Performance Study of LU Decomposition on the Programmable GPU[⋆]

Fumihiko Ino, Manabu Matsui, Keigo Goda, and Kenichi Hagihara

Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
ino@ist.osaka-u.ac.jp

Abstract. With the increasing programmability of GPUs (graphics processing units), these units are emerging as an attractive computing platform not only for traditional graphics computation but also for general-purpose computation. In this paper, to study the performance of programmable GPUs, we describe the design and implementation of LU decomposition as an example of numerical computation. To achieve this, we have developed and evaluated some methods with different implementation approaches in terms of (a) loop processing, (b) branch processing, and (c) vector processing. The experimental results give four important points: (1) dependent loops must be implemented through the use of a render texture in order to avoid copies in the video random access memory (VRAM); (2) in most cases, branch processing can be efficiently handled by the CPU rather than the GPU; (3) as Fatahalian et al. state for matrix multiplication, we find that GPUs require higher VRAM cache bandwidth in order to provide full performance for LU decomposition; and (4) decomposition results obtained by GPUs usually differ from those by CPUs, mainly due to the floating-point division error that increases the numerical error with the progress of decomposition.

1 Introduction

The GPU [1, 2] is a single-chip processor, which is designed to accelerate rendering tasks for interactive visualization. Recently, GPUs on commodity PC graphics cards are emerging as a novel high performance computing (HPC) platform with providing faster floating-point operations than CPUs [3]. Newly added functionalities such as programmability and branch capability make them an attractive HPC platform not only for visualization purposes but also for general purposes.

Such new functionalities also activate the use of modern GPUs for solving numerical problems. Thompson et al. [4] implement matrix multiplication on a GPU, achieving three times higher performance compared with a simple CPU implementation. Larsen et al. [5] compare their GPU implementation with ATLAS [6], a cache-optimized CPU implementation. They present two requirements for making their GPU implementation competitive against ATLAS: one is a significant increase of VRAM access speed and the other is that of graphics chip core clock. To approach these requirements from

[⋆] This work was partly supported by JSPS Grant-in-Aid for Scientific Research on Priority Areas (16016254).

the software side, Hall et al. [7] propose a VRAM cache and bandwidth aware algorithm with its theoretical evaluation. Their algorithm is evaluated on real graphics cards by Fatahalian et al. [3]. This experimental evaluation shows that higher VRAM cache bandwidth is yet essential for GPUs to outperform ATLAS.

In addition to the problem of matrix multiplication, there are a wide variety of numerical applications running on the GPU: the conjugate gradient method [8–10], the Gauss-Seidel method [8], the projected Jacobi method [10], and the fast Fourier transform [11]. Thus, many researchers try to accelerate numerical computations using the GPU. However, it is still not clear what kinds of design guidelines will yield higher performance on GPUs, mainly due to the rapid advances in GPU architectures. Furthermore, most vendors rarely disclose the details of their GPU architectures.

The goal of our work is to analyze the performance behavior of the GPU, aiming at making clear the design guidelines for GPU accelerated numerical computations. To achieve this, we focus on the problem of LU decomposition, which is used for ranking top 500 supercomputers. We present its design with different implementation approaches in terms of (a) loop processing, (b) branch processing, and (c) vector processing. We also show some performance studies using commodity PC graphics cards.

To the best of our knowledge, the key contributions of the paper are (1) the design guidelines for the above implementation issues (a)–(c) and (2) the first GPU implementation for LU decomposition.

The paper is organized as follows. Section 2 presents a brief overview of the GPU architecture and summarizes prior strategies for solving numerical problems by the GPU. Section 3 describes the implementation approaches that compose our methods, and then Section 4 shows the performance studies obtained on modern graphics cards. Finally, Section 5 concludes the paper.

2 Graphics Processing Unit (GPU)

2.1 Overview of Architecture

The rendering task, which GPUs originally accelerate, is to compute pixels on the 2-D image by projecting polygonal objects (triangles) located in the 3-D space. In order to accelerate this compute-intensive task, modern GPUs [12, 2] employ a pipeline architecture as shown in Fig. 1. Due to limited space, we introduce only the programmable part of this pipeline, namely vertex processors (VPs) and fragment processors (FPs).

VPs and FPs are vector processors with 4-length vector registers. These processors have the following characteristics.

VP: VPs are capable of fast geometric transformation in order to accelerate the projection of vertices of polygons onto the 2-D space. They are based on a MIMD structure [13] that allows applying different operations simultaneously to multiple vertices. Here, the polygonal data must be transferred from the main memory to the VRAM by using graphics APIs such as OpenGL [14] and DirectX [15].

FP: FPs are capable of rapid mapping of textures onto the 2-D image, aiming at producing more realistic images. To perform this, they obtain projected pixels (called

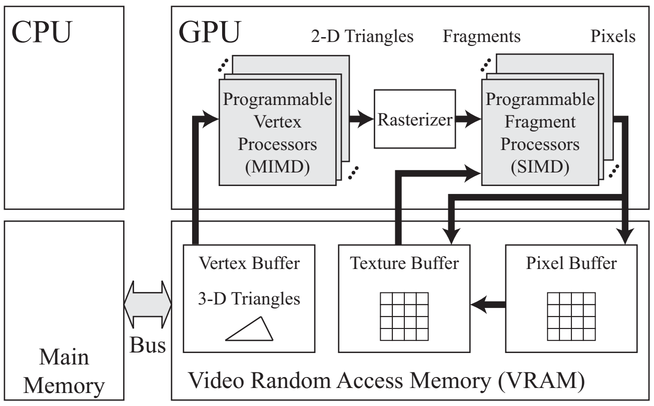


Fig. 1. GPU pipeline architecture.

fragments) from the rasterizer, and then execute some mathematical operations between the pixels and textures. Here, the textures are read from the VRAM, and the mapping results are written to any buffer on the VRAM. FPs are based on a SIMD structure [13] that allows applying the same operation simultaneously to multiple fragments. There are two characteristics that must be mentioned. (C1) FPs support 4-length vector operations, because they deal with 4-channel (RGBA) data representing red, green, blue colors and opacity. (C2) Because textures are generally 2-D images, FPs can be regarded as vector processors that execute independent operations on each element on a matrix.

As we mentioned in Section 1, recent advances have removed many limitations that earlier GPUs have. For example, earlier GPUs do not have any control flow mechanism. Furthermore, only short programs were executable due to the limitation on instruction count. In contrast, modern GPUs allow more instructions with branch capability and some GPUs follow a 32-bit single floating-point number representation based on the IEEE standard [16]. Furthermore, by using the graphics APIs mentioned above, the rendered results can be transferred (readback) from the VRAM to the main memory.

2.2 Prior Strategies for Accelerating Numerical Computations on the GPU

Recent work [3, 7–11] uses only FPs for numerical computation while earlier work [4] uses VPs. This is due to lower performance of current VPs, which are not competitive against CPUs [7]. For example, as we show later, VPs in nVIDIA's GeForce cards [17] provide 338 million vertices/s (namely, vectors/s), whereas FPs provide 3.6 billion texels/s (vectors/s). Due to this performance characteristic, we also use only FPs for LU decomposition.

In order to maximize the efficiency on FPs, prior work focuses on characteristics C1 and C2. For example, in a case of matrix multiplication $\mathbf{XY} = \mathbf{Z}$, each of elements Z_{ij} can be computed independently. Therefore, FPs are allowed to simply render the result matrix $\mathbf{Z}$ into a VRAM buffer by referring two textures each containing matrix data

```
1: Algorithm RightLookingLUDecomposition {
2:   for ($i = 0; i < N; i++$) {
3:     for ($j = i + 1; j < N; j++$) {
4:       $A_{ji} = A_{ji}/A_{ii}$; /* update L */
5:       for ($k = i + 1; k < N; k++$)
6:         $A_{jk} -= A_{ik} * A_{ji}$; /* update U */
7:     }
8:   }
9: }
```

Fig. 2. Right-looking LU decomposition algorithm.

$\mathbf{X}$ and $\mathbf{Y}$, respectively. Thus, a doubly-nested loop without any dependencies between loop iterations can be efficiently processed by a single pass of the data through the pipeline. Furthermore, to enable vectorization, some researchers pack the matrix data into the 4-channel texture format. They store an $N \times N$ matrix in an $N/4 \times N$ texture, multiplying the elements on four rows and one column at once.

Although the single-pass rendering approach mentioned above is effective for independent nested loops, it must not be applied to a dependent nested loop, because such a loop cannot be processed simultaneously due to dependencies between loop iterations. This is one of the problems addressed in the paper.

The single-pass rendering approach is also inapplicable to programs whose size exceeds the limitation on the instruction count. This limitation can be resolved by multi-pass rendering, which aims at emulating the entire execution by dividing the program into small parts. In this method, data is repeatedly passed through the pipeline with varying the program parts at each pass.

In summary, prior work presents the following four guidelines for yielding higher performance on GPUs:

- Apply single-pass rendering to independent doubly-nested loops that contain a large amount of computation;
- Pack matrix data into the 4-channel texture format to enable vectorization and to reduce the usage of VRAM;
- Reduce the amount and number of data transfer between the GPU and the CPU;
- Reduce the number of VRAM accesses to save the VRAM bandwidth.

3 LU Decomposition on the GPU

This section presents the design of LU decomposition on the GPU. Table 1 summarizes our methods with their theoretical performance.

3.1 LU Decomposition

LU decomposition is a method for solving a linear system $\mathbf{Ax} = \mathbf{b}$. It factorizes a matrix $\mathbf{A}$ into two triangular matrices: a lower matrix $\mathbf{L}$ and an upper matrix $\mathbf{U}$. Then,

Table 1. Theoretical performance of proposed methods M1, M2, M3, and M4.

	Vectorization	Branch	Loop	Rendering pass		VRAM copy	
				Number	Weight	Number	Amount (B)
M1	No	CPU	Copying Switching	$2N$	1	2N 0	$32N(N^2-1)/3$ 0
M2	No	GPU	Copying Switching	N	1	N 0	$16N(N^2-1)/3$ 0
M3	Yes	CPU	Copying Switching	$8N$	$1/4$	$2N$ 0	$2N(4N^2+3N-4)/3$ 0
M4	Yes	GPU	Copying Switching	$4N$	$1/4$	$2N$ 0	$2N(4N^2+3N-4)/3$ 0

the solution $\mathbf{x}$ is computed by forward and backward substitution. There are three algorithms for this method: right-looking (see Fig. 2), left-looking, and Crout algorithms [18]. Given an $N \times N$ matrix, these algorithms require the same $O(N^3)$ time but differ in parallelism and locality of data access.

Among these algorithms, we select the right-looking version for GPUs, which have much smaller caches than CPUs [2]. The reason why we select this version is that its reference area at each decomposition step is smaller than the others. Thus, we think that current GPUs are not suited to algorithms that require larger caches. Note here that we currently do not consider pivoting because our main focus is the performance study of GPUs.

3.2 Design Policy

To realize LU decomposition on the GPU, the following three issues must be resolved.

(a) Loop processing: There are dependencies between the outer i loop iterations in Fig. 2. Due to these dependencies, we cannot simply apply single-pass rendering to LU decomposition. A naive solution is to use a multi-pass rendering approach.
(b) Branch processing: While matrix multiplication applies the same operation to all matrix elements, LU decomposition uses two different operations, each for computing matrices $\mathbf{L}$ and $\mathbf{U}$ (lines 4 and 6 in Fig. 2). Therefore, we have to select the correct operation depending on the location of matrix elements. Thus, branch processing is required for this selection because FPs are SIMD processors.
(c) Vector processing: As same as for matrix multiplication, we should pack matrix data into the 4-channel format to enable vectorization.

In the following we describe our implementation approaches that address the issues mentioned above. We present two approaches for each issue. As shown in Table 1, our proposed methods are combinations of these approaches. The design policies of these methods are as follows:

– Method M1 eliminates branch operations from the GPU program, though it requires more passes;

```
1: Algorithm TwoPassLUDecomposition {
2:   for ($i = 0; i < N; i++$) {
3:     for ($j = i + 1; j < N; j++$) /* rendering */
4:       $A_{ji} = A_{ji}/A_{ii}$; /* update L */
5:     for ($j = i + 1; j < N; j++$) /* rendering */
6:       for ($k = i + 1; k < N; k++$)
7:         $A_{jk} -= A_{ik} * A_{ji}$; /* update U */
8:   }
9: }
```

(a)

```
1: Algorithm OnePassLUDecomposition {
2:   for ($i = 0; i < N; i++$) {
3:     for ($j = i + 1; j < N; j++$) { /* rendering */
4:       for ($k = i; k < N; k++$) {
5:         if ($i == k$) $A_{ji} = A_{ji}/A_{ii}$; /* update L */
6:         else $A_{jk} -= A_{ik} * A_{ji}/A_{ii}$; /* update U */
7:       }
8:     }
9:   }
10: }
```

(b)

Fig. 3. Proposed methods. (a) Two-pass method M1 and (b) single-pass method M2.

- In contrast, method M2 achieves less passes, though it includes branch operations in the GPU program;
- The remaining methods M3 and M4 exploit vectorization on the basis of methods M1 and M2, respectively.

3.3 Loop Processing

To consider the issue of loop processing, we first characterize the dependencies between loop iterations. The dependencies in the right-looking algorithm are as follows:

1. The outer i loop iterations cannot independently be processed;
2. The inner k loop iterations can independently be processed;
3. The inner k loop iterations must be processed after completing the assignment (line 4) in the middle j loop.

Due to the first dependency mentioned above, multi-pass rendering is necessary for LU decomposition. The key idea here is that, according to characteristic C2, single-pass rendering can be applied to the inner jk loops if these loops are restructured into an independent loop. The following reconstruction methods can be considered.

- Loop decomposition (method M1, Fig. 3(a)): In this method, a nested loop for updating $\mathbf{L}$ and $\mathbf{U}$ (lines 3–7 in Fig. 2) is decomposed into two loops (lines 3–4 and lines 5–7 in Fig. 3). These decomposed loops cannot be processed at once. However, each loop can be processed in parallel. Therefore, this method requires two passes for rendering $\mathbf{L}$ and $\mathbf{U}$, as illustrated in Fig. 4(a) and (b).
- Loop fusion (method M2, Fig. 3(b)): In this method, the assignment for updating $\mathbf{L}$ (line 4 in Fig. 2) is moved into the inner k loop (line 5–6 in Fig. 3(b)). This eliminates the dependencies so that enables parallel processing of the inner jk loops. Thus, this method requires only a single pass for updating $\mathbf{L}$ and $\mathbf{U}$, as illustrated in Fig. 4(c). However, it increases the time complexity because the assignment is moved into the inner loop (line 6 in Fig. 3(b)).

The next issue to be addressed is how data can be iteratively passed through the pipeline. There are two strategies for this issue.

- Copying strategy using a pixel buffer: FPs write their computation results into a pixel buffer. After this, the buffer context is copied to a texture for the next succeeding pass. This strategy requires the copy overhead.

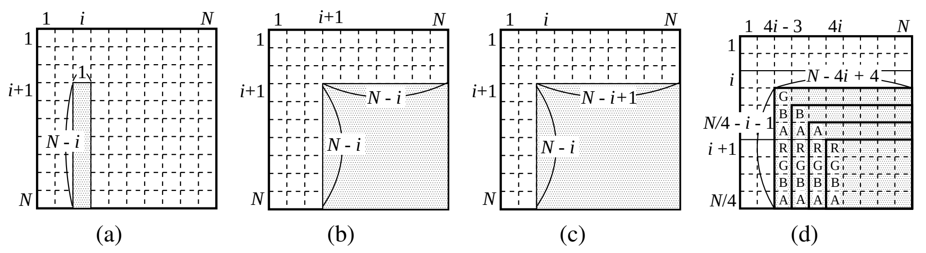


Fig. 4. Matrix data rendered at the i-th pass, where $1 \leq i \leq N$. (a,b) Method M1 renders matrices **L** and **U** in two passes. (c) Method M2 renders them at once. (d) Method M4 integrates vectorization with M2 by packing matrix data into the 4-channel (RGBA) format.

– Switching strategy using a render texture: This strategy uses two textures, each for input and output of FPs. At every pass, FPs switch these textures to prevent VRAM copies. This strategy requires the switch overhead instead of the copy overhead.

3.4 Branch Processing

We now describe how methods M1 and M2 resolve the issue of branch processing. Branches in method M1 are handled by the CPU. In this method, the entire loop is mapped to two rendering passes, as we mentioned in Section 3.3. Therefore, the CPU takes the responsibility for loading the appropriate GPU program with its rendering area. Thus, branches are naturally handled by the CPU. As a result, the GPU is allowed to concentrate on executing the given program without any control flow. On the other hand, method M2 requires the GPU to process branches. This can be easily implemented by comparing i and k, the location of matrix elements as shown in Fig. 3(b).

In summary, although CPU implementations do not include branch operations, their GPU versions may include them due to the SIMD architecture. If branch conditions are expressed by the location in matrix data, such branches can be eliminated from the GPU program but with more rendering passes.

3.5 Vector Processing

As same as for matrix multiplication [3, 7], we also apply vectorization to our methods M1 and M2 in order to reduce the execution time. Fig. 4(d) illustrates how the matrix data are mapped to the texture format. In this method, an $N/4 \times N$ texture represents an $N \times N$ matrix, enabling applying vector operations to the data on four rows and one column. Note here that we cannot apply them in other directions, for example, to the data on one row and four columns, because there are dependencies between different columns (the outer i loop iterations).

Applying vectorization then raises another issue to be addressed. The issue is that, as shown in Fig. 4(d), the appropriate channel must be selected in order to perform correct

Table 2. Specification of experimental environments.

GPU	nVIDIA GeForceFX 5900Ultra	nVIDIA QuadroFX 3400
Core clock	450MHz	350MHz
Texture fill-rate	3.6Gpixels/s	5.6Gpixels/s
VRAM capacity	128MB	256MB
VRAM bandwidth	27.2GB/s	28.8GB/s
Texture cache capacity	Undisclosed	Undisclosed
Texture cache bandwidth	11.4GB/s	15.6GB/s
Graphics bus	AGP8X	PCI Express
CPU	Pentium 4 2.6GHz	Pentium 4 2.8GHz
OS	Red Hat Linux 9	Windows XP

rendering for each column. For example, at the top-left corner of rendering area, we can see that all of the RGBA channels must be rendered for $i = 4, 8, \ldots$, whereas only the GBA channels must be rendered for $i = 1, 5, \ldots$.

This issue is the same branch issue addressed in Section 3.4, because its branch condition can be expressed by the location in matrix data. Therefore, we solve it in the same manner. That is, we implement four GPU programs, each renders the GBA, BA, A, and RGBA channels, respectively, and then switch them in a cyclic manner.

4 Performance Study

We now show performance studies in order to analyze the performance behavior of the GPU. We study its behavior from the following three viewpoints: design guidelines for implementation issues (a)–(c); efficiency in terms of cache bandwidth; and numerical error.

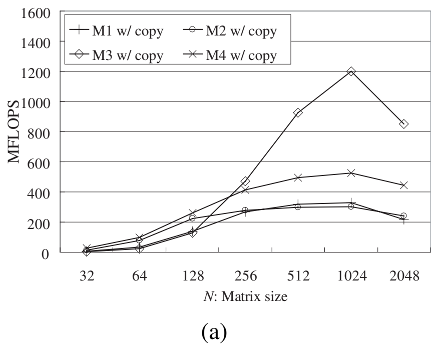
Table 2 shows the specification of our two machines used for the study. We have implemented the four methods using the C++ language, the OpenGL library, and the Cg language [19]. Note here that render-to-texture functionality is not yet available on Linux systems. Therefore, we used only the copying strategy for the Linux system.

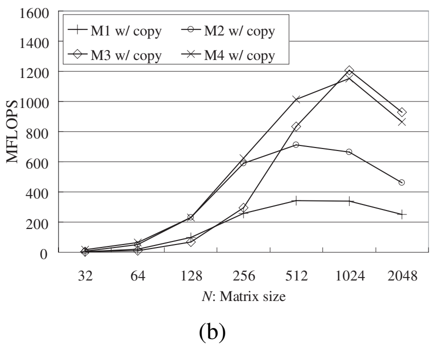
Fig. 5 shows the measured performance for different matrix sizes N. We can see that the Quadro card yields the best performance of 1.6 GFLOPS for $N = 1024$ by using method M3 with the switching strategy. On the other hand, the GeForce card reaches 1.2 GFLOPS for $N = 1024$ by using method M3 with the copying strategy.

In this figure, we can also see that methods M2 and M4 on Quadro provide relatively higher performance than those on GeForce. This indicates that the newer generation of Quadro reduces the branch overhead on FPs, as compared to GeForce, because these methods differ from methods M1 and M3 in the use of branch operations.

We next investigate the breakdown of execution time to present design guidelines for implementation issues (a)–(c). Table 3 shows the breakdown measured on Quadro.

– (a) On loop processing. The switching strategy prevents copies in the VRAM, so that spends no time T for the VRAM copy, as shown in Table 3. However, instead of this overhead, the switch overhead is observed in the CPU time C. For example, method M2 increases time C from 95 ms to 128 ms. However, this switch overhead





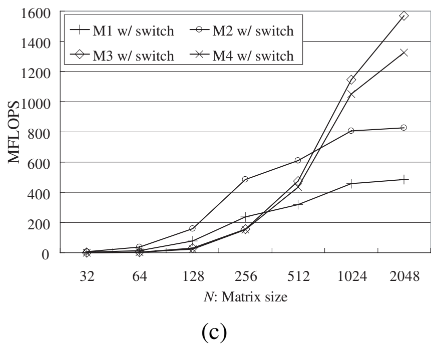


Fig. 5. Measured performance for different matrix sizes. (a) GeForce card with the copying strategy. Quadro card (b) with the copying strategy and (c) with the switching strategy.

is small enough to the entire time A. This is also true for methods M3 and M4, which require more switches due to vectorization. Therefore, in most cases, the switching strategy seems better than the copying strategy. One concern is portability because Linux systems currently support only the copying strategy.

– (b) On branch processing. As compared to method M1, method M2 increases the GPU time G as matrix size N increases. We can see this increase also in its vectorization version M4. This increase is due to the branch overhead occurred on FPs. Thus, for larger matrices, branch operations should be processed by the CPU in order to obtain higher performance. In contrast, for smaller matrices, method M1 provides higher performance than method M2. This opposite result is due to the switch overhead mentioned above. That is, though method M1 eliminates branch operations from the GPU program, it requires an additional overhead for switching GPU programs. This overhead results in longer time C, increasing the entire time A especially for smaller N. Thus, there is a tradeoff relation between the GPU branch and the CPU branch. The tradeoff point is determined by the computational amount

Table 3. Breakdown of measured time on Quadro. A, G, C, and T represent the entire time, the GPU calculation time, the CPU calculation time, and the VRAM copy time, respectively.

N	M1 w/ copying (ms)				M2 w/ copying (ms)				M3 w/ copying (ms)				M4 w/ copying (ms)			
	A	G	C	T	A	G	C	T	A	G	C	T	A	G	C	T
32	9	6	2	1	7	6	1	1	39	12	26	1	51	1	48	1
64	13	8	3	2	10	7	2	1	55	15	38	3	64	3	56	5
128	26	14	6	6	19	12	4	3	86	20	60	6	94	6	82	6
256	79	41	11	26	55	36	6	13	160	36	108	15	160	17	126	16
512	438	250	24	164	335	240	13	82	365	102	201	62	360	84	214	63
1024	3291	2022	64	1205	2691	2050	37	604	1306	566	391	350	1334	592	391	351
2048	34942	21629	108	13206	30545	23752	95	6698	10079	5875	781	3422	10489	6307	761	3421

N	M1 w/ switching (ms)				M2 w/ switching (ms)				M3 w/ switching (ms)				M4 w/ switching (ms)			
	A	G	C	T	A	G	C	T	A	G	C	T	A	G	C	T
32	8	5	3	—	6	5	1	—	50	28	21	—	63	43	21	—
64	15	7	8	—	9	6	3	—	69	38	31	—	77	46	32	—
128	26	13	13	—	17	10	7	—	103	42	61	—	114	57	57	—
256	67	38	29	—	49	36	13	—	208	68	140	—	206	69	136	—
512	318	243	75	—	306	264	42	—	418	149	269	—	409	164	245	—
1024	1603	1470	133	—	1756	1690	66	—	1096	596	500	—	1135	650	485	—
2048	11564	11249	315	—	13309	13181	128	—	4477	3483	994	—	5048	4124	924	—

associated with a single pass of rendering. In this experiment, the tradeoff point is $N = 512$.

– (c) On vector processing. The timing benefits of vectorization can be observed in time G. For example, applying vectorization to M1 reduces time G from 11249 ms to 3483 ms. Thus, the vectorization effect is almost the same value as the vector length. In addition to this obvious result, vectorization also contributes to the reduction of the VRAM copy time T when using the copying strategy. This reduction comes from the data packing required for vectorization. Actually, time T in M3 is 3422 ms, which is almost 1/4 of time T in M1. Thus, vectorization is essential to reduce both the GPU time and the VRAM copy time.

We next investigate the efficiency from the viewpoint of cache bandwidth. As Fatahalian et al. [3] did, we also modified GPU programs such that the programs only access the matrix data without performing mathematical operations. Then, by measuring the GPU time, namely the access time, and using the theoretical amount of data accesses in Table 1, we obtain the throughput of our methods. The best throughput on GeForce is 8.6GB/s when using method M3 with the copying strategy for $N = 1024$ and that of Quadro is 11.4GB/s when using M3 with the switching strategy for $N = 2048$. According to these results and theoretical bandwidth in Table 2, the efficiency of cache bandwidth reaches 75% on GeForce and 73% on Quadro. These values are similar to those of matrix multiplication [3]. Thus, we find that GPUs require higher VRAM cache bandwidth in order to provide full performance for LU decomposition.

In terms of FLOPS, the efficiency of our methods is estimated as at most 30%. This efficiency is not competitive against that of CPU implementations. For example, some CPU implementations [20–22] optimize locality to achieve higher cache utilization, so that achieve an efficiency of more than 80%. Therefore, more efficient methods are required to make GPUs a competitive HPC platform,

Table 4. Floating-point errors in unit in last place. Errors on Quadro are measured by Paranoia [23]. In the IEEE standard [16], the result is rounded to the nearest representable number.

Operation	IEEE754	Quadro
Multiplication	$[-0.5, 0.5]$	$[-0.78125, 0.625]$
Division	$[-0.5, 0.5]$	$[-1.19902, 1.37442]$
Subtraction	$[-0.5, 0.5]$	$[-0.75, 0.75]$
Addition	$[-0.5, 0.5]$	$[-1, 0]$

Finally, we investigate computation results in terms of numerical errors. In most cases, there are differences between the CPU and GPU results. These differences are due to the floating-point error, as presented in Table 4. As Hillesland et al. [23] observed, our GPUs also do not establish error bounds compatible with the IEEE standard, though they have the same floating-point representation. In particular, division has larger error than the other operations, because it is implemented by a combination of reciprocal and multiplication. Unfortunately, this division error is critical for LU decomposition, because it increases and propagates the entire error at each decomposition step.

Furthermore, though recent GPUs deal with single-precision floating-point numbers, they do not support double-precision numbers. Thus, errors caused by this limited precision are not essentially addressed yet.

5 Conclusions

We have presented the design and implementation of LU decomposition on the programmable GPU. To study the performance behavior of modern GPUs, we have developed and evaluated some implementation approaches in terms of (a) loop processing, (b) branch processing, and (c) vector processing.

The experimental results give four important points: (1) for dependent loops, the switching strategy using a render texture avoids copies in the VRAM, reducing execution time by 50%; (2) there is a tradeoff relation between the CPU branch and the GPU branch, and the CPU branch provides higher performance for the decomposition of matrices larger than 512×512; (3) the efficiency of floating-point operations is at most 30%, and as Fatahalian et al. state for matrix multiplication, GPUs also require higher cache bandwidth in order to provide full performance also for LU decomposition; and (4) GPUs usually provide different decomposition results from those obtained using a CPU, mainly due to the floating-point division error that increases the numerical error with the progress of decomposition.

Thus, as same as for matrix multiplication, we find that current GPUs are not so suited well for LU decomposition. However, as Moreland et al. [11] pointed out, GPUs are rapidly increasing their performance beyond the Moore's law [24]. Therefore, we believe that this architecture will emerge as an attractive HPC platform, at least for applications where the error is not a critical problem.

References

1. Fernando, R., ed.: GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. Addison-Wesley, Reading, MA (2004)
2. Pharr, M., Fernando, R., eds.: GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley, Reading, MA (2005)
3. Fatahalian, K., Sugerman, J., Hanrahan, P.: Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In: Proc. SIGGRAPH/EUROGRAPHICS Workshop Graphics Hardware (GH'04). (2004) 133–137
4. Thompson, C.J., Hahn, S., Oskin, M.: Using modern graphics architectures for general-purpose computing: A framework and analysis. In: Proc. 35th IEEE/ACM Int'l Symp. Microarchitecture (MICRO'02). (2002) 306–317
5. Larsen, E.S., McAllister, D.: Fast matrix multiplies using graphics hardware. In: Proc. High Performance Networking and Computing Conf. (SC2001). (2001)
6. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. Parallel Computing **27** (2001) 3–35
7. Hall, J.D., Carr, N.A., Hart, J.C.: Cache and bandwidth aware matrix multiplication on the GPU. Technical Report UIUCDCS-R-2003-2328, University of Illinois (2003)
8. Krüger, J., Westermann, R.: Linear algebra operators for GPU implementation of numerical algorithms. ACM Trans. Graphics **22** (2003) 908–916
9. Bolz, J., Farmer, I., Grinspun, E., Schröder, P.: Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. ACM Trans. Graphics **22** (2003) 917–924
10. Moravánszky, A.: Dense Matrix Algebra on the GPU. (2003) `http://www.shaderx2.com/shaderx.PDF`.
11. Moreland, K., Angel, E.: The FFT on a GPU. In: Proc. SIGGRAPH/EUROGRAPHICS Workshop Graphics Hardware (GH'03). (2003) 112–119
12. Fernando, R., Harris, M., Wloka, M., Zeller, C.: Programming graphics hardware. In: EUROGRAPHICS 2004 Tutorial Note. (2004)
13. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing. second edn. Addison-Wesley, Reading, MA (2003)
14. Shreiner, D., Woo, M., Neider, J., Davis, T., eds.: OpenGL Programming Guide. fourth edn. Addison-Wesley, Reading, MA (2003)
15. Microsoft Corporation: DirectX (2005) `http://www.microsoft.com/directx/`.
16. Stevenson, D.: A proposed standard for binary floating-point arithmetic. IEEE Computer **14** (1981) 51–62
17. Montrym, J., Moreton, H.: The GeForce 6800. IEEE Micro **25** (2005) 41–51
18. Dongarra, J.J., Duff, I.S., Sorensen, D.C., Vorst, H.V.D., eds.: Solving Linear Systems on Vector and Shared Memory Computers. SIAM, Philadelphia, PA (1991)
19. Mark, W.R., Glanville, R.S., Akeley, K., Kilgard, M.J.: Cg: A system for programming graphics hardware in a C-like language. ACM Trans. Graphics **22** (2003) 896–897
20. Naruse, A., Sumimoto, S., Kumon, K.: Optimization and evaluation of linpack benchmark for Xeon processor. IPSJ Trans. Advanced Computing Systems **45** (2004) 62–70 (In Japanese).
21. Goto, K., van de Geijn, R.: On reducing TLB misses in matrix multiplication. Technical Report CS-TR-02-55, The University of Texas at Austin (2002)
22. Dongarra, J.J., Luszczek, P., Petitet, A.: The LINPACK benchmark: past, present and future. Concurrency and Computation: Practice and Experience **15** (2003) 803–820
23. Hillesland, K.E., Lastra, A.: GPU floating point paranoia. In: Proc. 1st ACM Workshop General-Purpose Computing on Graphics Processors (GP^2'04). (2004) C–8 `http://www.cs.unc.edu/~ibr/projects/paranoia/`.
24. Moore, G.E.: Cramming more components onto integrated circuits. Electronics **38** (1965) 114–117