiple times

Input: \((\text{rand}(n)-0.5) \ast \exp(\phi \ast \text{randn}(n))\)
Input: \((\text{rand}(n)-0.5) \times \exp(\phi \times \text{randn}(n))\)

In this case (\# of splits = 5\~6, \# of batch tasks = 25\~30), even block-size \(b=512\) can achieve a competitive performance when batched BLAS is used.

"Expected" performance: The performance based on the cost for \(st\) times of non-block & non-batched DGEMM

* \(s\): \# of splits of mat A
* \(t\): \# of splits of mat B

In this case, \(s = 5\) & \(t = 5\) and 6 (\(n \geq 8192\)): the expected overhead is 25\~30x of DGEMM
**Performance (φ=1)**

**Input:** \((\text{rand}(n)-0.5) \times \exp(\phi \times \text{randn}(n))\)

---

**Performance (phi=1, TitanV)**

![Graph showing performance (GFlops) vs. problem size (MBytes)](image)

**Memory Consumption**

![Graph showing memory consumption vs. problem size (MBytes)](image)
Performance ($\phi=1$)

Input: $(\text{rand}(n)-0.5) \times \exp(\phi \times \text{randn}(n))$

Performance (phi=1, TitanV)

Breakdown

- Split
- DGEMM
- Sum
- Other

Problem size (m=n=k)

GFlops (exact operations)
Input: \((\text{rand}(n) - 0.5) \times \exp(\phi \times \text{randn}(n))\)

- \(b=4096\) (\(\phi=0\))
- Expected (\(\phi=0\))
- \(b=1024\) (\(\phi=1\))
- Expected (\(\phi=1\))
- \(b=512\) (\(\phi=2\))
- Expected (\(\phi=2\))
- \(b=512\) (\(\phi=4\))
- Expected (\(\phi=4\))
- \(b=512\) (\(\phi=8\))
- Expected (\(\phi=8\))

- \text{GFlops (exact operations)}
- \text{GFlops (double-precision, for cublasDgemm)}

Problem Size (\(m=n=k\))

- \text{Performance (TitanV)}
- \text{b: block-size}

Observed / Expected [%]
Performance ($\phi=0 \sim 8$)

Input: $(\text{rand}(n)-0.5) \times \exp(\phi \times \text{randn}(n))$

![Graph showing performance and memory consumption](image-url)
Accuracy test

Comparison with MPFR–2048bits
(matrix size: 64x64, on TitanV)

Our implementation
DGEMM (cublas)

Relative error =
\[
\frac{\max(|C_{target}[i]| - |C_{MPFR}[i]|)}{\max(|C_{MPFR}[i]|)}
\]

- Compared with MPFR (2048bits), ExDGEMM has no error for any cases (there is nothing to plot)
Further optimization (future work)

- **Blocking towards inner-product direction**
  - It may increase the chance to reduce # of splits since it depends the matrix dimension towards the inner product direction as well as the max value
  - But it increases memory consumption and # of summations to compute the final result

- **Skipping of zero calculations**
  - Split matrices which have lower digits information may include many zero elements (sparse matrix)
  - Sparse GEMM (SpMM) may be used for those computations

- **Auto-tuning**
  - For determining the optimal block-size and use/non-use of batched BLAS
  - For block-size, there is a tradeoff between performance and memory consumption
Conclusion

**Summary**
- The first full GPU implementation of accurate (correct rounding) and reproducible DGEMM with Ozaki scheme
- Implementation techniques
  - Matrix blocking for reducing memory consumption
  - Use of batched BLAS for improving performance
- More than 90% of expected performance can be achieved if the optimal block-size is used (the performance is DGEMM-performance-bound)

**Future work**
- Further optimizations
  - For small matrices
    - Automatic determining of optimal block-size
      - Blocking has a tradeoff between memory-consumption and performance
- Comparison with other methods (ReproBLAS, ExBLAS scheme, etc.)
- Implementation of Full set BLAS
High-Performance Implementation of Reproducible and Accurate Matrix-Multiplication

June 27, 2018
PMAA 18 @ Zurich
Daichi Mukunoki (Tokyo Woman’s Christian University)
Roman Iakymchuk (KTH Royal Institute of Technology)
Stef Graillat (Sorbonne University)
Takeshi Ogita (Tokyo Woman’s Christian University)
ExBLAS scheme

- **Overview**
  - A reproducible + accurate method
    - Reproducibility is achieved by computing with correct-rounding
  - Combination of Floating-Point Expansion (FPE) + Super-Accumulator (SuperAcc)

- **Floating-Point Expansion (FPE)**
  - Based on error free transformations [Dekker and Knuth]
  - Used for avoiding (reducing) the access to SuperAcc (it’s heavy)

- **Super-Accumulator (SuperAcc)**
  - A long accumulator which can cover the exponent range of double-precision
  - It consists of array of 64-bit integer (39 elements)
Further optimization

Splitting of matrix A:
• Determining the number of splitting (= finding the max number in row-direction) requires non-sequential memory access
• To avoid the performance degradation, we transposed the matrix A before splitting
• The computations of split matrices are performed using batched DGEMM-TN
• The transposition cost is negligibly small

Note: although this approach is actually effective to improve the performance of splitting of mat A, the performance of cuBLAS DGEMM-TN is unstable when the matrix size is large: the total performance decreases when this approach applies on Titan V (we didn’t apply this in our current implementation)
Accuracy test

Comparison with MPFR-2048bits
(matrix size: 64x64, on TitanV)

Relative error = max(|C_target[i] - C_MPFR[i]|)
max(|C_MPFR[i]|)

1×10⁻²₀
1×10⁻₁₅
1×10⁻₁₀
1×10⁻⁵
1
10
10⁰
10⁶
10⁹
10¹²
10¹⁵
10¹⁸

Condition number

Relative error

Comparison with MPFR (2048bits), ExDGEMM has no error for any cases
(impossible to plot the results with logscale axis)
Input: \((\text{rand}(n)-0.5) \times \exp(\phi \times \text{randn}(n))\)
Input: \((\text{rand}(n)-0.5) \times \exp(\phi \times \text{randn}(n))\)