JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 26 No. 1, pp. 1-9, January 2021
https://doi.org/10.9708/jksci.2021.26.01.001

Parallel Algorithm of Conjugate Gradient Solver using OpenGL Compute Shader

Hongly Va*, Do-keyong Lee*, Min Hong**

*Student, School of Software Convergence, Soonchunhyang University, Asan, Korea
*Student, School of Software Convergence, Soonchunhyang University, Asan, Korea
**Professor, Dept. of Software Convergence, Soonchunhyang University, Asan, Korea

[Abstract]

OpenGL compute shader is a shader stage that operate differently from other shader stage and it can be used for the calculating purpose of any data in parallel. This paper proposes a GPU-based parallel algorithm for computing sparse linear systems through conjugate gradient using an iterative method, which perform calculation on OpenGL compute shader. Basically, this sparse linear solver is used to solve large linear systems such as symmetric positive definite matrix. Four well-known matrix formats (Dense, COO, ELL and CSR) have been used for matrix storage. The performance comparison from our experimental tests using eight sparse matrices shows that GPU-based linear solving system much faster than CPU-based linear solving system with the best average computing time 0.64ms in GPU-based and 15.37ms in CPU-based.

▸ **Key words**: Linear Solving, Conjugate Gradient, Sparse Matrix, Parallel GPU, OpenGL Compute Shader

[요 약]

OpenGL compute shader는 다른 shader 단계와 다르게 동작하며, 병렬로 모든 데이터를 계산하는 데 사용할 수 있다. 본 논문은 OpenGL compute shader에서 반복 켤레 기울기 방법을 통해 희소 선형 시스템을 계산하기 위한 GPU 기반의 병렬 알고리즘 제안하였다. 제안된 희소 선형 해결 방법은 대칭인 양의 정부호 행렬과 같은 대형 선형 시스템을 해결하기 위해 사용된다. 본 논문은 이 알고리즘을 사용하여 매트릭스 형식이 다른 8가지 예제들에 대해서 CPU와 GPU를 기반으로 한 성능 비교 결과를 제공한다. 본 논문은 4가지 잘 알려져 있는 매트릭스 형식(Dense, COO, ELL and CSR)을 매트릭스 저장소를 사용하였다. 8개의 희소 매트릭스를 사용한 성능 비교 실험에서 GPU 기반 선형 해결 시스템이 CPU 기반 선형 해결 시스템보다 훨씬 빠르며, GPU 기반에서 0.64ms, CPU 기반에서 15.37ms의 평균 컴퓨팅 시간을 제공한다.

▸ **주제어**: 선형 해결, 반복 켤레 기울기법, 희소 행렬, 병렬 GPU, OpenGL Compute Shader

• First Author: Hongly Va, Corresponding Author: Min Hong
*Hongly Va (vahonglykhmer@gmail.com), School of Software Convergence, Soonchunhyang University
*Do-keyong Lee (dooky606@daum.net), School of Software Convergence, Soonchunhyang University
**Min Hong (mhong@sch.ac.kr), Dept. of Software Convergence, Soonchunhyang University
• Received: 2021. 01. 15, Revised: 2021. 01. 28, Accepted: 2021. 01. 28.

http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945

I. Introduction

One of computational science and modeling problems is system of linear equation which is the fundamental part of linear algebra of modern mathematical problems [1]. The system of equation could be expressed by $Ax = B$ where A is the coefficient matrix, B is right hand side vector and x is an unknown vector to be solved. It is very important to study classical algorithms for solving these linear system of equation and to choose which algorithm to solve the problem. Two major methods are normally used which are direct method and iterative method. In direct method, the solver algorithm tends to give the exact solution for the linear equation. On the other hand, iterative method solves a system of linear equation with an approximation solution and based on the correctness comparing to the exact solution, the process continues solving until the error rate is tidy enough [2].

Many iterative solver methods have been used for linear solving. In case the linear system is symmetric, positive and definite, we may use conjugate gradient (CG) as an iterative method for solving the system of linear equation quickly [3]. The performance of CG method can be speedup by using parallel computing in GPU which many processors simultaneously execute. Additionally, GPU-based can be use for solve large linear system faster than CPU due to many processor.

Generally, GPU (Graphic Processing Unit) was using for rendering purpose which commonly handle for computer graphics operation [4]. Beside that, GPU also provides functionality to access general computing, which called GPGPU (General Purpose Graphic Processing Unit) [5]. Since the GPU run on thousands of processing cores in massive parallelism, it's offering the efficient calculation and provide real-time processing performance. The benefit of GPGPU is suitable for large scale data with many computation operations, which using SIMD (Single Instruction Multiple Data) architecture to produce effective results in term of latency, bandwidth and memory accessing [6]. Currently, the most well-known technologies that support GPGPU programming consist of NVIDIA's CUDA, AMD's OpenCL, OpenMP and GLSL [7], which supporting for GPU-based parallel processing This makes such a possibility for a C/C++ program to utilize GPU's ability with large data in parallel.

Therefore, in this paper, we present a GPU algorithm to solve a system of linear equation by the conjugate gradient method in parallel GPU architecture which using OpenGL compute shader.

II. Preliminaries

1. Related works

GPU has been using as a parallel algorithm for decades and created tons of opportunities for research due to massive speed computation [8]. Likewise, the potential of GPU has been used for parallel conjugate gradient method to solve poisson problems [9]. In previous research from Helfenstein and Koko [10], the parallel preconditioned conjugate gradient using GPU was proposed for linear system which using symmetric, positive definite matrix and SSOR precondition. Existing implementation of CG algorithm used CUDA and OpenCL, GPU language for high performance computing [11]. In Mukunoki's research [22], CG method was implemented on CPU and GPU and conducted numerical experiments on both different platforms then compared to existed work based on the ExBLAS library. Additionally, conjugate gradient was used in finite-element method (FEM) which exhibits parallelism on a GPU [23].

Likewise, Sparse Matrix-Vector Multiplication (SpMV) has been studied in order to accelerate performance computation using GPU-based algorithm. Different formats of sparse matrices was used to gain the advantage of storage and time consumption [12-14].

2. OpenGL Compute Shader

OpenGL 4.3, released in 2012, contains compute shaders that are used exclusively for computing and rendering purpose. The compute shader is a shader stage that operate differently from other shader stage [15]. Furthermore, SSBO (Shader Storage Buffer Object), which is a buffer object that has been added instead of UBO (Uniform Buffer Object). SSBO acts as global memory in GPU which has potential to read and write data to memory with wide range arrays [16].

Executing compute shaders require a call to glDispatchCompute(), the CPU's OpenGL API function, and also defines a three-dimensional block of workgroup that determines where work group are in execution space. The computing space has a global workgroup (all work items within all workgroups) and local workgroup. Each local work group is then divided again into a number of invocations or work item. Each work item in a local work group also known as compute shader or kernel which is programmable and c like syntax. Figure 1 demonstrates the scheme of workload dimension.

3. Matrix format

The coefficient matrix which has m rows and n columns is a matrix that contains the coefficient of the variable of sets in linear equation. There are plenty of representation methods to store the information of the coefficient matrix, each with different storage format, attribute of computation, storage reduction and method of fetching matrix entries. There may a situation when matrix contains most zero values, such matrix is known as sparse matrices. In terms of storage, sparse matrix contains only non-zero values since zero values are meaningless to a system of linear equation.

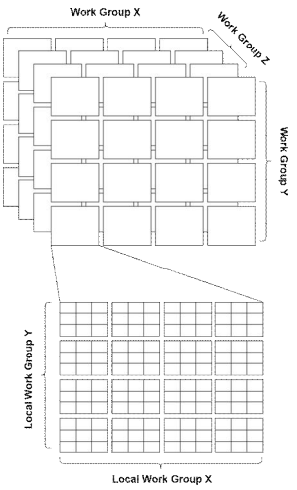

Fig. 1. Scheme of compute workload

3.1 Dense Format

11	0	0	0	0
21	0	23	0	25
0	32	33	34	0
0	42	0	0	45
0	0	0	54	0

Fig. 2. Example Matrix M (5x5)

Suppose we have matrix M with five rows and five columns as shown in Figure 2, dense format contains each element value existed in matrix. This storage format is well-suited when the number of nonzero values is greater than zero value. In general, dense matrices can be represented using a simple two dimensions array. Due to the key aspect of vector architecture with SIMD (Single Instruction Multiple Data), a vector that contains all elements locating in matrix is required. The vector fetching data from left to right and top to bottom of the matrix. Figure 3 shows a dense format of matrix M.

11	0	0	0	0	21	...	45	0	0	0	54	0

Fig. 3. Example of Dense Matrix format for matrix M

3.2 ELLPACK Format

Since matrix M has only 10 nonzero values in 25 elements of the matrix, dense matrix format is

wasting plenty of element to contain zero values. The ELLPACK format uses two Row-by-K to store the matrix M which K is the maximum number of non-zeroes value per row in matrix [17]. ELLPACK requires vector A to store nonzero values and vector Col to store the column indices of nonzero value. Figure 4 shows the example of ELLPACK format of matrix M.

Vector A

11	0	0
21	23	25
32	33	34
42	45	0
54	0	0

Vector Col

0	-1	-1
0	2	5
1	2	3
1	4	-1
3	-1	-1

Fig. 4. ELLPACK matrix format as matrix for matrix M

In this paper, since vector architecture is used, the above matrix format is configured as a transposed vector and the sparse matrix is stored in the column major order.

3.3 Coordinate Format

The Coordinate format is also known as COO which store only nonzero value in the matrix. COO format is the most precise and simple format in term of memory allocation and data accessing that lead this format become properly used for the unstructured pattern of nonzero values locating in the matrix. It stores nonzero elements as an array of triplet [18], which each triplet contains row, column and value of nonzero elements respectively. For these reasons, COO format requires 3-by-nonzero element for storage. Figure 5 shows the example of sparse matrix storage with COO format of matrix M.

Row

0	1	1	1	2	2	2	3	3	4

Col

0	0	2	4	1	2	3	1	4	3

Value

11	21	23	25	32	33	34	42	45	54

Fig. 5. Example of COO Matrix format for matrix M

3.4 Compresses Sparse Row Format

Compresses Sparse Row format (CSR) is an almost similar COO format [19]. To store M matrix with 5x5 dimension, it requires three separate array, which first array (A) is used to store nonzero values, second array (J) is used for store the column indices of nonzero elements, and last array (I) is used for store number of each nonzeros value existed in each row of the matrix M plus number of previous nonzeros elements. In contrast, array I always starts with 0 and ends with the number nonzero, which makes it has a size equal to the row number plus one. Figure 6 illustrates the method of CSR format.

A

11	21	23	25	32	33	34	42	45	54

J

0	0	2	4	1	2	3	1	4	3

I

0	1	4	7	9	10

Fig. 6. Example of CSR Matrix format

III. The Implementation

Conjugate Gradient method [20] is an iterative method for solving equation $Ax = B$ with given matrix A, vector B and unknown vector x. The conjugate gradient method is given by following equation.

$$r_o = p_o = B - Ax_o \quad (1)$$

$$\alpha_i = \frac{r_i^T r_i}{(p_i^T A p_i)} \quad (2)$$

$$x_{i+1} = x_i + \alpha_i p_i \quad (3)$$

$$r_{i+1} = r_i - \alpha_i A p_i \quad (4)$$

$$\beta_i = \frac{r_{i+1}^T r_{i+1}}{(r_i^T r_i)} \quad (5)$$

$$p_{i+1} = r_{i+1} + \beta_i p_i \quad (6)$$

Equation (1) is initial step of the algorithm. Commonly the input vector x can be defined as zero vector for initial solution in the system. So, equation (1) can be written as :

$$r_o = p_o = B \qquad (1.1)$$

From Equation (2) to (6) are routine to define solution with a finite number of iteration that yielding a new approximation solution. The conjugate gradient also converges in n iteration.

In order to implement conjugate gradient using CPU and GPU there were similarities. However, GPU implementation required reading data process from file and data transfer from CPU to SSBO that we use as a buffer for GPU computing as presented in Figure 7. The iterative method of the conjugate gradient method consists of one matrix-vector multiplication, three vector dot products, and three vector operations.

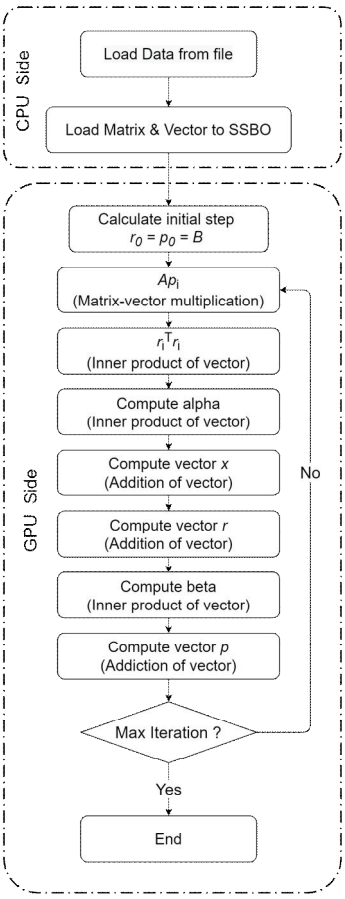

Fig. 7. Conjugate Gradient GPU implementation

Sparse matrix-vector multiplication (SpMV) operation is a crucial operation and requires a suitable matrix format for accelerating. For each SpMV compute shader, workload dimension is different in term of SSBO access by each invocation.

For dense format SpMV algorithm is presented in Table 1, we require two-dimension of work item or invocation which size of X refers to the number row in matrix and Y dimension refer to the number of columns in matrix. In order to access each matrix element by index of invocation properly, the proposed algorithm uses *gl_GlobalInvocationID* which is exact value of *WorkGroupID* * *WorkGroupSize* + *LocalInvocationID*. Since dense matrices are mapping to buffer and accessing arrays like, index of buffer computation is following by equation in Figure 8.

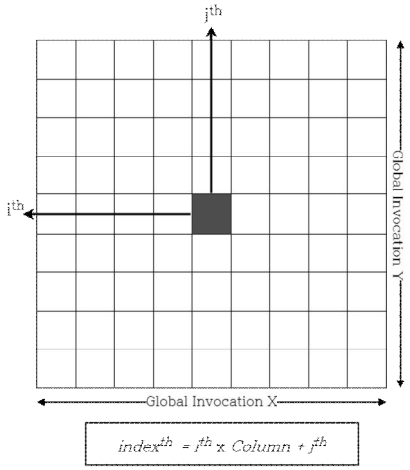

Fig. 8. Assessing matrix by 2D workgroup and index

Once a product of each element of dense matrix with vector P is completed, new data should be written concurrently to result buffer. There could had problem on writing data process which called races in global data. To solve that, *atomicAdd* is used for writing new data to SSBO simultaneously.

Table 1. Dense format SpMV Compute Shader

Algorithm: SpMV on Dense format
Input: SSBO A, P, Row, Column Output: SSBO Result 1: i → gl_GlobalInvocationID.x 2: j → gl_GlobalInvocationID.y 3: if i < Row and j < Column then 4: index → i * Column + j 5: result → A[index] * P[j] 6: atomicAdd(Result[i], result) 7: end if

Table 2 demonstrated the algorithm of SpMV on ELL format. The algorithm requires one-dimension of invocation, which global size equal to length of Value A array and ACol (Column Vector) array. Since compute shader is used, we may potentially send data through uniform variable (Row & densest) to compute shader. This algorithm runs as one thread per matrix row and coalesced memory access which made memory access to array A and ACol turn into a single transaction.

Table 2. ELL format SpMV Compute Shader

```
Algorithm: SpMV on ELL format
Input: SSBO A, ACol, P, Row, densest
Output: SSBO Result
 1: i → gl_GlobalInvocationID.x
 2: if i < Row then
 3:   sum → 0
 4:   for col → 0 to densest do
 5:     idx → col*Row + i
 6:     j → ACol[idx]
 7:     if j ≠ -1 then
 8:       sum → sum + A[idx] * P[j]
 9:     end if
10:  end for
11: Result[i] → sum
12: end if
```

Table 3. COO format SpMV Compute Shader

```
Algorithm: SpMV on COO format
Input: SSBO A, Rows, Cols, P
Output: SSBO Result
 1: i → gl_GlobalInvocationID.x
 2: if i < nnz then
 3:   value = A[i]
 4:   r → Rows[i]
 5:   c → Cols[i]
 6:   value → value * P[c]
 7:   atomicAdd(Result[r],value)
 8 end if
```

SpMV on COO format as shown in Table 3 process each nonzero element which made global invocation index used for access each nonzero value to product with vector and explicitly row and column index of matrix stored in SSBO is used as well. After completing sparse matrix product with vector, result is written to Result SSBO by *atomicAdd* operation.

Table 4 illustrates the CSR SpMV compute shader which, almost identical to COO format excepts compressed row array (IA array) and uses one thread per row of matrix for parallel computing which commonly called scalar kernel.

Table 4. CSR format SpMV Compute Shader

```
Algorithm: SpMV on CSR format
Input: SSBO A, IA, JA, P, Row
Output: SSBO Result
 1: i → gl_GlobalInvocationID.x
 2: if i < Row then
 3:   value → 0
 3:   rowStart → IA[i]
 4:   rowEnd → IA[i+1]
 5:   for j = rowStart to rowEnd do
 6:     value → value + A[j] * P[JA[j]]
 7:   end for
 8:   Result[i] → value
 9: end if
```

IV. Experiment Results

In our experiment, we conducted a measurement execution time for conjugate gradient method at one iteration by using a naive approach (CPU-based) running on single core compared to our approach which uses parallelized GPU-based linear solving implemented by OpenGL compute shader. We also provide the different result between each sparse matrix format in Table 5 and Table 6. Therefore, C++ Library is used for retrieving execution time for CPU implementation. Likewise, OpenGL is used for obtaining duration of GPU algorithm as well. The proposed algorithm is implemented with Visual Studio 2013 and run on Window platform with i7-00 CPU, 16GB RAM and use NVDIA GeForce GTX 1070 8GB VRAM.

Sparse matrices dataset from [21] is used for the proposed method as well. Each right hand side vector element are made by sum of each row of matrices. Table 5 shows the matrix information which are used to experimental test. All matrices using for experimental test are general problem of

computational fluid dynamics problem except lhr07. lhr07 is problem of chemical process simulation and kineticBatchReactor_2 is problem of optimal control problem. In addition, all matrices are real and structured. However, only ex15 is symmetric and positive definite and kineticBatchReactor_2 is symmetric but indefinite.

Table 5. Sparse Matrices Data

Name	Size	Nonzero	rate
cavity21	4,562x4,562	131,735	0.63%
lhr07	7,337x7,337	154,660	0.28%
goodwin	7,320x7,320	324,772	0.60%
ex15	6,867x6,867	98,671	0.20%
shyy41	4,720x4,720	20,042	0.08%
kineticBatchReactor_2	4,361x4,361	44,840	0.23%
tols4000	4,000x4,000	8,784	0.05%
rdb5000	5,000x5,000	29,600	0.11%

In Table 8, the comparison of performance results have been shown that GPU-based parallel linear solving system with dense format is average 233 times faster than CPU-based linear solving system, but it wastes significant amount of memory compares with other formats. ELL format wastes less memory for allocation and average 174 times faster than CPU-based linear solving system, but it is not suitable for unstructured pattern of nonzero values. While COO format is the most uses in general pattern and takes less time for computation in both CPU-based and GPU-based linear solving system with average speed up ratio 20 times. CSR format also has a good performance in both implementation with the result that the GPU-based linear solving system is average 24 times faster than CPU-based linear solving system.

Furthermore, the comparison of average computing time for each format of CPU-based and GPU-based linear solving system have shown that CSR format improves the performance of conjugate gradients algorithm. Beside that, ELL also offers an acceptable performance but this format can not be used as general representation for sparse matrix.

Table 6. CPU Implemented CG Algorithm perform for one iteration and measure in millisecond

Matrices	Dense	ELL	COO	CSR
cavity21	1,623	45	30	20
lhr07	4,137	76	35	25
goodwin	4,241	133	76	49
ex15	3,750	19	22	15
shyy41	1,751	4	3	2
kineticBatchReactor_2	1,518	961	10	7
tols4000	1,318	60	1	1
rdb5000	1,986	4	7	4
Avg. compt. Time	2540.5	162.75	23	15.37

Table 7. GPU Implemented CG Algorithm perform for one iteration and measure in millisecond

Matrices	Dense	ELL	COO	CSR
cavity21	6.7	0.66	0.88	0.66
lhr07	21.59	0.67	1.1	0.7
goodwin	20.36	0.73	1.51	0.81
ex15	18.23	0.21	0.94	0.31
shyy41	6.85	0.77	0.79	0.85
kineticBatchReactor_2	6.39	1.16	0.76	0.6
tols4000	5	0.6	0.56	0.44
rdb5000	7.65	0.77	0.83	0.78
Avg. compt. Time	11.59	0.69	0.92	0.64

Table 8. Speed up ratio (CPU/GPU)

Matrices	Dense	ELL	COO	CSR
cavity21	242.23	68.18	34.09	30.3
lhr07	191.61	113.43	31.81	35.71
goodwin	208.3	182.19	50.11	60.49
ex15	205.7	90.47	23.4	48.38
shyy41	255.62	5.19	3.79	2.35
kineticBatchReactor_2	237.55	828.44	13.15	11.66
tols4000	263.6	100	1.78	2.27
rdb5000	259.6	5.19	8.43	5.12
Avg	233.02	174.13	20.82	24.53

V. Conclusion

The proposed GPU-parallel algorithm for sparse linear solving by conjugate gradient method using OpenGL compute shader for any system matrices are symmetric, positive and definite. Different storage format of sparse matrices are used in order to achieve excellent performance and saving memory as much as possible. The proposed

method also have composed GPU library which supports with OpenGL compute shader by utilize data using atomic operation on SSBO.

In conclusion, GPU-based parallel linear solving algorithm for computing sparse linear systems through conjugate gradients using an iterative method, which applies CSR format as matrix storage offers the best performance with average computing time 15.37 ms in CPU-based parallel linear solving algorithm and 0.64 ms in GPU-based parallel linear solving algorithm for all cases. ELL format also achieve good performance in GPU-based with average executing time 0.69 ms and COO format is 0.92 ms. The result proves that CSR format is a suitable format in general case of sparse matrix in term of storage and parallelization which lead the CG solver achieve great performance.

In future work, our approach can use for solving dynamic simulation problems such as node-node constraints in order to maintain distance of each constraint in deformable object by using GPU-based CG solver.

ACKNOWLEDGEMENT

This work was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF-2019R1F1A1062752) funded by the Ministry of Education, and was supported by the Soonchunhyang University Research Fund.

REFERENCE

[1] S. Abbasbandy, A. Jafarian, and R. Ezzati, “Conjugate gradient method for fuzzy symmetric positive definite system of linear equations,” Applied Mathematics and Computation, Vol. 171, No. 2, pp. 1184-1191, 2005. DOI: https://doi.org/10.1016/j.amc.2005.01.110

[2] M. Kryshchuk, and J. Lavendels, “Iterative Method for Solving a System of Linear Equations,” Procedia Computer Science, Vol. 104, pp. 133-137, 2017. DOI: https://doi.org/10.1016/j.procs.2017.01.085

[3] A. Bunse-Gerstner, and R. Stöver, “On a conjugate gradient-type method for solving complex symmetric linear systems,” Linear Algebra and its Applications, Vol. 287, No. 1−3, pp. 105-123, 1999. DOI: https://doi.org/10.1016/S0024-3795(98)10091-5

[4] A. Cano, “A survey on graphic processing unit computing for large-scale data mining,” WIREs Data Mining and Knowledge Discovery, Vol, 8, No. 1, 2018. DOI: https://doi.org/10.1002/widm.1232

[5] J. Sim, et al, “A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications,” Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Vol. 47, No. 8, pp. 11-22, 2012. DOI: https://doi.org/10.1145/2145816.2145819

[6] D. Gerzhoy, X. Sun, M. Zuzak, and D. Yeung, “Nested MIMD-SIMD Parallelization for Heterogeneous Microprocessors,” ACM Trans. Archit, Vol. 16, No. 4, 2019. DOI: https://doi.org/10.1145/3368304

[7] W. Shin, K. H. Yoo and N. Baek, “Large-Scale Data Computing Performance Comparisons on SYCL Heterogeneous Parallel Processing Layer Implementations,” In Appl. Sci., Vol. 10, No. 5. pp. 1656, 2020. DOI: https://doi.org/10.3390/app10051656

[8] J. S. Kirtzic, “A parallel algorithm design model for the gpu architecture,” Ph.D. Dissertation. University of Texas at Dallas, USA. Advisor(s) Ovidiu Daescu., 2012.

[9] M. Ament et al, "A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform," 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, Pisa, pp. 583-592, 2010. DOI: https://doi.org/10.1109/PDP.2010.51

[10] R. Helfenstein, and J. Koko, “Parallel preconditioned conjugate gradient algorithm on GPU,” Journal of Computational and Applied Mathematics, Vol. 236, No. 15. pp. 3584-2590, 2012. DOI: https://doi.org/10.1016/j.cam.2011.04.025

[11] A. C. Ahamed, and F. Magoulès, “Conjugate gradient method with graphics processing unit acceleration: CUDA vs OpenCL,” Advances in Engineering Software, Vol. 111, pp. 32-42, 2017. DOI: https://doi.org/10.1016/j.advengsoft.2016.10.002

[12] M. M. Baskaran, and R. Bordawekar, “Optimizing Sparse Matrix-Vector Multiplications on GPUs,” Computer Science. Vol. 8, pp. 812-47, 2009.

[13] A. Benatia, W. Ji, Y. Wang and F. Shi, “Sparse Matrix Format Selection with Multiclass SVM for SpMV on GPU,” 2016 45th International Conference on Parallel Processing (ICPP), Philadelphia, pp. 496-505, 2016. DOI: https://doi.org/10.1109/ICPP.2016.64

[14] N. Bell and M. Garland, “Implementing sparse matrix-vector

multiplication on throughput-oriented processors," Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, Portland, pp. 1-11, 2009. DOI: https://doi.org/10.1145/1654059.1654078

[15] D. Shreiner, G. Sellers, J. Kessenich and B. Licea-Kane , "OpenGL programming guide: The Official guide to learning OpenGL," version 4.3. Addison-Wesley, 2013.

[16] N. J. Sung, Y. J. Choi and M. Hong, "Parallel Structure Design Method for Mass Spring Simulation," Journal of the Korea Computer Graphics Society. Vol. 25, pp. 55-63, 2019. DOI: https://doi.org/10.15701/kcgs.2019.25.3.55

[17] N. Bell and M. Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA," NVIDIA Technical Report NVR-2008-004, December 2008.

[18] W. Cao, L. Yao, Zo. Li, Y. Wang and Z. Wang, "Implementing Sparse Matrix-Vector multiplication using CUDA based on a hybrid sparse matrix format," 2010 International Conference on Computer Application and System Modeling, pp. V11-161-V11 -165, 2010. DOI: https://doi.org/10.1109/ICCASM.2010.5623237

[19] G. E. Blelloch, et al, "Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors," Technical Report. Carnegie Mellon University, USA. 1993.

[20] M. R. Hestenes, and E. Stiefel, "Methods of conjugate gradients for solving linear systems," Journal of research of the National Bureau of Standards, Vol. 49, pp. 409-436, 1952. DOI: https://doi.org/10.6028/jres.049.044

[21] SuiteSparse Matrix Collection, https://sparse.tamu.edu/

[22] D. Mukunoki, K. Ozaki, T. Ogita, and R. Iakymchuk " Conjugate Gradient Solvers with High Accuracy and Bit-wise Reproducibility between CPU and GPU using Ozaki scheme, " In The International Conference on High Performance Computing in Asia-Pacific Region, pp. 100- 109, 2021. DOI: https://doi.org/10.1145/3432261.3432270

[23] Pikle, N.K., Sathe, S.R. and Vyavhare, A.Y. "GPGPU-based parallel computing applied in the FEM using the conjugate gradient algorithm: a review," Sādhanā, Vol.43, No. 111 2018. DOI: https://doi.org/10.1007/s12046-018-0892-0

Authors

Hongly Va received the B.S degree in Information Technology Engineering from Royal University of Phnom Penh, in 2019. He is currently an assistant professor at the Dep. of Software Convergence at Soonchunhyang University, Korea. His research interests are in Computer graphics, Virtual Reality, parallel computing, Physically-based Modeling and Simulation.

Do-keyong Lee received the B.S degree in Law from Baekseok University in 2018. She received the M.S degree in ICT Convergence Rehabilitation Engineering from Soonchunhyang University, Korea in 2020. Now she is undertaking a Ph.D of Software Convergence Engineering courses as a member of the computer graphics lab at Soonchunhyang University. Her research interests are Computer Graphics, Physically based Modeling and Simulation

Min Hong received B.S. degree in Computer Science from Soonchunhyang University in 1995. He received his M.S. degree in Computer Scienceand Ph.D. degree in Bioinformatics from the University of Colorado in 2001 and 2005, respectively. Dr. Hong is a professor at the Dept. of Computer Software Engineering, Soonchunhyang University, Korea. His research interests are in Computer Graphics, Mobile Computing, Physically based Modeling and Simulation.